Hardware Implementation of Sequential Feature Extraction and Efficient Search-Store methods in Automatic Speech Recognition

A Project Report

submitted by

AKHIL REDDY PAKALA

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY &
MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
May 2019

THESIS CERTIFICATE

This is to certify that the thesis entitled **Hardware Implementation of Sequential Feature Extraction and Efficient Search-Store methods in Automatic Speech Recognition**, submitted by **Akhil Reddy Pakala** (**EE14B041**), to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelors of Technology** and **Master of Technology**, is a bonafide record of the research work carried out by him under my 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. Nitin Chandrachoodan	**Dr. Nagendra Krishnapura**
Research Guide	Research Co-Guide
Associate Professor	Associate Professor
Dept. of Electrical Engineering	Dept. of Electrical Engineering
IIT-Madras, 600 036	IIT-Madras, 600 036

Place: Chennai

Date: 10th May 2019

ACKNOWLEDGEMENTS

Foremost, I am very grateful to Prof.Nitin Chandrachoodan for guiding me throughout the project. His expertise in the area and discussions with him has motivated me to learn more. It has been a great learning curve for me under his guidance. Also, I would like to express my gratitude to Dr. Nagendra Krishnapura for his motivation and guidance during the project. I would also like to thank Dr. Janakiraman who has taught me Digital IC Design, a course which fueled my interest towards digital VLSI.

I would also like to thank Prithvi, Arjun, Radhika, Dheepika and all other labmates for stimulating discussions in various concepts and their inputs to this project. I would like to thank my parents and my brother for the belief they had and support they have given throughout my life. I am very thankful to my friends who have supported me through thick and thin of my life.

ABSTRACT

KEYWORDS: **SDSoC, Vivado_hls, Frame, Features, Token.**

Automatic speech recognition has numerous applications in areas where human interaction with devices is needed. Major problems in implementation include real time decoding of text, reduce power consumption, implementing with minimum resources. In this thesis we will first discuss about the implementation of extracting features from speech signal and architectures proposed for each module. Implementation of feature extraction sequentially is done using SDSoC tool. Optimisation techniques which reduce the resource usage are discussed along with the results.

Viterbi decoding takes most of time the in speech recognition. Binary search tree and binary heap techniques are used to reduce the timings taken for search and storing best techniques. These functions are implemented in viavdo_hls. In this thesis, we discuss the successful implementation of binary heap and binary tree techniques. Results showing the reduction of significant amount of on chip memory usage with less percentage of increase in word error rate(WER) using binary heap.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ASR	Automatic Speech Recognition
MFCC	Mel Frequency Cepstral Coefficient
MFP	Mel Filter Processing
CMVN	Cepstral Mean And Variance Normalization
LDA	Linear Discriminant Analysis
WER	Word Error Rate
BST	Binary Search Tree

CHAPTER 1

INTRODUCTION

1.1 Introduction to ASR

Automatic speech recognition(ASR) or simply speech recognition is a piece of technology that allows us to convert voice into text. Speech is an easy and faster form to interact with devices. It offers numerous applications in areas where man-machine interaction is needed . Ideally speech recognition should be able to cover like large-vocabulary, continuous, speaker-independent, real-time speech recognition.

In a speech device, a microphone coverts speech signals to electrical signals, which are then sampled and digitized for further processing.These samples go through MFCC which is key for extracting features i.e., the components that represents the speech. These are then compared then with trained data sets through different models. Hidden Markov model which is existent from long time has been widely used in decoding features. State of art technologies used some of neural network methods.

Research started way back in 1970's but the applications have been relatively simple due to the limitations in power. One more factor that is stopping us from using speech recognition in wider arena is it's computational requirements. They are trading off performance with accuracy .

1.2 Motivation

Key to solve the low power issues in speech recognition system is to move it from software to completely hardware or coexistent with hardware. Hardware can process different parts parallely which improves performance. Hardware (circuit based) speech decoders instead of cloud based decoders will be highly efficient in applications that doesn't require internet capabilities. While the current speech recognition chips can run in real time, there are different issues like on chip memory usage and low memory bandwidth associated with it.Our project mainly focuses on implementing real time speech recognition system addressing these issues.

1.3 Organization of Thesis

Outline of the thesis is as follows. In chapter 2, we introduce different parts of speech recognition mainly front-end which is extracting features from speech. Each block and it's proposed architecture is explained. In chapter 3, we discuss the implementation of Mel filter processing , logarithm and DCT blocks in verilog. In chapter 4, we discuss about implementing the same on zedboard using SDSoC tool and various issues faced. In chapter 5 , we introduce decoding the features part and how implementing different data structures is necessary for improving performance. Binary search tree and maxheap implementation will be discussed. Results for different active states and timings for different data structures have been put down.

CHAPTER 2

FEATURE EXTRACTION

2.1 MFCCs

Feature extraction is method of identifying the components of audio that are good for identifying the linguistic content excluding noise, background, noise etc. The phoneme of sound that is produced is equivalent to the shape of the sound that is filtered by vocal tract. The shape of the vocal tract manifests itself in the envelope of the short time power spectrum, and MFCC's are accurate way of representing this envelope. For example if an audio sample is passed into MFCCs, equivalent of this audio in text is represented in features.

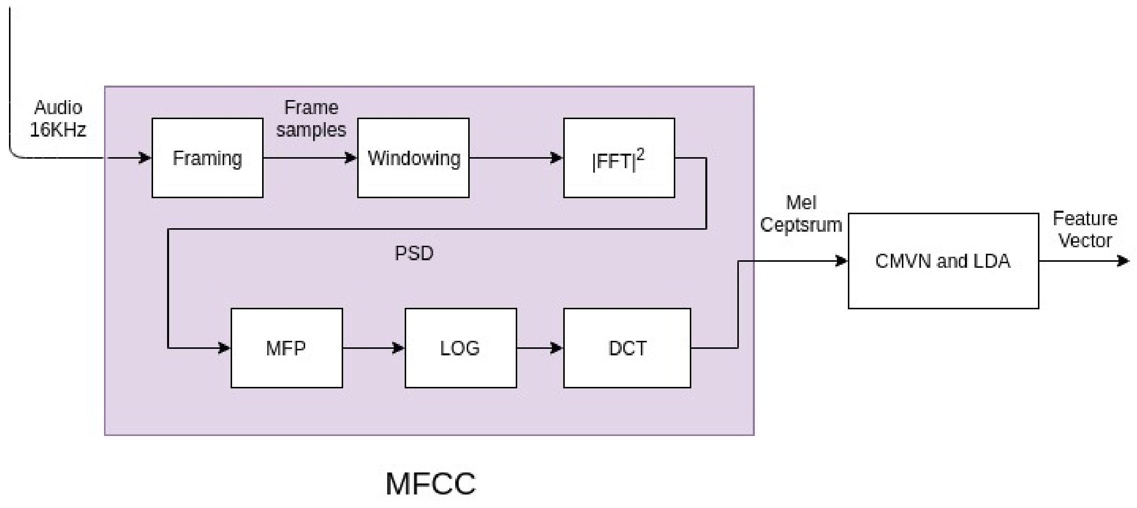


Figure 2.1: Block level architecture of feature extraction

Input audio signal is sampled at 16KHz and first part framing divides the audio signal into 25ms frames. Each frame has 400 samples and frame step is 10ms, i.e., next frame starts at 160th sample. In next step these 400 samples are passed through Hamming Window to reduce spectral effects. 112 zeroes are then padded to these samples to be sent to 512 point FFT module. This FFT module computes the power spectral density for each frame.

Our discussion in this thesis will mainly focus on extracting feature vector from power spectral density.

2.2 Mel-Filter Bank Processing

2.2.1 Calculating mel filterbank

The Mel scale relates perceived frequency, or pitch, of a pure tone to its actual measured frequency. Incorporating this scale makes our features match more closely what humans hear.

$$M(f) = 1125 \ln(1 + f/700) \tag{2.1}$$

and for converting back to frequency

$$M^{-1}(m) = 700(exp(m/1125) - 1) \tag{2.2}$$

where:

f = frequency in Hz

m = melscale

In general no of filter banks would be 20-40. In our case we are sticking to 23 filter banks. Lower and upper frequencies are 20Hz and 8000KHz respectively. These frequencies are converted to melscale by using (2.1) and values are 31.69 Mels for and 2834.91 Mels for 8000KHz. As there are N = 23 filterbanks, we need N+2 points on frequency axis. Divide 23 points uniformly between 31.69 Mels and 2834.91 Mels.

m(i) = 31.69, 148.49, 264.49, ..., 2834.91.

Converting back to frequency

h(i) = 20, 98.76, 185.52, ..., 8000

$$f(i) = floor((nfft + 1) * h(i)/samplerate) \tag{2.3}$$

This results in the following sequence

f(i) = 0, 3, 5,..., 256.

First filterbank starts at 1st point and reaches peak at 2nd point and falls down to zero at 3rd point. Second one starts at second point and reaches peak at 3rd point and falls down to zero at 4th point etc.

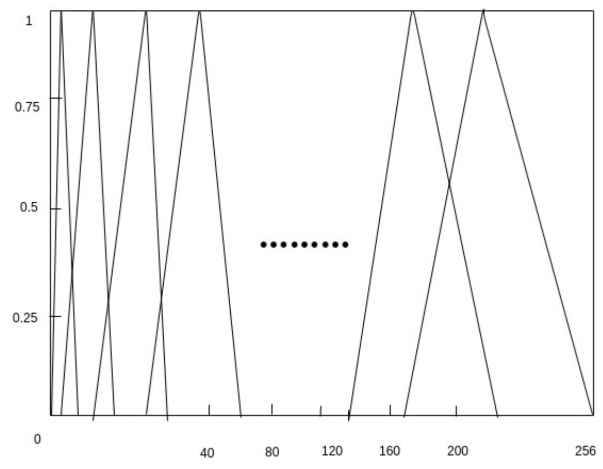


Figure 2.2: Mel-filterbanks

2.2.2 Architecture Explained

Power spectral density (256 bins) are passed through 23 filters to get 23 mel-coefficients. Each mel-coefficient represents the sum of spectral components when passed through filter i.e., PSD of each bin is multiplied by corresponding bin in filter and all these values are added up to get mel-coefficient. Calculating all the mel-coefficients parallelly requires 23 multipliers with following architecture which takes 256 clock cycles(256 multiplications). If these mel-coefficients are calculated sequentially, it would take 5888 (256×23) clock cycles. This architecture would not help us with resources or timing. Considering the fact that most of the multiplications are zeros and property of filters that each filter reaches it's peak at next filter starting point, a new architecture was proposed.

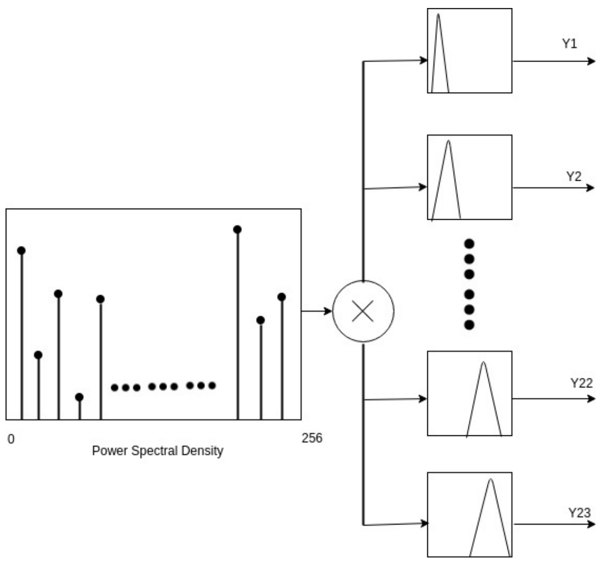


Figure 2.3: Architecture of MFP with 23 multipliers

Mel-filter banks property can be exploited to remove multiplications that gives zero. Each Filter bank can be separated as left filter bank i.e., bin starting from zero and reaching peak, similarly right filter bank starts peak and falls zero. Left filter banks of each filter bank can be integrated and resulting in non-overlapping filter bank called Left Active filter bank and similarly Right Active Filter Bank.

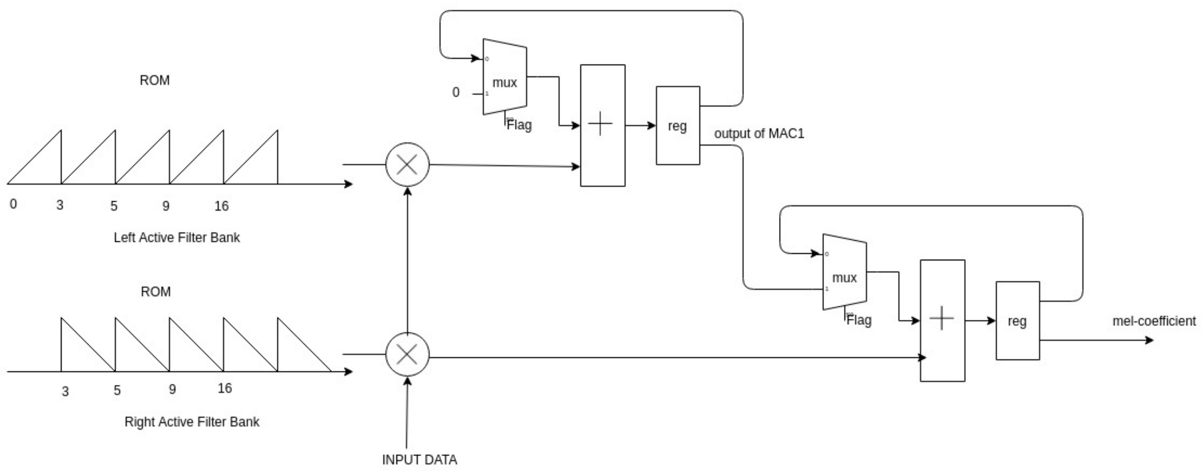


Figure 2.4: New Architecture of MFP with 2 multipliers

Input data contains psd of 256 bins each streaming at 1 input per clock cycle. Values of frequency bins where filter banks goes to zero will be stored in ROM and an additional circuitry using comparator and counter will be required for setting flag to 1 or 0. Psd value of a bin gets multiplied by corresponding bins in both left and right active filter bank. When flag is set to 1, sum of spectral of left filter bank i.e., output of MAC1 is passed to MAC2. Next time whenever flag is set to 1 output from MAC2 will be mel coefficient and simultaneously output of MAC1 is passed to MAC2. Using this architecture, time taken is 256 clock cycles and only 2 multipliers are used. This architecture can be modified to use 1 MAC unit which would take 512 clock cycles.

2.3 Logarithm

Mel coefficients obtained from MFP have large dynamic range.Human ears are less sensitive to slight differences in amplitude at high amplitudes than low amplitudes. So, logarithm is used to compress this dynamic range. There are no default logarithm function in verilog. So, there is a need to design synthesizable logarithm function with high precision and less resources.

$$\begin{aligned}
\ln Z &= \log_2 Z \times \log_2 e \\
Z &= m2^K \\
\log_2 Z &= K + \log_2 m \\
\text{Let} \quad \log_2 m &= x_0 x_1 x_2 \dots x_n \\
\log_2 m &= x_0 + \frac{x_1}{2} + \frac{x_2}{4} + \cdots + \frac{x_n}{2^n} \\
m &= 2^{x_0 + \frac{x_1}{2} + \frac{x_2}{4} + \cdots + \frac{x_n}{2^n}}
\end{aligned}$$

where:

K = integer part

$log_2 m$ = fractional part and $m \in [1,2)$

x_0,x_1,. . . ,x_n are bits in $Q_{1.n}$ format

From the above equations the values of x_i can be calculated using iterative algorithm which is seen in the flow graph (2.5). An architecture for calculating ln is proposed which can be seen in (2.6) and (2.7)

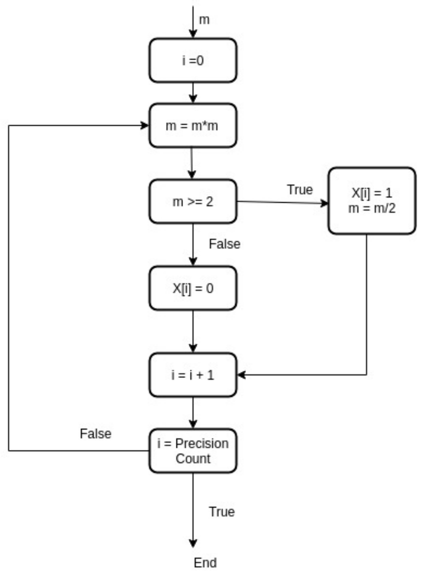


Figure 2.5: Flow graph of $log_2 m$ calculation

2.3.1 Logarithm Architecture

This architecture can be applied to any positive value in $Q_{m.n}$ format.In our case input Z is represented in $Q_{32.16}$ format. Input Z is also an input to Leading one detector (LOD) whose output is an integer where leading one in Z (from MSB) is detected. Subtracting the no of fractional bits from this integer gives us the integer part of log$_2$Z. Using barrel shifter, shift the left shift the value of Z by (31-K) bits. This gives us m which lies in $[1, 2)$ in $Q_{1.15}$ format. Output of this barrel shifter is sent to module 2.7 that calculates logarithm of m i.e, fractional part.

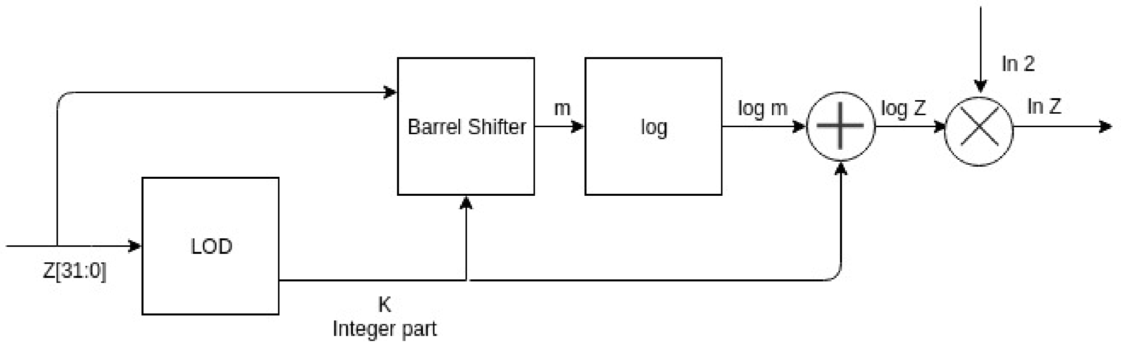


Figure 2.6: Proposed ln architecture

Whenever the count(i) is zero, input is being sent to this module where m gets multiplied by itself and gives Y. If m is greater than 2 i.e., if Y[30] is 1 then x_i will be 1 else 0. Also m is right shifted by one bit if Y[31] is 1 and stored in reg. This is iterated precision(16) no of times to get the fractional part.

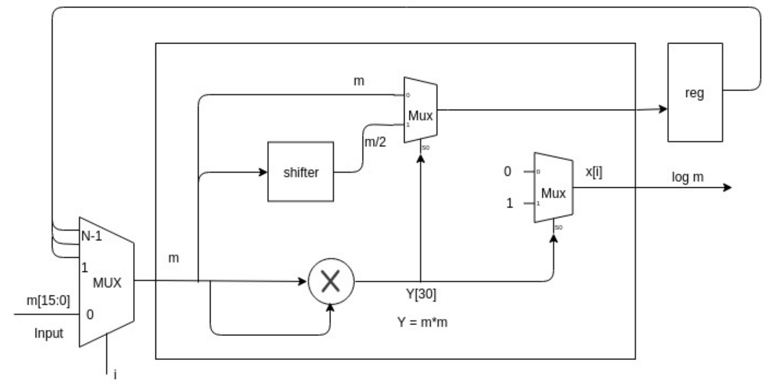


Figure 2.7: Proposed architecture for calculating fractional part of logarithm

2.4 Discrete Cosine Transform (DCT)

The spectrum obtained after applying log of mel spectrum i.e, log mel spectrum should be converted back to time domain. This helps removing the last few coefficients which are not really necessary. These higher-order coefficients that you discard are more noise-like and are not important to train on. DCT is performed by matrix multiplication of log of mel coefficients with DCT coefficients there by giving mel-cepstral coefficients.

DCT matrix coefficients (23×13) are streamed 1 per clock cycle. Mel coefficients are stored in serial shift registers which are connected back. Whenever the count reaches 23 flag is set to 1 and count resets back to zero using counter circuit. As the data gets streaming in count increases 1 per clock cycle, DCT coefficients gets multiplied by Mel coefficients and added up cumulatively in

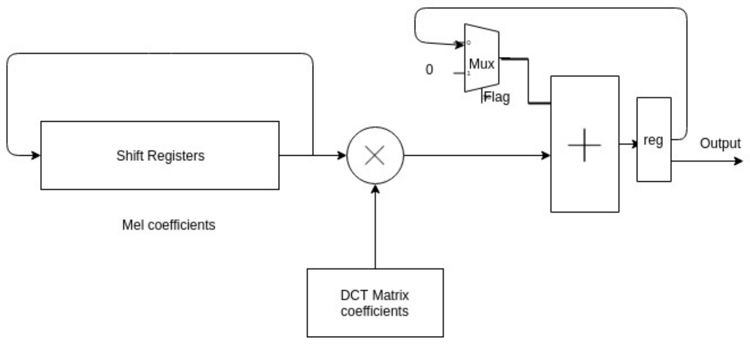


Figure 2.8: DCT architecture with 1 Multiplier

MAC register. When count reaches 23 i.e., flag is set to 1, output will be Mel cepstral coefficient. And input to MAC adder gets reset to 0, also shift registers complete one cycle. No of clock cycles it would take with above architecture (2.8) would be 299 clock cycles and uses only 1 multiplier.

2.5 Cepstral Mean and Variance Normalization(CMVN)

Automatic speech recognition (ASR) involves in many real-world contexts where it encounters adverse acoustic environments. It's performance is sensitive to noise contamination of speech signal. The goal of robust feature extraction is features that are minimally distorted by noise. CMVN is an efficient technique in terms of computation that is used for noise cancellation. Mean and variances can be calculated using formula 2.4 and features are normalized using formula 2.5

$$\text{mean}[i] = ((1 - w) \times \text{mean}[i]) + (w \times \text{feat\_in}[i]) \tag{2.4}$$

$$\text{feat\_out}[i] = \frac{(\text{feat\_in}[i] \text{ - } \text{mean}[i])}{\text{variance}[i]} \tag{2.5}$$

where:

w = 0.002

feat_out = features after normalization

2.6 Splicing

Splicing is storing the data of 9 frames mel cepstral coefficients for each frame. Spliced features of n^{th} frame contains coefficients from $(n-4)^{th}$ frame to $(n+4)^{th}$ frame. Each time there is a new frame, other frames data gets dumped out in similar to FIFO module.

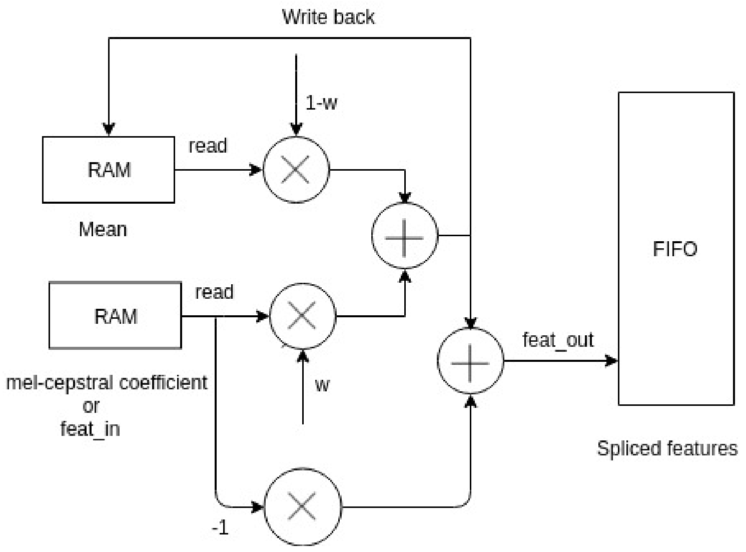


Figure 2.9: proposed CMVN and splicing Architecture

2.7 Linear Discriminant Analysis (LDA)

LDA matrix coefficients (117×40) are streamed 1 per clock cycle. Spliced features (117) are stored in serial shift registers which are connected back. Whenever the count reaches 117 flag is set to 1 and count resets back to zero using counter circuit. As the data gets streaming in count increases 1 per clock cycle, LDA matrix coefficients gets multiplied by Spliced features and added up cumulatively in MAC register. When count reaches 117 i.e., flag is set to 1, output will be features. And input to MAC adder gets reset to 0, also shift registers complete one cycle. No of clock cycles it would take with above architecture (2.10) would be 4680 clock cycles and uses only 1 multiplier.

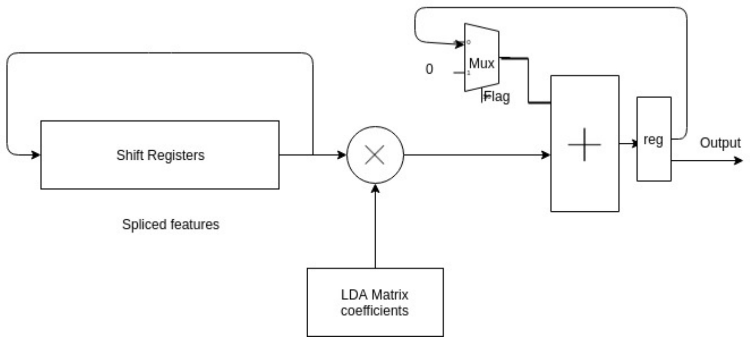


Figure 2.10: LDA architecture with 1 Multiplier

CHAPTER 3

VERILOG IMPLEMENTATION OF FEATURE EXTRACTION

3.1 Communication between modules

Important part of any large design is how well you connect output of one module as input of another module. Handshake based interface will be used in our modules. This will avoid designing around our own assumptions of timing when output of one module is ready. Handshake based interface will let the modules communicate between themselves when inputs/outputs are ready.

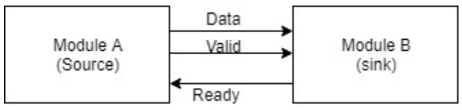


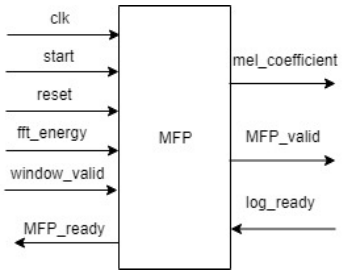
Figure 3.1: Handshake interface between two modules

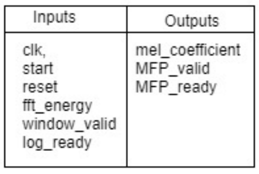
Module A and module B are the two modules which are communicating through handshaking protocol. A is source module which sends it's output through data signal and B accepts its as an input. Valid and ready are handshaking signals which helps in communicating between two modules. Valid signal will be high when input from module A(source) is valid. Ready signal will be high when module B (sink) is ready to accept inputs. When both valid and ready signals are high, data is passed through data wire from source to sink. As we are concerned about synchronous design, valid and ready will change it's state only at positive edge or negative edge of clock.

3.2 Mel-Filter Bank Processing

3.2.1 Implementation

All three modules are implemented in verilog using Xilinx Vivado 2018.2 tool. MFP module has inputs clk, reset, start, fft_energy, window_valid and log_ready. Outputs being mel_coefficient, MFP_valid, MFP_ready. I/O interface to MFP module can be seen in (3.2). Clock, reset and start are global signals to all the three modules. Outputs from windowing module fft_energy and window_valid are being passed as inputs which are simulated using testbench. window_valid will be high when there is valid input from windowing module. Similarly output MFP_valid will be high after output is computed from MFP module.





Inputs	Outputs
clk, start reset fft_energy window_valid log_ready	mel_coefficient MFP_valid MFP_ready

Figure 3.2: I/O interface for MFP Block

A mealy based finite state machine i.e., output depends on present input and state is being implemented which can be seen in (3.3).

IDLE(S0) : Initial state S0 is IDLE where module does no computation and waits for a start signal. If there is start signal we start_en flag is set to high which stays high till next reset occurs. If either of MFP_ready or window_valid is not high, it enters next state S1. If both of them are high state machine enters S2 directly.

WAIT_FOR_VALID_DATA_AND_READY(S1) : In state S1, it waits till both window_valid and MFP_ready is high and enters S2. Else it stays in same state.

COMPUTE_MEL(S2) : MFP module has three combinational blocks, comparator and 2 MAC units.As designed in the architecture (2.4), input data fft_energy is streamed each clock cycle. Frequency values where mel filter banks reaches zero along with left and right mel filter bank values are stored in ROM. In state S2 count is set to zero and incremented each clock cycle whenever window_valid and MFP_ready are high. Each clock input data multiplied with left filter bank is accumulated in MAC1 and input data multiplied with right filter bank is accumulated in MAC2. Comparator sets compare flag to one whenever count is equal to values of frequency where filter bank goes to zero. When flag is raised to one, accumulated value in MAC1 is passed to MAC2 and MAC1 accumulator value is set to zero. Synchronously at same clock edge, output from MAC2 i.e, value of mel coefficient is taken and stored in registers. If all the 23 coefficients are stored, state machine enters S3 state.

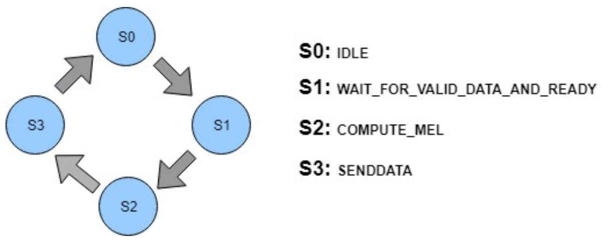


Figure 3.3: Finite state machine implementation of MFP

SENDDATA(S3) : If the ready signal of next module i.e., log_ready is high, mel coefficients are sent to logarithm module. Else it waits for the log_ready signal to be high. After sending all the coefficients it enters state S2 for next data to come in.

3.3 Logarithm

3.3.1 Implementation

LOG module has inputs clk, reset, start, mel_coefficient, MFP_valid. Outputs being log_mel_coefficient, log_valid, log_ready. I/O interface to LOG module can be seen in (3.6). Outputs from MFP module mel_coefficient and MFP_valid are being passed as inputs to LOG module. MFP_valid will be high when there is valid input from MFP module. Similarly output log_valid will be high after log_mel_coefficients are being computed from LOG module.

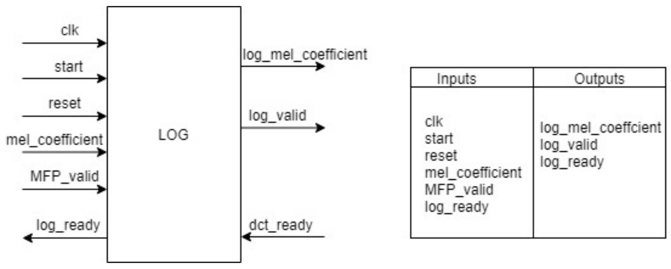


Figure 3.4: I/O interface for logarithm Block

Finite state machine implementation to this LOG module based on architecture(2.6) can be seen in (3.5).

IDLE(S0) : Initial state S0 is IDLE where module does no computation and waits for a start signal. If there is start signal we start_en flag is set to high which stays high till next reset occurs. If either of log_ready or MFP_valid is not high, it enters next state S1. If both of them are high state machine enters S2 directly.

WAIT_FOR_VALID_DATA_AND_READY(S1) : In state S1, it waits till both MFP_valid and log_ready is high and enters S2. Otherwise it stays in same state.

COMPUTE(S2) : In this state log_ready is set to zero initially because LOG module cannot accept next input while logarithm of one input is computed. Leading one in input is found using Priority encoder block and stored as integer part which takes one clock cycle. Using barrel shifter input is shifted and sent to combinational FRAC module which computes fractional part. Now count(i) variable is set to zero. Referring to (2.5) m value which is multiplied by itself and stored in another register. If the value 2nd bit from MSB is one i.e, m*m is greater than 2, one is being written into ith bit of fractional part. This combinational block can be seen in architecture (2.7). When count equals to PRECISION, fractional value of logarithm is known. Integer part and fractional part are added and being sent to multiplier block. Logarithm to the base 2 of input value is multiplied by constant ($\log_e 2$) to get $\log_e$ of a number.

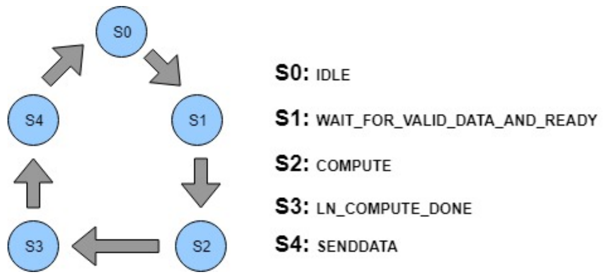


Figure 3.5: Finite state machine implementation of logarithm

LN_COMPUTE_DONE(S3) : In this state S3, now that the $\log_e$ of input is computed log_ready is set to high so that the LOG module accepts the new input. Another count variable for tracking no of inputs whose logarithm is done is incremented. If this variable is less than total number of inputs i.e., total no of mel_coefficinets state machine enters S2 and LOG module accepts new input. After computing logarithm of all coefficients, state machine enters state S4.

SENDDATA(S4) : If the ready signal of next module i.e., dct_ready is high, log_mel_coefficients are sent to DCT module. Otherwise it waits for the dct_ready signal to be high. After sending all the coefficients it enters state S2 for next data.

3.4 DCT

3.4.1 Implementation

DCT module has inputs clk, reset, start, log_mel_coefficient, log_valid. Outputs being dct_coefficient, dct_valid, dct_ready. I/O interface to DCT module can be seen in (3.2). Outputs from logarithm module log_mel_coefficient and log_valid are being passed as inputs to DCT module. log_valid will be high when there is valid input from logarithm module. Similarly output dct_valid will be high after dct coefficients are being computed from DCT module. DCT module has two combinational blocks comparator and a MAC unit.

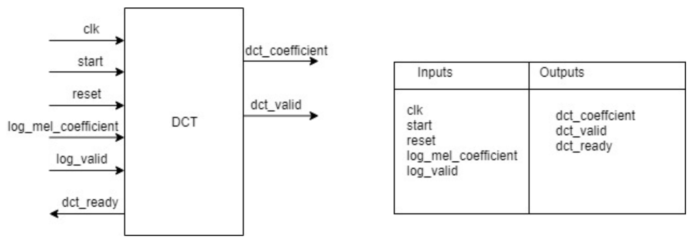


Figure 3.6: I/O interface for DCT Block

Finite state machine implementation to this DCT module based on architecture(2.8) can be seen in (3.7).

IDLE(S0) : Initial state S0 is IDLE where module does no computation and waits for a start signal. If there is start signal we start_en flag is set to high which stays high till next reset occurs. If either of dct_ready or log_valid is not high, it enters next state S1. If both of them are high state machine enters S2 directly raising save _en flag to high.

WAIT_FOR_VALID_DATA_AND_READY(S1) : In state S1, it waits till both log_valid and dct_ready are high and enters S2 raising save _en flag to high or stays in same state otherwise.

SAVE_AND_COMPUTE(S2) : All the log_mel_coefficients need to be saved in registers because they are used repetitively in multiplication. Count is set to zero and increased each clock cycle. In this state when both log_valid and dct_ready are high, dct_matrix_coefficients are being multiplied with log_mel_coefficients and accumulated in MAC unit. Comparator sets the compare flag to high when count is equal to 23. Whenever compare flag is one, output is stored and value in accumulator is set to zero. Once all the log_mel_coefficients are stored, save_en flag is set to zero and state machine enters S3.

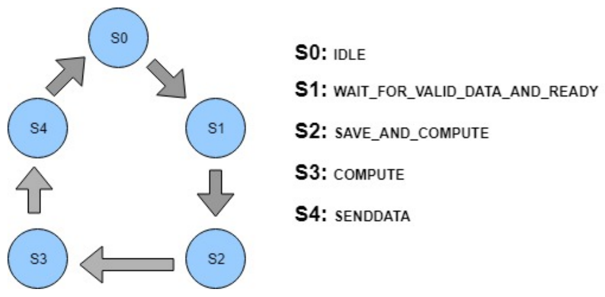


Figure 3.7: Finite state machine implementation of DCT

COMPUTE(S3) : In this state S3, all the saved log_mel_coefficients are passed as inputs to MAC unit one per clock cycle. These values are multiplied with dct_matrix_coefficients and accumulated in MAC unit similar to state S2. When all the 13 dct_coefficients are computed, state machine enters state S4.

SENDDATA(S4) : If the ready signal of next module is high, dct_coefficients are sent to next module. Otherwise it waits for the next module ready signal to be high. After sending all the 13 coefficients it enters state S2.

3.5 Results

Implementation of feature extraction was targeted on Artix 7-FPGA platform. Targeted device chosen was xc7z010clg400-1 which has 28K programmable logic cells, 17,600 Look-Up Tables (LUTs), 35,200 Flip-Flops, 60 Block RAM each capable of storing 36Kb. 60 DSP slices each of which capable of performing 18×25 bit signed multiplication.

3.5.1 Timing results

MFP : Input psd data 256 values are multiplied by left and right mel filter bank values. At the end of 256 clock cycles, all 23 mel coefficinets are computed. Refer to (2.4) architecture.

Module	No of clock cycles
MFP	256
LOG	460
DCT	299
Total time	1015

Table 3.1: Table showing time taken for each module

LOG : After getting mel coefficient as input, it takes 1 cycle for getting leading one using priority encoder, Precision no of cycles (17) for calculating fractional part, 1 clock cycle for multiplying ($\log_e 2$) with logarithm to get ($\log_e$) of input. 1 extra cycle for ready signal to be high and MFP module to recognize and send input. It takes 20 cycles for computing logarithm for each input and overall 460 (20×23) clock cycles.

DCT : Input dct_matrix_coefficients (23×13) are multiplied by log_mel_coefficients each clock cycle. All the mel cepstral coefficients are computed at end of 299^{th} clock cycle.

3.5.2 Resources

DSPs : In MFP block, 2 MAC units each of which does 24×46 bit multiplication. Each DSP unit can perform 25×18 bit multiplication, so MFP block uses 6 DSP units. In LOG block, there are two mutliplications 17×17 bit and 24×18 bit which uses 2 DSP blocks. In DCT module only one 24×18 bit multiplication is done which requires 1 DSP block.In total it would require 9 DSPs.
BRAM: In MFP block left mel and right mel filter bank values needed to be stored. Both of them has 256 values of 24 bits each which requires 2 BRAM (18Kb) equivalent to 1 BRAM(36Kb). In DCT block dct_matrix_coefficients needed to be stored. 299 values of 19 bits each requires 1 BRAM (18Kb) equivalent to 0.5 BRAM (36Kb).

Resource	Used	Available
DSPs	9	80
BRAM	1.5	60
LUT	1303	17200
FF	1047	35200

Table 3.2: Table shows utilization summary of resources

All the registers which are stored in distributed RAM, logic variables, logic operations performed adds up to 1303 LUTs which can be seen table(3.2).

CHAPTER 4

SEQUENTIAL IMPLEMENTATION OF FEATURE EXTRACTION IN SDSoC

4.1 Overview of SDSoC and Zedboard

System on Chip (SoC) is an integrated circuit device which has processor, memory , FPGA and other components on a single chip. SDSoC is a Xilinx Vivado tool allows us to program on SoC. It automates function acceleration in programmable logic generating both ARM and FPGA bitstreams. It allows us to separate a function, estimate the performance and resources.

Processor	ARM A9 cortex	max frequency 667MHz
Main memory	DDR3	512MB
On chip memory	BRAM	1.2MB

Table 4.1: Table shows specifications of zedboard

Device that was targeted for the implementation was zedboard which belongs to Zynq-7000 family of SoCs.Zedboard has ARM A9 cortex processor with Artix 7 FPGA included with it. It has external DDR3 memory of 512MB and on chip BRAM size of 1.2MB. It uses AXI3 bus interface for transferring memory from PS to PL.

4.2 Optimisation Techniques

In speech recognition, feature extraction and GMM are computationally intense functions. These functions require more resources than other functions. Feature extraction takes less than 5% of whole real time decoding. Instead of using resources on feature extraction, sequential implementation will save us resources which can be used by decoding part. Extra latency will be added at once and for all if sequentially implemented. Time budget for feature extraction will be 10ms as decoding should also be done in 10ms. Modules starting MFP until LDA have been implemented in floating point and then changed to fixed point as fixed point would take less time with same number of resources.

4.2.1 Optimisation 1: Read all constants from external memory

This implementation requires constants left_mel, right_mel, dct_matrix_coefficients and lda coefficients to be stored. These constants are actually stored in BRAMs available on FPGA. These coefficients require 7 BRAMs in total.We can save these BRAMs by reading these from external memory. These four coefficients are given as inputs to hardware function. This implementation takes more time, but it is acceptable as we have enough relaxation on timing.

4.2.2 Optimisation 2 : Array Mapping

All the arrays (memory) in not completely occupied most of the times . If two BRAMs are not completely occupied and if both of them can be mapped into single array, a BRAM can be saved. In this case both of the arrays cannot be accessed at same time. This would increase the timings. Splicing function has three arrays which takes 3 BRAMs can be mapped into 1 array. This saves us 2 BRAMs

4.2.3 Optimisation 3: Combine Coefficients in external memory into single array

When a function is moved into hardware in SDSoC, inputs to the hardware function are moved from external memory. Data movers like ACP,AFP are used to move data using AXI3 bus interface from external memory. These data movers write each of the input into a BRAM near hardware function. Hardware function reads these inputs from BRAM. Four coefficients are given as four inputs to hardware function which now takes 4 BRAMs. We are trying to implement sequentially which does not require accessing all coefficients simultaneously. Left and right mel coefficients needed to be accessed simultaneously. So, they are stored alternatively in a single array followed by dct coefficients and lda coefficients. This allows us to reduce the BRAM count from 4 to 1 near hardware site.

4.2.4 Optimisation 4: Limit the multipliers

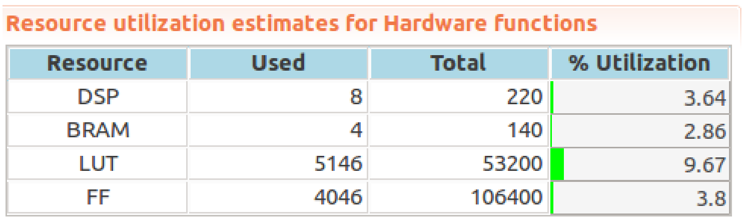
Number of multipliers can be limited to a function using directives. Three functions MFP, LOG, DCT have been into one function and other three functions CMVN, Splicing, LDA into another function. Each function is limited with one multiplier. Limiting the both functions with only one multiplier gives timing error i.e., takes more time than 10ns which is clock period. This is due to lot of routing around the DSP block.

4.3 Results

Feature extraction is implemented in SDSoC and performance estimate is taken. Performance estimate is calculated each time an optimisation is applied. Without any optimisation it would take 12 BRAMs. After all the optimsations are applied it is reduced down to 4 BRAMs. Feature extraction starting from MFP has 2 inputs and an output which takes 3 BRAMs. 1BRAM is used for 3 arrays which are mapped into 1 in splicing function. Each 32×32bit multiplication takes 4 DSPs. As number of multipliers are limited to 2, it would take 8 DSPs.

Resource	Without Opt	Opt1	Opt2	Opt3	Opt4
BRAM	12	9	7	4	4
DSPs	25	25	25	25	8

Table 4.2: Table shows changes in resources after applying each optimisation


Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	8	220	3.64
BRAM	4	140	2.86
LUT	5146	53200	9.67
FF	4046	106400	3.8

Figure 4.1: Performance estimate of feature extraction

Framing , windowing and FFT when combined with above described hardware functions would take 150k clock cycles (¡ 10ms). It takes 12 DSPs in total when complete feature extraction is done.

CHAPTER 5

VITERBI DECODING

5.1 Viterbi search

5.1.1 HMM-GMM scoring

Markov property states that at time t, if there are past and present states, future states depends only on present states. Hidden Markov model states that states in this case remains hidden and generate other possible outcomes.

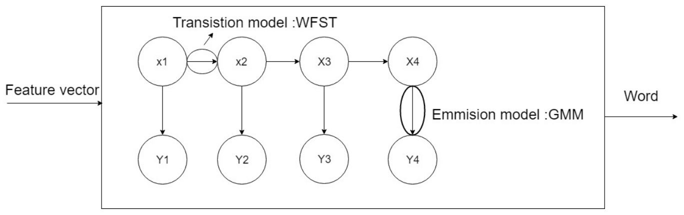


Figure 5.1: HMM model

Every state is associated with input label, output label and weight. The probability of one state going to another state independent of a feature i.e., weight which is determined by WFST model. This transition model is composition of 4 WFSTs (aka H.C.L.G fst) namely

H : Contains HMM information

C : Represents context dependency

L : Lexical FST

G : Grammar FST

Each state has next set of states associated that it might go in next frame. The probability of one going to next state if given a feature vector is calculated using Gaussian Mixture Model (GMM). Let y_t be the acoustic vector and x_t be the state, the The pdf of loglikelihood of this vector to be emitted by a state x_t is calculated using formula(5.1).

$$Log(p(y_t|x_t)) = \sum_{m=1}^{N} g_m + \sum_{d=1}^{D}[\frac{(y_{t,d} \times \mu_{m,d})}{\sigma_{m,k}^2} - \frac{y_{t,d}^2}{2\sigma_{m,k}^2}] \tag{5.1}$$

Viterbi search begins with no hypothesis and develops new set of active hypothesis from feature vectors represented by tokens in our decoder. The forward pass of viterbi search propagates hypothesis from current frame to next frame.

Each hypothesis is fetch from current frame(t) active state list. These tokens are stored in array in SRAM. Next set of tokens are stored in another SRAM and will be swapped at the end of each frame. WFST model is stored in main memory. All the outgoing arcs information for each token is read from WFST model. Fetching the arcs from main memory takes most of the time. For each frame , we traverse through current list of tokens and get all outgoing arcs for each token. And for each arc, cost associated with going from one state to other state for given feature vector is computed using GMM and added to arc weight which gives total cost of arc. Information of next set tokens(t+1) are being stored for current frame. If two tokens from the current set of tokens goes to same next state, the path which gives the best cost is being retained. This state space grows exponentially unless a beam is used. A beam is used and will only retain the tokens whose cost is better than the beam cost. Next set of tokens (t+1) will become current tokens for next frame and serach goes on till the end of audio.

5.1.2 Problems in Implementation

1. Due to limitation of on-chip memory, no of tokens has to be limited to N. This will increase word error rate (WER). Best N tokens in terms of cost has to be stored. Data structure that stores best N tokens need to be used.
2. For every frame array search has to be performed on active states list. This will degrade performance of decoder. Better search algorithm has to be used.

5.2 Binary Search Tree

5.2.1 Hash Maps vs Binary Search Tree

Hash maps is an other way storing and searching the states in tokens list. It is implemented in hash tables where it stores key value pair. State IDs are the value and key function is used for computing key whose input will be state ID. We have 2.3M state IDs from WFST model and we are storing the top N tokens for each frame. Hash maps takes O(1) time for searching the states assuming that key value for each state is different. If two states have same key value it's a collision. These collisions are resolved by linear probing. In short hash tables are implemented as two dimensional arrays($N \times 1$, $\frac{N}{2} \times 2, \ldots$). If an array is filled,it goes to next array whose key is not equal to key of that array. Searching requires more time and inefficient for above cases. Binary search tree on other hand offers a smooth searching and inserting methods which doesn't require any extra time for rearranging. For a random data it is difficult to find a key function for state IDs ranging up to 2.3M and minimize the collisions. If the no of states goes high, binary search tree is preferred over hash maps because increase in one level of binary search tree can accommodate 2^n values. So, binary search is preferred as search and inserting the state list.

5.2.2 Memory Mapping

Binary search trees have nodes which points to address of it's left or right node. The left and right nodes have pointers which points to it's left and right nodes. In hardware, a pointer should always point to scalar of known data type like int or float. This is limitation of hardware. So, different memory mapping has to be done. So, every node will store left and right address as offset from zeroth element in token i.e., index of array as integer.

5.2.3 Search and Insert in Binary tree

In viterbi search, every frame has list of tokens which have list of states and outgoing arcs to next state. Outgoing arcs state information are to be stored in binary tree. Each token will now have a state id, left id, right id and parent id to be inserted in binary tree. All the three values are of data type integer and store the index of corresponding token. First token arrives will be root of the binary tree. Token count is initialized to zero and incremented each time a new token comes. This token count will be the index of the array. Every time a new token comes it's state id searched in binary tree using algorithm (refer algorithm here). The state id is compared with root, if it equal to root index of root will be returned, else if it is less than root we read left id of root and go to left element, else we go to right element. We repeat this process until state id is found or we reach the end of the tree. If there no state id in the list of tokens it is inserted until number of elements reaches maximum count N.

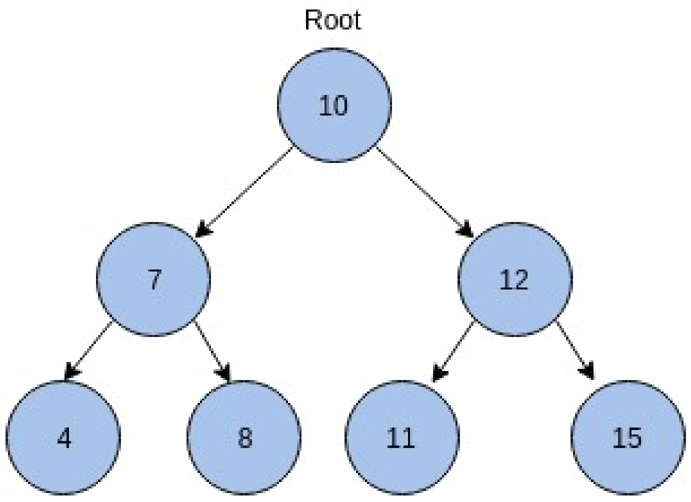


Figure 5.2: Binary search tree

5.2.4 Delete in Binary tree

Best N tokens are to be stored in array for better WER. Whenever number of tokens reaches N and if a new token is to be inserted in binary tree, a token from binary tree should be deleted. Deleting a node in binary tree should also balance the binary tree. Deleting and balancing a binary tree has three cases.

No child : If the node to be deleted has no child, remove the node and left/right id of the parent to null.
Right or Left child : If the node to be deleted has only left/right child. Replace the node with corresponding left/right child. Change the parent id of child to parent id of node to be deleted.

Both children : If the node to be deleted has both children, find the inorder successor of the node to be deleted. Inorder successor is the minimum value of the state id of left tree of the node to be deleted. Replace the node with it's in order successor and change the left, right and parent ids of elements that gets

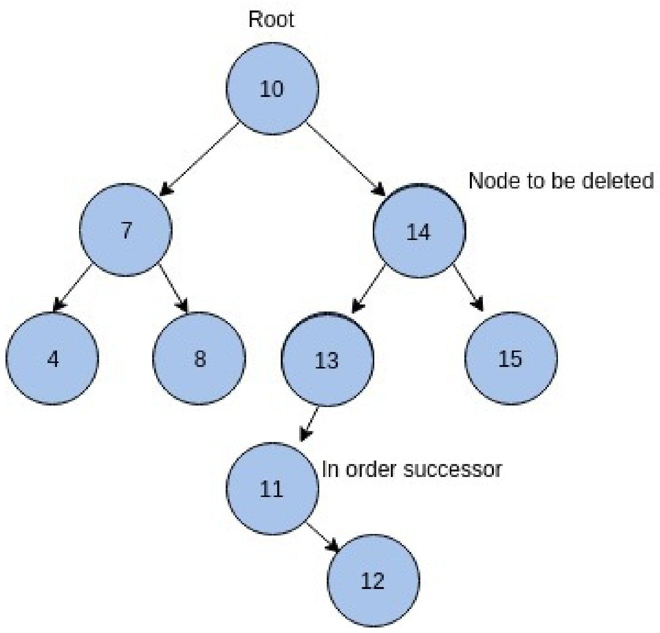


Figure 5.3: Binary tree showing node to be deleted and it's inorder successor

affected by this rearrangement. See figure (5.3).

5.2.5 Odd Even Binary tree

Current binary tree implementation takes worst case time of O(n). This will degrade the performance of decoder. Sequence of state ids are in increasing order to an extent. Separating them to odd and even and storing them in separate binary trees would reduce the time for searching. Each time a new state id comes, it is checked if it is odd or even and correspondingly searched or inserted in their respective trees. Total number of elements combined in both odd and even binary tree is constant. So, no extra memory is used except storing separate odd and even tree root node indexes.

5.3 Max Heap

Heap data structure can be used to store best N tokens in current frame. Max heap property states that node value should be greater than both the children which can be seen in (5.4). This way root node of the heap i.e., cost heap has maximum cost. Each element in heap has state id and cost value.

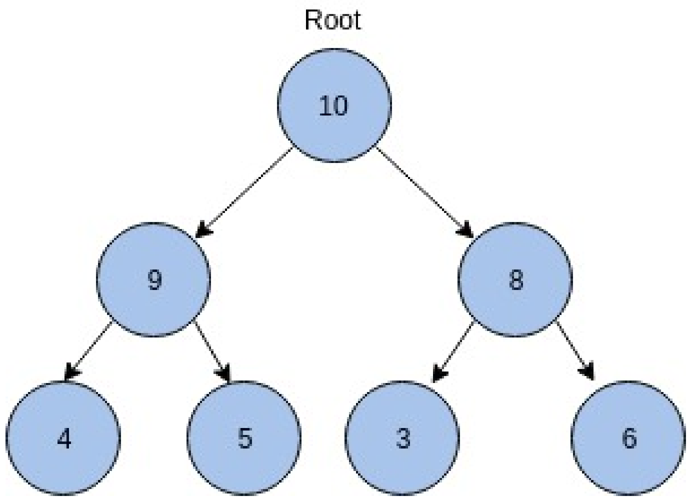


Figure 5.4: Max heap

Insert in heap: Indexing of each element in heap follows a definite rule. Root element will be first element in heap i.e, with index zero. Every element with k index has it's left child index as 2k+1 and right child index as 2k+2. Each time token count will be increased if a new element is inserted. Every new element is inserted in it's token count position first. It's value is compared with it's parent value. If the value of new element is greater than it's parent, cost and state id values are swapped with parent values. This iterates until new element is less than it's parent.

Update in heap : If a new element with same state id and better cost is encountered, it should be replaced in heap. New element has less cost than previous element with cost, therefore heap has to be re balanced. If the element is positioned at k, it's cost has to be compared to maximum cost among children. If cost of element is less than it's child, state ids and cost values have to be swapped with child with maximum cost. This iterates till the heap is balanced. If the heap is full and a new element with best cost than root is encountered, root is replaced with new element in similar fashion.

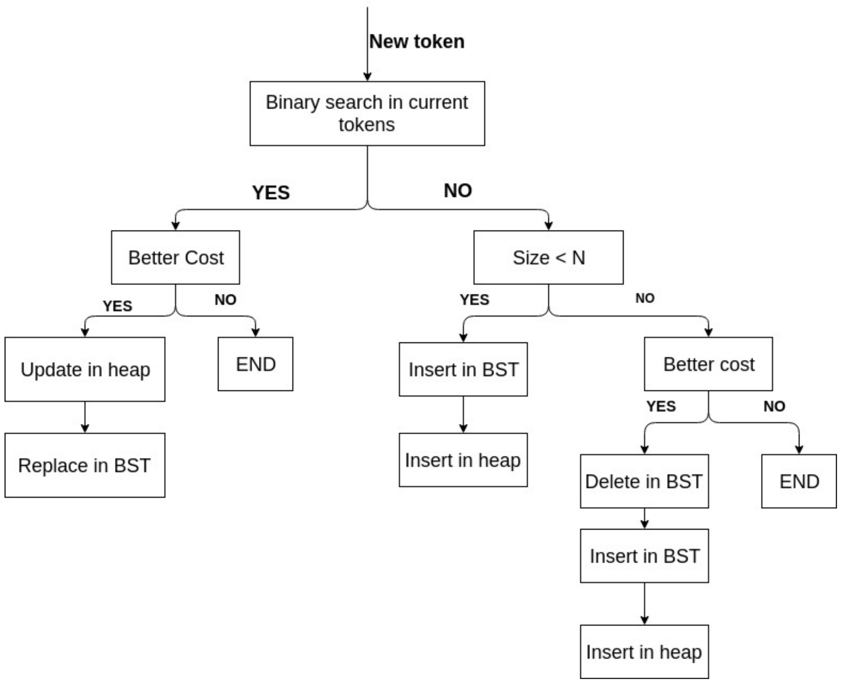


Figure 5.5: Figure shows overall flow of code in Viterbi decoding

5.4 Results

Implementation of functions search and insert in binary search tree along with insert and update heap have done in floating point c. Each of these functions have been optimised and timings for each of these functions have been calculated using vivado_hls.

Array	Binary search tree	Odd even binary search tree
1128 sec	12.55 sec	10.33 sec

Table 5.1: Table shows timings for 45 sec waveform for various approaches for 8k tokens

Array search takes 1128 sec where as naive binary search takes 12.55 sec. Odd even binary search tree optimises even further down to 10.33 sec. Timings for odd even binary search tree is tabulated 5.2 varying number of tokens.For searching an element in binary search tree, tree has to be checked to a certain depth. Average of these timings for all searches for a 45 sec waveform is calculated. Timing decreases from 10.33 sec to 4.38 sec as we go from N = 8k to N = 1k

Number of tokens (N)	Search timings in odd even binary tree		
	Avg depth	No of calls	Timings(in sec)
1k	12	11.55M	4.38
2k	13.33	15.33M	6.28
4k	14.94	18.59M	8.70
8k	16.45	20.21M	10.33

Table 5.2: Table shows search timings for 45 sec waveform

Word error rate (WER) is tested over 12 test wave files comprising of 2700 words. WER is measured using Levenshtein distance method. It is the method that is used for measuring difference between two sequences. This word error rate is calculated after removing trailing words. As the number of tokens decrease from 8k to 1k, word error rate increases from 15.26 % to 15.76 % if the best tokens are to be stored for each frame. Word error rate increases from 17.32% to 78.92% abruptly if first N tokens are stored for each frame. For N = 512 word error rate rises upto 18.46%.

Number of tokens	Word Error Rate (WER)	
(N)	Best N tokens	First N tokens
1k	15.76	78.92
2k	15.66	51.33
4k	15.48	32.06
8k	15.26	17.32

Table 5.3: Table shows word error rate for best and first N tokens

CHAPTER 6

CONCLUSION

Feature extraction part of speech recognition is implemented sequentially in zedboard. This would reduce the power consumption when compared to normal feature extraction. In Viterbi decoding, part of viterbi forward pass is implemented by coordinating odd even binary search tree and binary heap which would allow us to complete speech recognition in real time and with less memory.

After implementing binary heap with binary search tree, token can be decreased from 8k to 1k without much effects. With just 0.5% increase in WER, significant amount($>$ 50%)on chip memory can be saved. Number of calls for each function is reduced as number of tokens decreased. Search timings in array shows a reduction of 57.5% when token count is reduced. Timings of other computationally intense functions like GMM which takes large chunk of time in speech recognition is also reduced. 23% reduction in timings for GMM are observed.

Future works include the implementation of speech using neural networks instead of HMM-GMM scoring.

REFERENCES

1. Michael Price, James Glass and Anantha P. Chandrakasan, *A 6 mW, 5,000-Word Real-Time Speech Recognizer Using WFST Models*. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 50, NO. 1, 2015.

2. Patrick J. Bourke, *A Low-Power Hardware Architecture for Speech Recognition Search*. PHD-THESIS, MAY 2011.

3. A. M. Mansour, A. M. El-Sawy, M. S. Aziz, and A. T. Sayed *A New Hardware Implementation of Base 2 Logarithm for FPGA*.International Journal of Signal Processing Systems Vol. 3, No. 2, December 2015

4. Xilinx. *SDSoC Profiling and Optimization Methodology Guide(UG1235)*. Xilinx(v2018.3) User Guide, 2019.

5. Xilinx. *SDx Pragma Reference Guide(UG1253)*. Xilinx(v2018.2) User Guide, 2018.