Electronics and Computer Science

FACULTY OF PHYSICAL AND APPLIED SCIENCES

UNIVERSITY OF SOUTHAMPTON

Ricardo da Silva

April, 2013

Speech Recognition on Embedded Hardware

Supervisor: Dr Steve Gunn

Examiner: Dr Nick Harris

A project report submitted for the award of

Electronic Engineering with Mobile and Secure Systems

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

A project report submitted for the award of
Electronic Engineering with Mobile and Secure Systems

SPEECH RECOGNITION ON EMBEDDED HARDWARE

by Ricardo da Silva

This report presents a proof of concept system that is aimed at investigating Hidden Markov Model based speech recognition in embedded hardware. It makes use of two new electronic boards, based on a Spartan 3 FPGA and an ARMv5 Linux applications processor, that are currently under development at the University of Southampton. Speech data is pre-processed in Linux, by performing spectral analysis followed by a Discrete Cosine transformed filter-bank analysis, to produce Mel Frequency Cepstral Coefficients. Given this observation data, the FPGA performs the most computationally intensive part of a modern HMM based speech recognition system – evaluating the state emission probabilities. It is shown that the FPGA is capable of performing these calculations faster than a software implementation on the Linux processor. The resulting system is also a valuable example of how these two boards may be used in a practical setting.

Contents

List of Figures

Acknowledgements

Thanks to Steve Gunn, Srinandan Dasmahapatra

Statement of Originality

I, Ricardo da Silva , declare that the project entitled *Speech Recognition on Embedded Hardware* and the work presented in the project are both my own, and have been generated by me as the result of my own work. I confirm that:

- this work was done wholly or mainly while in candidature for a degree at this University;
- where any part of this project has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project is entirely my own work;
- I have acknowledged all main sources of help;
- where the project is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Ricardo da Silva

Nomenclature and Abbreviations

$\lambda(\pi, a, b)$ A fully defined Hidden Markov Model

μ_j N-dimensional vector of means for state j, indexed from 1 to N

Ω_j N-dimensional vector of pre-computed scaling factors for state j

π The set of initial state probabilities

σ_j N-dimensional vector of standard deviations for state j, indexed from 1 to N

a_{ij} The probability of transitioning from state i to state j

$b_j(O)$ The probability of state j emitting observation O

K_j Pre-computed constant for state j

DTW Dynamic Time Warping

FPGA Field Programmable Gate Array

GDP Gaussian Distance Pipeline

HMM Hidden Markov Model

LTIB Linux Target Image Builder

Chapter 1

Introduction

1.1 Goals

At the highest level, the primary goal of this project is to implement part of a modern Hidden Markov Model (HMM) based speech recognition system, with the constraint that it must be done on embedded hardware. This and other important goals are outlined here.

1.1.1 Speech Recognition

Most modern HMM based speech recognition systems are extremely complex, and can take years to design and optimise for a particular implementation. The goal of this project is not to implement such a system, but rather to explore the possibilities of what may be achieved with a low power applications processor and a relatively small FPGA (Field Programmable Gate Array . In particular, the aim is to use the applications processor (running Linux) to perform pre-processing of speech data, and use the FPGA to perform mathematical manipulations of the observations. Part of this goal is to evaluate the performance, in terms of calculation speed, of the FPGA relative to a conventional processor. Finally, it is hoped that the system developed here may be later used as a basis for future research into the subject.

1.1.2 The Micro Arcana

The Micro Arcana is a new hardware platform aimed at undergraduate students, currently under development by Professor Steve Gunn. In terms of hardware, one of the project goals is to further development of the Micro Arcana family of boards, and provide a valuable example of how they may be usefully combined. The aim is to build the entire speech recognition system on two of the Micro Arcana boards, making it a self-contained embedded design. In addition, part of the project is setting up and configuring these two boards, so that they may be easily picked up by undergraduates.

1.1.3 Theoretical understanding

An important goal of the project is to develop a higher level understanding of the algorithms used in speech recognition, and to get experience designing a large-scale embedded application. This project encompasses a very wide range of subjects, including intelligent algorithms, digital systems design, embedded processing, and hardware design.

1.2 Motivation

Speech recognition is an interesting computational problem, for which there is no fool-proof solution at this time. Recently the industry for embedded devices and small-scale digital systems has expanded greatly, but in general these devices do not have the power or speed to run speech recognition. FPGAs may present a way of increasing the capability of such systems, as they are able to perform calculations much faster than traditional microprocessors. The versatility of embedded systems, combined with the challenges of a complex system such as speech recognition, makes this an appealing area to explore.

As hardware platforms go, most of the Micro Arcana is still very new and untested, as it is still under development. In addition, there are not many examples of how they may be used, and very little documentation. In order to improve their reception by students, it would greatly help to have proven use cases and examples of how these boards may be used individually and together. Using a larger FPGA (such as an Altera DE board) was considered during the planning stage of this

project, but it was decided that it would be more beneficial and interesting to develop and use the Micro Arcana.

1.3 Results and Personal Contribution

The project implements two parts of a modern speech recognition system, using two development boards from the Micro Arcana family. The implementation, described in detail in Chapter 4, uses an FPGA to perform the most computationally expensive part of HMM based recognisers – scoring the states of each HMM model for a given input vector. Essentially, the ARM Linux based "L'Imperatrice" is used as the application controller, and is connected to the FPGA based "La Papessa" board. The processor reads Microsoft WAV format speech files, performs the necessary pre-processing and sends observation vectors to the FPGA. Given an observation, the FPGA processes it and sends back scores for each state in the speech model, which represent the probability of that state emitting the observation. Given these scores, the next step for a speech recogniser would be to perform Viterbi decoding (possibly using token-passing or a similar algorithm) in order to find the most probable sequence of HMMs, and thus eventually find the most probable word or phoneme sequence spoken.

Accomplishing this required substantial research into HMM based speech recognition algorithms, embedded Linux, and digital design. The two Micro Arcana boards are very new, and some vital parts on them were completely untested before this project. Although there is significant research into the use of FPGAs for speech recognition, most cases use large FPGAs, and often a relatively fast PC to perform pre-processing. This project is designed to be a proof of concept exercise – it explores the capabilities of the boards, and provides an example of how they may be used together. Furthermore, the results are satisfactory, and point towards an FPGA being an appropriate platform for performing these calculations.

Chapter 2

Speech Recognition Techniques and Tools

2.1 Speech Recognition Systems

In general, "Speech Recognition" refers to the process of translating spoken words or phrases into a form that can be understood by an electronic system, which usually means using mathematical models and methods to process and then decode the sound signal. Translating a speech waveform into this form typically requires three main steps [11]. The raw waveform must be converted into an "observation vector", which is a representative set of data that is compatible with the chosen speech model. This data is then sent through a decoder, which attempts to recognise which words or sub-word units were spoken. These are finally processed by language modeller, which imposes rules on what combinations of words of syntax are allowed. This project focusses on implementing pre-processing, and the first stage of the decoder, as these are interesting tasks from an electronic engineering point of view.

There are a variety of different methods and models that have been used to perform speech recognition. An overview of the most popular will be described here, and then the chosen technique (HMMs) is described in Section 2.2.

2.1.1 Tor's Algorithm

"Tor's Algorithm" is a very simple speech recognition system [3], capable of accurate speaker dependent speech recognition for a small dictionary of about ten words. It is based on a fingerprinting model where each word in the dictionary must be trained to form an acoustic "fingerprint". This fingerprint is based on the time variations of the speech signal after being filtered appropriately. Then recognition is reduced to finding the Euclidean distance squared between the input vector and each of the stored fingerprints. The 'best' match is the word with the smallest distance from the input.

It is likely that this system is easily implementable in this project's time frame, but it was judged too simplistic and therefore not interesting enough to warrant implementing. However, importantly, it outlines two major components of any speech recognition system – pre-processing and decoding (recognition). More complex systems essentially just use more complex speech models and pre-processing methods.

2.1.2 Dynamic Time Warping

Speech, by nature, is not constrained to be at a certain speed – the duration of words will vary between utterances, and a speech recognition system should be able to handle this. Dynamic Time Warping (DTW) is essentially the process of expanding and contracting the time axis, so that waveforms may be compared, independent of talking speed. Combined with a dynamic programming technique for finding the optimal 'warp' amount, it became a widely used approach to solving the problem of speech duration modelling [7]. One useful property of DTW is that it may offer good performance even with little training, as it only needs one word as a template [11]. Conversely, the performance of DTW based systems cannot be increased much with more training, unlike Hidden Markov Models. Although DTW is better than Tor's algorithm, it is also judged to be relatively old and rarely used technology, and therefore not as interesting as HMMs for a project focus.

2.2 Hidden Markov Models

By far the most prevalent and successful approach to modern speech recognition uses Hidden Markov Models for the statistical modelling and decoding of speech [6]. The flexibility inherent in HMMs is key to their success, as a system can be made more and more accurate by simply improving the HMM models or training the models further. The classic tutorial paper by Rabiner ([13]) is one of the best references for HMMs in speech recognition, and provides a very good overview of modern systems. However, a brief summary of the fundamentals of HMMs is given here. The following sections are based heavily on [13] and [19].

An N-state Markov Chain can be described as a finite state machine of N nodes with an $N \times N$ matrix of probabilities which define the transitions between each state. According to the notation in [13], the elements of this matrix are defined as $a_{ij} = P(\text{state at time } t = j | \text{state at time } t - 1 = i)$. To make this a Hidden Markov Model, each state is assigned an emission probability for every possible observation, which defines how likely that state will emit that observation. In this case, the actual position in the state machine is not observable – only the state emissions are (thus "Hidden" Markov Model). The probability that a state j will emit observation O is defined as $b_j(O)$, and may be either a discrete value or a continuous distribution depending on the nature of the observations. Thus, an HMM is defined entirely by the matrices a and b, and a set of initial probabilities for each state, π, collectively denoted as $\lambda(\pi, a, b)$.

For speech recognition, the performance is substantially improved by using continuous HMMs, as it removes the need to quantise the speech data which is, by nature, continuous [10]. A common distribution used for continuous probabilities is the multivariate Gaussian Mixture, which is essentially a weighted summation of several different Normal distributions [4]. However, for use in HMMs, the computational complexity is greatly reduced if the covariance matrix is diagonal (i.e., the components of each Gaussian are uncorrelated). This requirement can lead to extra pre-processing of observation data in order to remove correlation between the components.

2.2.1 Levels of Complexity

The simplest HMM based systems use a single HMM for every word in the recognition dictionary. Given a set of observations, each HMM can be scored based on

the probability that it would output the observations. The HMM with the highest score is taken as the recognised word. The most apparent limitation of this system is that a very large amount of training would be required if a dictionary of substantial size was to be used. At the very least, one sample of each word would need to be recorded to train the full system, which would be a very time consuming process. However, for simple applications (voice dialling, for example) this is manageable.

The next step up in complexity from single word HMMs is models that composed of sub-word utterances (phonemes). This allows a smaller set of HMMs to be used for much larger dictionary recognition, as words are recognised based on sequences of sub-word HMMs. Thus instead of searching through a single HMM to recognise a word, the recognition process becomes a search through a trellis of multiple HMMs in order to find the best path through them. The most simple HMM system of this form is based on mono-phones, of which there are about 50 in the English language.

Even more complexity (and, potentially, recognition accuracy) can be introduced by using bi- or tri-phone HMMs, which model transitions between two or three mono-phones. Using this form of HMM will increase the acoustic model size greatly however, as there are many possible combinations of mono-phones in the English language. However, it allows context dependent scoring of phonemes, including HMMs that model word endings and starts, or silences. In the Sphinx 3 recognition engine, the internal states of these HMMs are referred to as "Senones", and the term has been adopted and used extensively in this project [15].

2.3 Speech Pre-Processing

Speech signals are complex waveforms and cannot be processed without some form of feature extraction which reduces the complexity whilst retaining the important features. In modern speech recognition systems the two most common methods of analysing and representing speech are: [8]

- Linear Predictive Coding (LPC)
- Mel-Frequency Cepstral Coefficients (MFCCs)

Both these methods attempt to model the movement and dynamics of the human vocal tract and auditory perception. LPC is more suited to speaker recognition

(the process of identifying voices, rather than words), whilst MFCCs are more useful for speech recognition [1].

The Mel-Frequency Cepstrum is based on a filterbank analysis with a cepstral transformation, which is required due to the high correlation between filterbank amplitudes. The human ear perceives sound on a non-linear frequency scale, and one way of improving recognition performance is by using a similar scale for analysis of speech. A filterbank analysis can be used to perform this discrimination between different frequencies, and the frequency bins are usually spaced using the Mel frequency scale. However, the filterbank amplitudes are highly correlated, which greatly increases the computational complexity of the HMM based recogniser as the covariance matrix will not be diagonal. In order to correct this, a discrete linear cosine transform is taken on the log filterbank amplitudes, finally resulting in a set of Mel Frequency Cepstral Coefficients. The HTK (Section 2.4) defaults to using twelve MFCC filterbank bins. [19] [11]

In order to attain MFCCs, a sampling rate must be chosen such that enough data is gathered while allowing sufficient processing time. In addition, to perform Fourier transforms on the speech, the incoming signal must be windowed appropriately. The HTK has a set of default values for these parameters, which are assumed to be appropriate.

An improvement to both LPC and MFCCs is to compute time derivatives in the feature extraction process, which gives a better idea of how the signal changes over time. In addition, the log energy of each sample may also be computed to also boost recognition ability.

2.4 The HTK and VoxForge

The Hidden Markov Model Toolkit (HTK) is a set of tools and libraries for developing and testing HMMs, primarily for speech processing and recognition tools [19]. Given a model structure and a set of transcribed speech recordings (a speech corpus), a set of HMMs may be trained using the HTK. This includes performing all pre-processing in a number of formats, and testing recognition capabilities of a model [17].

Voxforge is an open source speech corpus which is aimed at facilitating speech recognition development. It provides pre-compiled acoustic models – essentially

large sets of HMMs – in the format created by HTK, licensed under the GPL (GNU General Public License) [2]. The alternative would be to use another speech corpus (such as TIMIT [5]), and then use the HTK to design and train the acoustic model. This is potentially a very time consuming process, so Voxforge is useful because it essentially cuts this step out. In addition, the Voxforge model may be easily adapted to a specific person's voice using only a few minutes of transcribed speech. However, the model is very complex, with $\tilde{8}000$ tri-phone context-dependent HMMs with multivariate Gaussian output probabilities. Thus, implementing a recogniser system based on this model requires a lot more work than if a simpler model was used, such as one based on discrete (output probability) HMMs. However, modern speech recognisers are likely to use a model that is as complex, if not more so.

2.5 Embedded Hardware and Speech Silicon

A wide range of speech recognition software (commercial and open source) exists for desktop computers or laptops. However, speech recognition for embedded systems is less widespread. In a real speaker independent, context-dependent system there could be thousands of these states, each requiring a large number of calculations, depending on how complex the models are. The processing power required for this is often not available on embedded processors. Recently there has been increased research into the use of DSPs and FPGAs for speech recognition [12]; of particular interest is Stephen Melnikoff's PhD Thesis [11], and the Speech Silicon architecture [14]. The former investigates a variety of HMM based recognisers on an FPGA, using a PC to perform pre and post processing. The latter details a data driven FPGA based architecture capable of performing recognition on medium-sized vocabularies.

These two systems are good guides for what is possible and important to implement in a speech recognition system. In addition, they present efficient ways of performing certain tasks, such as Gaussian distance calculations. Several parts of the system implemented in this project were influenced by the way similar parts were implemented by Melnikoff and Speech Silicon. However, both Melnikoff and Speech Silicon perform an in-depth analysis of an entire speech recognition system based on programmable logic. As such, both of them require relatively large FPGAs – far larger than the Micro Arcana FPGA board. Thus they were more useful from a theoretical point of view, rather than architecturally.

Chapter 3

Design Theory and Approach

This chapter provides details of the relevant theory behind HMM based speech recognition systems, as well as a description of the various development environments used during the project.

3.1 The HMM based model

Due to the flexibility of HMMs, and the complexity of speech, there have been several different approaches to building speech models (the sheer size of the HTK book indicates how much flexibility exists). However, at this stage, the implementation of these algorithms is a more interesting pursuit, rather than devising the best way of modelling speech. Therefore, it was decided to use the pre-designed models from Voxforge for this project, and build the hardware to work with these models. Thus, various parameters were fixed from the start, including:

- Sampling rate of audio: 8kHz.
- Window size: 25ms (duration of observation frames).
- Frame period: 10ms (time between observation frames).
- Pre-processing output: 12 MFCCs, 12 MFCC derivatives, 1 Energy measure.
- Output probabilities of HMM states: Single Gaussian distribution, 25-element mean and variance vectors.

- Number of monophones: 51 (Includes a monophone for silence. This is also the number of transition matrices).
- Number of senones: $\tilde{7}000$.
- Number of HMMs: $\tilde{8}300$[1] (each with 3 outputting states).

The only modification made to the Voxforge models was that they were adapted for the author's voice, primarily to gain confidence with using the HTK and HMMs. Please see Appendix D for the scripts and HTK configuration files used to generate these models.

The term "outputting states" refers to states that produce an observation – most of the HMMs have 5 states in total, but the first and last are non-emitting. The transition probabilities between states one and two, and between states four and five, are primarily used to model inter-HMM probabilities for decoding purposes. The senones are context dependent, that is, there are many different senones for each monophone, each with different predecessor and successor monophones.

3.1.1 The HMM tasks

For an HMM model, denoted as λ, there are usually three important problems:

- Design and train the model to accurately represent real data – adjusting λ to maximise $P(O|\lambda)$.
- Finding the probability that an HMM produced a given observation sequence, $P(O|\lambda)$.
- Finding the 'best' path through a trellis of HMMs and states to produce a given observation sequence.

For this project, the first problem is solved by using Voxforge (2.4). The second problem is potentially very computationally expensive, as the speech model may be complex or large. In particular, this step requires evaluating the output probability of each state in the model for every new observation frame, which is particularly time consuming if the HMMs have continuous output distributions. In modern

[1] There are more HMMs than senones because some senones are used in more than one HMM

speech recognition systems, this step regularly accounts for up to 70% of the total processing time [9]. This is the step that the project focusses on.

In all literature encountered, the Viterbi algorithm is the preferred method for solving the final problem. It is an iterative approach to solving the optimisation problem, and has the added bonus that not much data needs to be stored during the calculation [14]. This problem is beyond the scope of the current project, but a full explanation of the Viterbi decoding process is available from [13],[16], [18].

3.1.2 Senone scoring

As described in previous sections, the FPGA is to be used to compute HMM emission probabilities for every senone in the model, for every observation vector. In this system the new vectors arrive once every 10ms, and there are about 7000 senones that must be evaluated. The mathematical operations required to do this are now outlined.

If the observation vector at time t is denoted as $\mathbf{O}_t = O_{t1}, O_{t2}, ..., O_{tN}$, then the score of senone j is $b_j(\mathbf{O}_t)$ – the probability of that senone producing $\mathbf{O}_t$. The output probability of each senone is an N-dimensional multivariate Normal distribution, represented by N-element vectors of means μ_j and standard deviations σ_j. Usually an $N \times N$ covariance matrix would be required, but due to the statistical nature of Mel Frequency Cepstral Coefficients, this matrix is diagonal, and thus can be represented with N elements. Therefore the score is given by:

$$b_j(\mathbf{O}_t) = \mathcal{N}_N(\mathbf{O}_t; \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j^2) \tag{3.1}$$

$$= \prod_{n=1}^{N} \frac{1}{\sigma_{jn}\sqrt{2\pi}} \exp\left(-\frac{(O_{tn} - \mu_{jn})^2}{2\sigma_{jn}^2}\right) \tag{3.2}$$

However, hardware computation of this equation may be greatly simplified by taking logarithms of both sides, removing the need to evaluate exponentials:

$$-\ln(\mathcal{N}(\mathbf{O}_t; \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j^2)) =$$

$$\left[\frac{N}{2}\ln(2\pi) + \sum_{n=1}^{N}\ln(\sigma_{jn})\right] + \sum_{n=1}^{N}(O_{tn} - \mu_{jn})^2\left[\frac{1}{2\sigma_{jn}^2}\right] \tag{3.3}$$

Furthermore, the square bracketed terms in Equation 3.3 do not depend on the observation, and thus may be pre-computed. The final equation is reduced to subtract, square, multiply, and accumulate:

$$\ln(\mathcal{N}(\mathbf{O}_t; \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j)) = K_j - \sum_{n=1}^{N}(O_{tn} - \mu_{jn})^2 \Omega_{jn} \tag{3.4}$$

Where the precomputed values are:

$$K_j = -\frac{N}{2}\ln(2\pi) - \sum_{n=1}^{N}\ln(\sigma_{jn})$$

$$\Omega_{jn} = \frac{1}{2\sigma_{jn}^2}$$

3.2 Hardware environment

As the Micro Arcana is still under active development, part of the project involved setting up and testing the two boards that were used.

3.2.1 L'Imperatrice

Several important features on the ARM-based "L'Imperatrice" board are still very untested, including parts essential to the project. It is based on a Freescale iMX23 ARMv5 applications processor. To be used for the project, the following items were required (in order of importance):

- Native or cross compiler set-up
- Application UART functionality
- GPIO functionality

A Linux Target Image Builder (LTIB) environment, which is primarily used for setting up board support packages (BSP), was installed on an Ubuntu virtual machine. It has been used to build and test various kernel configurations, and also includes full cross-compiler support for the board. It essentially provides a platform on which software for L'Imperatrice may be developed and deployed. In

addition to LTIB, the ArchLinux build system (ABS) was investigated as a potential alternative to LTIB. The primary advantage of the ABS is that only a small number of files need to be distributed, which, when run, will download and compile all dependencies of the build. An ABS configuration exists for the Olinuxino, a Linux board also based on the Freescale iMX23, which may be tweaked to suit the L'Imperatrice. However, due to lack of time and the relative ease of the LTIB set-up, this was not explored.

3.2.2 La Papessa

The Xilinx FPGA-based "La Papessa" board is also being actively developed, and some of its features have not been tested. In order to facilitate the development of code on the La Papessa board, several combinations of software environments were explored. The FPGA is a Xilinx Spartan XC3S50AN, which is compatible with the Xilinx ISE Webpack design software package. However, one drawback to the ISE Webpack is its lack of support for synthesis in SystemVerilog. Besides being syntactically more powerful, SystemVerilog is the HDL that is currently taught to all new undergraduates at the University of Southampton. Having some documentation of a proven way to use SystemVerilog with this board would improve its reception and usage. In addition, SystemVerilog has advantages over Verilog for verification and simulation, which will be used to improve the design.

Synplify Pro/Premier is an alternative HDL synthesis tool, which is compatible with the Xilinx software toolchain and also supports SystemVerilog. For primarily this reason, it was decided that Synplify Premier would be used for synthesis during the project. The other design tasks (port mapping, programming file generation) are accomplished with ISE Webpack (See Appendix C for detailed description of this process).

3.3 Risk Analysis and Contingency Planning

Efforts were made to divide the project work up into sections that were independent. If at all possible, it was modularised, so that if one section became infeasible, it could be dropped without affecting the outcome of the final product greatly. The Micro Arcana especially, was completely unknown, and therefore several alternatives were lined up in case it became impossible to use it. The major risks identified are presented in Table 3.1, along with possible solutions.

Table 3.1: Summary of identified risks

Risk	**Negative Impact**	**Solution**
La Papessa's on-board SRAM may not work, thus making it impossible to store a large number of scores.	Low	Reduce model size. Proof of concept is more important than storing many scores at this point.
GPIO on L'Imperatrice may not work, or inter-board communication could be very hard.	Low	Develop and test the two modules separately. Communications is a minor issue that can be solved later, if the main systems work.
The Xilinx XC3S50 FPGA may be far too small to implement any useful algorithm or pipeline.	High	A University owned Altera development board, which has a far larger FPGA on it, can be used instead.
L'Imparatrice may have too many non-functional parts, which would take too long to fix before being usable.	High	A Raspberry Pi (ARM based Linux board) can replace it, as these are known to be functional and available.
Designing speech pre-processing code may be too time consuming or difficult.	Medium	The HTK is capable of producing observation vectors in the correct format, which can then be sent to the FPGA.

A variety of measures were taken in order to protect against the possibility of work being lost. Primarily, the source code and designs were backed up on external storage, as well as being regularly uploaded to a Github repository. The 'Git' version control software was used throughout the project. The primary benefit was that it enforced a regular process of adding changes, validating them, and committing them to the repository. This helps keep development on-track and focussed, as well as preserving sets of code that work. It also provides a logbook style commit history, which allowed the progress of the project to be observed (see Appendix B.3.

Chapter 4

Implementation

4.1 System Overview

The hardware related goals outlined in Chapters 1 and 2 can be summarised as:

1. Design a system in programmable logic that can efficiently evaluate Equation 3.4.
2. Design a C program to pre-process speech data according to the required form described in Section 3.1.

The overall system layout, shown in Figure 4.1, is comprised of two primary blocks – the processor and the FPGA (on L'Imperatrice and La Papessa boards respectively). The entire system is powered from a single supply connected to the L'Imperatrice battery connector; La Papessa is powered through a ribbon cable between the two boards. This was done in order to minimise the amount of external circuitry needed, and to show that the two devices are able to work together fairly easily.

From the list above, the first task is implemented on La Papessa, and the second on L'Imperatrice. These two blocks will now be examined in greater detail.

4.2 Number Format

Before beginning implementation of any part of the project, a number representation which would be appropriate for the FPGA had to be decided upon. Firstly,

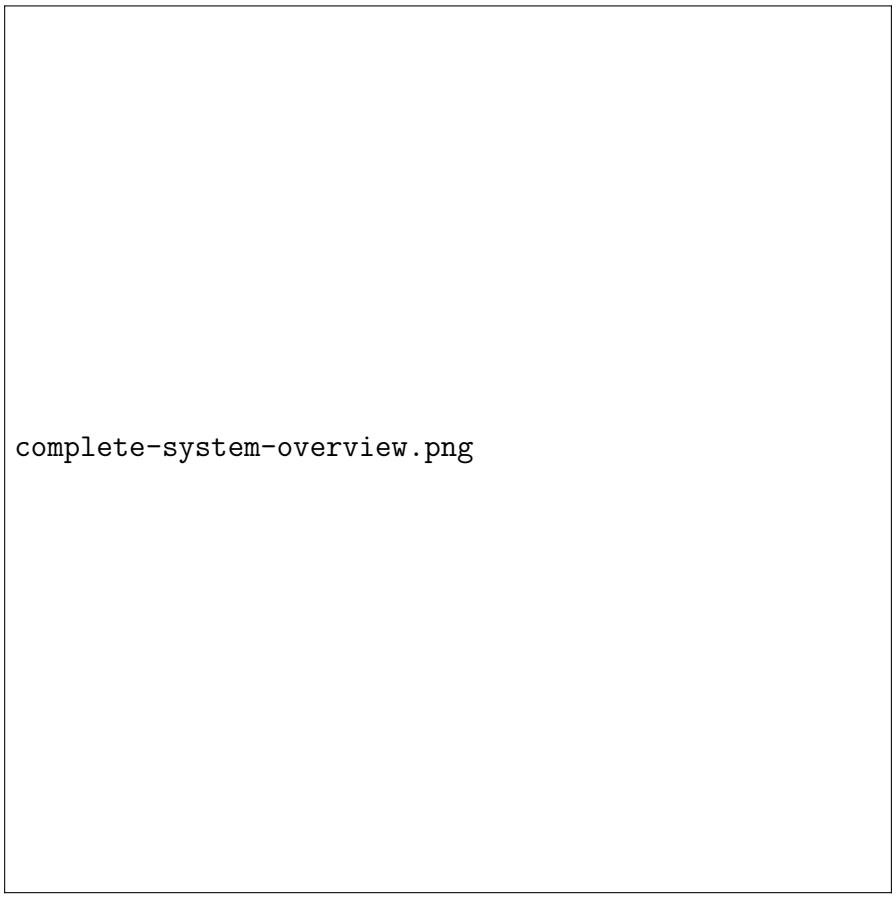

Figure 4.1: Complete system overview

it was recognised that using floating-point arithmetic on the FPGA would not be worth the effort, and therefore some form of fixed-point system was needed. Further, the number magnitudes vary greatly between stages and parameters in the system. In order to solve this problem, different scaling factors were used to bring most of the parameters to a similar magnitude.

The inputs and outputs of the Gaussian Distance Pipe, the module that performs the core calculation, are all signed 16-bit fixed point numbers, with varying scaling factors. In particular, the k parameter was generally larger than x, *mean*, and *omega*, and thus was scaled down. The scaling factors were decided primarily by analysing the HMM models, to determine the largest and smallest numbers used.

These decisions were influenced by Melnikoff [11], and the HTK, as they both use scaled 16-bit numbers to represent the parameters and scores.

4.3 La Papessa (The FPGA)

The system on La Papessa performs these main tasks:

- Receives observation vectors from L'Imperatrice.
- Computes the score (Equation 3.4) of every senone in the model, with the given observation.
- Normalises the senone scores.
- Sends the scores back to L'Imperatrice.

4.3.1 Top Level Module

A simplified diagram of the top level module is given in Figure 4.2, showing the main components of the system. This module included the main controller logic, which essentially waited for a new observation vector, then cycled through the necessary operation states. Figure 4.3 shows an ASM of this logic, and also outlines the main areas of the design that need explanation.

The top level module is also responsible for handling access to the on-board SRAM chip, which several modules need to write or read from. It essentially multiplexes the required signals, and leaves them floating (high impedance) when they are not needed. The "Debug signals" shown in Figure 4.2 are a number of internal signals that are routed to output ports in order to facilitate hardware debugging.

4.3.2 Gaussian Distance Pipeline

The Gaussian Distance Pipeline (GDP) is the core component of the system, and computes Equation 3.4. It is a relatively simple 4-stage pipeline, with one stage for every step in the equation (subtract, square, scale, accumulate). Although the gains from using a pipeline in this case are relatively small, it would be very useful if more complex models were used. The Speech Silicon [14] project had a substantially more complex GDP, as their senones have several Gaussian distributions that must be mixed to produce the final output distribution. A block diagram of the GDP module is shown in Figure 4.4.

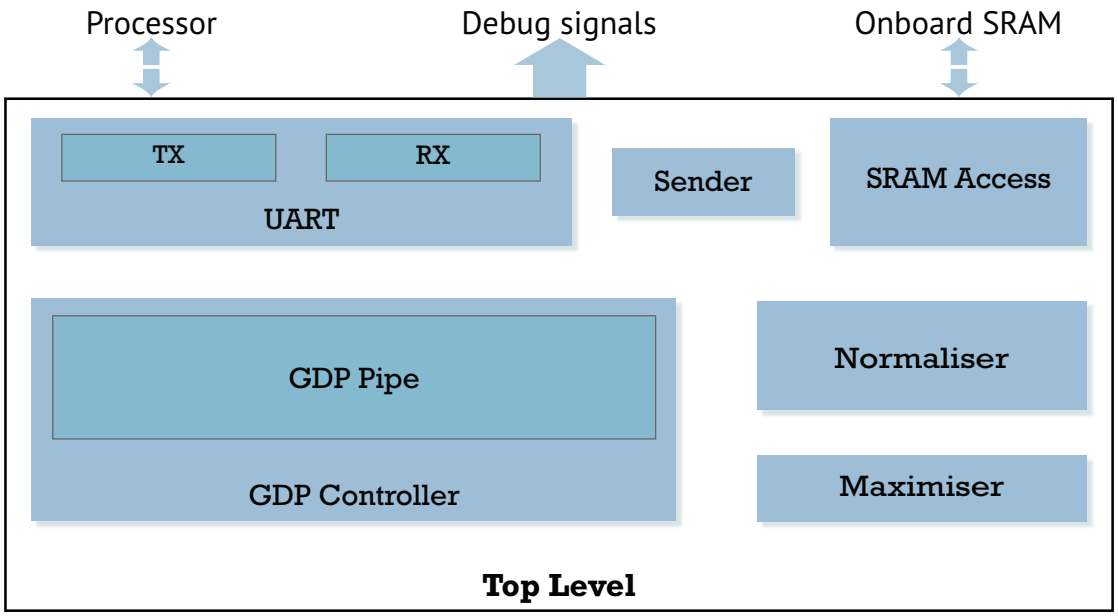

Figure 4.2: Component diagram of the Top Level Module

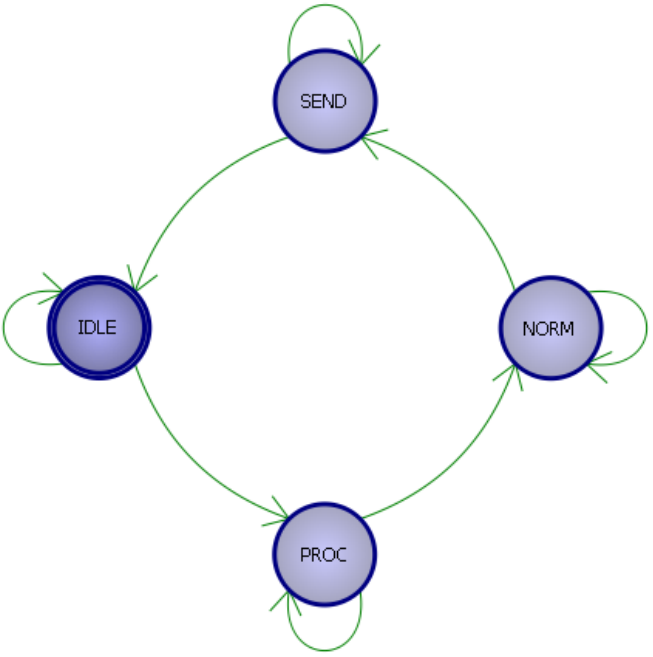

Figure 4.3: ASM diagram of Top Level Module

In this module, '`n_senones`' and '`n_components`' are both parameter inputs, which determine the number of senones and the number of components per mean and variance in the model. This allowed the design to be scaled down as size constraints became restrictive.

The pipeline itself is a static object without much control logic, and thus requires

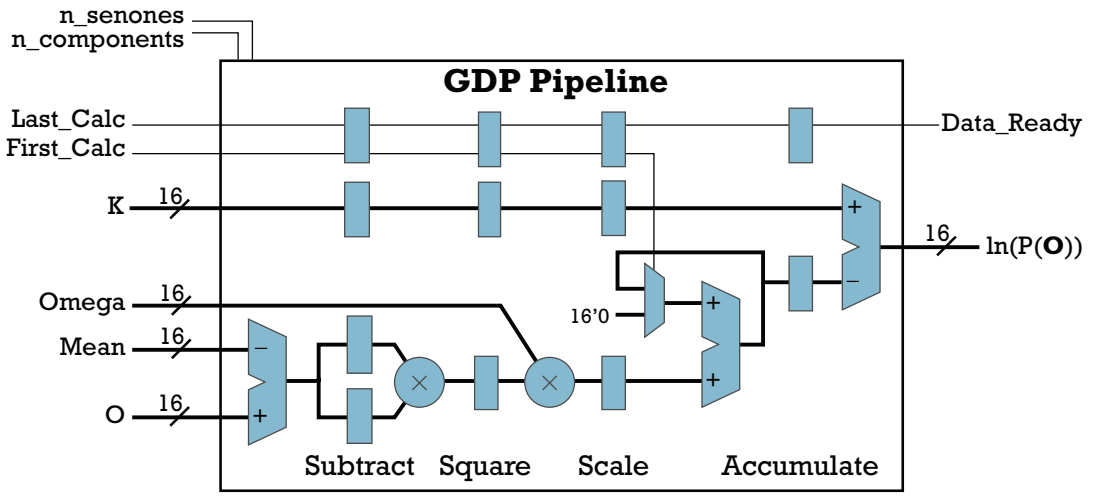

Figure 4.4: Gaussian Distance Pipeline block diagram

a controller which sequentially uses it to score each senone in the model. This controller is a simple state machine of only two states (`IDLE` and `LOAD_GDP`), which begins feeding the pipe when a '`new_vector_available`' input flag is asserted. This module is shown in Figure 4.5. A '`last_senone`' output flag is asserted when the GDP produces the last senone score. When the controller is in the `LOAD_GDP` state, it essentially loops through the senones in the model, extracting their parameters and sending them to the GDP.

In order to facilitate the extraction and manipulation of senone parameters, a SystemVerilog structure was created, shown in Listing 4.1. A SystemVerilog ROM module, connected to the controller, was populated with the senone parameters, stored in this structure. Thus, when '`new_vector_available`' is asserted, the controller simply counts from 0 up to `n_senones` (the number of senones in the model), and pulls the required parameters out of the ROM.

```
typedef struct packed {  /* num: typedef logic signed [15:0] num; */
  num k;
  num [n_components-1:0] omegas;
  num [n_components-1:0] means;
} senone_data;
```

Listing 4.1: Senone parameter data structure

After asserting `new_vector_available`, and providing the relevant vector on the '`x`' input, the top level module will see senone scores being sequentially produced, along with an index or ID which is unique to each senone. The Top Level module

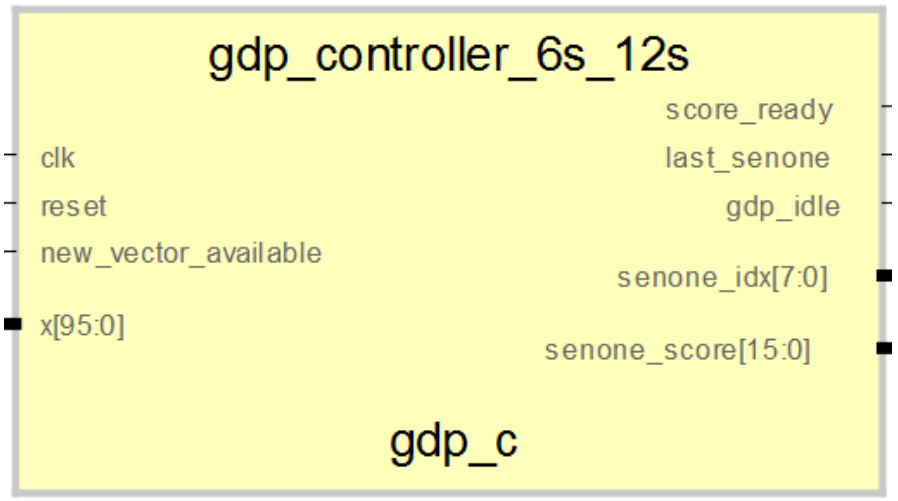

Figure 4.5: Gaussian Distance Pipe Controller block diagram

is then responsible for storing the score in SRAM, and a "Maximiser" module registers the highest score seen.

4.3.3 Normalising Scores

The Speech Silicon architecture included a module which normalised the senones before they were used for decoding. A very similar module is implemented here to perform the same normalisation. The highest score is found while senones are being evaluated, and then this score is subtracted from all the final scores. This causes the senone with the highest score to have a score of 0, which corresponds to a probability of 1 (the scores are log probabilities, see Section 3.1.2).

4.3.4 SRAM

In order for the normaliser to access the senone scores, they must be stored somewhere as they are processed through the GDP. One option would be to store them on the FPGA itself, where they would be accessible to all the modules. However, this is impractical due to the potential size of an HMM model, and thus the large amount of RAM that would be required. A better alternative would be to use external SRAM with low latency that is made accessible to whichever modules need it.

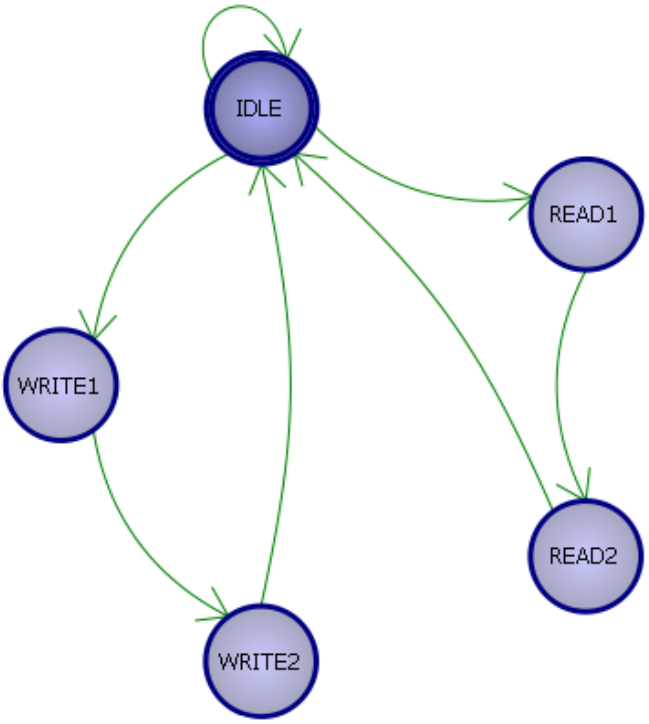

Figure 4.6: ASM of the SRAM access module

Revision C of La Papessa board includes an on-board SRAM chip, which has an 8 bit data bus and 21 bit address bus, and was completely untested before this work. In order to use it easily, a module was created primarily to interface the 8 bit data bus with the 16 bit number system used. It's only operations are to write and read 16 bit values to the SRAM, and signal when it is ready or idle. This is accomplished with the state machine shown in Figure 4.6.

4.3.5 Communications

The primary method of communication between the FPGA and processor is a standard UART bus, running at 115200 baud. The communications module on the FPGA is essentially comprised of standard UART receive/transmit modules and a module that is wrapped around them to provide a higher level of data abstraction. Because numbers are all 16 bits long in this system, and UART words are normally 8 bits, one of the wrapper purposes is to receive and transmit 16 bit numbers. In addition, it is known beforehand that the FPGA should receive a certain number of bytes per observation vector. This allows the UART module to wait until a 'packet' with that number of bytes has arrived, before signalling to the main controller that a new vector has arrived. In this implementation,

a buffer is simply filled up as new bytes arrive, and then is passed to the main controller when full. However, it may happen that the UART module erroneously receives a byte, causing the buffer to be one byte fuller than it should be, and therefore causing the last byte of a new observation to be dropped. In order to work around this, a '`new_vector_incoming`' flag is added, which will empty the buffer when asserted by the processor. Another (possibly better way) of avoiding errors would be to automatically empty the buffer if new data is not received for a time-out period. However, this is left as a possible future enhancement, as the `new_vector_incoming` flag is sufficient for now.

The Baudticker module generates clock signals for the transmit and receive modules. The transmit module requires a signal at approximately 115200Hz, but the receive module requires a signal at 8 times that frequency. The receiver oversamples the input `rx` signal at this higher frequency, which enables it to detect and ignore signal glitches.

The Sender module is fairly simple in its operation – it loops through all the senone scores stored in external SRAM and sends them over UART to the processor.

4.4 L'Imperatrice (The processor)

L'Imperatrice performs these main tasks:

- Read speech data from a WAV file.
- Pre-process – pre-emphasise and window data, calculate FFT, calculate MFCCs, 'Lifter' MFCCs.
- Convert MFCCs to the correct binary format.
- Send this observation vector to La Papessa.
- Receive and display scores.

4.4.1 Pre-processing

Instead of reading data in from a microphone, the system uses pre-recorded speech stored in WAV formatted audio files. This method was chosen because it allows the pre-processing to be very easily tested, without worrying that the input data was

changing. In addition, it allowed the results to be compared with pre-processed data from other libraries, such as the HTK, in order to verify correct operation.

The audio files were prepared specially, in a format that is simple to read and use. The audio manipulation software 'Audacity'[1] was used to record and store speech as uncompressed (Microsoft) WAV files with unsigned 8-bit PCM encoding. As Audacity does not support saving files with 8kHz sampling rates, the 'sox'[2] utility was used to downsample them from 48kHz to 8kHz.

In pre-processing the data, an attempt was made to match the processes that the HTK used as closely as possible, thus making it easier to verify that the process works correctly. Samples from the audio files are read in sequential blocks of between 80 and 200 samples, depending on the window size required. These blocks are windowed with a Hamming window to remove discontinuities at the edges which could cause excess spectral noise. The DFT is taken of this data, using the FFTW library, which had to be specially packaged and compiled for use in LTIB (See Appendix C.2.3). Finally, the magnitude is taken, so that the spectral data is fully real and may be used to calculate Mel Frequency Cepstral Coefficients (MFCCs). An external library (LibMFCC) was used for this purpose, due to its availability and the time constraints on the project. The last operation, shown in Equation 4.1, is to perform Liftering[3] on each of the coefficients, where L is a parameter. This results in the cepstral coefficients having similar magnitudes [19], which is particularly convenient when they must be represented with a 16-bit fixed-point number.

$$c'_n = \left(1 + \frac{L}{2} sin \frac{\pi n}{L}\right) c_n \tag{4.1}$$

4.4.2 GPIO and Application UART

In order to communicate with the FPGA, the processor required access to a serial port. The ideal solution is the built-in Application UART port on the iMX23 chip, which only needs configuring. In addition, access to GPIO from inside a C program was required to assert the `new_vector_incoming` signal.

Both GPIO and Application UART must be enabled by selecting the relevant entry in the Kernel configuration menu at compile time (See Appendix C.2). Using

[1] Audacity is available at http://audacity.sourceforge.net/, last checked April 2013.
[2] SOX (Sound eXchange) is available at http://sox.sourceforge.net/, last checked April 2013.
[3] The name "Liftering" comes from the HTK book [19]

the Application UART from a C program requires opening the serial port file (`/dev/ttySP1`, usually) and using the `termios.h` library to configure it correctly. The options used to configure the serial port are shown in Listing 4.2.

```
/* Set important serial port parameters: */
stty.c_cc[VMIN] = 0;                          // No blocking read
stty.c_cc[VTIME] = 1;                         // 0.1s: Max wait for data
stty.c_cflag = (stty.c_cflag & ~CSIZE) | CS8; // 8 bit chars
stty.c_iflag &= ~IGNBRK;                      // Ignore break commands
stty.c_lflag = 0;
stty.c_oflag = 0;
stty.c_iflag &= ~(IXON | IXOFF | IXANY);      // No flow ctrl
stty.c_cflag |= (CLOCAL | CREAD);             // enable receiver
stty.c_cflag &= ~(PARENB | PARODD);           // No parity
stty.c_cflag &= ~CSTOPB;                      // Send 1 stop bit
stty.c_cflag &= ~CRTSCTS;
```

Listing 4.2: Serial Port Configuration

The GPIO is accessible in more than one way – either through sysfs or via direct register access. The kernel is configured to include the sysfs GPIO interface, which creates an entry under `/sys/class/gpio` that allows GPIO pins to be written and read as standard files. Alternatively, the 'Olinuxino' board project (which uses the same processor as L'Imperatrice) includes C code to directly write and read the GPIO registers as mapped memory[4]. The project uses the second method, as it was straightforward to adapt the existing code for the purposes of the project. In addition, direct memory access is far faster than the sysfs gpio interface; in tests, it was capable of switching frequencies up to 2MHz while the sysfs interface only managed about 200Hz.

4.5 Support Software

A set of utilities were written in Common Lisp in order to facilitate and accelerate various parts of the project development. Common Lisp was chosen due to its extreme power and flexibility, and because the IDE used, Emacs SLIME, is considered superior to any other. In particular, these utilities helped with the following tasks (See Appendix E for documentation):

- Parsing HMM definitions created by HTK, and extracting senone parameters
- SystemVerilog testbench generation

[4] Accessible on the Olinuxino Github repository at https://github.com/OLIMEX/OLINUXINO/

- Senone data file generation
- Verifying hardware functionality
- Number format conversion tasks

The HTK stores HMM definitions (along with transition matrices and senone definitions) in a human-readable plain-text Ascii file. However, due to the very large model size, it was impractical to attempt to copy out parameters by hand. For this reason, a parser was created that read an HMM definition file and stored the extracted data in a useful structure. Another set of utilities was created to aid converting between floating point numbers and the custom fixed-point representations used.

Using these two sets of utilities, it was possible to automatically generate C header files and SystemVerilog modules containing the parameters, in a suitable format. In addition, it became possible to easily and quickly generate testbenches to test the GDP with many different senones.

Chapter 5

System Testing and Analysis

The testing methods and results are documented in this chapter. The two primary blocks, described in the last chapter, were tested independently and then together, and will be presented here in the same manner. A wide range of tools and techniques have been used to perform the testing, and to identify and fix errors encountered.

5.1 FPGA Design and Test Methodology

There were several phases of development, and thus testing, of this part of the design. Initially, the full system was written and simulated in ModelSim, which supports SystemVerilog assertion based verification. Each of the components of the design was built and tested separately, and then incrementally joined together and simulated as a whole. Finally, the modules were tested in hardware – first individually, and then as a full system.

A set of testbenches were developed for use within ModelSim, which tested each of the modules used by the system. Most of the modules were fairly simple to simulate and verify, as the expected behaviour is generally constant and determinate. For example, verifying the UART clock generator was a case of instantiating it, asserting the enable signal, and verifying the frequency of the output signals. However, some modules and their testbenches deserve extra attention, as special methods were used to test them. In several areas, the benefits of using SystemVerilog become clear, as assertions greatly simplified the validation and testing process.

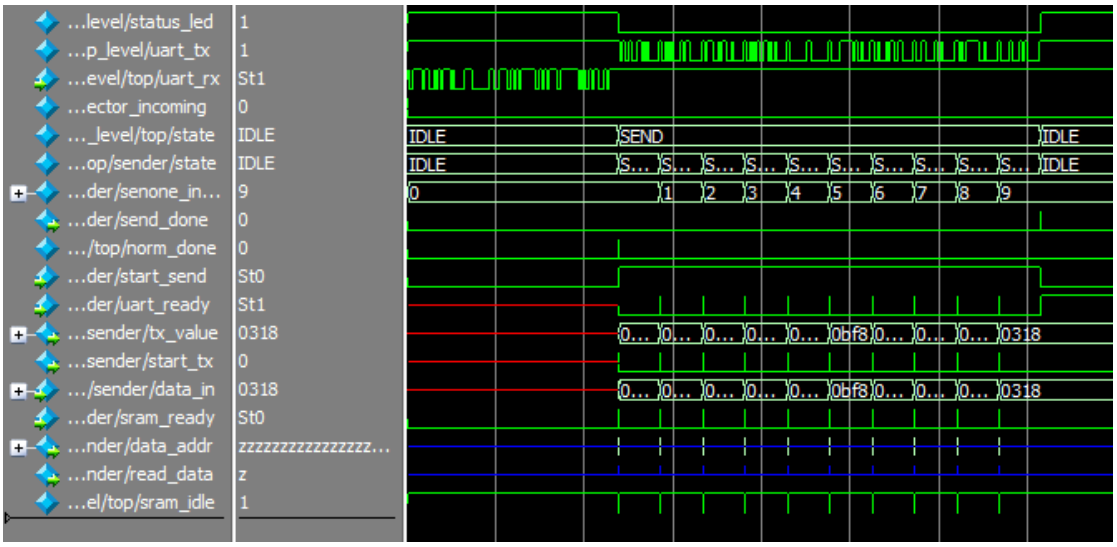

Figure 5.1: Simulation waveforms showing a full cycle of the top level state machine

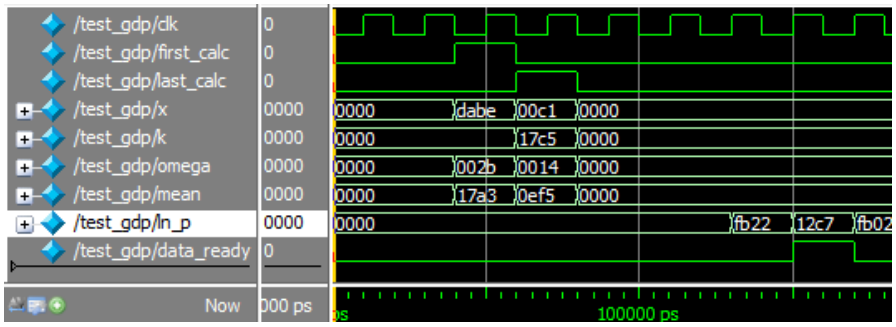

Figure 5.2: GDP testbench waveform

Figure 5.1 shows a simulation of the system performing a full cycle, from receiving an observation to sending back scores.

5.2 Gaussian Distance Calculation Block

The Gaussian Distance Pipe was initially tested by creating a testbench which provided the sequence of inputs that the GDP required, and then asserted that the correct score was provided. Figure 5.2 shows the correct operation of this module, with a set of test data being used as inputs to the module. The expected results were determined using the software GDP and the binary conversion software described in Section 4.5, thus confirming the validity of the results produced.

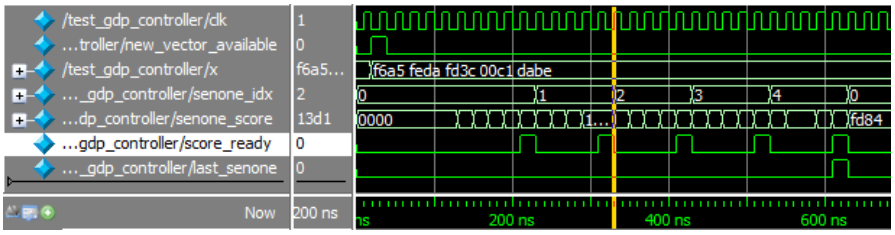

Figure 5.3: GDP Controller testbench waveform and error messages

The Gaussian Distance pipe controller testbench gives a new observation to the controller, and then watches as senone scores are produced by the module. SystemVerilog assertions are used to check whether correct scores are arriving at the time they should. However, in order to test a large number of different inputs (which would result in different score outputs), a set of software utilities were written to automatically generate the testbench code. This software made use of the HMM definition parser and a software version of the GDP that had already been created. This allowed a very large number of senones to be tested in a matter of minutes, and determine how well the GDP pipe was working. Figure 5.3 shows the GDP Controller being tested.

Being able to easily test many senones was important because the use of fixed-point arithmetic inevitably causes numerical errors that vary with the operations and numbers being used. The GDP may produce the correct result for one set of inputs, but another set of inputs may produce an erroneous result that was caused by the fixed-point number not having high enough precision. In fact, in most cases, the least significant bits of the result were wrong. Because of this, it was important to determine the distribution of the error magnitudes, in order to decide whether the number format needed changing. Automatic testbench generation made this far easier, as testbenches could be created which automatically displayed which senone scores where wrong, and by how much. The final system tolerated errors in the last 6 bits of the result, as this corresponds to less than 0.5% of the total value.

From Figure 5.3, it is also possible to observe the (ideal) time the GDP takes to calculate a score. In this case (when `n_components=5`), each senone takes 5 clock cycles (100ns with a 50MHz clock), after an initial pipeline loading delay.

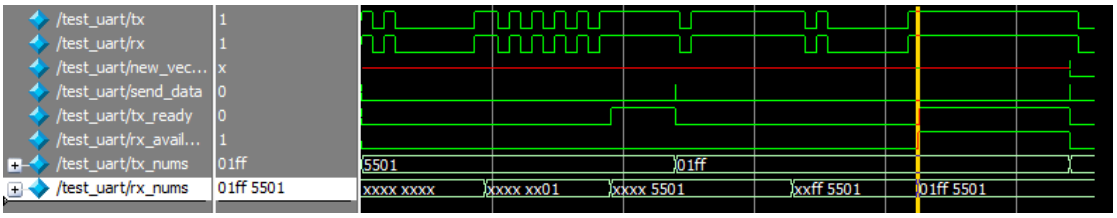

Figure 5.4: UART testbench waveform

5.2.1 Synthesis and Hardware Testing

Unfortunately, due to the small size of the FPGA used, the initial synthesised design did not fit on it. The full Voxforge model had over 7000 senones, and each observation had 25 parameters of 2 bytes each – over 700kB of storage would be required for the full model. As this far exceeds the resources available, the model size was reduced for the project. The number of statistical parameters was reduced to 6, and only 12 senones were scored at once, bringing the FPGA slice usage to about 90%.

A variety of tools were used to repeatedly test the functionality of this hardware. A Bus Pirate[1] was used to communicate with the FPGA, which allows hexadecimal values to be easily sent and received over UART. In order to debug the SystemVerilog, a large number of signals were routed to output pins, which were then monitored with a Saleae Logic analyser.

5.3 UART Communications

Simulating and testing the UART module was done by creating two instances of the module, and cross-connecting their RX and TX pins. This allowed both transmit and receive to be tested, and confirmed correct operation. An example of this testbench running is shown in Figure 5.4.

After simulation, the UART module was finally verified by programming the FPGA with the UART module, and a simple controller which echoed back whatever it received. An FTDI USB to serial cable connected the FPGA to a computer, so that the setup could be confirmed to work with a real UART connection. In

[1]Multi-purpose debugging tool that supports many different protocols. See http://dangerousprototypes.com/docs/Bus_Pirate

addition, a Saleae Logic analyser was used to examine and ensure the absence of glitches in the UART signals generated by the FPGA.

5.4 SRAM Access

A custom SystemVerilog module was designed in order to test writing and reading values to the on-board SRAM chip, as it was a completely untested part of the board. The module performed fairly basic operations, such as looping through a set of values, writing them to SRAM, and then reading them out again. Debug information was displayed on the output pins, allowing the process to be monitored. This confirmed that the chip worked as expected, and would be usable for the project.

5.5 Pre-Processing

There were several different stages to the pre-processing, and these were tested individually. First and foremost, the FFT code was tested by using input data containing known sinusoids. The library used, FFTW, is well established and proven to work, and so the primary purpose here was to test the speed of the FFT. It ran almost instantaneously (execution took 0.000000s, according to the Kernel time.h library), giving the correct results.

The other major operation that required extensive testing was the MFCC calculation. The other operations, pre-emphasis, windowing and liftering, are fairly simple mathematical operations that are easy to verify. One of the desired properties of the system is that it would produce MFCCs that matched those produced by the HTK. Unfortunately, this was not quite achieved. In most tests, the energy coefficient matched, but the others were relatively different. This implies either that there is a limitation or problem with the MFCC library used, or that the HTK performs extra (or different) pre-processing.

In addition, the library is very inefficient, causing the MFCCs to be produced relatively slowly.

5.6 Software Based GDP Speed

The Gaussian Distance calculations were implemented in a C program in order to compare its performance with that of the FPGA. It was found that the processor takes, on average, about $10\mu s$ to calculate a single senone score – about 100 times slower than the FPGA. However, this version used double floating point precision, and thus is far more accurate than the FPGA. Reducing the accuracy and using fixed-point values would definitely increase its speed.

Chapter 6

Project Evaluation and Reflection

This chapter presents an analysis of the final solution, evaluating its primary strengths and weaknesses, and the progress as a whole.

6.1 Analysis of Solution

6.1.1 The FPGA

The FPGA used was very small, and the full required design could not fit on it. In particular, the size of the model had to be reduced, so that each Gaussian mean and variance had fewer than 25 components, and not all 7000 senones were processed. However, what has been implemented is adequate to prove the concept, and demonstrate the advantages gained from using an FPGA. In addition, it was known from the start that the full model would never fit, and so although this is a practical limitation, it does not make the project less worthwhile.

Various benchmarks have been presented in the form of timing data. It was shown that the FPGA was capable of calculating senone scores far faster than the traditional processor, albeit with an acceptable loss of accuracy. This is similar to results achieved by Melnikoff [11] and Speech Silicon [14], and supports the principle that custom-made hardware is faster than general purpose processors. For real-time speech recognition, getting the senone scores as fast as possible is very important, as it allows more time for decoding tasks. Thus, using an FPGA may be beneficial, especially in embedded systems. It is possible to extrapolate, from the data gathered, the system speed when a larger model is implemented. If the

full model (7000 senones, 25 components) was used, the scores would be computed in about 3.5ms, leaving 6.5ms (of a 10ms window) for pre-processing and decoding (this excludes communications time). If the embedded process was used to calculate the scores, it would take about 150ms, although the scores would be more accurate.

A big disadvantage of using an FPGA, in such a system, is the need to communicate to it. The communication is an added step that adds time to the senone scoring process, and, if it is not fast enough, may render the use of an FPGA not worthwhile. The implemented communication method (UART) is extremely slow and is the weakest point of the system, as is obvious in Figure 5.1 which shows a complete cycle of the top level state machine. Although it is not visible, the state machine goes through PROC and GDP, but these stages are far faster than the UART communication. UART was used primarily for the ease of implementation, and because at this stage real-time operations are not required. However, for this system to be realistically useful, a better communication method needs to be developed – either a far faster serial bus or some form of hybrid serial-parallel connection.

6.1.2 The Processor

To be used in a real system, an important requirement is that all the pre-processing be much faster than the frame period. With the current design, the only component breaking this requirement is the MFCC computation, due to the library's slowness and inefficiency. All the other steps take a tiny fraction of the frame period, allowing a large amount of time for processing, and in later designs, decoding. The external LibMFCC library is not optimised for the project's requirements at all, but it was convenient, and served its purpose. The HTK has a far faster implementation of this step, and could have been ported to the project code, had more time been available.

However, what has been implemented is a good demonstration of the capabilities of this processor, and how it is capable of working with the FPGA.

6.2 Deviations from Original Goals

Originally, while planning the project, the goal had been to implement a complete speech recognition system, from pre-processing to Viterbi decoding (see Appendix A). However, as more was learnt about the systems involved, it became clear that it was far too large a subject to attack in a single project. Thus, the biggest change from initial plans was to narrow the project's focus down to a particular area of speech recognition. However, with respect to the focussed goal, the results achieved are very satisfactory.

Development began by following the Gantt chart in Appendix B.1, which was also presented in the project Interim Report. However, when building a software decoder proved to be far more time consuming than expected, the decision was made to change the focus, as mentioned above. The remaining progress followed the Gantt chart in Appendix B.2[1]. The majority of the design and implementation was completed in a timely manner, thus showing that the revised goals were more realistic.

[1]Both Gantt charts were created and modified using Gantter (http://app.gantter.com)

Chapter 7

Conclusions and Further Work

This report presented a proof of concept system for carrying out speech recognition related operations on embedded hardware. The system is built on two boards from the Micro Arcana family of development boards, and collectively performs two fundamental tasks:

- Speech pre-processing, with Mel Frequency Cepstral Coefficients as the final observation vector.
- Scoring the states of HMMs in a speech model, given this observation vector.

A background of the relevant speech recognition theory was given, along with evaluations of various other techniques and approaches. Descriptions and analyses of the completed design were presented, along with information on how it was tested. In addition to the hardware based system, a multi-purpose software toolkit was created that greatly helped during the design and testing stages of the project.

The principle goal was to design and implement part of an HMM based speech recognition system in embedded hardware, in order to evaluate and learn about such a system. This has been fairly successful; the conclusions are given below.

7.1 Usefulness of Results

Due to size limitations of the FPGA, it was not possible to use a complete HMM speech model with the implemented system. However, the intention was never

to build something that was usable with such a model, but rather to attempt to judge the usefulness of an FPGA in this situation. With the results achieved it was possible to estimate the relationship between the system speed and model size, and thus show that the FPGA was capable of producing results faster than a traditional processor.

Furthermore, this project is a valuable example of how two of the Micro Arcana family may be connected and used in a practical setting. Some of the problems encountered are certainly not unique to this design, and therefore the solutions presented may be helpful in other projects.

As a development platform, there is potential for further investigation into embedded speech recognition using the two systems built here. Indeed, there are several different aspects of the design that may be expanded upon or improved, as described in the further work section. This is not a failing of the current project; speech recognition is such a large and complex area that only a small part of it could possibly be implemented in the time available.

7.2 Future Work

There are many possibilities for developing this project further – either by improving the system already built, or implementing other speech recognition tasks such as decoding. With the system in its current state, there are a few areas that may be interesting to investigate.

Primarily due to time constraints, the Mel Frequency Cepstral Coefficients were computed using an existing library (LibMFCC). However, LibMFCC is extremely slow, and should definitely be optimised. In addition, it would be very beneficial if the pre-processing generated the same MFCCs as those generated by the HTK. This would require an in-depth analysis of the HTK system in order to determine the exact sequence of operations that are performed.

Communications between the boards needs improvement before the system can be considered realistically usable. Better communications would need to be far faster, and possibly include features such as error-checking, hand-shaking, and control sequences. Designing a better communications method between two of the Micro Arcana boards would be an interesting project, due to the size and speed constraints of the family.

This project only implemented the first stages of a speech recognition engine, and did not at all focus on tasks such as decoding or language modelling. However, it provides an environment where these systems may be implemented and used. There is huge potential for further work that could focus on any one of these areas.

References

[1] Personal interview, srinandan dasmahapatra, Nov 2012.

[2] Voxforge, 2012. URL http://www.voxforge.org/.

[3] Tor Aamodt. A simple speech recognition algorithm for ece341, 04 2003. URL http://www.eecg.toronto.edu/~aamodt/ece341/speech-recognition/.

[4] J.A. Bilmes. What hmms can do. *IEICE TRANSACTIONS on Information and Systems*, 89(3):869–891, 2006.

[5] Linguistic Data Consortium. Timit acoustic-phonetic continuous speech corpus. URL http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC93S1.

[6] SJ Cox and G. Britain. *Hidden Markov models for automatic speech recognition: theory and application*. Royal Signals & Radar Establishment, 1988.

[7] Sadaoki Furui. *Digital Speech Processing, Synthesis, and Recognition*. Marcel Dekker, 1989.

[8] S.K. Gaikwad, B.W. Gawali, and P. Yannawar. A review on speech recognition technique. *International Journal of Computer Applications IJCA*, 10(3): 24–28, 2010.

[9] Chunrong Lai, Shih-Lien Lu, and Qingwei Zhao. Performance analysis of speech recognition software. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2002.

[10] Tomoko Matsui and Sadaoki Furui. Comparison of text-independent speaker recognition methods using vq-distortion and discrete/continuous hmms. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 2, pages 157–160. IEEE, 1992.

[11] S.J. Melnikoff. *Speech recognition in programmable logic*. PhD thesis, University of Birmingham, 2003.

[12] S. Nedevschi, R.K. Patra, and E.A. Brewer. Hardware speech recognition for user interfaces in low cost, low power devices. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 684–689. IEEE, 2005.

[13] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[14] J. Schuster, K. Gupta, R. Hoare, and A.K. Jones. Speech silicon: an fpga architecture for real-time hidden markov-model-based speech recognition. *EURASIP Journal on Embedded Systems*, 2006(1):10–10, 2006.

[15] CMU Sphinx. *Sphinx 3 System Design Documentation*. CMU Sphinx.

[16] Saeed V. Vaseghi. *Advanced Digital Signal Processing*. John Wiley and Sons, fourth edition, 2008.

[17] P.C. Woodland, J.J. Odell, V. Valtchev, and S.J. Young. Large vocabulary continuous speech recognition using htk. In *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, volume 2, pages II–125. IEEE, 1994.

[18] S.J. Young, NH Russell, and JHS Thornton. *Token passing: a simple conceptual model for connected speech recognition systems*. University of Cambridge, Department of Engineering, 1989.

[19] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, Valtcho Valtchev, and Phil Woodland. *The HTK Book*. Cambridge University Engineering Department, 2009.

Appendix A

Project Brief

The project brief is included here for reference.

Speech Recognition using a Xilinx FPGA and an ARM9 development board

In general, Speech recognition refers to the process of translating spoken words or phrases into a form that can be understood, which for an electronic system usually means using mathematical models and methods to process and then decode the sound signal. Speech recognition technology is commonly found in modern consumer electronics, and most people who own a computer or an Apple iPhone are aware of the capabilities. Most people are also very aware of the limitations of such systems. Most require extensive training to be effective, and even then often don't work well in noisy environments or when the speech is hurried or otherwise different from the training samples.

If speech recognition was broken down into two sections, they would be 'pre-processing' and 'decoding'. The decoding stage usually decides how fast the speech recognition engine is, as it usually involves fairly complex statistical calculations. The speed of speech recognition can be improved either by improving the decoding methodology, or by increasing the speed at which the current calculations are performed.

A series of electronic development boards (The 'Micro Arcana') is currently being designed by the University of Southampton in order to provide undergraduates with versatile and powerful prototyping platforms. However, only a few sample projects exist for these boards, and a greater number of examples would be useful

to demonstrate their abilities. This series includes an FPGA board (La Papessa), and an ARM9 processor based mini-computer capable of running Linux.

As an FPGA is capable of performing mathematical operations very fast, using one may be a way of increasing the speed of the decoding stags. In addition, Hidden Markov Models and Viterbi decoding (popular ways of decoding speech) benefit from a parallel architecture, such as that found on an FPGA. The La Papessa board will be used to perform this task. As the FPGA on it is relatively small, part of the pre-processing will be performed by the ARM board, which already has microphone-in circuitry.

In order to test the capabilities of the system designed, a set of speech samples will be prepared, and then used to test the FPGA based system against a software package. This software package will be run on the ARM board so that it runs at the same clock speed as the engine that will be created.

If the FPGA based system proves to be faster than a pure software implementation, it may be extended in several directions. Initially, it will be built to recognise a limited vocabulary of words, to test the effectiveness of the method. It may then be improved to perform more complex analysis on speech, such as language or grammar based recognition. In addition, an FPGA may be useful for voice recognition - the act of recognising a specific speaker, rather than the words.

Appendix B

Project Management

B.1 Interim Gantt Chart

This Gantt chart (Figure B.1) was also presented in the interim report, and proved to be far too optimistic about the time certain tasks would take.

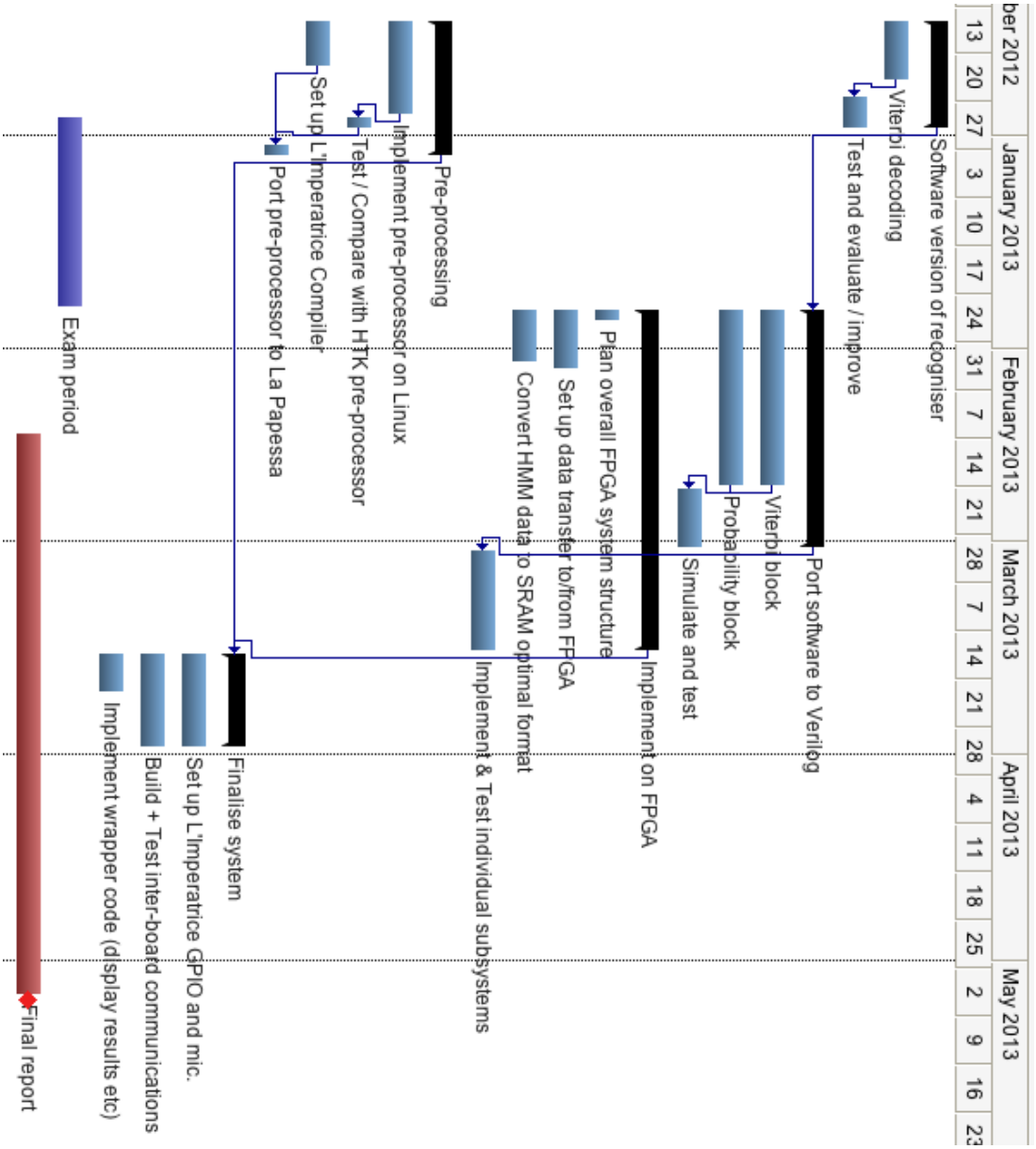

Figure B.1: Interim Gantt chart

B.2 Final Gantt Chart

This Gantt chart (Figure B.2) illustrates the rate of progress that was actually accomplished.

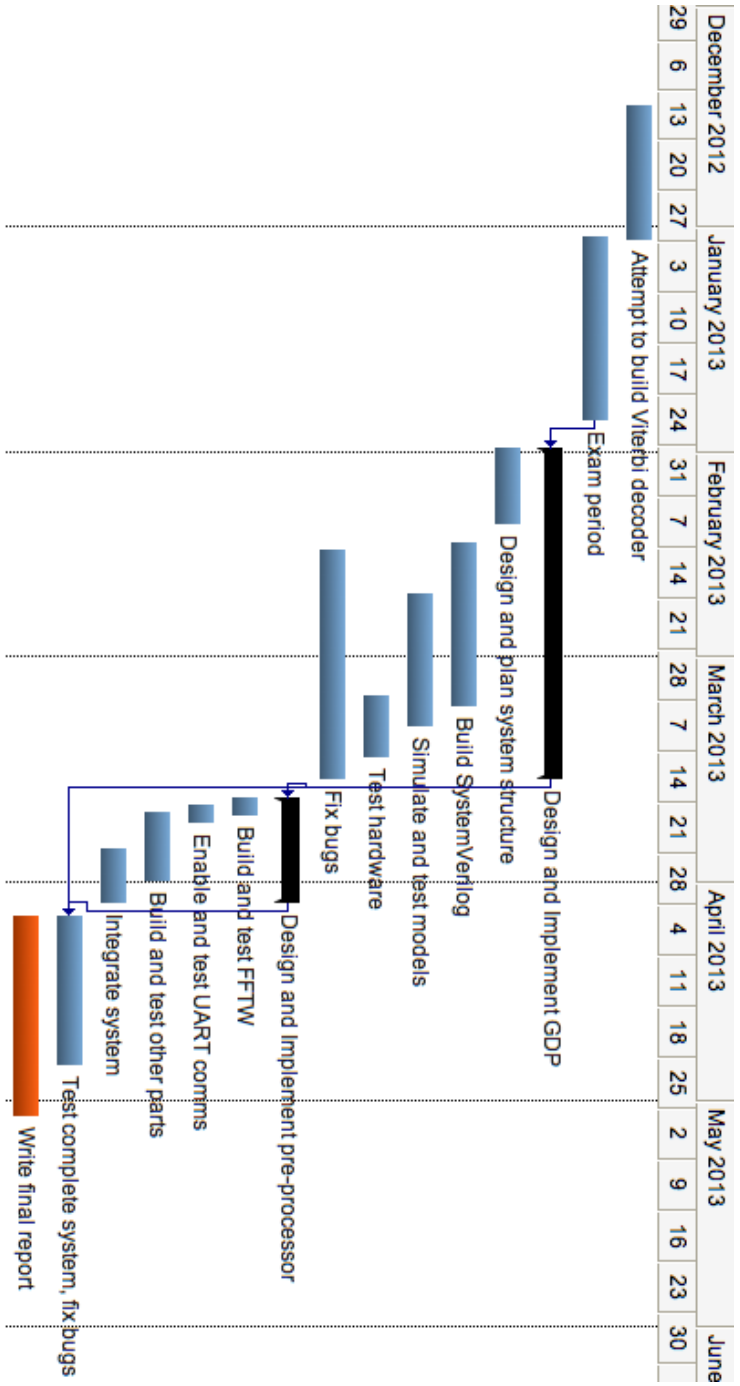

Figure B.2: Final Gantt chart

B.3 Git Commit Log

Below is a log of the commits made to this project's git repository.

(2013-04-24) Restructured SystemVerilog into Synplify and Modelsim projects
(2013-04-21) Added a program to test double to uint16_t
(2013-04-21) Wording tweaks, mainly additions to analysis.
(2013-04-21) Fixed software normalisation (reversed subtraction)
(2013-04-21) Added some appendix structure and content
(2013-04-21) Started adding a fixed point version to GDP-in-C
(2013-04-20) A bunch of writing, reorganisation, better appendices structure...
(2013-04-20) Updated to use more senone data
(2013-04-20) Added normalisation to feed_obs, reformatted verilog senone data
(2013-04-20) Fixed float-¿uint bug in D_TO_U16
(2013-04-20) Updated Main for the latest FPGA version
(2013-04-20) Added a Copy target, to simplify things...
(2013-04-19) Fixed bug in gconst extraction
(2013-04-18) Added a software GDP and GPIO testing code
(2013-04-18) Yay for writing
(2013-04-17) More writing...
(2013-04-16) Minor tweaks and patches to code, lots of report. Commit before refactoring Testing section.
(2013-04-09) Mainly changes to System and Approach
(2013-04-04) Added outlines to System, Testing, Appendices, and writing to Approach
(2013-04-04) Fixed typo in Abstract
(2013-04-03) More writing in Approach section. Mainly.
(2013-04-03) Built a version of the GDP in C, tweaks to binary and extract lisps
(2013-04-01) Removed mfc and wav files from git...
(2013-03-31) Added missing chapter containers, Started trying to write...
(2013-03-31) Added sampling from WAV files, tidied up system
(2013-03-28) More introduction. Mainly.
(2013-03-28) Tweaks to extract
(2013-03-28) Integrated files, created Makefile
(2013-03-25) Re-ran the voxforge adaptation, separated 48kHz wavs from the 8kHz

wavs
(2013-03-25) Added report structure and Tested FFTW
(2013-03-18) Fixed mistake in new vector pin #define
(2013-03-18) Send and receive working correctly
(2013-03-18) First attempt at the C application
(2013-03-14) Hardware working. Normalise disabled
(2013-03-13) Tweaks to plan after srg meeting
(2013-03-12) Made Synplify versions of system, works in hardware, sometimes
(2013-03-12) Added constraints list for La Papessa Rev C
(2013-03-11) Made send state machine. Full send test with sram model - working.
(2013-03-11) Added normaliser code, top level sram connections and sram test-bench
(2013-03-10) More SV, uart working now, started final report
(2013-03-08) Changed uart to use num buffers, added Max module, etc
(2013-03-08) Added top level SV code
(2013-03-07) Added testbench and data generation, finished gdp controller + tester
(2013-03-06) GDP works. Added more to binary utils
(2013-03-04) Started SV conversion, added binary lisp code
(2013-02-25) Added missing Verilog file
(2013-02-25) Added Verilog, improvements to export.lisp
(2013-02-12) More capabilities... or not
(2013-01-31) Big messy commit. Most important: extract.lisp, citations
(2012-12-12) Interim report, night before
(2012-12-12) Updated Gaussian algo to use precomputed values
(2012-12-11) Fixed error in bib
(2012-12-11) Additions to report and more citations
(2012-12-10) Added readme
(2012-12-10) First huge commit

Appendix C

Development Environment

C.1 FPGA design cycle

As mentioned in the report, Synplify Premier was used to synthesise the SystemVerilog code. Then, the ISE Webpack (including the Floorplanner software) was used to assign ports to physical pins, and generate programming files. Finally, urJTAG was used to actually program the device over JTAG.

In order to use Xilinx design tools with Synplify, the correct environment variable must be set. Under the Options menu, select 'P&R Environment Options'. The XILINX variable must be set to the path of the Xilinx ISE tools (e.g., C:\Xilinx\14.1\ISE_DS\ISE). From the same menu, go to the Xilinx Submenu, and now 'Start ISE Project Navigator' should work correctly.

In addition, the implementation options for Synplify must be correctly set – in particular the correct part must be selected, and appropriate timing constraints entered. The onboard 50MHz clock needs one such constraint.

Once this is set up, the design process roughly follows this sequence:

1. Write SystemVerilog code in Synplify.
2. Run Synplify implementation; fix errors until it synthesises correctly.
3. Open project with Xilinx design tools (via Tools menu).
4. Either manually edit the constraints file and add net to port mappings, or use the Floorplanner (post-synthesis) tool to do this.

5. Run Webpack implementation. If mapping fails, the design may be too big to fit on the device.
6. Generate .bit programming file.
7. Open the 'iMPACT' software (which comes with the ISE Webpack).
8. Create a new project, and select the .bit file created in the previous step.
9. From the Output menu, select create new SVF file.
10. Perform whichever actions required from the list on the left – programming the flash takes a long time, so if still in the testing stages, only run "Program FPGA".
11. End the SVF output (from the same menu as before). This step is very important!
12. Open urJTAG and connect to the programming cable.
13. Use 'svf programming-file.svf' to transfer the design to La Papessa!

As this process is fairly long, it may be useful to speed some parts up. In particular, instead of repeatedly assigning pins with Floorplanner, it's easier to copy the constraints file once everything is assigned correctly. Then when Webpack is opened again, and the constraints are automatically overwritten, simply copy in the correct pin assignments. The constraints used for this project are available in the file archive, in the DevEnv folder.

C.2 LTIB usage

C.2.1 Cross compiling

To compile code that will work on L'Imperatrice, the binaries installed by LTIB must be used. They are normally stored in:

```
/opt/freescale/usr/local/gcc-x-glibc-x-x/arm-none-gnueabi/
arm-none-linux-gnueabi/bin/
```

Running the following command will add this to the path ($ is the prompt):

```
$ PATH=/opt/freescale/usr/local/gcc-4.1.2-glibc-2.5-nptl-3/
arm-none-linux-gnueabi/arm-none-linux-gnueabi/bin/:$PATH
```

Then running 'gcc' will automatically use the arm version.

C.2.2 GPIO and UART

Both GPIO and UART are optional components that must be enabled in the LTIB configuration.

C.2.3 Compiling FFTW for LTIB

In many cases one may want to build a library or package to install on L'Imperatrice. For example, one may want to perform real time Fast Fourier Transforms using the FFTW library, which has native support for ARMv5 devices. The general procedure, as described in the FAQ at http://ltib.org/documentation-LtibFaq, is:

1. Prepare the source files
2. Build and install the package to the rootfs using LTIB
3. Add -I and -L flags to gcc to cross compile with the libraries that were just installed.

The FFTW source can be downloaded from http://www.fftw.org/download.html. In order to add it to LTIB, a custom RPM spec file was created, available in the project file archive in the DevEnv folder. Then the ltib binary is used to correctly configure and build FFTW from source, and add it to the list of installed packages. This allows FFTW to be selected in the 'Packages list' in the LTIB configuration menu, and thus installed in the rootfs.

When compiling a C program that uses FFTW, the relevant library and include paths must be added. In this project, a Makefile was used to automate the build process.

Appendix D

Voxforge

D.1 Audacity Recording

The settings used to record the speech samples using Audacity are shown below.

- Sampling rate: 48KHz
- Output format: Uncompressed Microsoft WAV, 16bit unsigned
- Output channels: 1 (mono)

D.2 Model Adaptation

A set of speech models were downloaded from the Voxforge website, which were analysed and used throughout the project. However, they were first adjusted to the author's voice, in order to gain familiarity with various tools and technologies involved.

A script to perform the adaptation was created, according to the Voxforge adaptation tutorial. The required operations are listed here.

Note: v3.2 HTK binaries should be used. Newer ones introduce a feature which makes these commands un-usable.

```
1 # Path to HTK binaries
2 HTK=~/Documents/University/Part3/Part-3-Project/HTK/htk-bin-3.2.1
3
4 # Downsample the recorded audio to 8kHz -> wav/sample{1-31}.wav
```

```
5  ./downsample.pl FilesToBeDownsampled wav 48000 8000
6
7  # Create the mfccs from the downsampled audio -> mfccs/sample{1-31}.mfc
8  $HTK/HCopy -A -D -T 1 -C wav_config -S codetrain.scp
9
10 # Re-Align the data
11 # Watch for errors in this!!! -> adaptPhones.mlf
12 $HTK/HVite -A -D -T 1 -l '*' -o SWT -b SENT-END -C config -H macros -H hmmdefs -i
       adaptPhones.mlf -m -t 250.0 150.0 1000.0 -y lab -a -I adaptWords.mlf -S
       adapt.scp dict tiedlist
13
14 # Build regression class tree -> hmm16/{hmmdefs,macros}
15 $HTK/HHEd -H macros -H hmmdefs -M hmm16 regtree.hed tiedlist
16
17 # Static adaptation time!
18 # 1. Global adaptation -> global.tmf
19 $HTK/HEAdapt -C config -g -S adapt.scp -I adaptPhones.mlf -H hmm16/macros -H
       hmm16/hmmdefs -K global.tmf -t 250.0 150.0 3000.0 tiedlist
20
21 # 2. Transform models -> hmmAdapt/{hmmdefs,macros}
22 $HTK/HEAdapt -C config -S adapt.scp -I adaptPhones.mlf -H hmm16/macros -H hmm16/
       hmmdefs -J global.tmf -M hmmAdapt -t 250.0 150.0 3000.0  -j 0.9 -i 10
       tiedlist
```

Appendix E

Support Software Documentation

This Appendix documents use of the Common Lisp utility tool-kit described in the report. All development was carried out in Emacs, with the excellent SLIME being the main reason to do so. The Common Lisp implementation used is SBCL, with Quicklisp used to install the only dependency, cl-ppcre. The two files containing these utilities are extract.lisp and main.lisp, included in the project file archive.

The best method of using these utilities is to load both files into the SLIME environment, using the slime C-c C-k command sequence. This will make all the functions and data available on the top-level prompt (REPL). Then they can all be used and adapted as necessary, without requiring any fancy packaging.

Much of the code is self documenting, so this chapter is primarily for overview.

E.1 Data structures

Four data structures were provided to store various parts that were extracted:

- `t-matrix`: Transition matrices
- `senone`: Senones
- `hmm`: Full HMM
- `oframe`: Observation frame

E.2 Parsing

The primary function used to parse an HMM definition file is 'parse-hmmdefs'. This takes a single argument – the path to a file. The function returns three collections; lists of hmms, transition matrices, and senones that were found in the file.

The function is written using a set of macros which add the ability to define 'blocks' in files, and easily search through a given file for these blocks. When a block is found, the required variables are automatically extracted and assigned to variables, which may then be used to populate data structures as desired.

In addition, an file with observation data may also be parsed, using the 'parse-input' function. This function assumes that the file was created using a command similar to:

```
HList -e 200 -i 26 -o -h sample1.mfc > sample1-data.txt
```

E.3 Binary utilities

A set of conversion utilities were created to aid moving from the custom binary fixed point notation used, and real floating point decimal values.

The important functions are 'bin2dec' and 'dec2bin' which perform the actions their names suggest. However, before they can be used, the binary format must be defined, by creating a set of 'bins' which represent the value stored by each position in a binary number. This is done with 'makebins'.

E.4 System Modelling

The GDP was implemented primarily so that the hardware could be easily tested. It is designed as a closure, so that subsequent calls to it behave as they would if it was a hardware pipeline.

Furthermore, various helper functions were built to automatically feed a list of observations into the GDP, along with a specified list of senones, and to return the correct scores.

E.5 Automatic file generation

Functions were built to automatically create files containing testbenches, and senone data.

The 'print-senone-data' function creates a file containing a set of senone data, in either C or SystemVerilog code.

The 'print-testbench' function builds a SystemVerilog testbench to test an observation vector. It performs all the necessary conversions from floating point to hex, and also prints out assertions to confirm the answers given.

Both of these functions use the 'printl' macro, which was designed to make it easier to print multiple lines into a stream.

Appendix F

File Archive Contents

Below is a full tree and description of the files included in the archive submitted along with this report.

`Source/GDP-in-C`
- A C program which calculates senone scores; built to get an idea of the time a conventional processor takes to perform the calculations.

`Source/GPIO-test`
- A program to help measure the switching speed of memory mapped GPIO.

`Source/Utilities`
- The Common Lisp software utilities. The HMM definitions used for the project are also in this folder.

`Source/C-backend`
- This is the main pre-processor application code. A Makefile is included to facilitate building the code, which should then run on L'Imperatrice. A sample WAV file is also included.

`Source/SystemVerilog-Main`
- This is the main SystemVerilog code that was built for the project.

`Source/SystemVerilog-Simulation`
- These are the testbenches and models required to test the modules provided in SystemVerilog-Main

`Binaries/`
- This folder contains a set of pre-compiled binary files and SVF files which may

be transferred to L'Imperatrice or La Papessa without modification. This includes binaries for all the programs stored in the Source directory. There are two SVF files – one writes the program to flash on the FPGA; the other is only a temporary (fast) write.

`DevEnv/`
- This folder contains various files that were important in the development environment used.

Below is a tree of the files included in the archive.

```
Archive
          Binaries
                  fftw-test
                  gdp-test
                  gpio-test
                  pre-process
                  top-level-12s6.svf
                  top-level-12s6c-flash.svf
          DevEnv
                  La Papessa Contraints.txt
                  fftw.spec
          Source
              C-backend
                      Makefile
                      gpio.c
                      gpio.h
                      libmfcc
                              libmfcc.c
                              libmfcc.h
                      main.c
                      main.h
                      obj
                              libmfcc
                      preProcess.c
                      preProcess.h
                      sample.c
                      sample.h
                      sample1.wav
                      serial.c
                      serial.h
              FFTW
                      fftw_test.c
                      libmfcc.c
                      libmfcc.h
              GDP-in-C
                      gdp-test.c
              GPIO-test
                      gpio.c
                      gpio.h
                      main.c
              SystemVerilog-Main
                      all_s_data.sv
                      gdp.sv
                      gdp_controller.sv
```

```
                        max.sv
                        normaliser.sv
                        send.sv
                        sram.sv
                        top_level.sv
                        uart
                                baudgen.sv
                                uart_rx.sv
                                uart_tx.sv
                        uart.sv
                SystemVerilog-Simulation
                        sram_model.sv
                        test_baudgen.sv
                        test_gdp.sv
                        test_gdp_controller.sv
                        test_send.sv
                        test_top_level.sv
                        test_uart.sv
                Utilities
                    extract.lisp
                    hmmdefs
                    main.lisp

14 directories, 51 files
```