

Application-specific Instruction Set Processor for Speech Recognition

CHEUNG Man Ting

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

©The Chinese University of Hong Kong
September 2005

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or whole of the materials in the thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.



Acknowledgements

I would like to thank my supervisor, Professor Choy Chiu-Sing. He gives me the vision, guidance and support that contribute to the formation of this thesis. He also provides me with excellent research and study environment in a well organized and established VLSI team. His intelligence, preciseness and patience make his students work conscientiously and always try to strive for excellence in the research. I would like to thank Professor Chan Cheong-Fat and Professor Pun Kong-Pang for their constructive comments on the work. Thanks also to Mr Yeung Wing-Yee, who maintains our laboratory equipment and design tools with great effort. Special thanks are given to Qin Chao for his continuous support and discussion on the speech algorithms and training toolkit.

I am greatly grateful to the graduate students in VLSI laboratory, especially Yu Chun-Pong, Xin Ling, Xu Ke, Han Wei, So Pui-Tak, Chan Chi-Hong, Leung Pak-Keung and Andy Kwok for their kind assistance throughout my research work. Lastly, I highly appreciate my parents for encouraging and supporting me during two-year MPhil study.

Abstract of thesis entitled:
**Application-specific Instruction Set Processor for Speech
Recognition**
Submitted by **CHEUNG MAN TING**
for the degree of **Master of Philosophy**
in **Electronic Engineering**
at **The Chinese University of Hong Kong** in
September 2005.

This research presents an investigation on the implementation of automatic speech recognition (ASR) using Application Specific Instruction Set Processor (ASIP) methodology. The ASIP approach bridges the gap between traditional purely software and purely hardware designs. Combining the optimized hardware datapaths with some application specific instructions, it can speed up the operation with significant improvement.

Our research implements the double-mixture HMM-based isolated word speech recognizer. Specialized instructions have been developed for the computationally intensive calculation of the output probability in the process of Viterbi search algorithm.

The ASIP is fabricated with a $0.35\ \mu$ CMOS technology. The time required to complete one recognition is about 1 second at the working frequency of 5 MHz. This is at least one magnitude faster than a comparable design. Further enhancement can reduce this speed by a half, doubling the performance. The recognition time is fast enough in real-time speech applications. On the other hand, the proposed speech recognition running on ASIP platform attains the accuracy of 93.2 %, which is approximately the same recognition accuracy as the software recognition. It is obvious that our design can meet the stringent requirement of both time-critical and highly accurate speech application.

摘要

這次研究的題目是利用專用指令處理器 (Application Specific Instruction Set Processor, ASIP)來進行語音識別自動化的應用。專用指令處理器在傳統的純軟件和純硬件的設計中起了橋樑作用，它針對某一個應用，產生專用指令，並與經優化後的硬件運行路徑結合，從而大大提高了整體運行的效率。

這次研究是實現雙混合(double-mixture)的隱馬爾可夫模型 (Hidden Markov Model, HMM) 的單字語音識別器。我們開發了專用指令，用來執行運算量巨大的維特比搜尋演算法 (Viterbi search algorithm)，以計算各輸出概率。

專用指令處理器使用 0.35 微米互補性氧化金屬半導體 (Complementary Metal-Oxide Semiconductor, CMOS) 技術製造。此處理器可以在 5 兆的時鐘頻率下以 1 秒時間從單字的字庫中識別一個單字。與其他相近的研究比較，此識別速度是異常迅速。若處理器利用專用指令來執行語音應用，其識別單字的速度將提升一倍，如此迅速的識別時間是可以進行實時 (Real-time) 語音應用的。另一方面，建議中的語音專用指令處理器的識別單字準確度達 93.2%，這與使用相同算法的語音識別軟件一樣精確。由此可見，專用指令處理器能符合高要求的設計，可以在既快速又準確的要求下執行語音應用。

Contents

1	Introduction	1
1.1	The Emergence of ASIP	1
1.1.1	Related Work	3
1.2	Motivation	6
1.3	ASIP Design Methodologies	7
1.4	Fundamentals of Speech Recognition	8
1.5	Thesis outline	10
2	Automatic Speech Recognition	11
2.1	Overview of ASR system	11
2.2	Theory of Front-end Feature Extraction	12
2.3	Theory of HMM-based Speech Recognition	14
2.3.1	Hidden Markov Model (HMM)	14
2.3.2	The Typical Structure of the HMM	14
2.3.3	Discrete HMMs and Continuous HMMs	15
2.3.4	The Three Basic Problems for HMMs	17
2.3.5	Probability Evaluation	18
2.4	The Viterbi Search Engine	19
2.5	Isolated Word Recognition (IWR)	22
3	Design of ASIP Platform	24
3.1	Instruction Fetch	25
3.2	Instruction Decode	26
3.3	Datapath	29

3.4	Register File Systems	30
3.4.1	Memory Hierarchy	30
3.4.2	Register File Organization	31
3.4.3	Special Registers	34
3.4.4	Address Generation	34
3.4.5	Load and Store	36
4	Implementation of Speech Recognition on ASIP	37
4.1	Hardware Architecture Exploration	37
4.1.1	Floating Point and Fixed Point	37
4.1.2	Multiplication and Accumulation	38
4.1.3	Pipelining	41
4.1.4	Memory Architecture	43
4.1.5	Saturation Logic	44
4.1.6	Specialized Addressing Modes	44
4.1.7	Repetitive Operation	47
4.2	Software Algorithm Implementation	49
4.2.1	Implementation Using Base Instruction Set	49
4.2.2	Implementation Using Refined Instruction Set	54
5	Simulation Results	56
6	Conclusions and Future Work	60
	Appendices	62
A	Base Instruction Set	62
B	Special Registers	65
C	Chip Microphotograph of ASIP	67
D	The Testing Board of ASIP	68
	Bibliography	69

List of Tables

1.1	Trade-off Comparison among GPP, ASIP and ASIC	2
3.1	The Architectural Parameters of the ASIP	24
3.2	The Processor Usage of Different Functional Units	28
3.3	Exploiting Data Parallelism by Observing Data Access Pattern .	33
4.1	The Booth Encoding Table	40
4.2	The Pointer Update Algorithm of Circular Addressing Mode . .	46
4.3	The Speech Parameters for Recognition	53
5.1	The Specification of Fabricated Chip	56
5.2	The Simulation Results of Recognition Accuracy	58
A.1	The Data Processing Instructions	62
A.2	The Bit Manipulation Instructions	63
A.3	The Flow Control Instructions	63
A.4	The Boolean Operation Instructions	63
A.5	The Configuration Instructions	64
A.6	The Memory Manipulation Instructions	64
B.1	The Organization of Special Purpose Registers	65

List of Figures

- 1.1 ASIP Fill the Gap Between Performance and Flexibility [1] . . . 2
- 1.2 Overview of ASIP Design Flow 7
- 1.3 The Short-time Stationary Characteristic of Speech Signal . . . 10
- 2.1 Automatic Speech Recognition System 11
- 2.2 The Algorithm for Front-end MFCC Computation 12
- 2.3 The Simplified View of Hidden Markov Model 14
- 2.4 The HMM with Single-mixture Distribution 16
- 2.5 The HMM with Double-mixture Distribution 16
- 2.6 The Lattice Structure of Simplified Viterbi Search 21
- 2.7 The Training Process for Generating HMM Reference Models . 23
- 2.8 Using HMMs for Isolated Word Recognition 23
- 3.1 The Organization of the ASIP Architecture 25
- 3.2 The Structure of Instruction Fetch Unit 26
- 3.3 The Structure of Instruction Decoder 27
- 3.4 The Structure of the Base Datapath 30
- 3.5 The Structure of the Memory Hierarchy 31
- 3.6 The Structure of Register File for Data Parallelism 33
- 3.7 The Large-Scale View of Register File for Data Parallelism . . . 34
- 3.8 The Datapath of Address Generation Unit 35
- 4.1 The Datapath of MAC unit 38
- 4.2 The Encoding of Booth Using Multiplier 39
- 4.3 The Booth Encoding Logic 40
- 4.4 The Diagram of 4:2 Compressor 40

4.5	The CSA with 4-bit CLA	41
4.6	The Whole Process of Fast Multiplication	42
4.7	The Pipeline Organization of the Platform	43
4.8	The Datapath of Saturation Logic	45
4.9	The Output of FFT Algorithm	46
4.10	The Structure of Loop Controller	48
4.11	The Content of Stack	48
4.12	Examples of Instruction Condensation and Distillation	54
4.13	Examples of Instruction Condensation and Distillation	55
5.1	The Program Skeleton of Viterbi Search	57
5.2	The Simplified Diagram of PCB	57

Chapter 1

Introduction

1.1 The Emergence of ASIP

Application Specific Integrated Circuits (ASICs) was once the best solution for different applications. An ASIC implementation has fully customized datapaths and control logic. Though its performance can be optimized in terms of size, speed and power consumption, ASIC is not flexible enough since its focus is mainly on hardware part only. Any design errors found in the chip will lead to additional manufacturing delays and costs.

On the contrary, Programmable implementations allow high degree of reusability by reprogramming the devices to perform various tasks. The key benefits of relying on software design part are lower development costs, shorter time-to-market cycles and easier adaptation to the modification of market requirements. However, it is known that the programmable devices like general-purpose processors reach limitations in running critical missions of real-time applications. Compared with ASICs, general-purpose processors dissipate more power and show lower performance. To balance the trade-off among performance, flexibility and other design constraints as shown in Table 1.1, this leads to the development of Application Specific Instruction Set Processors (ASIP).

ASIP is a programmable device whose architecture and instruction set are optimized for a specific application area. It speeds up the application by shortening the crucial path in the way of tuning the instruction set and introducing

	GPP	ASIP	ASIC
Performance	Low	High	Very High
Flexibility	Excellent	Good	Poor
HW Design Effort	Nil	Large	Very Large
SW Design Effort	Small	Large	Nil
Power	Large	Medium	Small
Engineering Cost	Low	Medium	High
Time-to-market	Short	Medium	Long
Market Volume	Low	Medium	High

Table 1.1: Trade-off Comparison among GPP, ASIP and ASIC

special hardware accelerators. It turns out that defining an optimal instruction set and formulating the composition of hardware functions and software functions are the key tasks in ASIP design. For this reason, ASIP are currently developed to bridge the gap between the performance and flexibility of pure hardware and pure software solutions as shown in Figure 1.1.

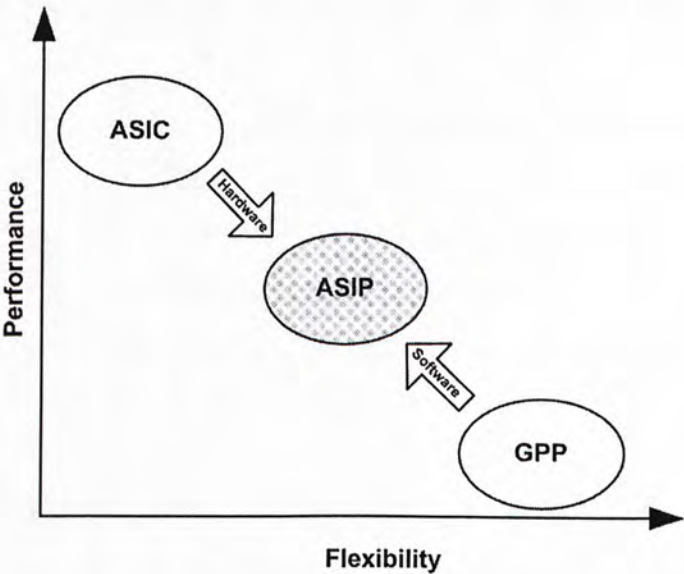


Figure 1.1: ASIP Fill the Gap Between Performance and Flexibility [1]

1.1.1 Related Work

Currently existing ASIP design environments can be divided into two streams. Some design environments are just based on pre-defined processor platforms and provide limited options for user customization. Other environments provide architectural processor description language for the designers to describe their target architectures by inserting some user-defined structures in the base one.

R.E.A.L of Philips [2], Xtensa of Tensilica [3], Jazz DSP processor of Improv Systems [4] and ARCtangent-A5 of ARC [5] are the commercial design environments using pre-defined processor platforms approach.

Xtensa of Tensilica is a configurable, extensible and synthesizable RISC (reduced instruction set computer) processor with load store architecture. Its base architecture has a compact 16- and 24-bit instruction set comprising of 80 instructions. The configurable parameters include the choice of 32 or 64 general-purpose 32-bit registers, the size of cache, the write buffer size, the availability of designer defined instruction execution unit and etc. Designers can define the mnemonic, the encoding, and the semantics of single cycle instructions using TIE language. In addition, the development environment includes ANSI C/C++ compiler, linker, assembler, debugger, code profiler, and instruction set simulator.

The R.E.A.L of Philips is customizable DSP having two independent 16x16 bit multipliers, four parallel 16-bit ALUs which can be combined into two 40-bit ALUs (including eight overflow bits each), and a number of parallel shifters and saturators in base architecture. Besides a standard 16- and 32-bit instruction set, there are additional Application Specific Instructions (ASIs), which allow the full parallelism of the DSP to be exploited. The ASI concept allows up to 256 VLIW instructions in a 96-bit width look-up table inside the R.E.A.L. DSP. These are triggered by a special class of 16 bit instructions, stored in the normal program memory. The ASI look-up table can be a RAM (for prototype chips), ROM, a synthesized netlist, or a combination of these. If the ASI table is implemented in RAM, then its contents can be modified using the JTAG port, or under DSP program control by writing to dedicated registers within

the DSP.

The ARCTangent-A5 of ARC is a four stages 32-bit RISC processor that can be configured and extended match the application requirements. Designers can customize the processor in two ways: configuration and extension. Configuration is the ability to change existing features of the processor, such as the main-memory and auxiliary-bus widths; the size and organization of the instruction and data caches; or the size of local memory and DSP XY memory. Extension is the ability to add entirely new features to the processor such as a 32x32-multiply instruction, a USB peripheral and user-defined application-specific extensions. The resulting core is generated to HDL code together with synthesis scripts, simulation make-files, documentation and an automated test environment.

The Jazz DSP processor of Improv Systems is a configurable VLIW processor for their proprietary Programmable System Architecture (PSA). Improv employs this architecture that can scale from a single, uniquely configured Jazz DSP processor core, to a system level platform implementation that consists of many of these uniquely configured Jazz processors in an interconnect structure defined by shared memory maps between the processors. Each processor instance can be customized by custom RTL blocks and instructions to create a designer-defined DSP core. The Jazz PSA Composer Tool Suite provides designers with automatically generated synthesizable HDL code and a full set of software design tools including the debugger, simulator and profiler.

Other design environments using architecture description languages include the design environment of Retarget Compiler Technologies [6], LISA Processor Design Platform [7] [8], MetaCore [9] and PEAS-III [10].

The design environment of Retarget Compiler Technologies is based on the processor modelling language nML. nML offers designers the abstraction level for describing a processor architecture and instruction set (ISA), which serves as an input to the various tools. nML captures the specification of the processor's instruction set, together with sufficient structural information to enable efficient compilation. Processor designers can describe alternative instruction-

set architectures in nML. The support-tools for corresponding architecture are automatically available. Once the architecture has been optimized in nML, the processor description can be translated automatically into an HDL model. This HDL description can be synthesized with commercially available synthesis tools, for ASIC or FPGA implementation.

The LISA Processor Design Platform (LPDP) tool-suite is based on the machine description LISA. Starting from architecture descriptions in the LISA language, software development tools can be generated including HLL C-compiler, assembler, linker, simulator, debugger frontend. LISA is a language which aims at the formal description of programmable architectures, their peripherals, and external interfaces. The language elements of LISA enable the description of different aspects of processor architectures like behaviour, instruction set coding and syntax. The language LISA and its generic machine model are able to produce bit- and cycle/phase-accurate models of systems that consist of programmable architectures and peripheral hardware components. Moreover, synthesizable HDL (VHDL, Verilog, SystemC) code of the target processor can be generated which can be processed by the standard synthesis tools.

MetaCore is a DSP-oriented ASIP development system that can generate efficient ASIP using benchmark-driven design methodology. The heart of the MetaCore system is a predefined microarchitecture. The design style of the predefined microarchitecture is parameterized and pipelined. The architectural parameters include register file size, bus width, address space of each memory, and bit width of functional blocks. The specification of the target ASIP in the MetaCore system is described using the structural specification language MSL and behavioural specification language MBL. MSL is used to specify the data path structure of the target microarchitecture, while MBL is used to specify the architectural parameters and the behaviour of instructions for the target ASIP. The MSL description consists of declarations of hardware resources such as busses, latches, multiplexer, functional units, and interconnections among the hardware resources. A synthesis tool called SMART is used to translate the given processor specification into the corresponding HDL code of the target

ASIP equipped with the user-defined application-specific instructions.

PEAS-III is an architectural level processor design environment based on a micro-operation description of instructions. In the environment, designers model the target processor with the following five items:

1. Architecture parameters such as the number of pipeline stages and the number of delayed branch slots
2. Declarations of resources to be included in processor (e.g. ALUs, registers)
3. Instruction format definitions which include interrupt conditions and the number of execution cycles of interrupt conditions and the number of execution cycles of interrupt
4. Micro-operation descriptions of instructions and interrupts. PEAS-III synthesizes the datapath and the control logic of the processor, and generates a simulation model and synthesizable VHDL descriptions of the processor.

1.2 Motivation

We try to design the speaker-independent isolated word speech recognizer using ASIP methodology. There are many alternatives of accomplishing the speech recognition application, ranging from entirely hardware ASICs to completely software implementation. It is inflexible to cope with late design changes in ASICs approach while it largely ignores the potential enhancement in hardware structure in software approach. Related ASIP work mentioned in previous section mainly emphasize the hardware architecture that is common to the generic DSP applications, but not a particular application domain. Their work also overlooks the possibility of optimizing the application in the eyes of software developers. It is equally important to consider the design from software's point of view in the hope of converting the complex algorithm into a simpler mathematical form, extracting application-specific instruction for repetitive operations as well as fully integrating the inherent advantages of software and hardware

co-design. We try to find out any optimizations which are feasible in the ASIP from both software and hardware designers' perspectives.

1.3 ASIP Design Methodologies

The philosophy of ASIP is the exploitation of optimized user-defined instruction set and datapath. It is an instruction-level programmable processor with an architecture and instruction sets tuned to a specific application. Design of ASIP requires a good balance between flexibility and performance to provide the most optimal solution. It also covers multi-disciplinary areas like computer architecture and logic design, DSP algorithm analysis, software programming and integration of the hardware platform. As a consequence, designing the ASIP represents a hardware/software co-design task. An overview of the entire ASIP design flow is depicted in Figure 1.2.

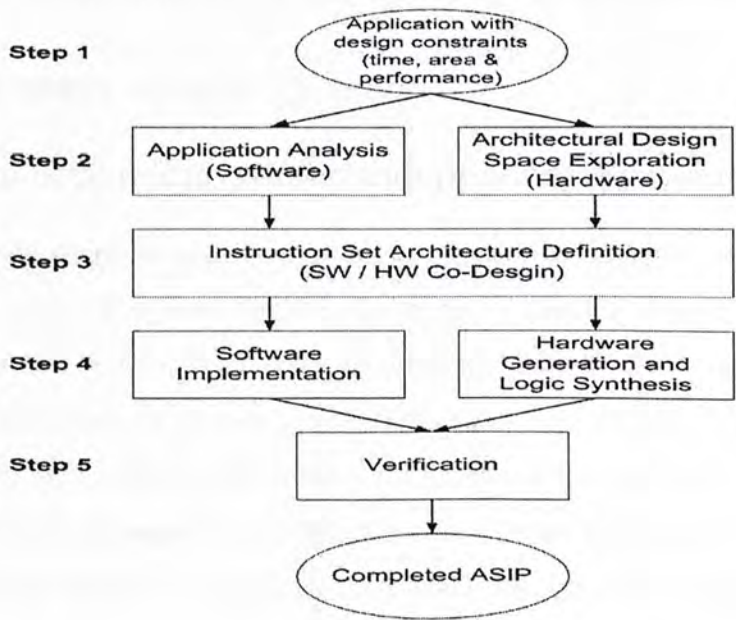


Figure 1.2: Overview of ASIP Design Flow

There are mainly 5 steps for the ASIP design flow.

1. The design flow starts with the consideration of targeted application, specification and design constraints such as timing, area and performance.

2. It involves the partitioning of the software and hardware analysis. A high-level language like C is written to have a thorough understanding on the algorithms. Possible architecture is also studied for a specific application that requires the minimum hardware costs.
3. A new instruction set is defined to act as an interface between the software implementation and hardware platform.
4. Programs are written by the pre-defined instructions while Hardware Description Language (HDL) like VHDL/Verilog is used for hardware generation and logic synthesis.
5. The final design must be verified to see if it still meets the specification requirements and works under different constraints.

1.4 Fundamentals of Speech Recognition

Classification of ASR systems

Speaker-dependent versus independent System

Speaker dependent recognition systems are trained to recognize some particular speaker's voice. This is accomplished through a training session, which allows the speakers to record their voice beforehand. The system is tailor-made to some specific speaker's pronunciations, inflections, and accents. The recognition accuracy is high since speaker's voice are pre-stored in database.

In contrast, Speaker independent systems mean that any individual can speak directly to the computer without going through the training process of his own voice. That means speaker who utters via microphone may not necessarily have his own voice pre-recorded in the database. Speaker-independent approaches are the only ones that make sense where speech training process is impossible. It is sometimes difficult to expect every user to go through the trouble of training his own voice first in the recognition system before they have

any applications. The system may have lower recognition accuracy compared with speaker dependent systems, but it retains flexibility.

Small versus Large Vocabulary Size

The real issue is how big a vocabulary is required by the application and how much of the vocabulary can be made active at one time. For example, an office dictation application might require a vocabulary of 30,000 words while an industrial inspection task might require only 300 words. The maximum number of words that are active at one time can depend on memory availability, recognition accuracy, and response time required by different applications. It is obvious that the smaller the vocabulary size, the less memory required, the higher accuracy as well as the faster response time.

Portable versus Non-Portable Hardware System

Some speech recognition applications like manufacturing inspection or environmental surveillance require portable hardware. Size, power, and memory locations are the major limitations of the design that must be considered. Other systems such as mainframe-based or stationary desktop computers tend to use more powerful processors and memory-intensive algorithms to achieve better speech recognition performance.

The Properties of Speech

Speech utterances are unpredictable, time-varying and random in nature. There are wide range of possibilities to represent a speech signal, including energy, pitch, tone and other related parameters. Probably, the most effective way of modeling the speech signal is the short-time stationary segmentation. By dividing the speech into the same period region with overlapping, many separated speech frames are formed. As the time period of each frame is very short (10 ~20 ms), that segmented speech frame can be assumed to be stationary. Figure 1.3 illustrates the short-time stationary characteristic of speech signal.

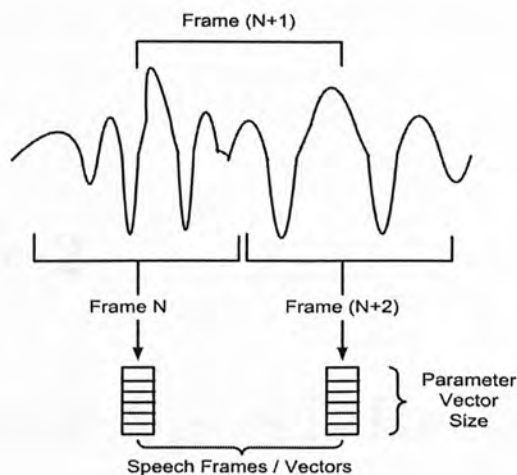


Figure 1.3: The Short-time Stationary Characteristic of Speech Signal

The final extracted vectors are the feature vectors that represent the special characteristics of the speech. It is these feature vectors that act as the input for the later process of speech recognition.

1.5 Thesis outline

The remainder of the thesis is organized as follows:

Chapter 2 describes the completed procedure of automatic speech recognition in theory, including the front-end feature extraction and back-end HMM-based Viterbi search.

Chapter 3 briefly introduces the design of ASIP platform. Its focus is on the hardware architecture, functional description of each module and the design space of datapath exploration.

Chapter 4 presents the practical implementation of speech recognition on the ASIP platform. There is wide discussion on different optimization techniques, the working mechanisms and design considerations.

Chapter 5 proves the effectiveness of implementing speech recognition on ASIP by revealing the performance in terms of speed and recognition accuracy.

Chapter 6 summarizes the overall research work and suggestions for the future work in this area.

Chapter 2

Automatic Speech Recognition

2.1 Overview of ASR system

This section describes an HMM-based Isolated Word Recognition (IWR) system that can be divided into two parts, the front-end and back-end processing. The front-end includes data acquisition and feature extraction. Through the input via microphone and a codec from which digitized speech data are generated, it produces important and useful acoustic features for speech recognition. The back-end is the HMM-based Viterbi search where Gaussian mixture computation and memory usage take place frequently. This finally leads to the result of which word is being spoken. Figure 2.1 shows the block diagram of the speech recognition system.

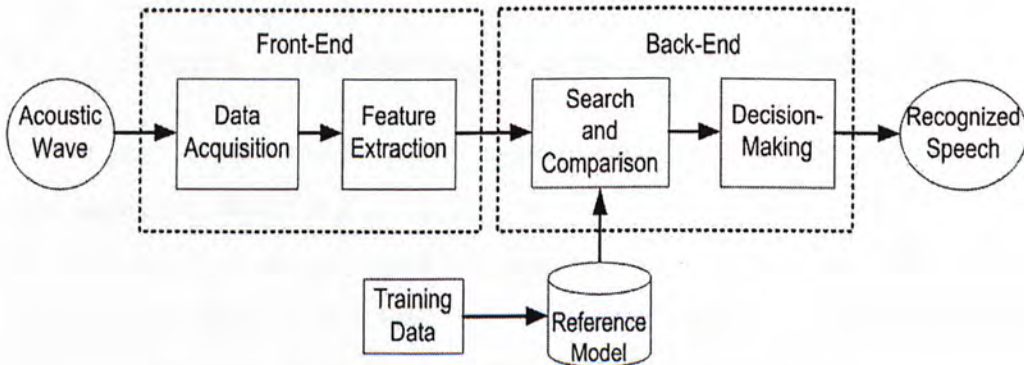


Figure 2.1: Automatic Speech Recognition System

2.2 Theory of Front-end Feature Extraction

The acoustic feature generated by the signal processing front-end is Mel-Frequency Cepstral Coefficients (MFCC), which are calculated using the real cepstrum, defined as the inverse Fourier transform of the log spectrum:

$$c_s(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log|S(w)|e^{jwn} dw$$

where $S(w)$ is the spectrum of the speech signal. The acoustic features consist of 12 cepstral coefficients together with energy. The features are energy normalized and cepstral mean normalized based on each short time segment. It is known that the performance of a speech recognition system can be greatly enhanced by adding dynamic time derivatives to basic static parameters. However, only the static coefficients and first-order dynamic time derivative coefficients are included in the feature vectors with consideration of tight memory as well as the extensive computation time.

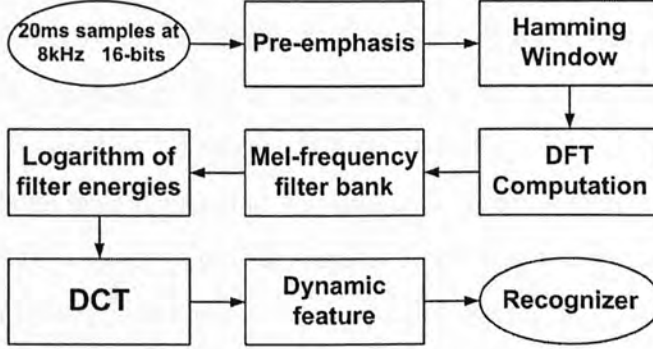


Figure 2.2: The Algorithm for Front-end MFCC Computation

In practice, the mel-frequency cepstral coefficients can be computed using the algorithm shown in Fig 2.2.

The digitized speech signal $s(n)$ derived from the data acquisition step is 16-bit linear sampled at 8 kHz. It is a common practice to pre-emphasize the signal by applying the first-order differential equation,

$$\tilde{s}(n) = s(n) - \alpha \cdot s(n-1)$$

where α is the pre-emphasis coefficient which should be in the range $0 \leq \alpha \leq 1$. Then, the pre-emphasized speech signal $\tilde{s}(n)$ is segmented into frames with size of 20 ms. With overlapping of 10 ms between frames, each frame is multiplied by a 160-point Hamming window $w(n)$.

$$x(n) = w(n) \cdot \tilde{s}(n)$$

Next the spectral magnitude of the windowed signal is computed by Discrete Fourier Transform (DFT). The magnitude is then processed by a series of overlapping triangular filters, $H_m(k)$, which are centered at equally spaced frequencies in the mel-scale, to find an estimation of mel-spectrum. The logarithmic scale is taken to produce a weighted log energy $Y(m)$. This results in computation of the total energy in the m th band,

$$Y(m) = \log_{10} \left[\sum_{k=0}^{N-1} |X(k)|^2 \cdot H_m(k) \right], 0 \leq m \leq M \quad (2.1)$$

where $X(k)$ is the DFT of the windowed speech signal, $H_m(k)$ is the filter-bank coefficients, N is the length of a frame and M is the number of filters. The weighted log energy is real and even, so the inverse Fourier Transform can be implemented as a Discrete Cosine Transform (DCT). This transformation decorrelates features so that the diagonal covariance matrices can be used instead of full covariance matrices. Cepstral coefficients have rather different dynamics, the higher coefficients show the smaller variance. It is desirable to have a constant dynamic range across coefficients for modeling purposes. One way to reduce these differences is to apply liftering windows which weight cepstral coefficients $C(k)$ differently,

$$\hat{C}(k) = C(k) \cdot \left\{ 1 + \frac{M}{2} \cdot \sin \left(\frac{k\pi}{M} \right) \right\}, 0 \leq k < M \quad (2.2)$$

where M is the number of filters. Finally, the first-order time derivatives of feature vectors are estimated to represent the dynamic characteristics of speech signals.

2.3 Theory of HMM-based Speech Recognition

2.3.1 Hidden Markov Model (HMM)

Though speech signal is well-known for its variability, the spectral properties of the frames of a pattern can be characterized by Hidden Markov Model (HMM), one well-recognized and widely used statistical method. The underlying assumption of HMM is that the speech signal can be well modeled as a parametric random process.

2.3.2 The Typical Structure of the HMM

An HMM is characterized by the number of states N , the state transition probability matrix A , the observation symbol probability distribution B and the initial state distribution π . Given an HMM $\lambda = (A, B, \pi)$ and an observation sequence O , we wish to calculate the probability of the observation sequence $P(O|\lambda)$. These probabilities are a measure of how well the data match each state in the model. With left-to-right topology, the formal model for the HMM is shown in Figure 2.3.

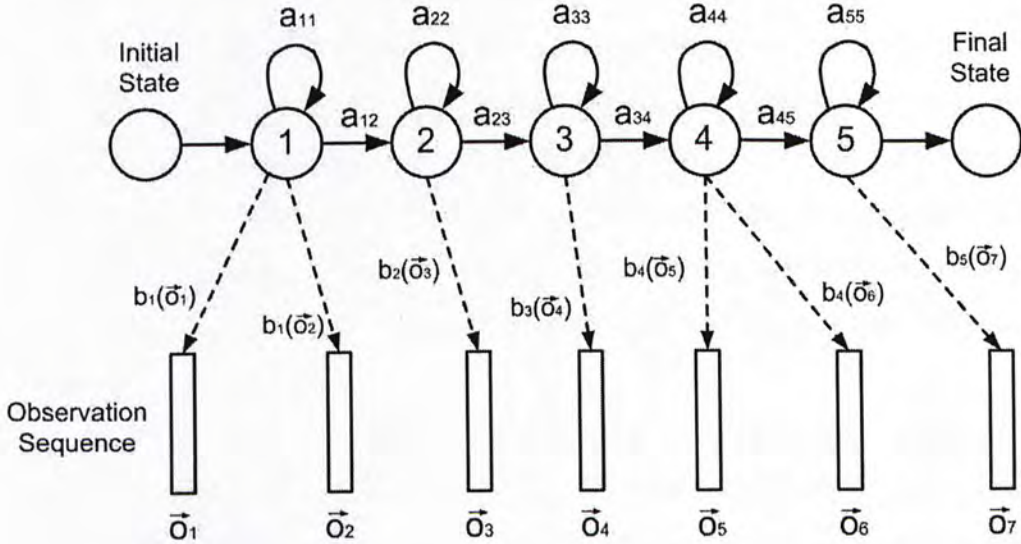


Figure 2.3: The Simplified View of Hidden Markov Model

The state transition probability matrix $A = \{a_{ij}\}$, which indicates the prob-

ability for state i to change to state j .

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

Note that the sum of the probabilities of all transitions with the same current state must be equal to one.

$$\sum_{j=1}^N a_{ij} = 1$$

The observation sequence $O = \{\vec{o}_t\} = (\vec{o}_1 \vec{o}_2 \dots \vec{o}_T)$, where $1 \leq t \leq T$ and T is the number of observations in the sequence. The observation symbol probability $B = \{b_j(\vec{o}_t)\}$, it is the probability density function for each state and the argument \vec{o}_t is an acoustic feature vector.

2.3.3 Discrete HMMs and Continuous HMMs

There are two different forms of HMMs, including the discrete observation HMM and the continuous observation HMM.

For the discrete observation HMM, it is restricted to the production of a finite set of discrete observations. Vector Quantization(VQ) is used to associate each continuous feature vector with a discrete value. Vector Quantization is a sampling process of continuous signals and this results in a serious loss of data. In reality, the observations are usually representations of continuous signals in most applications. VQ of these continuous signals can degrade the performance significantly.

For the continuous HMMs, the observations are continuous (or vectors). It would be beneficial to model continuous speech signals directly with continuous observation densities. The most general representations of continuous observation density is in the form of Gaussian probability density function (pdf). Figure 2.4 shows the single-mixture distribution, which means that there is only one pdf at each state in HMM.

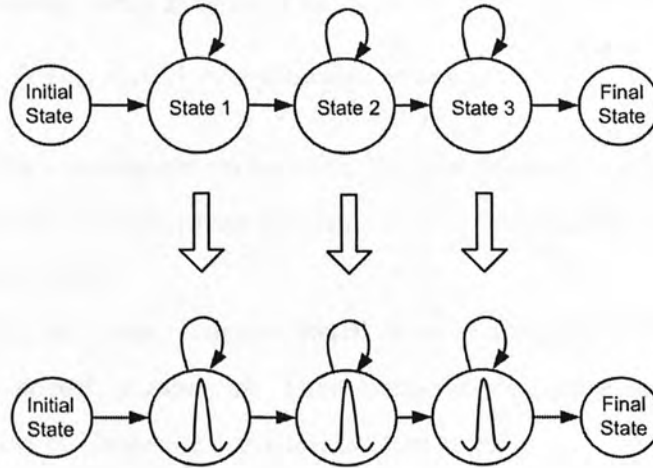


Figure 2.4: The HMM with Single-mixture Distribution

For such a pdf mentioned above, it is not sufficiently flexible to accurately model the variation which occurs between different acoustic vectors that correspond to a state. This is particular true if the models are used to characterize speech from a number of speakers. Thus, Gaussian double mixtures are typically used to model broad sources of variability. Figure 2.5 shows the double-mixture distribution, which means that there are two pdf at each state in HMM to accurately model the highly varied speech.

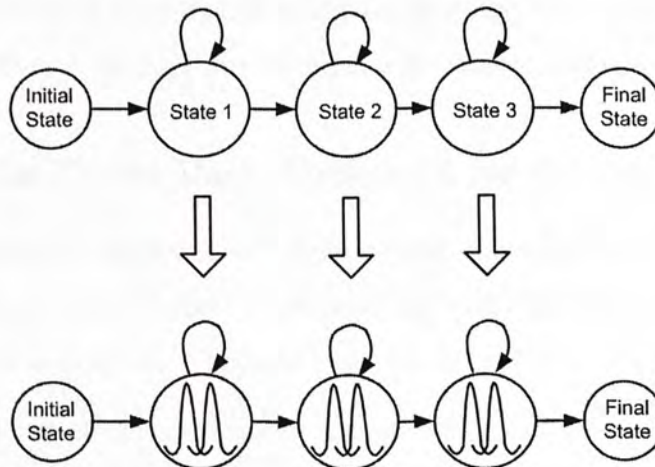


Figure 2.5: The HMM with Double-mixture Distribution

The most popular form of the output probability density function (pdf) is

Gaussian density, which is defined as

$$b_j(\vec{o}_t) = \hat{N}(\vec{o}_t, \vec{\mu}_j, U_j) = \frac{1}{\sqrt{(2\pi)^D \det(U_j)}} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1} (\vec{o}_t - \vec{\mu}_j)} \quad (2.3)$$

where \vec{o}_t is the observation vector with the dimensionality of D at time t , and \hat{N} is a Gaussian pdf with mean vector $\vec{\mu}_j$ and the determinant of the covariance matrix U_j in state j .

In practise, only one Gaussian distribution is not sufficient to appropriately estimate the speech parameters. Thus multi-variate Gaussian density, which is weighted sums of Gaussian densities, is often used.

$$b_j(\vec{o}_t) = \sum_{k=1}^M c_{jk} \hat{N}(\vec{o}_t, \vec{\mu}_{jk}, U_{jk}) = \sum_{k=1}^M c_{jk} \frac{1}{(2\pi)^D \det(U_{jk})} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_{jk})^T U_{jk}^{-1} (\vec{o}_t - \vec{\mu}_{jk})} \quad (2.4)$$

where c_{jk} is the mixture coefficient for the k th mixture in state j , M is the number of mixtures per state and \hat{N} is a multivariate Gaussian with mean vector $\vec{\mu}_{jk}$ and the determinant of the covariance matrix U_{jk} for the k th mixture component in state j . The mixture coefficient c_{jk} satisfies the following constraints:

$$\begin{aligned} \sum_{k=1}^M c_{jk} &= 1, \quad 1 \leq j \leq N \\ c_{jk} &\geq 0, \quad 1 \leq k \leq M \end{aligned}$$

It is flexible to alter mixture densities to sufficiently approximate the arbitrary-shaped densities if an appropriate number of components are used.

2.3.4 The Three Basic Problems for HMMs

Given HMM with model $\lambda = (A, B, \pi)$ and the observation sequence $O = (\vec{o}_1 \vec{o}_2 \dots \vec{o}_T)$, there are three basic problems that must be solved for the model to implement in real-world applications. These problems are listed as follows:

1. How do we efficiently compute the probability of the observation sequence $P(O|\lambda)$ given the observation sequence O and the model λ ?
2. How do we choose the corresponding state sequence $q = (\vec{q}_1 \vec{q}_2 \dots \vec{q}_T)$ that best describes the observations given the observation sequence O and the model λ ?

3. How do we adjust the model λ to maximize $P(O|\lambda)$?

Problem 1 is the recognition problem, given an output sequence and a model, what is the probability that the model could have created the sequence. The problem can also be viewed as calculating scores to find out how well a given model matches a given observation sequence. Comparing new data to the models of known signals can solve the recognition problem. If there are V words to be recognized, then there will be V distinct HMM to model each word separately. The recognition result of the unknown word is based on the final scores of each word model that match the given observation sequence (input feature vector). The word whose model score is the highest will be selected as the recognition output.

Problem 2 is the sequence problem, given an output sequence and a model, what is the optimal and most likely sequence of states that could have created the output sequence. Segmenting the training sequence of each word into more states is the solution to sequence problem because it makes refinements of the model and improve its capability of modeling the spoken word sequences.

Problem 3 is the training problem, given the output sequence and the topology, how can the parameters of a model be adjusted to maximize the probability that the model creates the output sequence. The training problem can be solved by finding optimal model parameters of the known data and training the reference data to create the best models.

2.3.5 Probability Evaluation

To implement speech recognition, we need to calculate the probability of the observation sequence $O = (\vec{o}_1 \vec{o}_2 \dots \vec{o}_T)$, given the HMM model λ , i.e., $P(O|\lambda)$. The most straightforward way of calculating it is through enumerating every possible state sequence of length T .

Consider one fixed N -state sequence $q = (\vec{q}_1 \vec{q}_2 \dots \vec{q}_T)$, where q_1 is the initial state. The probability of observation sequence is obtained by summing

the probability over all possible state sequence q as

$$P(O|\lambda) = \sum_{q_1, q_2, \dots, q_T} \pi_{q_1} b_{q_1}(\vec{o}_1) a_{q_1 q_2} b_{q_2}(\vec{o}_2) \dots a_{q_{T-1} q_T} b_{q_T}(\vec{o}_T) \quad (2.5)$$

From Equation 2.5, the interpretation of the probability computation can be illustrated as follows. Initially (at time $t=1$) we are in state q_1 with probability π_{q_1} , and generate the symbol \vec{o}_1 in this state with probability $b_{q_1}(\vec{o}_1)$. The time changes from t to $t+1$ (at time=2) and we make a transition from state q_1 to state q_2 with probability $a_{q_1 q_2}$, and generate symbol \vec{o}_2 with probability $b_{q_2}(\vec{o}_2)$. This process continues until we reach the last transition (at time T) from state q_{T-1} to state q_T with probability $a_{q_{T-1} q_T}$, and generate symbol \vec{o}_T with probability $b_{q_T}(\vec{o}_T)$ [11].

The calculation of $P(O|\lambda)$ involves $2T \cdot N^T$ order of calculations because there are N possible states at every $t = 1, 2, \dots, T$, that can be reached (i.e., there are N^T possible state sequences), and for each such state sequence about $2T$ calculations are required for each term in the summation of Equation 2.5. It is difficult or infeasible to compute probability in this way even for small values of N and T . For example, state $N=2$ of HMM with total frames $T=200$ of the speech signal, there are $2 \cdot 200 \cdot 2^{200} \approx 10^{62}$ computations. Fortunately, an efficient algorithm called Viterbi can be used to solve the problem. The main difference is that instead of summing the probabilities of transitions from all states in the previous method as shown in Equation 2.5, only the most probable transition is considered and the rest is discarded. But one more step is needed to trace back from the most probable final state which reveals the most probable state sequence.

2.4 The Viterbi Search Engine

The Viterbi algorithm aims at finding the best state sequence, $q = (\vec{q}_1 \vec{q}_2 \dots \vec{q}_T)$ for the given observation sequence $O = (\vec{o}_1 \vec{o}_2 \dots \vec{o}_T)$. The highest score $\delta_t(i)$ is defined as

$$\delta_t(i) = \max_q P[q_1 q_2 \dots q_{t-1}, q_t = i, o_1 o_2 \dots o_t | \lambda]$$

$\delta_t(i)$ is the best score with the highest probability along a single path, at time t , which accounts for the first t observations and ends in state i . The array $\psi_t(j)$ is used to keep track of the argument that maximized the probability for each t and j . The complete procedure for implementing the Viterbi algorithm [11] can be stated in three steps:

1. Initialization

$$\delta_1(i) = \pi_i b_i(\vec{o}_1), \quad 1 \leq i \leq N$$

$$\psi_1(i) = 0$$

2. Recursion

$$\delta_t(j) = \left[\max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} \right] \cdot b_j(\vec{o}_t), \quad 1 \leq j \leq N, \quad 2 \leq t \leq T$$

$$\psi_t(j) = \arg \max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij}$$

3. Termination

$$P = \max_{1 \leq i \leq N} [\delta_T(i)]$$

$$q_T = \arg \max_{1 \leq i \leq N} [\delta_T(i)]$$

It should be clear that a lattice structure efficiently implements the computation of the Viterbi procedure as shown in Figure 2.6. The recursion forms the basis of the so-called Viterbi algorithm. This algorithm can be visualised as finding the best path through a matrix where the vertical dimension represents the states of the HMM and the horizontal dimension represents the frames of speech (i.e. time). Each large dot in the picture represents the probability of observing that frame at that time and each arc between dots corresponds to a transition probability. The path always goes in the direction with higher probability. For example, the starting point is at time=1 and at state=A (i.e. 1A). The path can go to two directions either. One direction is $1A \rightarrow 2B$. Another is $1A \rightarrow 2A$. Suppose the route for $1A \rightarrow 2B$ has a higher probability than the one for $1A \rightarrow 2A$, then the path will simply go upwards with a 45-degree direction to reach point 2B. The paths are grown from left-to-right and column-by-column. The comparison process continues until it reaches both of the final state and the last speech frame (i.e. 5D).

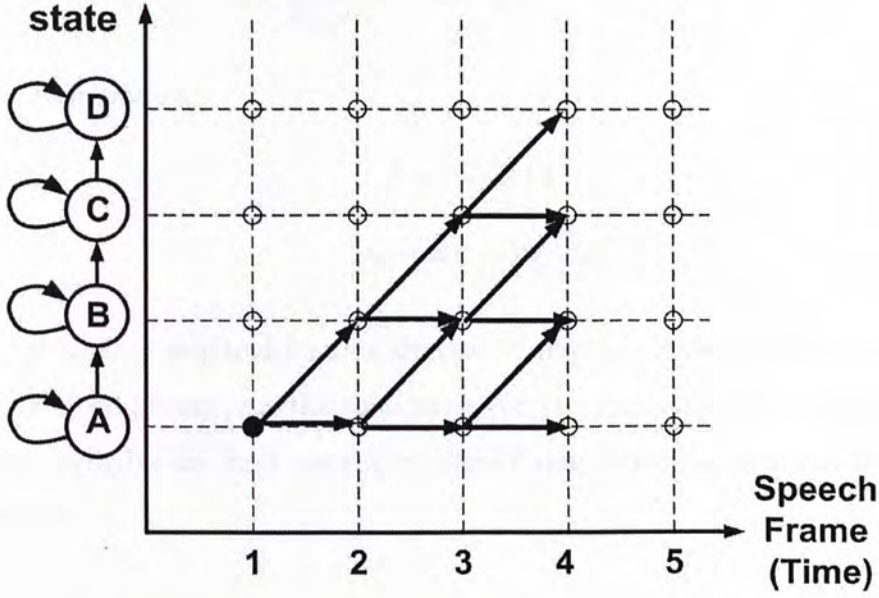


Figure 2.6: The Lattice Structure of Simplified Viterbi Search

Alternative Viterbi Implementation

The previous section of the Viterbi algorithm requires frequent operation of multiplication, which is unfavorable for the hardware implementation. By taking logarithms of the model parameters, there is no need to have any multiplication operations. The multiplication is converted to addition after taking logarithms. The main procedures of modified Viterbi algorithm [11] are shown as belows:

1. Preprocessing

$$\tilde{\pi}_i = \log(\pi_i), \quad 1 \leq i \leq N$$

$$\tilde{b}_i(\vec{o}_t) = \log[b_i(\vec{o}_t)], \quad 1 \leq i \leq N, \quad 1 \leq t \leq T$$

$$\tilde{a}_{ij} = \log(a_{ij}), \quad 1 \leq i, j \leq N$$

2. Initialization

$$\tilde{\delta}_1(i) = \log(\delta_1(i)) = \tilde{\pi}_i + \tilde{b}_i(\vec{o}_1), \quad 1 \leq i \leq N$$

$$\psi_1(i) = 0$$

3. Recursion

$$\tilde{\delta}_t(j) = \log(\delta_t(j)) = \max_{1 \leq i \leq N} [\tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}] + \tilde{b}_j(\vec{o}_t)$$

$$\psi_t(j) = \arg \max_{1 \leq i \leq N} [\tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}], \quad 1 \leq j \leq N, \quad 2 \leq t \leq T$$

4. Termination

$$\begin{aligned} \tilde{P} &= \max_{1 \leq i \leq N} [\tilde{\delta}_T(i)] \\ q_T &= \arg \max_{1 \leq i \leq N} [\tilde{\delta}_T(i)] \end{aligned}$$

The calculation required for this alternative implementation is on the order of only N^2T additions plus the calculation for preprocessing. The preprocessing cost is negligible for most systems because it only performs once and the values are saved.

2.5 Isolated Word Recognition (IWR)

To build the speaker-independent HMM-based isolated word recognizer, assume there is a vocabulary of V words to be recognized and each word is modeled by distinct HMM, there are mainly two parts that are crucial for the implementation of isolated word recognizer.

1. Offline Training
2. Real-time Isolated Word Recognition

Offline Training

Each word in the vocabulary has a training set of K utterances of the word. Given a set of training utterances corresponding to a particular word model, the parameters of that model (A, B, π) can be determined automatically by a robust and efficient HTK toolkit, which is primarily designed for building HMM-based speech processing tools, in particular recognizers. Provided that a sufficient number of representative utterances of each word can be collected, a HMM λ_v can be constructed which implicitly models all sources of variability inherent in real speech. The training procedure is shown in Figure 2.7.

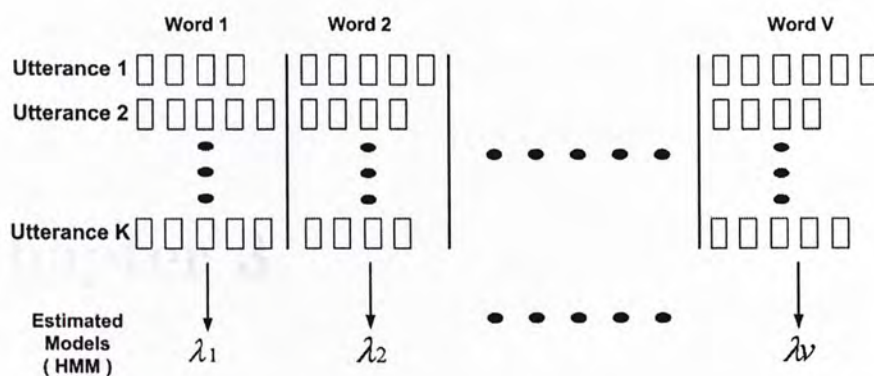


Figure 2.7: The Training Process for Generating HMM Reference Models

Real-time Isolated Word Recognition

For each unknown word to be recognized, the processing shown in Figure 2.8 must be carried out. The front-end process of the real-time speech extracts the useful features and represents those features in the form of observation sequence O . The likelihood of each model generating that observation sequence O of unknown word is calculated and the most likely model identifies the word. In other words, it is necessary to have calculation of model likelihoods for all possible models, $P(O|\lambda)$, $1 \leq v \leq V$. The recognition result can be found by selecting the word whose model likelihood is the highest.

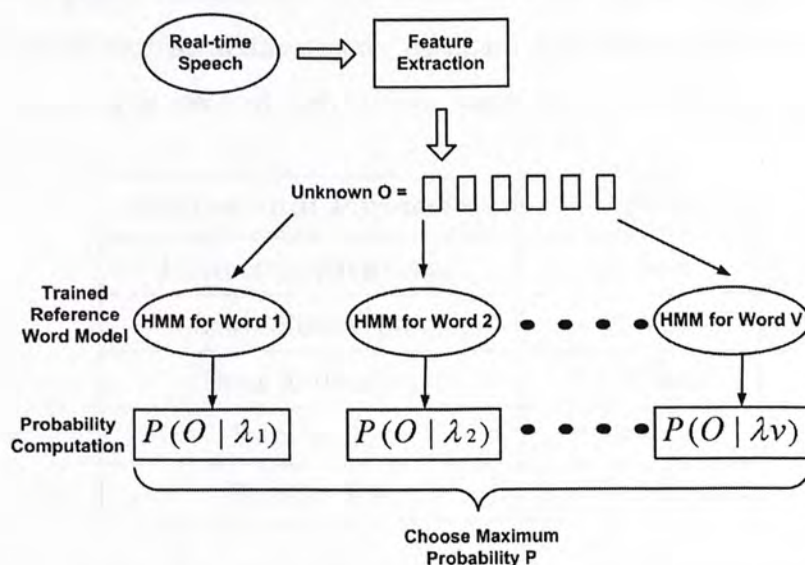


Figure 2.8: Using HMMs for Isolated Word Recognition

Chapter 3

Design of ASIP Platform

Our ASIP processor [12] mainly focuses on the digital signal processing applications like speech, audio and video. In order to meet the real-time requirements of various applications, the ASIP processor has been specially designed for applications which are computationally intensive and repetitive.

The design goal of the processor is to maximize the degree of reusability by re-programming it with efficient application specific instruction set and minimizing the impact on timing and power consumption when changing the architectural parameters.

The proposed architecture of the processor is divided into four parts , namely instruction fetch, instruction decode, datapath and register file system as shown in Figure 3.1. The selected architectural parameters are listed in Table 3.1.

Architectural Parameters	Values
Instruction Addressing	16 bits
Instruction Width	24 bits
Data Addressing	2 x 16 bits
Data Width	16 bits
Register File	2 x 64 x 16 bits

Table 3.1: The Architectural Parameters of the ASIP

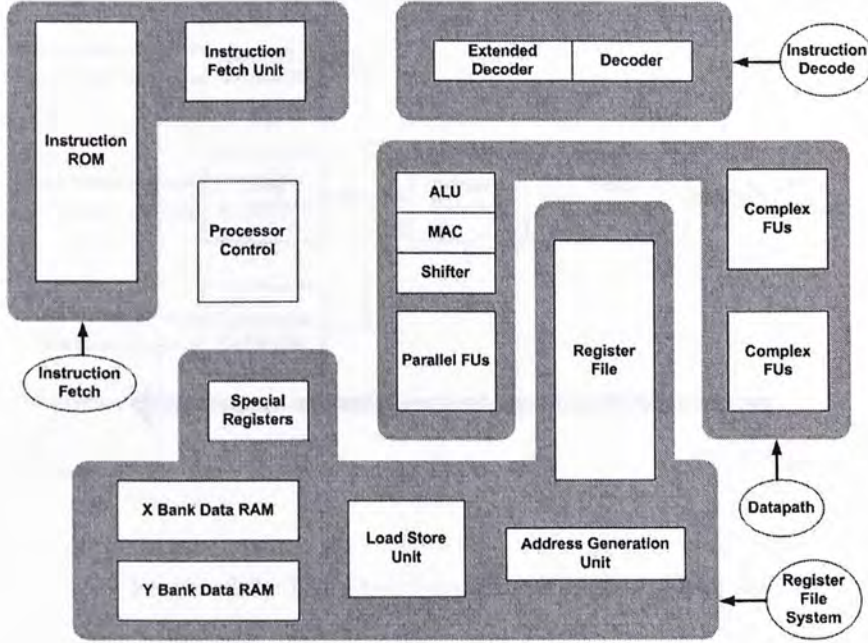


Figure 3.1: The Organization of the ASIP Architecture

3.1 Instruction Fetch

The Instruction Fetch Unit (IFU) is responsible for reading instructions from program memory, passing them to the instruction decoder and updating the program counter.

The structure of the instruction fetch unit is shown in Figure 3.2. It consists of five major modules, including program counter, address selector, branch controller, loop controller and subroutine controller.

IFU begins to operate autonomously as soon as reset is released. The program counter is a register that stores the current position of the program. This stored value is used as the instruction address for fetching instructions. On the other hand, this value is also passed to the instruction decoder for being a reference for branching and other program flow control activities. When a branch is executed, the fetch unit must stop fetching instructions from the current stream and change the program counter to the new value. When the loop or subroutine is called, the value of program counter is stored into stack and later this value will be retrieved when the loop is completed or return from the

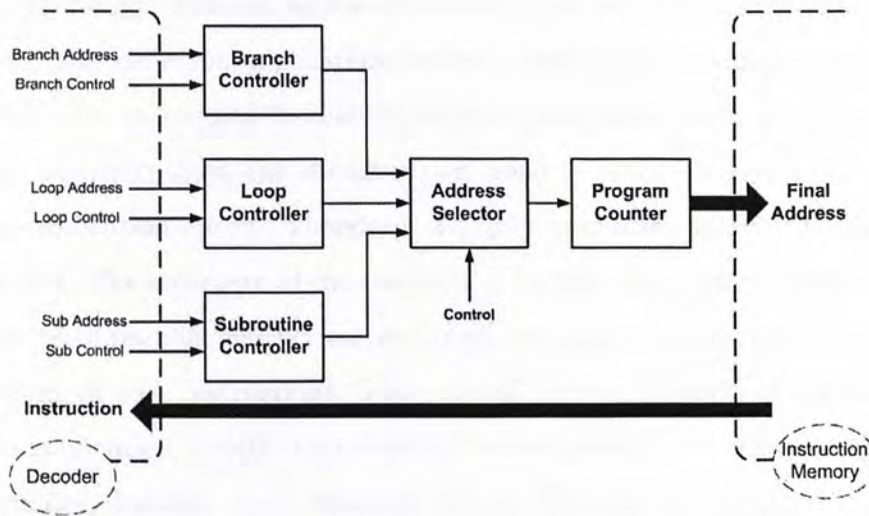


Figure 3.2: The Structure of Instruction Fetch Unit

subroutine. The program counter updates its content autonomously with the new address provided by the address selector. The address selector controls the content of the program counter. It calculates the new address based on the addresses from instruction decoder or other addresses with the control signals of branch, loop and subroutine operations. Depending on the status of the processor and the requests from those modules, the address selector supplies the appropriate address to update the program counter. In this way, branches, loops and subroutine calls can be realised.

3.2 Instruction Decode

The instruction decoder is responsible for converting the fetched instruction into useful information for different parts of the processor. Basically, it has three tasks to do :

1. Identify the fetched instruction
2. Interpret the encoded part of the instruction and generate the corresponding control signals and opcodes
3. Dispatch the decoded information to the corresponding modules

In ASIP design, different application specific instructions are introduced to accommodate different applications needs. Inevitably, the instruction decoder needs to be redesigned frequently, which is not favoured in design reuse. To meet our design goal, the changing part must be isolated in order to minimize the modification effort. Therefore, a highly modularized instruction decoder is designed. The structure of the instruction decoder is shown in Figure 3.3. The decoding of parallel instructions and complex instructions is separated from the decoding of base instructions. Two internal decoder modules are dedicated for these application specific instructions. The contents of the parallel and complex instruction decoder can be changed without altering the others.

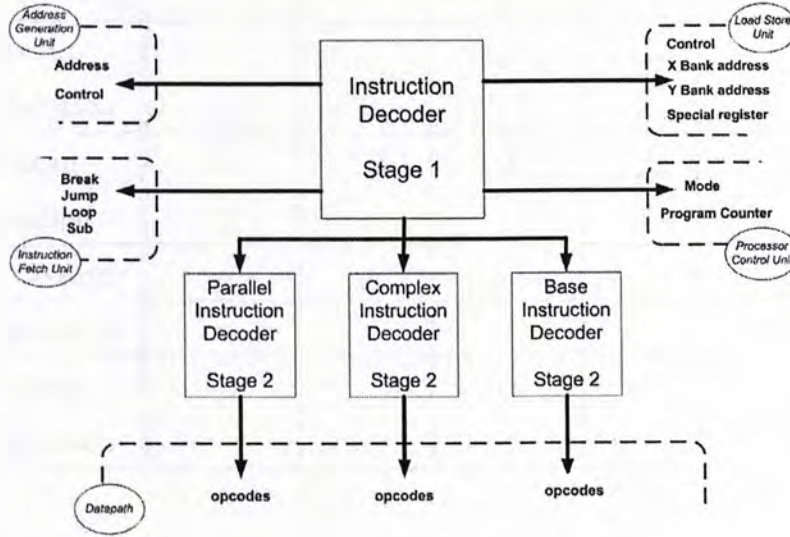


Figure 3.3: The Structure of Instruction Decoder

Secondly, the whole decoder is divided into two levels in order to match the pipeline organization. It is natural that the instructions involving execution of datapath is assigned to the second level which is closer to the datapath. This partitioning has two advantages :

1. The numbers of pipeline registers can be saved because the data that passed along the pipeline stages is the encoded part of the instruction instead of the massive control signals and opcodes
2. The unused modules can be turned off efficiently.

Table 3.2 lists the modules that are activated in the execution of different classes of instructions. It shows that the second level can be disabled when the processor is doing flow control, configuration and memory manipulation. Moreover, the first level can activate one of the modules in second level only, as the base instructions, parallel instructions and complex instructions are independent.

Classification	Instruction Fetch Unit	Processor Control Unit	Address Generation Unit	General Register File	Special Register File	Datapath
Data Processing			✓	✓		✓
Bitwise Manipulation			✓	✓		✓
Boolean Operation			✓	✓		✓
Flow Control	✓	✓				
Configuration					✓	
Memory Manipulation			✓	✓		

Table 3.2: The Processor Usage of Different Functional Units

Referring to Figure 3.3 and Table 3.2, the ASIP can be partitioned into different units for easier design and maintenance. The Instruction Fetch Unit (IFU) and Processor Control Unit (PCU) are responsible for the flow control operations, namely break, jump, loop and subroutine call and return. The Address Generation Unit (AGU) calculates the new address based on the various addressing modes. The Datapath executes the corresponding operation via General Register File. All data processing, bitwise manipulation and boolean operations require the input values to be stored in the register before they can be further processed. Special Registers allows the programmer to initialize the parameters at the beginning of the code segments. The programmer can pre-

define parameters like the memory/register start address, size and step.

To make good use of this partition, the class of the fetched instruction has to be identified as soon as possible. Hence, the instruction encoding is first based on the classification of the instructions then the number of bits needed for the arguments, so that the fetched instruction can be classified within the first few bits.

3.3 Datapath

The base datapath is designed for general DSP application. Similar to other general digital signal processors, the number representation is two's complement and the heart of this datapath is a multiplier-and-accumulator (MAC). The structure of the base datapath is shown in Figure 3.4. In the centre is a 16x16 40-bit MAC. It is made of 3:2 compressors in Wallace tree configuration, an adder and a 40-bit register for accumulation. The most significant eight bits are guard bits for avoiding overflow. The adder in the MAC is also responsible for addition and subtraction operations.

For shift operation, a barrel shifter is used in the datapath. It can shift the accumulator by at most thirty two bits to left and to right in arithmetic or logical manner. The shift distance can be defined by the immediate value from the instruction or the value stored in the register file. In addition, this shifter is also used to implement normalization operation. After the exponent instruction, the exponent of the current accumulated value is stored into a special register. This stored value is used as the shift distance of the shift operation when executing the normalization instruction.

There are also other modifiers for the accumulator : 1) logical unit for bitwise logic manipulation including AND, OR, XOR and NOT; 2) absolute unit for working out the absolute value of the accumulator; 3) negation unit for converting the accumulator to its opposite sign. The modified value is stored back to the accumulator.

Besides data processing, there is a comparison unit for comparing the two

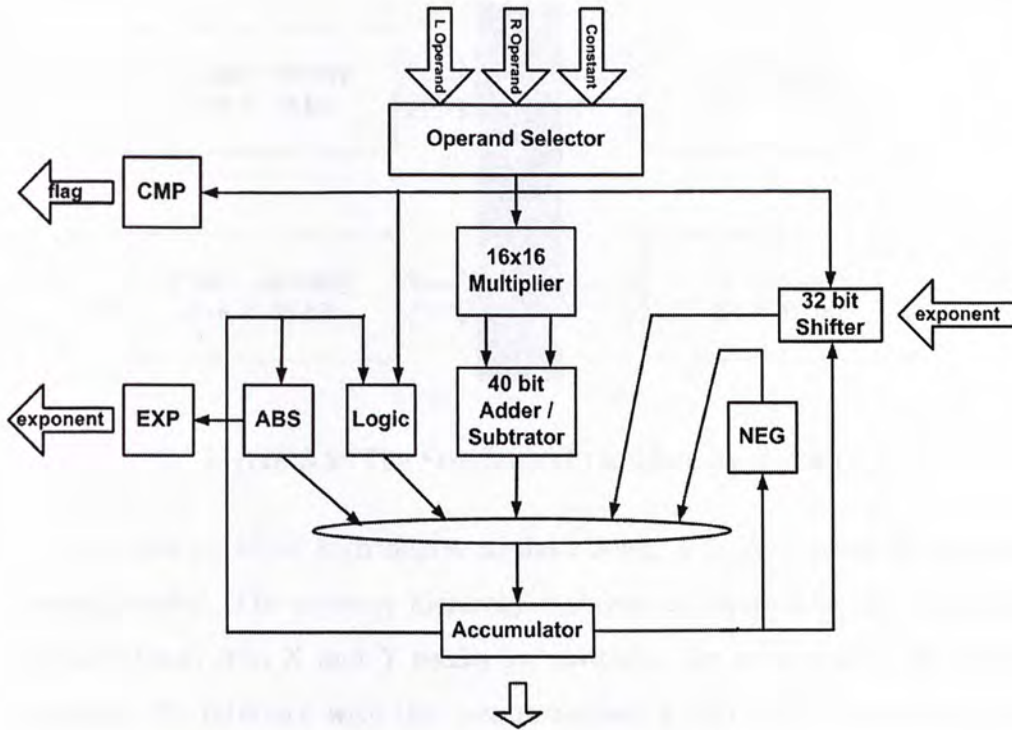


Figure 3.4: The Structure of the Base Datapath

values in the register file or comparing the accumulator with one stored value in the register file. The comparison unit can report six conditions : 1) equal; 2) not equal; 3) greater than; 4) less than; 5) greater than or equal; 6) less than or equal. The result is stored as conditional flags in special register.

A complete instruction set description is tabulated in Appendix A.

3.4 Register File Systems

3.4.1 Memory Hierarchy

In common with other current DSPs, the platform uses a dual Harvard architecture where one program memory and two separate data memories (labelled X and Y) are used. This avoids conflicts between program and data fetches, and many DSP operations map naturally onto dual memory space. For example, the data for convolution or cross correlation can be stored separately in X and Y memories.

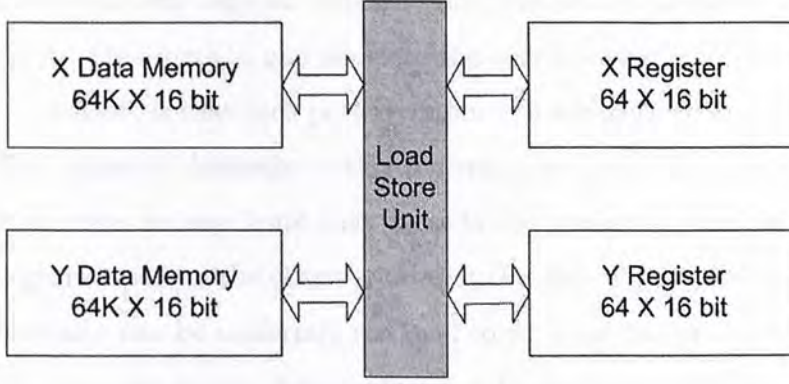


Figure 3.5: The Structure of the Memory Hierarchy

In order to allow high degree of data reuse, a large register file is highly recommended. The memory hierarchy is shown in Figure 3.5. The register file is partitioned into X and Y banks for matching the organization of the data memory. To interface with the data memories, a load store unit is used. It is designated for reading from and writing to the data memories in bulk.

3.4.2 Register File Organization

To supply enough operands to the parallel datapath without introducing any conflict, the register file is designed to be multi-ported. However, when the number of ports grows, the performance of the register file deteriorates in terms of delay, area and power consumption. Previous research work presented the impact to the performance of a register file with increasing number of arithmetic units in [13]. It showed that for N arithmetic units, the area of the register file grows as N^3 , the delay as $N^{3/2}$, and the power consumption as N^3 . The main reason is that more arithmetic units need more ports for parallel execution, which implies an exponentially growth in the complexity of the address decoder and the interconnection between the arithmetic units and the register file. Inevitably, this can cause a great impact to the platform when scaling up the parallel datapath. Partitioning register file into multiple banks was reported to be an effective solution to slowing down the performance deterioration in [13][14][15][16]. The design philosophy of multi-banked register files is to distribute

the ports to different register banks, so that the number of ports per bank can be reduced. This method can alleviate the complexity of the interconnection, but the drawback is that each port is confined to access the corresponding bank only. The primary challenge of this scheme is to avoid the number of simultaneous accesses to any bank that exceeds the available ports on each bank. Our design is based on the observation that the data which need to be accessed simultaneously can be uniformly assigned to different banks for many DSP algorithms. In other words, data conflict can be omitted if the data is carefully assigned to suitable banks.

It is a good practice to study the data access pattern of DSP algorithms. Our focus is on the convolution and correlation algorithms, which are fundamental and commonly implemented in most DSP applications. They also show a huge amount of parallelism and favour the analysis of parallel data access pattern. The general mathematical form for both of the convolution and correlation algorithms can be combined as

$$y(n) = \sum_{i=0}^{N-1} P(i)Q(n \pm i)$$

where $P(n)$ and $Q(n)$ are the two digitized input signals, N is the length of $P(n)$ or $Q(n)$, $y(n)$ is the output signal. It is assumed that $N = 8$ and there are four functional units (FU) in the parallel datapath. In the table, the bold items are the arithmetic operations. The operation **mul** represents a multiplication; operation **mac** is a multiplication-and-accumulation; operation **add** is an addition. The four functional units have their own accumulator for temporary storage which are notated in acc and t indicates a particular time instant.

Observing the data dependency of the access pattern in Table 3.3, it is possible to further partition each register bank into two blocks. The first block contains data with index $2n$ and the second one contains index with $2n+1$. In this arrangement, each functional unit possesses a block of X bank and a block of Y bank. The functional units only need to access data from the local blocks, and there is no need to access data across other blocks. As a result, each block is only required to provide one read port for the functional unit.

t	FU0				FU1			
0	mul	P_0	Q_n		mul	P_1	$Q_{n\pm 1}$	
1	mac	P_2	$Q_{n\pm 2}$	Acc_0	mac	P_3	$Q_{n\pm 3}$	Acc_1
2	mac	P_4	$Q_{n\pm 4}$	Acc_0	mac	P_5	$Q_{n\pm 5}$	Acc_1
3	mac	P_6	$Q_{n\pm 6}$	Acc_0	mac	P_7	$Q_{n\pm 7}$	Acc_1
4	add	Acc_0	Acc_1					

Table 3.3: Exploiting Data Parallelism by Observing Data Access Pattern

The corresponding structure of the register file is shown in Figure 3.6. $P(n)$ and $Q(n)$ are supposed to be stored in X and Y banks separately. This structure illustrates a way to assembly an four read ports register file with four individual local read port register block. It is easy to scale up the architecture for further

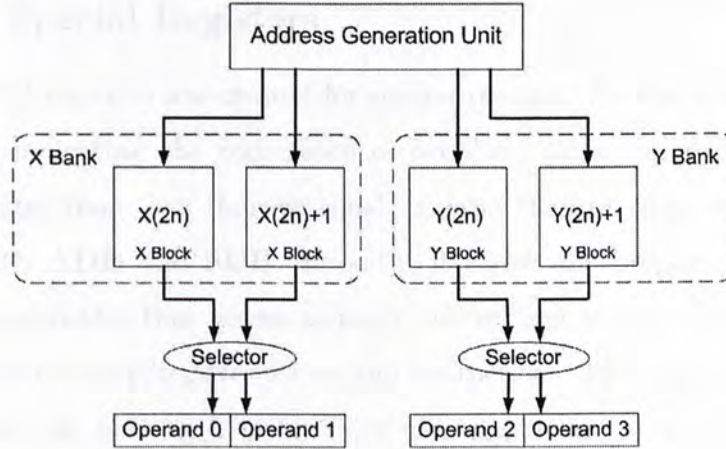


Figure 3.6: The Structure of Register File for Data Parallelism

exploiting data parallelism as shown in Figure 3.7. For J functional units and N data length, the index of X, Y block starts from $\frac{N}{J}n$, $\frac{N}{J}n + 1$, $\frac{N}{J}n + 2$, to the last index $\frac{N}{J}n + (\frac{N}{J} - 1)$, where N is the power of 2 and J is an even number. It is particularly suitable for computationally intensive DSP algorithms because many calculations can be done simultaneously, though using more hardware resources for rapid operations.

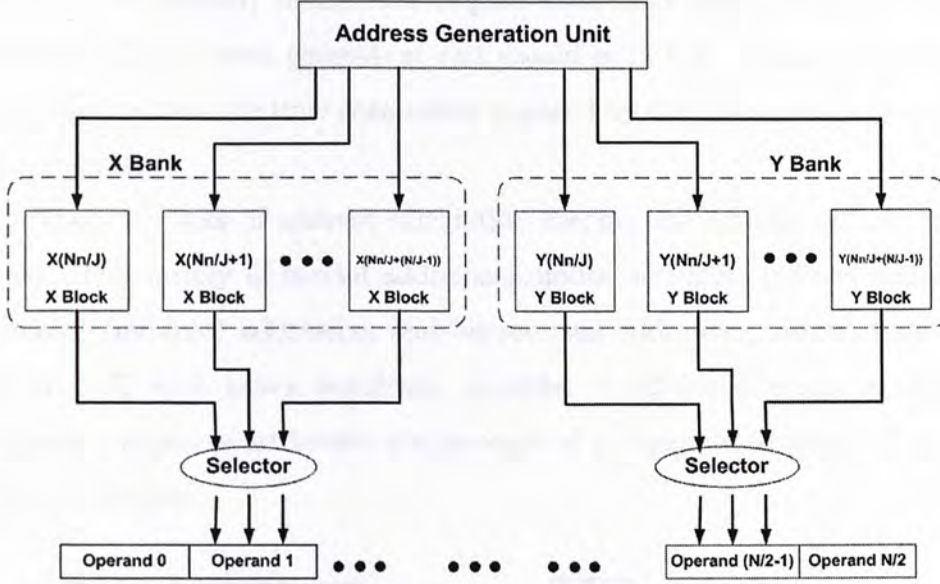


Figure 3.7: The Large-Scale View of Register File for Data Parallelism

3.4.3 Special Registers

The special registers are created for several reasons. Firstly, it marks the status flags indicating the occurrence of overflow, carry, equal/not equal, less than/greater than, less than or equal/ greater than or equal for instructions like **CMP**, **ADD** and **SUB**. Secondly, it allows the programmer to initialize the parameters that access memory and register through Load Store Unit (LSU). For memory/register access and instructions like **LOAD** and **STORE**, parameters like memory/register start address, size and step can be initialized at the beginning of the code segment. It is also feasible to set up the specialized addressing mode (e.g. circular/bit-reversed) through the special registers. The organization of special registers provide the high degree of flexibility. It is applicable to base instructions as well as parallel instructions. The details of special registers are shown in Appendix B.

3.4.4 Address Generation

The function of the address generation unit is to provide the addresses of the operands required to carry out the DSP operations. Since many instructions,

such as the multiply instruction, require more than one operand for their execution. The address generation unit should work fast enough to provide the addresses within the time constraints imposed by the instruction execution requirements.

There are lots of address calculation keeping the address generation unit busy. The variety of special addressing modes, including indirect addressing, circular (modulo) addressing and bit-reversed addressing modes, burden the main ALU with heavy workload. In order to efficiently compute those addresses, a separate arithmetic unit is required to compute addresses in the DSP implementation.

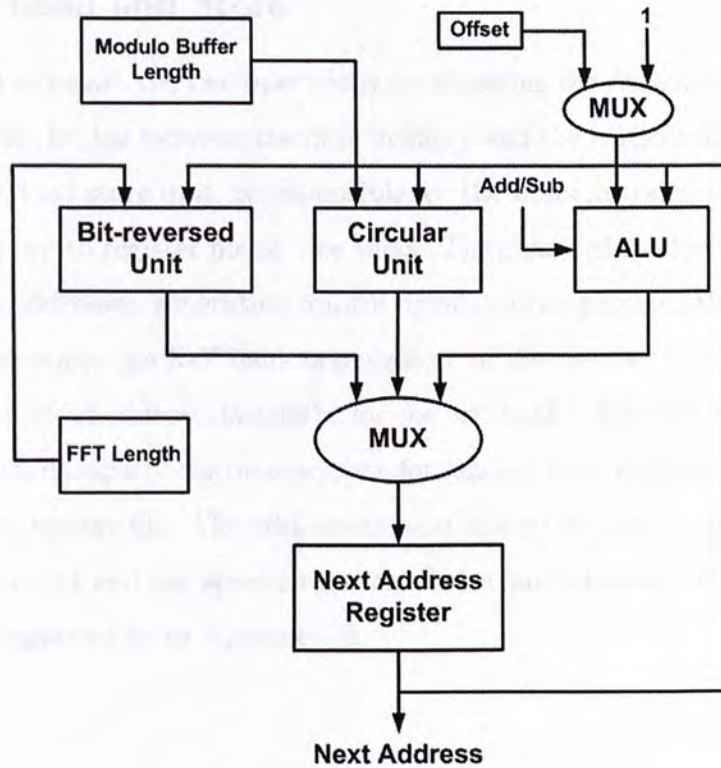


Figure 3.8: The Datapath of Address Generation Unit

The diagram of address generation unit is shown in Figure 3.8. It typically involves the following operations :

1. Getting the immediate address from a register, memory location or instruction operand.

2. Adding or subtracting the current address by 1 for program counter to fetch the instruction from memory.
3. Incrementing or decrementing the current address by an offset for jump, loop operations and subroutine calls.
4. Generating new address by applying circular addressing algorithm to handle a continuous stream of incoming data samples.
5. Generating new address by applying bit-reversed addressing mode to implement some DSP algorithms like the Fast Fourier Transform (FFT).

3.4.5 Load and Store

Load and store are the two operations for accessing the data memory, and they are the only bridge between the data memory and the register file. A dedicated hardware, load store unit, is responsible for the tasks in transferring data from data memory to register file or vice versa. The duties of the load store unit are providing addresses, generating control signals and dispatching the fetched data. To accommodate the X-Y bank organization of the memory, the load store unit has independent address datapaths for the two banks. For each bank, there are one address datapaths for memory, one for reading from register file and one for writing to register file. The address datapaths are the same as those in address generation unit and the special registers that attached to the address datapath are also organized as in Appendix B.

Chapter 4

Implementation of Speech Recognition on ASIP

4.1 Hardware Architecture Exploration

Application Specific Instruction Set Processor (ASIP) differs from general DSP Processor because ASIP targets at a specific class of application. ASIP involves thorough analysis of the algorithm in the hope of optimizing both the structure of datapath and application-oriented instruction set. Different techniques will be discussed in the following section for speaker-independent isolated word recognition system.

4.1.1 Floating Point and Fixed Point

DSP algorithm implementations deal with signals and coefficients. To use a fixed-point DSP device efficiently, one must consider to represent feature coefficients using fixed-point 2's complement representation. Typically, the coefficients are fractional numbers. Floating-point numbers can provide wide dynamic range of numerical representation by normalizing all numbers into the same format with sign bit, exponent and mantissa. Though floating-point computation can provide calculation with a good precision, it suffers from speed degradation since it requires more execution time and more complex hardware to run the routine compared with the fixed-point computation. For example,

floating-point additions require the exponents to be normalized before the addition of the mantissas. While floating-point multiplications require addition of exponents besides the multiplication of mantissas.

Our ASIP is a 16-bit fixed-point processor, the only format allowed for each number is a 16-bit integer, which ranges from 0 to 65535 for unsigned number or -32768 to +32767 for signed number.

4.1.2 Multiplication and Accumulation

Multiply-accumulate (MAC) operations are useful features for matrix operations, such as convolution for filtering, dot product and even for polynomial evaluation. They are used to implement the mathematical function in the form of $A + BC$. It is important for most DSP applications which require the accumulation of the products of a series of successive multiplications. In our design, MAC consists of a 16 X 16 multiplier followed by the 40 bit adder/subtractor unit and an additional register called accumulator. In general, the product of

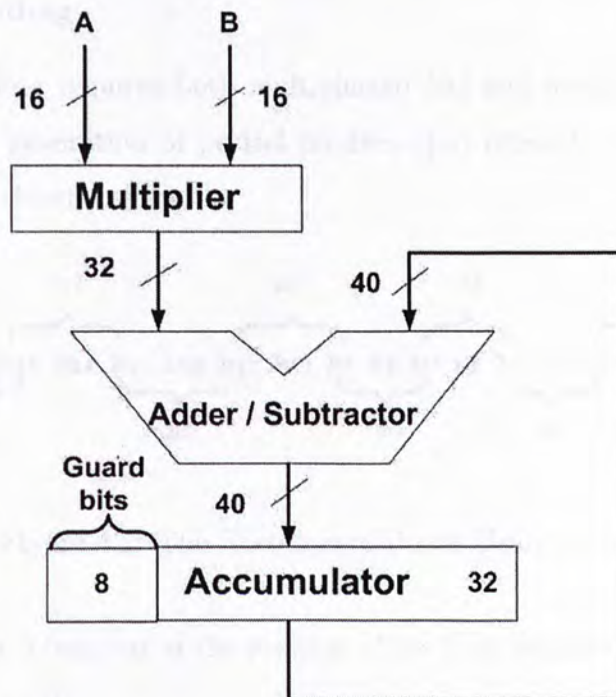


Figure 4.1: The Datapath of MAC unit

a 16 X 16 multiplication is 32 bits. The extra most significant 8 bits in the adder/subtractor unit and accumulator are guard bits. When repetitive MAC operations are performed, the accumulated sum grows with each MAC operation if the inputs of the multiplier are not normalized. This increases the number of bits required to represent the result without loss of accuracy. One way of handling this growth is to provide extra bits in the accumulator. These extra bits, the so-called guard bits, allow for the growth of the accumulated sum as more and more product terms are added up. The datapath of MAC unit is shown in Figure 4.1.

The most critical part of MAC unit is the multiplier. Efficient algorithm of multiplication [17] can enhance the computational speed remarkably. The multiplier is divided into three parts, namely the Booth encoding, Wallace tree and the addition. That means all partial products are first generated using Booth encoding, followed by blocks of 4:2 Wallace tree that compress the 4 inputs into 2 outputs (sum and carry) and the final 32-bit addition.

Booth Encoding

Booth Encoding requires both multiplicand (A) and multiplier (B) to be its inputs. The generation of partial product (pp) depends on the encoding of multiplier as shown belows :

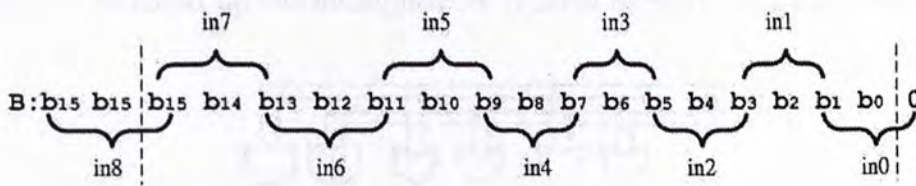
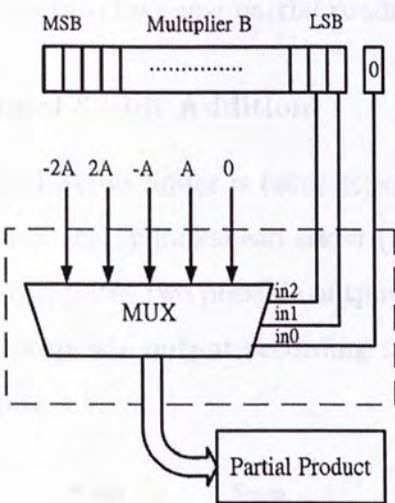


Figure 4.2: The Encoding of Booth Using Multiplier

1. An zero is inserted in the position of the least significant bit of multiplier B (i.e. bit 0).
2. Two sign extended bits are inserted in the position of the most significant bit of multiplier B (i.e. bit 15).

- 3. Consecutive three bits form one “word” and act as the input of Booth encoder.
- 4. Partial product (pp) is the output of Booth encoder which depends on the multiplicand A. The Booth encoding table is shown as follows :



in ₂	in ₁	in ₀	output (pp)
0	0	0	0
0	0	1	A
0	1	0	A
0	1	1	2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

Figure 4.3: The Booth Encoding Logic Table 4.1: The Booth Encoding Table

Wallace Tree Compressor

In this multiplier, both the Booth algorithm and Wallace tree array block [18] are used to speed up the multiplication process by enhancing parallelism. The

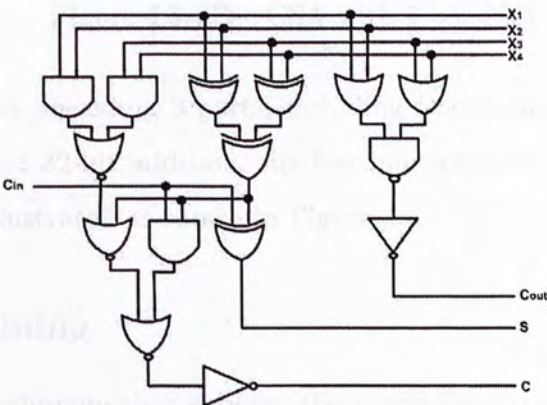


Figure 4.4: The Diagram of 4:2 Compressor

Wallace array block is made of numbers of 4:2 compressor, which means there are four inputs and two outputs in each compressor as shown in Figure 4.4. The inputs are partial products (X_1, X_2, X_3, X_4) and outputs are sum (S) and carry (C). The addition of the partial products uses the 4:2 compressor to sum up the partial products concurrently. In other words, it compresses four partial products into two new partial products (S, C) simultaneously.

The final 32-bit Addition

The final 32-bit adder is carry select adder (CSA) and it is constructed from a eight 4-bit carry lookahead adder (CLA) to propagate the carry at high speed. It pre-computes two possible outputs (sum) with carry=0 or carry=1 and select the appropriate output according to the carry. The structure of CSA is shown in Figure 4.5.

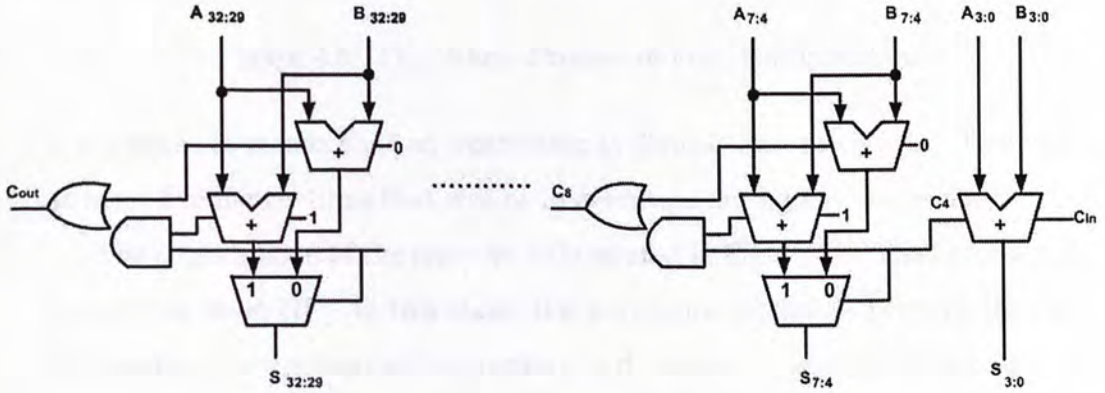


Figure 4.5: The CSA with 4-bit CLA

Combining the preceding 3 parts, including Booth encoder, Wallace array block and the final 32-bit addition, the fast multiplication process can be implemented and illustrated as shown in Figure 4.6.

4.1.3 Pipelining

Pipelining is a technique that exploits the parallelism among the instructions in sequential instructions. The platform is a typical pipelined processor with

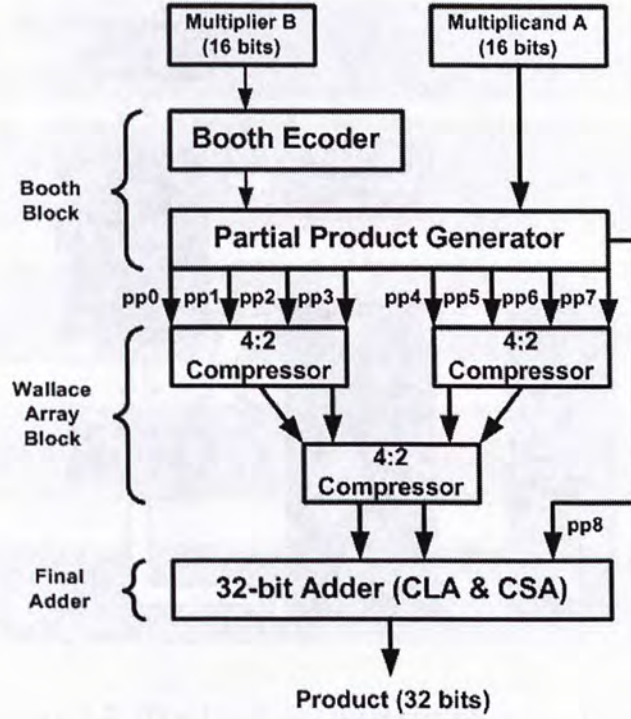


Figure 4.6: The Whole Process of Fast Multiplication

five stages. It means that an instruction is divided into five stages. There are at most five instructions that will be in execution during any single clock cycle.

The organization of the pipeline is illustrated in Figure 4.7. The first stage is instruction fetch (IF). In this stage, the instruction fetch unit provides instruction address to the instruction memory and fetches the corresponding instruction into the processor. Then the processor moves to decode stage (DEC). The fetched instruction is decoded into commands and operands. Meanwhile, the address generation unit calculates for operand address calculation. The third stage is read stage (RD). The major task is to read the operands from the register file. Similarly, the load store unit also accesses the register file for preparing store operation. Some instruction decoding works that related to datapath is completed in this stage. The forth stage is execution stage (EX). All the data processing, Boolean manipulation tasks are performed there. The load store unit accesses the data memory in this stage. The last stage is writeback (WB). The processed data is written back to the register file. On the other hand, the

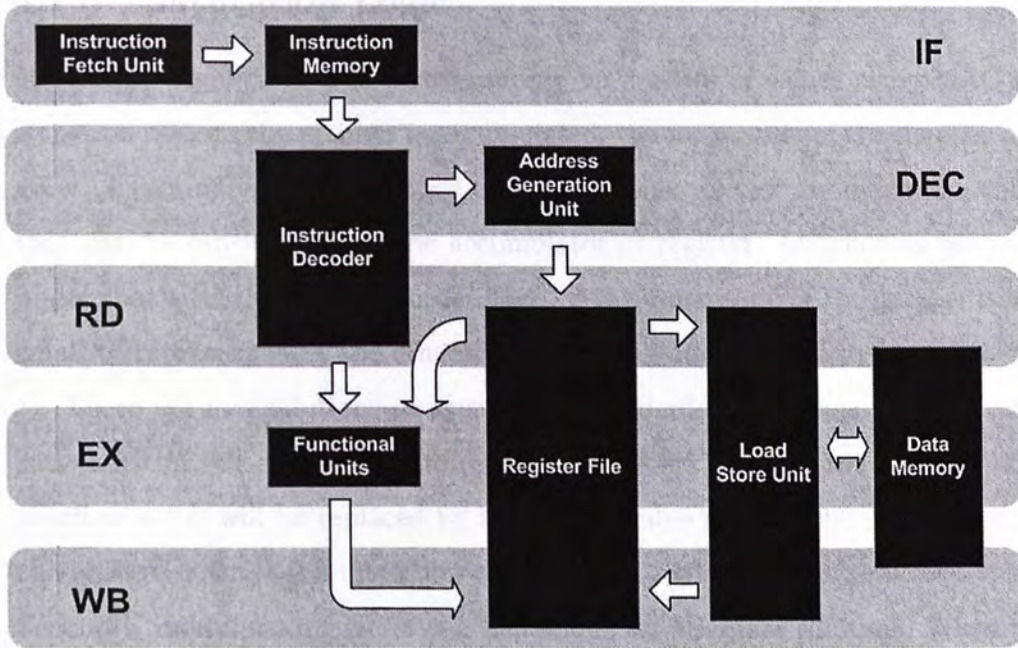


Figure 4.7: The Pipeline Organization of the Platform

load store unit puts the loaded data into the register file.

4.1.4 Memory Architecture

The simplest processor memory structure is a single bank of memory that contains single set of address and data lines. Both program instructions and data are stored in the single memory. It is called the Von Neumann architecture, which is common in most non-DSP processors.

Such memory architecture, however, is not sufficient to handle large amount of data with considerable access speed in DSP applications. Thus, our design adopts Harvard architecture, which holds program instructions and data separately in three distinct memory locations labeled as instruction ROM, X and Y data memory. The instructions are stored in the ROM, while feature vectors of the real-time incoming speech and the speech reference parameters are stored in X and Y data memory respectively.

4.1.5 Saturation Logic

Many DSP applications involve summing up a series of values using MAC instruction. When the number is accumulated, the magnitude of the number will grow. Eventually, the magnitude of the sum may exceed the maximum value that can be represented by the accumulator or register. Overflow is occurred under that situation. On the other hand, if the magnitude of the number is too small to represent it by the minimum value, underflow is said to occur.

There are two solutions to overflow and underflow problems. Firstly, saturation arithmetic can be used to represent numbers that are out of range. The overflow value will be replaced by the largest value that can be represented by the processor while the underflow value will be replaced by the smallest value. Secondly, modular arithmetic can be seen as an alternate solution. When the values of data lie out of the range of the largest and smallest representable numbers, these values are wrapped around into the range using modular arithmetic relative to the smallest representable number. Modular arithmetic is sometimes referred to 'clock arithmetic' for integers, where numbers 'wrap around' after they reach a certain value (the modulus). For example, while $8 + 6$ equals to 14 in conventional arithmetic, the actual answer is 2 if it is implemented in modulo 12 arithmetic. It is because 2 is the remainder after dividing 14 by the modulus 12. Thus, it is better to use saturation arithmetic instead of the modular arithmetic to avoid an error known as the wrap around error. The datapath of the saturation logic is shown in Figure 4.8. It is particularly useful in the parallel instruction to enhance the accuracy by eliminating the arithmetic right shift operation that scales down the values significantly.

4.1.6 Specialized Addressing Modes

Different addressing modes can be employed to speed up the DSP real-time implementation, namely circular addressing mode and bit-reversed addressing mode. They are used for various DSP algorithms like filtering or Fast Fourier Transform (FFT), which require the large amount of data to be handled as well

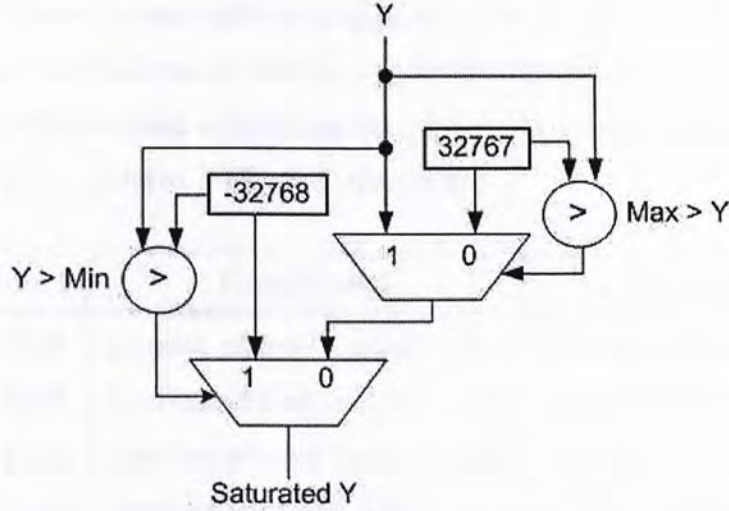


Figure 4.8: The Datapath of Saturation Logic

as a fixed address index pattern can be observed. The following sections will have a detailed discussion on these specialized addressing modes.

Circular Addressing Mode

The circular addressing mode is usually used to handle a continuous stream of incoming data samples (coefficients). There is a buffer that act as a storage place for those samples. Usually the length of buffer depends on the number of samples. The more the samples, the larger the buffer. Sometimes, it wastes lots of memory saving numerous coefficients without further re-using it again when those stored data are no longer needed afterwards. To reserve the tight memory for other purposes, it is better to keep the data in a circular buffer instead of a general buffer. In a circular buffer, successive data samples are stored in sequential buffer locations until the end of the buffer is reached. Then the next incoming data will be saved at the beginning of the buffer once the previous data is stored in the end of the buffer. The actual operation of the circular addressing mode can be represented in a mathematical form

$$\text{next address} = (\text{current address} \pm \text{step}) \% \text{size}$$

where step is the movement range of the pointer (PTR) which holds current address and size is the buffer length. The PTR can be incremented or decre-

mented. There are two additional registers to mark the position of the start address and end address of the circular buffer. They are called start address register (SAR) and end address register (EAR). There are total four cases for calculating the updated PTR of circular buffer :

Condition 1	Condition 2	Updated PTR
$SAR < EAR$	$(\text{current address} + \text{step}) > \text{size}$	$(\text{current address} + \text{step}) - \text{size}$
$SAR < EAR$	$(\text{current address} + \text{step}) < \text{size}$	$(\text{current address} + \text{step}) + \text{size}$
$SAR > EAR$	$(\text{current address} - \text{step}) > \text{size}$	$(\text{current address} - \text{step}) - \text{size}$
$SAR > EAR$	$(\text{current address} - \text{step}) < \text{size}$	$(\text{current address} - \text{step}) + \text{size}$

Table 4.2: The Pointer Update Algorithm of Circular Addressing Mode

Bit-reversed Addressing Mode

Special data access capability is important in the Fast Fourier Transform (FFT) algorithm implementation. The FFT is a fast algorithm that transforms a time-domain signal into its frequency-domain representation. However, there is a drawback of FFT operation. It takes the input in a natural order, but results in an irregular outputs shown in Figure 4.9. Note that the bit-reversed pattern is just a mirror reflection of the original input pattern.

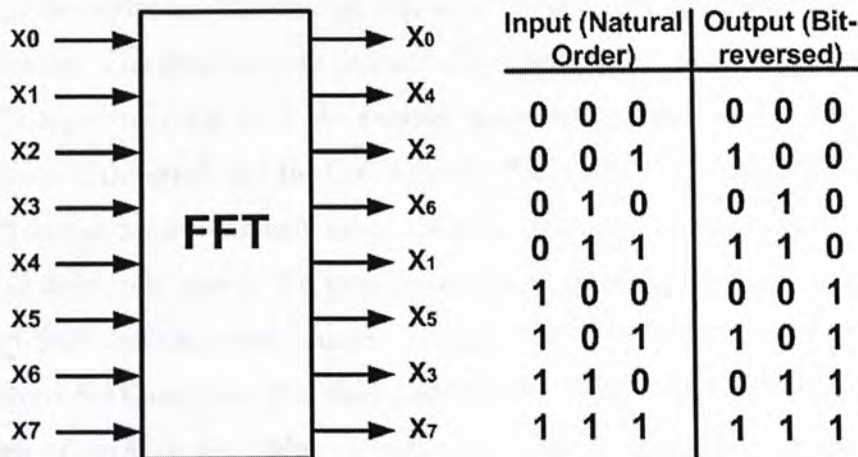
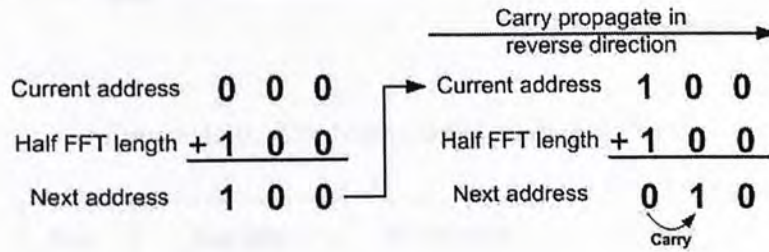


Figure 4.9: The Output of FFT Algorithm

Fortunately, there is still a traceable rule to generate the bit-reversed pattern. From Figure 4.9, the length of FFT is 8. The actual operation of the bit-reversed addressing mode can be represented in a mathematical form

$$next\ address = (current\ address + \frac{1}{2} (FFT\ Length))$$

In the example, the start address is 0 and half of FFT length is $8/2=4$, thus next address is equal to current address $000 + \text{half FFT length } 100 = 100$ with no carry generated. The next address becomes current address. However, current address $100 + \text{half FFT length } 100$ does produce a carry. Please notice that the carry is propagated in the reverse direction (i.e. from right to left) as shown belows.



4.1.7 Repetitive Operation

Loops are complicated tasks for instruction fetching. A dedicated controller is used to maintain the current status of a loop and to handle the address calculation. The structure of the controller is shown in Figure 4.10. The internal control logic interacts with the request from instruction decoder, controls the operation of the stack and the loop counter. The status of the loop is temporarily stored in the stack. The content of the stack is shown in Figure 4.11. The first one bit field indicates if the loop operation is currently running or not. The second field indicates the number of lines of instructions covered by a loop. The third field indicates the start position of a loop. The forth field states the number of iteration left. When a static loop is set up, the current status and the setup data of the loop (start address and size) are pushed into the stack and the loop tag is set to one. The number of iterations is stored in the loop counter.

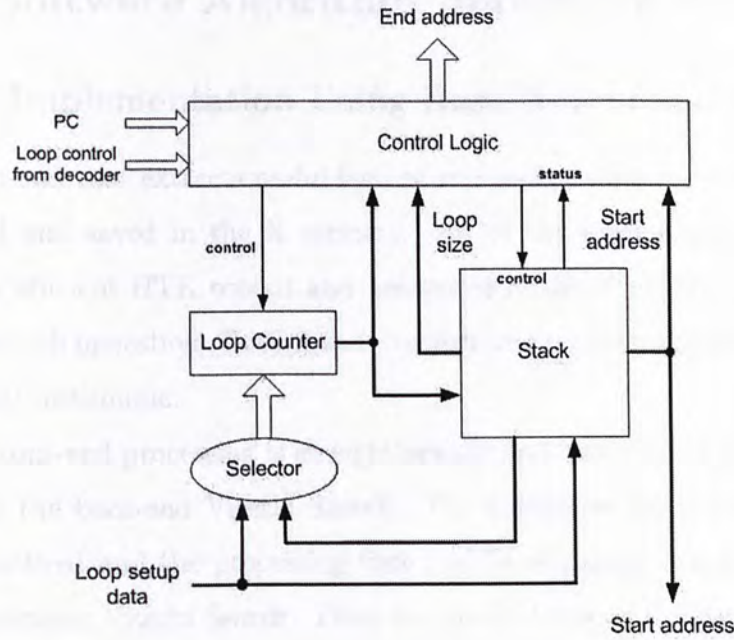


Figure 4.10: The Structure of Loop Controller

loop	loop size	start address	iteration
------	-----------	---------------	-----------

Figure 4.11: The Content of Stack

Based on the setup data, the control logic can figure out the end position of the loop and the current relative position of the program in the loop. At the end of each iteration, the loop counter is decreased by one, and the program counter is updated with the start address of the loop. Once the loop counter reaches zero, the stored loop status is popped out and the previous status can be maintained.

Sometimes, the application maybe so complicated that multiple nested levels of static loops are required. The total number of levels depends on the number of entries of the stack. Hence the size of the stack should be considered in application analysis in order to match the behaviour of the target application.

4.2 Software Algorithm Implementation

4.2.1 Implementation Using Base Instruction Set

The front-end that extracts useful feature vectors from incoming speech is pre-processed and saved in the X memory. All of the word models are trained offline by efficient HTK toolkit and pre-stored in the Y memory for back-end Viterbi search operation. Both feature vectors and model parameters are 16-bit fixed-point arithmetic.

The front-end processing is straightforward and less computationally intensive than the back-end Viterbi Search. The procedure for feature extraction is standardized and the processing time can be negligible compared with the time-consuming Viterbi Search. Thus, we mainly focus on the back-end Viterbi search in our project.

Recall from Section 2.3.3 and 2.4, the simplest implementation is single-mixture Viterbi search. The recognition result is found by calculating the output probability density function recursively

$$\begin{aligned} b_j(\vec{o}_t) &= \hat{N}(\vec{o}_t, \vec{\mu}_j, U_j) \quad , \quad 1 \leq j \leq N, \quad 2 \leq t \leq T \\ &= \frac{1}{\sqrt{(2\pi)^D \det(U_j)}} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1} (\vec{o}_t - \vec{\mu}_j)} \end{aligned} \quad (4.1)$$

$$\begin{aligned} \delta_t(j) &= \left[\max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} \right] \cdot b_j(\vec{o}_t) \\ &= \left[\max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} \right] \cdot \frac{1}{\sqrt{(2\pi)^D \det(U_j)}} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1} (\vec{o}_t - \vec{\mu}_j)} \end{aligned} \quad (4.2)$$

where \vec{o}_t is the observation vector with the dimensionality of D at time t , and \hat{N} is a Gaussian pdf with mean vector $\vec{\mu}_j$ and the determinant of the covariance matrix U_j in state j .

The probability value is small and further multiplications lead to an intermediate result which is so small that the computer cannot represent it accurately. Underflow is expected under this situation. To solve the problem, the logarithm

of probability is taken (i.e. natural log). The Equation 4.2 can be re-written as

$$\begin{aligned}
 \tilde{\delta}_t(j) = \ln[\delta_t(j)] &= \ln[\max\{\delta_{t-1}(i) \cdot a_{ii}, \delta_{t-1}(i) \cdot a_{ij}\} \cdot b_j(\vec{o}_t)] \\
 &= \max[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}] + \tilde{b}_j(\vec{o}_t) \\
 &= \max[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}] + \ln[\hat{N}(\vec{o}_t, \vec{\mu}_j, U_j)] \\
 &= \max[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}] + \ln\left[\frac{1}{\sqrt{(2\pi)^D \det(U_j)}}\right] \\
 &\quad + \left(-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1}(\vec{o}_t - \vec{\mu}_j)\right)
 \end{aligned} \tag{4.3}$$

From Equation 4.3, the first term $\max[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}]$ is the selection of higher probability from two possible search paths (stay in the same state or advance to next state). $\tilde{\delta}_{t-1}(i)$ is calculated previously because the search is a recursive process. Both \tilde{a}_{ii} and \tilde{a}_{ij} are model parameters that can be pre-stored in the memory. The second term is simply a constant since vector size D and covariance U_j are known before the implementation. As there is growing concern in the computation efficiency of different arithmetic operations, it is realized that addition and multiplication compute faster than division in most hardware platform. To avoid division, we need to pre-compute the inverted data. We can pre-calculate and pre-store the constant $\frac{1}{\sqrt{(2\pi)^D |U_j|}}$ instead of $\sqrt{(2\pi)^D |U_j|}$ for the Gaussian probability computation. The third term $\left(-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1}(\vec{o}_t - \vec{\mu}_j)\right)$ are composed of 1 X 26 \vec{o}_t , 1 X 26 $\vec{\mu}_j$ and 1 X 26 U_j^{-1} as there are 26 elements ($D=26$) for each parameter that forms a vector. Thus, the third term of Equation 4.3 can be analyzed in the following way :

$$\begin{aligned}
 &-\frac{1}{2}(\vec{o}_t - \vec{\mu}_j)^T U_j^{-1}(\vec{o}_t - \vec{\mu}_j) \\
 &= -\frac{1}{2} \begin{bmatrix} o_{t1} - \bar{\mu}_{j1} \\ o_{t2} - \bar{\mu}_{j2} \\ o_{t3} - \bar{\mu}_{j3} \\ \dots \\ o_{t26} - \bar{\mu}_{j26} \end{bmatrix} \begin{bmatrix} \frac{1}{U_{j1}} & \frac{1}{U_{j2}} & \dots & \frac{1}{U_{j26}} \end{bmatrix} [o_{t1} - \bar{\mu}_{j1} \ o_{t2} - \bar{\mu}_{j2} \dots o_{t26} - \bar{\mu}_{j26}]
 \end{aligned}$$

$$= \sum_{i=1}^{26} (o_{ti} - \bar{\mu}_{ji})^2 \left(-\frac{1}{2U_{ji}} \right)$$

The complicated matrix form of the third term can actually be viewed from a simple perspective using subtraction, multiplication and addition. Since the equation displays the accumulated summation property, MAC instruction can be used to speed up the operations. Remember that all the arithmetic operations supply the input through the registers. The efficiency of operation can be further enhanced if specific registers are assigned for storing the input values. The address of registers can be generated (incremented by 1) automatically through the address generation unit, assuming that the address of registers align in a sequential order. It bypasses the datapath of the ALU unit because there is no need to have any addition to find out the next updated address. To reduce the number of used registers, circular addressing mode can be utilized to override the previous loaded input data by the next incoming ones. It is feasible to do so as the vector size is fixed (26 parameters) for each speech frame.

To model the wide variability of speech appropriately, we can consider implementing the speech recognition process with double mixture Viterbi search. Recall from Section 2.3.3, the output probability density function with double mixture is defined as

$$b_j(\vec{o}_t) = \sum_{k=1}^M c_{jk} \hat{N}(\vec{o}_t, \vec{\mu}_{jk}, U_{jk}) = \sum_{k=1}^M c_{jk} \frac{1}{(2\pi)^D \det(U_{jk})} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_{jk})^T U_{jk}^{-1} (\vec{o}_t - \vec{\mu}_{jk})} \quad (4.4)$$

where c_{jk} is the mixture coefficient for the k th mixture in state j , M is the number of mixtures per state and \hat{N} is a multivariate Gaussian with mean vector $\vec{\mu}_{jk}$ and the determinant of the covariance matrix U_{jk} for the k th mixture component in state j . To have a simple expression, rewrite equation (4.1) as

$$\hat{N} = B e^J \quad (4.5)$$

Thus equation (4.4) can be expressed as

$$\begin{aligned} b_j(\vec{o}_t) &= \sum_{k=1}^M c_k B_k e^{J_k} = \sum_{k=1}^M G_k e^{J_k}, \quad G_k = c_k B_k \\ \ln[b_j(\vec{o}_t)] &= \ln \left(\sum_{k=1}^M G_k e^{J_k} \right) \end{aligned} \quad (4.6)$$

where G_k is constant for the k th mixture of each state and can be pre-calculated and pre-stored in ROM. A simple and accurate method can be developed for the log-add operation ($\ln \sum \exp$) in equation (4.6). As we use a double-mixture Gaussian model ($M=2$) throughout our implementation, select the larger $G_{k \max} e^{J_{k \max}}$ in equation (4.6) and divide each operand of add operation by it

$$\ln[b_j(\vec{o}_t)] = \ln G_{k \max} + J_{k \max} + \ln \left\{ \left(1 + \sum_{k=1, k \neq k \max}^M \frac{G_{k \min}}{G_{k \max}} e^{J_{k \min} - J_{k \max}} \right) \right\}$$

Since $\frac{G_{k \min}}{G_{k \max}} e^{J_{k \min} - J_{k \max}}$ is always kept less than one, $\left(1 + \sum_{k=1, k \neq k \max}^M \frac{G_{k \min}}{G_{k \max}} e^{J_{k \min} - J_{k \max}} \right)$ is less than M . This makes $\left(1 + \sum_{k=1, k \neq k \max}^M \frac{G_{k \min}}{G_{k \max}} e^{J_{k \min} - J_{k \max}} \right)$ lie in a finite region which is always greater than one but less than two. It is recommended to ignore this complicated term for easier implementation since its value is small and the removal of this term will not degrade the recognition accuracy significantly. The completed equation that implements the Viterbi search [19] is simplified as

$$\begin{aligned} \tilde{\delta}_t(j) = \ln[\delta_t(j)] &= \max \left[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij} \right] + \tilde{b}_j(\vec{o}_t) \\ &= \max \left[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij} \right] + \ln G_{k \max} + J_{k \max} \\ &= \max \left[\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij} \right] + \ln \left[c_{k \max} \frac{1}{\sqrt{(2\pi)^{26} \det(U_j)}} \right] \\ &\quad + \sum_{i=1}^{26} (o_{ti} - \bar{\mu}_{ji})^2 \left(-\frac{1}{2U_{ji}} \right) \end{aligned} \quad (4.7)$$

To have a systematic way of viewing the implementation of speech recognition, the whole program flow can be illustrated as follows:

1. Initialize registers for storing the address of look-up table of various model parameters, including mean, variance, transition probability, etc.
2. Compute $J_{k \max}$ i.e., $\sum_{i=1}^{26} (o_{ti} - \bar{\mu}_{ji})^2 \left(-\frac{1}{2U_{ji}} \right)$. Subtraction is performed first, followed by the multiplication and arithmetic right shift. The right shift can prevent the overflow of the intermediate values. MAC instruction is finally executed for the accumulated sum and multiplication.

3. Compute $\ln[G_{k \max}]$ i.e., $\ln[c_{k \max}] + \ln\left[\frac{1}{\sqrt{(2\pi)^{26} \det(U_j)}}\right]$. Only one addition is required since both $c_{k \max}$ and U_j are constants, which means the logarithm of those two terms can be pre-calculated and pre-stored in the memory.
4. Calculate $\left[\max\{\tilde{\delta}_{t-1}(i) + \tilde{a}_{ii}, \tilde{\delta}_{t-1}(i) + \tilde{a}_{ij}\}\right]$. The term $\tilde{\delta}_{t-1}(i)$ can be obtained from previous step while \tilde{a}_{ii} and \tilde{a}_{ij} are constants. There are totally two additions plus one selection of the path with higher output probability.
5. Repeat procedures 2 ~ 4 until all speech frames are processed and all word models are compared with the feature vectors of the incoming speech. LOOP instruction can be used for repetitive tasks.

To enhance the recognition speed, multiple-stage pipeline discussed in Section 4.1.3 can be applied to speech recognizer. Given that the speech parameters shown in Table 4.3, the design takes 6 cycles to calculate one output probability

Speech Parameters	Value
Number of Words	11
Number of Frames	190
Number of States	8
Number of Mixtures	2
Feature Vector Size	26

Table 4.3: The Speech Parameters for Recognition

δ , there are 4 cycles for two consecutive multiplications and other operations like reading constants from memories, subtraction, shift, addition, comparison and selection are all executed within that 6 cycles. Thus it requires $6 \times 11 \times 190 \times 8 \times 2 \times 26 \approx 5 \times 10^6$ clock cycles to recognize one word. It implies that it is possible to recognize one word in 1 second if the operating frequency is 5 MHz.

4.2.2 Implementation Using Refined Instruction Set

The refinement of software implementation is based on the instruction-level design enhancement. There are two methods of optimization derived from the pre-defined base instruction set, mainly distillation and condensation.

Distillation corresponds to the process of eliminating the infrequently used pre-defined instructions. This can save the special hardware resources implementing that pre-defined instruction by replacing it with a sequence of instructions. On the contrary, condensation is the process of replacing a frequently used instruction sequence by a newly defined application specific instruction. Extra hardware deployment is expected to have a perfect match between the architecture and the refined instruction set. Figure 4.12 shows an example of

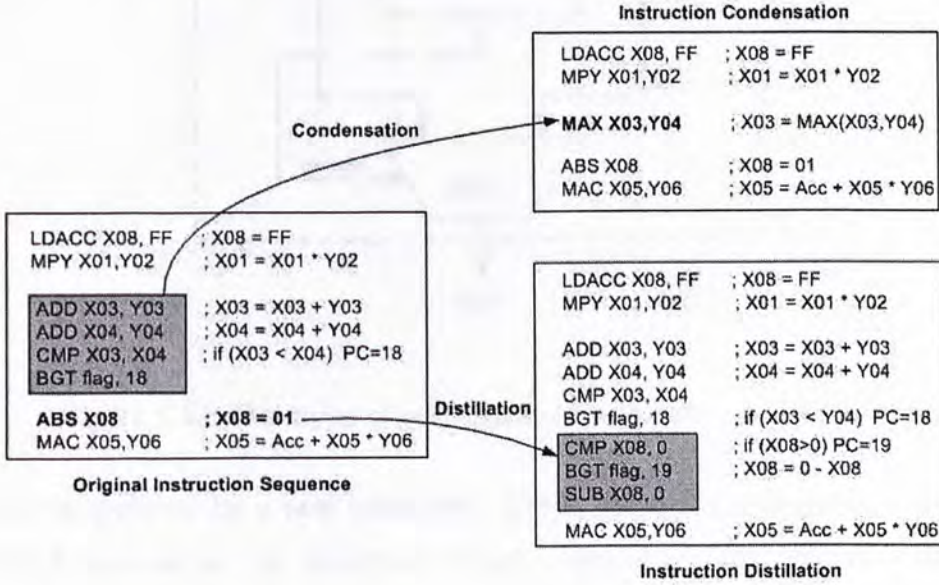


Figure 4.12: Examples of Instruction Condensation and Distillation

instruction distillation and instruction condensation [9]. In our project, pre-defined instructions such as **ABS**, **NORM** and **EXP** are seldom used. Those instructions are transformed into a new instruction sequence by distillation, i.e., substitution of the instruction **ABS** (absolute value) by a three-instruction sequence. It takes more clock cycles to run while lowering the hardware costs. Conversely, the repetitive and frequent operation occurs in the implementation

of add/compare selection of the Viterbi search process. Only the path with the highest output probability will be considered. Additional hardware functional unit called Add, Compare, Select Unit (ACSU) [20] is particularly designed for the sake of optimized implementation of instruction **MAX**. The structure of ACSU is shown in Figure 4.13. It can be seen that an instruction sequence

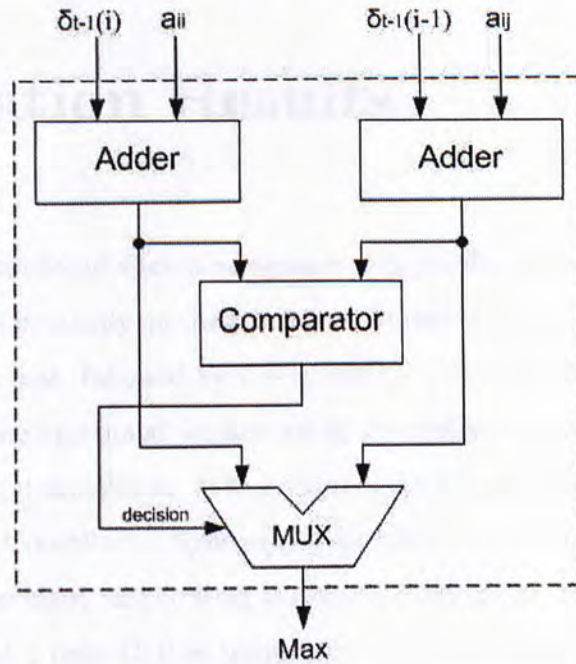


Figure 4.13: Examples of Instruction Condensation and Distillation

can be replaced by a new instruction **MAX** through condensation. Without **MAX** instruction, the inefficient branch action is required to access different code segments after the comparison of two register values. The register that holds the larger value is selected and the corresponding code segments will be executed. The use of **MAX** instruction speeds up the Viterbi search procedure. It takes fewer clock cycles by eliminating the overhead of branch process. It is possible to recognize one word in 0.5 second if the operating frequency is 5 MHz. Compared with the base instruction set, the recognition time is reduced to half of the original one if refined instruction set is used at the same operating frequency.

Chapter 5

Simulation Results

A speaker-independent speech recognizer with double-mixture HMM is devised. Its major focus is mainly on the Cantonese isolated words. The design specification is written first, followed by the modeling of Verilog HDL using behavioral description. The functional verification of the design is performed under simulation environment SimVision. If there is no error found, the design is synthesized in the Design Compiler of Synopsys. The physical design like floor planning, automatic placement and routing is done in Silicon Ensemble with AMS 0.35-micron 4 metal 2 poly CMOS technology. The specification of fabricated chip is depicted in Table 5.1 while the chip microphotograph is shown in Appendix C.

Specification	Value
CMOS Technology	0.35 μ m
Area (NAND2 equ.)	132K
Area (μ m \times μ m)	2600 \times 2600
Number of Pads	120
Operating Voltage	3.3V

Table 5.1: The Specification of Fabricated Chip

The whole speech recognizer is implemented by writing programs based on the base instruction set, which is defined by the platform. Figure 5.1 outlines the simplified program skeleton. The main theme of Viterbi algorithm is to

find the maximum likelihood of the word model that matches the incoming speech. The details of program flow are also discussed in Section 4.2.1. By first

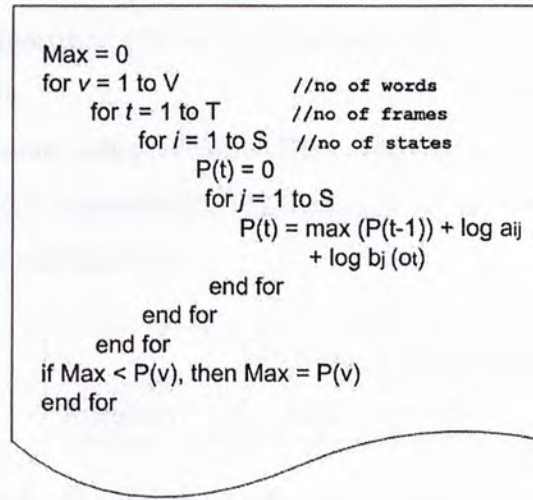


Figure 5.1: The Program Skeleton of Viterbi Search

implementing the algorithm in high level language (HLL), it is easy to convert it to the ASIP platform using completed base instruction set in Appendix A.

To verify the ASIP chip, a PCB board is made in Appendix D. Figure 5.2 is a brief diagram of the PCB. The program is preloaded in the ROM. The X

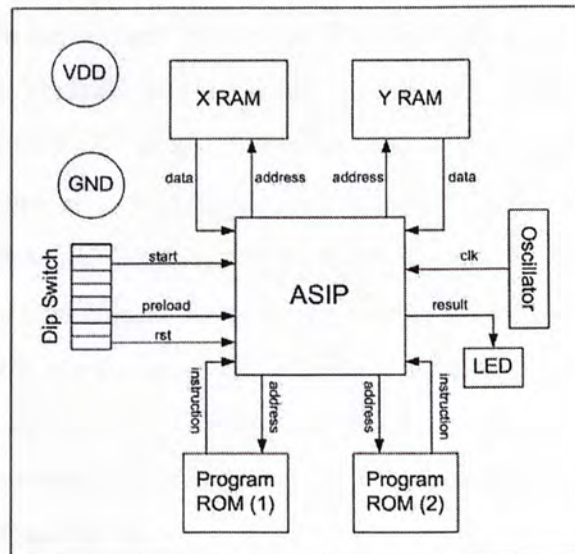


Figure 5.2: The Simplified Diagram of PCB

RAM and Y RAM store the extracted feature vectors and trained parameters of word models respectively. When the chip is power-on, the start signal indicates the beginning of the recognition process. The code segments describing the Viterbi search algorithm will be executed sequentially. The LED displays the recognition result.

Simulation results using software (HTK) [21] and our hardware platform are shown in Table 5.2 respectively. The accuracy of hardware simulation is only 0.7 % lower than software one.

	Software	Hardware
Accuracy(%)	93.9	93.2

Table 5.2: The Simulation Results of Recognition Accuracy

In our speech recognition system, the sampling rate of the speech is 8 kHz, the frame period is 20 ms and frame rate is 10 ms. The vocabulary in our experiment contains 11 Cantonese words. Each word has two syllables. Training data include 2200 utterances from 5 male and 5 female native speakers. Each of the word is modeled by an HMM, which are trained offline by the HTK toolkit. 734 utterances from another 10 male and 10 female native speakers are used for performance evaluation. Both training and testing utterances are recorded via microphone channel under the similar acoustic environment with signal-to-noise ratio (SNR) of 10 dB. The time required to complete one recognition is about 1 second at the working frequency of 5 MHz. The recognition time can be reduced to 0.5 second if refined instruction is used. Profiling of time information can be achieved based on the calculation in Section 4.2.1. According to the simulation results from Static Timing Analysis (STA) using PrimeTime, the maximum operation frequency can reach 86 MHz. This allows the chip to recognize one word in only 47 ms. The recognition time is fast enough in real-time implementation.

Compared with the past research work [22], it takes 60 seconds for their chip to recognize the word with working frequency of 17 MHz. Their vocabulary is

moderately-sized with around 60 monosyllable words and the technology for fabricated chip is CMOS. It is possible for our design to extend the vocabulary size by occupying more memory. The recognition time is approximately 6 seconds for 60 words. This is at least one magnitude faster than the previous work. Further refinement reduces this speed by a half. Moreover, that previous work operates its chip at a higher working frequency. It implies that more power is expected to be consumed. Our proposed platform shows great improvement in terms of execution time and power consumption. The achievement is particularly important in real-time applications.

Conclusions

Chapter 6

Conclusions and Future Work

Conclusions

This work aims at converting the conventional speech recognition algorithm from a research engine to a practical real-time system on ASIP platform. It targets at the isolated word recognition on an arbitrary moderately-sized vocabulary. The proposed ASIP design methodology is proven to be effective in the practical implementation. The time required to complete one word recognition is about 1 second at the working frequency of 5 MHz. The recognition time can be reduced to half of the original one if refined instruction is used. As our platform can reach the maximum operation frequency 86 MHz, this allows the chip to recognize one word in only 47 ms. The recognition time is fast enough in real-time speech applications. On the other hand, the proposed speech recognition running on ASIP platform attains approximately the same recognition accuracy as the software recognition. It is obvious that our design can meet the stringent requirement of both time-critical and highly accurate speech application. Most importantly, our research demonstrates the beauty of ASIP methodology by having software/hardware co-design. The speech algorithm is thoroughly analyzed in order to convert the complicated algorithm into simple mathematical form. Special hardware Add-Compare-Select Unit(ACSU) is deployed for the frequently repetitive Viterbi search. Finally, application-specific instruction set is exploited to bridge the gap between the software and hardware

parts. It is believed that the ASIP approach requires multi-disciplined expertise with background in digital signal processing, software engineering practice, logic and arithmetic design as well as computer architectures.

Future Work

The ASIP design provides a framework of implementing DSP algorithm efficiently. The research mainly focuses on the double mixture HMM-based speech recognizer. The recognition accuracy can be enhanced further if higher-order mixture is considered. However, the computation will be increased accordingly. This arouses the issue of scaling up the base platform to meet more stringent requirement. Scaling up the design is not a difficult task because the original platform can be extended to a more complicated one. More powerful instruction sets can be developed in parallel and complex forms. By selecting the useful instructions from the base one, the mixed utilization of originally selected base, newly defined parallel and complex instructions can show tremendous improvement in the demanding application.

In addition to higher-order mixture, there are many other speech algorithms that can be executed, including speaker identification, speech verification, speech synthesis for text-to-speech, etc. Since our ASIP platform has different instruction sets, it is feasible for developer to shift from one application to another with programmable instruction set. They can also define another set of instructions that are specific to their application. In this way, the ASIP design is flexible and powerful enough to have multiple applications.

Appendix A

Base Instruction Set

Mnemonic	Input \rightarrow Output	Description
MAC	(Reg, Reg, Acc) \rightarrow Acc	Multiply two values of registers and accumulate
MPY	(Reg, Reg) \rightarrow Acc	Multiply two values of registers
ADD	(Reg, Reg) \rightarrow Acc	Add two values of registers together
SUB	(Reg, Reg) \rightarrow Acc	Subtract one value of registers from another one
ADDC	(Reg, Reg, Flag) \rightarrow Acc	Add two values of registers together with carry
SUBB	(Reg, Reg, Flag) \rightarrow Acc	Subtract one value of registers from another one with borrow
ADDA	(Reg, Offset, Acc) \rightarrow Acc	Add an offset-able value of register to accumulator.
SUBA	(Reg, Offset, Acc) \rightarrow Acc	Subtract an offset-able value of register from accumulator.
NEG	Acc \rightarrow Acc	Invert the sign of the accumulator.
ABS	Acc \rightarrow Acc	Take the absolute value of the accumulator.
EXP	Acc \rightarrow SReg	Determine the exponent of the accumulator.
NORM	(Acc, SReg) \rightarrow Acc	Normalize the accumulator to the exponent stored in the special register.
SH	(Acc, Reg) \rightarrow Acc	Shift the accumulator with the signed value in a register (+ left, - right).
SHK	(Acc, Value) \rightarrow Acc	Shift the accumulator with the immediate signed value (+ left, - right)

Table A.1: The Data Processing Instructions

Mnemonic	Input → Output	Description
NOT	Acc → Acc	Bitwise NOT of the accumulator
OR	(Reg, Offset, Acc) → Acc	Bitwise OR of the accumulator with an offset-able value from register.
AND	(Reg, Offset, Acc) → Acc	Bitwise AND of the accumulator with an offset-able value from register.
XOR	(Reg, Offset, Acc) → Acc	Bitwise XOR of the accumulator with an offset-able value from register.

Table A.2: The Bit Manipulation Instructions

Mnemonic	Input → Output	Description
BEQ	Flag → PC	Branch if equal to flag is asserted.
BNE	Flag → PC	Branch if not equal to flag is asserted.
BLT	Flag → PC	Branch if less than flag is asserted.
BGT	Flag → PC	Branch if greater than flag is asserted.
BLE	Flag → PC	Branch if less or equal to flag is asserted.
BGE	Flag → PC	Branch if greater or equal to flag is asserted.
SR	Value → (PC, stack)	Subroutine call
JP	Value → PC	Unconditional jump
LOOP	(size, cycle) → (PC, stack)	Static looping
RET	stack → PC	Return from subroutine call or break a static loop.
NOP	NA	No operation

Table A.3: The Flow Control Instructions

Mnemonic	Input → Output	Description
CMP	(Reg, Reg) → Flag	Compare two values from registers and assert the condition flag.
CMPACC	(Reg, Offset, Acc) → Flag	Compare the accumulator with a value from register and assert the condition flag.

Table A.4: The Boolean Operation Instructions

Mnemonic	Input → Output	Description
CONF4	(Value, Pos) → SReg	Write a nibble to a special register without altering other bits.
CONF8	(Value, Pos) → SReg	Write a byte to a special register without altering other bits.
CONF16	Value → SReg	Write a word to a special register.

Table A.5: The Configuration Instructions

Mnemonic	Input → Output	Description
MOV	Reg → Reg	Move a register content to another register
LOAD	Mem → Reg	Load a value from data memory to register.
STORE	Reg → Mem	Store a register content to data memory.
LDACC	LOP/ROP (bypass DP) → Acc	Load an immediate value to accumulator. LOP/ROP[15:0] → ACC
STACC	Acc → Reg	Store the accumulator to register.

Table A.6: The Memory Manipulation Instructions

Reg	- register content
Acc	- accumulator content
Flag	- status and conditional flags
Offset	- shift the value to the left by 16 bits
Offset-able	- a value can be set to be offset
SReg	- special register content
PC	- programme counter
Stack	- programme stack
Value	- immediate value
Size	- the number of instructions within a static loop
Cycle	- the number of iterations of a static loop
NA	- not available
Pos	- a position of a nibble or a byte in a 16 bits value (0:low word, 1:high word)
Mem	- data memory

Appendix B

Special Registers

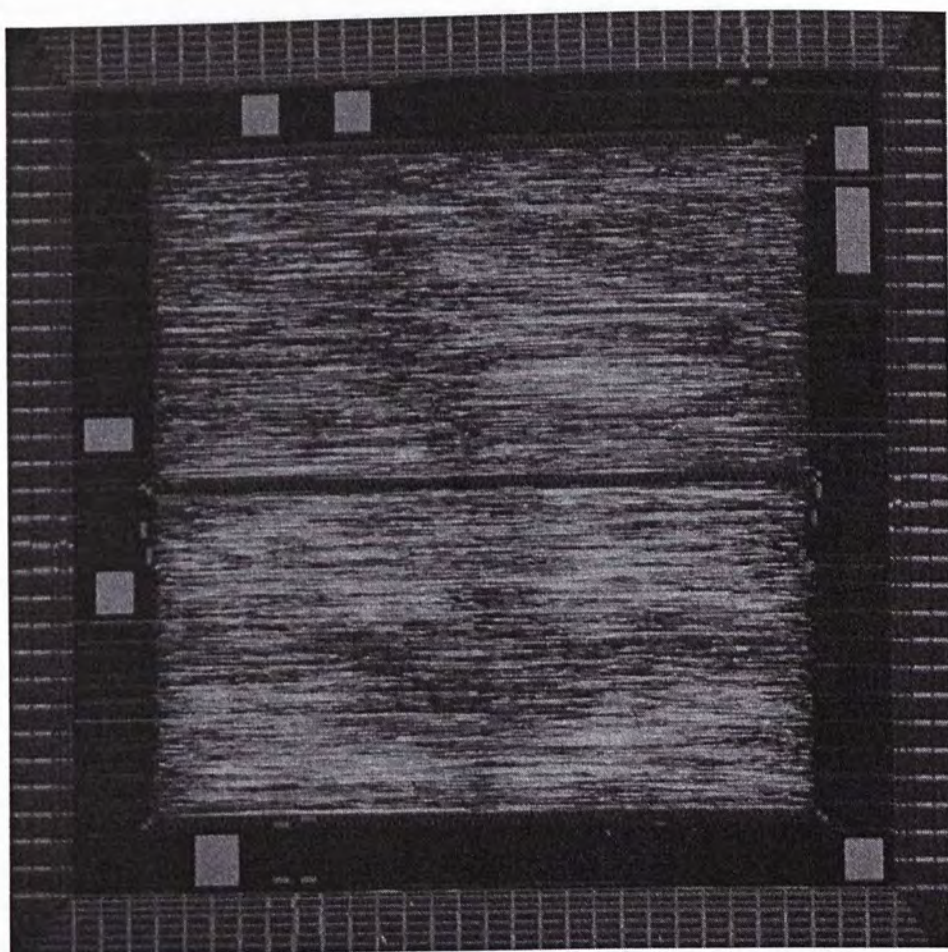
Table B.1: The Organization of Special Purpose Registers

	15					8					7		0				
0	INT (RESERVED)																
1	ov40	ov32	EQ	NE	LT	GT	LE	GE	0	0	EXP						
2	LSU XDATA address																
3	LSU XDATA size																
4	LSU XDATA step																
5	LSU YDATA address																
6	LSU YDATA size																
7	LSU YDATA step																
8	0	0	LSU XREG LD address						0	0	LSU XREG LD size						
9	0	0	LSU YREG LD address						0	0	LSU YREG LD size						
10	0	0	LSU XREG LD step						0	0	LSU YREG LD step						
11	0	0	LSU XREG ST address						0	0	LSU XREG ST size						
12	0	0	LSU YREG ST address						0	0	LSU YREG ST size						
13	0	0	LSU XREG ST step						0	0	LSU YREG ST step						
14	LSU Configuration																
15	0	X/Y	SFU LOP address						0	0	SFU LOP size						
16	0	X/Y	SFU ROP address						0	0	SFU ROP size						
17	0	0	SFU LOP step						0	0	SFU ROP step						
18	0	X/Y	SFU WB address						0	0	SFU WB size						
19	0	0	0	0	SFU Conf					0	0	SFU WB step					
20	PFU0 ~ PFU3 Configuration																

21	0	X/Y	PFU0 LOP address			0	0	PFU0 LOP size
22	0	X/Y	PFU0 ROP address			0	0	PFU0 ROP size
23	0	0	PFU0 LOP step			0	0	PFU0 ROP step
24	0	X/Y	PFU0 WB address			0	0	PFU0 WB size
25	0	0	0	0	PFU0 interval	0	0	PFU0 WB step
26	0	X/Y	PFU1 LOP address			0	0	PFU1 LOP size
27	0	X/Y	PFU1 ROP address			0	0	PFU1 ROP size
28	0	0	PFU1 LOP step			0	0	PFU1 ROP step
29	0	X/Y	PFU1 WB address			0	0	PFU1 WB size
30	0	0	0	0	PFU1 interval	0	0	PFU1 WB size
31	0	X/Y	PFU2 LOP address			0	0	PFU2 LOP size
32	0	X/Y	PFU2 ROP address			0	0	PFU2 ROP size
33	0	0	PFU2 LOP step			0	0	PFU2 ROP step
34	0	X/Y	PFU2 WB address			0	0	PFU2 WB size
35	0	0	0	0	PFU2 interval	0	0	PFU2 WB size
36	0	X/Y	PFU3 LOP address			0	0	PFU3 LOP size
37	0	X/Y	PFU3 ROP address			0	0	PFU3 ROP size
38	0	0	PFU3 LOP step			0	0	PFU3 ROP step
39	0	X/Y	PFU3 WB address			0	0	PFU3 WB size
40	0	0	0	0	PFU3 interval	0	0	PFU3 WB size
.								
.								
.								
	PFU _{n-3} ~ PFU _n Configuration							
.								
.								
.								
	0	X/Y	PFU _n LOP address			0	0	PFU _n LOP size
	0	X/Y	PFU _n ROP address			0	0	PFU _n ROP size
	0	0	PFU _n LOP step			0	0	PFU _n ROP step
	0	X/Y	PFU _n WB address			0	0	PFU _n WB size
	0	0	0	0	PFU _n interval	0	0	PFU _n WB size

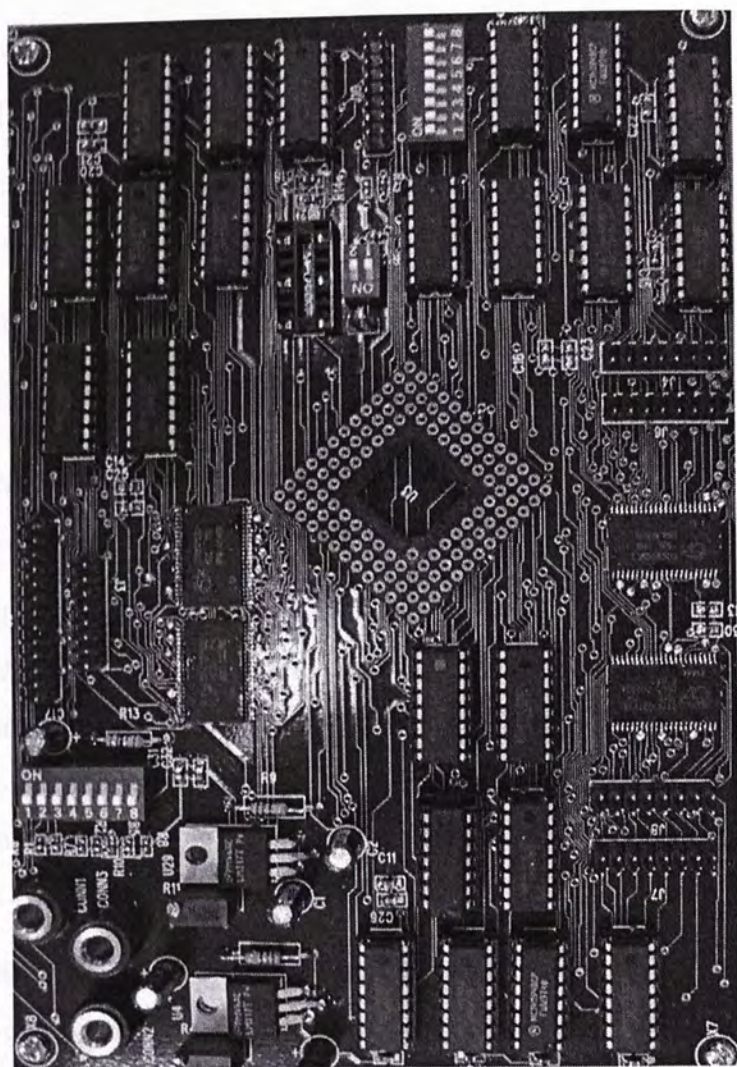
Appendix C

Chip Microphotograph of ASIP



Appendix D

The Testing Board of ASIP



Bibliography

- [1] J. G. Cousin, O. Sentieys, and D. Chillet, "Multi-algorithm asip synthesis and power estimation for dsp applications," in *IEEE International Symposium on Circuits and Systems*, pp. 621 – 624, May 2000.
- [2] P. Kievits, E. Lambers, C. Moerman, and R. Woudsma, "R.e.a.l. dsp technology for telecom baseband processing," in *Proceedings of ICSPAT*, 1998.
- [3] R. E. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, issue. 2, pp. 60 – 70, Mar-Apr 2000.
- [4] "Improv systems inc., jazz psa/jazz dsp." <http://www.improvsys.com>.
- [5] "Arc cores ltd., arctangent processor." <http://www.arccores.com>.
- [6] "Target compiler technologies, chess/checkers is a retargetable tool-suite." <http://www.retarget.com/products-more.html>.
- [7] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, issue. 11, pp. 1338 – 1354, Nov 2001.
- [8] "Institute of integrated signal processing systems, aachen university of technology, germany, lisa processor design platform." <http://www.iss.rwth-aachen.de/lisa/lpdp.html>.

- [9] J. H. Yang, B. W. Kim, S. J. Nam, Y. S. Kwon, D. H. Lee, J. Y. Lee, C. S. Hwang, Y. H. Lee, S. H. Hwang, I. C. Park, and C. M. Kyung, "Metacore: An application-specific programmable dsp development system," *IEEE Journal of Solid-State Circuits*, vol. 8, issue. 2, pp. 173 – 183, Apr. 2000.
- [10] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "Peas-iii: an asip design environment," in *Proceedings of International Conference on Computer Design*, pp. 430 – 436, Sept. 2000.
- [11] L. Rabiner and B. H. Juang, *Fundamentals of Speech Recognition*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [12] Y. L. Kwok, "Design of application-specific instruction set processors with asynchronous methodology for embedded digital signal processing applications," M.Phil. thesis, The Chinese University of Hong Kong, The Department of Electronic Engineering, Nov. 2004.
- [13] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proceedings of International Symposium on HPCA-6*, pp. 375 – 386, Jan. 2000.
- [14] J. H. Tseng and K. Asanovic, "Banked multiported register files for high-frequency superscalar microprocessors," in *Proceedings of 30th ISCA*, pp. 62 – 71, June 2003.
- [15] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *Proceedings of MICRO-35*, Nov. 2002.
- [16] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, pp. 305 – 310, Aug. 1998.
- [17] J. Mori, M. Nagamatsu, M. Hirano, S. Tanaka, M. Noda, Y. Toyoshima, K. Hashimoto, H. Hayashida, and K. Maeguchi, "A 10 ns 54x54-b paral-

- lel structured full array multiplier with 0.5- μ m cmos technology," *IEEE Journal of Solid-State Circuits*, vol. 26, issue. 4, pp. 600–606, Apr. 1991.
- [18] M. Nagamatsu, S. Tanaka, J. Mori, K. Hirano, T. Noguchi, and K. Hatanaka, "A 15-ns 32x32-b cmos multiplier with an improved parallel structure," *IEEE Journal of Solid-State Circuits*, vol. 25, issue. 2, pp. 494 – 497, Apr. 1990.
- [19] W. Han, K. W. Hon, C. F. Chan, T. Lee, C. S. Choy, K. P. Pun, and P. C. Ching, "A real-time chinese speech recognition ic with double mixtures," in *Proceedings of 5th International Conference*, pp. 926 – 929, Oct. 2003.
- [20] G. Fettweis and H. Meyr, "High-speed parallel viterbi decoding: algorithm and vlsi-architecture," *IEEE Communications Magazine*, vol. 29, issue. 5, pp. 46 – 55, May 1991.
- [21] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, *The HTK Book (for HTK Version 3.1)*. Cambridge University Engineering Department, 2001.
- [22] K. Nakamura, Q. Zhu, S. Maruoka, T. Horiyama, S. Kimura, and K. Watanabe, "Speech recognition chip for monosyllables," in *Design Automation Conference, Proceedings of the ASP-DAC*, pp. 396 – 399, Feb 2001.

CUHK Libraries



004270438