# An HMM-Based Speech Recognition IC

Han Wei

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Electronic Engineering

Supervised by
Prof. C. F. Chan

# Abstract

Automatic speech recognition has received a great deal of attention in the past decade and a wide variety of isolated word recognition systems have been used in many applications. Speech recognizers based on hidden Markov models (HMMs) have less computation compared with other speech recognizers. Thus HMM technology is most widely used in speech recognition.

The models of the speech recognition system can be trained as one mixture or multi mixtures. More mixtures in HMMs will result more computation requirements and more complicated design, but the recognition accuracy will be better. In this thesis a double-mixture hidden Markov model based isolated word recognizer IC is presented.

Using a table look-up approach, the new design is smaller and more accurate comparing with existing designs, and added advantage of this design is that the architecture can be extended to higher-order mixture HMM based speech recognizer with minor modifications.

The test chip is fabricated with a 0.35μ CMOS technology. The chip can operate at 20MHz at 3.3V, and at this frequency the recognition time is 0.5 sec for a 50-word speech library. Tested with 353 speech data from AURORA 2 database, the chip's recognition accuracy is 93.8%, which is as accurate as a software recognizer using the same algorithm.

# 摘要

語音識別自動化在最近的十幾年中受到了很大的關注。各種各樣的單字識別系統在很多應用中都有使用。和其他的語音識別系統比較，基於隱馬爾可夫模型（Hidden Markov Model，HMM）的語音識別器需要較少的計算量。因此，隱馬爾可夫模型技術在語音識別中備受廣泛應用。

語音識別系統中的模型可以被訓練成單一混合（single mixture）或 多混合（multi mixtures）模型。在隱馬爾可夫模型中，混合越多，所需要的計算量越大，設計也越複雜，但是能得到更加精確的識別結果。本論文介紹了一塊基於雙混合（double-mixture）隱馬爾可夫模型的單字語音識別集成芯片。

使用了查詢表（look-up table）的方法，這個新的設計和現有的設計相比更小更精確，並且具有額外的優點：通過微小的修改，此結構可以推廣到基於高次混合（higher-order mixture）隱馬爾可夫模型的語音識別器。

測試芯片使用 0.35 微米 CMOS 技術制造。此芯片可以在 3.3 伏電壓，20 兆的時鍾頻率下工作。在此條件下，從一個 50 個單字的字庫中識別一個單字的時間是 0.5 秒。我們使用了 AURORA 2 數據庫中的 353 個語音信號來測試這塊芯片，它的準確率是 93.8%，和使用相同算法的語音識別軟件一樣精確。

# Acknowledgements

I acknowledge gratefully the valuable guidance and encouragement given by my supervisor, Prof. C.F. Chan. He has worked with me and provided me continuous comments, patience, supervision, and encouragement throughout the lengthy and demanding project. I would also like to express my gratitude to Prof. Lee Tan for his insightful suggestions and assistance during my research work. In addition, a special expression of thanks goes to the research assistant, Mr. Hon Kwok Wai, for his important assistance in my research. Without their willing support this work would not have been possible. Also I would like to thank Prof. C.S. Choy and Prof. K.P. Pun for their kind assistance.

Thanks also to my colleagues, Mr. Cheng Wan Chi, Mr. Chan Wing Kin, Mr. Leung Pak Keung, Miss. Yeung Wing Ki and Mr. Yu chun Pong, and the laboratory technician Mr. Yeung Wing Yee and those have been referred to in this thesis.

# Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

## 1.1. Speech Recognition

Automatic speech recognition by machine has been a goal of research for more than four decades. Broadly speaking, there are three approaches to speech recognition: the acoustic-phonetic approach, the pattern recognition approach, and the artificial intelligence approach. The pattern recognition approach is a commonly used method for speech recognition because of three reasons: simplicity of use; robustness and invariance to different speech vocabularies, users, features sets, pattern comparison algorithms, and decision rules; proven performance. A popular pattern recognition method is the HMM (Hidden Markov Model) approach, which uses statistical information inherent in the speech data in recognition, e.g., mean and covariance.

In most pattern recognition systems, there are four main steps: feature extraction, in which a sequence of measurements is made on the input signal to define the test pattern; pattern training, in which one or more test patterns of the same class are used to created a pattern representative of the features of that class; pattern comparison, in which the unknown test pattern is compared with each class reference pattern and a measure of similarity between the test pattern and each reference pattern is computed; decision logic, in which the reference pattern

similarity scores are used to decide which reference pattern best matches the unknown test pattern. A block diagram of a pattern-recognition speech recognizer is shown in Figure 1-1 [1].

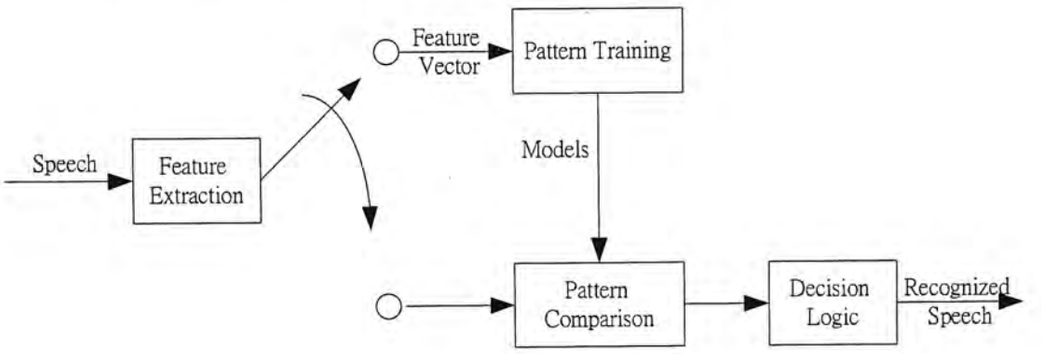



Figure 1-1. A Block Diagram of a Pattern Recognition Speech Recognizer

In a pattern recognition system, the performance is sensitive to the amount of training data available, the speaking environment and transmission characteristics of the medium used to create the speech.

In Figure 1-1, the training part can be viewed as a separate system and normally be implemented by some software using a PC. The rest three parts make up a complete speech recognition system. Such a speech recognizer can be realized by software in a PC or by hardware using a DSP board. But our goal is to develop a new platform which has the smallest area and the lowest price so that it can be utilized in varies applications, e.g., electrical intellectual pet or smart house. A system on chip fulfills these requirements. It can be carried anywhere and embedded inside any products. Moreover, the cost per system is very low compared to DSP and PC system.

We divide the speech recognizer into two parts, the feature extraction and the

pattern comparison & decision logic blocks. In this thesis we design an IC which realizes the comparison and the decision blocks first.

## 1.2. ASIC Design with HDLs

Generally speaking, there are two design flows in ASIC (Application-Specific Integrated Circuit) design. One is from schematic to layout, the other is coding in hardware description language (HDL) and then synthesis and auto-layout. The second design flow is often employed in digital circuit design for that there are always thousands of gates in one design so that it is impossible to draw schematics gate by gate.

By using hardware description languages, designers can easily manage the complexity of large designs containing several million gates, and modify and re-use designs to keep pace with improvements in technology. The most significant gain that results from the use of a HDL design is that a working circuit can be synthesized automatically from a language-based description, bypassing laborious steps that characterize manual design methods (e.g., logic minimization with Karnaugh maps). HDL-based designs are becoming an industry design standard [2].

Verilog and VHDL are two of the most popular hardware description languages, both are Institute of Electrical and Electronic Engineers (IEEE) standards. This

project is designed with Verilog HDL. A typical design flow of Verilog HDL-based ASICs is shown in Figure 1-2 [3].

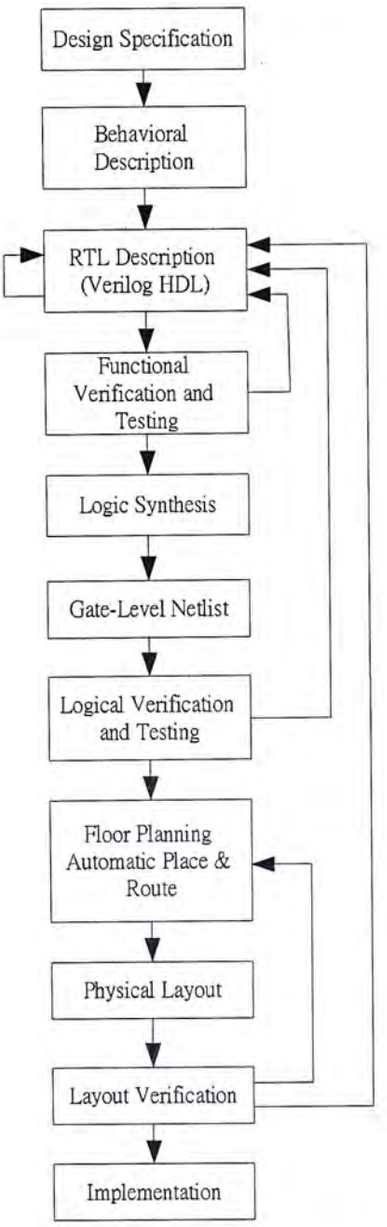



Figure 1-2. Design Flow of Verilog HDL-Based ASICs

The designs specifications are written first. They describe abstractly the functionality, interface, and overall architecture of the digital circuit. Then a behavioral description is created to analyze the design in terms of functionality, performance, and other high-level issues. After the behavioral description is converted to an RTL description, logic synthesis tools convert the RTL

description to a gate-level netlist. The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. After verification, the labyout could be sent for fabrication. Most digital design activity is concentrated on manually optimizing the RTL description of the circuit. After the RTL design is finished, CAD tools are used to assist designers in further processes [3][4].

# Chapter 2 Theory of HMM-Based Speech Recognition

## 2.1. Speaker-Dependent and Speaker-Independent

A speech recognition system can be trained as speaker-dependent (SD) or speaker-independent (SI). The difference is that in a SD system one model is trained with speeches from one person while in a SI system speeches from different speakers can be found in the training data of one model. For a given speech recognition task, a SD system normally performs better than a SI system, as a sufficient amount of data is available to adequately train the speaker-dependent templates, or models. However, when the amount of speaker specific training data is limited, this is not guaranteed because of the lack of reliability in the calculated reference parameters [1].

## 2.2. Frame and Feature Vector

Speech is a time varying signal. However, in a very short period of time (10ms

to 20ms), speech is fairly stationary. So a speech signal can be divided into several segmentations and each of them is called a frame.

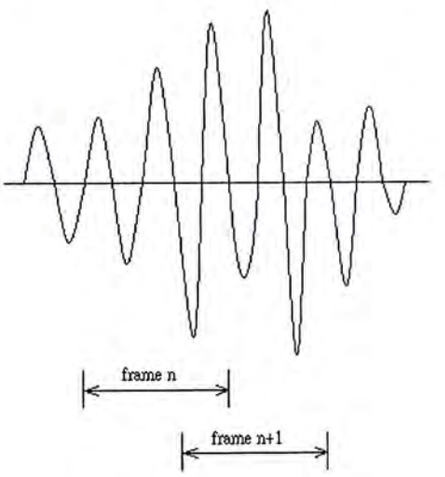



Figure 2-1. Examples of One Frmes of Speech Signal

In a speech-recognition system, the signal-processing front end converts a frame to some type of parametric representation for further analysis and processing. These include the short time energy, zero crossing rates, level crossing rates, and other related parameters. These information are generally at a considerably lower information rate. In the front end processing of this project, one frame is converted to a 26-element vector, in which there are 12 MFCC's (Mel-Frequency Cepstrum Coefficients), 12 first-order derivatives of these coefficients, and the energies of the above two sets of coefficients respectively.

## 2.3. Hidden Markov Model

One well-known and widely used speech recognition algorithm is the hidden

Markov model (HMM) approach [5]. It uses the statistic information inherent in the speech signals and provides a natural and highly reliable way of recognizing speech for a wide range of applications.

## 2.3.1. Markov Model

Consider a system that can be described at any time as being in one of a set of N distinct states, at regularly spaced, discrete times, the system undergoes a change of state according to a set of state-transition probabilities $a_{ij}$ with the following properties.

$$a_{ij} > 0 \qquad \forall i, j$$
$$\sum_{j=1}^{N} a_{ij} = 1 \qquad \forall i$$

This system is called an observable Markov model because at each instant of time the system is in one of the set of states, where each state corresponds to an observable event.

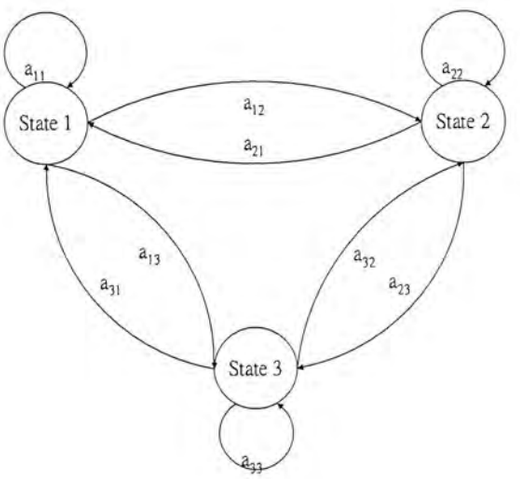



Figure 2-2. A Three-State Markov Model

## 2.3.2. Hidden Markov Model

In an observable Markov model each state corresponds to a deterministically observable event. The output of such system in any given state is not random. But this model is too restrictive to be applied to many real problems. So the observable Markov model is extended to include the case in which the observation is a probabilistic function of the state. The resulting model is called a hidden Markov model.

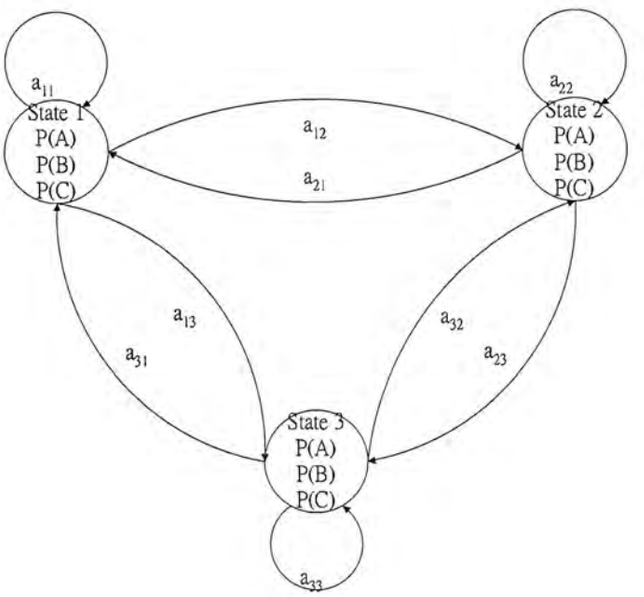



Figure 2-3. A Three-State Hidden Markov Model

Figure 2-3 shows a three-state hidden Markov model. Within each state there are three possible observation symbols, each one corresponding to a possible output of this system. So even the system has been decided in a particular state, the output of the system still has three choices.

A hidden Markov model can be driven from a Markov model. Figure 2-4(a) is a two-state Markov model and in Figure 2-4(b) an equivalent one-state hidden

Markov model is illustrated. Inside the only state of this hidden Markov model, there are two possible observations corresponding to the two states in the Markov model in Figure 2-4(a).

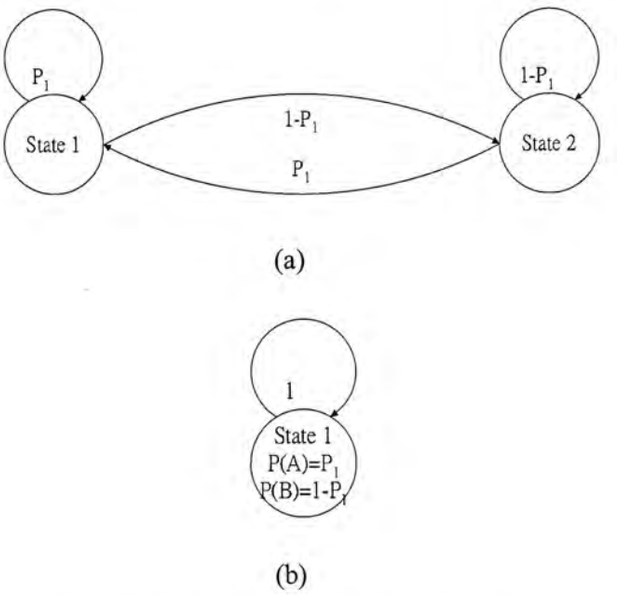



Figure 2-4. (a) A Two-State Markov Model (b) An Equivalent One-State Hidden Markov Model

## 2.3.3. Elements of an HMM

An HMM for discrete symbol observations is characterized by the following elements:

1. $N$, the number of states in the model. In the coin-tossing experiment, each state corresponds to a distinct biased coin.
2. $M$, the number of distinct observation symbols per state. The observation symbols correspond to the physical output of the system being modeled. For the coin-tossing experiment the observation symbols were heads or tails.
3. The state-transition probability distribution $A=\{a_{ij}\}$, where $a_{ij}\geq 0$ for all $i, j$.

4. The observation symbol probability distribution $B=\{b_i(k)\}$, in which $b_i(k)$ defines the symbol distribution in state $i$. In the coin-tossing experiment, it is the probability of Head or Tail in the state Coin1 or Coin2.
5. The initial state distribution $\pi=\{\pi_i\}$, in which $\pi_i$ means the probability of the initial state is state $i$.

A complete specification of an HMM requires specification of two model parameters $N$ and $M$, specification of observation symbols, and specification of three sets of probability $A$, $B$ and $\pi$.

## 2.3.4. Types of HMMs

HMMs can be classified by the structure of the transition matrix $A$ of the Markov chain. One special case is the fully connected HMM or an ergodic model. In this kind of model every state can be reached in a single step from every other state (Figure 2-5), where

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

For some applications, especially in processing speech signals whose properties change over time in a successive manner, a left-right HMM can model the observed properties of the signals better than the standard ergodic model. In such a left-right model, as time increases, the state index increases or stays the

same (Figure 2-6). This model has a fundamental property that the state-transition coefficients have the following property:

$$a_{ij} = 0, \qquad j < i$$

And the state sequence must begin in state 1 and end in state N.

$$\pi_i = \begin{cases} 0, & i \neq 1 \\ 1, & i = 1 \end{cases}$$

One additional constraint is placed on the state-transition coefficients of the models used in this project.

$$a_{ij} = 0, \qquad j > i+1$$

That is, one state can only be reached from its previous one or itself. The state-transition matrix is in the form of the following:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix}$$

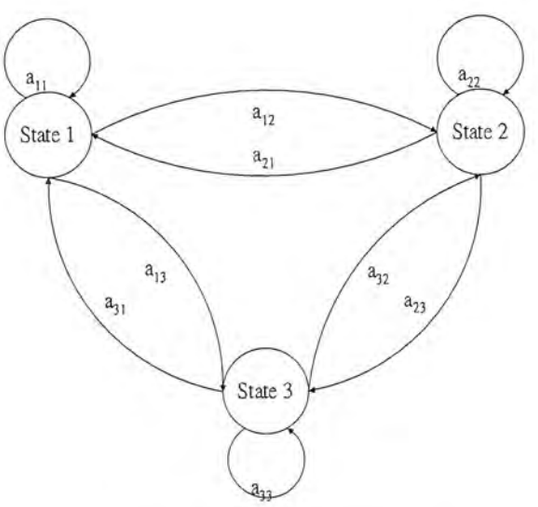



Figure 2-5. A Fully Connected HMM

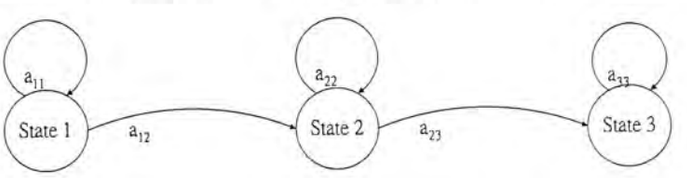



Figure 2-6. A Left-Right HMM

## 2.3.5. Continuous Observation Densities in HMMs

In many real-world applications the observations are often continuous signals. Although it is possible to convert such continuous signal representations into a sequence of discrete symbols (that is, use an observable Markov model to model these continuous signals), it would be advantageous to use HMMs with continuous observation densities to model continuous signal representations directly.

To use a continuous observation density, the model probability density function (pdf) is in the following finite mixture form:

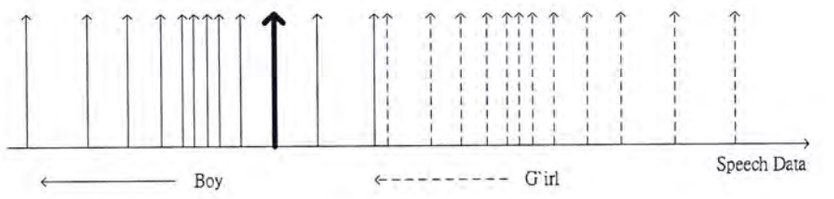
$$b_j(o) = \sum_{k=1}^{M} c_{jk} \mathrm{N}(o, \mu_{jk}, U_{jk}), \qquad 1 \le j \le N$$

Where $o$ is the observation vector being modeled, $c_{jk}$ is the mixture coefficient for the $k^{\text{th}}$ mixture in state $j$ and **N** is any log-concave or elliptically symmetric density (e.g., Gaussian, which is used in this project) with mean vector $\mu_{jk}$ and covariance matrix $U_{jk}$ for the $k^{\text{th}}$ mixture component in state $j$. The mixture coefficient $c_{jk}$ satisfies the following constraint:

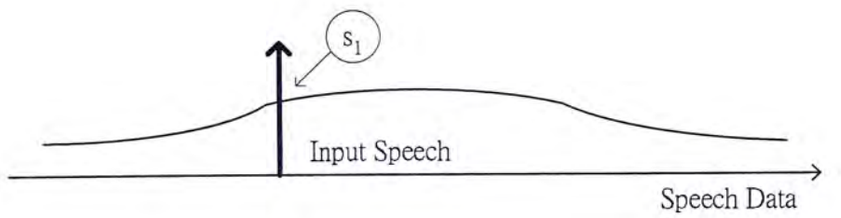
$$\sum_{k=1}^{M} c_{jk} = 1, \qquad 1 \le j \le N$$
$$c_{jk} \ge 0, \qquad 1 \le j \le N, \qquad 1 \le k \le M$$

Assume we have speeches from both boys and girls (Figure 2-7(a)) for a same word, each vector in the figure is one speech data from a boy or girl. Using these data we trained a single-mixture model and a double-mixture model for this particular word separately and then use these two models to recognize an

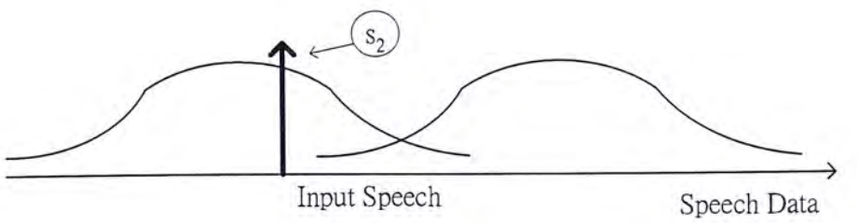
input speech, which is one of the training data from the boys. Here s1 and s2 are scores to be compared in the decision logic part, and a larger score means the input data is more probable to be the word which the model stands for. Obviously with a two-mixture model the speech recognizer is able to give out a better recognition accuracy than only using a single mixture model in recognition (Figure 2-7 (b)(c)). In this project a double-mixture HMM is used to model the speech signals.




(a)




(b)




(c)

Figure 2-7. (a) Training Speeches From Boys and Girls (b)(c) Recognition Result

## 2.3.6. Three Basic Problems for HMMs

Given an HMM in the form of $\lambda=(A, B, \pi)$, there are three problems faced when this model is used in real-world applications:

1. How to efficiently compute $P(O|\lambda)$, the probability of the observation sequence $O=(o_1o_2...o_T)$, here $o_t$ is the input feature vector at time $t$ in speech recognition systems;
2. How to choose a corresponding state sequence $q=(q_1q_2...q_T)$ to best explain the observation sequence $O=(o_1o_2...o_T)$;
3. How to adjust the model parameters $\lambda=(A, B, \pi)$ to maximize $P(O|\lambda)$.

To design an isolated-word speech recognizer, first we have to design a separate N-state HMM for each word of a $V$ word vocabulary. V is the number of words in the vocabulary. This task is done by using the solution to the third problem. Then by using the solution to the second problem we can segment each of the word training sequences into states to make refinements of the model to improve its capability of modeling the spoken word sequences. Finally recognition of an unknown word is performed using the solution to the first problem to score each word model based on the given test observation sequence, and select the word whose model score is highest.

## 2.4. Probability Evaluation

To do speech recognition, we wish to calculate the probability of the observation sequence, $O=(o_1 o_2 o_3 ... o_T)$, given the model $\lambda$, and then we can compare the probabilities obtained from this calculation to make the recognition decision. The most straightforward way of doing this is enumerating every possible state sequence of length T, computing the probability of the observation sequence $O$ given a fixed-state sequence $q=(q_1 q_2 q_3 ... q_T)$ and then summing these probabilities over all possible state sequence $q$. That is,

$$P(O \mid \lambda) = \sum_{q_1, q_2, q_3, ..., q_T} \pi_{q_1} b_{q_1}(o_1) a_{q_1 q_2} b_{q_2}(o_2) ... a_{q_{T-1} q_T} b_{q_T}(o_T) \qquad (2.1)$$

The direct calculation of the above equation involves $2T \cdot N^T$ calculations (N is the number of states in the model), since there are $NT$ possible state sequences (at every $t=1, 2, 3, ..., T$, there are $N$ possible states that can be reached) and for each such state sequence about $2T$ calculations are required for each term in the sum of equation (2.1). It is almost computationally infeasible. Even for a small value of $N$, e.g., $N$=3, for a speech input composed of 100 frames ($T$=100), there are around $10^{50}$ computations.

An alternative to equation (2.1) is that the probability can be approximated by only considering the most likely state sequence, that is

$$P(O \mid \lambda) = \max_q (\pi_{q_1} b_{q_1}(o_1) a_{q_1 q_2} b_{q_2}(o_2) ... a_{q_{T-1} q_T} b_{q_T}(o_T)) \qquad (2.2)$$

The above equation also requires numerous computations when being directly calculated. Fortunately there is a simple recursive procedure existing which allows the equation to be calculated very efficiently. It is called the Viterbi

Algorithm [6].

## 2.4.1. The Viterbi Algorithm

The Viterbi algorithm is used to find the single best state sequence. It is useful in both model training and speech recognition. To find the single best state sequence $q = (q_1 q_2 q_3 ... q_T)$ for the given observation sequence $O = (o_1 o_2 o_3 ... o_T)$, we need to define the highest probability along a single path at time t

$$\delta_t(i) = \max_q P(q_1 q_2 q_3 ... q_{t-1}, q_t = i, o_1 o_2 o_3 ... o_t \mid \lambda)$$

$\delta t(i)$ accounts for the first $t$ observations and ends in state $i$. The complete procedure of the Viterbi algorithm can be stated as follows:

1. Initialization

$$\delta_1(i) = \pi_i b_i(o_1), \quad 1 \le i \le N$$

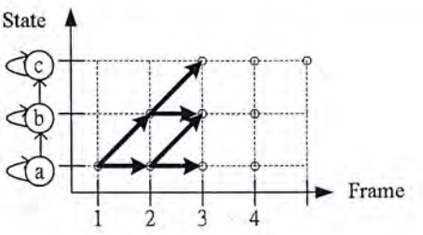
2. Recursion

$$\delta_t(j) = \max_{1 \le i \le N} (\delta_{t-1}(i) a_{ij}) b_j(o_t), \quad \begin{matrix} 2 \le t \le T \\ 1 \le j \le N \end{matrix}$$

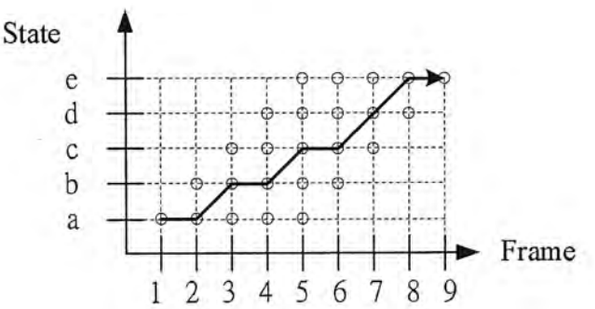
3. Termination

$$P_{final} = \max_{1 \le i \le N} (\delta_T(i))$$

The recursion step is the heart of the Viterbi algorithm. The above procedures should be clear that a lattice (or trellis) structure efficiently implements the computation of the Viterbi algorithm.




(a)




(b)

Figure 2-8. (a) (b) Searching in the Lattice Structure

In Figure 2-8(a), the initialization sets state *a* as the starting point. That is, the first frame corresponds to state *a*. The recursion step determines the maximum probability of a transition path. As illustrated in Figure 2-8(a), each state has two possible paths, one is from its proceeding state and the other is from itself. The algorithm calculates the probabilities of these two paths and only keeps the path with the higher probability. In this example, point *b3* has two possible paths from point *a2* and point *b2* respectively. Assume the path $a_2 \rightarrow b_3$ has a higher probability than the path $a_3 \rightarrow b_3$. Then after calculating and comparing two probabilities of these two paths the algorithm will replace the temporal probability by the larger one. The search will be continued until it reaches the very last state. The termination step determines the final probability of the search as illustrated in Figure 2-8(b).

## 2.4.2. Alternative Viterbi Implementation

The Viterbi algorithm in the preceding section needs multiplications, which is not suitable for hardware implementation. Thus by taking logarithms of the model parameters, the algorithm can be implemented without any multiplication. The main procedures of the modified Viterbi algorithm then become:

1. Preprocessing

$$\tilde{\pi}_i = \ln(\pi_i), \quad 1 \le i \le N$$
$$\tilde{b}_i(o_t) = \ln(b_i(o_t)), \quad 1 \le i \le N, \quad 1 \le t \le T$$
$$\tilde{a}_{ij} = \ln(a_{ij}), \quad 1 \le i \le N, \quad 1 \le j \le N$$

2. Initialization

$$\tilde{\delta}_1(i) = \ln(\delta_1(i)) = \tilde{\pi}_i + \tilde{b}_i(o_1), \quad 1 \le i \le N$$

3. Recursion

$$\tilde{\delta}_t(j) = \ln(\delta_t(j)) = \max_{1 \le i \le N}(\tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}) + \tilde{b}_j(o_t), \quad \begin{matrix} 2 \le t \le T \\ 1 \le j \le N \end{matrix}$$

4. Termination

$$\tilde{P}_{final} = \max_{1 \le i \le N}(\tilde{\delta}_T(i))$$

The calculation required for this alternative implementation is on the order of $N^2T$ additions. Because the preprocessing can be performed once and saved, its cost is negligible for most systems.

# Chapter 3 HMM-based Isolated Word Recognizer Design Methodology

To build a speaker-independent isolated word recognize, assume we have a vocabulary of $V$ words and each word is modeled by a distinct HMM. To do isolated word recognition, we must perform the following:

1. For each word in the vocabulary, we need to build an HMM $\lambda_v$. For each word in the vocabulary there is a training set of $K$ utterances. With these utterances which appropriately represent the characteristics of the word, we can estimate the model parameters ($A$, $B$, $\pi$) that optimize the likelihood of the training set observation vectors for the $v^{th}$ word.
2. For each unknown word to be recognized, the processing is shown in Figure 3-1. First, by MFCC feature analysis the speech signal is extracted into observation sequence $O$; then calculate the model likelihoods for all possible models; finally by selecting the word whose model likelihood is highest the system gives the result.

This project is focused on the probability computation and decision made block. The probability computation step is generally performed using the Viterbi algorithm.

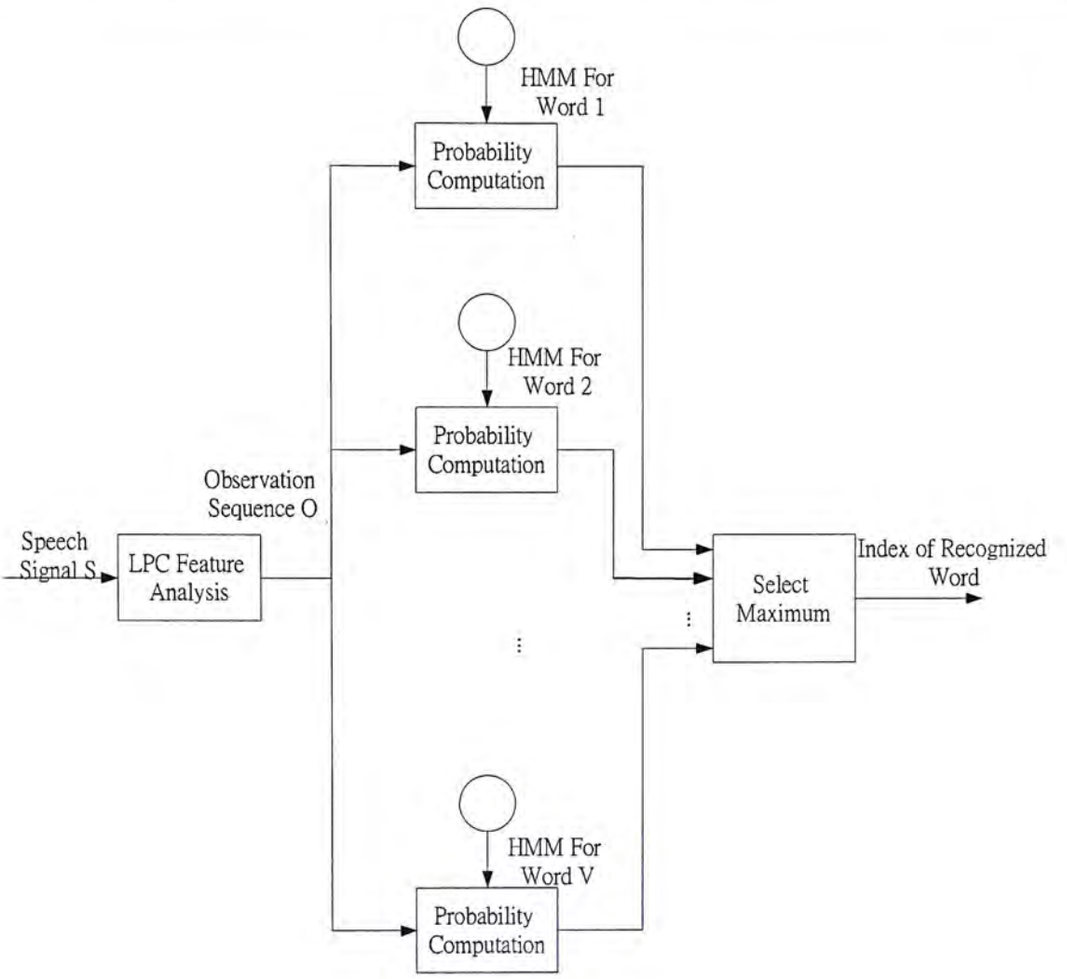



Figure 3-1. HMM Based Recognition Process

An 8-state left-right HMM is trained in advance for each word in the vocabulary. The state-transition matrix for this model is

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{44} & a_{45} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a_{66} & a_{67} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{77} & a_{78} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

One state in this model can only be reached from its previous state or itself. The state sequence ends at state 8 and the initial state distribution $\pi$={1, 0, 0, 0, 0, 0, 0, 0}. This type of HMM can properly model speech signals whose properties

change over time in a continuous manner.

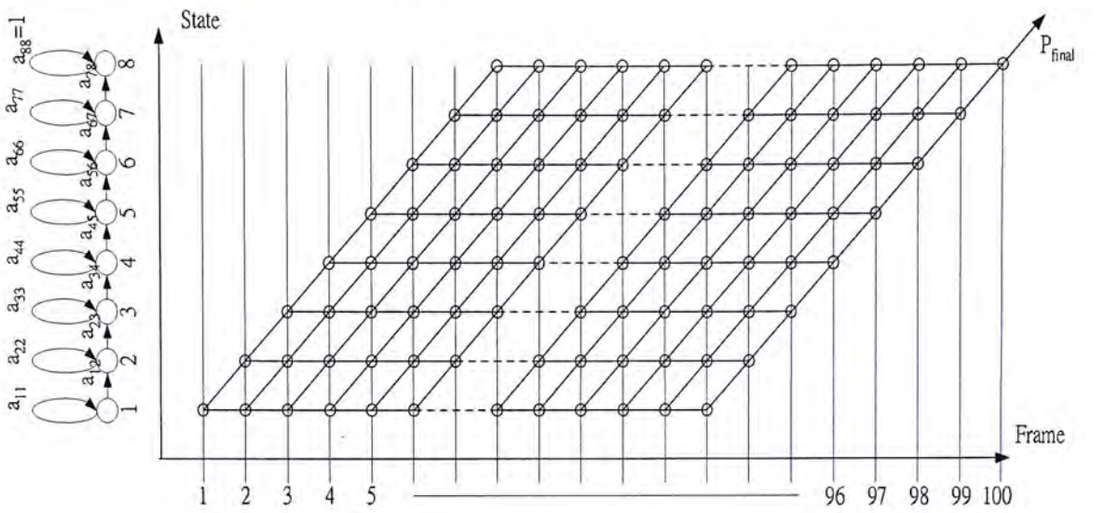



Figure 3-2. The Lattice Structure of the Speech Recognizer

Figure 3-2 shows the lattice structure of the searching engine of this speech recognizer. The horizontal coordinate is the frame number of the input speech which is to be recognized. The vertical coordinate is state index of the model of a particular word in the vocabulary. Each point in this lattice structure corresponds to $b_i(o_t)$, that is, the probability of the input feature vector $o_t$ given the state $i$. Assume at time t the feature vector $o_t$ corresponds to state $i$. As time changes from $t$ to $t$+1 and the input feature vectors change from $o_t$ to $o_{t+1}$, the next state would be $i$ or $i$+1. In the figure, the lines from one point to another point represent these probable state transfers. After comparison the searching engine will keep on the path which has a high probability. The probability calculation ends only when the searching engine reaches both the last input feature vector and the last state of the model.

For a speech input whose length is $T$ frames, the probability of the most probable state sequence that ends at state 8 is

$$
\begin{aligned}
&P_{final} = \delta_T(8) \\
&\delta_t(i) = \max\{\delta_{t-1}(i)a_{ii},\ \delta_{t-1}(i-1)a_{(i-1)i}\}b_i(o_t), \quad \begin{matrix} 2 \le t \le T \\ 2 \le i \le 8 \end{matrix} \\
&\delta_1(i) = \begin{cases} b_1(o_1), & i = 1 \\ 0, & i \ne 1 \end{cases}
\end{aligned} \tag{3.1}
$$

It is to be discussed in the following sections.

# 3.1. Speech Recognition Based On Single Mixture

While there is only a discrete probability density used within each state of the word model, the model is trained as a single-mixture HMM. Given a speech observation $O = (o_1 o_2 o_3 ... o_T)$, the probability density function of each point in the lattice structure of this system would be

$$b_i(o_t) = \mathrm{N}(o_t, \mu_i, U_i), \quad 1 \le i \le N$$

where $o_t$ is the input feature vector with a dimensionality of $n$ at time $t$ (in this project $n$=26). $N$ is a multivariate Gaussian with mean vector $\mu_i$ and covariance matrix $U_i$ in state $i$.

$$
\begin{aligned}
\mathrm{N}(o, \mu, U) &= \frac{1}{\sqrt{(2\pi)^n |U|}} e^{-\frac{1}{2}(o-\mu)' U^{-1} (o-\mu)} \\
&= \frac{1}{\sqrt{(2\pi)^{26} |U|}} e^{-\frac{1}{2}(o-\mu)' U^{-1} (o-\mu)}
\end{aligned} \tag{3.2}
$$

Hence equation (3.1) can be written as

$$P_{final} = \delta_T(8)$$

$$\delta_t(i) = \max\{\delta_{t-1}(i)a_{ii},\ \delta_{t-1}(i-1)a_{(i-1)i}\}b_i(o_t)$$
$$= \max\{\delta_{t-1}(i)a_{ii},\ \delta_{t-1}(i-1)a_{(i-1)i}\}\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}e^{-\frac{1}{2}(o_t-\mu_i)'U_i^{-1}(o_t-\mu_i)}, \quad \begin{matrix} 2 \le t \le T \\ 2 \le i \le 8 \end{matrix}$$

$$\delta_1(i) = \begin{cases} b_1(o_1) = \dfrac{1}{\sqrt{(2\pi)^{26}|U_1|}}e^{-\frac{1}{2}(o_1-\mu_1)'U_1^{-1}(o_1-\mu_1)}, & i = 1 \\ 0, & i \neq 1 \end{cases}$$

As stated in Chapter 2.4.2, to reduce hardware complexity and prevent underflow, the probability is often implemented in logarithmic domain. Take logarithms of $\delta_t(i)$, the above equation will become

$$\tilde{\delta}_t(i) = \ln(\delta_t(i))$$
$$= \ln(\max\{\delta_{t-1}(i)a_{ii},\ \delta_{t-1}(i-1)a_{(i-1)i}\}\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}e^{-\frac{1}{2}(o_t-\mu_i)'U_i^{-1}(o_t-\mu_i)})$$
$$= \max\{\tilde{\delta}_{t-1}(i)+\ln(a_{ii}),\ \tilde{\delta}_{t-1}(i-1)+\ln(a_{(i-1)i})\}+\ln(\frac{1}{\sqrt{(2\pi)^{26}|U_i|}})+(-\frac{1}{2}(o_t-\mu_i)'U_i^{-1}(o_t-\mu_i)) \quad (3.3)$$

For a given state $i$ in the HMM, $|U_i|$ is constant. Thus $\ln(a_{ii})$, $\ln(a_{(i-1)i})$ and $\ln(\frac{1}{\sqrt{(2\pi)^{26}|U_i|}})$ are constant, $\tilde{\delta}_{t-1}(i)$ and $\tilde{\delta}_{t-1}(i-1)$ are scores obtained from the previous step. So the core step of calculating $\tilde{\delta}_t(i)$ is to compute the third part of the above polynomial.

At time t the input feature vector is denoted as $o_t = [x_{t1}x_{t2}x_{t3}...x_{t26}]$, and $x_{ij}$ is one of the 26 coefficients as stated before. For the $i^{th}$ state Gaussian, correspondingly there is a mean vector $\mu_i = [\bar{x}_{i1}\bar{x}_{i2}\bar{x}_{i3}...\bar{x}_{i26}]$ and a covariance matrix $U_i = [u_{i1}u_{i2}u_{i3}...u_{i26}]$, both are composed of 26 elements too. Then the third term of equation (3.2) would be

$$-\frac{1}{2}(o_t-\mu_i)'U_i^{-1}(o_t-\mu_i)=-\frac{1}{2}\begin{bmatrix}x_{t1}-\bar{x}_{i1}\\x_{t2}-\bar{x}_{i2}\\\dots\\x_{t26}-\bar{x}_{i26}\end{bmatrix}\cdot\begin{bmatrix}\frac{1}{u_{i1}} & \frac{1}{u_{i2}} & \dots & \frac{1}{u_{i26}}\end{bmatrix}\cdot\begin{bmatrix}x_{t1}-\bar{x}_{i1} & x_{t2}-\bar{x}_{i2} & \dots & x_{t26}-\bar{x}_{i26}\end{bmatrix}$$

$$=\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$$

As $u_{ij}$ is constant for a given state $j$, the factor $(-\frac{1}{2u_{ij}})$ can be calculated in advance and viewed as a constant too. Thus the third term of the polynomial can be simply implemented by some multipliers and adders. So by using a sing-mixture model, the probability of an input speech observation giving a most probable state sequence can be implemented in hardware just by some multipliers, adders and comparators. And there have been standard implementations for all of these blocks in digital circuit design.

## 3.2. Speech Recognition Based On Double Mixtures

As discussed in Chapter 2.3.5, the speech observations are continuous signals and it is advantageous to use HMMs with continuous observation densities to model the continuous characteristics directly, so some restrictions must be placed on the form of the model probability density function (pdf) to ensure that the parameters of the pdf can be re-estimated in a consistent way when training the model. For a double-mixture HMM isolated word recognizer, all the searching algorithms are the same as the single-mixture recognizer's except that a double-mixture Gaussian is used in the formula of the pdf.

For the t[th] frame, the corresponding pdf is

$$b_i(o_t) = c_{i1}\mathrm{N}(o_t,\ \mu_{i1},\ U_{i1}) + c_{i2}\mathrm{N}(o_t,\ \mu_{i2},\ U_{i2}),\quad 1 \le i \le 8$$

Replace **N** by the representation in equation (3.2), the above pdf would be

$$\begin{aligned} b_i(o_t) &= c_{i1}\mathrm{N}(o_t,\ \mu_{i1},\ U_{i1}) + c_{i2}\mathrm{N}(o_t,\ \mu_{i2},\ U_{i2}) \\ &= c_{i1}\frac{1}{\sqrt{(2\pi)^{26}|U_{i1}|}}e^{-\frac{1}{2}(o_t-\mu_{i1})'U_{i1}^{-1}(o_t-\mu_{i1})} + c_{i2}\frac{1}{\sqrt{(2\pi)^{26}|U_{i2}|}}e^{-\frac{1}{2}(o_t-\mu_{i2})'U_{i2}^{-1}(o_t-\mu_{i2})} \end{aligned} \tag{3.4}$$

Similarly taking the logarithms of pdf to reduce the hardware complexity, equation (3.4) becomes

$$\begin{aligned} \tilde{b}_i(o_t) &= \ln(b_i(o_t)) \\ &= \ln(c_{i1}\frac{1}{\sqrt{(2\pi)^{26}|U_{i1}|}}e^{-\frac{1}{2}(o_t-\mu_{i1})'U_{i1}^{-1}(o_t-\mu_{i1})} + c_{i2}\frac{1}{\sqrt{(2\pi)^{26}|U_{i2}|}}e^{-\frac{1}{2}(o_t-\mu_{i2})'U_{i2}^{-1}(o_t-\mu_{i2})}) \\ &\Rightarrow \ln(C_1e^{X_1} + C_2e^{X_2}) \end{aligned} \tag{3.5}$$

Equation (3.5) requires an add-log operation (ln(∑exp)). Normally the equation will be implemented by taking the larger factor out of the logarithm operation as follows:

$$\ln(C_1e^{X_1} + C_2e^{X_2}) = \max\{\ln C_1 + X_1,\ \ln C_2 + X_2\} + \ln(1 + \frac{C_{\min}}{C_{\max}}e^{X_{mix}-X_{\max}})$$

Here $C_{max}$ and $X_{max}$ are factors in $\max\{\ln C_1 + X_1,\ \ln C_2 + X_2\}$, and $C_{min}$ and $X_{min}$ are factors in $\min\{\ln C_1 + X_1,\ \ln C_2 + X_2\}$. There are two possible solutions to implement the function $\ln(1 + \frac{C_{\min}}{C_{\max}}e^{X_{mix}-X_{\max}})$.

1. Ignore the affect of $\ln(1 + \frac{C_{\min}}{C_{\max}}e^{X_{mix}-X_{\max}})$ [7]. The reason for replacing $\ln(C_1e^{X_1} + C_2e^{X_2})$ by $\max\{\ln C_1 + X_1,\ \ln C_2 + X_2\}$ is that

$\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}} \le 1$, thus $\ln(1+\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}}) \le \ln 2 < 0.7$. If the word to be recognized is distinguishable, that is, one of the probabilities $P(O|\lambda)$ is much larger than the others, this method works well. But if for an input observation, there are two words in the vocabulary whose models produce almost the same probabilities during calculation, then the ignored term $\ln(1+\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}})$ will be a determinant in recognition. Thus this implementation method will introduce large errors.

2. Use a polynomial $y = k(X_{\max} - X_{\min}) + m$ to approximate $\ln(1+\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}})$. First, we have to find out the interval in which the log-function can not be approximated to 0. Then evaluate $y - \ln(1+\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}})$ at points within this interval following a step of every LSB to minimize LMS (Least Mean Squares) to find out k and m [8]. This implementation is more accurate than the previous one, but it needs some special circuits to implement. Moreover, while a higher-mixture model is employed in recognition, the add-log operation is more complex so that a higher order polynomial is needed to approximate the add-log equation. This need more calculation in choosing $k_1$, $k_2$, ..., $k_n$ and $m$. Therefore the hardware implementation will be more complex too.

These are the two main existing implementations for add-log operation, each has its own advantages and disadvantages. In this project we introduce a new method to implement this add-log operation — with a table look-up method.

This approach has simplified the current design while introducing an acceptable computation error.

Because $0 \le \frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}} \le 1$ , $1 \le \ln(1+\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}}) \le \ln 2$ , for a double-mixture model, the value of the add-log factor is kept between 0 and 0.7, and is determined by $\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}}$ . So $\frac{C_{\min}}{C_{\max}} e^{X_{mix}-X_{\max}}$ can be set as the index of the look-up table while the content of the table is numbers between 0 and 0.7. The size of the look-up table is determined by the system requirement, which will be discussed in the next chapter. This table look-up approach has substantially reduced the complexity of the design, improved speed and accuracy. Moreover, the table look-up approach can implement higher order multi-mixture Gaussian pdf architecture based on a single-mixture model [9][10].

# Chapter 4 VLSI Implementation of the Speech Recognizer

As discussed in the Chapter 3, a speaker-independent speech recognizer with a two-mixture HMM can be implemented based on a single-mixture HMM speech recognizer, using the table look-up method. This implementation method can be extended to high-order-mixture HMM systems with little and very simple modifications. Thus this chapter will start from the design of a speech recognizer based on a single-mixture HMM, then the implementation of a two-mixture HMM based speech recognizer, and finally discuss how to applied the table look-up method to a high-order-mixture search engine.

## 4.1. The System Requirements

The speech recognizer in this project is designed for isolated words recognition. The vocabulary is composed of up to 64 words, and we assume the number of the feature vectors of each isolated word to be recognized is not more than 256, which are enough for pratical applications. All the feature vectors are composed of 26 elements and the HMM used to model the word in the vocabulary is a

left-right 8-state model.

All the feature vectors and the model parameters are pre-converted into 16-bit fixed point binary data with an accuracy equavilant to two places after the decimal point floating numbers. For example, a floating point number 32.163 will be converted into a 16-bit fix point number equal to 3216 (110010010000). Because no truncation is considered during the computation in this system, after the multiplication the factor $(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$ will be a 48-bit data and then the addition involves in the summation $\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$ is a 48-bit operation. Thus the pre-calculated constants $\ln\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}$ and $\ln a_{ii}$ are needed to be converted into 48-bit data before adding to $\sum_{j=1}^{26}(x_{ij}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$.

But as the model parameters $\mu_{ij}$ are stored together with the constants and they are 16-bit data, every constant has to be separated into three 16-bit data and stored in three corresponding units in the external memory.

## 4.2. Implementation of a Speech Recognizer with a Single-Mixture HMM

The speech recognizer based on a Single-Mixture HMM mainly realizes the

following algorithm:

$$\bar{P}_{final}=\bar{\delta}_T(8)$$

$$\bar{\delta}_t(i)=\max\left\{\bar{\delta}_{t-1}(i)+\ln a_{ii},\ \bar{\delta}_{t-1}(i-1)+\ln a_{(i-1)i}\right\}+\ln\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2\left(\frac{1}{2u_{ij}}\right),\quad \begin{matrix}2\le t\le T\\ 2\le i\le 8\end{matrix}$$

$$\bar{\delta}_1(i)=\begin{cases}\bar{b}_1(o_1)=\ln\dfrac{1}{\sqrt{(2\pi)^{26}|U_1|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{1j})^2\left(\dfrac{1}{2u_{1j}}\right), & i=1\\ 0, & i\neq 1\end{cases}$$

Here $T$ is the total number of the input feature vectors and the Viterbi algorithm is implemented in the logarithm field.

Figure 4-1 shows the structure of a speech recognizer which employs a single-mixture HMM when training the vocabulary models.

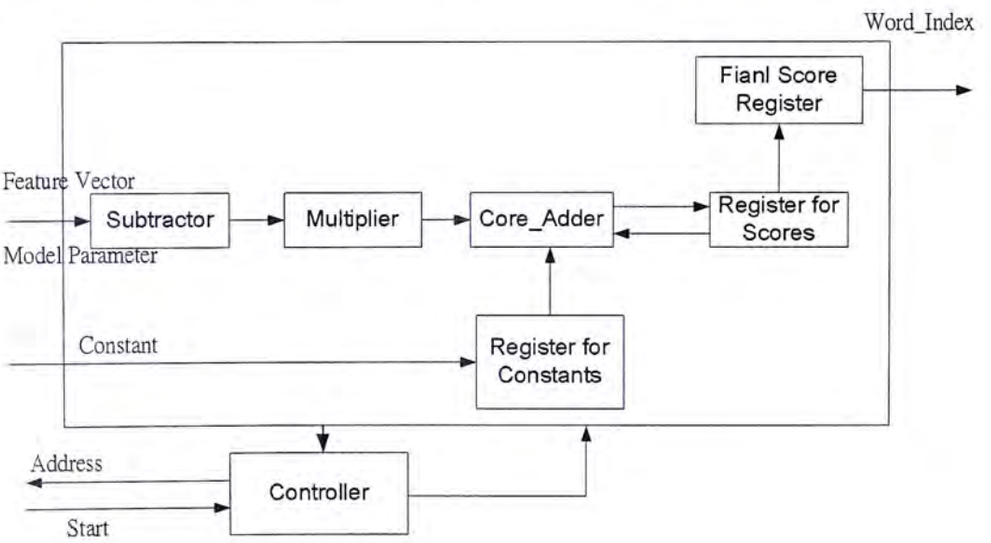



Figure 4-1. The Structure of a Single-Mixture HMM Based Speech Recognizer

This speech recognizer is composed of seven blocks, in which Controller is the master to control other blocks to realize the logarithmic Viterbi algorithm. These blocks are listed as below.

1. Registers. There are three registers in this recognizer.

   1). Register for Constants. It is used to store the constants $\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}$

and $\ln a_{ii}$ or $\ln a_{i(i+1)}$-$\ln a_{ii}$ (The reason to store $\ln a_{i(i+1)}$-$\ln a_{ii}$ instead of $\ln a_{i(i+1)}$ will be explained later). These two constants are 48-bit data but stored as three 16-bit data in the external memory, thus we need to combine these three 16-bit data into one 48-bit data when the system fetches data. Thus this register is composed of two 48-bit storage elements, each of which stores one of the two constants when the system calculates the scores of one of the points in the lattice structure and their contents will be replaced by a new set of constants when come to calculate the next point.

2). Register for Scores. This register is used to store the scores of the points in the lattice structure. Together with Core_Adder, these two blocks complete most of the searching work in the Viterbi algorithm. The scores stored in the register are results from the adder, which are 48-bit data, and they will also be used by Core_Adder as one of the factors of the addition. Because in the searching process, every point has two paths to be reached, one from the previous state and the other from the same state as illustrated in Figure 4-2. The scores of these two paths have to be compared and the larger one is the right score of this point. Then as shown in Figure 4-3, we need eight 48-bit storage elements to store the temporal scores of each point, which are the computation results of the path from the same state, and one more 48-bit element, called "Register Temp", to store the other scores resulting from the path thought the previous state. The score in the "Register Temp" will be compared with the score in one corresponding storage element at a proper time and then the larger one

will be stored in this storage element as the final score up to this point. The comparison and replacement steps are shown in the Figure 4-3 as illustrated by 1, 2, 3.... When the searching process ends, the final score is stored in the $8^{th}$ storage element and will be pass to Final Score Register. This register for scores is composed of nine 48-bit storage elements with some embedded comparison logic.

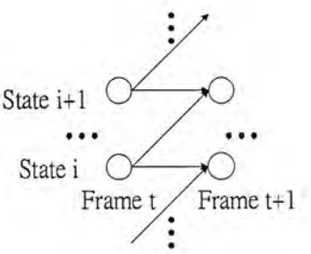



Figure 4-2. Part of the Lattice Structure

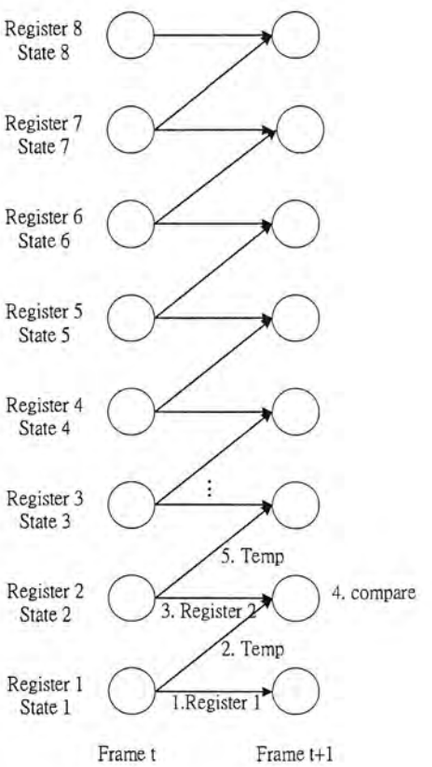



Figure 4-3. Searching Process in the Lattice Structure

3). Final Score Register. This 48-bit register is used to store the largest

final probability of the input observation. Given one word model in the vocabulary, once the search process finishes, the final probability which is stored in the $8^{th}$ storage element of the register for scores will be passed to the final score register. This register then compares the new probability with the one that already stored (previous final score) inside it. If the new one is bigger than the old one, then the new score will replace the old one and the register will also set the recognized word index to the index of the word whose model has a larger probability during the searching process. This word index is the system output. If the new score is smaller than the old one, then the new score will be discarded.

2. Subtractor. This block is used to implement $x_{tj} - \bar{x}_{ij}$. For a given input $x_{tj}$ which is the $j^{th}$ element of the input observation $o_t$ at time $t$, Subtractor finds out the corresponding $\bar{x}_{ij}$, which is the $j^{th}$ component of the Gaussian's mean vector $\mu_i$ of the $i^{th}$ state and calculates the difference between them. Because for a given word model in the vocabulary, $\bar{x}_{ij}$ is constant. Then we pre-store $-\bar{x}_{ij}$ instead of $\bar{x}_{ij}$ in the external memory to avoid the on-chip subtraction. Thus $x_{tj} - \bar{x}_{ij}$ becomes $x_{tj} + (-\bar{x}_{ij})$, which actually is an addition operation. Here we use a 16-bit carry-look-ahead adder [11] to implement this addition, for both the inputs and output are 16-bit data. This adder has very a small propagation delay. The algorithm is illustrated as below.

$P = A \oplus B = A \mid B,$

$Q = A \,\&\, B,$

$C_0 = 0,$

$C_i = G_{i-1} \mid (P_{i-1} \,\&\, C_{i-1}),\ \ i>0$

$S = A \oplus B \oplus C$

Here A and B are two operands of the addition, P and Q are the propagate and generate term, C and S are carry and summation. And all operators involved here are bit operator. The result of Subtractor is the input to the multiplier that follows it. This Subtractor also acts as a passageway. When the system fetches in the $j^{th}$ covariance $u_{ij}$ in the $i^{th}$ state Gaussian's covariance matrix $U_i$, Subtractor does nothing except passing it to Multiplier.

3. Multiplier. It is used to implement $(x_{ij} - \bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$. Although there are a square operation and a multiplication in this factor, we use a 48-bit modified Booth multiplier to realize both square and multiplication to save chip area [12]. The truth table of the modified Booth encoder is shown in table 1. The multiplicand to be encoded is a 16-bit data in both the multiplications. Before encoding, '0' will be added to the right of the multiplicand. By doing so, the number of partial products of the multiplication is reduced to half of the original one and these partial products can be easily calculated by means of bit shifting or negation of the other multiplicand. Also, before summing these partial products up by the full adders, a sign bit must be extended in the partial products, and a constant must be added to the sum of the partial products, starting from the

position n. n is the bit number of the other multiplicand. The extended sign bit is '1' for a positive partial product and '0' for a negative one. And the form of the constant is (1010101...010111), where there are (m/2)-1 zeros and m is the bit number of the multiplicand. In this project m=16. Figure 4-4 gives a simple example of the modified Booth multiplication.

$$X = -2^{m-1}x_{m-1} + \sum_{i=0}^{m-2} x_i 2^i$$

$$X = \sum_{i-0}^{\frac{m}{2}-1} d_i 4^i \quad , \quad X = x_{m-1}, x_{m-2}, \ldots, x_0 \text{ (two's complement)}$$

| $x_{2i+1}$ | $X_{2i}$ | $X_{2i-1}$ | $d_i$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 |
| 1 | 0 | 0 | -2 |
| 1 | 0 | 1 | -1 |
| 1 | 1 | 0 | -1 |
| 1 | 1 | 1 | 0 |

Table 1. The Truth Table of the Modified Booth Encoder

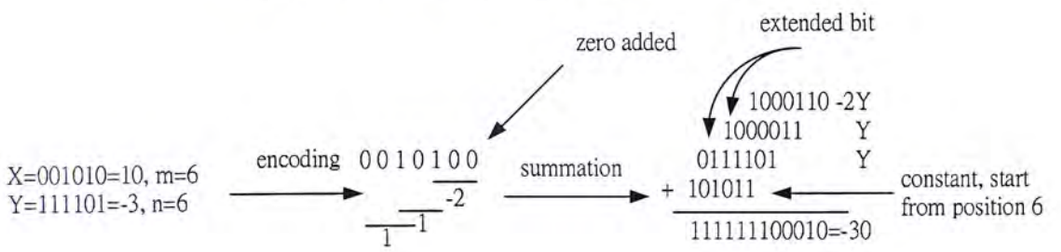



Figure 4-4. A Example of the Modified Booth Multiplication

This modified Booth multiplier first calculate $(x_{tj} - \bar{x}_{1j})^2$, in which $x_{tj} - \bar{x}_{1j}$ is the computation result of Subtractor. Then Multiplier fetches the result of $(x_{tj} - \bar{x}_{1j})^2$ back and multiplies it with $-\frac{1}{2u_{1j}}$, which is also the output of Subtractor and produces the factor $(x_{tj} - \bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$. Here we can see the advantages of using Subtractor as a passageway when

fetching the factor $-\frac{1}{2u_{1j}}$. By using this method the Multiplier's inputs are the output of Subtractor and the computation result of itself instead of the system's input. Then it is much easier for us to trace the data in processing and control the system to fetch input data. The final output of Multiplier $(x_{tj}-\bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$ is one of the Core_Adder inputs.

4. Core_Adder. It is one of the main blocks in the speech recognizer. We use a 48-bit carry-look-ahead adder here for there is no truncation employed in the system. This adder sums up the output of Multiplier, $(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$, and then adds the constant $\ln\frac{1}{\sqrt{(2\pi)^{26}|U_i|}}$ and the previous score of $(t\text{-}1)^{\text{th}}$ frame of this state $\max\{\bar{\delta}_{t-1}(i)+\ln a_{ii},\ \bar{\delta}_{t-1}(i-1)+\ln a_{(i-1)i}\}$ which is stored in "Register for Scores" to the result of the above summation. After that, as illustrated in Figure 4-3, Core_Adder will add the transition coefficient $\ln a_{ii}$ or $\ln a_{i(i+1)}$ to the result of the above operation and these two addition results will be stored in "Register for Scores" according to the judgment method stated above. Here after adding $\ln a_{ii}$ and getting the first addition result, we add $\ln a_{i(i+1)}$-$\ln a_{ii}$ which is pre-calculated to the above addition result instead of $\ln a_{ii}$ to the original value to avoid keeping the original value in Core_adder so that less control signal is needed.

5. Controller. It is the heart of the system. Under the control of this block, the system is able to read feature vectors and model parameters from the

external memories. Inside it, we use several counters to count the system clock cycles and controls the operations of other blocks. Also the frame number of the input observations and the word number of the vocabulary are stored in Controller.

The speech recognizer works as follows:

The feature vectors of the word to be recognized and the models of system vocabulary are pre-processed and pre-stored in two external memories respectively in the form of 16-bit data. Before the system starts to work, the word number of the vocabulary is read into the system and stored in a register. Then the system is waiting for a start signal and once it is detected, the system reads the frame number of the observation into a register and begins calculation. First $x_{11}$ and $-\bar{x}_{11}$ are fetched into the system and Subtractor calculates the difference between $x_{11}$ and $\bar{x}_{11}$, and then pass the result and $-\frac{1}{2u_{11}}$ to Multiplier. Multiplier computers the factor $(x_{11}-\bar{x}_{11})^2(-\frac{1}{2u_{11}})$ and this result together with other $(x_{1j}-\bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$ are summed up in Core_Adder. Here $2 \le j \le 26$, $j$ is the index of the 26 components. After $\sum_{j=1}^{26}(x_{1j}-\bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$ is computed, Core_Adder will add the constant $\ln\frac{1}{\sqrt{(2\pi)^{26}|U_1|}}$ and the previous scores at this state to $\sum_{j=1}^{26}(x_{1j}-\bar{x}_{1j})^2(-\frac{1}{2u_{1j}})$. After that, the logarithmic state-transition constant $\ln a_{11}$ and then $\ln a_{12}-\ln a_{11}$ is added

one by one. The two partial probabilities are to be compared and stored in Register for Scores. The constants $\ln\frac{1}{\sqrt{(2\pi)^{26}|U_1|}}$, $\ln a_{11}$ and $\ln a_{12} - \ln a_{11}$ are read into Register for Constants during some vacant clock cycles. The above calculation and searching along one word model continues until the system reaches the last frame of the input observation and then the final probability is stored in Final Score Register. This process is repeated for each word model in the vocabulary. At the end of every computation the final probability is to be compared with the previous score in Final Score Register and the larger one will be recorded while the corresponding word index of the model in the vocabulary is kept by the system. Thus after all the calculations have been finished, the system will output a complete signal and a word index which indicates the recognition result. To prevent the system continuing meaningless computation, Controller will stop the whole system and set an overflow signal to high immediately when one of Subtractor, Multiplier and Core_Adder is overflow.

## 4.3. Implementation of a Speech Recognizer with a Double-Mixture HMM

A speech recognizer based on a double-mixture HMM is designed to realize the following algorithm, which is similar to the single-mixture system except that the pdf computation has to consider two Gaussians for both the mixtures.

$$\bar{P}_{final}=\bar{\delta}_T(8)$$

$$\bar{\delta}_t(i)=\max\{\bar{\delta}_{t-1}(i)+\ln a_{ii},\ \bar{\delta}_{t-1}(i-1)+\ln a_{(i-1)i}\}+\ln\frac{c_{i\max}}{\sqrt{(2\pi)^{26}|U_{i\max}|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{i\max j})^2(\frac{1}{2u_{i\max j}})+\ln(1+\frac{C_{i\min}}{C_{i\max}}e^{X_{i\min}-X_{i\max}}),$$

$$2\le t\le T$$

$$2\le i\le 8$$

$$\bar{\delta}_1(i)=\begin{cases}\bar{b}_1(o_1)=\ln\frac{c_{1\max}}{\sqrt{(2\pi)^{26}|U_{1\max}|}}+\sum_{j=1}^{26}(x_{1j}-\bar{x}_{1\max j})^2(\frac{1}{2u_{1\max j}})+\ln(1+\frac{C_{1\min}}{C_{1\max}}e^{X_{1\min}-X_{1\max}}), & i=1\\ 0, & i\neq 1\end{cases}$$

Here $C_i$ and $X_i$ stand for $\frac{c_i}{\sqrt{(2\pi)^{26}|U_i|}}$ and $\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$. As discussed in Chapter 3, the pdf in the above equations can be implemented by a look-up table.

First of all, the larger $C_{i\max}e^{X_{i\max}}$ should be selected. Assume that $C_{i\max}e^{X_{i\max}}\ge C_{i\min}e^{X_{i\min}}>0$, take logarithm of this inequality, we will get

$$\ln(C_{i\max}e^{X_{i\max}})\ge\ln(C_{i\min}e^{X_{i\min}})$$

$$\ln C_{i\max}+X_{i\max}\ge\ln C_{i\min}+X_{i\min}$$

$$\ln\frac{c_{i\max}}{\sqrt{(2\pi)^{26}|U_{i\max}|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{i\max j})^2(-\frac{1}{2u_{i\max j}})\ge\ln\frac{c_{i\min}}{\sqrt{(2\pi)^{26}|U_{i\min}|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{i\min j})^2(-\frac{1}{2u_{i\min j}})$$

Thus in a double-mixture HMM speech recognition system, to select the larger $C_{i\max}e^{X_{i\max}}$ when computing the pdf, we only need to compute $\ln\frac{c_i}{\sqrt{(2\pi)^{26}|U_i|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$ for both the mixtures as what we have done in the single-mixture system and then compare these two scores to get the larger one. After that, the factor $\ln(1+\frac{C_{i\min}}{C_{i\max}}e^{X_{i\min}-X_{i\max}})$ is to be realized by a

look-up table. The look-up table's index is decided by $\frac{C_{i\min}}{C_{i\max}} e^{X_{i\min}-X_{i\max}}$, which can be implemented in logarithmic field as follows:

$$\ln \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min}-X_{i\max}} = (\ln C_{i\min} + X_{i\min}) - (\ln C_{i\max} + X_{i\max})$$

Obviously the two operands of the index-decision subtraction are results from the previous calculation and only one more subtraction step is needed to obtain the index of the look-up table.

A block diagram of this double-mixture HMM based speech recognizer is shown in Figure 4-5.

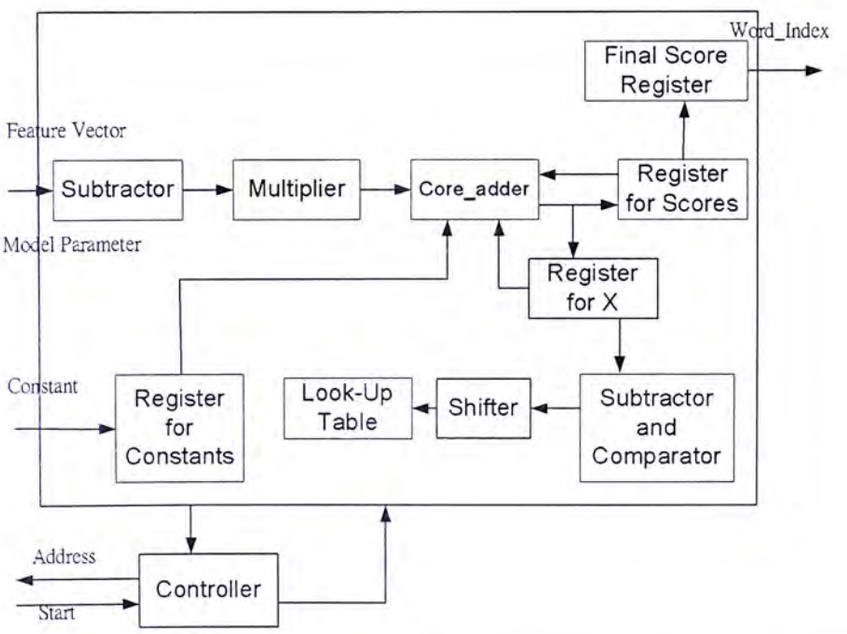



Figure 4-5. A Block Diagram of The Double-Mixture HMM Based Speech Recognizer

There are four more blocks in the speech recognizer based on a double-mixture HMM compared with the single-mixture based one. They are: Register for X, Subtractor and Comparator, Shifter, and Look-Up Table. The functions of these blocks are listed as below.

1. Register for X. This register is used to store

$\ln\frac{c_i}{\sqrt{(2\pi)^{26}|U_i|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$ of the two mixtures, which is the calculation result from Core_adder. Only after Core_adder finishes addition for both of the two mixtures can we perform the comparison to select the larger one and also obtain the index of the look-up table. So for the $t^{th}$ frame given the $i^{th}$ state in the word model, after Core_adder finishes computing $\ln C_1+X_1$ of the first mixture, that is, $\ln\frac{c_{i1}}{\sqrt{(2\pi)^{26}|U_{i1}|}}+\sum_{j=1}^{26}(x_{tj}-\bar{x}_{i1j})^2(-\frac{1}{2u_{i1j}})$, the final value should be stored in Register for X and then Core_adder continues to compute $\ln C_2+X_2$ of the second mixture. Register for X is composed of two 48-bit storage element, each to store one value of $\ln C+X$ for a given time $t$ and state $i$.

2. Subtractor and Comparator. In this block, we compare $\ln C_1+X_1$ and $\ln C_2+X_2$ to select the larger one and also compute the index of the look-up table. As discussed before, the index of the look-up table is implemented in the logarithmic field as $(\ln C_{i\min}+X_{i\min})-(\ln C_{i\max}+X_{i\max})$, after the two operands are calculated by Core_adder and stored in Register for X, Subtractor and Comparator accesses Register for X to get these two numbers and then calculates $(\ln C_{i1}+X_{i1})-(\ln C_{i2}+X_{i2})$. If the result is larger or equal to 0, Subtractor and Comparator will tell Core_adder to add the logarithm factor to $\ln C_{i1}+X_{i1}$, and if not, the logarithm factor will be added to $\ln C_{i2}+X_{i2}$. At the same time the absolute subtraction result will be passed to the next two stages to be converted into the actual index of the look-up table.

3. Shifter. This block is used to shift the output of Subtractor and Comparator to a fixed scale. Because different model employs a different scaling factor in pre-process when being converted into 16-bit data, the result of $|(\ln C_{i1} + X_{i1}) - (\ln C_{i2} + X_{i2})|$ is of different scale too and can not be directly seen as the index of the look-up table which should be a fix number. Thus we apply a fixed scale on the index of the look-up table and use Shifter block to shift the value of $|(\ln C_{i1} + X_{i1}) - (\ln C_{i2} + X_{i2})|$ to this fixed scale before it is used to generate the actual index of the look-up table. The number of shift bit is indicated by a shift number which is decided when pre-processing feature vectors and model parameters.

4. Look-Up Table. The look-up table is stored in an external memory together with the models of the system vocabulary, and the contents of the look-up table are 48-bit data but stored as three 16-bit data. The block Look-Up Table in the block diagram actually is an address-conversion machine, whose input is the shifted absolute value of $(\ln C_{i1} + X_{i1}) - (\ln C_{i2} + X_{i2})$ and the outputs are the actual addresses of the look-up table, which correspond to the locals of three parts of $\ln(1 + \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min} - X_{i\max}})$. Then these three 16-bit data are read into Register for Constants in where they will be combined to a 48-bit data and ready for addition in Core_adder. The contents of the look-up table are decided as follows. As $0 < \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min} - X_{i\max}} \le 1$ , $\ln 1 < \ln(1 + \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min} - X_{i\max}}) \le \ln 2$ , thus

$0 < \ln(1 + \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min} - X_{i\max}}) < 0.7$ . Besides, this system uses a fix-point binary number representation method and two bits after the digital point is considered when converting the decimal feature vectors and model parameters into the binary ones. Apply the same criterion to the content of the look-up table, thus there should be altogether 70 values starting from 0 and ending in 0.69. Because no truncation is employed in this system, the addition between $\ln C_{\max} + X_{\max}$ and $\ln(1 + \frac{C_{i\min}}{C_{i\max}} e^{X_{i\min} - X_{i\max}})$ is a 48-bit operation, the values stored in this look-up table should be scaled into 48-bit binary numbers. Accordingly we then divide the shifted value of $|(\ln C_{i1} + X_{i1}) - (\ln C_{i2} + X_{i2})|$ into 70 ranges and each of the ranges corresponds to one 48-bit binary number in the look-up table.

Other blocks which are same as those in the speech recognizer based on a single-mixture model are of similar functions as stated before.

The working mechanism of the double-mixture HMM based speech recognition with a look-up table is as follows.

The feature vectors of the word to be recognized and the models of system vocabulary together with the contents of the look-up table are pre-processed and pre-stored in two external memories respectively in the form of 16-bit data, similar to what we have done in speech recognition with a single-mixture HMM. Before the system starts to work, the word number of the vocabulary and the shift number are read into the system. The shift number is used by Shifter. After

the initialization, at the time the system detects a start signal, it reads the frame number T of the observation and begins calculation. For the first frame given the first mixture in the first state of the first word model, $\sum_{j=1}^{26}(x_{1j}-\bar{x}_{11j})^2(-\frac{1}{2u_{11j}})+\ln\frac{c_1}{\sqrt{(2\pi)^{26}|U_{11}|}}$ is computed as what have been done in the single-mixture HMM based speech recognizer. After that, the result is stored in Register for X and the system repeats the above procedure to calculate $\sum_{j=1}^{26}(x_{1j}-\bar{x}_{12j})^2(-\frac{1}{2u_{12j}})+\ln\frac{c_2}{\sqrt{(2\pi)^{26}|U_{12}|}}$, given the second mixture of the same state. After these two scores are both stored in Register for X, Subtractor and Comparator compares $\ln C_1+X_1$ and $\ln C_2+X_2$ and gives the difference between them. Then Shifter shifts $|(\ln C_{i1}+X_{i1})-(\ln C_{i2}+X_{i2})|$ with the shift number that is stored in the system. The shifted absolute value is passed to the next block Look-Up Table and converted into three actual addresses of the external memory. The system then fetches the factor $\ln(1+\frac{C_{i\min}}{C_{i\max}}e^{X_{i\min}-X_{i\max}})$ and adds it with the larger $(\ln C_{\max}+X_{\max})$ term which is determined by Subtractor and Comparator. Up to now one score that corresponding to the first frame given the first state in the double-mixture model of the first word in the vocabulary is obtained, and the rest computations and searching procedures are the same as in the single-mixture HMM based speech recognizer.

For comparison, the structure of a hardware recognizer without the look-up table is shown in Figure 4-6. The term $\ln(1+\frac{C_{i\min}}{C_{i\max}}e^{X_{i\min}-X_{i\max}})$ is ignored in this recognizer. To calculate a pdf, first $\ln C_1+X_1$ and $\ln C_2+X_2$ are computed

separately for both mixtures as what have been done in the above procedures. Then we just select the larger term $\ln C_{max}+X_{max}$ as the value of a pdf and continue calculation with it. From Figure 4-6 we will find that the complexity of this recognizer is not reduced much compared with the double-mixture HMM based speech recognizer with a look-up table. The recognition accuracies of these two hardware recognizers will be compared in the next chapter.

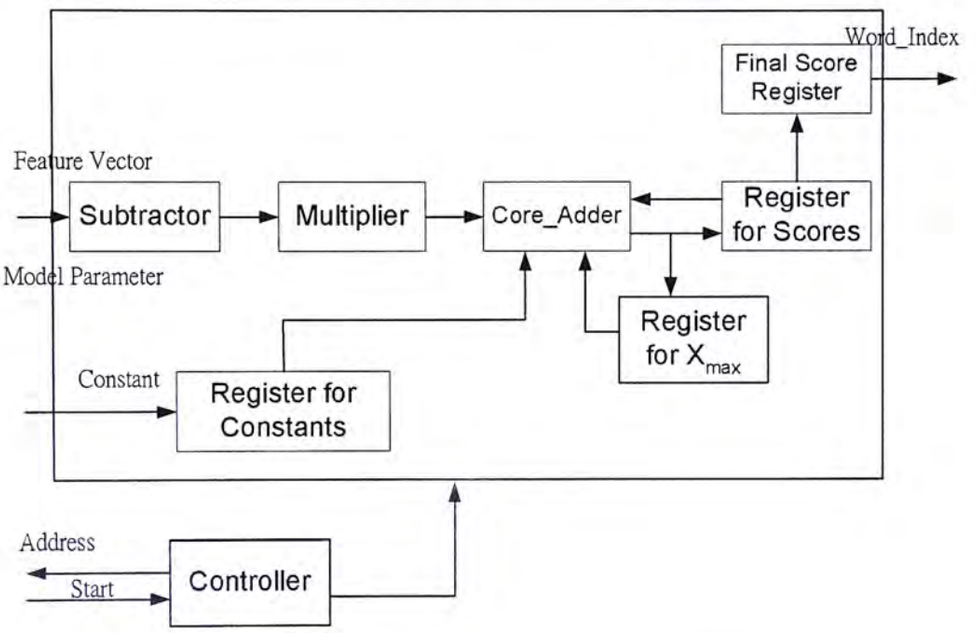



Figure 4-6. The Structure of the Recognizer Without a Look-Up Table

## 4.4. Extend Usage in High Order Mixtures HMM

Even in isolated word recognition, choice of a mixture number that is larger than 1 will provide a better recognition performance. In a high order Gaussian system, the pdf will be in the following form:

separately for both mixtures as what have been done in the above procedures. Then we just select the larger term $\ln C_{max}+X_{max}$ as the value of a pdf and continue calculation with it. From figure 4-7 we will find that the complexity of this recognizer is not reduced much compared with the double-mixture HMM based speech recognizer with a look-up table. The recognition accuracies of these two hardware recognizers will be compared in the next chapter.

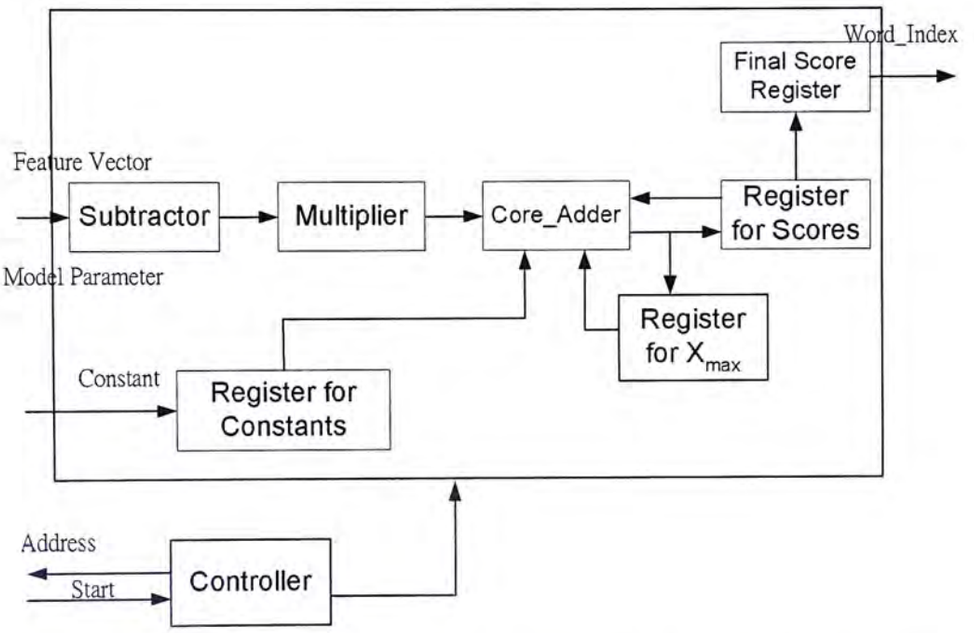



Figure 4-6. The Structure of the Recognizer Without a Look-Up Table

## 4.4. Extend Usage in High Order Mixtures HMM

Even in isolated word recognition, choice of a mixture number that is larger than 1 will provide a better recognition performance. In a high order Gaussian system, the pdf will be in the following form:

$$b_i(o_t) = \sum_{k=1}^{M} c_{ik} \mathrm{N}(o_t, \mu_{ik}, U_{ik}) = \sum_{k=1}^{M} c_{ik} \frac{1}{\sqrt{(2\pi)^n |U_{ik}|}} e^{-\frac{1}{2}(o_t-\mu_{ik})' U_{ik}^{-1} (o_t-\mu_{ik})}$$

Where M is the number of the mixtures. Express the above equation in different form, and take logarithm to simply the calculation and avoid overflow, the equation will be:

$$\overline{b}_i(o_t) = \ln(b_i(o_t)) = \ln(\sum_{k=1}^{M} C_k e^{X_k})$$

Select the largest factor of the addition $C_{\max} e^{X_{\max}}$ out of the summation, the equation will be:

$$\overline{b}_i(o_t) = \ln C_{\max} + X_{\max} + \ln(1 + \sum_{\substack{k=1 \\ k \neq \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}}) \qquad (4.1)$$

As stated in the chapter 3, the above equation can be implemented by three methods:

1. Ignore the effect of $\sum_{\substack{k=1 \\ k \neq \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}}$. This is the simplest implementation way. But as more mixtures is employed when training the models, this method will be with lower recognition accuracy compared with the other two implementation ways.

2. Use a polynomial to approximate $\ln(1 + \sum_{\substack{k=1 \\ k \neq \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}})$. This is an implementation method with relatively high recognition accuracy. But the number of variables in the polynomial is equal to the number of mixture M, and as M becomes larger, it is more and more difficult to find out the coefficients of the polynomial $y = a_1 x_1 + a_2 x_2 + \ldots + a_{M-1} x_{M-1}$. Also it is

not convenient to realize the above polynomial in hardware design.

3. Table look-up approach. This method is simple in realization but with an acceptable recognition accuracy. The logarithmic summation has a limited rang of value because every factor in the summation is less than or equal to 1.

$$\frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}} \le 1$$

$$\sum_{\substack{k=1 \\ k \ne \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}} \le M - 1$$

$$1 + \sum_{\substack{k=1 \\ k \ne \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}} \le M$$

$$0 \le \ln(1 + \sum_{\substack{k=1 \\ k \ne \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}}) \le \ln M$$

Thus the logarithmic equation can be implemented by a look-up table whose contents are values between 0 and ln$M$. The index of this look-up table can be decided by the method that has been used in the design of the speech recognizer based on a double-mixture HMM.

$$\ln(\sum_{\substack{k=1 \\ k \ne \max}}^{M} \frac{C_k e^{X_k}}{C_{\max} e^{X_{\max}}}) = \ln(\sum_{\substack{k=1 \\ k \ne \max}}^{M} e^{(\ln C_k + X_k) - (\ln C_{\max} + X_{\max})}) \qquad (4.2)$$

It is also an add-log operation, but the error introduced here has little effect on equation (4.1). Then to simply the design, the look-up table's index can be decided on the maximum factor in the summation in equation (4.2), $\{\max\{(\ln C_k + X_k) - (\ln C_{\max} + X_{\max}), 1 \le k \le M, k \ne \max\}$. A block diagram of speech recognizer with a high-order mixture HMM based on this table look-up method can be:

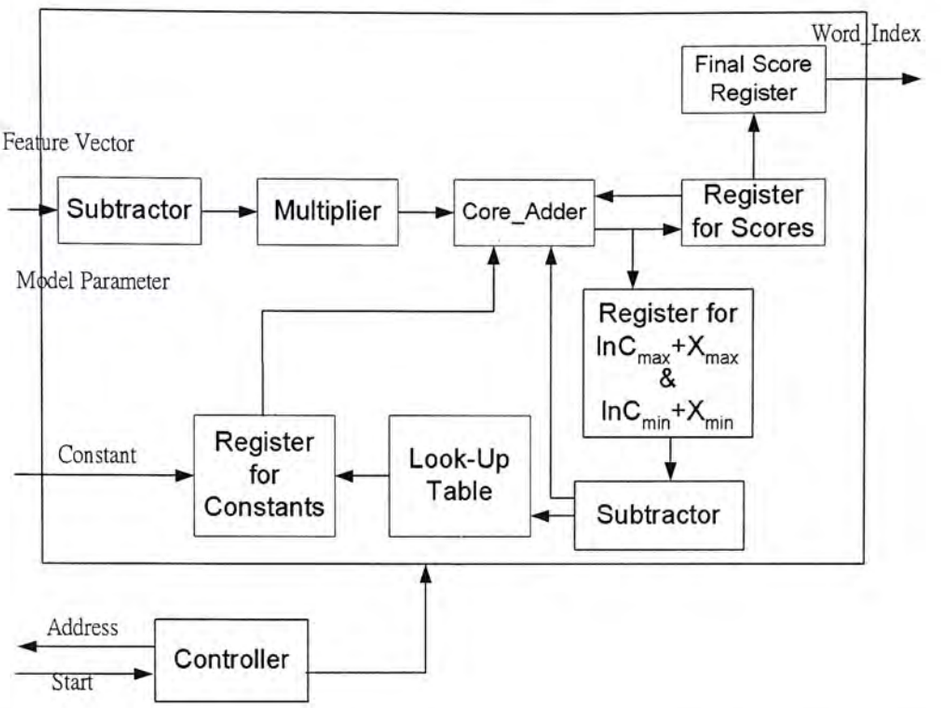



Figure 4-7. A Block Diagram of Speech Recognizer with a High-Order Mixture HMM

This speech recognition works almost as same as the speech recognition which is based on a double-mixture HMM. For one point in the lattice structure, first the recognizer computes the factor $\ln C_1 + X_1$ and $\ln C_2 + X_2$ given the first two mixtures in the HMM and stores them in the register for $\ln C_{\max} + X_{\max}$ and $\ln C_{\min} + X_{\min}$. Then after the third mixture's $\ln C_3 + X_3$ is computed, the contents in the register will be renewed with the maximum and the minimum values. After all mixtures are went through and $\ln C_{\max} + X_{\max}$ and $\ln C_{\min} + X_{\min}$ are obtained, the next block Subtractor will compute their difference and pass it to the block Look-Up Table, as in the double-mixture HMM speech recognition system. Then the largest $\ln C_{\max} + X_{\max}$ will be added with a logarithmic factor from the look-up table and the calculation of the score in this point is finished.

From the above procedure we can see that with the table look-up method, a

single-mixture HMM speech recognizer can be extended to a double-mixture HMM recognizer easily and then a high order mixture HMM speech recognizer can be implemented by minor modifications. The only differences between these high order mixture HMM systems is the size of the look-up table, which is dependent upon the number of the mixtures M.

## 4.5. Pipelining and the System Timing

We have designed a double-mixture HMM based speech recognizer as a synchronous system. In the block diagram, all the blocks except Multiplier consume one clock cycle in working and Multiplier uses two cycles to perform one multiplication. Also one clock cycle is needed to read in the data from the external memories. Therefore pipelining is employed in this system to improve system performance.

Pipelining refers to the partitioning of a process into successive, synchronized stages such that multiple processor, each in a stage different to others, can be executed in parallel. Pipelining techniques are aimed at improving the system throughput. It has the effect of shortening the clock cycle, but the latency of a single instruction or operation will be increased because extra delays are introduced to the basic clock cycle due to the latching of intermediate results [1].

With the pipelining technology, the number of clock cycles that the system uses to compute one partial probability $\delta$ is reduced from more then 10 to only 4, which are consumed in the two successive multiplications $(x_{ij} - \bar{x}_{ij})^2(-\frac{1}{2u_{ij}})$.

All other work is completed within these 4 cycles, including accessing external memories, subtraction, addition and comparison. Thus given that a typical frame number is 128 for an isolated word and the system vocabulary is composed of 50 most frequently-used words, the number of clock cycles that is required to recognize one word is about $10^6$, which can be illustrated from the following equation.

number of cycles × number of mixtures × number of elements in one feature vector × number of states in the HMM × (number of frames - 7) × number of words in the vocabulary = $4 \times 2 \times 26 \times 8 \times (128 - 7) \times 50 \approx 10^6$

In the above equation the number of frames is subtracted by 7, because only one to seven points are needed to calculate their scores at the first and the last 7 frames and therefore it can been seen that the number of frames which will be computed through all 8 states is 7 less than the original number.

If the system is working at an operating frequency of 20MHz, the time required is to recognizer one isolated word under the condition of the above is about 0.5 second. This system speed is acceptable for real-time applications.

We have designed a double-mixture test chip to verify our design. The test chip was designed with Verilog HDL. The design was synthesized with Synopsys and the layout was generated by Cadence place and route tools Silicon Ensemble. A complete list of the Verilog description of the test chip os listed in

Appendix I "Verilog Code of the Double-Mixture HMM Based Speech Recognition IC (RTL Level)". The test chip was fabricated by a 0.35 micron CMOS technology.

# Chapter 5 Simulation and IC Testing

## 5.1. Simulation Result

In this project a speaker-independent speech recognizer for isolated word based on a double-mixture hidden Markov model was designed. The design flow is shown in Figure 5-1.

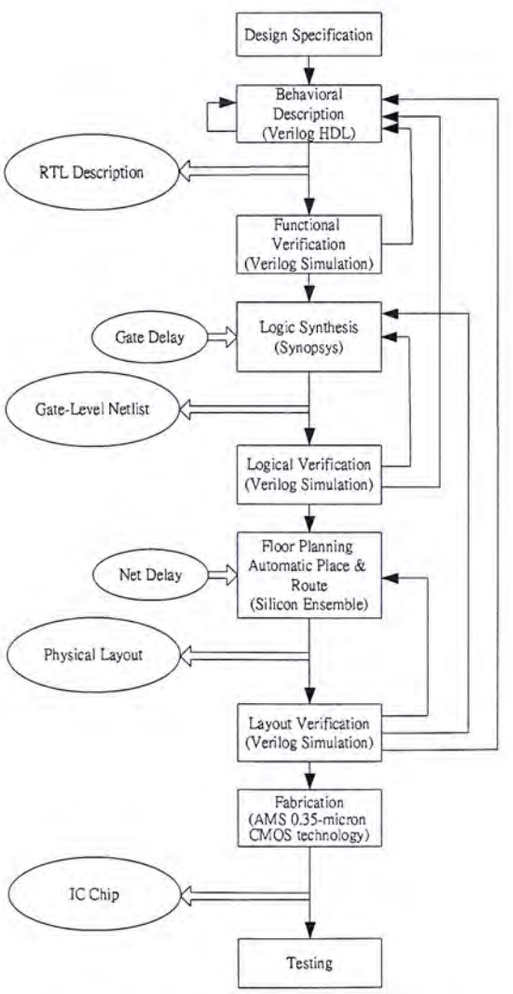



Figure 5-1. Design Flow of This Project

We have simulated the new design against two different references. One is a software recognition system using the same algorithm [13], and the other is a similar hardware recognizer without the look-up table. In the software recognition system there is almost no approximation in pdf calculation and no need to convert the input data into 16-bit fix-point binary number, the recognition result can be viewed as a theoretical one. In the hardware recognizer without the look-up table, only $\ln C_{\max} + X_{\max}$ is considered in pdf calculation and the term $\ln(1+\frac{C_{i\min}}{C_{i\max}} e^{X_{i\min}-X_{i\max}})$ is ignored. This is equal to using an all-zero look-up table in the proposed architecture (Figure 4-5).

The test speech data are 353 speeches from AURORA 2 database [14][15]. These speeches were first imported into a software feature extraction program to be converted into the feature vectors. These feature vectors are needed as inputs for all the three recognizers. After simulation we checked if the recognizers gave the correct recognition results. The simulation results are tabulated in Table 2. We can see that if we just approximate $\ln(C_1 e^{X_1} + C_2 e^{X_2})$ by $\ln C_{\max} + X_{\max}$, the recognition accuracy dropped by 1.4% compared with the theoretical results. As discussed in chapter 4, we can increase the recognition accuracy with a look-up table. The test results indicate that the recognition accuracy has increased by 0.9% with a look-up table design.

| | Software | Hardware | |
|---|---|---|---|
| | | Without LUT | With LUT (proposed) |
| Word Accuracy (%) | 94.3% | 92.9% | 93.8% |

Table 2. Simulation Results

# 5.2. Testing

A double-mixture HMM based speech recognizer with a look-up table test chip is fabricated with AMS 0.35-micron CMOS technology, and the specification is shown in Table 3 (Appendix II).

| Specification | Value |
|---|---|
| CMOS Technology | 0.35um |
| Operating Voltage | 3.3V |
| Total gate count | 30000 |
| Die area | 12.25 sq.mm |
| Package | PGA100 |

Table 3. Specifications of the New Speech IC

The 100-pin package includes data and address lines for two external memories, the system's inputs and outputs (e.g. reset, start signal, recognized word index, done signal), power pins, and some pins to keep track on the internal data in terms that there is any error occurs (Appendix III).

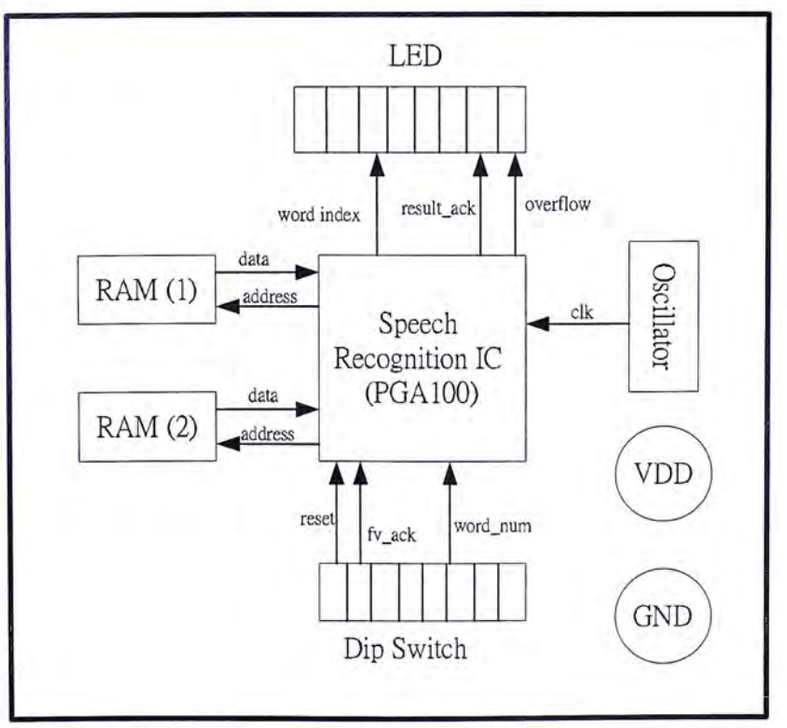



Figure 5-2. A Brief Diagram of the PCB

A PCB has been made to test the new speech IC (Appendix IV). Figure 5-2 is a brief block diagram of the PCB. In this PCB, RAM (1) is used to store the HMM of the system vocabulary, and RAM (2) is used to store feature vectors. These data are obtained from a software program and are written into these two RAMs during power up. The address and data of the rams are 16-bit data. A 6-bit dip switch is used to set the number of the words in the vocabulary, which can be varied from 1 to 63. The fv_ack signal initializes the recognition process. Every time the system detects a fv_ack single, it will start calculation from the beginning. The outputs are connected to 8 LEDs, among them 6 are for the recognition result Word_Index, one Result_Ack and one Overflow.

We started with some simple functional tests to make sure that the IC is working. For example, we set the word number of the vocabulary to 10 and the parameters of the different word models in RAM(1) to all 1, all 2, ..., all 10, separately. In RAM(2), we wrote 10 to all the storage units. The recognition result was 10, which was as same as the theoretical result. This simple functional test was performed several times and every time a right result was obtained.

Then we came to the more complicated speech tests. This time we verified the new chip with the same set of AURORA 2 test data used in the simulation. With the same word models and feature vectors, we obtained the exactly same recognition results as the simulation. Also we wrote all zero into the look-up table to verify the recognition accuracy of a double-mixture speech recognizer without the look-up table. The recognition results were identical with the

simulation. These functional tests have demonstrated that the IC can recognize real-world speech. We recorded that the maximum operating frequency of the speech IC is around 62.5 MHz and the average power consumption is 56.7mW at 20MHz.

Finally we connected our testing chip with a DSP board to perform real-time speech recognition. The function of the DSP board is to generate the feature vectors of the speech to be recognized. Figure 5-3 is a block diagram of this real-time speech recognition testing system. The word models were pre-trained by a software program and written into RAM(1) during power up. Then a person said a word towards the microphone. The DSP board generated the feature vectors of the input speech and wrote them into RAM(2). After that, the DSP board gave the speech recognition IC a start signal. In a very short time, the recognition result would be shown by the LEDs.

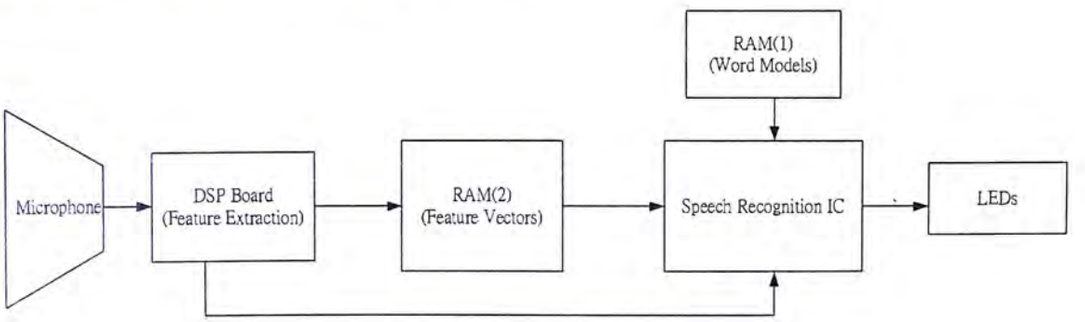



Figure 5-3. Block Diagram of the Real-Time Speech Recognition Testing System

# Chapter 6 Discussion and Conclusion

In this thesis a new architecture of multi-mixture HMM based speech recognizer has been presented. Using a table look-up method, higher order mixture HMM speech recognizer can be implemented with accuracy, matching software recognizer.

Since the add-log operation is unavoidable in the pdf calculation and its effect is significant to the final recognition result, finding an uncomplicated, universal and accurate implementation of this operation is important for the design of a speech recognizer. The table look-up method is such an implementation method that can fulfill these requirements. From Table 2 it can be seen that in a double-mixture HMM system, the proposed new speech recognition IC has approximately the same recognition accuracy as the software recognizer, and its recognition results are better than that of the other hardware recognizer without the look-up table.

Moreover, this new technique can be applied to different high-order mixture systems with minor modifications, as stated in chapter 4.5. In those designs the only difference between the high-order mixture systems is the size of the look-up table. For example, given an accuracy equivalent to two places after the decimal point floating number, there are only 70 and 140 values in the look-up

table for a double-mixture HMM system and a four-mixture HMM system, respectively.

However, truncation is not considered when designing this double-mixture HMM speech recognition IC. All internal buses are 48 bits long, which requires more operation time, more areas and more power. But as a trade-off, truncation will introduce approximation error into the calculation of the pdf. If too many bits are truncated, the recognition accuracy will be too low that the system is not suitable for real-world applications. The effect should be carefully considered when designers intend to employ truncation in the implementation of speech recognition system.

# Reference

[1]. Fundamentals of Speech Recognition
Lawrence Rabiner, Biing-Hwang Juang
Prentice Hall

[2]. ASIC System Design with VHDL: A Paradigm
Steven S. Leung, Michael A. Shanblatt
Kluwer Academic Publishers

[3]. Verilog HDL: A Guide to Digital Design and Synthesis
Samir Paluitkar

[4]. Advanced Digital Design with the Verilog HDL
Michael D. Ciletti
Prentice Hall

[5]. "A Tutorial on Hidden Markov Models and Selected Applications in speech Recognition"
R. Rabiner
Proceedings of the IEEE, Volume: 77 Issue: 2, pp. 257-286 Feb. 1989

[6]. "Efficient Viterbi Scoring Architecture For HMM-Based Speech Recognition System"
Y. S. Cho, J. Y. Kim and H. S. Lee
IEEE Electronics Letters Vol. 28, No. 25, pp. 2338-2340, Dec. 1992

[7]. "An Efficient VLSI Architecture for HMM-Based Speech Recognition"
J. M. Jou, Y. H. Shiau and C. J. Huang
Electronics, Circuits and Systems, 2001. ICECS 2001.
The 8th IEEE International Conference, Vol.1, pp. 469–472, 2001

[8]. "A VLSI Implementation of Pdf Computations in HMM Based Speech Recognition"
J. Pihl, T. Svendsen and M. H. Johmsen
TENCON '96. Proceedings., 1996 IEEE TENCON.
Digital Signal Processing Applications, Vol.1, pp. 241-246, 1996

[9]. "A VLSI Wordprocessing Subsystem for a Real Time Large Vocabulary Continuous Speech Recognition System"
Stölzle, S. Narayanaswamy, K. Kornegay, J. Rabaey and R. W. Brodersen
Custom Integrated Circuits Conference, 1989., Proceedings of the IEEE 1989, pp. 20.7/1-20.7/5, 1989

[10]. "Low Power VLSI Architecture of Viterbi Scorer for HMM-Based Isolated Word Recognition"
G. Park, K. S. Cho and J. D. Cho
Quality Electronic Design, 2002. Proceedings. International Symposium on, pp. 235-239, 2002

[11]. http://www.seas.upenn.edu/~ee201/lab/CarryLookAhead/CarryLookAhead F01.html

[12]. http://www-zeuthen.desy.de/ape/html/APEmille/Documentation/Hardware/Tmille/talu_ch9.pdf

[13]. "The HTK BOOK (for HTK Version 2.2)"
S. J. Young, P.C. Woodland and W. J. Byrne,
Entropic Ltd., Jan. 1999

[14]. http://www.elda.fr/proj/aurora2.htm

[15]. "The AURORA experimental framework for the performance evaluation of speech recognition systems under noisy conditions"
H.-G. Hirsh and D. Pearce
Proceedings of ISCA ITRW ASR 2000, Paris, France, September 2000

[16]. "An Hmm-Based Speech Recognition IC"
Wei Han; Kwok-Wai Hon; Cheong-Fat Chan; Tam Lee; Chiu-Sing Choy; Kong-Pang Pun; Ching, P.C.;
Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on , Volume: 2 , May 25-28, 2003
Page(s): 744 -747

# Appendix I Verilog Code of the Double-Mixture HMM Based Speech Recognition IC (RTL Level)

## ✧ Subtractor

```
module cla16_dm(sumout, overflow, result_ack, in1, in2, out_sel, clk, reset);
/* ----- carry look ahead adder ----- */
input [15:0] in1;
input [15:0] in2;
input out_sel;      /* ----- select the output of (1'b0)x-xmean or (1'b1)variance ----- */
input clk;
input reset;
input result_ack;
output [15:0] sumout;
output overflow;
wire [15:0] in1_reg, in2_reg;
reg [15:0] sum;
reg overf;
wire [15:0] p, g;
wire [15:0] carout;
assign sumout = sum;
assign overflow = overf;
/* ----- when result_ack == 1, all the inputs are set to 0 to save power ----- */
assign in1_reg = (result_ack == 0) ? in1 : 16'b0;
assign in2_reg = (result_ack == 0) ? in2 : 16'b0;
assign p = in1_reg | in2_reg;
assign g = in1_reg & in2_reg;
assign carout[0] = g[0];
assign carout[15:1] = g[15:1] | p[15:1] & carout[14:0];
always @(posedge clk or negedge reset)
begin
if (reset == 0)
begin
      sum <= 0;
      overf <= 0;
end
else
begin
      if (out_sel == 0)
            sum[15:0] <= in1_reg[15:0] ^ in2_reg[15:0] ^ {carout[14:0] , 1'b0};
      else
            sum[15:0] <= in2_reg[15:0];
      if (out_sel == 0)
            if (carout[15] ^ carout[14] == 1)
                  overf <= 1;
end
```

```
end
endmodule
```

## ✧ Multiplier

```
module booth_dm(out1, in1, in2);
parameter zee = 33'bz;
input [2:0] in1;
input [31:0] in2;
output [32:0] out1;
assign out1 = (in1 == 3'b000) ? 33'b0 : zee;
assign out1 = (in1 == 3'b001) ? {in2[31], in2} : zee;
assign out1 = (in1 == 3'b010) ? {in2[31], in2} : zee;
assign out1 = (in1 == 3'b011) ? {in2[31:0], 1'b0} : zee;
assign out1 = (in1 == 3'b100) ? ~{in2[31:0], 1'b0} + 1'b1 : zee;
assign out1 = (in1 == 3'b101) ? (~{in2[31], in2}) + 1'b1 : zee;
assign out1 = (in1 == 3'b110) ? (~{in2[31], in2}) + 1'b1 : zee;
assign out1 = (in1 == 3'b111) ? 33'b0 : zee;
endmodule

module fulladder_dm(cout, sumout, in1, in2, in3);
input [47:0] in1, in2, in3;
output [47:0] cout, sumout;
assign sumout = (in1^in2)^in3;
assign cout = ((in1^in2)&in3)|(in1&in2);
endmodule

module cla48_dm(sumout, in1, in2, counter_cla48, clk, reset);
/* ----- carry look ahead adder ----- */
input [47:0] in1;
input [47:0] in2;
input counter_cla48;
input clk;
input reset;
output [47:0] sumout;
reg [47:0] sum;
wire [47:0] p, g;
wire [47:0] carout;
assign sumout = sum;
assign p = in1 | in2;
assign g = in1 & in2;
assign carout[0] = g[0];
assign carout[46:1] = g[46:1] | p[46:1] & carout[45:0];
always @(posedge clk or negedge reset)
begin
if (reset == 0)
        sum <= 0;
else
begin
        if (counter_cla48 == 0)
                sum[47:0] <= in1[47:0] ^ in2[47:0] ^ {carout[46:0] , 1'b0};
```

```
end
end
endmodule

module multiplier_dm(mulout, in_sel, in1, counter_cla48, clk, reset);
/* ----- booth multiplier ----- */
input [15:0] in1;
input in_sel, clk, reset;
input counter_cla48;
output [47:0] mulout;
wire [32:0] boothout1, boothout2, boothout3, boothout4, boothout5, boothout6, boothout7,
boothout8;
wire [47:0] cout1, cout2, cout3, cout4, cout5, cout6, cout7;
wire [47:0] mulout1, mulout2, mulout3, mulout4, mulout5, mulout6, mulout7, mulout_wire;
wire [31:0] xmean_sq;
assign xmean_sq = (in_sel == 1'b0) ? {{16{in1[15]}}, in1} : mulout_wire;
assign mulout = mulout_wire;
reg [47:0] cout7_reg, mulout7_reg;
booth_dm booth1_dm(boothout1, {in1[1:0], 1'b0}, xmean_sq);
booth_dm booth2_dm(boothout2, in1[3:1], xmean_sq);
booth_dm booth3_dm(boothout3, in1[5:3], xmean_sq);
booth_dm booth4_dm(boothout4, in1[7:5], xmean_sq);
booth_dm booth5_dm(boothout5, in1[9:7], xmean_sq);
booth_dm booth6_dm(boothout6, in1[11:9], xmean_sq);
booth_dm booth7_dm(boothout7, in1[13:11], xmean_sq);
booth_dm booth8_dm(boothout8, in1[15:13], xmean_sq);
fulladder_dm fulladder1_dm(cout1, mulout1, {14'b0, ~boothout1[32], boothout1},
                    {12'b0, ~boothout2[32], boothout2, 2'b0},
                    {10'b0, ~boothout3[32], boothout3, 4'b0});
fulladder_dm fulladder2_dm(cout2, mulout2, {8'b0, ~boothout4[32], boothout4, 6'b0},
                    {6'b0, ~boothout5[32], boothout5, 8'b0},
                    {4'b0, ~boothout6[32], boothout6, 10'b0});
fulladder_dm fulladder3_dm(cout3, mulout3, {2'b0, ~boothout7[32], boothout7, 12'b0},
                    {~boothout8[32], boothout8, 14'b0},
                    {15'b010101010101011, 33'b0});
fulladder_dm fulladder4_dm(cout4, mulout4, {cout1[46:0], 1'b0}, mulout1, {cout2[46:0],
1'b0});
fulladder_dm fulladder5_dm(cout5, mulout5, mulout2, {cout3[46:0], 1'b0}, mulout3);
fulladder_dm fulladder6_dm(cout6, mulout6, {cout4[46:0], 1'b0}, mulout4, {cout5[46:0],
1'b0});
fulladder_dm fulladder7_dm(cout7, mulout7, {cout6[46:0], 1'b0}, mulout6, mulout5);
cla48_dm cla48_dm(mulout_wire, {cout7_reg[46:0], 1'b0}, mulout7_reg, counter_cla48, clk,
reset);
always @(posedge clk or negedge reset)
begin
      if (reset == 0)
      begin
            cout7_reg <= 0;
            mulout7_reg <= 0;
      end
else
begin
            /* ----- adding a registers between fulladder and cla48 ----- */
```

```
            mulout7_reg <= mulout7;
            cout7_reg <= cout7;
end
end
endmodule
```

## ✧ Core_Adder

```
module core_cla48_dm(sumout, overflow, in_sel, in1, reg48_in, reg16_in, regx_in, clk, reset);
input [47:0] reg48_in;
input [47:0] in1;
input [47:0] regx_in;
input [47:0] reg16_in;
input [1:0] in_sel;
input clk;
input reset;
output [47:0] sumout;
output overflow;
reg [47:0] sum;
reg carry;
reg overf;
wire [47:0] p, g;
wire [47:0] carout;
wire [47:0] intmp1;
/* ----- select the input between the multiplier output and the reg16 output ----- */
assign intmp1 = (in_sel == 2'b00) ? in1 : 48'bz;
assign intmp1 = (in_sel == 2'b10) ? reg48_in : 48'bz;
assign intmp1 = (in_sel == 2'b11) ? reg16_in : 48'bz;
assign sumout = sum;
assign overflow = overf;
assign p = intmp1 | regx_in;
assign g = intmp1 & regx_in;
assign carout[0] = g[0];
assign carout[47:1] = g[47:1] | p[47:1] & carout[46:0];
always @(posedge clk or negedge reset)
begin
if (reset == 0)
begin
        sum <= 0;
        carry <= 0;
        overf <= 0;
end
else
begin
        carry <= carout[47];
        sum[47:1] <= intmp1[47:1] ^ regx_in[47:1] ^ carout[46:0];
        sum[0] <= intmp1[0] ^ regx_in[0];
        if (carout[47] ^ carout[46] == 1)
        begin
                overf <= 1;
        end
```

```
end
end
endmodule
```

## ✧ Register for X

```
module regx_dm(dout, dout_x1, dout_x2,
        adder_in, start, load_adder, store, load_x2, sel_x2, wr_en_regx, fv_ack,
        clk, reset);
output [47:0] dout, dout_x1, dout_x2;
input [47:0] adder_in;
input start;
input load_adder, store, load_x2;
input sel_x2;
input wr_en_regx;
input fv_ack;
input clk, reset;
reg [47:0] dout;
reg [47:0] reg_x1, reg_x2;
reg flag;
assign dout_x1 = reg_x1;
assign dout_x2 = reg_x2;
always @(posedge clk or negedge reset)
begin
    if (reset == 0)
    begin
        dout <= 0;
        reg_x1 <= 0;
        reg_x2 <= 0;
        flag <= 0;
    end
    else
    begin
    if (fv_ack == 1)
    begin
        dout <= 0;
        reg_x1 <= 0;
        reg_x2 <= 0;
        flag <= 0;
    end
    else
    begin
        if (start == 1)
        begin
            dout <= 0;
            reg_x2 <= adder_in;
            flag <= 1;
        end
        else
        if(sel_x2 == 1)
        begin
```

```
                reg_x1 <= reg_x2;
                dout <= 0;
                flag <= 1;
            end
            else
            if (store == 1)
            begin
                reg_x2 <= adder_in;
                dout <= reg_x1;
                flag <= 1;
            end
            else
            if (load_x2 ==1)
            begin
                reg_x1 <= adder_in;
                dout <= reg_x2;
                flag <= 1;
            end
            else
            if (load_adder == 1)
            begin
                dout <= 0;
                reg_x1 <= adder_in;
                flag <= 1;
            end
            else
            if (flag == 1)
            begin
                dout <= dout;
                flag <= 0;
            end
            else
            if (wr_en_regx ==1)
                dout <= adder_in;
            else
                dout <= dout;
        end
        end
end
endmodule
```

## ✧ Subtractor and Comparator

```
module subcomp_dm(out, sel_x2, in1, in2, en, clk, reset);
input [47:0] in1, in2;
input en;
input clk, reset;
output [47:0] out;
output sel_x2;
reg [47:0] out;
```

```
reg sel_x2;
wire [47:0] im2;
wire [47:0] p, g;
wire [47:0] c;
wire [47:0] sum, sum1;
assign im2 = -in2;
assign p = in1 | im2;
assign g = in1 & im2;
assign c[0] = g[0];
assign c[47:1] = g[47:1] | p[47:1] & c[46:0];
assign sum[47:1] = in1[47:1] ^ im2[47:1] ^ c[46:0];
assign sum[0] = in1[0] ^ im2[0];
assign sum1 = -sum;
always @(posedge clk or negedge reset)
begin
        if (reset == 0)
        begin
                out <= 0;
                sel_x2 <= 0;
        end
        else
        begin
                if (en == 1)
                begin
                        if(sum[47] == 0)
                        begin
                                out <= sum1;
                                sel_x2 <= 0;
                        end
                        else
                        begin
                                out <= sum;
                                sel_x2 <= 1;
                        end
                end
                else
                        out <= out;
                        if (sel_x2 == 1)
                        sel_x2 <= 0;
        end
end
endmodule
```

## ✧ Shifter

```
module shift_dm(shift_num, datain, dataout, overflow, en, clk, reset);
input [47:0] datain;
input [5:0] shift_num;
input en;
input clk, reset;
output [47:0] dataout;
```

```
output overflow;
reg [47:0] dataout;
reg overflow;
always @(posedge clk or negedge reset)
begin
if(reset == 0)
begin
        dataout <= 0;
        overflow <= 0;
end
else
begin
                if(en==1)
                case (shift_num)
                        6'd0: dataout <= datain;
                        6'd1: begin dataout[47:1] <= datain[46:0]; dataout[0] <= 0;
                                overflow <= ~((datain[47:46]==2'b11)&&(datain[45:0]!= 0)); end
                        6'd2: begin dataout[47:2] <= datain[45:0]; dataout[1:0] <= 0;
                                overflow <= ~((datain[47:45]==3'b111)&(datain[44:0]!= 0)); end
                        6'd3: begin dataout[47:3] <= datain[44:0]; dataout[2:0] <= 0;
                                overflow <= ~((datain[47:44]==4'hf)&(datain[43:0]!= 0)); end
                        6'd4: begin dataout[47:4] <= datain[43:0]; dataout[3:0] <= 0;
                                overflow <= ~((datain[47:43]==5'b11111)&(datain[42:0]!= 0)); end
                        6'd5: begin dataout[47:5] <= datain[42:0]; dataout[4:0] <= 0;
                                overflow <= ~((datain[47:42]==6'b111111)&(datain[41:0]!= 0));end
                        6'd6: begin dataout[47:6] <= datain[41:0]; dataout[5:0] <= 0;
                                overflow <= ~((datain[47:41]==7'b1111111)&(datain[40:0]!= 0)); end
                        6'd7: begin dataout[47:7] <= datain[40:0]; dataout[6:0] <= 0;
                                overflow <= ~((datain[47:40]==8'hff)&(datain[39:0]!= 0)); end
                        6'd8: begin dataout[47:8] <= datain[39:0]; dataout[7:0] <= 0;
                                overflow <= ~((datain[47:39]==9'b111111111)&(datain[38:0]!= 0));
end
                        6'd9: begin dataout[47:9] <= datain[38:0]; dataout[8:0] <= 0;
                               overflow <= ~((datain[47:38]==10'b1111111111)&(datain[37:0]!= 0));
end
                        6'd10: begin dataout[47:10] <= datain[37:0]; dataout[9:0] <= 0;
                                overflow <= ~((datain[47:37]==11'b11111111111)&(datain[36:0]!=
0)); end
                        6'd11: begin dataout[47:11] <= datain[36:0]; dataout[10:0] <= 0;
                                overflow <= ~((datain[47:36]==12'hfff)&(datain[35:0]!= 0)); end
                        6'd12: begin dataout[47:12] <= datain[35:0]; dataout[11:0] <= 0;
                                overflow <= ~((datain[47:35]==13'b1111111111111)&(datain[34:0]!=
0)); end
                        6'd13: begin dataout[47:13] <= datain[34:0]; dataout[12:0] <= 0;
                                overflow <=
~((datain[47:34]==14'b11111111111111)&(datain[33:0]!= 0)); end
                        6'd14: begin dataout[47:14] <= datain[33:0]; dataout[13:0] <= 0;
                                overflow <=
~((datain[47:33]==15'b111111111111111)&(datain[32:0]!= 0)); end
                        6'd15: begin dataout[47:15] <= datain[32:0]; dataout[14:0] <= 0;
                                overflow <= ~((datain[47:32]==16'hffff)&(datain[31:0]!= 0)); end
                        6'd16: begin dataout[47:16] <= datain[31:0]; dataout[15:0] <= 0;
                                overflow <=
```

```
~((datain[47:31]==17'b11111111111111111)&(datain[30:0]!= 0)); end
                6'd17: begin dataout[47:17] <= datain[30:0]; dataout[16:0] <= 0;
                    overflow <=
~((datain[47:30]==18'b111111111111111111)&(datain[29:0]!= 0)); end
                6'd18: begin dataout[47:18] <= datain[29:0]; dataout[17:0] <= 0;
                    overflow <=
~((datain[47:29]==19'b1111111111111111111)&(datain[28:0]!= 0)); end
                6'd19: begin dataout[47:19] <= datain[28:0]; dataout[18:0] <= 0;
                    overflow <= ~((datain[47:28]==20'hfffff)&(datain[27:0]!= 0)); end
                6'd20: begin dataout[47:20] <= datain[27:0]; dataout[19:0] <= 0;
                    overflow <=
~((datain[47:27]==21'b11111111111111111111)&(datain[26:0]!= 0)); end
                6'd21: begin dataout[47:21] <= datain[26:0]; dataout[20:0] <= 0;
                    overflow <=
~((datain[47:26]==22'b111111111111111111111)&(datain[25:0]!= 0)); end
                6'd22: begin dataout[47:22] <= datain[25:0]; dataout[21:0] <= 0;
                    overflow <=
~((datain[47:25]==23'b1111111111111111111111)&(datain[24:0]!= 0)); end
                6'd23: begin dataout[47:23] <= datain[24:0]; dataout[22:0] <= 0;
                    overflow <= ~((datain[47:24]==24'hffffff)&(datain[23:0]!= 0)); end
                6'd24: begin dataout[47:24] <= datain[23:0]; dataout[23:0] <= 0;
                    overflow <=
~((datain[47:23]==25'b11111111111111111111111)&(datain[22:0]!= 0)); end
                6'd25: begin dataout[47:25] <= datain[22:0]; dataout[24:0] <= 0;
                    overflow <=
~((datain[47:22]==26'b111111111111111111111111)&(datain[21:0]!= 0)); end
                6'd26: begin dataout[47:26] <= datain[21:0]; dataout[25:0] <= 0;
                    overflow <=
~((datain[47:21]==27'b1111111111111111111111111)&(datain[20:0]!= 0)); end
                6'd27: begin dataout[47:27] <= datain[20:0]; dataout[26:0] <= 0;
                    overflow <= ~((datain[47:20]==28'hfffffff)&(datain[19:0]!= 0)); end
                6'd28: begin dataout[47:28] <= datain[19:0]; dataout[27:0] <= 0;
                    overflow <=
~((datain[47:19]==29'b11111111111111111111111111)&(datain[18:0]!= 0)); end
                6'd29: begin dataout[47:29] <= datain[18:0]; dataout[28:0] <= 0;
                    overflow <=
~((datain[47:18]==30'b111111111111111111111111111)&(datain[17:0]!= 0)); end
                6'd30: begin dataout[47:30] <= datain[17:0]; dataout[29:0] <= 0;
                    overflow <=
~((datain[47:17]==31'b1111111111111111111111111111)&(datain[16:0]!= 0)); end
                6'd31: begin dataout[47:31] <= datain[16:0]; dataout[30:0] <= 0;
                    overflow <= ~((datain[47:16]==32'hffffffff)&(datain[15:0]!= 0)); end
                6'd32: begin dataout[47:32] <= datain[15:0]; dataout[31:0] <= 0;
                    overflow <=
~((datain[47:15]==33'b11111111111111111111111111111)&(datain[14:0]!= 0)); end
                6'd33: begin dataout[47:33] <= datain[14:0]; dataout[32:0] <= 0;
                    overflow <=
~((datain[47:14]==34'b111111111111111111111111111111)&(datain[13:0]!= 0)); end
                6'd34: begin dataout[47:34] <= datain[13:0]; dataout[33:0] <= 0;
                    overflow <=
~((datain[47:13]==35'b1111111111111111111111111111111)&(datain[12:0]!= 0)); end
                6'd35: begin dataout[47:35] <= datain[12:0]; dataout[34:0] <= 0;
                    overflow <= ~((datain[47:12]==36'hfffffffff)&(datain[11:0]!= 0)); end
```

```
                    6'd36: begin dataout[47:36] <= datain[11:0]; dataout[35:0] <= 0;
                        overflow <=
    ~((datain[47:11]==37'b1111111111111111111111111111111111111)&(datain[10:0]!= 0)); end
                    6'd37: begin dataout[47:37] <= datain[10:0]; dataout[36:0] <= 0;
                        overflow <=
    ~((datain[47:10]==38'b11111111111111111111111111111111111111)&(datain[9:0]!= 0)); end
                endcase
    end
    end
endmodule
```

## ✧ Look-Up Table

```
module lut_dm(out, in, shift_overf, clk, reset);
input [47:0] in;
input   shift_overf;
input clk;
input reset;
output [47:0] out;
reg [47:0] out;
always @(posedge clk or negedge reset)
begin
if( reset == 0 )
      out <= 0;
else
begin
      if(shift_overf == 1)
      begin
            out[15:0]<=16'h401c;
            out[31:16]<=16'h401b;
            out[47:32]<=16'h401a;
      end
      else
      begin
      if(in <-48'd93164986044373)
      begin
            out[15:0]<=16'h401c;
            out[31:16]<=16'h401b;
            out[47:32]<=16'h401a;
      end
      if(in>=-48'd93164986044373 && in <-48'd73749886739974)
      begin
            out[15:0]<=16'h403c;
            out[31:16]<=16'h403b;
            out[47:32]<=16'h403a;
      end
      if(in>=-48'd73749886739974 && in <-48'd64675093199182)
      begin
            out[15:0]<=16'h405c;
            out[31:16]<=16'h405b;
            out[47:32]<=16'h405a;
```

```
end
if(in>=-48'd64675093199182 && in <-48'd58667410285668)
begin
        out[15:0]<=16'h407c;
        out[31:16]<=16'h407b;
        out[47:32]<=16'h407a;
end
if(in>=-48'd58667410285668 && in <-48'd54157692787162)
begin
        out[15:0]<=16'h409c;
        out[31:16]<=16'h409b;
        out[47:32]<=16'h409a;
end
if(in>=-48'd54157692787162 && in <-48'd50538762671801)
begin
        out[15:0]<=16'h40bc;
        out[31:16]<=16'h40bb;
        out[47:32]<=16'h40ba;
end
if(in>=-48'd50538762671801 && in <-48'd47511075648541)
begin
        out[15:0]<=16'h40dc;
        out[31:16]<=16'h40db;
        out[47:32]<=16'h40da;
end
if(in>=-48'd47511075648541 && in <-48'd44904631927252)
begin
        out[15:0]<=16'h40fc;
        out[31:16]<=16'h40fb;
        out[47:32]<=16'h40fa;
end
if(in>=-48'd44904631927252 && in <-48'd42613605013445)
begin
        out[15:0]<=16'h411c;
        out[31:16]<=16'h411b;
        out[47:32]<=16'h411a;
end
if(in>=-48'd42613605013445 && in <-48'd40567622782138)
begin
        out[15:0]<=16'h413c;
        out[31:16]<=16'h413b;
        out[47:32]<=16'h413a;
end
if(in>=-48'd40567622782138 && in <-48'd38717509258398)
begin
        out[15:0]<=16'h415c;
        out[31:16]<=16'h415b;
        out[47:32]<=16'h415a;
end
if(in>=-48'd38717509258398 && in <-48'd37027543589924)
begin
        out[15:0]<=16'h417c;
        out[31:16]<=16'h417b;
```

```
        out[47:32]<=16'h417a;
    end
    if(in>=-48'd37027543589924 && in <-48'd35470959086759)
    begin
        out[15:0]<=16'h419c;
        out[31:16]<=16'h419b;
        out[47:32]<=16'h419a;
    end
    if(in>=-48'd35470959086759 && in <-48'd34027179919926)
    begin
        out[15:0]<=16'h41bc;
        out[31:16]<=16'h41bb;
        out[47:32]<=16'h41ba;
    end
    if(in>=-48'd34027179919926 && in <-48'd32680047850082)
    begin
        out[15:0]<=16'h41dc;
        out[31:16]<=16'h41db;
        out[47:32]<=16'h41da;
    end
    if(in>=-48'd32680047850082 && in <-48'd31416641653973)
    begin
        out[15:0]<=16'h41fc;
        out[31:16]<=16'h41fb;
        out[47:32]<=16'h41fa;
    end
    if(in>=-48'd31416641653973 && in <-48'd30226466348532)
    begin
        out[15:0]<=16'h421c;
        out[31:16]<=16'h421b;
        out[47:32]<=16'h421a;
    end
    if(in>=-48'd30226466348532 && in <-48'd29100881369237)
    begin
        out[15:0]<=16'h423c;
        out[31:16]<=16'h423b;
        out[47:32]<=16'h423a;
    end
    if(in>=-48'd29100881369237 && in <-48'd28032687876098)
    begin
        out[15:0]<=16'h425c;
        out[31:16]<=16'h425b;
        out[47:32]<=16'h425a;
    end
    if(in>=-48'd28032687876098 && in <-48'd27015824838488)
    begin
        out[15:0]<=16'h427c;
        out[31:16]<=16'h427b;
        out[47:32]<=16'h427a;
    end
    if(in>=-48'd27015824838488 && in <-48'd26045141208395)
    begin
        out[15:0]<=16'h429c;
```

```
        out[31:16]<=16'h429b;
        out[47:32]<=16'h429a;
end
if(in>=-48'd26045141208395 && in <-48'd25116222408564)
begin
        out[15:0]<=16'h42bc;
        out[31:16]<=16'h42bb;
        out[47:32]<=16'h42ba;
end
if(in>=-48'd25116222408564 && in <-48'd24225256301488)
begin
        out[15:0]<=16'h42dc;
        out[31:16]<=16'h42db;
        out[47:32]<=16'h42da;
end
if(in>=-48'd24225256301488 && in <-48'd23368928326696)
begin
        out[15:0]<=16'h42fc;
        out[31:16]<=16'h42fb;
        out[47:32]<=16'h42fa;
end
if(in>=-48'd23368928326696 && in <-48'd22544338505747)
begin
        out[15:0]<=16'h431c;
        out[31:16]<=16'h431b;
        out[47:32]<=16'h431a;
end
if(in>=-48'd22544338505747 && in <-48'd21748935061068)
begin
        out[15:0]<=16'h433c;
        out[31:16]<=16'h433b;
        out[47:32]<=16'h433a;
end
if(in>=-48'd21748935061068 && in <-48'd20980460810904)
begin
        out[15:0]<=16'h435c;
        out[31:16]<=16'h435b;
        out[47:32]<=16'h435a;
end
if(in>=-48'd20980460810904 && in <-48'd20236909498671)
begin
        out[15:0]<=16'h437c;
        out[31:16]<=16'h437b;
        out[47:32]<=16'h437a;
end
if(in>=-48'd20236909498671 && in <-48'd19516489926122)
begin
        out[15:0]<=16'h439c;
        out[31:16]<=16'h439b;
        out[47:32]<=16'h439a;
end
if(in>=-48'd19516489926122 && in <-48'd18817596274496)
begin
```

```
        out[15:0]<=16'h43bc;
        out[31:16]<=16'h43bb;
        out[47:32]<=16'h43ba;
end
if(in>=-48'd18817596274496 && in <-48'd18138783375145)
begin
        out[15:0]<=16'h43dc;
        out[31:16]<=16'h43db;
        out[47:32]<=16'h43da;
end
if(in>=-48'd18138783375145 && in <-48'd17478745971041)
begin
        out[15:0]<=16'h43fc;
        out[31:16]<=16'h43fb;
        out[47:32]<=16'h43fa;
end
if(in>=-48'd17478745971041 && in <-48'd16836301220389)
begin
        out[15:0]<=16'h441c;
        out[31:16]<=16'h441b;
        out[47:32]<=16'h441a;
end
if(in>=-48'd16836301220389 && in <-48'd16210373852535)
begin
        out[15:0]<=16'h443c;
        out[31:16]<=16'h443b;
        out[47:32]<=16'h443a;
end
if(in>=-48'd16210373852535 && in <-48'd15599983507843)
begin
        out[15:0]<=16'h445c;
        out[31:16]<=16'h445b;
        out[47:32]<=16'h445a;
end
if(in>=-48'd15599983507843 && in <-48'd15004233886964)
begin
        out[15:0]<=16'h447c;
        out[31:16]<=16'h447b;
        out[47:32]<=16'h447a;
end
if(in>=-48'd15004233886964 && in <-48'd14422303407774)
begin
        out[15:0]<=16'h449c;
        out[31:16]<=16'h449b;
        out[47:32]<=16'h449a;
end
if(in>=-48'd14422303407774 && in <-48'd13853437125386)
begin
        out[15:0]<=16'h44bc;
        out[31:16]<=16'h44bb;
        out[47:32]<=16'h44ba;
end
if(in>=-48'd13853437125386 && in <-48'd13296939715708)
```

```
begin
        out[15:0]<=16'h44dc;
        out[31:16]<=16'h44db;
        out[47:32]<=16'h44da;
end
if(in>=-48'd13296939715708 && in <-48'd12752169358868)
begin
        out[15:0]<=16'h44fc;
        out[31:16]<=16'h44fb;
        out[47:32]<=16'h44fa;
end
if(in>=-48'd12752169358868 && in <-48'd12218532387458)
begin
        out[15:0]<=16'h451c;
        out[31:16]<=16'h451b;
        out[47:32]<=16'h451a;
end
if(in>=-48'd12218532387458 && in <-48'd11695478587646)
begin
        out[15:0]<=16'h453c;
        out[31:16]<=16'h453b;
        out[47:32]<=16'h453a;
end
if(in>=-48'd11695478587646 && in <-48'd11182497059843)
begin
        out[15:0]<=16'h455c;
        out[31:16]<=16'h455b;
        out[47:32]<=16'h455a;
end
if(in>=-48'd11182497059843 && in <-48'd10679112560828)
begin
        out[15:0]<=16'h457c;
        out[31:16]<=16'h457b;
        out[47:32]<=16'h457a;
end
if(in>=-48'd10679112560828 && in <-48'd10184882261648)
begin
        out[15:0]<=16'h459c;
        out[31:16]<=16'h459b;
        out[47:32]<=16'h459a;
end
if(in>=-48'd10184882261648 && in <-48'd9699392865859)
begin
        out[15:0]<=16'h45bc;
        out[31:16]<=16'h45bb;
        out[47:32]<=16'h45ba;
end
if(in>=-48'd9699392865859 && in <-48'd9222258041064)
begin
        out[15:0]<=16'h45dc;
        out[31:16]<=16'h45db;
        out[47:32]<=16'h45da;
end
```

```
if(in>=-48'd9222258041064 && in <-48'd8753116123769)
begin
        out[15:0]<=16'h45fc;
        out[31:16]<=16'h45fb;
        out[47:32]<=16'h45fa;
end
if(in>=-48'd8753116123769 && in <-48'd8291628063361)
begin
        out[15:0]<=16'h461c;
        out[31:16]<=16'h461b;
        out[47:32]<=16'h461a;
end
if(in>=-48'd8291628063361 && in <-48'd7837475575941)
begin
        out[15:0]<=16'h463c;
        out[31:16]<=16'h463b;
        out[47:32]<=16'h463a;
end
if(in>=-48'd7837475575941 && in <-48'd7390359482800)
begin
        out[15:0]<=16'h465c;
        out[31:16]<=16'h465b;
        out[47:32]<=16'h465a;
end
if(in>=-48'd7390359482800 && in <-48'd6949998211831)
begin
        out[15:0]<=16'h467c;
        out[31:16]<=16'h467b;
        out[47:32]<=16'h467a;
end
if(in>=-48'd6949998211831 && in <-48'd6516126443062)
begin
        out[15:0]<=16'h469c;
        out[31:16]<=16'h469b;
        out[47:32]<=16'h469a;
end
if(in>=-48'd6516126443062 && in <-48'd6088493881986)
begin
        out[15:0]<=16'h46bc;
        out[31:16]<=16'h46bb;
        out[47:32]<=16'h46ba;
end
if(in>=-48'd6088493881986 && in <-48'd5666864146504)
begin
        out[15:0]<=16'h46dc;
        out[31:16]<=16'h46db;
        out[47:32]<=16'h46da;
end
if(in>=-48'd5666864146504 && in <-48'd5251013755061)
begin
        out[15:0]<=16'h46fc;
        out[31:16]<=16'h46fb;
        out[47:32]<=16'h46fa;
```

```
end
if(in>=-48'd5251013755061 && in <-48'd4840731205143)
begin
        out[15:0]<=16'h471c;
        out[31:16]<=16'h471b;
        out[47:32]<=16'h471a;
end
if(in>=-48'd4840731205143 && in <-48'd4435816132630)
begin
        out[15:0]<=16'h473c;
        out[31:16]<=16'h473b;
        out[47:32]<=16'h473a;
end
if(in>=-48'd4435816132630 && in <-48'd4036078543615)
begin
        out[15:0]<=16'h475c;
        out[31:16]<=16'h475b;
        out[47:32]<=16'h475a;
end
if(in>=-48'd4036078543615 && in <-48'd3641338111346)
begin
        out[15:0]<=16'h477c;
        out[31:16]<=16'h477b;
        out[47:32]<=16'h477a;
end
if(in>=-48'd3641338111346 && in <-48'd3251423531755)
begin
        out[15:0]<=16'h479c;
        out[31:16]<=16'h479b;
        out[47:32]<=16'h479a;
end
if(in>=-48'd3251423531755 && in <-48'd2866171931821)
begin
        out[15:0]<=16'h47bc;
        out[31:16]<=16'h47bb;
        out[47:32]<=16'h47ba;
end
if(in>=-48'd2866171931821 && in <-48'd2485428325659)
begin
        out[15:0]<=16'h47dc;
        out[31:16]<=16'h47db;
        out[47:32]<=16'h47da;
end
if(in>=-48'd2485428325659 && in <-48'd2109045113770)
begin
        out[15:0]<=16'h47fc;
        out[31:16]<=16'h47fb;
        out[47:32]<=16'h47fa;
end
if(in>=-48'd2109045113770 && in <-48'd1736881621429)
begin
        out[15:0]<=16'h481c;
        out[31:16]<=16'h481b;
```

```
            out[47:32]<=16'h481a;
        end
        if(in>=-48'd1736881621429 && in <-48'd1368803672575)
        begin
            out[15:0]<=16'h483c;
            out[31:16]<=16'h483b;
            out[47:32]<=16'h483a;
        end
        if(in>=-48'd1368803672575 && in <-48'd1004683195992)
        begin
            out[15:0]<=16'h485c;
            out[31:16]<=16'h485b;
            out[47:32]<=16'h485a;
        end
        if(in>=-48'd1004683195992 && in <-48'd644397860881)
        begin
            out[15:0]<=16'h487c;
            out[31:16]<=16'h487b;
            out[47:32]<=16'h487a;
        end
        if(in>=-48'd644397860881 && in <-48'd287830739229)
        begin
            out[15:0]<=16'h489c;
            out[31:16]<=16'h489b;
            out[47:32]<=16'h489a;
        end
        if(in>=-48'd287830739229)
        begin
            out[15:0]<=16'h48bc;
            out[31:16]<=16'h48bb;
            out[47:32]<=16'h48ba;
        end
        end
end
end
endmodule
```

## ✧ Register for Constants

```
module reg16_dm(data_out, data_in, addr, wr_en, fv_ack, clk, reset);
output [47:0] data_out;
input [15:0] data_in;
input [1:0] addr;
input wr_en;
input fv_ack;
input clk;
input reset;
reg [47:0] data_reg;
assign data_out = data_reg;
always @(posedge clk or negedge reset)
begin
```

```
        if (reset == 0)
        begin
                data_reg <= 48'd0;
        end
        else
        begin
                if (fv_ack == 1)
                        data_reg <= 48'd0;
                else
                begin
                if (wr_en == 1)
                begin
                        case (addr)
                                2'b10 : data_reg[47:32] <= data_in;
                                2'b01 : data_reg[31:16] <= data_in;
                                2'b00 : data_reg[15:0] <= data_in;
                        endcase
                end
                end
        end
end
endmodule
```

## ✧ Register for Scores

```
module reg48_dm(data_out1, data_in, in_sel, out_sel,
                wr_en, out_en_reg48, frame_counter, word_end_index,
                word_start_index, last_frame_counter, result_ack, //out_sel,
                counter_reg48, fv_ack, clk, reset);
output [47:0] data_out1;
input result_ack;
input [47:0] data_in;
input [3:0] in_sel;
input clk, reset;
input wr_en;
input out_en_reg48;
input [7:0] frame_counter;
input word_end_index;
input word_start_index;
input [2:0] last_frame_counter;
input [2:0] out_sel;
input [6:0] counter_reg48;
input fv_ack;
reg [47:0] data_reg1, data_reg2, data_reg3, data_reg4, data_reg5, data_reg6, data_reg7,
        data_reg8, data_reg_tmp;
/* ----- data_reg{1-8} store the cost without changing state, dtata_reg_tmp store the cost
with changing state ----- */
reg [47:0] data_out1_reg;
assign data_out1 = data_out1_reg;
always @(posedge clk or negedge reset)
begin
```

```
        if (reset == 0)
        begin
                data_reg1 <= 0;
                data_reg2 <= 0;
                data_reg3 <= 0;
                data_reg4 <= 0;
                data_reg5 <= 0;
                data_reg6 <= 0;
                data_reg7 <= 0;
                data_reg8 <= 0;
                data_reg_tmp <= 0;
                data_out1_reg <= 0;
        end
        else
        begin
                if (fv_ack == 1)
                begin
                        data_reg1 <= 0;
                        data_reg2 <= 0;
                        data_reg3 <= 0;
                        data_reg4 <= 0;
                        data_reg5 <= 0;
                        data_reg6 <= 0;
                        data_reg7 <= 0;
                        data_reg8 <= 0;
                        data_reg_tmp <= 0;
                        data_out1_reg <= 0;
                end
                else
                begin
                /* ----- assign different register to output ----- */
                if (out_en_reg48 == 1)
                begin
                        case (out_sel)
                                3'b000 : data_out1_reg <= data_reg1;
                                3'b001 : data_out1_reg <= data_reg2;
                                3'b010 : data_out1_reg <= data_reg3;
                                3'b011 : data_out1_reg <= data_reg4;
                                3'b100 : data_out1_reg <= data_reg5;
                                3'b101 : data_out1_reg <= data_reg6;
                                3'b110 : data_out1_reg <= data_reg7;
                                3'b111 : data_out1_reg <= data_reg8;
                        endcase
                end
                /* ----- reset all the parameters when calculation finished, in order to save power
----- */
                if (last_frame_counter[0] == 1 || result_ack == 1)
                        data_reg1 <= 0;
                if (word_start_index == 1 || result_ack == 1)
                begin
                        data_reg2 <= 0;
                        data_reg3 <= 0;
                        data_reg4 <= 0;
```

```
        data_reg5 <= 0;
        data_reg6 <= 0;
        data_reg7 <= 0;
        //data_reg8 <= 0;
        data_reg_tmp <= 0;
end
if (frame_counter == 8'd1)
        data_reg8 <= 0;
/* ----- storing the input when wr_en == 1 ----- */
if (result_ack == 0)
begin
        if (wr_en == 1)
        begin
                case (in_sel)
                        4'b0_000 : data_reg1 <= data_in;
                        4'b0_001 : data_reg2 <= data_in;
                        4'b0_010 : data_reg3 <= data_in;
                        4'b0_011 : data_reg4 <= data_in;
                        4'b0_100 : data_reg5 <= data_in;
                        4'b0_101 : data_reg6 <= data_in;
                        4'b0_110 : data_reg7 <= data_in;
                        4'b0_111 : data_reg8 <= data_in;
                        4'b1_000 : data_reg_tmp <= data_in;
                        4'b1_001 : data_reg_tmp <= data_in;
                        4'b1_010 : data_reg_tmp <= data_in;
                        4'b1_011 : data_reg_tmp <= data_in;
                        4'b1_100 : data_reg_tmp <= data_in;
                        4'b1_101 : data_reg_tmp <= data_in;
                        4'b1_110 : data_reg_tmp <= data_in;
                        4'b1_111 : data_reg_tmp <= data_in;
                        default;
                endcase
        end
end
/* ----- swap the registers when the data_reg_tmp > data_reg{1-8} ----- */
if (result_ack == 0)
begin
        if (counter_reg48 == 7'd37)
        begin
                if (out_sel == 3'b000)
                begin
                        case (frame_counter)
                        8'd1 : data_reg2 <= data_reg_tmp;
                        8'd2 : data_reg3 <= data_reg_tmp;
                        8'd3 : data_reg4 <= data_reg_tmp;
                        8'd4 : data_reg5 <= data_reg_tmp;
                        8'd5 : data_reg6 <= data_reg_tmp;
                        8'd6 : data_reg7 <= data_reg_tmp;
                        8'd7 : data_reg8 <= data_reg_tmp;
                        endcase
                end
        end
        if (counter_reg48 == 7'd36)
```

```
begin
        if (out_sel == 3'd2 && wr_en == 1'b1)
        begin
                if ({data_reg_tmp[47], data_reg2[47]} != 2'b10)
                begin
                        if ({data_reg_tmp[47], data_reg2[47]} == 2'b01)
                                data_reg2 <= data_reg_tmp;
                        else
                        if (data_reg_tmp > data_reg2)
                                data_reg2 <= data_reg_tmp;
                end
        end
        if (out_sel == 3'd3 && wr_en == 1'b1)
        begin
                if ({data_reg_tmp[47], data_reg3[47]} != 2'b10)
                begin
                        if ({data_reg_tmp[47], data_reg3[47]} == 2'b01)
                                data_reg3 <= data_reg_tmp;
                        else
                        if (data_reg_tmp > data_reg3)
                                data_reg3 <= data_reg_tmp;
                end
        end
        if (out_sel == 3'd4 && wr_en == 1'b1)
        begin
                if ({data_reg_tmp[47], data_reg4[47]} != 2'b10)
                begin
                        if ({data_reg_tmp[47], data_reg4[47]} == 2'b01)
                                data_reg4 <= data_reg_tmp;
                        else
                        if (data_reg_tmp > data_reg4)
                                data_reg4 <= data_reg_tmp;
                end
        end
        if (out_sel == 3'd5 && wr_en == 1'b1)
        begin
                if ({data_reg_tmp[47], data_reg5[47]} != 2'b10)
                begin
                        if ({data_reg_tmp[47], data_reg5[47]} == 2'b01)
                                data_reg5 <= data_reg_tmp;
                        else
                        if (data_reg_tmp > data_reg5)
                                data_reg5 <= data_reg_tmp;
                end
        end
        if (out_sel == 3'd6 && wr_en == 1'b1)
        begin
                if ({data_reg_tmp[47], data_reg6[47]} != 2'b10)
                begin
                        if ({data_reg_tmp[47], data_reg6[47]} == 2'b01)
                                data_reg6 <= data_reg_tmp;
                        else
                        if (data_reg_tmp > data_reg6)
```

```
                                        data_reg6 <= data_reg_tmp;
                                end
                        end
                        if (out_sel == 3'd7 && wr_en == 1'b1)
                        begin
                                if ({data_reg_tmp[47], data_reg7[47]} != 2'b10)
                                begin
                                        if ({data_reg_tmp[47], data_reg7[47]} == 2'b01)
                                                data_reg7 <= data_reg_tmp;
                                        else
                                        if (data_reg_tmp > data_reg7)
                                                data_reg7 <= data_reg_tmp;
                                end
                        end
                        if (out_sel == last_frame_counter && last_frame_counter !=0)
                        begin
                                if (wr_en == 1'b1)
                                begin
                                        if ({data_reg_tmp[47], data_reg8[47]} != 2'b10)
                                        begin
                                                if ({data_reg_tmp[47], data_reg8[47]} == 2'b01)
                                                        data_reg8 <= data_reg_tmp;
                                                else
                                                if (data_reg_tmp > data_reg8)
                                                        data_reg8 <= data_reg_tmp;
                                        end
                                end
                        end
                        if (out_sel == 3'd0 && wr_en == 1'b1 && frame_counter != 8'd7)
                        begin
                                if ({data_reg_tmp[47], data_reg8[47]} != 2'b10)
                                begin
                                        if ({data_reg_tmp[47], data_reg8[47]} == 2'b01)
                                                data_reg8 <= data_reg_tmp;
                                        else
                                        if (data_reg_tmp > data_reg8)
                                                data_reg8 <= data_reg_tmp;
                                end
                        end
                end
        end
    end
    end
end
endmodule
```

## ✧ Final Score Register

```
module final_score_reg_dm(word_index_out,
                word_end_index, final_comp_index, fv_ack, data_fscore, clk, reset);
```

```
output [5:0] word_index_out;
input word_end_index, final_comp_index, clk, reset;
input fv_ack;
input [47:0] data_fscore;
reg [47:0] data_reg;
reg [5:0] word_index_out_reg;
reg [5:0] word_index_counter;
assign word_index_out = word_index_out_reg;
always @(posedge clk or negedge reset)
begin
	if (reset == 0)
	begin
		data_reg <= 48'h800000000000;
		word_index_out_reg <= 0;
		word_index_counter <= 0;
	end
	else
	begin
		if (fv_ack == 1)
		begin
			/* ----- reset all the parameters when fv_ack == 1 ----- */
			data_reg <= 48'h800000000000;
			word_index_out_reg <= 0;
			word_index_counter <= 0;
		end
		else
		begin
			/* ----- counting the number of words have been calculated ----- */
			if (word_end_index == 1)
				word_index_counter <= word_index_counter + 1;
/* -- replace the old word index when the new global cost is higher than the old global cost -- */
			if (final_comp_index == 1 && word_index_counter != 0)
			begin
				if ({data_fscore[47], data_reg[47]} != 2'b10)
				begin
					if ({data_fscore[47], data_reg[47]} == 2'b01)
					begin
						data_reg <= data_fscore;
						word_index_out_reg <= word_index_counter;
					end
					else
					if (data_fscore > data_reg)
					begin
						data_reg <= data_fscore;
						word_index_out_reg <= word_index_counter;
					end
				end
			end
		end
	end
end
endmodule
```

## ✧ Controller

```
module control_dm(final_comp_index, result_ack,
                  regx_load_adder, regx_load_x2, regx_start, subcomp_en, counter_reg48,
regx_store, wr_en_regx,
                  addr_reg16, wr_en_reg16,
                  lut_out, rom_out,
                  shift_num, shift_en,
                  frame_counter, address_common, address_ram, address_rom,
                  in_sel_mul,
                  out_sel_cla16,
                  in_sel_cla48, in_sel_reg48, out_sel_reg48, wr_en_reg48, out_en_reg48,
                  fv_ack, word_end_index, word_start_index, last_frame_counter, word_index_out,
word_num, in1,
                  clk, reset);
output final_comp_index;
output regx_load_adder;
output regx_load_x2;
output regx_store;
output regx_start;
output subcomp_en;
output [6:0] counter_reg48;
output result_ack;
output [1:0] addr_reg16;
output wr_en_reg16;
output [7:0] frame_counter;
output [4:0] address_common;
output [7:0] address_ram;
output [10:0] address_rom;
output in_sel_mul, out_sel_cla16;
output [1:0] in_sel_cla48;
output [3:0] in_sel_reg48;
output [2:0] out_sel_reg48;
output wr_en_regx;
output wr_en_reg48;
output out_en_reg48;
output word_end_index;
output word_start_index;
output [2:0] last_frame_counter;
output [5:0] shift_num;
output shift_en;
input [5:0] word_num;
input [5:0] word_index_out;
input fv_ack;
input clk, reset;
input [15:0] in1;
input [47:0] lut_out;
input [15:0] rom_out;
reg regx_load_x2_reg;
reg regx_start_reg;
reg regx_load_adder_reg;
```

```
reg subcomp_en_reg;
reg regx_state;
reg regx_store_reg;
reg db;
reg [1:0] lut_index;
reg [6:0] counter;
reg [1:0] counter_4;
reg counter_4_start;
reg [7:0] frame_num;     /* ----- frame_num = (total number of frame) - 8 ----- */
reg in_sel_mul_reg;      /* ----- select multiplication between (1'b0)xx and (1'b1)xxy ----- */
reg out_sel_cla16_reg;   /* ----- select the output of (1'b0)x-xmean or (1'b1)variance ----- */
reg [3:0] in_sel_reg48_reg;      /* ----- select the registers in the reg48 to store input ----- */
reg [3:0] in_sel_reg48_reg_tmp;      /* ----- use to generate the in_sel_reg48_reg from
out_sel_reg48_reg ----- */
reg [2:0] out_sel_reg48_reg;   /* ----- select the output from the registers in the reg48 ----- */
reg wr_en_regx_reg;      /* ----- write enable of the reg48_reg, active high ----- */
reg wr_en_reg48_reg;
reg [1:0] in_sel_cla48_reg;      /* ----- select the input between the multiplier output and the
reg16 output ----- */
reg out_en_reg48_reg;   /* ----- set the output from (1'b1) the registers inside or (1'b0) the
input   ----- */
reg [7:0] frame_counter_reg; /* ----- frame number counter (counting up to total frame number
- 8) ----- */
reg [2:0] last_frame_counter_reg; /* ----- counting the last 8 frames ----- */
reg word_end_index_reg;      /* ----- indicate the end of the calculation of one word ----- */
reg word_start_index_reg;      /* ----- indicate the start of the calculation of one word ----- */
reg fv_ack_reg;    /* ----- set to high when fv_ack_reg == 1 ----- */
reg [1:0] addr_reg16_reg;      /* ----- set the address of the registers in the reg16 ----- */
reg wr_en_reg16_reg;    /* ----- write enable of the reg16, active high ----- */
reg [4:0] addr_32bit;      /* ----- first five bits of output address, counting from 0 to 25 ----- */
reg [7:0] frame_num_addr;      /* ----- address select the feature vectors at appropriate frame
time ----- */
reg [2:0] state_counter_reg;      /* ----- pointing at the state now being calculating ----- */
reg [2:0] state_counter_reg1;
reg [5:0] word_addr;      /* ----- pointing at the word now being calculating ----- */
reg [5:0] word_addr1;
reg [4:0] tm_gc;   /* ----- Transition Matrix and Gaussian Constant ----- */
reg result_ack_reg;      /* ----- set to high when the search process finished ----- */
reg final_comp_index_reg;
reg read_const;
reg [5:0] shift_num;
reg shift_set;
reg shift_en;
assign final_comp_index = final_comp_index_reg;
assign result_ack = result_ack_reg;
assign addr_reg16 = addr_reg16_reg;
assign wr_en_reg16 = wr_en_reg16_reg;
assign word_start_index = word_start_index_reg;
assign last_frame_counter = last_frame_counter_reg;
assign word_end_index = word_end_index_reg;
assign frame_counter = frame_counter_reg;
assign in_sel_mul = in_sel_mul_reg;
assign out_sel_cla16 = out_sel_cla16_reg;
```

```
assign in_sel_reg48 = in_sel_reg48_reg;
assign out_sel_reg48 = out_sel_reg48_reg;
assign wr_en_regx = wr_en_regx_reg;
assign in_sel_cla48 = in_sel_cla48_reg;
assign out_en_reg48 = out_en_reg48_reg;
assign counter_reg48 = counter;
assign wr_en_reg48 = wr_en_reg48_reg;
assign regx_load_x2 = regx_load_x2_reg;
assign regx_load_adder = regx_load_adder_reg;
assign regx_start = regx_start_reg;
assign subcomp_en = subcomp_en_reg;
assign regx_store = regx_store_reg;
assign address_common = (lut_index==2'b00)? lut_out[4:0] : 5'bz;
assign address_common = (lut_index==2'b01)? lut_out[20:16] : 5'bz;
assign address_common = (lut_index==2'b10)? lut_out[36:32] : 5'bz;
assign address_common = (lut_index==2'b11)? addr_32bit : 5'bz;
assign address_ram = {frame_num_addr};
assign address_rom = ({result_ack, lut_index, read_const}==4'b1_11_0) ? {4'b0000,
word_index_out} : 11'bz;
assign address_rom = ({result_ack, lut_index, read_const}==4'b0_00_0) ? lut_out[15:5] : 11'bz;
assign address_rom = ({result_ack, lut_index, read_const}==4'b0_01_0) ? lut_out[31:21] :
11'bz;
assign address_rom = ({result_ack, lut_index, read_const}==4'b0_10_0) ? lut_out[47:37] :
11'bz;
assign address_rom = ({result_ack, lut_index, read_const}==4'b0_11_0) ? {db, counter[1],
word_addr, state_counter_reg} : 11'bz;
assign address_rom = ({result_ack, lut_index, read_const}==4'b0_11_1) ? {db, counter[1],
word_addr1, state_counter_reg1} : 11'bz;
/* counter[1] == 1 is mean, counter[1] == 0 is variance */
always @(posedge clk or negedge reset)
begin
        if (reset == 0)
        begin
                counter <= 7'b0000010;
                counter_4 <= 0;
                counter_4_start <= 0;
                in_sel_mul_reg <= 0;
                out_sel_cla16_reg <= 0;
                in_sel_reg48_reg <= 0;
                out_sel_reg48_reg <= 0;
                wr_en_regx_reg <= 0;
                in_sel_cla48_reg <= 0;
                out_en_reg48_reg <= 0;
                frame_counter_reg <= 0;
                in_sel_reg48_reg_tmp <= 0;
                last_frame_counter_reg <= 0;
                word_end_index_reg <= 0;
                word_start_index_reg <= 0;
                fv_ack_reg <= 0;
                addr_reg16_reg <= 0;
                wr_en_reg16_reg <= 0;
                addr_32bit <= 5'b11111;
                frame_num_addr <= 0;
```

```
        word_addr <= 0;
        word_addr1 <= 6'b111111;
        state_counter_reg <= 0;
        state_counter_reg1 <= 0;
        tm_gc <= 0;
        result_ack_reg <= 0;
        wr_en_reg48_reg <= 0;
        regx_load_x2_reg <= 0;
        regx_load_adder_reg <= 0;
        regx_start_reg <= 0;
        subcomp_en_reg <= 0;
        regx_state <= 0;
        regx_store_reg <= 0;
        final_comp_index_reg <= 0;
        db <= 1;
        lut_index <= 2'b11;
        frame_num <= 0;
        read_const <= 0;
        shift_set <= 0;
        shift_num <= 0;
        shift_en <= 0;
end
else
begin
        if(shift_set == 0)
        begin
                counter <= 7'd1;
                db <= 1;
                word_addr <= 0;
                state_counter_reg <= 0;
                addr_32bit <= 5'b11111;
                if(counter == 7'd1)
                begin
                        shift_num <= rom_out[5:0];
                        shift_set <= 1;
                end
        end
        else
        begin
        if (word_num == word_addr && counter == 7'd36)
        begin
                result_ack_reg <= 1;
                fv_ack_reg <= 0;
        end
        if (fv_ack_reg == 1)
        begin
                if (regx_state == 1 && counter == 7'd111)
                        shift_en <= 1;
                else
                        shift_en <= 0;
                if (counter == 7'd11 && regx_state == 0)
                        lut_index <= 2'b00;
                else
```

```
                if (counter == 7'd15 && regx_state == 0)
                        lut_index <= 2'b01;
                else
                if (counter == 7'd19 && regx_state == 0)
                        lut_index <= 2'b10;
                else
                        lut_index <= 2'b11;
        /* ----- start address calculation ----- */
                /* ----- change the address to latch transition matrix and gaussian constant
----- */
                if (counter == 7'd23)
                begin
                        addr_32bit <= 5'b11101;
                        tm_gc <= addr_32bit;
                        read_const <= 1;
                end
                else
                if (counter == 7'd27)
                begin
                        addr_32bit <= 5'b11110;
                        tm_gc <= addr_32bit;
                        read_const <= 1;
                end
                else
                if (counter == 7'd31)
                begin
                        addr_32bit <= 5'b11111;
                        tm_gc <= addr_32bit;
                        read_const <= 1;
                end
                else
                if (counter == 7'd35)
                begin
                        addr_32bit <= 5'b11010;
                        tm_gc <= addr_32bit;
                end
                else
                if (counter == 7'd39)
                begin
                        addr_32bit <= 5'b11011;
                        tm_gc <= addr_32bit;
                end
                else
                if (counter == 7'd43)
                begin
                        addr_32bit <= 5'b11100;
                        tm_gc <= addr_32bit;
                end
                else
                if (counter == 7'd36 || counter == 7'd40 || counter == 7'd44 || counter ==
7'd24
                        || counter == 7'd28 || counter == 7'd32)
                begin
```

```
                addr_32bit <= tm_gc;
                read_const <= 0;
            end

            if (counter_4 == 2'b00)
            begin
                /* ----- the first 5 address bits counting from 0 to 25 (26 feature
vectors) ----- */
                if (addr_32bit == 5'd25)
                begin addr_32bit <= 0;
                    db <= ~db;
                end
                else
                    addr_32bit <= addr_32bit + 1;
                /* ----- pointing at the state now being calculating ----- */
                if (addr_32bit == 5'd25 && regx_state == 1)
                    if (state_counter_reg == frame_counter_reg)
                    begin
                        state_counter_reg <= 0;
                        state_counter_reg1 <= state_counter_reg;
                    end
                    else
                    begin
                        state_counter_reg <= state_counter_reg + 1;
                        state_counter_reg1 <= state_counter_reg;
                    end
                /* ----- calculating the address of the last eight frame time ----- */
                if (addr_32bit == 5'd25 && regx_state == 1)
                begin
                    if (last_frame_counter_reg == 3'd1 && state_counter_reg ==
3'b110)
                    begin
                            state_counter_reg <= 0;
                            state_counter_reg1 <= state_counter_reg;
                            frame_num_addr <= frame_num_addr + 1;
                    end
                    else
                    if (last_frame_counter_reg == 3'd2 && state_counter_reg ==
3'b101)
                    begin
                            state_counter_reg <= 0;
                            state_counter_reg1 <= state_counter_reg;
                            frame_num_addr <= frame_num_addr + 1;
                    end
                    else
                    if (last_frame_counter_reg == 3'd3 && state_counter_reg ==
3'b100)
                    begin
                            state_counter_reg <= 0;
                            state_counter_reg1 <= state_counter_reg;
                            frame_num_addr <= frame_num_addr + 1;
                    end
                    else
```

```
                        if (last_frame_counter_reg == 3'd4 && state_counter_reg ==
3'b011)
                        begin
                                state_counter_reg <= 0;
                                state_counter_reg1 <= state_counter_reg;
                                frame_num_addr <= frame_num_addr + 1;
                        end
                        else
                        if (last_frame_counter_reg == 3'd5 && state_counter_reg ==
3'b010)
                        begin
                                state_counter_reg <= 0;
                                state_counter_reg1 <= state_counter_reg;
                                frame_num_addr <= frame_num_addr + 1;
                        end
                        else
                        if (last_frame_counter_reg == 3'd6 && state_counter_reg ==
3'b001)
                        begin
                                state_counter_reg <= 0;
                                state_counter_reg1 <= state_counter_reg;
                                frame_num_addr <= frame_num_addr + 1;
                        end
                        else
                        if (last_frame_counter_reg == 3'd7 && state_counter_reg ==
3'b000)
                        begin
                                state_counter_reg <= 0;
                                state_counter_reg1 <= state_counter_reg;
                                frame_num_addr <= 0;
                                word_addr <= word_addr + 1;
                        end
                        else
                        if (state_counter_reg == frame_counter_reg || state_counter_reg
== 3'b111)
                                frame_num_addr <= frame_num_addr + 1;
                end
        end
        if (frame_counter_reg == 8'd0 && regx_state == 0 && counter == 7'd36)
                word_addr1 <= word_addr1+1;
end
/* ----- end address calculation ----- */
if (counter == 7'd109 && regx_state == 1'b1)
        regx_start_reg <= 1'b1;
else
        regx_start_reg <= 1'b0;
if ((counter == 7'd17 || counter == 7'd21 || counter == 7'd33)
                && regx_state == 1'b0)
begin
        if   (frame_counter_reg == 0 && word_addr ==0)
                regx_store_reg <= 1'b0;
        else
                regx_store_reg <= 1'b1;
```

```
end
else
        regx_store_reg <= 1'b0;
if (counter == 7'd106)
        in_sel_cla48_reg <= 2'b11;
else
if (counter == 7'd18 && regx_state == 1'b0)
        in_sel_cla48_reg <= 2'b10;
else
if (counter == 7'd22 && regx_state == 1'b0)
        in_sel_cla48_reg <= 2'b11;
else
if (counter == 7'd34 && regx_state == 1'b0 )
        in_sel_cla48_reg <= 2'b11;
else
        in_sel_cla48_reg <= 2'b00;
if ((counter == 7'd19 || counter == 7'd23 || counter == 7'd35)
                && regx_state == 1'b0)
begin
        if(frame_counter_reg == 0 && word_addr == 0)
                regx_load_x2_reg <= 1'b0;
        else
                regx_load_x2_reg <= 1'b1;
end
else
        regx_load_x2_reg <= 1'b0;
if ((counter == 7'd23 || counter == 7'd35)
                && regx_state == 1'b0 && frame_counter_reg != 0)
        wr_en_reg48_reg <= 1'b1;
else
        wr_en_reg48_reg <= 1'b0;
if (counter == 7'd112)
        regx_state <= ~regx_state;
if (regx_state == 1 && counter == 7'd110)
        subcomp_en_reg <= 1;
else
        subcomp_en_reg <= 0;
if (regx_state == 0)
begin
        if (counter == 7'd109)
                regx_load_adder_reg <= 1;
        else
                regx_load_adder_reg <= 0;
end
if (counter == 7'd35 && frame_counter_reg == 0 && regx_state == 0)
        final_comp_index_reg <= 1;
else
        final_comp_index_reg <= 0;
if (counter_4_start == 1)
begin
        if (regx_state == 1)
                if (counter == 7'd103)
                        out_en_reg48_reg <= 1'b1;
```

```
                    else
                        out_en_reg48_reg <= 1'b0;
                if (counter_4 == 2'b00 || counter == 7'd107 ||
                    (last_frame_counter_reg == 3'd7 && counter == 7'd114))
                    wr_en_regx_reg <= 1;
                else
                    wr_en_regx_reg <= 0;
            end
            /* ----- output the appropriate transition matrix and gaussian constant ----- */
            /* ----- latching the transition matrix and gaussian constant into the reg16 module
-----*/
            if ((counter == 7'd12 && regx_state == 0)|| (counter == 7'd16 && regx_state ==
0)||
                (counter == 7'd20 && regx_state == 0)|| counter == 7'd24
                || counter == 7'd28 || counter == 7'd32 || counter == 7'd36 || counter == 7'd40
|| counter == 7'd44)
                wr_en_reg16_reg <= 1;
            else
                wr_en_reg16_reg <= 0;
            /* ----- setting the address of the reg16 module ----- */
            if (counter == 7'd12 || counter == 7'd32 || counter == 7'd44)
                addr_reg16_reg <= 2'b00;
            else
            if (counter == 7'd16 || counter == 7'd28 || counter == 7'd40)
                addr_reg16_reg <= 2'b01;
            else
            if (counter == 7'd20 || counter == 7'd24 || counter == 7'd36)
                addr_reg16_reg <= 2'b10;
            /* ----- setting the fv_ack_reg to high when fv_ack == 1 ----- */
            if (fv_ack == 1)
            begin
                counter <= 7'd1;
                counter_4 <= 0;
                counter_4_start <= 0;
                in_sel_mul_reg <= 0;
                out_sel_cla16_reg <= 0;
                in_sel_reg48_reg <= 0;
                out_sel_reg48_reg <= 0;
                wr_en_regx_reg <= 0;
                in_sel_cla48_reg <= 0;
                out_en_reg48_reg <= 0;
                frame_counter_reg <= 0;
                in_sel_reg48_reg_tmp <= 0;
                last_frame_counter_reg <= 0;
                word_end_index_reg <= 0;
                word_start_index_reg <= 0;
                fv_ack_reg <= 1;
                addr_reg16_reg <= 0;
                wr_en_reg16_reg <= 0;
                addr_32bit <= 5'b11111;
                frame_num_addr <= 0;
                word_addr <= 0;
                word_addr1 <= 6'b111111;
```

```
                state_counter_reg <= 0;
                state_counter_reg1 <= 0;
                tm_gc <= 0;
                result_ack_reg <= 0;
                wr_en_reg48_reg <= 0;
                regx_load_x2_reg <= 0;
                regx_load_adder_reg <= 0;
                regx_start_reg <= 0;
                subcomp_en_reg <= 0;
                regx_state <= 0;
                regx_store_reg <= 0;
                final_comp_index_reg <= 0;
                db <= 0;
                lut_index <= 2'b11;
                read_const <= 0;
            end
            if (counter == 2)
                frame_num <= in1[7:0];
            /* ----- generation of all the counters to be used ----- */
            if (fv_ack_reg == 1 && fv_ack != 1)
            begin
                if (word_end_index == 1'b1)
                    frame_counter_reg <= 8'b11111111;
                else
                if (counter == 7'd114 && out_sel_reg48_reg == 0 && regx_state == 0)
                    frame_counter_reg <= frame_counter_reg + 1;
                if (counter == 7'd114)
                    counter <= 7'd11;
                else
                    counter <= counter + 1;

                counter_4 <= counter_4 + 1;

                if (counter == 7'd7)
                    counter_4_start <= 1;
            end
            word_start_index_reg <= word_end_index_reg;
            /* ----- reset the parameter for calculating the new word ----- */
            if (counter == 7'd112 && frame_counter_reg == frame_num &&
out_sel_reg48_reg == 0 &&
                regx_state == 1'b1)
            begin
                word_end_index_reg <= 1;
                last_frame_counter_reg <= 0;
            end
            else
                word_end_index_reg <= 0;
            /* ----- select the multiplication of xx or xxy ----- */
            in_sel_mul_reg <= counter[1];
            /* ----- out_sel_cla16_reg == 1 means the output is variance,
                else out_sel_cla16_reg == 0 means the output is the
                subtracted result ----- */
            out_sel_cla16_reg <= ~counter[1];
```

```
/* ----- in_sel_cla48_reg == 1 means the addition between the Gaussian constant
         and transition matrix elements output of the reg48 module, else
         in_sel_cla48_reg == 0 means the addition between the output of the reg48
         module and the output of the multiplier ----- */
if (regx_state == 1)
if (counter == 7'd111)
begin
      /* ----- select the output of the reg48 module at the last eight frame time -----
*/
      if (last_frame_counter_reg == 3'd0 && out_sel_reg48_reg == 3'b111 &&
          frame_counter_reg == frame_num)
      begin
            last_frame_counter_reg <= 3'd1;
            out_sel_reg48_reg <= 3'b001;
      end
      else
      if (last_frame_counter_reg == 3'd1 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd2;
            out_sel_reg48_reg <= 3'b010;
      end
      else
      if (last_frame_counter_reg == 3'd2 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd3;
            out_sel_reg48_reg <= 3'b011;
      end
      else
      if (last_frame_counter_reg == 3'd3 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd4;
            out_sel_reg48_reg <= 3'b100;
      end
      else
      if (last_frame_counter_reg == 3'd4 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd5;
            out_sel_reg48_reg <= 3'b101;
      end
      else
      if (last_frame_counter_reg == 3'd5 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd6;
            out_sel_reg48_reg <= 3'b110;
      end
      else
      if (last_frame_counter_reg == 3'd6 && out_sel_reg48_reg == 3'b111)
      begin
            last_frame_counter_reg <= 3'd7;
            out_sel_reg48_reg <= 3'b111;
      end
      else
      /* ----- select the output of the reg48 module at the first eight frame time
```

```
----- */
                    case ({frame_counter_reg, out_sel_reg48_reg})
                        11'b00000000_000 : out_sel_reg48_reg <= 0;
                        11'b00000001_001 : out_sel_reg48_reg <= 0;
                        11'b00000010_010 : out_sel_reg48_reg <= 0;
                        11'b00000011_011 : out_sel_reg48_reg <= 0;
                        11'b00000100_100 : out_sel_reg48_reg <= 0;
                        11'b00000101_101 : out_sel_reg48_reg <= 0;
                        11'b00000110_110 : out_sel_reg48_reg <= 0;
                        default : out_sel_reg48_reg <= out_sel_reg48_reg + 1;
                    endcase
                end
                /* ----- in_sel_reg_reg48[3] is used to control the write operation
                    on the odd or even entry of the reg_48 ----- */
                if (regx_state == 0)
                if (counter == 7'd35)
                    in_sel_reg48_reg[3] <= 1'b1;
                else
                    in_sel_reg48_reg[3] <= 1'b0;
                /* ----- because in_sel_reg48_reg and out_sel_reg48_reg has 2 bit difference -----
*/
                if (counter == 7'd33)
                    in_sel_reg48_reg <= out_sel_reg48_reg;
                end
        end
end
endmodule
```

## ✧ Top

```
module int_signal_dm(int_signal_out, mulout, sumout_cla48, reg48_out1,
                    reg16_out, regx_out, subcomp_out, shift_out, lut_out,
                    mod_sel, bit_sel, clk, reset);
input [47:0] mulout, sumout_cla48, reg48_out1, reg16_out;
input [47:0] regx_out, subcomp_out, shift_out, lut_out;
output [7:0] int_signal_out;
input reset, clk;
input [2:0] mod_sel;
input [2:0] bit_sel;
reg [7:0] int_signal_out_reg;
assign int_signal_out = int_signal_out_reg;
always @(posedge clk or negedge reset)
begin
    if (reset == 0)
    begin
        int_signal_out_reg <= 0;
    end
    else
    begin
        if (mod_sel == 3'b000)
        begin
```

```
        case (bit_sel)
                3'd0 : int_signal_out_reg <= mulout[7:0];
                3'd1 : int_signal_out_reg <= mulout[15:8];
                3'd2 : int_signal_out_reg <= mulout[23:16];
                3'd3 : int_signal_out_reg <= mulout[31:24];
                3'd4 : int_signal_out_reg <= mulout[39:32];
                3'd5 : int_signal_out_reg <= mulout[47:40];
                default;
        endcase
end
else
if (mod_sel == 3'b001)
begin
        case (bit_sel)
                3'd0 : int_signal_out_reg <= sumout_cla48[7:0];
                3'd1 : int_signal_out_reg <= sumout_cla48[15:8];
                3'd2 : int_signal_out_reg <= sumout_cla48[23:16];
                3'd3 : int_signal_out_reg <= sumout_cla48[31:24];
                3'd4 : int_signal_out_reg <= sumout_cla48[39:32];
                3'd5 : int_signal_out_reg <= sumout_cla48[47:40];
                default;
        endcase
end
else
if (mod_sel == 3'b010)
begin
        case (bit_sel)
                3'd0 : int_signal_out_reg <= reg48_out1[7:0];
                3'd1 : int_signal_out_reg <= reg48_out1[15:8];
                3'd2 : int_signal_out_reg <= reg48_out1[23:16];
                3'd3 : int_signal_out_reg <= reg48_out1[31:24];
                3'd4 : int_signal_out_reg <= reg48_out1[39:32];
                3'd5 : int_signal_out_reg <= reg48_out1[47:40];
                default;
        endcase
end
else
if (mod_sel == 3'b011)
begin
        case (bit_sel)
                3'd0 : int_signal_out_reg <= reg16_out[7:0];
                3'd1 : int_signal_out_reg <= reg16_out[15:8];
                3'd2 : int_signal_out_reg <= reg16_out[23:16];
                3'd3 : int_signal_out_reg <= reg16_out[31:24];
                3'd4 : int_signal_out_reg <= reg16_out[39:32];
                3'd5 : int_signal_out_reg <= reg16_out[47:40];
                default;
        endcase
end
else
if (mod_sel == 3'b100)
begin
        case (bit_sel)
```

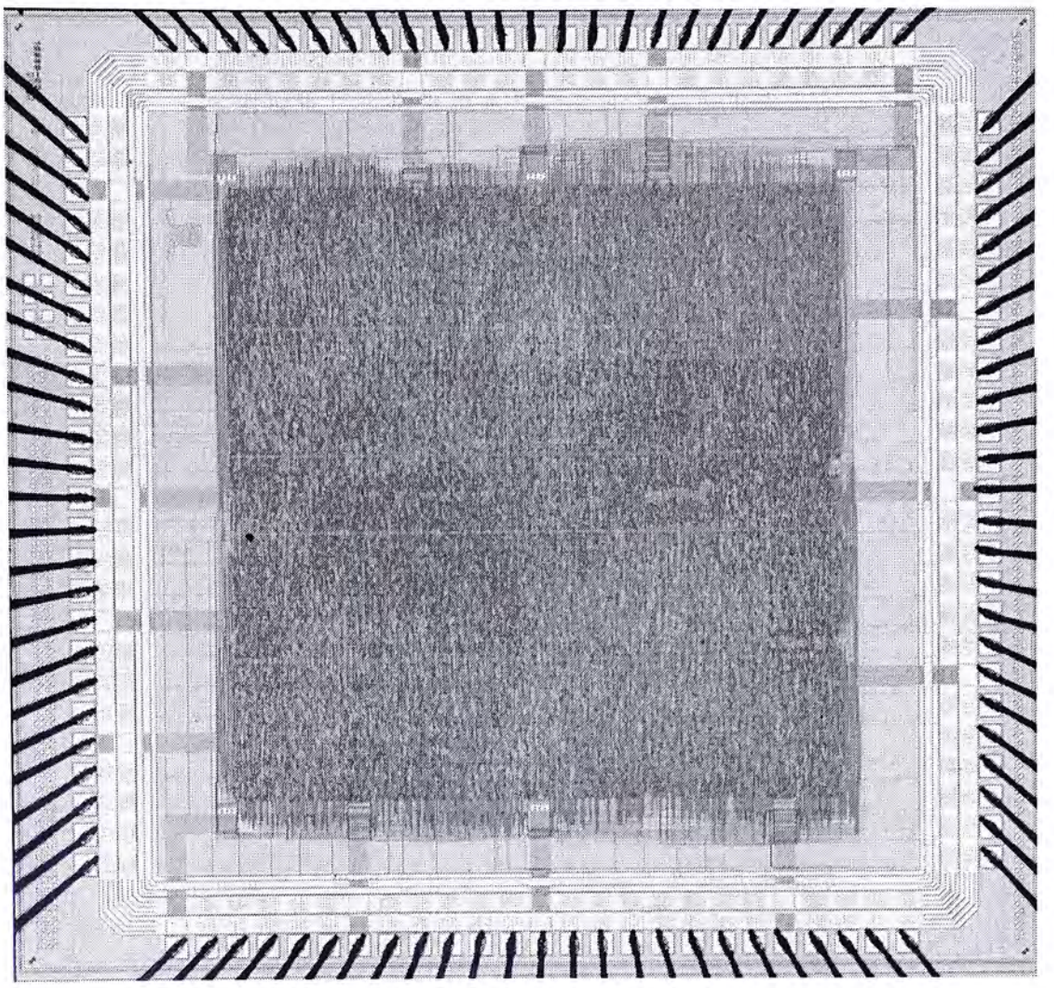
```
                        3'd0 : int_signal_out_reg <= regx_out[7:0];
                        3'd1 : int_signal_out_reg <= regx_out[15:8];
                        3'd2 : int_signal_out_reg <= regx_out[23:16];
                        3'd3 : int_signal_out_reg <= regx_out[31:24];
                        3'd4 : int_signal_out_reg <= regx_out[39:32];
                        3'd5 : int_signal_out_reg <= regx_out[47:40];
                        default;
                endcase
        end
        else
        if (mod_sel == 3'b101)
        begin
                case (bit_sel)
                        3'd0 : int_signal_out_reg <= subcomp_out[7:0];
                        3'd1 : int_signal_out_reg <= subcomp_out[15:8];
                        3'd2 : int_signal_out_reg <= subcomp_out[23:16];
                        3'd3 : int_signal_out_reg <= subcomp_out[31:24];
                        3'd4 : int_signal_out_reg <= subcomp_out[39:32];
                        3'd5 : int_signal_out_reg <= subcomp_out[47:40];
                        default;
                endcase
        end
        else
        if (mod_sel == 3'b110)
        begin
                case (bit_sel)
                        3'd0 : int_signal_out_reg <= shift_out[7:0];
                        3'd1 : int_signal_out_reg <= shift_out[15:8];
                        3'd2 : int_signal_out_reg <= shift_out[23:16];
                        3'd3 : int_signal_out_reg <= shift_out[31:24];
                        3'd4 : int_signal_out_reg <= shift_out[39:32];
                        3'd5 : int_signal_out_reg <= shift_out[47:40];
                        default;
                endcase
        end
        else
        if (mod_sel == 3'b111)
        begin
                case (bit_sel)
                        3'd0 : int_signal_out_reg <= lut_out[7:0];
                        3'd1 : int_signal_out_reg <= lut_out[15:8];
                        3'd2 : int_signal_out_reg <= lut_out[23:16];
                        3'd3 : int_signal_out_reg <= lut_out[31:24];
                        3'd4 : int_signal_out_reg <= lut_out[39:32];
                        3'd5 : int_signal_out_reg <= lut_out[47:40];
                        default;
                endcase
        end

    end
end
endmodule
```

```
module top_core_dm(int_signal_out, result_ack, address_common, address_ram, address_rom,
overf,
                in1, in2, word_num, mod_sel, bit_sel,
                fv_ack, clk, reset);
output overf;       /* ----- overflow output ----- */
output [7:0] int_signal_out;     /* ----- output of the internal signal ----- */
output [7:0] address_ram;        /* ----- ram address for latching the feature vector ------ */
output [10:0] address_rom;       /* ----- rom address for latching the mean, variance, gaussian
constant,
                                 transition matrix. the recognized word will also be shown at the
                                 rom address bus ----- */
output [4:0] address_common; /* ----- common address for first five bits of address_ram and
address_rom ----- */
output result_ack; /* ----- set to high when the search process finished ----- */
input [5:0] word_num;   /* ----- number of words stored in rom ----- */
input fv_ack;           /* ----- indicate the start of searching ----- */
input [15:0] in1, in2;      /* ----- in1 is the data from ram, in2 is the data from rom ----- */
                    /* ----- address 8'd31 of ram stores frame_num and will be latched in from
in1 ----- */
input clk, reset;
input [2:0] mod_sel;        /* ----- select the internal output of the four signals between mulout,
                              sumout_cla48, reg48_out1, data_fscore ----- */
input [2:0] bit_sel;/* ----- select the 8 bit outputs within the internal signal ---- */
wire overf_cla16;  /* ----- overflow flag of 16bit carry look ahead adder ----- */
wire overf_cla48;  /* ----- overflow flag of 48bit carry look ahead adder ----- */
wire [5:0] word_index_out;
wire [1:0] addr_reg16;
wire wr_en_reg16;
wire word_start_index;
wire [47:0] reg16_out;
wire [1:0] in_sel_cla48;
wire out_sel_cla16;
wire in_sel_mul;
wire wr_en_reg48;
wire wr_en_regx;
wire [3:0] in_sel_reg48;
wire [2:0] out_sel_reg48;
wire [15:0] sumout_cla16;
wire [47:0] mulout, sumout_cla48, reg48_out1;
wire out_en_reg48;
wire [7:0] frame_counter;
wire word_end_index;
wire [2:0] last_frame_counter;
wire [6:0] counter_reg48;
wire [47:0] lut_out;
wire [47:0] subcomp_out;
wire [47:0] regx_out1;
wire [47:0] regx_out2;
wire [47:0] regx_out;
wire [47:0] lut_in;
wire [5:0] shift_num;
wire overf_shift;
wire shift_en;
```

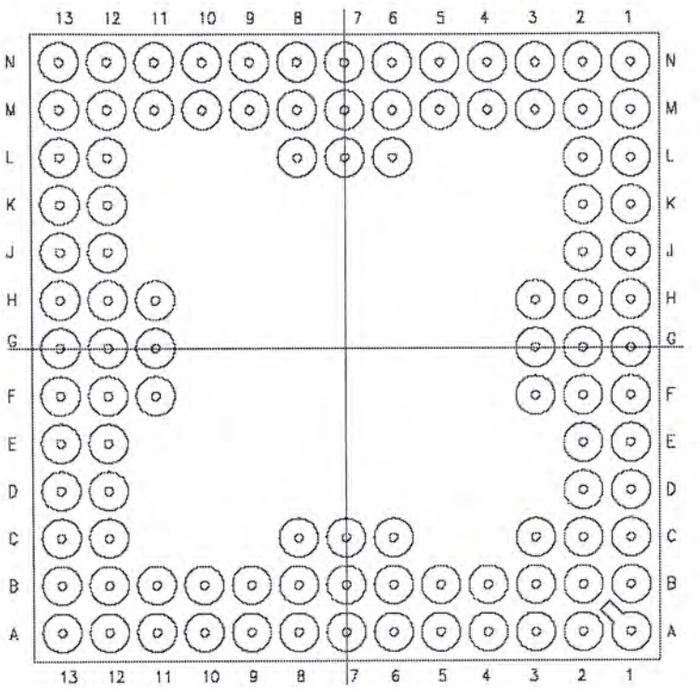
```
reg fv_ack_d;
reg [15:0] in1_d, in2_d;
always @(posedge clk or negedge reset)
begin
if (reset == 0)
begin
        fv_ack_d <= 0;
        in1_d <= 0;
        in2_d <= 0;
end
else
begin
        fv_ack_d <= fv_ack;
        in1_d <= in1;
        in2_d <= in2;
end
end
assign overf = overf_cla16 | overf_cla48;
int_signal_dm int_signal_dm
                (.int_signal_out(int_signal_out), .mulout(mulout), .sumout_cla48(sumout_cla48), .
                reg48_out1(reg48_out1),
                .reg16_out(reg16_out), .regx_out(regx_out), .subcomp_out(subcomp_out),
                .shift_out(lut_in), .lut_out(lut_out),
                .mod_sel(mod_sel), .bit_sel(bit_sel), .clk(clk), .reset(reset));
cla16_dm cla16_dm
                (.sumout(sumout_cla16), .overflow(overf_cla16),
                .result_ack(result_ack), .in1(in1_d), .in2(in2_d),
                .out_sel(out_sel_cla16), .clk(clk), .reset(reset));
multiplier_dm multiplier_dm
                (.mulout(mulout), .in_sel(in_sel_mul),
                .in1(sumout_cla16), .counter_cla48(counter_reg48[0]), .clk(clk), .reset(reset));
core_cla48_dm core_cla48_dm
                (.sumout(sumout_cla48), .overflow(overf_cla48), .in_sel(in_sel_cla48),
                .in1(mulout), .reg48_in(reg48_out1), .reg16_in(reg16_out),
                .regx_in(regx_out),
                .clk(clk), .reset(reset));
reg48_dm reg48_dm
                (.data_out1(reg48_out1), .data_in(sumout_cla48),
                .in_sel(in_sel_reg48), .out_sel(out_sel_reg48),
                .wr_en(wr_en_reg48), .out_en_reg48(out_en_reg48),
                .frame_counter(frame_counter), .word_end_index(word_end_index),
                .word_start_index(word_start_index), .last_frame_counter(last_frame_counter),
                .result_ack(result_ack),
                .counter_reg48(counter_reg48), .fv_ack(fv_ack_d),
                .clk(clk), .reset(reset));
control_dm control_dm
                (.final_comp_index(final_comp_index), .regx_load_adder(regx_load_adder),
                .regx_load_x2(regx_load_x2), .regx_start(regx_start),
                .subcomp_en(subcomp_en), .counter_reg48(counter_reg48), .regx_store(regx_stor
                e),
                .lut_out(lut_out), .rom_out(in2_d),
                .shift_num(shift_num), .shift_en(shift_en),
                .result_ack(result_ack),
```

```
        .addr_reg16(addr_reg16), .wr_en_reg16(wr_en_reg16), .frame_counter(frame_cou
        nter),
        .address_common(address_common), .address_ram(address_ram), .address_rom(a
        ddress_rom),
        .in_sel_mul(in_sel_mul),
        .out_sel_cla16(out_sel_cla16), .in_sel_cla48(in_sel_cla48), .in_sel_reg48(in_sel_re
        g48),
        .out_sel_reg48(out_sel_reg48), .wr_en_regx(wr_en_regx), .wr_en_reg48(wr_en_re
        g48),
        .out_en_reg48(out_en_reg48),
        .fv_ack(fv_ack_d),
        .word_end_index(word_end_index), .word_start_index(word_start_index),
        .last_frame_counter(last_frame_counter),
        .word_index_out(word_index_out),
        .word_num(word_num), .in1(in1_d), .clk(clk), .reset(reset));
        reg16_dm reg16_dm
        (.data_out(reg16_out), .data_in(in2_d), .addr(addr_reg16), .wr_en(wr_en_reg16), .f
        v_ack(fv_ack_d),
        .clk(clk), .reset(reset));
final_score_reg_dm final_score_reg_dm
        (.word_index_out(word_index_out),
        .word_end_index(word_end_index), .final_comp_index(final_comp_index),
        .fv_ack(fv_ack_d), .data_fscore(sumout_cla48), .clk(clk),
        .reset(reset));
regx_dm regx_dm
        (.dout(regx_out), .dout_x1(regx_out1), .dout_x2(regx_out2), .adder_in(sumout_cla
        48),
        .start(regx_start), .load_adder(regx_load_adder), .store(regx_store),
        .load_x2(regx_load_x2), .sel_x2(sel_x2), .wr_en_regx(wr_en_regx), .fv_ack(fv_ac
        k_d),
        .clk(clk), .reset(reset));
subcomp_dm subcomp_dm
        (.out(subcomp_out), .sel_x2(sel_x2), .in1(regx_out1), .in2(regx_out2),
        .en(subcomp_en), .clk(clk), .reset(reset));
lut_dm lut_dm
        (.out(lut_out), .in(lut_in), .shift_overf(overf_shift), .clk(clk), .reset(reset));
shift_dm shift_dm
        (.shift_num(shift_num), .datain(subcomp_out), .dataout(lut_in), .overflow(overf_s
        hift),   .en(shift_en), .clk(clk),
        .reset(reset));
endmodule
```

# Appendix II Chip Microphotograph



# Appendix III Pin Assignment of the Speech Recognition IC




Top View of PGA 100

| Pin Name | Pin Number | IN/OUT | Description |
|---|---|---|---|
| vdd3allp_01 | B 2 | IN | VDD |
| PAD_I_mod_sel_0 | B 1 | IN | select an internal block to check its output : bit 0 |
| PAD_I_mod_sel_1 | C 2 | IN | select an internal block to check its output : bit 1 |
| PAD_I_mod_sel_2 | C 1 | IN | select an internal block to check its output : bit 2 |
| PAD_I_bit_sel_0 | D 2 | IN | select 8 bits of an internal block's output : bit 0 |
| PAD_I_bit_sel_1 | D 13 | IN | select 8 bits of an internal block's output : bit 1 |
| PAD_I_bit_sel_2 | E 2 | IN | select 8 bits of an internal block's output : bit 2 |
| PAD_I_word_num_0 | E 1 | IN | word number of the vocabulary : bit 0 |
| gnd3allp_01 | F 3 | IN | GND |
| PAD_I_word_num_1 | F 2 | IN | word number of the vocabulary : bit 1 |
| PAD_I_word_num_2 | F 1 | IN | word number of the vocabulary : bit 2 |
| PAD_I_word_num_3 | G 2 | IN | word number of the vocabulary : bit 3 |

| | | | |
|---|---|---|---|
| vdd3allp_02 | G 3 | IN | VDD |
| PAD_I_word_num_4 | G 1 | IN | word number of the vocabulary : bit 4 |
| PAD_I_word_num_5 | H 1 | IN | word number of the vocabulary : bit 5 |
| PAD_I_fv_ack | H 2 | IN | the Start signal |
| gnd3allp_02 | H 3 | IN | GND |
| PAD_I_in1_0 | J 1 | IN | feature vector input : bit 0 |
| PAD_I_in1_1 | J 2 | IN | feature vector input : bit 1 |
| PAD_I_in1_2 | K 1 | IN | feature vector input : bit 2 |
| PAD_I_in1_3 | K 2 | IN | feature vector input : bit 3 |
| PAD_I_in1_4 | L 1 | IN | feature vector input : bit 4 |
| PAD_I_in1_5 | M 1 | IN | feature vector input : bit 5 |
| vdd3allp_03 | L 2 | IN | VDD |
| PAD_I_in1_6 | N 1 | IN | feature vector input : bit 6 |
| PAD_I_in1_7 | M 2 | IN | feature vector input : bit 7 |
| PAD_I_in1_8 | N 2 | IN | feature vector input : bit 8 |
| gnd3allp_03 | M 3 | IN | GND |
| PAD_I_in1_9 | N 3 | IN | feature vector input : bit 9 |
| PAD_I_in1_10 | M 4 | IN | feature vector input : bit 10 |
| PAD_I_in1_11 | N 4 | IN | feature vector input : bit 11 |
| PAD_I_in1_12 | M 5 | IN | feature vector input : bit 12 |
| PAD_I_in1_13 | N 5 | IN | feature vector input : bit 13 |
| vdd3allp_04 | L 6 | IN | VDD |
| PAD_I_in1_14 | M 6 | IN | feature vector input : bit 14 |
| PAD_I_in1_15 | N 6 | IN | feature vector input : bit 15 |
| PAD_I_in2_0 | M 7 | IN | model parameter input : bit 0 |
| gnd3allp_04 | L 7 | IN | GND |
| PAD_I_in2_1 | N 7 | IN | model parameter input : bit 1 |
| PAD_I_in2_2 | N 8 | IN | model parameter input : bit 2 |
| PAD_I_in2_3 | M 8 | IN | model parameter input : bit 3 |
| vdd3allp_05 | L 8 | IN | VDD |
| PAD_I_in2_4 | N 9 | IN | model parameter input : bit 4 |
| PAD_I_in2_5 | M 9 | IN | model parameter input : bit 5 |
| PAD_I_in2_6 | N 10 | IN | model parameter input : bit 6 |

| | | | |
|---|---|---|---|
| gnd3allp_05 | M 10 | IN | GND |
| PAD_I_in2_7 | N 11 | IN | model parameter input : bit 7 |
| PAD_I_in2_8 | N 12 | IN | model parameter input : bit 8 |
| PAD_I_in2_9 | M 11 | IN | model parameter input : bit 9 |
| PAD_I_in2_10 | N 13 | IN | model parameter input : bit 10 |
| vdd3allp_06 | M 12 | IN | VDD |
| PAD_I_in2_11 | M 13 | IN | model parameter input : bit 11 |
| PAD_I_in2_12 | L 12 | IN | model parameter input : bit 12 |
| PAD_I_in2_13 | L 13 | IN | model parameter input : bit 13 |
| PAD_I_in2_14 | K 12 | IN | model parameter input : bit 14 |
| PAD_I_in2_15 | K 13 | IN | model parameter input : bit 15 |
| gnd3allp_06 | J 12 | IN | GND |
| PAD_I_reset | J 13 | IN | reset signal |
| PAD_O_overf | H 11 | OUT | indicate the chip has been overflow |
| PAD_O_address_ram_0 | H 12 | OUT | address of the external memory storing feature vector : bit 0 |
| PAD_O_address_ram_1 | H 13 | OUT | address of the external memory storing feature vector : bit 1 |
| PAD_O_address_ram_2 | G 12 | OUT | address of the external memory storing feature vector : bit 2 |
| vdd3allp_07 | G 11 | IN | VDD |
| PAD_O_address_ram_3 | G 13 | OUT | address of the external memory storing feature vector : bit 3 |
| PAD_O_address_ram_4 | F 13 | OUT | address of the external memory storing feature vector : bit 4 |
| PAD_O_address_ram_5 | F 12 | OUT | address of the external memory storing feature vector : bit 5 |
| PAD_O_address_ram_6 | F 11 | OUT | address of the external memory storing feature vector : bit 6 |
| PAD_O_address_ram_7 | E 13 | OUT | address of the external memory storing feature vector : bit 7 |
| PAD_O_address_rom_0 | E 12 | OUT | address of the external memory storing model parameter : bit 0 |
| PAD_O_address_rom_1 | D 13 | OUT | address of the external memory storing model parameter : bit 1 |
| gnd3allp_07 | D 12 | IN | GND |
| PAD_O_address_rom_2 | C 13 | OUT | address of the external memory storing model parameter : bit 2 |
| PAD_O_address_rom_3 | B 13 | OUT | address of the external memory storing model parameter : bit 3 |
| PAD_O_address_rom_4 | C 12 | OUT | address of the external memory storing model parameter : bit 4 |

| | | | |
|---|---|---|---|
| PAD_O_address_rom_5 | A 13 | OUT | address of the external memory storing model parameter : bit 5 |
| PAD_O_address_rom_6 | B 12 | OUT | address of the external memory storing model parameter : bit 6 |
| PAD_O_address_rom_7 | A 12 | OUT | address of the external memory storing model parameter : bit 7 |
| PAD_O_address_rom_8 | B 11 | OUT | address of the external memory storing model parameter : bit 8 |
| PAD_O_address_rom_9 | A 11 | OUT | address of the external memory storing model parameter : bit 9 |
| PAD_O_address_rom_10 | B 10 | OUT | address of the external memory storing model parameter : bit 10 |
| PAD_O_address_common_0 | A 10 | OUT | common address of the two external memories : bit 0 |
| vdd3allp_08 | B 9 | IN | VDD |
| PAD_O_address_common_1 | A 9 | OUT | common address of the two external memories : bit 1 |
| PAD_O_address_common_2 | C 8 | OUT | common address of the two external memories : bit 2 |
| PAD_O_address_common_3 | B 8 | OUT | common address of the two external memories : bit 3 |
| PAD_O_address_common_4 | A 8 | OUT | common address of the two external memories : bit 4 |
| PAD_O_int_signal_out_0 | B 7 | OUT | internal block's output : bit 0 |
| gnd3allp_08 | C 7 | IN | GND |
| PAD_O_int_signal_out_1 | A 7 | OUT | internal block's output : bit 1 |
| PAD_O_int_signal_out_2 | A 6 | OUT | internal block's output : bit 2 |
| PAD_O_int_signal_out_3 | B 6 | OUT | internal block's output : bit 3 |
| PAD_O_int_signal_out_4 | C 6 | OUT | internal block's output : bit 4 |
| PAD_O_int_signal_out_5 | A 5 | OUT | internal block's output : bit 5 |
| vdd3allp_09 | B 5 | IN | VDD |
| PAD_O_int_signal_out_6 | A 4 | OUT | internal block's output : bit 6 |
| PAD_O_int_signal_out_7 | B 4 | OUT | internal block's output : bit 7 |
| PAD_O_result_ack | A 3 | OUT | done signal |
| PAD_I_clk_s | A 2 | IN | clk for the double-mixture model |
| PAD_I_clk | B 3 | IN | clk for the single-mixture model |
| PAD_I_dm_swap | A 1 | IN | select the double-mixture model or the single-mixture model |

# Appendix IV The Testing Board of the IC