

# Energy-scalable Speech Recognition Circuits

by

Michael Price

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

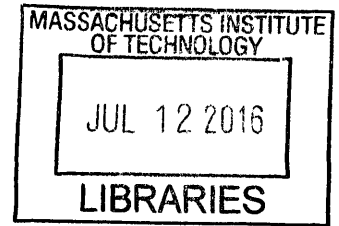
Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.



Signature redacted

Author .....  
Department of Electrical Engineering and Computer Science  
May 19, 2016

Signature redacted

Certified by .....  
Anantha Chandrakasan  
Vannevar Bush Professor of Electrical Engineering and Computer Science  
Thesis Supervisor

Signature redacted

Certified by .....  
James Glass  
Senior Research Scientist  
Thesis Supervisor

Signature redacted

Accepted by .....  
Professor Leslie A. Kolodziejski  
Chair, Department Committee on Graduate Students



# Energy-scalable Speech Recognition Circuits

by

Michael Price

Submitted to the Department of Electrical Engineering and Computer Science  
on May 19, 2016, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

## Abstract

As people become more comfortable with speaking to machines, the applications of speech interfaces will diversify and include a wider range of devices, such as wearables, appliances, and robots. Automatic speech recognition (ASR) is a key component of these interfaces that is computationally intensive. This thesis shows how we designed special-purpose integrated circuits to bring local ASR capabilities to electronic devices with a small size and power footprint.

This thesis adopts a holistic, system-driven approach to ASR hardware design. We identify external memory bandwidth as the main driver in system power consumption and select algorithms and architectures to minimize it. We evaluate three acoustic modeling approaches—Gaussian mixture models (GMMs), subspace GMMs (SGMMs), and deep neural networks (DNNs)—and identify tradeoffs between memory bandwidth and recognition accuracy. DNNs offer the best tradeoffs for our application; we describe a SIMD DNN architecture using parameter quantization and sparse weight matrices to save bandwidth. We also present a hidden Markov model (HMM) search architecture using a weighted finite-state transducer (WFST) representation. Enhancements to the search architecture, including WFST compression and caching, predictive beam width control, and a word lattice, reduce memory bandwidth to 10 MB/s or less, despite having just 414 kB of on-chip SRAM. The resulting system runs in real-time with accuracy comparable to a software recognizer using the same models. We provide infrastructure for deploying recognizers trained with open-source tools (Kaldi) on the hardware platform.

We investigate voice activity detection (VAD) as a wake-up mechanism and conclude that an accurate and robust algorithm is necessary to minimize system power, even if it results in larger area and power for the VAD itself. We design fixed-point digital implementations of three VAD algorithms and explore their performance on two synthetic tasks with SNRs from -5 to 30 dB. The best algorithm uses modulation frequency features with an NN classifier, requiring just 8.9 kB of parameters.

Throughout this work we emphasize energy scalability, or the ability to save energy when high accuracy or complex models are not required. Our architecture exploits scalability from many sources: model hyper-parameters, runtime parameters such as beam width, and voltage/frequency scaling. We demonstrate these concepts with results from five ASR tasks, with vocabularies ranging from 11 words to 145,000 words.

Thesis Supervisor: Anantha Chandrakasan

Title: Vannevar Bush Professor of Electrical Engineering and Computer Science

Thesis Supervisor: James Glass

Title: Senior Research Scientist



## Acknowledgments

All of the acknowledgments in my Master's thesis<sup>1</sup> remain valid with equal weight.

Thanks to my sponsors for making this research possible. My first year was supported by a Joan and Irwin Jacobs fellowship. The remainder of the work was sponsored by Quanta Computer Inc. as part of the Qmulus Project. Terence Huang, Joe Polifroni, Jamey Hicks, Rongquen Chen, and Ted Chang were very supportive of the work and offered insight on the use cases for our technology. The company provided extreme hospitality on our two trips to their headquarters in Taiwan, which were rare opportunities for someone who had never left North America. TSMC provided access to their 65 nm process technology and libraries, and fabricated our chip designs through their University Shuttle Program. Synopsys, Cadence, and Mentor Graphics provided the EDA tools used to design and verify these chips; Xilinx provided FPGA boards and synthesis tools for prototyping. Thanks also to the corporate members of the Center for Integrated Circuits and Systems (CICS), who are always engaging at the biannual reviews, and probably paid for part of this research somehow.

Anantha Chandrakasan was the first to invite me back to MIT and the prime motivator of this work. He allowed me to choose from a wide variety of research topics and granted broad freedom in my day-to-day work, which is the primary reason I went to graduate school. He has a strong vision for each of his many research efforts and high (perhaps unrealistic, but well-intentioned) expectations for his students to follow. Working with Anantha has provided a window into the life of a highly successful and globally respected leader—probably the hardest working person that I know.

Jim Glass has been a statesman and a role model as a research advisor. He agreed to collaborate with Anantha and myself in the unfamiliar and caveat-rich world of hardware. He would meet with me every week, discussing the technical roadblocks that we needed to overcome or preparing for interactions with our sponsors. He helped drive me towards higher application performance (e.g., accuracy), making our circuit contributions more valuable. And he organized events, from reading groups to backyard barbecues, that perpetuated a sense of community.

Victor Zue brought patience and wisdom as a member of the thesis committee, helping me present the work effectively for the long term. He provided a steady stream of high-quality pastries and tea leaves for the graduate students, which were a pleasant surprise (even as someone who has become desensitized to free food). He also arranged the productive collaboration between MIT and Quanta, of which this research was a small part.

---

<sup>1</sup>Please see <http://hdl.handle.net/1721.1/52771>.

Vladimir Stojanovic was an ideal graduate counselor, having supervised my Master's thesis work and recommended that I pursue graduate school (which I belatedly accepted). Peter Hagelstein offered considerate advice and moving stories from his colorful career. Vivienne Sze served on my RQE committee and offered occasional advice on research and career choices. Najim Dehak taught me a lot about audio classification and provided frequent entertainment, even audible from long distances.

The support staff at MIT make students' lives easier and also more entertaining. Margaret Flaherty and Marcia Davidson were both highly effective and professional administrators, in their own unique ways. Mike McIlrath has a difficult yet crucial job as MTL's foundry liason and CAD software administrator. Mike Hobbs and Bill Maloney (MTL) and Scott Cyphers, Steve Ruggiero, Jonathan Proulx, and Garrett Wollman (CSAIL) assisted with compute and network infrastructure.

It's been a pleasure working with my fellow students in Anantha's group and the SLS group. As a part-time member of both groups, I was able to participate in all of the parties but avoid any responsibilities. I'm fortunate to consider you friends as well as coworkers. Frank Yaul has been a polymath and inspiration, always ready to discuss anything from inference algorithms to PCB designs, or escape to rock climbing or ice skating. Dave Harwath, Ekapol Chuangsuwanich, and Stephen Shum helped with designing and understanding speech algorithms, and also convinced me to drink beer. Mehul Tikekar, Priyanka Raina, Chiraag Juvekar, and Arun Paidimarri helped with the difficult difficulties of digital circuit design. My research program was fairly self-contained (isolated), making it especially rewarding to collaborate with Bonnie Lam, Xinkun Nie, Skanda Koppula, Mandy Korpusik, Eann Tuan, Patricia Saylor, and Sumit Dutta on their projects. Many others, including my office mates Ann, Xue, Yonatan and Jen, and lab mates Phil and Nachiket, made my days more interesting. I won't truly celebrate until all of you have graduated.

Other people at MIT who have rounded out my career (i.e., provided excellent distractions from research) include Ed Moriarty of the Edgerton Center, and Ken Stone and Hayami Arakawa of the Hobby Shop. (My apologies for deserting them, as I have "graduated" to my own household electronics and wood-working facility.)

My wife Michelle has surprised me with her continued love, and provided a world apart from the noise and confusion of daily life that I can retreat to.

To everyone acknowledged here, I offer my appreciation and a permanent invitation to meet again.

# Contents

<b>Glossary</b>	<b>18</b>
<b>1 Introduction</b>	<b>23</b>
1.1 Motivation . . . . .	23
1.2 Scope . . . . .	24
1.3 ASR formulation . . . . .	26
1.4 Prior art . . . . .	30
1.5 Contributions . . . . .	31
1.6 Guide to chapters . . . . .	32
<b>2 Acoustic modeling</b>	<b>33</b>
2.1 What is the most accurate acoustic model with 1 MB of parameters? . . . . .	34
2.1.1 Gaussian mixture model (GMM) . . . . .	35
2.1.2 Subspace Gaussian mixture model (SGMM) . . . . .	37
2.1.3 Deep neural network (DNN) . . . . .	39
2.1.4 Model comparison and discussion . . . . .	41
2.2 Neural network implementation . . . . .	43
2.2.1 Architecture and general tradeoffs . . . . .	43
2.2.2 Scalability . . . . .	45
2.2.3 Sequencer and EU design . . . . .	46
2.2.4 Sigmoid . . . . .	49
2.3 Summary . . . . .	52
<b>3 Search</b>	<b>53</b>
3.1 Architecture . . . . .	54

3.2	Building blocks . . . . .	57
3.2.1	Hash table . . . . .	57
3.2.2	State list . . . . .	59
3.2.3	WFST . . . . .	60
3.2.4	Beam width control . . . . .	67
3.2.5	Word lattice . . . . .	69
3.3	Interactions . . . . .	75
3.4	Performance and scalability . . . . .	75
3.5	Summary . . . . .	78
<b>4</b>	<b>Voice activity detection</b>	<b>79</b>
4.1	Accuracy influences power consumption . . . . .	79
4.2	Algorithms . . . . .	80
4.2.1	Energy-based (EB) . . . . .	80
4.2.2	Harmonicity (HM) . . . . .	82
4.2.3	Modulation frequencies (MF) . . . . .	82
4.3	Architecture . . . . .	85
4.3.1	Top level . . . . .	85
4.3.2	Energy-based (EB) . . . . .	86
4.3.3	Harmonicity (HM) . . . . .	87
4.3.4	Modulation frequencies (MF) . . . . .	88
4.4	Performance comparisons . . . . .	89
4.5	Summary . . . . .	92
<b>5</b>	<b>Infrastructure</b>	<b>93</b>
5.1	Circuits . . . . .	93
5.1.1	Front-end . . . . .	93
5.1.2	ASR control . . . . .	95
5.1.3	Supervisory logic . . . . .	96
5.2	Design tools . . . . .	99
5.2.1	Binary-backed data structures . . . . .	99
5.2.2	Unit testing with hardware/software equivalence . . . . .	100
5.3	Software . . . . .	102



5.3.1	Kaldi modifications . . . . .	102
5.3.2	Model suite preparation . . . . .	103
5.4	Summary . . . . .	104
<b>6</b>	<b>IC Implementation</b>	<b>105</b>
6.1	Overview . . . . .	105
6.2	Synthesis flow information . . . . .	105
6.2.1	Clock gating: explicit and implicit . . . . .	109
6.2.2	Floorplan . . . . .	110
6.2.3	Timing considerations . . . . .	111
6.3	Test setup . . . . .	113
6.3.1	PCB . . . . .	114
6.3.2	FPGA . . . . .	115
6.3.3	Software . . . . .	116
6.4	Results . . . . .	117
6.4.1	Voltage/frequency scaling . . . . .	118
6.4.2	ASR power consumption . . . . .	119
6.4.3	ASR subsystem efficiency . . . . .	121
6.4.4	Explicit clock gating . . . . .	123
6.4.5	VAD . . . . .	124
6.5	Summary . . . . .	126
<b>7</b>	<b>Conclusion</b>	<b>127</b>
7.1	Summary . . . . .	127
7.1.1	Contributions . . . . .	127
7.1.2	Performance and scalability . . . . .	129
7.2	Future work . . . . .	129
7.2.1	Design and application refinement . . . . .	130
7.2.2	System integration . . . . .	132
7.2.3	Leveraging technological change . . . . .	133



# List of Figures

1-1	Power gated speech recognizer concept. The scope of the work includes the software and circuit components within the dashed boxes. . . . .	25
1-2	Progression of research program through three IC tapeouts. . . . .	27
1-3	HMM formulation of ASR. Separate frameworks are used to evaluate the transition probabilities $p(x_{t+1} x_t)$ (bottom) and observation probabilities $p(y_t x_t)$ (right). . . . .	28
2-1	Signal processing steps used to extract mel-frequency cepstral coefficients (MFCCs). . . . .	33
2-2	Heat map of MFCCs within two tied states, taken from 894 WSJ utterances and projected to 2 dimensions using principal components analysis (PCA). . . . .	34
2-3	Minimum distortion quantizer bins (black vertical lines) follow the empirical distribution of the parameters the quantizer is used to compress (in this example, GMM means for the first dimension of the feature space). . . . .	36
2-4	GMM bandwidth and accuracy with independently varied quantization bit depths for (means, variances). Nominal ASR accuracy can be achieved with 5–7 bits. . . . .	37
2-5	Bandwidth/accuracy tradeoff for GMM acoustic model. Each color represents a model size (number of tied states and total number of Gaussian components), as shown in the legend. The stars plotted in each color represent different combinations of mean and variance quantization depth. . . . .	38
2-6	Bandwidth/accuracy tradeoff for SGMM acoustic model. Each color represents a model size (number of tied states and total number of substates), as shown in the legend. The stars plotted in each color represent different quantization depths applied to all parameters. . . . .	39
2-7	Schematic of NN evaluation. . . . .	40
2-8	Bandwidth (left) and accuracy (right) of 512-node, 6-layer NN acoustic models trained with different fractions of nonzero (NZ) weights. . . . .	42

2-9	Bandwidth/accuracy tradeoffs across many NN acoustic models. The dimensions and weight matrix sparsity of each model are shown in the legend. . . . .	42
2-10	Bandwidth/accuracy comparison of acoustic model frameworks, using the best results from each framework. . . . .	43
2-11	Block diagram of SIMD NN evaluator, relying on a shared sequencer and 32 execution units. . . . .	44
2-12	Execution units and local memory are grouped into chunks that can be reconfigured for different network sizes. . . . .	45
2-13	NN sequencer architecture supporting sparse and quantized weight matrices. . . . .	47
2-14	Compressed neural network data structure. Each layer is stored in either sparse or dense format (whichever uses fewer bytes). . . . .	48
2-15	Variable-width run length encoding is used to represent the column indices of nonzeros within each row of a sparse weight matrix. . . . .	48
2-16	Execution unit block diagram (pipelining, addresses, and control signals not shown). . . . .	49
2-17	The sigmoid function and simple piecewise polynomial fits. . . . .	50
2-18	Approximation error of piecewise polynomial fits to sigmoid function, comparing Taylor and Chebyshev series. . . . .	51
2-19	Block diagram of piecewise polynomial sigmoid evaluator. . . . .	51
3-1	Baseline search architecture with two-frame state list memory. . . . .	54
3-2	WFST preprocessing with multi-word output symbols. Left: Adding $\epsilon$ arcs to bypass arbitrary length chains of only $\epsilon$ arcs [16]. Right: Compound word labels are created for arcs that bypass multiple non- $\epsilon$ output labels. . . . .	55
3-3	Improved search architecture including word lattice and merged state lists. . . . .	56
3-4	Data storage format for SRAM-based hash table with open addressing. Linear probing is used to resolve collisions. . . . .	57
3-5	State transition diagrams for SRAM-based hash table. (Resizing is performed as needed in between commands, using the same basic operations.) . . . . .	58
3-6	Data fields for state list entries and in-flight hypotheses. The architecture is designed to preserve memory locality. . . . .	60
3-7	Guide to common state list operations, building on functionality provided by the hash table. . . . .	61
3-8	WFST data structure: a key/value store where keys are state IDs, and values contain all outgoing arcs from each state. . . . .	62

3-9	WFST fetch architecture with caching and model compression. . . . .	62
3-10	WFST cache architecture, including eviction queue and hash table so that variable-length objects can be stored in a circular buffer. . . . .	63
3-11	WFST cache performance with varying beam width. A wider beam increases the working set size. . . . .	64
3-12	WFST cache performance can be optimized by setting a maximum cacheable state length which depends on the task and beam width. . . . .	65
3-13	Example WFST, along with the OpenFST representation (totaling 832 bits) for states 1 and 4.	66
3-14	Compressed format (272 bits) for the states in Figure 3-13. An 8-bit header configures the storage format for each state. . . . .	66
3-15	The changing level of ambiguity in the speech signal causes search workload to vary over time. . . . .	67
3-16	A histogram of relative log-likelihoods (left) is used to approximate the CDF (right) and predict an appropriate beam width for the desired state cutoff. . . . .	69
3-17	State lattice generated during decoding (left); state snapshot concept (right). . . . .	70
3-18	Word lattice example (left); word lattice snapshot concept (right). . . . .	70
3-19	Data structures used in word lattice. . . . .	71
3-20	Write bandwidth comparison of state lattice and word lattice approaches. Time-averaged bandwidth was measured on one of the harder utterances in the nutrition test set. . . . .	75
3-21	WER vs. beam width for each ASR task, evaluated in Kaldi. . . . .	76
3-22	Runtime scales with beam width and varies between utterances. . . . .	77
3-23	WER (left) and RTF (right) for search using varying soft and hard cutoffs, compared to Kaldi. (Hardware recognizer is run at 80 MHz.) . . . . .	78
4-1	Energy-based VAD operation including waveform (top), estimated frame-level energy and SNR (middle), and energy difference scores (bottom). . . . .	81
4-2	Harmonicity VAD operation including waveform (top) and harmonicity scores (bottom). . . . .	83
4-3	Modulation frequency (MF) feature extraction. The use of both short-term and long-term FFTs allows MF features to capture temporal patterns that are unique to speech. . . . .	83
4-4	Example of mean MF matrix for speech and non-speech (top), and 4 Hz MF spectra (bottom) with speech segments indicated by dashed lines, at 0 dB SNR. . . . .	84
4-5	Block diagram of VAD module that can selectively enable EB, HM, and MF algorithms. . . . .	85

4-6	The VAD provides a 65-tap antialiasing filter that evaluates 8 taps per cycle. . . . .	86
4-7	Architecture of energy-based VAD module. This module evaluates the difference of energy between two frames and weights it with an estimate of SNR. . . . .	87
4-8	Architecture of harmonicity VAD module. This module computes a normalized autocorrelation $\frac{r_{xx}}{r_{ww}}$ to obtain the amplitude ratio of periodic and non-periodic components, expressed in dB as harmonicity. . . . .	88
4-9	Architecture of MF VAD module. This module performs modulation frequency feature extraction and then classification. On-chip memory is used to store a small NN model for classification. . . . .	89
4-10	Frame-level ROC of each VAD on Aurora2 task at 10 dB average SNR. . . . .	90
4-11	Estimated system power, derived from ROC of Figure 4-10, with 1 mW downstream power and 5% speech duty cycle. VAD power figures are educated guesses. . . . .	90
4-12	EER vs. average SNR on Aurora2 task. . . . .	91
4-13	EER vs. average SNR on Forklift/Switchboard task. . . . .	91
5-1	Block diagram of the complete IC design, including top-level interfaces and internal ASR/VAD interfaces. . . . .	94
5-2	Front-end block diagram. The frontend buffers raw features and applies output transformations on-the-fly to save memory. . . . .	95
5-3	Block diagram of ASR controller. The decode controller module coordinates the operation of front-end, acoustic model, and search; interface responsibilities such as audio/feature logging are delegated to other modules. . . . .	97
5-4	State transition rules for supervisory logic controlling isolation, reset, clock, and power state of ASR. . . . .	97
5-5	VAD/ASR interfaces allow the ASR to retrieve earlier audio samples after it is woken up by the VAD. . . . .	98
5-6	Hierarchical binary-backed data structure: base class (left) and example implementations (right). . . . .	99
5-7	Tools for generating simulation models with hardware/software equivalence. . . . .	101
5-8	The <code>ModelSuite</code> data structure allows multiple Kaldi recognizers to be compiled into a single binary blob. This example shows a configuration supporting weather information (Jupiter) and business news (WSJ) tasks. . . . .	103

6-1	Top level pinout partitioned into ASR, VAD, and supervisory regions. . . . .	106
6-2	Physical design flowchart illustrating the dependencies between different steps of IC implementation. . . . .	108
6-3	Verification chains: ideal (left) and real (right). . . . .	108
6-4	Clock gating hierarchy: VAD clock domain (top), ASR clock domain (bottom). . . . .	109
6-5	Interaction between voltage areas is designed so that voltage differences are consistent with level shifter orientation. . . . .	110
6-6	The ASR/VAD chip floorplan groups logic into contiguous regions to simplify optimization by the placement and routing tool. . . . .	111
6-7	ASR/VAD chip layout. . . . .	114
6-8	A conventional digital IC test setup requires the coordination of several pieces of lab equipment. . . . .	115
6-9	Block diagram (top) and photo (bottom) of test PCB stack integrating power supplies and functional test interfaces. . . . .	116
6-10	Onboard programmable power supply regulator with current sense resistor inside the feedback loop. . . . .	117
6-11	FPGA-based adapter for generic host, memory, and control interfaces. . . . .	117
6-12	Software modules used for ASR/VAD testing and demonstration. . . . .	118
6-13	ASR/VAD chip photos: bare die (left) and 88-pin QFN package (right). . . . .	118
6-14	Maximum working frequency for ASR as a function of logic supply voltage. . . . .	119
6-15	Waveforms of selected power supply currents during ASR decoding. . . . .	120
6-16	Acoustic model efficiency varies by a factor of 3 due to voltage/frequency scaling. . . . .	122
6-17	Regression to estimate search efficiency from decoding results at different beam widths. . . . .	123
6-18	Search efficiency scales quadratically with the memory supply voltage. . . . .	124
6-19	Power consumption with and without explicit clock gating. . . . .	125
7-1	Vision for single-chip speech processing solution. . . . .	130





# List of Tables

1.1	Datasets used for ASR experiments. . . . .	29
2.1	Scalability owing to design of different NN acoustic models. Runtimes are quoted in clock cycles per second of utterance time. . . . .	46
3.1	Compression ratios for <i>HCLG</i> WFST models. . . . .	66
3.2	Algorithms used in word lattice generation. . . . .	72
3.3	Search workload for each ASR task. Workload is specified in hypotheses, and runtime in clock cycles, per second of utterance time. . . . .	77
5.1	Summary of host interface commands. . . . .	96
5.2	Summary of host interface responses. . . . .	96
6.1	Summary of ASR/VAD chip area. . . . .	111
6.2	Static timing verification scenarios covering the space of expected supply voltage combinations. “Nom” indicates the nominal $V_{DD}$ of the standard cell library and “Red” indicates a reduced $V_{DD}$ . . . . .	112
6.3	Timing optimization scenarios cover the extremes of possible multivoltage operating points. “Nom” refers to the nominal $V_{DD}$ of the standard cell library. . . . .	113
6.4	ASR/VAD chip power supplies. . . . .	119
6.5	Power consumption averaged over each ASR task. . . . .	121
6.6	Estimation of acoustic model efficiency over a single utterance. . . . .	122
6.7	Initial measurements of VAD power consumption. . . . .	125
7.1	ASR performance summary. . . . .	129
7.2	VAD performance summary. . . . .	129



# Glossary

ADC	analog-to-digital converter
API	application programming interface
ASIC	application-specific integrated circuit
ASR	automatic speech recognizer
BC	best-case
CDF	cumulative distribution function
CMOS	complementary metal-oxide semiconductor
CMVN	cepstral mean and variance normalization
CNN	convolutional neural network
DAC	digital-to-analog converter
DDR	double data rate
DNN	deep neural network
DRAM	dynamic random-access memory
EB	energy-based
EDA	electronic design automation
EER	equal error rate
EU	execution unit
FFT	fast Fourier transform
FIFO	first-in/first-out
FIR	finite impulse response
fMLLR	feature-space maximum likelihood regression
FPGA	field-programmable gate array
fps	frames per second
GDS	Graphic Database System
GMM	Gaussian mixture model
GPIB	General Purpose Interface Bus
HM	harmonicity
HMM	hidden Markov model
HTK	HMM Toolkit [91]

I <sup>2</sup> S	Inter-IC Sound
IC	integrated circuit
IEEE	Institute of Electrical and Electronics Engineers
IFFT	inverse FFT
IP	intellectual property
KWS	keyword spotting
LDA	linear discriminant analysis
LEF	Library Exchange Format
LM	language model
LRT	likelihood ratio test
LSTM	long short-term memory
LTFT	long-term Fourier transform
LUT	lookup table
MAC	multiply/accumulate
MAP	maximum a posteriori
MEMS	micro electromechanical system
MF	modulation frequency
MFCC	mel-frequency cepstral coefficient
MIG	Memory Interface Generator
MLC	multilevel cell
MMI	maximum mutual information
NN	neural network
PC	personal computer
PCA	principal components analysis
PCB	printed circuit board
QFN	quad flat no-leads
ReLU	rectified linear
RLE	run-length encoding
RNN	recurrent neural network
ROC	receiver operating characteristic
ROM	read-only memory
RTF	real-time factor
RTL	register transfer level
SGMM	subspace GMM
SIMD	single instruction/multiple data
SLC	single-level cell
SNR	signal-to-noise ratio
SPI	Serial Peripheral Interface
SPICE	Simulation Program with Integrated Circuit Emphasis

---

SRAM	static random-access memory
STFT	short-term Fourier transform
STL	Standard Template Library
SV	SystemVerilog
SVD	singular value decomposition
SVM	support vector machine
TC	typical-case
TDNN	time-delay neural network
TSMC	Taiwan Semiconductor Manufacturing Company
UBM	universal background model
UPF	Unified Power Format
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VAD	voice activity detector
WC	worst-case
WER	word error rate
WFST	weighted finite-state transducer
WL	word lattice
WSJ	Wall Street Journal [60]
XML	eXtensible Markup Language



# Chapter 1

## Introduction

### 1.1 Motivation

As of this writing, speech interfaces are commonly used for search and personal assistants on smartphones and PCs. As these interfaces evolve to offer a more natural user experience, people will become more comfortable interacting with machines by talking with them. The applications of speech interfaces will diversify and include a wider range of devices, such as wearables, appliances, and robots.

A hands-free speech interface consists of a wake-up mechanism, speech-to-text (recognizer), text-based dialogue processor, and text-to-speech (synthesizer). Dialogue processing could include natural language understanding and generation, or translation from a source language to a target language. The components examined by this thesis will frequently be referred to by their acronyms: voice activity detector (VAD) and automatic speech recognizer (ASR).

A VAD classifies an audio stream as speech or non-speech, labeling the start and end time of each speech segment. This is harder than it sounds: in real-world scenarios there can be loud or non-stationary background noise that is difficult to discriminate from speech with any simple algorithm. Some VAD algorithms use expertly tailored criteria based on the energy or spectral content of the signal, and others rely on supervised machine learning to classify feature vectors derived from the signal. It is especially important to optimize the power consumption of the VAD since it must be running at all times.

An ASR transcribes an audio stream into a sequence of words. ASR is usually expressed as a hidden Markov model (HMM) inference problem, where the hidden variables are states within a Markov process modeling speech production, and the observed variables are acoustic features. Transcribing (decoding) an utterance is equivalent to maximum a posteriori (MAP) inference over the HMM, for which we use the Viterbi algorithm. Statistical models are used to evaluate the transition probabilities between (discrete)

states and the observation likelihoods over (continuous) features. Over the years, people have developed and refined the ability to train the necessary models from large corpora of transcribed audio files. This training task can be performed offline using a compute cluster. The decoding task is run in real-time for each client device.

Real-time ASR decoding is computationally demanding. The exact workload depends on the task, but most devices supporting speech recognition either have high compute power (like a multicore x86 processor) or an Internet connection linking them to cloud servers. The computational requirements often increase as researchers identify improved modeling techniques. In order to bring speech interfaces to devices that are smaller and cheaper than PCs, or those that lack Internet connections, we need specialized hardware.

As others have shown [47], circuit designs tailored to specific tasks eliminate the overhead of general-purpose architectures such as x86 or ARM, reducing the energy cost of those tasks by 100x or more. There are also opportunities to modify algorithms and runtime parameters for favorable power/performance trade-offs. However, there is a big difference in convenience and energy cost between on-chip (small) and off-chip (large) memory that must be taken into account. Software designers typically work with a single flat address space and multi-GB/s bandwidth, relying on caching mechanisms provided by the processor. Porting algorithms to hardware without changing these assumptions does result in some efficiency gains, but shifts the bottleneck from processor to memory. Accommodating memory limitations is a recurring theme in low-power digital design, and ASR is no exception.

Another benefit of adding ASR capabilities to client devices is reduced latency. In dialogue systems, the application needs to react quickly to what the user is saying (as well as emotional or visual cues). Eliminating the roundtrip network latency from the device to a cloud-based recognizer can provide an improvement to the user experience, regardless of power consumption.<sup>1</sup>

## 1.2 Scope

We are developing a local speech recognition capability for embedded systems, as shown in Figure 1-1. A system designer needs to prioritize and optimize the power consumption of all components jointly. The scope of this thesis is the signal processing IC, designed with awareness of the external memory, and the infrastructure needed to port existing software recognizers to this platform. This memory and its I/O signaling can easily consume more power than the speech recognizer itself. Since speech interfaces are used infrequently (low duty cycle), the memory must be non-volatile. Flash and other non-volatile memories

---

<sup>1</sup>Ramon Prieto (Jibo), personal communication, Jan. 17, 2016.



have a higher energy-per-bit than DRAM, so there is a strong incentive to minimize memory bandwidth. The most affordable and widely used non-volatile memory is NAND flash. According to [27], normalized read energy is typically in the range of 50 pJ/bit for the single-level cell (SLC) variant of NAND flash and 100 pJ/bit (or 8 mW at 10 MB/s) for the more common multilevel cell (MLC) variant. It is worth improving the energy efficiency of the ASR processor until it reaches parity with the memory.

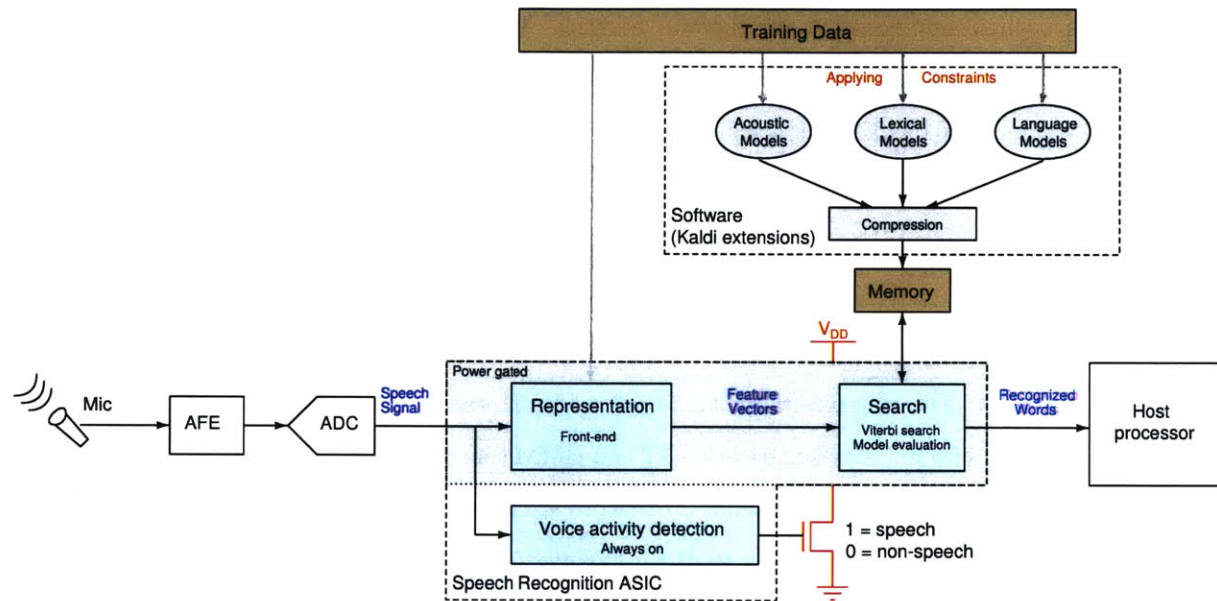


Figure 1-1: Power gated speech recognizer concept. The scope of the work includes the software and circuit components within the dashed boxes.

This thesis explores the idea of using the VAD decision output to power gate the ASR. Otherwise, the ASR would require an explicit wake-up command from the user (i.e., a “push to talk” button), or it would operate continuously and waste power during non-speech input. We will show that using a more sophisticated VAD algorithm (and hence a larger, higher-power VAD circuit) results in the lowest time-averaged system power by reducing false alarms that wake up the more power-hungry ASR and memory.

The rest of this thesis demonstrates digital IC implementations of a VAD and an ASR, and explains the design decisions that make the implementation feasible and minimize system power consumption. Many degrees of freedom and their impacts are considered in pursuit of more accurate, programmable, and scalable ASR capabilities:

- **Accuracy** is measured as the word error rate (WER) of the transcription result for a particular speech recognition task. This depends on both training and decoding techniques. We demonstrate a controlled loss of accuracy relative to state-of-the-art software speech recognizers across a variety of tasks.

- **Programmability** is the ability to execute different speech recognition tasks (e.g., different vocabularies, topics, speakers, and languages) by using different models and front-end representations. We allow a variety of models trained using the open-source Kaldi tools [64] to be converted into formats supported by the chip.
- **Scalability** is the ability to adjust the chip's speed and power consumption according to the needs of the speech recognition task (or a power budget). We demonstrate scalability at several levels: selecting models of appropriate complexity, adjusting runtime decoding parameters, and adjusting circuit voltages and clock frequencies.

This research resulted in the fabrication of three IC chips, summarized in Figure 1-2. The process has been cumulative, with the third chip integrating and refining (or replacing) structures developed in the other two:

- The first chip provides a proof of concept for low-power ASR and serves as a baseline for the remainder of our work. The chip was fabricated using a 65 nm process; it measures  $2.5 \times 2.5$  mm, and is programmable with industry-standard WFST and GMM speech models. This chip performs a 5,000 word recognition task in real-time with 13.0% word error rate, 6.0 mW core power consumption, and a search efficiency of approximately 16 nJ per hypothesis. This work is described in [66] and [67].
- The second chip was used to evaluate VAD algorithms within our system power model, considering both the power and accuracy of the VAD. This chip is operational, but we are withholding further measurement results due to a functional bug in one of the three algorithms.
- The third chip implements many enhancements to the baseline ASR design and allows the ASR to be power gated by decisions from an onboard VAD.

To simplify the discussion, we describe only the third chip in this thesis.

### 1.3 ASR formulation

A short summary of the ASR problem will help you understand the acoustic modeling and search techniques described in later chapters.

The ASR system has a front-end which transforms audio (e.g., 16-bit samples at 8 kHz or 16 kHz) into a lower-rate, higher-dimensional feature representation. We typically use 13-dimensional mel-frequency cepstral coefficient (MFCC) features [20] based on 25 ms frames of audio, with a 10 ms interval between frames. The MFCC features can be augmented with log-energy, time derivatives (e.g., deltas or double-deltas), and/or temporal context (features from previous frames) to improve performance. A typical utter-

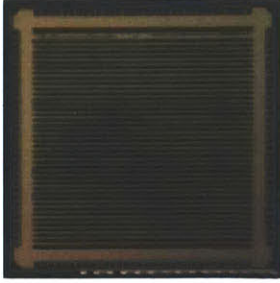
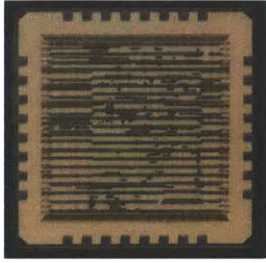
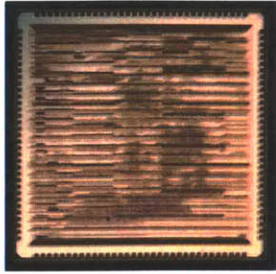
	ASR v1	VAD	ASR v2 + VAD
			
Dimensions	2.5 × 2.5 mm	1.84 × 1.84 mm	3.63 × 3.63 mm
Logic (kgates)	340	472	2088
Memory (kb)	2400	660	5840
Clock freq.	20–110 MHz	0.6–2.0 MHz	1.5–85 MHz
Description	<ul style="list-style-type: none"> <li>• WFST/GMM models</li> <li>• No training infrastructure; downloaded HTK models</li> <li>• Tested on WSJ eval92-5k: 13.0% WER, 6.0 mW</li> </ul>	<ul style="list-style-type: none"> <li>• Choice of 3 algorithms for power/accuracy tradeoff</li> <li>• SPI and I<sup>2</sup>S interfaces</li> <li>• 3–18 μW at supply voltages down to 0.49 V</li> </ul>	<ul style="list-style-type: none"> <li>• WFST/DNN models</li> <li>• Models can be generated from Kaldi recognizer</li> <li>• Multiple power domains, off-chip power gating</li> </ul>

Figure 1-2: Progression of research program through three IC tapeouts.

ance (e.g., sentence) lasts 1–10 seconds, or 100–1,000 frames. These feature vectors are handed off to the HMM.

The HMM is a graphical model expressing a relationship between hidden variables  $x_t$  and observed variables  $y_t$ . The regular structure of connections in this graph (Markov property) is exploited to allow approximate inference in linear time (Viterbi algorithm), making statistical speech recognition tractable.

Figure 1-3 is a high-level illustration of a speech HMM. The HMM uses knowledge of the underlying speech production process to model hidden state sequences  $x_t$ . To perform ASR we supply MFCC feature vectors for  $y_t$  and use the HMM to estimate the most likely values of  $x_t$ .

Using the HMM framework requires modeling the dependencies between variables, specifically the transition model  $p(x_{t+1}|x_t)$  and the emission model  $p(y_t|x_t)$ . The transition model incorporates information about the language, including its vocabulary and grammatical constraints; the hidden states  $x_t$  are discrete variables. The emission model describes how observations  $y_t$  vary depending on the unobserved state  $x_t$ ; in speech recognition, this is called the acoustic model. These observations reflect the speech signal generated by the speaker’s vocal tract, and the influence of the channel (the microphone and the acoustic environment).

Properly accounting for multiple levels of knowledge (e.g., context, pronunciation, and grammar) in the transition model compounded the complexity of early HMM decoders. The weighted finite-state transducer (WFST) is a state machine representation that allows each component to be described separately [54, 55] and

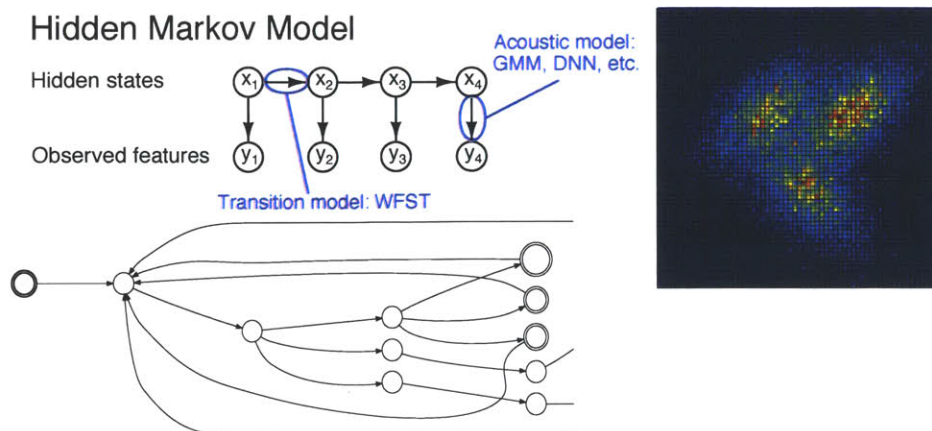


Figure 1-3: HMM formulation of ASR. Separate frameworks are used to evaluate the transition probabilities  $p(x_{t+1}|x_t)$  (bottom) and observation probabilities  $p(y_t|x_t)$  (right).

then composed into a single WFST encompassing the entire search space [1]. (Section 3.2.3 specifies these components more precisely.) The WFST can be viewed as a graph that is easily queried for the (sparse) transition probabilities  $p(x_{t+1}|x_t)$ , which convey the constraints of the language model and lexicon. All possible transitions between states are expressed as weighted, labeled arcs. A WFST-based decoder can complete tasks of varying complexity in different domains and languages by substituting the appropriate model parameters.

The HMM factorizes the joint probability of all observed and hidden variables as:

$$p(x, y) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1}|x_t)p(y_t|x_t)$$

In this formulation,  $x_t$  is a discrete variable referring to a state in a weighted finite-state transducer (WFST) [55];  $y_t$  is a real-valued vector. The hidden state sequence  $\mathbf{X} = \{x_0, x_1, \dots, x_{T-1}\}$  specifies a word sequence  $\mathbf{W}$  via the output labels of WFST arcs traversed by  $\mathbf{X}$ . Finding the most likely word sequence given a set of observations  $\mathbf{Y}$  is MAP inference:  $\mathbf{W}^* = \arg \max_{\mathbf{W}} p(\mathbf{W}|\mathbf{Y})$ . We perform MAP inference over states and then convert the result to words.

To estimate the state sequence from features, we use the Viterbi search algorithm. The algorithm maintains a list of hypotheses, or active states, at each time step. This search procedure is described in [39] and many other sources. An important step in Viterbi search is the forward update, which approximates the

likelihood of all reachable states at the next time step:

$$p(x_{t+1}) = \sum_{x_t} p(x_t) p(x_{t+1}|x_t) p(y_{t+1}|x_{t+1})$$

$$\approx \max_{x_t} p(x_t) p(x_{t+1}|x_t) p(y_{t+1}|x_{t+1})$$

In our implementations, the existing state likelihood  $p(x_t)$  is retrieved from an “active state list” in local memory. The transition probability  $p(x_{t+1}|x_t)$  is the weight of the WFST arc[s] leading from state  $x_t$  to state  $x_{t+1}$ . The emission probability  $p(y_{t+1}|x_{t+1})$  is an acoustic likelihood specified by the acoustic model.

There are many possible values for  $x_t$  because WFSTs for ASR typically have millions of states. To speed up decoding, ASR systems use a “beam search” variant of the Viterbi algorithm that discards unlikely hypotheses.<sup>2</sup> There are several possible criteria for saving hypotheses; for example, you could discard every hypothesis whose log-likelihood falls below a certain cutoff. In this scheme, the difference between the cutoff and the maximum log-likelihood is called the beam width. Decreasing (narrowing) the beam width saves computation time and introduces search errors.

The acoustic model describes the expected probability distribution of feature vectors, given a particular state  $x_t$ . Rather than using a separate distribution for each state in the WFST, the states are grouped into clusters (typically 1,000 to 10,000) called “tied states” or “senones.” This clustering is usually performed in a top-down fashion [92]. All states in each cluster share one probability density. Evaluating transition probabilities (from the WFST) and acoustic model likelihoods requires fetching model parameters from a large external memory.

Rather than designing one universal ASR model, we follow the standard practice of training different recognizers for different tasks. Some of the tasks we studied are listed in Table 1.1.

Name	Size (hr)	Vocabulary (approx.)	Channel	Speech type	LM Perplexity
TIDIGITS [45]	4.1	11	Telephone	Read	10.8
Jupiter [96]	73	2k	Telephone	Spontaneous	8.58
Nutrition [43]	2.7	7k	Wideband	Read	23.7
WSJ [60] (eval92-5k)	81	5k	Wideband	Read	65.6
WSJ [60] (dev93)	81	20k+	Wideband	Read	166

Table 1.1: Datasets used for ASR experiments.

Each task is based on a corpus of training and test data, and has its own (possibly domain-specific) vocabulary and acoustic properties. All tasks rely on N-gram language models, or LMs [26], ranging from

<sup>2</sup>There are usually diminishing returns to tracking more than a few thousand hypotheses, but this depends on the task.

unigram (TIDIGITS) to 4-gram (Jupiter). To convey the search complexity of each task, we specify the LM perplexity [37]; this can be interpreted as a branching factor, with higher perplexity indicating fewer constraints for search. (The perplexity of the TIDIGITS model is close to 11, since at any point in the search there are 11 possible words which are almost equally likely.<sup>3</sup>) Having a range of tasks lets us evaluate the scalability of the ASR implementation.

## 1.4 Prior art

Since the late 1980s, there have been sporadic efforts to exploit special-purpose hardware for speech applications. Application-specific integrated circuits (ASICs) for HMM-based speech recognition were reported as early as 1991 [80], followed by power-conscious implementations intended for portable devices [12]. The “In Silico Vox” project [48] created sophisticated hardware ports of Sphinx-3, a well-established software framework for ASR. Between 2006 and 2010 this project developed several hardware decoders on both single-FPGA and multi-FPGA platforms [49, 46, 11], achieving 10x faster than real-time performance on a Wall Street Journal (WSJ) task. These researchers had a broad focus on making hardware ASR work and scaling to larger tasks and lower power.

More specialized research efforts have examined changes in ASR algorithms and their applicability to hardware. One area of evolution has been in the use of WFSTs, which are a key component of the modern ASR framework. WFSTs are now commonplace in software speech recognizers, including commercial systems [72]. Perhaps due to long development cycles, most of the hardware implementations realized so far are based on non-WFST software decoders. Digital circuits using WFSTs were presented in 2008 [16], contributing an important observation that pre-processing the WFST could eliminate the need to consider unlabeled arcs recursively during each frame. Taking advantage of external DRAM memory with 3.2 GB/s bandwidth, [90] illustrated how to split the decoder’s active state list between internal and external memory to achieve better accuracy with limited internal SRAM.

Another focus area in ASR research has been acoustic modeling. All hardware ASR implementations we know of have used Gaussian mixture models (GMMs) with varying levels of complexity (number of tied states and number of Gaussian components per distribution). To speed up computation, designers have performed frame skipping and Gaussian selection [61] and used fixed-point parameters [9]. Meanwhile, deep neural networks (DNNs) and other neural network architectures have become popular for ASR due to their improved accuracy [34]. The circuits community has studied DNNs and developed efficient implementa-

---

<sup>3</sup>TIDIGITS uses the ten digits (zero through nine), but zero can be spoken as “oh;” this is treated as an 11th word.

tions, primarily for computer vision [15, 59].

Other recent work performed architectural exploration: choosing how to divide hardware resources between acoustic modeling and search, and two-pass decoding with the first pass performed by hardware [7]. Code generation tools can help rapidly evaluate different accelerator kernels specified in a high-level language [78]. Acoustic models generally see large efficiency gains from acceleration, and front-ends generally do not, but there is no consensus on the proper hardware/software partitioning for search—this seems to depend on the application.

The wide range of algorithms available for VAD also provides opportunities to apply special-purpose architectures. Power can be minimized through a variety of techniques, whether minimum-energy operation in deeply scaled CMOS [71], or mixed-signal design with adaptive feature granularity [5]. These efforts have brought VAD power consumption down to a few  $\mu\text{W}$ .

Contemporary efforts in ASR have continued expanding decoder capabilities for general-purpose transcription applications with a 60,000 word (or larger) vocabulary. For example, [41] proposed an architecture for achieving much higher throughput (127x faster than real-time) using an ASIC. Another effort [31, 30, 29] bridged the gap between high-performance and low-power applications, applying a series of optimizations to a Japanese-language system called Julius in order to obtain just 54 mW of power consumption and 82 MB/s of memory bandwidth during real-time decoding. Also, independently of this work, [3] discussed techniques of WFST pruning and search that are relevant to low-power ASR applications.

Throughout this thesis we will cite additional prior work relating to each aspect of ASR/VAD design.

## 1.5 Contributions

This thesis builds on previous efforts in two major areas: (1) circuit implementations supporting modern algorithms and frameworks; and (2) pushing down power consumption from a system-level perspective.

**Adopting modern algorithms:** While DNNs have been used in ASR for several years, this work may be the first realization of a hardware speech recognizer using DNNs. This revision is justified by our comparison of DNNs with two other types of acoustic models: GMMs and subspace GMMs.

Our work in [67] was to our knowledge the first ASIC speech decoder to support WFSTs, and the first to handle the entire speech recognition chain from audio to text. In this thesis, continued studies of HMM search and its demands on the WFST data structure lead to a revised architecture with WFST compression and a new caching strategy.

While WFSTs and DNNs are fairly generic, there are many variants to how they are constructed and used. We made our decoder compatible with Kaldi [64], a popular open-source library for speech recognition. This simplifies the process of deploying a recognizer on our hardware platform. We designed auxiliary hardware and software components to make this possible.

Algorithms used in IC implementations of VAD have also lagged behind software implementations. We demonstrate hardware implementations of three VAD algorithms adapted from recent literature.

**System-level power reductions:** We want to minimize the size and power of a speech-enabled system, as opposed to a single component (such as the core logic of an ASR chip). Modeling the power consumption of all relevant components led us to realize that external (off-chip) memory is a major bottleneck. Reducing the bandwidth expected of this memory is our top priority, dictating many of the design choices we made. The end result is that many ASR tasks can be run with less than 10 MB/s of bandwidth, comfortably within the limits of a single NAND flash die. This is significantly lower than the bandwidth targets of previous work. For comparison, [17] (which describes in-depth techniques for optimizing memory access) reports bandwidth of 219 MB/s. Our bandwidth reductions come from several sources that will be discussed in more detail below. We believe that the core area and power tradeoffs made to achieve these reductions are acceptable, especially since they are mitigated by CMOS scaling.

VAD is an important part of our system power reduction strategy. We provide a simple model for power consumption when VAD is used to power gate an ASR, and show how this guides the selection of VAD algorithms. Our VAD is designed to operate without any external memory, which is necessary to achieve  $\mu\text{W}$  power levels in the inactive state.

Wherever possible, we use low-power IC design techniques including clock gating, variation-aware timing constraints, and multiple power domains.

## 1.6 Guide to chapters

The remaining chapters describe the design of our ASR/VAD system in a holistic fashion, integrating the discussion of algorithms and architectures and quantifying the impact of each contribution on our goals of accuracy, programmability, and scalability. Chapters 2 and 3 focus on the largest components of the ASR: acoustic modeling and search. Chapter 4 considers VAD. Chapter 5 discusses software and hardware components that support these components and enable a system demonstration. Chapter 6 discusses the IC implementation process and testing results; Chapter 7 concludes.



## Chapter 2

# Acoustic modeling

The acoustic model has to evaluate the likelihood of input features  $\mathbf{y}_t$  with respect to a set of distributions  $p(\mathbf{y}|i)$ , where  $i$  is the index of a tied state. Many ASR systems use MFCC features [20], which are computed using the signal processing pipeline shown in Figure 2-1. The MFCC representation captures short-term spectral information with relatively few dimensions.

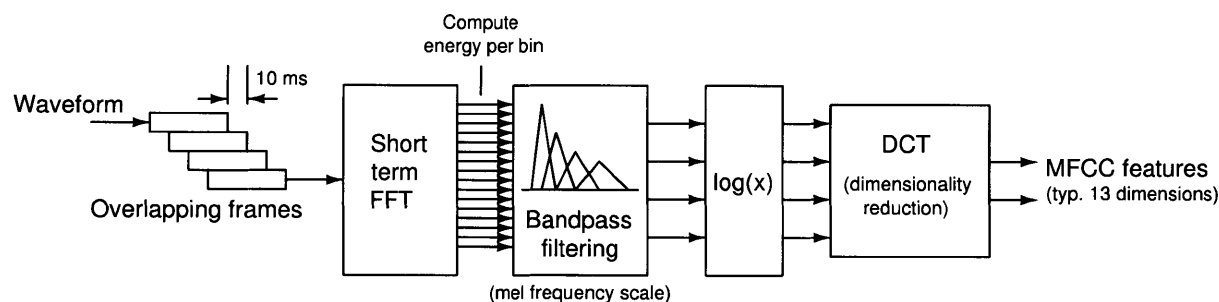


Figure 2-1: Signal processing steps used to extract mel-frequency cepstral coefficients (MFCCs).

Some examples of feature distributions (with the features reduced to 2-D for visualization) appear in Figure 2-2. The features are typically 10–40 dimensions, though they are often augmented with time derivatives, or with features from previous frames.

The accuracy of likelihoods computed by the acoustic model directly impacts the search workload and word accuracy of a recognizer. The general trend in ASR research has been to leverage growing compute resources and datasets to train increasingly elaborate models with more parameters. Implementers of low-power ASR systems cannot blindly follow this trend. Instead of searching for the most accurate acoustic model, we ask a slightly different question.

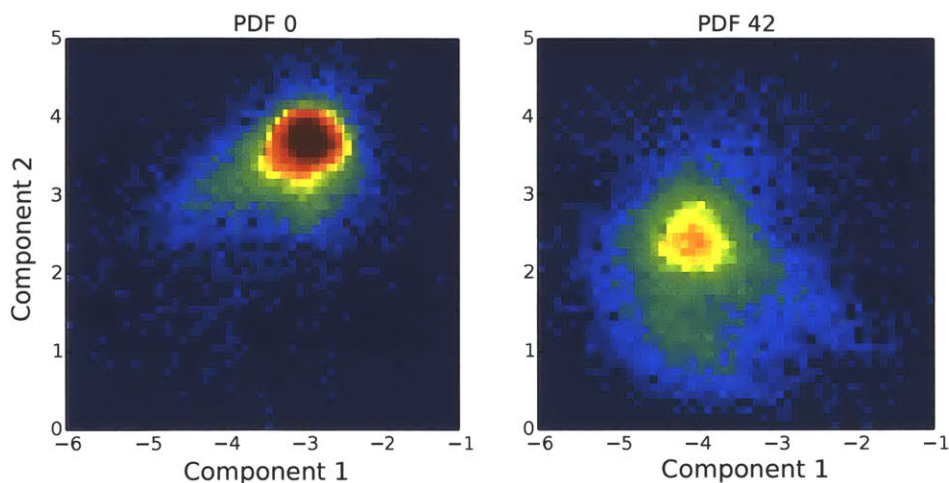


Figure 2-2: Heat map of MFCCs within two tied states, taken from 894 WSJ utterances and projected to 2 dimensions using principal components analysis (PCA).

## 2.1 What is the most accurate acoustic model with 1 MB of parameters?

We begin this discussion with a back-of-the-envelope calculation. Our hardware ASR architecture is optimized to reduce memory bandwidth, with a target of 10 MB/s. The frame rate is 100 fps, and with some degree of parallelism it should be possible to evaluate likelihoods for 10 frames concurrently; this means that the entire acoustic model is fetched about 10 times per second. To hit the 10 MB/s target, the model would have to fit within 1 MB or less.

This calculation shows that acoustic models for low-power ASR should be designed with the quantity and precision of parameters in mind. Our implementation does not have a hard limit for the acoustic model size, but memory bandwidth (and hence a large part of system power) is proportional to model size. Compressing models down to around 1 MB is a general guideline for keeping bandwidth in check.

There is a nontrivial choice of what modeling framework to use. Until around 2010, most ASR systems used Gaussian mixture models (GMMs) to specify each senone distribution. Most recent advances in acoustic models have relied on neural networks, whether feed-forward deep networks (DNN), recurrent networks (RNN), convolutional networks (CNN), long short-term memory cells (LSTM), or some combination of those [74]. Povey et al. also developed a subspace GMM (SGMM) framework that achieves improved accuracy over GMMs [63]. We characterized the tradeoff between word accuracy and memory bandwidth for GMM, SGMM, and DNN models in order to select a good framework for low-power, memory-limited

implementations.

Kaldi [64] is an open-source ASR framework that has been a godsend to ASR researchers and allows quick evaluations of different modeling techniques, once you have obtained the appropriate speech corpora. Kaldi provides C++ binaries and Bash scripts for feature extraction, acoustic modeling, decoding, and many other purposes.<sup>1</sup> We used Kaldi to train recognizers for several ASR tasks, summarized in Table 1.1. In order to model the accuracy degradation and bandwidth reduction of our hardware-oriented algorithm changes, we constructed a separate ASR decoder in C++ that could read and translate models created by Kaldi. The results reported in this section were generated by this custom decoder.

### 2.1.1 Gaussian mixture model (GMM)

The GMM models each distribution as follows:

$$P(\mathbf{y}_t|i) = \sum_{c=0}^{C_i-1} w_{ci} \mathcal{N}(\mathbf{y}; \mu_{ci}, \Sigma_{ci})$$

where  $\mathcal{N}(\mathbf{y}; \mu, \Sigma)$  is the multivariate normal distribution over  $\mathbf{y}$  with mean  $\mu$  and variance  $\Sigma$ .

GMMs for ASR decoding typically use diagonal covariance matrices, compensating for the loss of modeling power by using more mixture components. If we have  $I$  distributions, an average of  $C$  mixture components per distribution, and feature vectors with dimension  $D$ , the total number of parameters will be  $CI(1 + 2D)$ . Typical dimensions for a state-of-the-art GMM acoustic model might be  $I = 8000$ ,  $C = 20$ , and  $D = 40$ . With single-precision floating point values, the model would be 51 MB. Naive evaluation at 100 fps would require 5.1 GB/s of memory bandwidth.

One way to shrink a GMM is to use fixed-point mean and variance coefficients with reduced precision. With a nonlinear quantizer (e.g., Lloyd-Max minimum distortion quantizer [52]), acceptable accuracy can be obtained with as few as 5 bits for the mean and 3 bits for the variance [33]. The quantizer is computed from the empirical distribution of parameters, with quantization levels being packed more closely around common values as shown in Figure 2-3. Unless the feature space has been whitened, separate quantizers must be estimated for each feature dimension, but these are relatively small. This technique led to a 7.2x reduction in model size in [67]. Figure 2-4 shows the performance effect of different mean and variance quantization depths on a small GMM. More accurate quantizers decrease the word error rate (vertical axis), but in exchange, they increase the model size and hence the memory bandwidth (horizontal axis). In this case, there is no benefit to using more than 12 bits total for each mean/variance pair; those bits should be

<sup>1</sup>The documentation and source code for Kaldi are available at <http://www.kaldi-asr.org>.

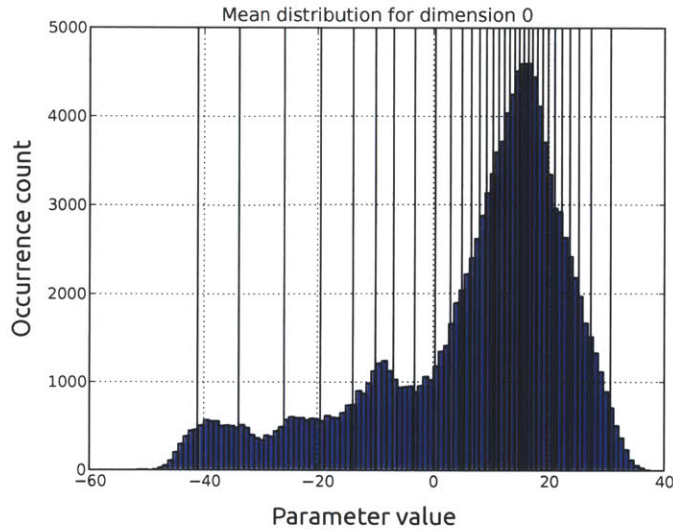


Figure 2-3: Minimum distortion quantizer bins (black vertical lines) follow the empirical distribution of the parameters the quantizer is used to compress (in this example, GMM means for the first dimension of the feature space).

evenly split (approximately) between the mean and variance.

Beam search considers a limited number of hypotheses (typically 1k to 10k) during each frame. Each hypothesis incorporates the likelihood from one senone distribution, and each group of hypotheses uses a subset of the distributions. Because the GMM doesn't share parameters between distributions, you can save memory bandwidth by evaluating each distribution on demand and saving (memoizing) the result for the remainder of the frame. This was the approach we used in [67], resulting in an additional 7.6x reduction in memory bandwidth (to 54.1 MB/s). This bandwidth reduction is mildly task-dependent: a more thorough search (e.g., wider beam) will require more likelihood calculations.

If you are willing to use more on-chip memory (and logic), you can evaluate the distributions for multiple frames in parallel [31]. It is complicated to predict which subset of distributions will be requested by the search, so it's easiest to simply evaluate the entire model. With enough parallel evaluations (i.e., more than 10), the bandwidth savings exceeds what you would get from on-demand evaluation within each frame sequentially.

With parallel evaluation, the on-chip memory required by a GMM is dominated by the storage of likelihood results. For the purpose of model comparisons, we assume all models are limited to 1 Mb of local storage. With a fixed memory size and numerical precision (32 bits), there is an inverse relationship between  $I$  (the number of distributions) and the number of frames that can be evaluated in parallel. Taking this into

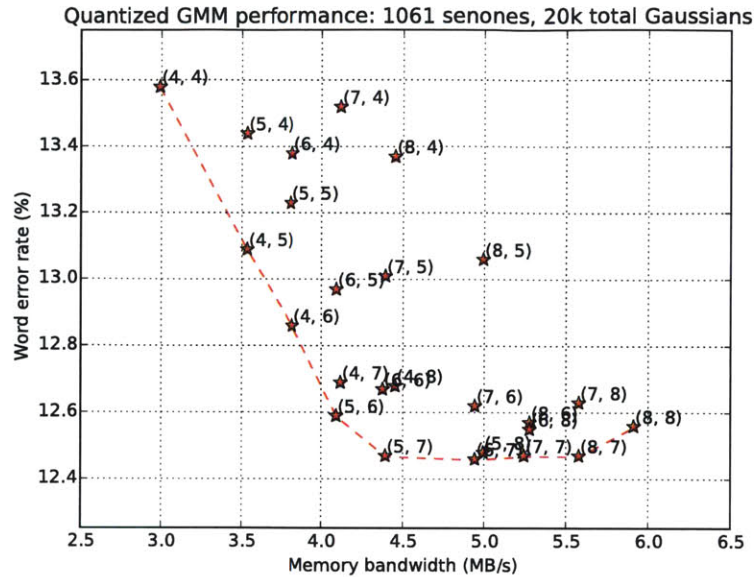


Figure 2-4: GMM bandwidth and accuracy with independently varied quantization bit depths for (means, variances). Nominal ASR accuracy can be achieved with 5–7 bits.

account, the performance of several hyperparameter combinations is shown in Figure 2-5. In each case, discriminative training with a maximum mutual information (MMI) criterion [88] has been used to improve accuracy; solid lines have been drawn through the points offering the best combinations of bandwidth and accuracy. Note that the bandwidth requirement of the smallest models (536 tied states) approaches 1 MB/s.

There is a steep tradeoff between memory bandwidth and recognition accuracy. This is generally true for all three modeling frameworks, although the details vary.

### 2.1.2 Subspace Gaussian mixture model (SGMM)

The SGMM is a collection of GMMs whose parameters vary in a low-rank vector space relative to a single GMM called the universal background model, or UBM [63]. Each distribution is modeled as follows:

$$P(\mathbf{y}_t|i) = \sum_{m=0}^{M_i-1} k_{im} \sum_{c=0}^{C-1} w_{cim} \mathcal{N}(\mathbf{y}; \mu_{cim}, \Sigma_c)$$

$$\mu_{cim} = \mathbf{M}_c \mathbf{v}_{im}$$

$$w_{cim} = \frac{\exp \mathbf{w}_c^T \mathbf{v}_{im}}{\sum_{c'=0}^{C-1} \exp \mathbf{w}_{c'}^T \mathbf{v}_{im}}$$

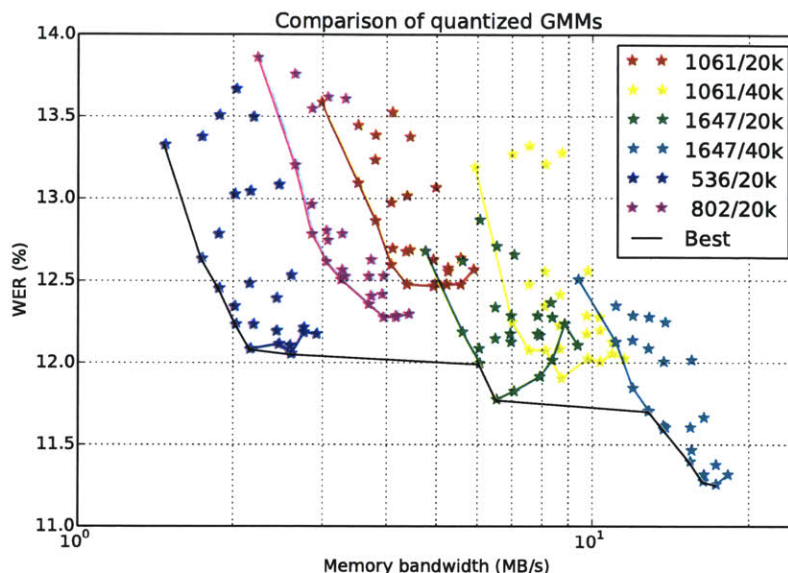


Figure 2-5: Bandwidth/accuracy tradeoff for GMM acoustic model. Each color represents a model size (number of tied states and total number of Gaussian components), as shown in the legend. The stars plotted in each color represent different combinations of mean and variance quantization depth.

where the vectors  $\mathbf{v}_{im}$  describe the location of parameters for distribution  $i$  in a low-rank subspace, and  $\mathbf{M}_c$  and  $\mathbf{w}_c$  define the subspace for UBM component  $c$ . This allows the use of full covariance matrices without data sparsity problems. There are two levels of mixing in this formulation. The first level, performed by the weights  $w_{cim}$ , creates unique mixtures for each  $(i, m)$  pair (which is called a substate). The second level of mixing, performed by the weights  $k_{im}$ , blends the substate mixtures to fit the training data in each tied state; this helps account for different ways that a single phonetic unit can be realized (i.e., depending on context).

Two aspects of the SGMM provide opportunities for bandwidth reduction:

- The parameters can be split into two groups: global parameters and state-specific parameters. It is possible to store the global parameters on-chip and fetch a subset of the state-specific parameters for each evaluation.
- “Gaussian selection” [42] can be performed to prune the set of mixture components for each frame, or for each parallel group of frames. Parameters relating to the pruned components, which would not contribute significantly to likelihoods, do not need to be fetched from memory. Posteriors for each component can first be evaluated against a smaller, diagonal covariance UBM to save time and bandwidth.

The quantization and parallelization techniques that we used for GMMs also apply to SGMMs. For a

fair comparison, the 1 Mb on-chip memory (which in the GMM case was used to store likelihoods only) is split between likelihoods and intermediate results, reducing the number of frames that can be evaluated in parallel. We obtain the bandwidth/accuracy tradeoff in Figure 2-6 for models with 1.2k or 2k tied states and 2k, 4k, or 8k total substates. All models used a 400-component UBM and selected the top 3 components for each frame. The SGMM is capable of better accuracy than the GMM (e.g., 9.4% vs. 11.3% WER) except at the lowest levels of memory bandwidth.

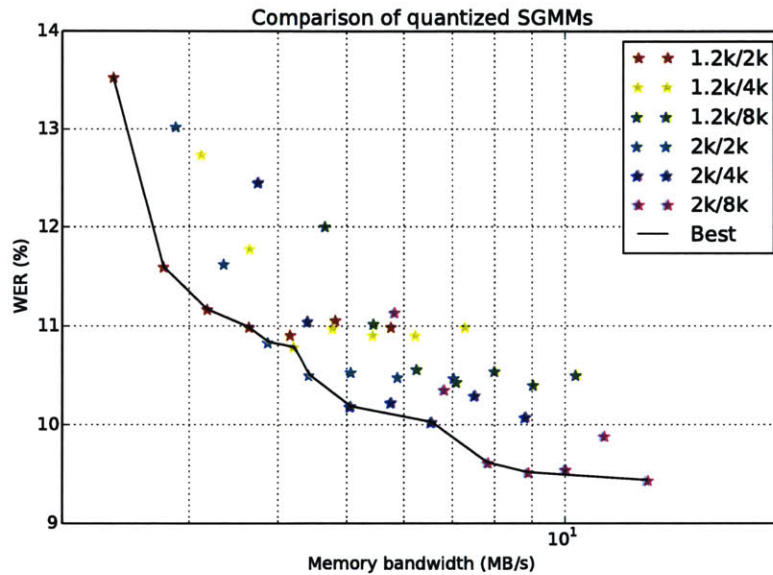


Figure 2-6: Bandwidth/accuracy tradeoff for SGMM acoustic model. Each color represents a model size (number of tied states and total number of substates), as shown in the legend. The stars plotted in each color represent different quantization depths applied to all parameters.

### 2.1.3 Deep neural network (DNN)

A neural network is a sequence of layers that operate on real-valued vectors, as shown in Figure 2-7. Each layer  $l$  performs an affine transformation of its input followed by an element-wise nonlinearity:

$$\mathbf{x}_{l+1} = g(\mathbf{W}_l \mathbf{x}_l + \mathbf{b}_l)$$

Neural networks with more than 2 layers are often called deep neural networks, or DNNs.

This is a “feed-forward” network topology that considers one feature vector at a time. Recent ASR research has demonstrated the usefulness of other layer types—recurrent and convolutional layers—in neural

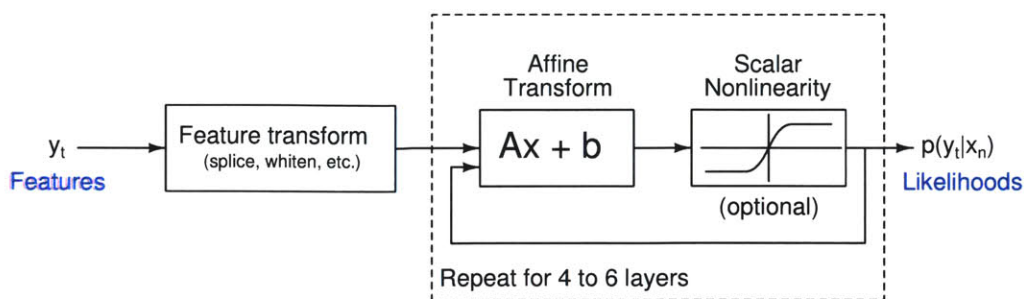


Figure 2-7: Schematic of NN evaluation.

networks. Recurrent layers (e.g., RNNs) accept input from the outputs of the previous layer at time  $t$  as well as the same layer at time  $t - 1$ . Convolutional layers (e.g., CNNs) apply a set of convolutions instead of a general affine transformation to the input vector. Supporting these layer types complicates the design process. With CNNs, the reuse of parameters necessitates more flexible architectures and mechanisms that limit on-chip and off-chip data movement. With RNNs, the dependency between frames precludes straightforward parallel evaluation. At the time of this work, only feed-forward networks were supported by Kaldi; support for convolutional and recurrent layers was added in 2015. We defer these modeling techniques to future work; see Section 7.2.<sup>2</sup>

Feed-forward networks usually perform better when the input features include temporal context. In our experiments, a 13-dimensional MFCC feature is concatenated with features from the previous 5 frames and the next 5 frames, so the NN's input dimension is 143. The NN's output dimension is the number of tied states; one forward pass through all layers jointly evaluates all of the modeled likelihoods.

Training is performed using stochastic gradient descent, using backpropagation to evaluate the gradients of the parameters with respect to an objective function. This is time-consuming, even with GPU acceleration. With feed-forward DNNs, training is typically run to convergence with a cross-entropy (frame accuracy) objective and then again with a minimum Bayes risk (phone accuracy) objective to further improve WER [86].

In decoding, most of the time is spent multiplying the layers' weight matrices by their input vectors. These weights also account for the majority of memory bandwidth. To save bandwidth, the weights can be quantized and multiple frames can be evaluated in parallel. Another trick reported in [93] is to create sparse matrices and store only the nonzero weights; that study found that 81–88% of the coefficients could be eliminated with no accuracy penalty, depending on the task (some additional space is needed to store the

<sup>2</sup>This is an interesting research area, but the author declined to invest time in experiments and design changes in pursuit of uncertain performance benefits. He needs to graduate eventually.



sparsity pattern). It helps to re-train the network with a fixed sparsity pattern. It is also possible to store a low-rank approximation of weight matrices computed via SVD [56]. We support this approach, but there is a risk of increased numerical error after two quantized matrix/vector multiplications (without a nonlinearity in between).

NN evaluation requires working memory proportional to the width of the largest layer. This memory is used most efficiently if all of the layers, including the output layer, have approximately the same number of outputs. This is in contrast to most software implementations where the output layer is large (2k–16k nodes) relative to the hidden layers (512–2k nodes).

Figure 2-8 illustrates the impact of weight quantization, using the same scalar quantization scheme as our GMM experiments. The different traces in these plots represent different levels of weight sparsity; all networks are the same size (512 nodes per layer; 6 hidden layers). Memory bandwidth is almost exactly linear in the quantization depth because most of the parameters are weights stored in bit-packed arrays. There is some variance in word error rates, but in general there is no gain from using more than 8 bits for weights. Subjectively acceptable accuracy is achieved with 5 bits per weight. (These experiments isolate the effect of the weights: biases are stored at 32-bit precision, and feature transforms are performed in floating point.) As stated by [85], the activation function (e.g., sigmoid) tends to compress numerical errors introduced in the affine transformation. Furthermore, numerical errors in the acoustic likelihoods do not impact system accuracy if they are sufficiently small. There is room to explore even more aggressive quantization and model compression with NNs (e.g., 3-level weights [81]).

Accuracy and bandwidth results for a variety of NNs, with varying dimensions and sparse weight matrices, are shown in Figure 2-9. For this task, small NNs perform almost as well as large NNs, echoing the results of [44]. A quantized sparse NN with 512 nodes per layer requires about 4 MB/s of memory bandwidth to evaluate, with 10% relative WER degradation compared to the best model tested.

#### 2.1.4 Model comparison and discussion

In order to compare GMMs, SGMMs and DNNs, we select the Pareto optimal subset of results from each model framework. This allows us to see the tradeoff between WER and memory bandwidth, assuming that the hyperparameters such as quantizer depth and model dimensions have been selected correctly. The results are shown in Figure 2-10 for the Wall Street Journal (WSJ) dev93 task [60]. Only the acoustic model’s contribution to memory bandwidth is included.

This comparison shows that the DNN and SGMM both offer competitive performance, even when bandwidth (and hence model size) are restricted. We selected the DNN framework for our implementation be-

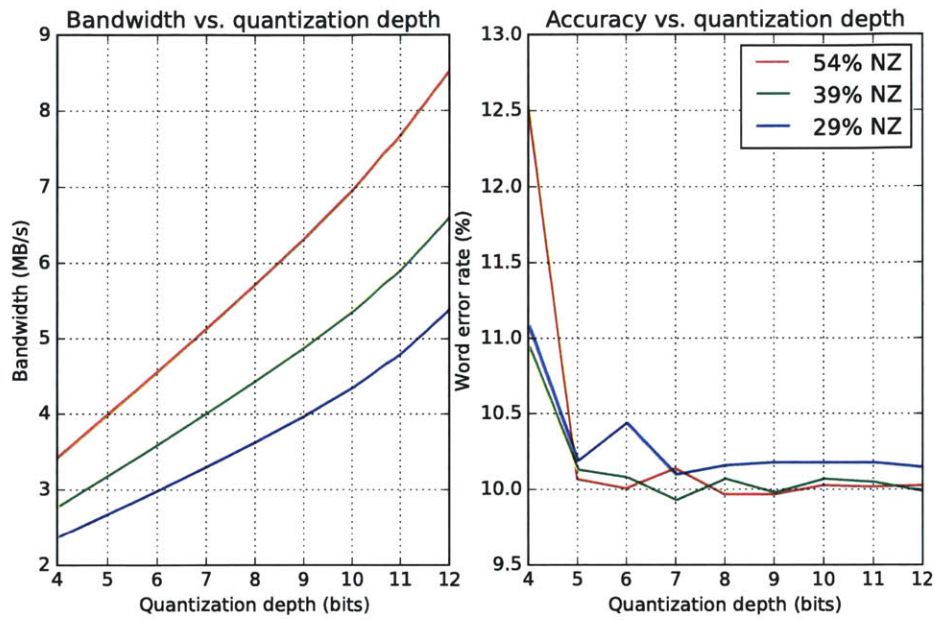


Figure 2-8: Bandwidth (left) and accuracy (right) of 512-node, 6-layer NN acoustic models trained with different fractions of nonzero (NZ) weights.

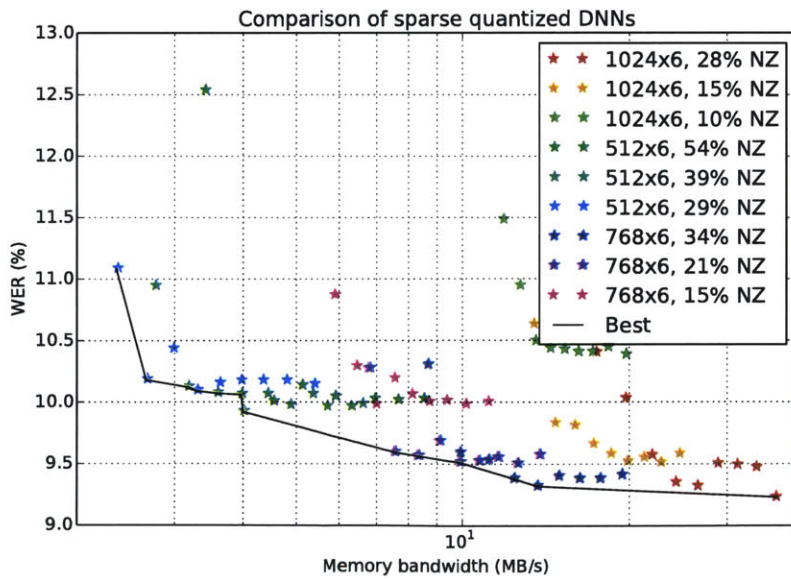


Figure 2-9: Bandwidth/accuracy tradeoffs across many NN acoustic models. The dimensions and weight matrix sparsity of each model are shown in the legend.

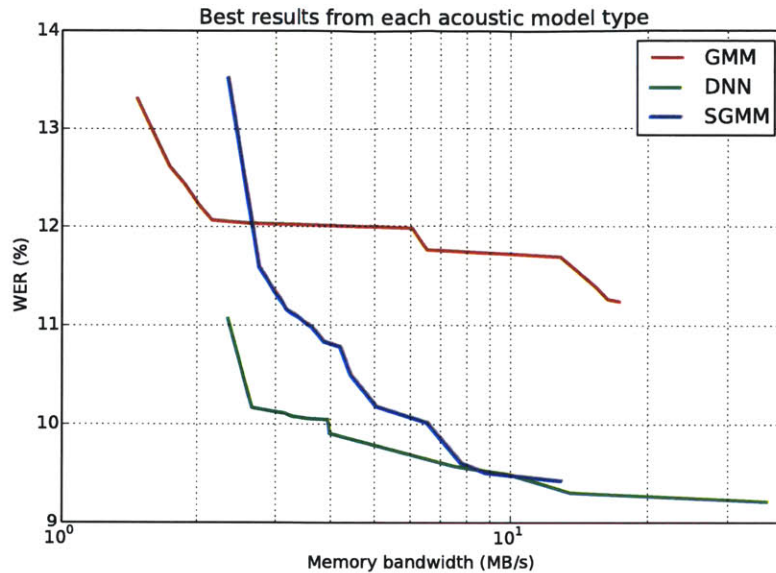


Figure 2-10: Bandwidth/accuracy comparison of acoustic model frameworks, using the best results from each framework.

cause:

- NN evaluation has a more repetitive structure than SGMM evaluation, leading us to expect smaller circuit area and design effort.
- The bulk of acoustic modeling research is based on NNs; future improvements developed by the ASR research community could be ported to hardware with incremental changes in architecture.

## 2.2 Neural network implementation

### 2.2.1 Architecture and general tradeoffs

We use a fixed-function SIMD architecture shown in Figure 2-11 to evaluate neural networks. The sequencer decodes a compressed parameter stream arriving from off-chip memory, and sends weight and bias coefficients to the execution units (EUs). Each EU has local memory for storing the feature vector, intermediate results, and log-likelihood outputs for one frame. This means the network parameters are the only data fetched from off-chip memory, and that memory access is read-only.

This architecture presents a challenge for scalability. Increasing the size of local memories would allow the use of wider networks (or more output targets). But it would also take up area that could otherwise be

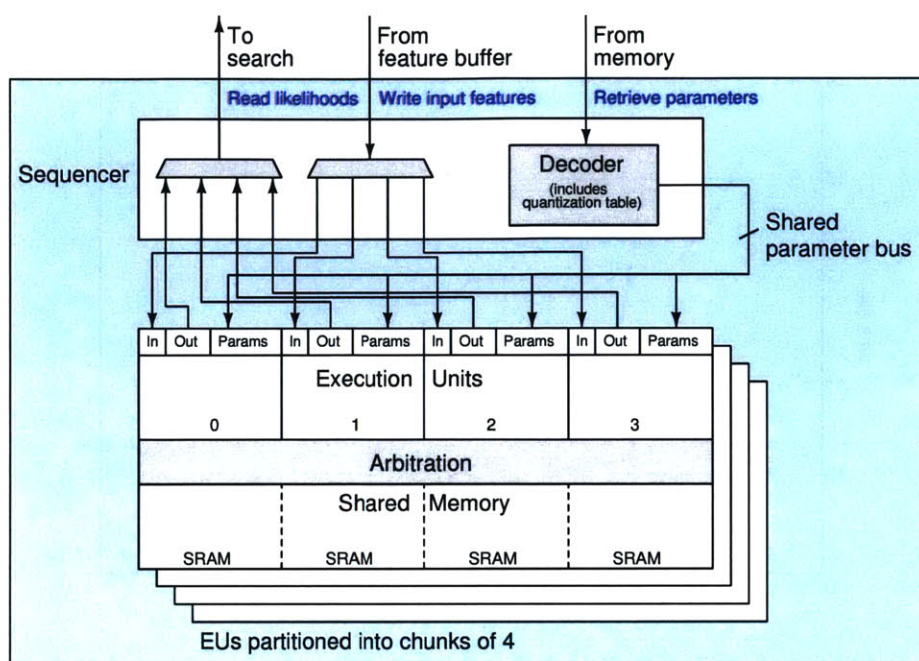


Figure 2-11: Block diagram of SIMD NN evaluator, relying on a shared sequencer and 32 execution units.

used for more execution units. Furthermore, SRAMs have some area overhead from sense amplifiers and I/O logic, meaning that area is not linearly proportional to depth (although it is proportional to width).

As a compromise, we group EUs into chunks which each arbitrate access to a group of SRAMs (Figure 2-12). For small networks, all EUs are active and each EU talks directly to one SRAM. For networks that are large enough to overflow one SRAM, every other EU is disabled, but the active EUs address two adjacent SRAMs. For a network that would overflow two SRAMs, three out of every four EUs are disabled, and the active EUs use four adjacent SRAMs. This scheme could be extended to handle a wider range of network sizes if required by the application.

We opted to provide 32 EUs (eight chunks of four EUs each), and support up to 1024 nodes per layer without penalty (4096 nodes maximum). Numerical precision was set conservatively, and could be further optimized; all real values are stored at 24-bit precision, so this arrangement requires 1.5 Mb of SRAM. To reduce area overhead, each EU has one single-port SRAM that stores both the input and output vectors for the current layer (rather than separate SRAMs for input and output). Having 32 EUs means that small networks only need to be evaluated  $3\frac{1}{8}$  times per second, reducing both memory bandwidth and clock frequency.

The model data structures are designed so that the entire model is accessed through a single sequential read, avoiding page access penalties. During model preparation, each layer is analyzed to select either a

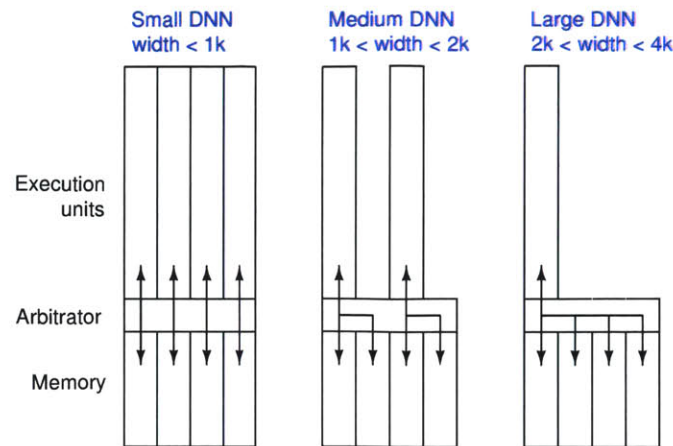


Figure 2-12: Execution units and local memory are grouped into chunks that can be reconfigured for different network sizes.

sparse or dense weight matrix format (whichever uses less space). In addition to affine layers, we support element-wise addition and multiplication operations which can be used for feature transforms.

### 2.2.2 Scalability

This architecture works in concert with model design to provide about two orders of magnitude in energy scalability across different ASR tasks. The time required to evaluate a NN model is dominated by the matrix/vector multiplications in affine layers, which grows with the square of the layer width (number of nodes). Furthermore, the maximum layer width dictates the amount of parallelism that is possible. This means that the overall computation time (and, to a first approximation, energy) grows with the cube of the network width.<sup>3</sup> Table 2.1 shows the model dimensions, bandwidth, and computation time for the NNs used in our ASR tasks.

Circuit area scales linearly with the number of EUs. This creates an area/power tradeoff: having more EUs allows lower memory bandwidth, and slightly lower core power due to reduced clock frequencies and voltages. The number of EUs in our implementation was limited by concerns about latency and area. Worst-case latency is proportional to the number of EUs, since search cannot proceed until the acoustic model is evaluated. With 32 EUs, the acoustic model occupies 44% of total ASR area.

Circuit area also scales linearly with the amount of memory provided to each EU. The memory area depends on width (i.e., numerical precision, which has not been explored in detail by this work), and depth

<sup>3</sup>The same is true of memory bandwidth, since the computation proceeds as quickly as parameters can be provided by the external memory over an 8-bit interface. However, there is some overhead due to internal stalls; this is generally negligible for dense networks but was measured at 14% for the sparse network on WSJ-dev93.

Task	Nodes per layer	Num. layers	Sparsity (% nonzero)	Model size	Runtime (cycles)
TIDIGITS	64	4	100	32 kB	112k
WSJ (eval92-5k)	256	6	100	577 kB	1.84M
Nutrition	256	6	100	771 kB	2.45M
Jupiter	512	6	100	2.62 MB	8.28M
WSJ (dev93)	1024	6	31	3.71 MB	13.2M

Table 2.1: Scalability owing to design of different NN acoustic models. Runtimes are quoted in clock cycles per second of utterance time.

(i.e., maximum NN layer width). It would make sense to shrink these memories in order to retarget the design to easier ASR tasks. For example, cutting this depth in half would not affect throughput or off-chip bandwidth for NNs with up to 512 nodes per layer, but would reduce memory area by 750 kb.

Unused EUs are clock gated. When evaluating wide networks,  $\frac{1}{2}$  or  $\frac{3}{4}$  of EUs are unused, as described above. Some EUs are also unused when performing the final acoustic model evaluation, unless the utterance length is a multiple of 32 frames. Finally, the user can limit the number of frames evaluated in parallel; this will increase amortized memory bandwidth and runtime, but decrease latency.

### 2.2.3 Sequencer and EU design

The interface between the sequencer and EU includes a command FIFO and a data FIFO that supplies uncompressed parameters as (`index`, `value`) pairs. The EU has little control logic of its own; it simply applies the commanded arithmetic operation and first argument (`value`) to a second argument from the proper memory location (`index`). These interfaces are broadcasted to all active EUs. Each active EU manipulates a vector pertaining to one frame, storing this vector in its local memory.

The sequencer is shown in Figure 2-13. It interprets the NN data structure as it is streamed in from external memory, one byte per clock cycle. This structure has a hierarchical form specified using our “binary-backed” methodology (see Section 5.2.1) and illustrated in Figure 2-14. A stack is used to keep track of the current location within this hierarchy. A NN is a list of layers; each layer can be specified in either sparse or dense form. Both forms store a bias vector in 24-bit Q8.16 format<sup>4</sup> and a quantized weight matrix (the matrix is excluded for layers that only perform element-wise multiplication or addition). Each weight matrix has its own quantizer, to account for different distributions of weights without loss of accuracy. As the first part of the structure is streamed in, Q8.16 quantizer levels are written to a LUT (up to 4096 levels, if 12-bit

<sup>4</sup>To clarify, this means that a 24-bit two’s-complement value is interpreted as a real number from -128 to 128. 1.0 is represented by `0x010000` and  $-\frac{1}{16}$  by `0xFF0000`.

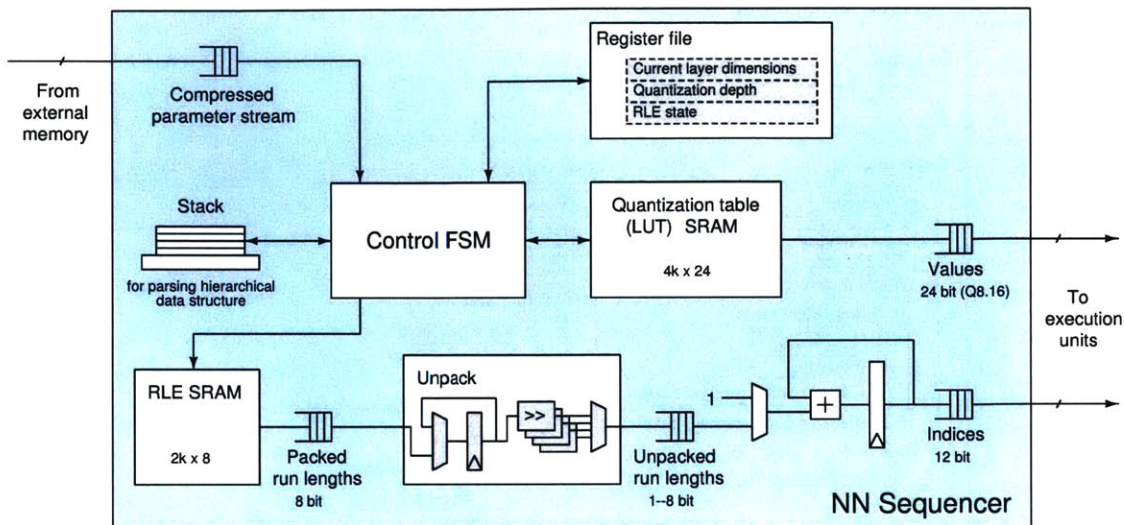


Figure 2-13: NN sequencer architecture supporting sparse and quantized weight matrices.

indices are used). In a dense layer, the next part of the stream is a bit-packed blob of quantizer indices. The sequencer reads each quantizer level from the LUT and pairs them with serially incremented indices (0, 1, 2, ...). Hence the sequencer is instructing the EUs to perform sequential matrix/vector multiplication in row-major order.

Sparse layers contain additional structures to improve compression of the weight matrix.<sup>5</sup> We suggest training NN weight matrices that are 10–50% nonzero; these matrices have a relatively uniform (random) sparsity pattern, which would be well-suited to compressed sparse row (CSR) format. However, the CSR format explicitly stores the column index of each nonzero value, which can more than double the storage size of the matrix. This is because column indices require 12 bits each (to accommodate our maximum layer size of 4096 nodes), but the nonzero values are represented by quantizer indices of 4–12 bits. To improve compression, we store the column indices of nonzero values as a run length encoding (RLE). An RLE for the sparsity pattern of the entire matrix would be quite large; to reduce the local memory required by the sequencer, we split the matrix into a list of sparse vectors (one per row). In the parameter stream, nonzero weights for each row are preceded by a local RLE of the zeros in that row. The output index counter is incremented by the run length (rather than 1, as in the dense mode) to obtain the correct column indices.

Figure 2-15 illustrates the RLE, which is parameterized by the bit width of each run length. In  $k$ -bit format, runs of at least  $2^k - 1$  zeros are handled by chaining. For example, to represent a run of 10 zeros

<sup>5</sup>The goal of this is to reduce memory bandwidth, although there are residual benefits in core power consumption due to reduced clock frequency and switching activity.

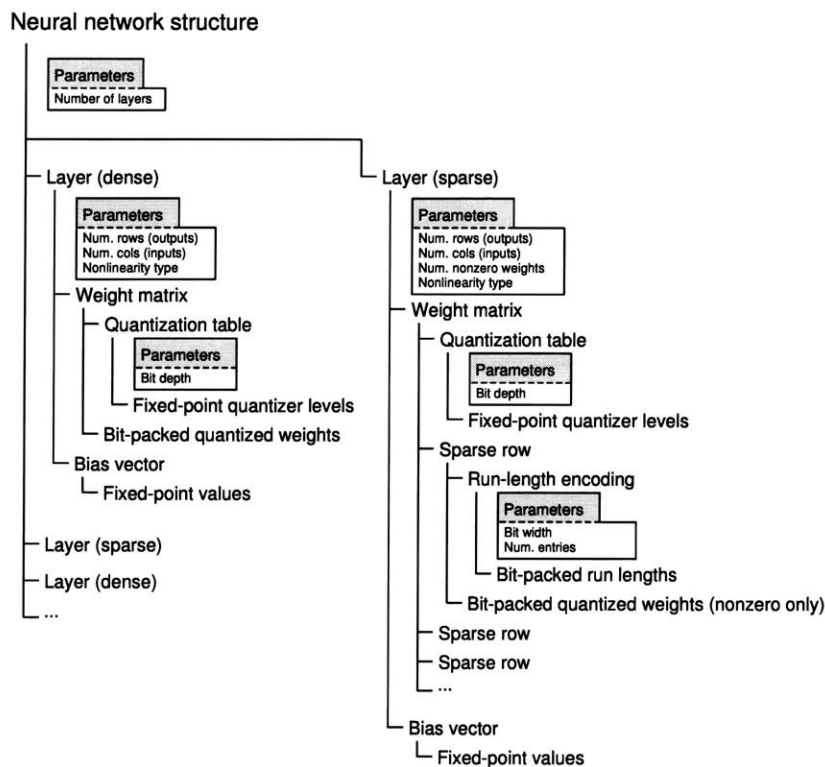


Figure 2-14: Compressed neural network data structure. Each layer is stored in either sparse or dense format (whichever uses fewer bytes).

in 3-bit form, we would use two entries: 111 and 011. The first entry tells the sequencer to increment the column index by 7 (without reading a nonzero value), and the second entry tells it to increment the column index by 3. A single entry can specify a run up to length  $2^k - 2$  (e.g., 6, when  $k = 3$ ). Regardless of the selection of  $k$ , the sequencer streams the RLE into an 8-bit wide memory and unpacks values as needed. The unique sparsity patterns of each row are described by the RLE representation that requires the minimum number of bits. Overall, the organization of the NN data structure optimizes model size and data locality within the parameter stream.

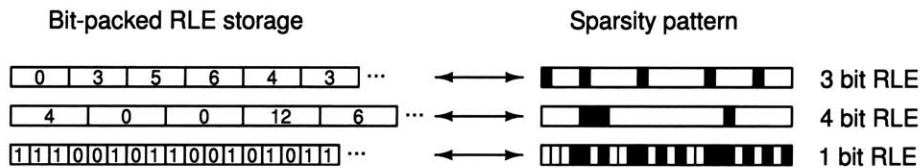


Figure 2-15: Variable-width run length encoding is used to represent the column indices of nonzeros within each row of a sparse weight matrix.



The EU performs arithmetic operations on a vector stored in its local memory; it is shown in Figure 2-16. Arithmetic units are fed by FIFOs from the sequencer and the local memory; control logic determines which arithmetic unit output to write back, depending on the command from the sequencer. Matrix/vector multiplication is performed with the aid of a 24-bit MAC, permitting a single-port SRAM to be used for local memory. (There is a brief stall to write the result at the end of each row.) The vector can be fed through an element-wise sigmoid or rectified linear (ReL) nonlinearity. The ReL nonlinearity should be used with caution because it does not suppress errors caused by quantized matrix coefficients. While ReL NNs deliver similar accuracy to sigmoid in floating-point software experiments, they require finer weight quantization (and thus more memory bandwidth) in the context of this architecture.

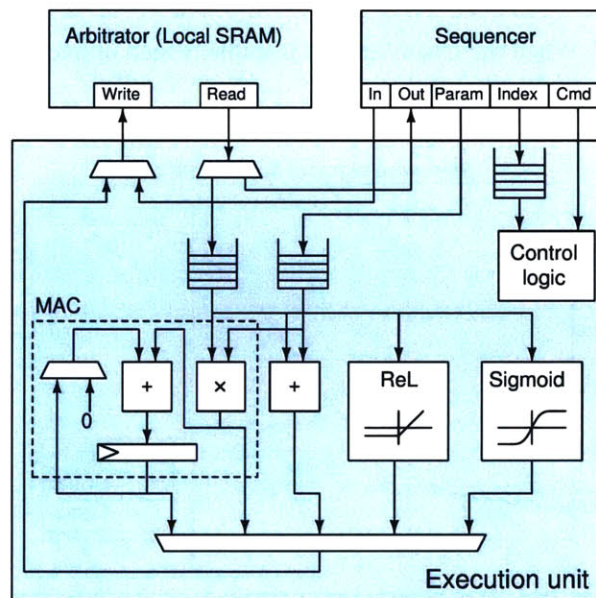


Figure 2-16: Execution unit block diagram (pipelining, addresses, and control signals not shown).

### 2.2.4 Sigmoid

The sigmoid operation  $\sigma(x) = \frac{1}{1+e^{-x}}$  could be implemented exactly in logic. However, given that there are 32 EUs, it's worth trying an approximation to save area. The WER impact of weight quantization and fixed-point arithmetic is tolerable; so is the impact of numerical errors in sigmoid outputs. We detected no difference in WER between sigmoid implementations with error bounds of  $10^{-4}$  and  $10^{-6}$ .

The accuracy of a transcendental function approximation depends on the order and granularity of the fit. At one extreme, a LUT is very granular (specifying the function value at hundreds or thousands of locations)

but low-order (since the estimate is a constant close to each location). At the other extreme, a convergent series would approximate the function over the entire domain, using many terms to minimize errors. Neither of these is well-suited to our application: a sufficiently accurate LUT would be too large, and a sufficiently high-order series expansion would be too complex to evaluate.

To compromise, we perform a piecewise low-order polynomial fit. Figure 2-17 plots the sigmoid function along with linear and cubic fits to illustrate the concept.  $\sigma(x)$  is odd,<sup>6</sup> so the fit needs be performed only for positive  $x$ . We partition the positive real axis into 6 regions between  $0 \leq x \leq 12$ ; for  $x > 12$  we simply return the asymptotic value  $\sigma(x) = 1$  (the error is small:  $\sigma(12) = 1 - 6.1 \times 10^{-6}$ ). Within each region, we specify a 5th order polynomial to track  $\sigma(x)$ . As shown in Figure 2-18, a Chebyshev polynomial provides better accuracy than the Taylor series by balancing errors throughout the region, instead of minimizing error near the center of the region. When the Chebyshev fit is implemented in fixed-point arithmetic, the error is below  $10^{-4}$  over the entire domain.

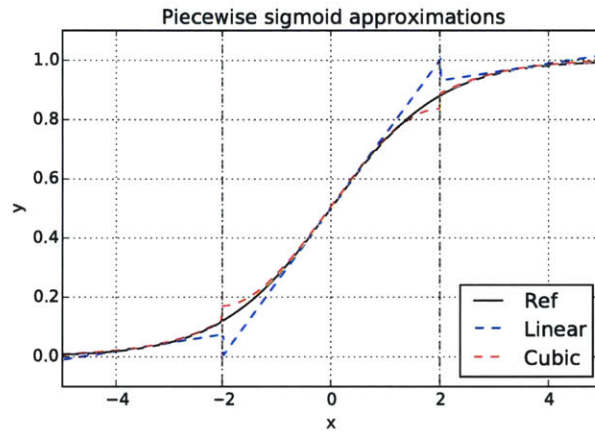


Figure 2-17: The sigmoid function and simple piecewise polynomial fits.

Figure 2-19 shows an unpipelined version of the sigmoid approximation circuit. A set of coefficients totaling 720 bits are stored in a register file. Comparators are used to select the appropriate set of coefficients based on the magnitude of  $x$ . The sign of  $x$  is removed and the polynomial is evaluated iteratively using Horner's method (e.g.,  $y = c_0 + x(c_1 + \dots + x(c_{m-1} + c_m x))$ ) at 24-bit precision. Results for negative  $x$  are complemented to account for the odd behavior of the sigmoid function:  $\sigma(-x) = 1 - \sigma(x)$ . For a polynomial of order  $m$ , the latency of this operation is  $m+2$  cycles. We compared pipelined and unpipelined versions of the sigmoid operator; we selected the unpipelined version since the performance penalty of its 7-cycle latency was small (relative to the overall time needed to multiply weight matrices) and easily justified

<sup>6</sup>At least within a constant.

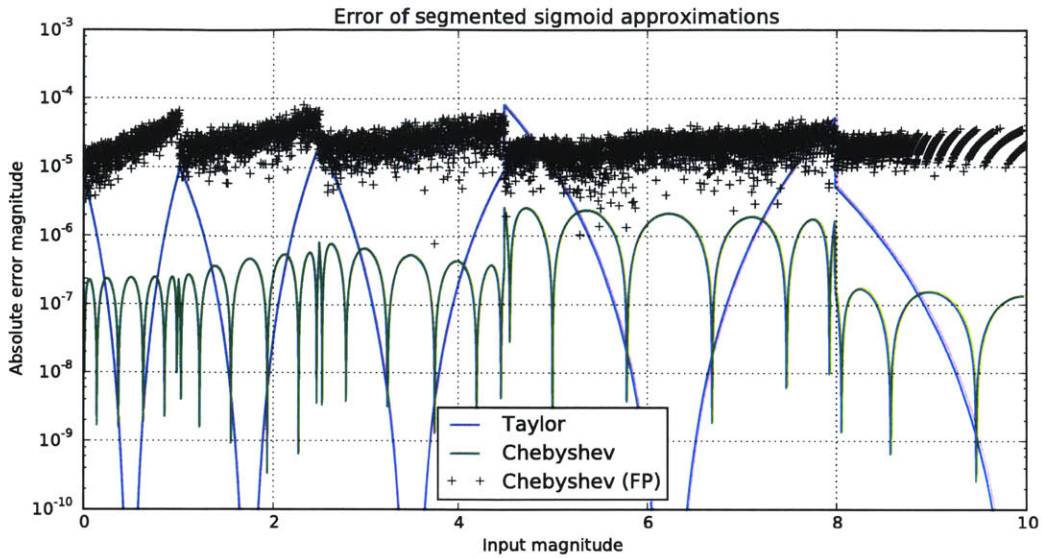


Figure 2-18: Approximation error of piecewise polynomial fits to sigmoid function, comparing Taylor and Chebyshev series.

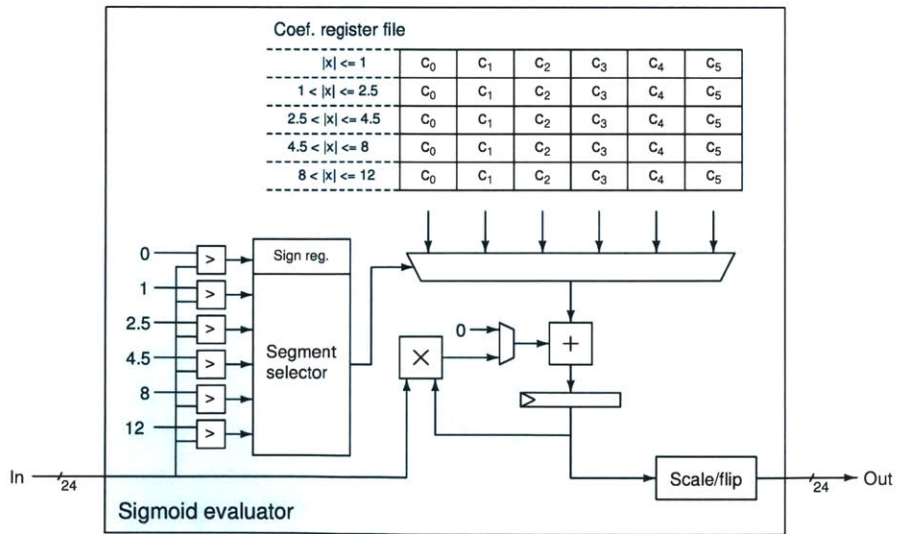


Figure 2-19: Block diagram of piecewise polynomial sigmoid evaluator.

by the area savings. Each instance of the sigmoid approximator has an area of 5.7k gates, including one multiplier.

## 2.3 Summary

This chapter discussed acoustic modeling in the context of a low-power hardware speech recognizer. The main departure from previous research has been to consider the memory bandwidth required to fetch model parameters, since this has a large impact on system power consumption. The accuracy advantage of DNNs over GMMs was maintained when the number of parameters was constrained. Fortunately, DNNs admit a straightforward circuit implementation and are tolerant of numerical errors caused by parameter quantization and approximations to the sigmoid function. The DNN architecture we presented uses 32 parallel execution units, and has a total area of 884k gates plus 1.72 Mb of SRAM. It evaluates a 1024 node, 6 layer sparse network in real-time at 13.2 MHz with 11.6 MB/s of external memory bandwidth.

## Chapter 3

# Search

Our hardware implementation of Viterbi beam search is designed to approach Kaldi decoding accuracy (within reasonable area limits) and minimize memory bandwidth. Scalability can be achieved via model design (vocabulary and language model pruning), and at runtime by changing the beam pruning parameters. Off-chip memory writes are reduced or eliminated by converting the state lattice to a word lattice.

The search sub-system interprets a weighted finite-state transducer (WFST), stored off-chip, that has been constructed for each recognizer. The WFST is a generic data structure that can model the speech production process at different levels of abstraction [55]. We use Kaldi’s *HCLG* WFST construction with modifications described in Section 5.3.2. WFSTs built for large vocabulary recognizers typically have 1–100M states and 2–4 arcs per state.

The search sub-system works with the acoustic model to find a word sequence for each utterance:

- At the beginning of the utterance, it creates a set of hypotheses based on the initial states of the WFST.
- With each new acoustic observation (feature vector), it propagates active hypotheses forward in time. This uses the Viterbi approximation:  $p(x_{t+1}) \approx \max_{x_t} p(x_t) p(x_{t+1}|x_t) p(y_{t+1}|x_{t+1})$ .
- At the end of the utterance, it applies final weights and reconstructs the most likely trajectory through the WFST. The sequence of output labels on these arcs is the word transcription.

A hypothesis links a particular state index  $X$  in the WFST with the likelihood of being in that state at a particular time, i.e.,  $p(x_t = X)$ , as well as any metadata that is necessary to reconstruct word sequences passing through that state. We often refer to the log-likelihood as a “score” which shows how attractive the hypothesis is. We also use the name “active state” for a hypothesis being considered at the current time.

### 3.1 Architecture

This section describes how to perform Viterbi search with limited local memory, establishing a baseline architecture and then explaining our refinements. The module has two state lists with a pipeline of processing stages in between, as shown in Figure 3-1. One list contains the state hypotheses for frame  $t$  and the other contains the state hypotheses for frame  $t + 1$ . Pipeline stages can be separated by FIFOs (not shown) to keep each stage operating at high utilization despite variable latency.

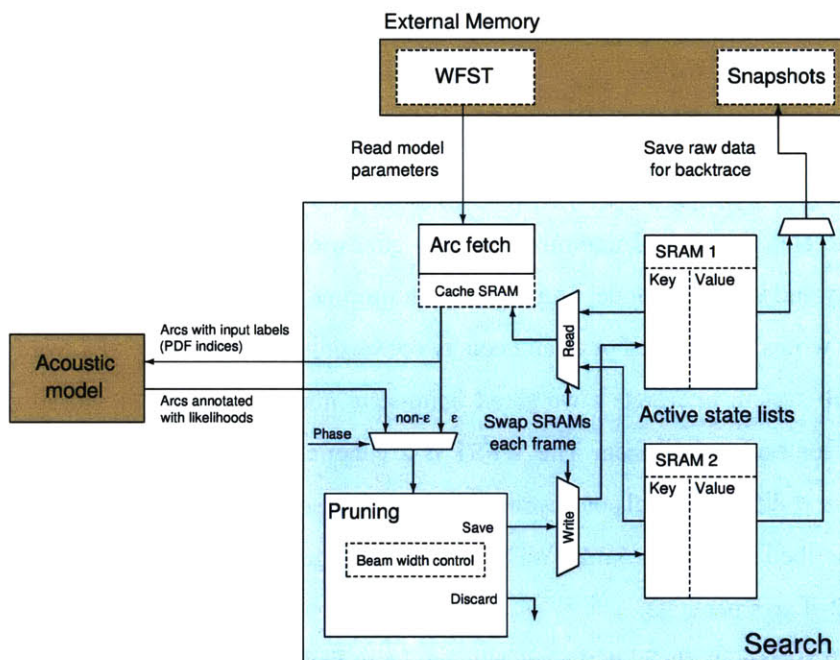


Figure 3-1: Baseline search architecture with two-frame state list memory.

At the beginning of the utterance, we insert a configurable starting state (i.e., state 0) with a likelihood of 1, and then expand  $\epsilon$  arcs to all of the other initial states of the WFST.

We perform a forward pass of Viterbi search at each time step (frame). At the beginning of the forward pass, the state list for frame  $t + 1$  is empty. Each state from frame  $t$  is read and the WFST is queried for arcs leaving that state. The destination of each arc is another state which becomes a hypothesis for frame  $t + 1$ . The likelihood of these new hypotheses includes the likelihood of being in the source state, the transition probability (arc weight) from the WFST, and the acoustic likelihood for the appropriate tied state distribution (arc input label). Following the Viterbi approximation, only the most likely arc into each state is considered for selecting and scoring hypotheses:  $p(x_{t+1}) \approx \max_{x_t} p(x_t) p(x_{t+1}|x_t) p(y_{t+1}|x_{t+1})$ .

The forward pass is divided into two phases: the “non- $\epsilon$ ” phase and the “ $\epsilon$ ” phase.<sup>1</sup> In the non- $\epsilon$  phase, all of the source states from frame  $t$  are expanded. However, only the arcs with non- $\epsilon$  (nonzero) input labels are considered; the  $\epsilon$ -input arcs are discarded, because they do not advance the hypothesis forward in time. Each non- $\epsilon$  arc is scored with a likelihood from the acoustic model, and the destination states with sufficiently high scores (more on this later) are saved as hypotheses for frame  $t + 1$ . Then, in the  $\epsilon$  phase, all of the source states from frame  $t + 1$  are expanded. Only the  $\epsilon$ -input arcs are considered, so the acoustic model is not needed. High-scoring arcs are saved as hypotheses for frame  $t + 1$  (they do not advance forward in time). In order to eventually recover the word sequence, the list of active states is saved after each forward pass (as a “snapshot” to external memory).

The  $\epsilon$  phase has a few quirks. Because we are reading and writing the same set of states (frame  $t + 1$ ), we need a way to avoid potentially infinite recursion. We preprocess the WFST so that all paths of only  $\epsilon$  arcs are bypassed by a single  $\epsilon$  arc having the same total weight [16]. Some paths may traverse multiple arcs with non- $\epsilon$  output labels, so we create multi-word output symbols (typically a small number) as necessary to bypass these paths; see Figure 3-2. We keep track of whether each state in frame  $t + 1$  was originally added in the non- $\epsilon$  phase or the  $\epsilon$  phase. If it was added in the  $\epsilon$  phase, we do not expand it; this is still correct because (thanks to WFST preprocessing) all states that are reachable at frame  $t + 1$  were reachable via a single  $\epsilon$  arc from a state that was added in the non- $\epsilon$  phase.

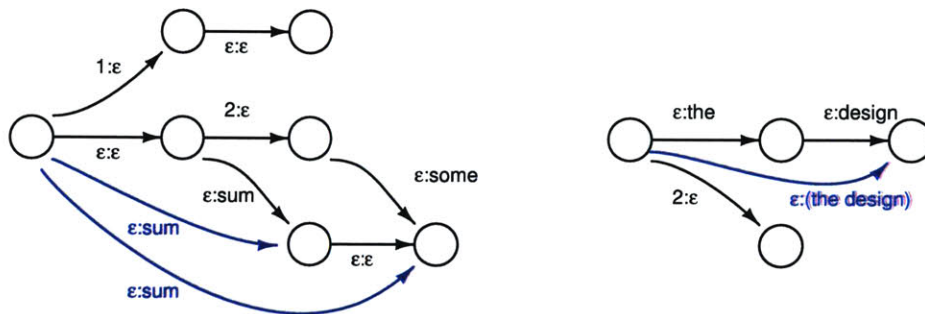


Figure 3-2: WFST preprocessing with multi-word output symbols. Left: Adding  $\epsilon$  arcs to bypass arbitrary length chains of only  $\epsilon$  arcs [16]. Right: Compound word labels are created for arcs that bypass multiple non- $\epsilon$  output labels.

After all of the audio has been consumed to create feature vectors, and all of the feature vectors have been consumed by forward search, it’s time for a final weight update. This involves reading all stored states and adding final weights from the WFST to their likelihood scores. Not all states have a final weight (or,

<sup>1</sup>In Kaldi’s decoders the non- $\epsilon$  phase is performed by the `ProcessEmitting()` method and the  $\epsilon$  phase is handled by `ProcessNonemitting()`.

equivalently, their final weight is infinite). If any of the states have a final weight, then only those are considered as valid endpoints of the search. If none of the states have a final weight, then all are considered valid. The most likely valid state is used as the endpoint for a backtrace which produces a sequence of word labels. This backtrace could be constructed by reading all of the state snapshots (stored in external memory) in reverse order.

Our first ASR chip in [67] implemented the search architecture shown in Figure 3-1. In experiments with a 5,000 word vocabulary, the memory bandwidth requirements of search were 6.02 MB/s (read) and 1.50 MB/s (write), and the overall energy efficiency was 16 nJ per hypothesis.

Several changes can be made to this baseline architecture without changing the behavior of the search algorithm. For a small memory area savings, we can consolidate the two state lists into a single one with separate fields for the “source” ( $t$ ) and “destination” ( $t + 1$ ) frames; this is beneficial because most states remain active for several frames at a time. More significant optimizations have been made to reduce off-chip memory bandwidth. WFST compression and caching techniques are used to reduce read bandwidth, and a word lattice is used to reduce write bandwidth. The resulting architecture is shown in Figure 3-3 and explained in the following sections.

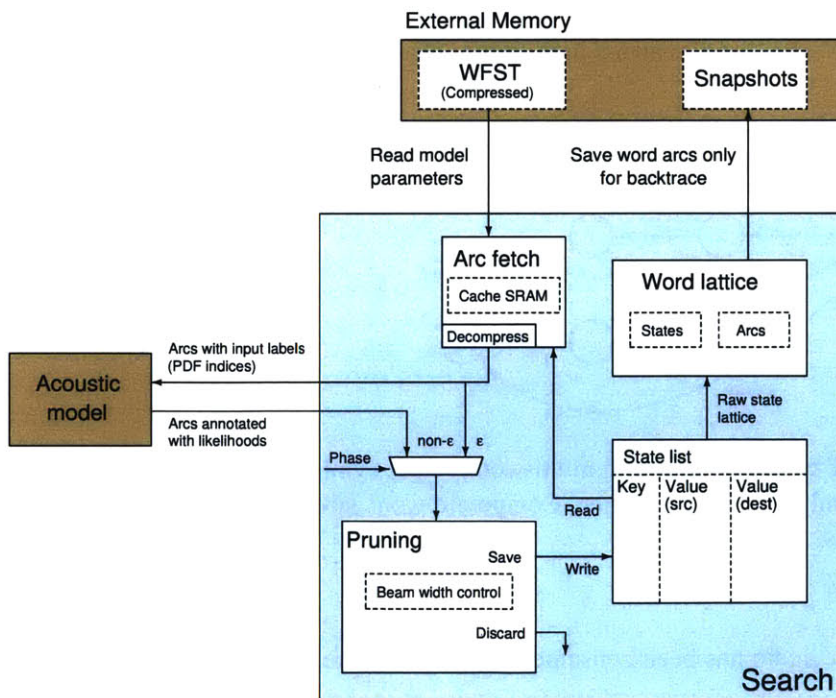


Figure 3-3: Improved search architecture including word lattice and merged state lists.



## 3.2 Building blocks

This section describes how we implemented the state list, WFST fetch, pruning, and word lattice functions required by our revised search architecture.

### 3.2.1 Hash table

In ASIC design it is helpful to keep data structures simple, so we transformed all of the algorithms on our chip to manipulate flat and associative arrays of packed structures. (There are no linked lists or trees, for example.) The state list, word lattice, and WFST cache described below rely on a hash table structure that we have implemented using SRAM. Content-addressable memory would be faster but require custom memory design and result in lower density [57]. Our hash table is an associative array (key/value store) supporting key-based and address-based read, write, and delete operations.

Each location in SRAM stores a (key, value) pair and occupancy flag as shown in Figure 3-4. Each key is translated to an address using the hash function—in our case, simply the lower  $m$  bits of the key (where the array size is  $2^m$ ). Because there are more possible keys than memory addresses, two different keys will sometimes hash to the same value (a collision). When we encounter a collision, we scan subsequent addresses until we find a slot that is empty (for inserting a new key) or matches our key (for reading, or writing an existing key). This is called linear probing [58]. This naive hash table algorithm is relatively simple to implement in hardware and provides sufficient performance for our purposes.<sup>2</sup>

	1 bit	$N_k$ bits	$N_v$ bits	
Address	Occupied?	Key	Value	
0	0	xxx	xxxxxxxx	
1	0	xxx	xxxxxxxx	
2	0	xxx	xxxxxxxx	
③	1	4A3	00101001	Key hashed to compute base address
4	0	xxx	xxxxxxxx	
5	1	5D5	10110000	Address incremented linearly on hash collision
6	1	105	01110111	
7	0	xxx	xxxxxxxx	

Figure 3-4: Data storage format for SRAM-based hash table with open addressing. Linear probing is used to resolve collisions.

Figure 3-5 enumerates the operations supported by the hash table and shows the state sequence used to execute each operation. To keep the keys in a consistent ordering, some keys need to be moved “up” in

<sup>2</sup>Despite the apparent simplicity of these algorithms, the hardware implementation is a cautionary tale about the need for thorough testing. See Section 5.2.2.

memory after deletion. The performance of key-based operations degrades as the load factor of the table increases; the table should never be completely filled (reasonable limits are 75–90% of capacity).

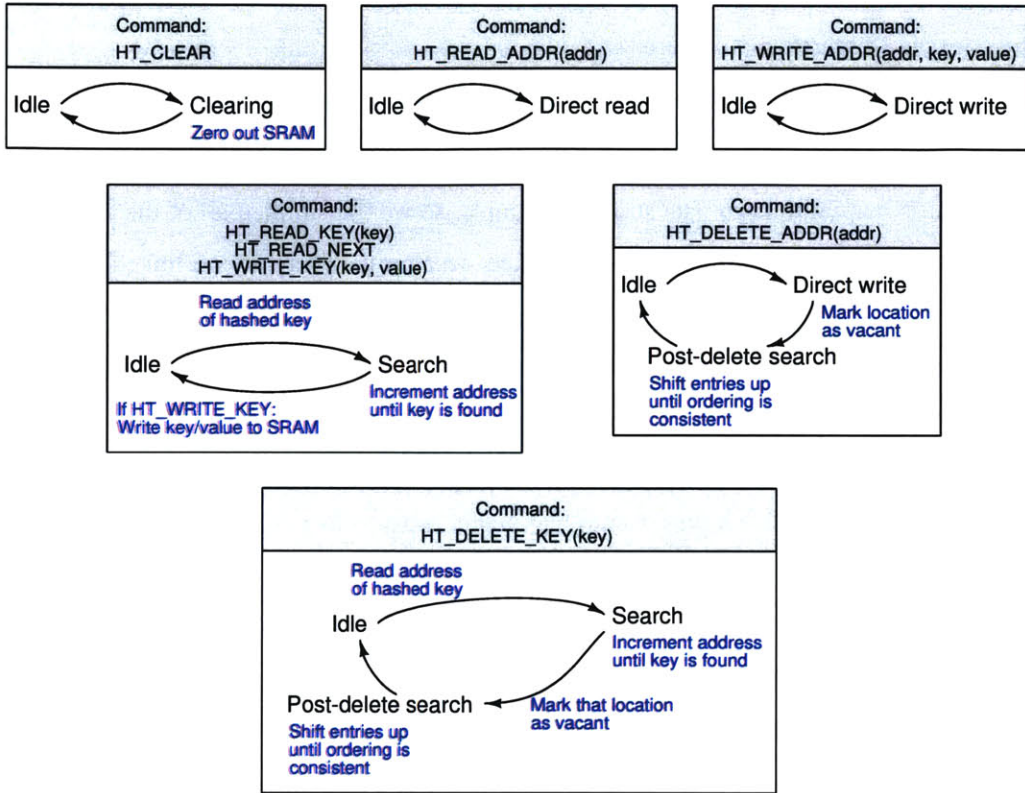


Figure 3-5: State transition diagrams for SRAM-based hash table. (Resizing is performed as needed in between commands, using the same basic operations.)

Address-based operations facilitate shortcuts: writing or deleting an entry immediately after reading it, and scanning the table for all stored keys.<sup>3</sup> The state list and word lattice do this regularly. However, iterating over all keys is wasteful if the hash table is nearly empty. To improve scalability, the hash table is dynamically resized. In our design, the initial table size is 32 entries. Up to the limit of the SRAM depth, the table will double in size when load factor exceeds  $\frac{3}{4}$ . It will halve in size when load factor diminishes below  $\frac{1}{16}$  (this threshold was chosen to reduce thrashing compared to other thresholds such as  $\frac{1}{4}$ ). Shrinking of the array can be temporarily disabled to ensure that all keys are found by a sequential scan.

In [67], dual-ported SRAMs were used so the hash table could read and write concurrently. We improved area efficiency by using single-port SRAMs, though this does add latency to delete operations (leading to a small increase in clock frequency for a given task).

<sup>3</sup>The hash table also has a “clear” operation that zeros out all memory locations, which is faster than explicitly reading and deleting all keys.

### 3.2.2 State list

The state list is the central data structure of our search architecture. The architecture has been designed to preserve memory locality. This is not only a benefit to energy efficiency, but a major convenience in a special-purpose digital design. It allows us to avoid having a shared memory hierarchy that connects to every module. Instead, each module has local memory for just the information it needs. There are only two external memory dependencies in the search sub-system: the WFST fetch and the word lattice.

Figure 3-6 shows the data structures stored in the state list and passed through each stage of the search pipeline:

- **State list:** This is a hash table storing active states (hypotheses). Every hypothesis is indexed by a particular WFST state, and linked with a (state, time) pair that we call a word lattice state (WL state); this will be explained in Section 3.2.5.
- **WFST request:** To expand a state in the WFST, we need to know its address and length (in bytes). In response, we receive a sequence of arcs. Each arc that we expand has a destination state, input and output labels, and weight. We no longer need to know the source state.
- **Scoring:** Before search is run, the acoustic model has computed a likelihood for each tied state (input label). For each unscored arc, we retrieve the appropriate likelihood and combine it with the source state score and arc weight to get the overall likelihood of the destination state. We no longer need the input label. (In the  $\epsilon$  phase, there is no acoustic likelihood.)
- **Pruning:** The scored arc has all the information necessary to save a new hypothesis in the state list, provided that its score exceeds the pruning cutoff (and that of any existing hypothesis at the same state). If the scored arc has a word label (i.e., non- $\epsilon$  output label), it will also be sent to the word lattice.

Bit fields have been sized conservatively to allow WFST sizes up to 4 GB, vocabularies up to 16M words, and acoustic models with up to 64k tied states. These fields could be reduced to save area in a more restrictive implementation.

A state diagram of the state list logic is shown in Figure 3-7. The state list is designed to perform a sequential scan (reading frame  $t$  states) while also performing reads, inserts, and updates of frame  $t + 1$  states. If it detects that the hash table has expanded, the scan must be restarted in order to ensure that all frame  $t$  states are read. It also provides a “prune in place” operation that deletes all states from frame  $t + 1$  whose likelihoods fall below the specified cutoff.

After each forward pass is completed, it performs an “advance” operation that moves information from

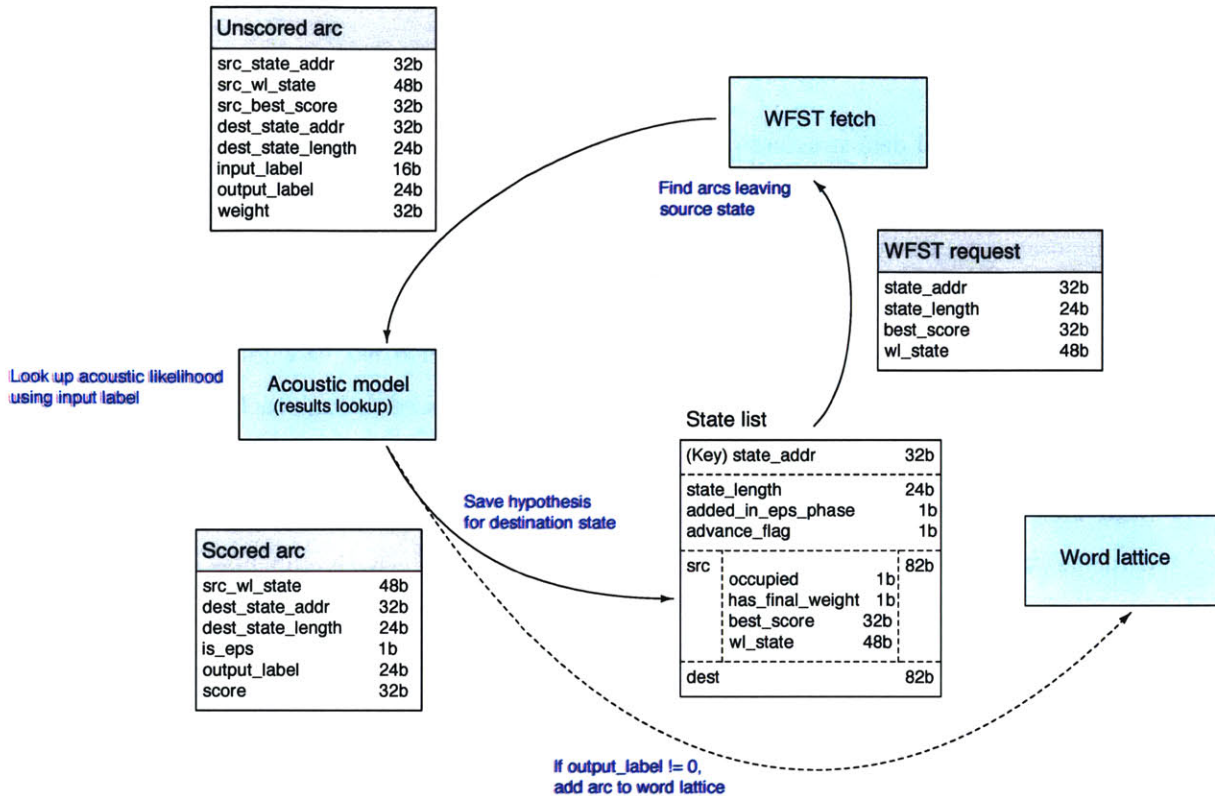


Figure 3-6: Data fields for state list entries and in-flight hypotheses. The architecture is designed to preserve memory locality.

frame  $t + 1$  fields to frame  $t$  fields, because  $t$  is being incremented. States that were present in frame  $t$  but not frame  $t + 1$  (i.e., those that are no longer active in the search) are deleted.

The state list is sized to trade between area and accuracy (high capacity is needed for larger recognition tasks). Our implementation has a capacity of 8192 states (223 bits per entry, or 1.8 Mb total). Merging the two frames and using single-ported hash tables allowed us to double capacity relative to [67] while using 11% less area.

### 3.2.3 WFST

The WFST encapsulates time-varying aspects of the HMM and is queried for possible transitions from each of its states. In speech recognizers supported by our architecture, the WFST is a composition of four training data sources (*HCLG*):

- *H*: The subphonetic model (different sounds produced over time within each phoneme).
- *C*: The context dependency (different ways of realizing each phoneme).

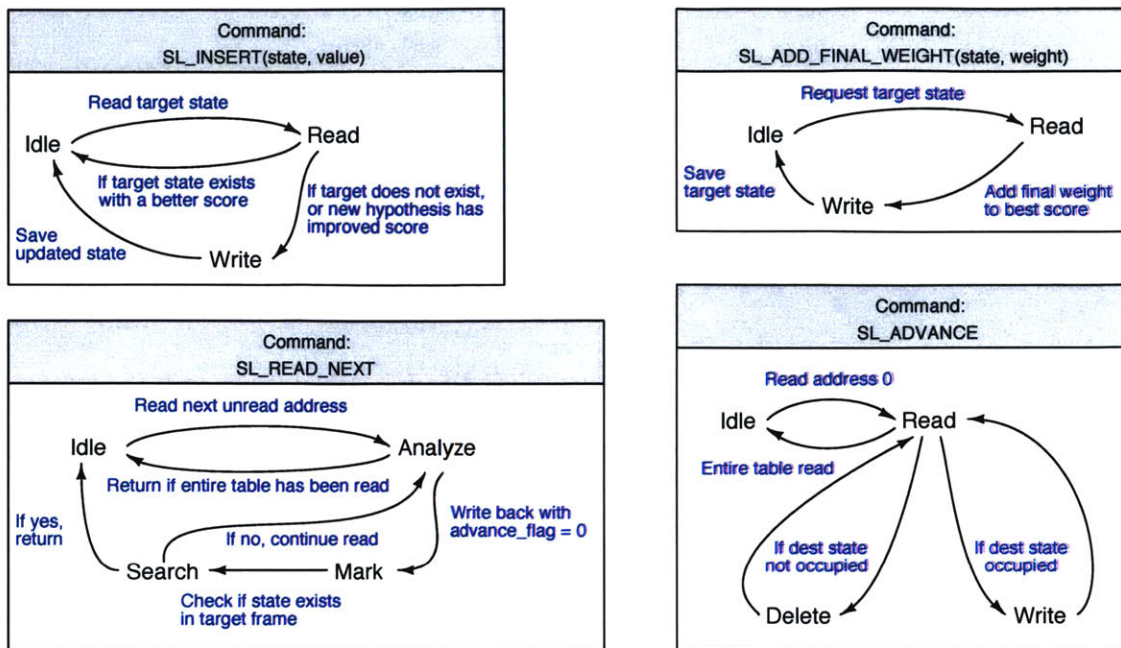


Figure 3-7: Guide to common state list operations, building on functionality provided by the hash table.

- *L*: The lexicon (phoneme sequences that make up each word).
- *G*: The language model (possible transitions between words).

There are no simple rules for computing the size of the WFST. The number of N-grams in the language model is a better predictor of the WFST size than the vocabulary. A 1,000-word recognizer with an unpruned trigram language model could have a larger WFST than a 100,000-word recognizer with a pruned bigram.

Viterbi search initialization, forward pass, final weight update, and backtrace all require access to the WFST. To simplify interfaces, the WFST is treated as a read-only key/value store as shown in Figure 3-8.

Each request to the WFST fetch unit includes the byte address and length (number of bytes) of the state to be read. This is treated as a monolithic value, even if not all of the information is needed. Reads executed during the forward pass include additional metadata:

- Likelihood of the hypothesis corresponding to the state being read
- Word lattice state ID associated with the hypothesis

The WFST fetch unit is shown in Figure 3-9. Compression and caching techniques described in this section reduce the memory bandwidth of the millions of read queries generated while decoding each utterance. The WFST is stored in external memory in compressed form. The WFST fetch circuit attempts to read each queried state from the cache, which stores individual states in the same format. Cache hits and misses both rely on the same logic to unpack the data. Each arc is then associated with the request metadata:

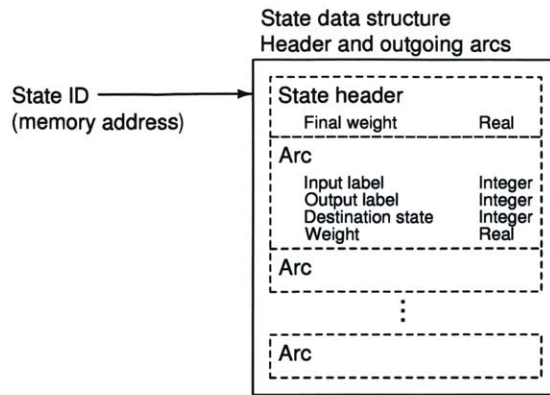


Figure 3-8: WFST data structure: a key/value store where keys are state IDs, and values contain all outgoing arcs from each state.

- The arc weight is added to the request likelihood.
- If the arc has an  $\epsilon$  output label, the request's word lattice state ID is preserved. If the arc has a non- $\epsilon$  output label, the word lattice state ID is set to (destination state, current frame).

Queries are processed in the order they are received.<sup>4</sup>

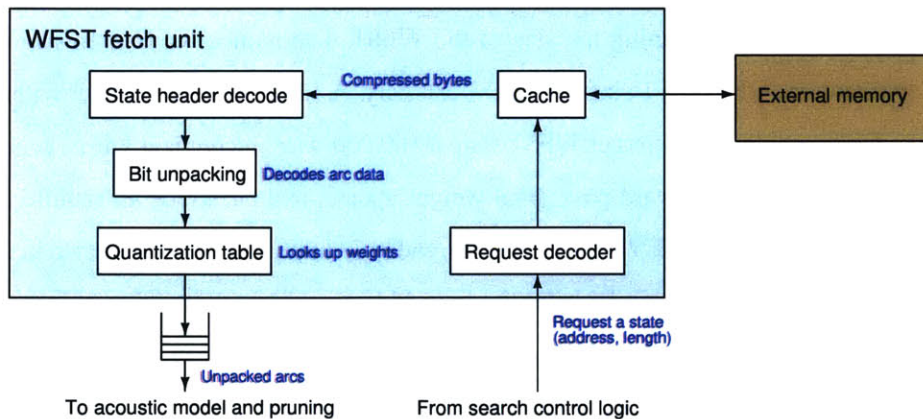


Figure 3-9: WFST fetch architecture with caching and model compression.

<sup>4</sup>Memory access patterns could be optimized by reordering requests. However, it is difficult to come up with a reordering scheme without first selecting a memory technology to optimize for. This design is memory-agnostic.

## WFST caching

While the WFST is a large and sparsely accessed structure, it is now a significant contributor to memory bandwidth given the acoustic model changes of Chapter 2.<sup>5</sup> We demonstrated in [67] that caching could be used to reduce page access penalties even though the working set (the set of active states that persist over time) was usually too large to achieve a high hit rate. As we discovered more recently, the working set size is strongly dependent on the specific models and pruning configuration used. Some ASR tasks simply don't need a large search space for acceptable WER. Caching improves scalability by offering larger gains on easier tasks. It's not a bug, it's a feature.

Compression also makes the cache look larger, relative to the information it is storing. To take advantage of this, we need to acknowledge and embrace the main challenge of WFST caching: we are caching objects of varying size, because different states have different out-degree. It makes no sense to store one (fixed-size) arc per cache line, and evict them independently, when we know that expanding a state always requires reading all of its arcs.<sup>6</sup> Hence our cache stores each state as a variable length sequence of bytes. The architecture is shown in Figure 3-10.

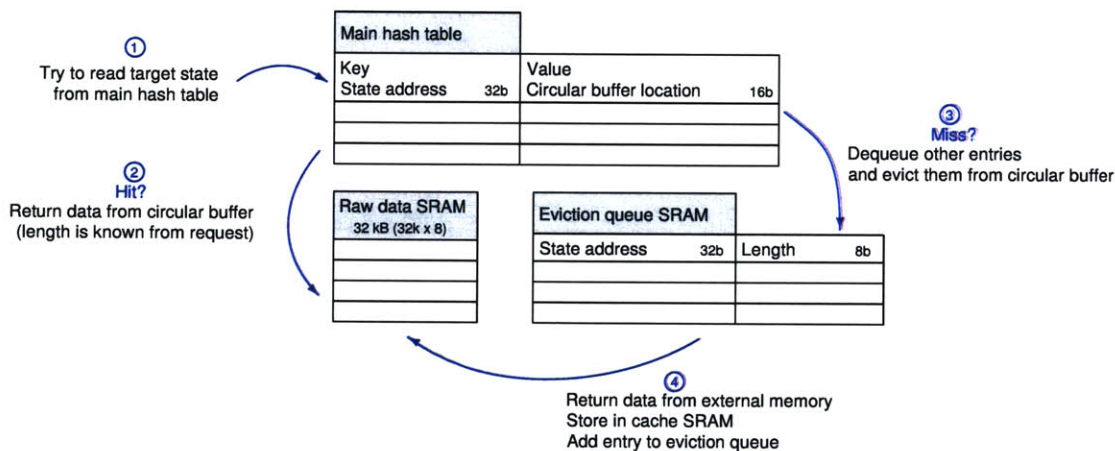


Figure 3-10: WFST cache architecture, including eviction queue and hash table so that variable-length objects can be stored in a circular buffer.

This cache implements a “least recently missed” eviction strategy, not because it is an effective eviction strategy, but because it allows variable-length objects to be stored in a circular buffer. Compressed state data is stored in a 32k×8 SRAM. A hash table keeps track of the location of each state: the key is a 32-bit state

<sup>5</sup>Performance engineering is like a game of whack-a-mole: once you've done a good job eliminating one bottleneck, another one pops up.

<sup>6</sup>You are welcome to design a system that can request only certain arcs, or only the state header, but your head is likely to explode. A compromise is suggested in Section 7.2.

ID, and the value is a 15-bit byte address into the circular buffer. After a cache miss, the requested state is read from external memory and appended to the circular buffer. The ID and length of the state are appended to an eviction queue (also a circular buffer) so that we know what data should be removed if space needs to be cleared for a future miss. The maximum cacheable state length is configurable, with a limit of 255 bytes. There is no minimum length, but the hash table and eviction queue have a capacity of 4096; if the average length of cached states is less than 8 bytes, the data buffer won't be fully utilized.<sup>7</sup>

Hit rates for this cache can significantly exceed those seen in [67]. Figure 3-11 shows the hit rate and WFST-related memory bandwidth for decoding a typical utterance with different beam widths. The maximum cacheable state length was 50 bytes. Hit rates are around 80% when decoding with a narrow beam, and decline gradually as beam width is increased. Figure 3-12 shows the effect of changing the maximum cacheable state length while holding beam width constant. This runtime parameter can be tuned to minimize aggregate memory bandwidth or to minimize page access penalties.

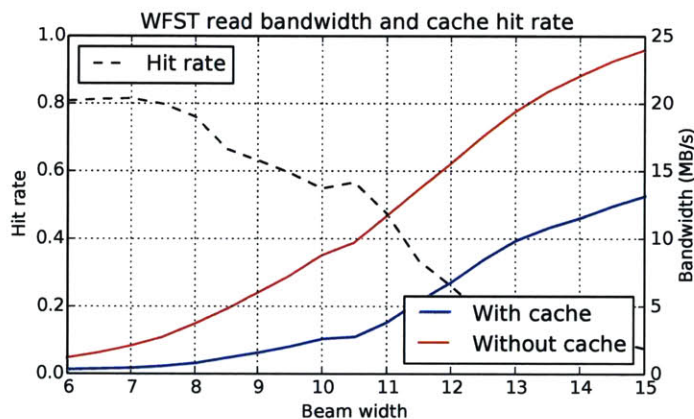


Figure 3-11: WFST cache performance with varying beam width. A wider beam increases the working set size.

Despite the use of a circular buffer, the bookkeeping adds significant overhead. In this design, the raw data buffer makes up 42% of SRAM bits. However, we were unable to devise a strategy with better performance for the same total area.

### WFST compression

A compressed WFST representation requires extra on-chip logic to decode, but directly impacts bandwidth—a straightforward case given that we are optimizing for system power rather than chip power. The compres-

<sup>7</sup>In our nutrition recognizer WFST, the average length across all states is 14.05 bytes and the median is 10 bytes. 3% of states have a length of less than 8 bytes, so it's unlikely that this limitation will be noticeable.



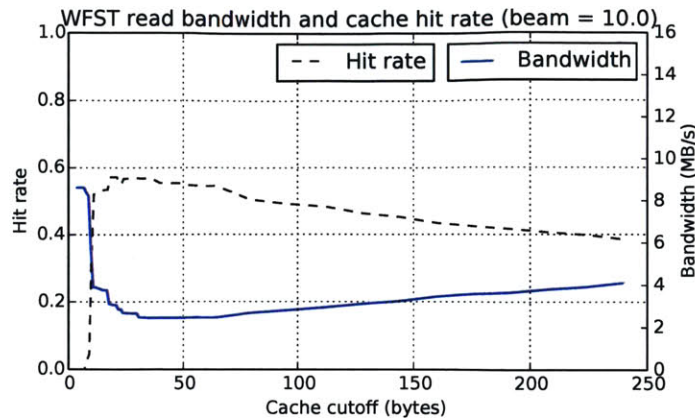


Figure 3-12: WFST cache performance can be optimized by setting a maximum cacheable state length which depends on the task and beam width.

sion works hand-in-hand with caching to reduce memory bandwidth.

Our WFST encoding is based on [33]. It preserves graph structure but quantizes weights (arc weights and state final weights) because high precision is unnecessary. Each state is stored as an 8-bit header followed by a sequence of arcs. The data structure is illustrated in Figures 3-13 and 3-14. Many states in an *HCLG* WFST exhibit some of the following properties:

- Self-loops, for which we don't need to store the destination state.
- All outgoing arcs have  $\epsilon$  output labels, so no output labels need to be stored.
- Outgoing arcs lead to nearby states; the destination state address can be stored in relative form using fewer bits than the absolute addresses.
- The input labels of outgoing arcs are in a narrow range (e.g., sequential) and can be stored in relative form with a per-state offset.

The flags in the state header control these selectively stored fields.

States are packed byte-aligned so that the byte address of a state (relative to the start of storage) can be used as a unique ID. Because the exact address of each state is initially unknown, we perform an iterative repacking process. State addresses are initialized under the assumption that each state takes up 8 bytes per outgoing arc. All of the states are then packed into the bit fields of Figure 3-14 and concatenated into a byte array. This allows us to get a better estimate of the packed address of each state, although it invalidates existing addresses. The last two steps are repeated until there are no further changes, and the destination state address of each arc is consistent with the byte offset of that state.

Table 3.1 shows the size reductions achieved for several recognizers, showing that this special-purpose

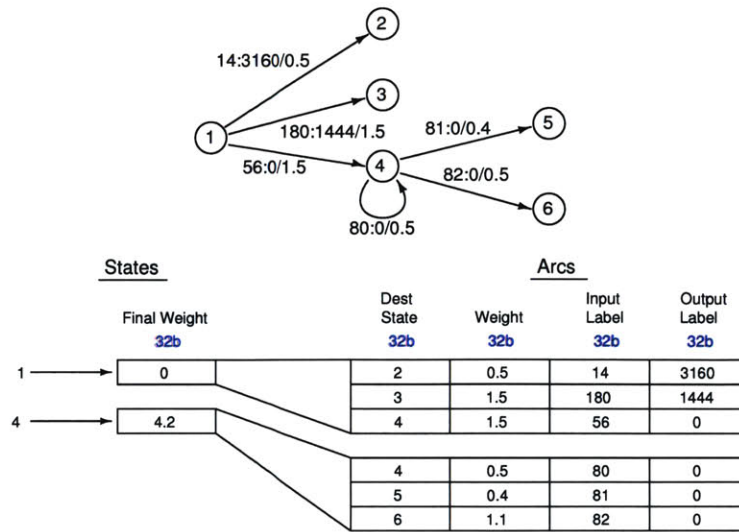


Figure 3-13: Example WFST, along with the OpenFST representation (totaling 832 bits) for states 1 and 4.

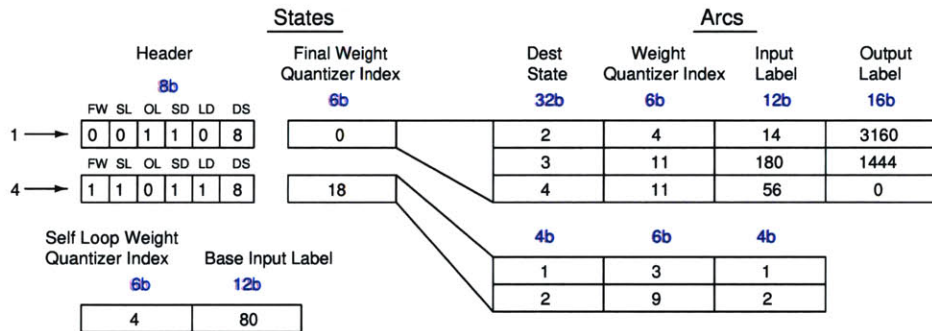


Figure 3-14: Compressed format (272 bits) for the states in Figure 3-13. An 8-bit header configures the storage format for each state.

encoding is about as effective as `gzip` on WFSTs. We do not achieve compression ratios as high as [33], but we think this is simply because the WFSTs in this study are larger. Bit widths scale with the  $\log_2$  of the values being stored, including labels and state addresses.

Task	Num. states	Num. arcs	OpenFST size	Compressed size	Ratio (vs. <code>gzip</code> )
TIDIGITS	104	228	4.9 kB	1.8 kB	0.366 (0.378)
Jupiter	1.0M	2.5M	51.6 MB	14.4 MB	0.279 (0.316)
Nutrition	1.3M	3.1M	66.4 MB	19.3 MB	0.291 (0.308)
WSJ-5k	6.1M	15.1M	316 MB	95 MB	0.302 (0.319)
WSJ-dev93	9.5M	22.9M	482 MB	135 MB	0.281 (0.349)

Table 3.1: Compression ratios for *HCLG* WFST models.

The total area required for WFST fetch (including decompression and caching) is 75k gates and 757 kb of SRAM.

### 3.2.4 Beam width control

The purpose of pruning is to decide which hypotheses should be saved for further consideration. If all valid hypotheses were saved, the search space would grow exponentially over time and eventually include all of the states in the WFST. ASR systems keep runtime in check by using a “beam search” variant of the Viterbi algorithm, pruning unlikely hypotheses during each forward pass [39]. More aggressive pruning reduces runtime but increases WER. The pruning can be relative (based on likelihood) or absolute (based on a maximum number of hypotheses). The desired number of hypotheses varies naturally over time along with ambiguity; for example, there is a high branching factor in the search near the beginning of a word. Figure 3-15 shows the number of hypotheses stored per frame when decoding an utterance with relative pruning.

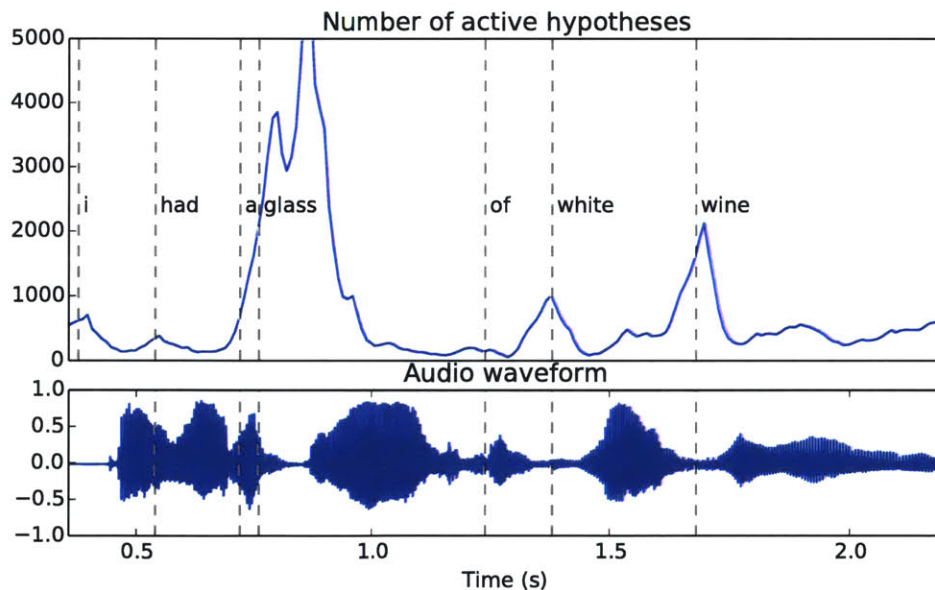


Figure 3-15: The changing level of ambiguity in the speech signal causes search workload to vary over time.

Hardware implementations need to use a combination of relative and absolute pruning because:

- There is a hard limit to the number of hypotheses that can be stored on-chip. Using relative pruning only, it would be difficult to prevent occasional overflows.<sup>8</sup>
- Using absolute pruning only would be inefficient because it would force the recognizer to store and

<sup>8</sup>It is possible to use the off-chip memory to handle such overflows [17], but this adds complexity and increases memory bandwidth; we decided that the costs of this feature would outweigh the benefits.

expand irrelevant hypotheses in order to meet its quota. Most of the time, a few hundred hypotheses are sufficient.

In [67], we regulated the beam width (relative pruning threshold) during each forward pass. Each WFST arc was annotated with the out-degree of its destination state, so we could predict the number of arcs to be expanded in the next frame. Beam width was then adjusted to maintain the proportion of arcs that were accepted. A similar strategy was employed in [31].

This work takes a different approach, adopting some of the ideas from Kaldi [65]. Intuitively, you would expect to get better performance by holding the beam width constant during each frame. This means that the likelihoods of all arcs are compared to the same accept/reject threshold. (Consider what would happen if the beam width fluctuated from arc to arc, and you ended up with the same number of hypotheses at the end. You would have accepted some lower-scoring hypotheses during positive excursions of beam width, taking the place of better hypotheses that were rejected during negative excursions.) Kaldi uses a constant beam width, but also has an absolute cutoff (typically 7k states). When this limit is exceeded, it sorts the likelihood scores to re-prune so the cutoff is matched exactly. It makes note of the resulting likelihood range and uses this to reduce the beam width for the next frame.

Our hardware implementation makes minor changes to the Kaldi approach. There is a configurable absolute cutoff, which must be lower than the maximum storage capacity of the memory (8192 states). Whenever this cutoff is reached during the forward pass, the beam width is reduced and the state list is immediately pruned in place using this reduced relative threshold. This means it is a “hard” limit on the number of states stored. This can result in overzealous pruning, but does not require sorting. In some cases, the state list can be pruned in place multiple times during a single frame.

Pruning in place is time consuming, so we also preemptively reduce the beam width when we expect to see many hypotheses. This is done with few hardware resources by storing a histogram of relative log-likelihoods. The cumulative distribution (CDF) of log-likelihoods is approximated by the cumulative sum of histogram bins. Linear interpolation of the CDF gives us an estimate of the beam width needed to hit a target state count, as illustrated in Figure 3-16. When a “soft” cutoff is exceeded, this estimate is used to initialize the beam width for the next frame. When the soft cutoff is not exceeded, the nominal beam width is used. This feature will only be engaged if the soft cutoff is lower than the hard cutoff. The soft cutoff logic has a post-synthesis area of 42k gates.

In short, we have introduced temporary “soft” and “hard” absolute pruning mechanisms to cope with limited on-chip memory. These complement the relative pruning that is always enabled, introducing occasional word errors when the nominal beam width is large enough to trigger additional pruning.

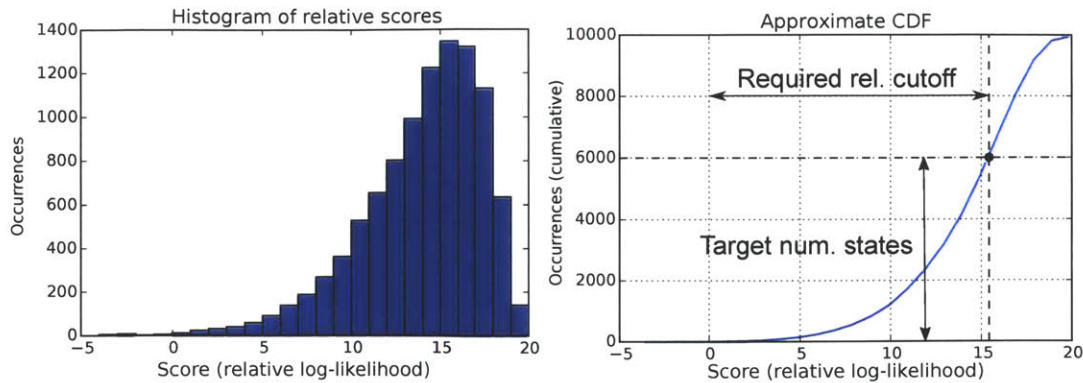


Figure 3-16: A histogram of relative log-likelihoods (left) is used to approximate the CDF (right) and predict an appropriate beam width for the desired state cutoff.

### 3.2.5 Word lattice

We designed an on-chip word lattice (WL) to mitigate the problem of memory writes caused by state list snapshots. NAND flash suffers from slow write performance (typically 5–10 MB/s per die) and limited endurance.

Viterbi search generates the equivalent of a state lattice, identifying possible trajectories through the states of the WFST (Figure 3-17). Because on-chip memory is only sufficient for two frames (one set of transitions), the state list must be saved to external memory after each forward pass; we call this a snapshot. The snapshot varies in size but is typically 10–50 kB, resulting in an average 1–5 MB/s of writes. At the end of decoding, all of the snapshots can be read in reverse order to perform a 1-best backtrace, resulting in an additional 1–5 MB/s amortized read bandwidth.

Most of the information in the state snapshots is not necessary for reconstructing word-level hypotheses.<sup>9</sup> A word lattice differs from a state lattice in that every arc has exactly one output (word) label. As implied by Figure 3-17, not all state transitions encountered by search generate words; 13% of the 3.1M arcs in our nutrition WFST have non- $\epsilon$  output labels. Each arc in the word lattice represents a chain of one or more arcs in the state lattice, the last of which has an output label. Because the word lattice is smaller than a state lattice covering the same search space, snapshots to the external memory are needed less frequently. This concept is illustrated in Figure 3-18.

Data structures used by the word lattice are shown in Figure 3-19. The word lattice consists of a hash table for storing states, and a flat array for storing arcs. States and arcs are stored separately because there can be multiple arcs leading into any state, representing alternative word hypotheses. Our hardware

<sup>9</sup>Although it is useful for acoustic model rescoring and reweighting.

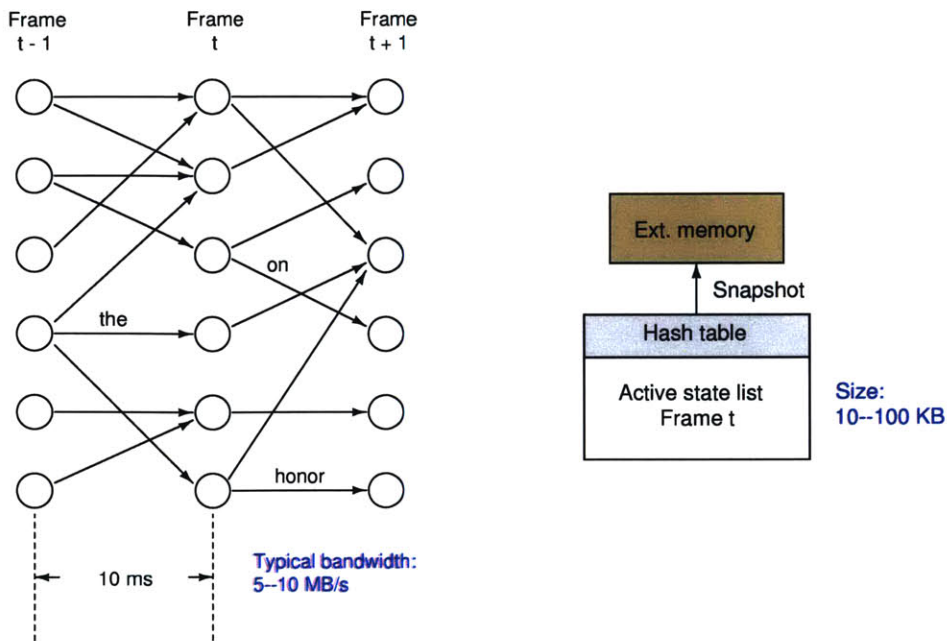


Figure 3-17: State lattice generated during decoding (left); state snapshot concept (right).

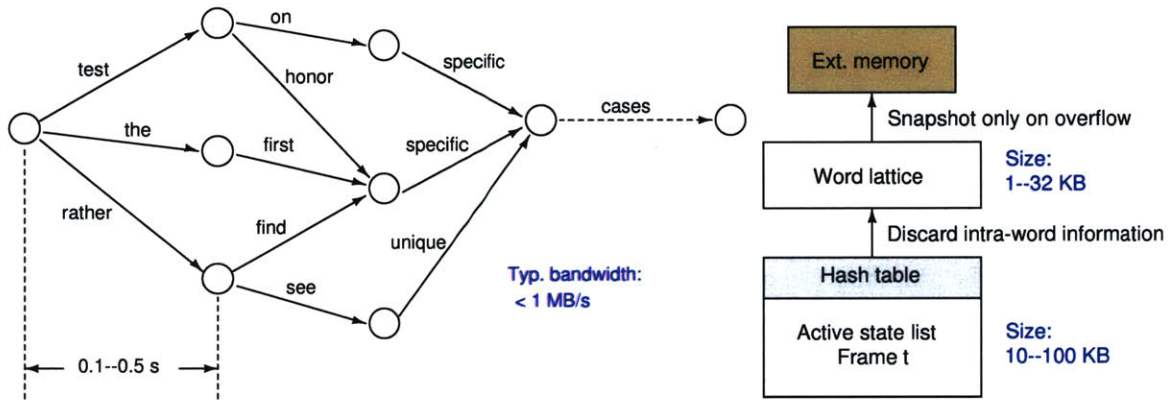


Figure 3-18: Word lattice example (left); word lattice snapshot concept (right).

implementation has a memory depth of 2048 for each table, with a limit of 1536 states (to keep hash table load below  $\frac{3}{4}$ ). If either limit is exceeded at any time, a snapshot is saved to external memory and the on-chip structures are cleared (except for metadata about the location of snapshots in the external memory). Correctness is maintained through up to 4096 snapshots per utterance (the capacity of our metadata storage).

The WFST state ID is not sufficient to uniquely identify a WL state because WFST states can be visited multiple times (for example, when a phrase matching a language model N-gram appears twice in an utterance). Loops can be prevented by using a tuple of (WFST state ID, number of words), but this isn't quite

enough. Throughout our search implementation, to prevent overflow of fixed-point values, the likelihood scores are stored relative to an offset (the best score in the current frame) that changes over time. To keep scores consistent for reverse pruning, we use (WFST state ID, frame index) tuples as WL state IDs. This results in more states being created, since there can be multiple instances of a (WFST state ID, number of words) pair across consecutive frames. The most similar structure we are aware of is the word trace mentioned in [75].

The word lattice must be coordinated with the state list. To insert an arc into the word lattice, we need to know the source and destination states. The destination state comes from a WFST arc, annotated with the current frame index. However, the source state is not necessarily the source of the same WFST arc; it is the destination of the previous WL arc, which may have been inserted many frames in the past. To keep track, we store a source WL state along with every hypothesis in the state list; this adds a significant area overhead (96 bits per entry).<sup>10</sup> This was yet another judgement call based on our desire to minimize off-chip bandwidth.

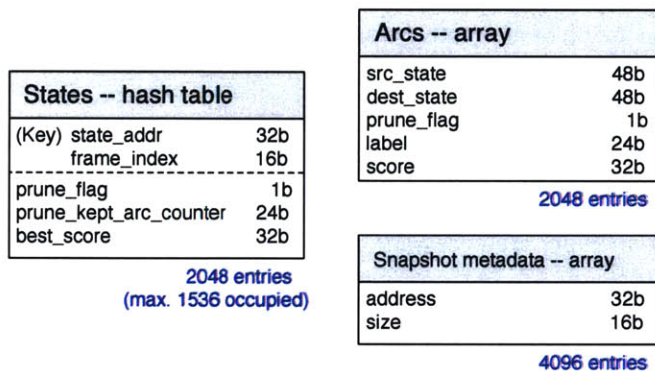


Figure 3-19: Data structures used in word lattice.

Algorithms used by the word lattice are summarized in Table 3.2; the non-trivial ADDWORD and FINISHPRUNE operations are detailed in Algorithms 1 and 2 respectively. The STARTPRUNE method is invoked before a forward pass. During the forward pass (both non- $\epsilon$  and  $\epsilon$  phases), the ADDWORD method is invoked whenever an arc with a word label is accepted. The KEEPSTATE method is invoked whenever any arc is accepted (whether or not it has a word label), so that the destination WL state and related arcs are preserved. At the end of the forward pass, the FINISHPRUNE method recovers space in the on-chip structures. Pruning is performed after every frame in order to minimize the frequency of snapshots. Pruning can preserve multiple arcs leading to a state, as long as their score is within a “lattice beam;” with a nonzero

<sup>10</sup>It’s necessary to store a separate 48-bit WL state for frames  $t$  and  $t + 1$  because the best-scoring arc leading to a state can come from different source states in different frames.

<b>Method name</b>	<b>Description</b>
STARTPRUNE	Sets a pruning flag on all states and arcs.
KEEPSTATE	Clears the pruning flag on a state.
ADDWORD	Inserts or updates a state; inserts an arc if the score falls within a lattice beam. See Algorithm 1.
FINISHPRUNE	Propagate pruning flags to preserve predecessors of active states. Deletes states and arcs still marked for pruning (i.e., no longer active). See Algorithm 2.
SENDSNAPSHOT	Serializes states and arcs and saves to external memory. Clears state and arc arrays; appends snapshot metadata entry.
LOADSNAPSHOT	Reads snapshot metadata; deserializes stream from external memory to populate state and arc arrays.

Table 3.2: Algorithms used in word lattice generation.

lattice beam, this gives us a graph of alternative word hypotheses rather than a chain. Duplicate arcs sharing the same word label are discarded.

The word lattice improves scalability. With a light workload, it is possible that the lattice will never exceed the capacity of 1536 states and 2048 arcs, because pruning is performed after each frame. In this case, no writes to external memory are necessary. Moderate workloads will generate enough word arcs to fill the on-chip memory and trigger snapshots occasionally. A heavy workload will trigger one or more snapshots per frame, resulting in bandwidth proportional to directly saving the state list.

Figure 3-20 shows the effect of the word lattice structure on off-chip memory writes. Heavy workloads force more frequent word lattice snapshots, but there is still an 8x bandwidth savings because not all arcs have word labels. Snapshots can be written to a circular buffer in the unused portion of the off-chip memory for wear-leveling.

The word lattice can perform a 1-best backtrace from any state, returning the sequence of word labels. The host can fetch the entire word lattice structure (either the on-chip portion, or the on-chip portion plus previous snapshots from external memory) in order to compute an N-best list or display the lattice. This backtrace can be performed in between frames in order to provide incremental recognition results (before a final backtrace at the end of the utterance).

Hardware implementation of the word lattice was complicated, and its area impact is significant (72k gates and 727 kb SRAM), but we believe the impact is worthwhile due to the 8x or larger reduction in off-chip memory writes.



**Algorithm 1** Insert a word arc into the lattice

---

```

procedure ADDWORD(states, arcs, w, ssrc, sdest, v)
  if LEN(arcs) = arc array capacity then SENDSNAPSHOT
  end if

  if ssrc not in states then                                ▷ Insert source state if it doesn't exist (i.e., previous snapshot)
    SCORE(states[ssrc]) ← 0
    PRUNEFLAG(states[ssrc]) ← false
  end if

  a ← false                                                ▷ a will be set to true if a new word arc has been accepted
  if sdest not in states then                                ▷ Insert destination state if it doesn't exist
    if LEN(states) = state table capacity then SENDSNAPSHOT
    end if
    SCORE(states[sdest]) ← v
    PRUNEFLAG(states[sdest]) ← true                          ▷ Prune flag will later be cleared if sdest persists in state lattice
    a ← true
  else                                                       ▷ If destination state exists, update score and check lattice beam
    if v < SCORE(states[sdest]) then
      SCORE(states[sdest]) ← v
    end if
    if v < SCORE(states[sdest]) + lattice beam then
      a ← true
    end if
  end if

  if a then                                                ▷ Arc leads to a new state or falls within lattice beam. Insert it.
    i ← LEN(arcs)
    RESIZE(arcs, i + 1)
    SRC(arcs[i]) ← ssrc
    DEST(arcs[i]) ← sdest
    LABEL(arcs[i]) ← w
    SCORE(arcs[i]) ← v
    PRUNEFLAG(arcs[i]) ← true
  end if
end procedure

```

---

---

**Algorithm 2** Remove pruned states/arcs from memory

---

**Require:** All states with an incoming arc inserted in frame  $t$  have PRUNEFLLAG = false

**Require:** All other states have PRUNEFLLAG = true

```

procedure FINISHPRUNE(states, arcs)
  for  $i \leftarrow \text{LEN}(\text{arcs}) - 1$  down to 0 do                                ▷ Propagate pruning flags backwards along arcs
     $s \leftarrow \text{DEST}(\text{arcs}[i])$ 
    if PRUNEFLLAG(states[ $s$ ]) and SCORE(arcs[ $i$ ]) < SCORE(states[ $s$ ]) + lattice beam then
      PRUNEFLLAG(arcs[ $i$ ])  $\leftarrow$  false
       $s \leftarrow \text{SRC}(\text{arcs}[i])$ 
      if  $s \neq \text{null}$  then
        PRUNEFLLAG(states[ $s$ ])  $\leftarrow$  false
      end if
    end if
  end for

  for all  $s$  in states do                                                    ▷ Remove any states that are still marked for pruning
    if PRUNEFLLAG(states[ $s$ ]) then
      DELETE(states[ $s$ ])
    end if
  end for

  Initialize  $s \leftarrow -1, n \leftarrow 0, i \leftarrow 0$ 
  while  $i < \text{LEN}(\text{arcs})$  do
    if PRUNEFLLAG(arcs[ $i$ ]) then
       $n \leftarrow n + 1$ 
      if  $i < s$  then  $s \leftarrow i$ 
      end if
    else
      if  $s < i$  then
        arcs[ $s$ ]  $\leftarrow$  arcs[ $i$ ]
         $s \leftarrow s + 1$ 
      end if
    end if
  end while
   $N \leftarrow \text{LEN}(\text{arcs}) - n$ 
  RESIZE(arcs,  $N$ )
end procedure

```

---

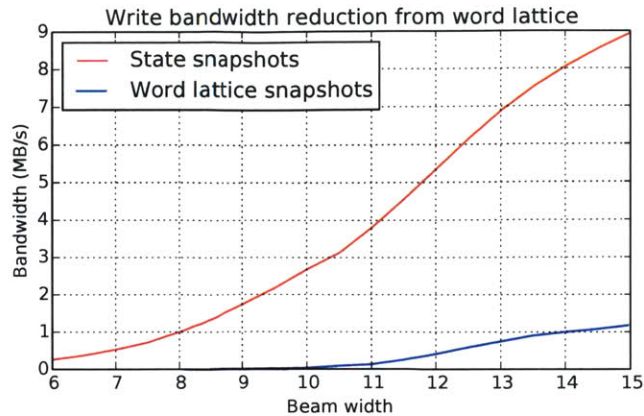


Figure 3-20: Write bandwidth comparison of state lattice and word lattice approaches. Time-averaged bandwidth was measured on one of the harder utterances in the nutrition test set.

### 3.3 Interactions

The search sub-system has FIFOs between pipeline stages to decouple their latency, but there is still the possibility of contention for external resources when the stages are not coordinated.

Sometimes the WFST fetch and word lattice will deadlock. If the word lattice overflows, it will request a memory write to save a snapshot, and cannot store any more word arcs until this write completes. If this happens while WFST fetch is streaming data from the external memory, and there is not enough slack in the pipeline, the WFST fetch memory read will stall. In this situation, we flush the read by ignoring all remaining WFST arcs, allowing the word lattice snapshot to proceed. We then re-fetch the same state from the WFST and ignore the arcs that have already been processed.

### 3.4 Performance and scalability

The search sub-system processes about one hypothesis every 40 cycles. The runtime of search depends strongly on the pruning parameters, primarily the beam width. Each ASR task comes with its own tradeoff between beam width and word error rate, which can be exploited to minimize search runtime. As with the acoustic model, energy consumption is slightly more than linear with respect to runtime. Memory bandwidth scales at around 7–8 bytes per hypothesis, or about 6 nJ to read from MLC flash; this is less than the 16 nJ/hypothesis cost of on-chip computation we attained in [67].<sup>11</sup>

Figure 3-21 shows the tradeoff between accuracy and beam width for each of our ASR tasks, evaluated

<sup>11</sup>See section 6.4 for estimates of the search efficiency of this architecture.

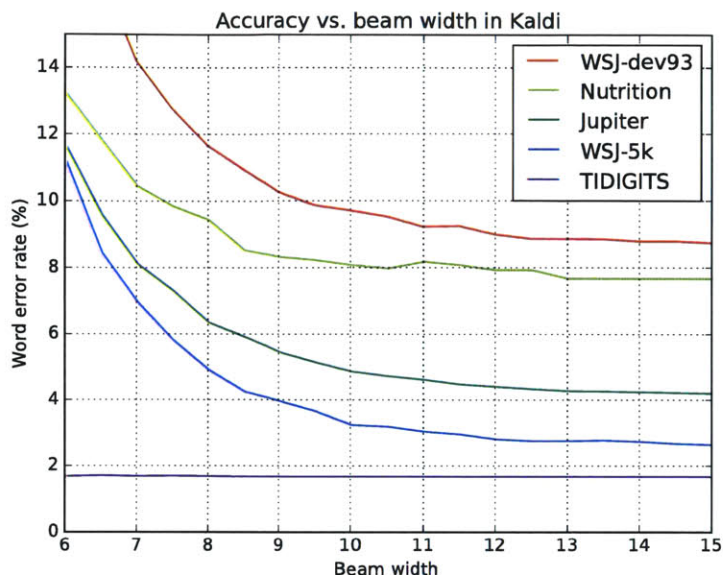


Figure 3-21: WER vs. beam width for each ASR task, evaluated in Kaldi.

using Kaldi.<sup>12</sup>

The scaling of search runtime (and hence energy) against beam width is plotted in Figure 3-22, sampling four utterances from the nutrition task. There is approximately a 10-fold range (e.g., 5M–50M cycles) across reasonable values of beam width. Remember that this pertains to the search sub-system, independently of the acoustic model. To run in real-time, the core clock frequency should exceed the sum of runtimes of search and acoustic model. (The 256x6 DNN that we used for the nutrition task takes 2.45M cycles per utterance second to run, so most of the time is spent on search. See Section 2.2.2 for a discussion of acoustic model scalability.)

Table 3.3 summarizes the search runtime on each task with the beam width set at a point of diminishing returns (5% relative WER degradation). Runtime includes the time used for word lattice pruning and snapshots, final weight updates, and backtraces. Search workload is uncorrelated with WFST size (excluding TIDIGITS); we suspect that the accuracy of the acoustic model is a confounding variable.<sup>13</sup>

In Figure 3-23, the decoding accuracy in terms of word error rate (WER) on the WSJ dev93 task is

<sup>12</sup>Performing this many tests on the hardware platform would be very time consuming, but since Kaldi has similar behavior (and can be run on a compute cluster) it is useful for evaluating these tradeoffs. Kaldi was run with a fixed acoustic weight of 0.09 to match the behavior of our hardware platform, which can't rescore lattices with different acoustic weights.

<sup>13</sup>It would be interesting to test on the same task with different language models to isolate the influence of WFST size on search workload. But even then, the different language models will result in different WER, so it would be difficult to extract broad conclusions. This is best viewed as an area for task-specific design exploration.

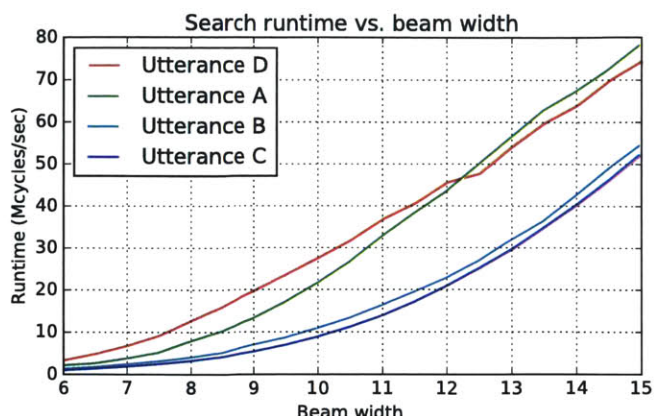


Figure 3-22: Runtime scales with beam width and varies between utterances.

Task	Vocab.	WFST states	Beam	WER	Workload	Bandwidth	Runtime
TIDIGITS	11	104	6.0	1.65%	15.5k	0.00 MB/s	0.64M
Jupiter	2k	1.0M	12.0	4.38%	301k	1.64 MB/s	11.8M
Nutrition	7k	1.3M	12.0	8.51%	844k	6.54 MB/s	36.1M
WSJ (eval92-5k)	5k	6.1M	12.5	2.91%	399k	3.01 MB/s	16.7M
WSJ (dev93)	145k	9.5M	11.0	9.04%	409k	3.26 MB/s	16.4M

Table 3.3: Search workload for each ASR task. Workload is specified in hypotheses, and runtime in clock cycles, per second of utterance time.

plotted along with decoding speed in terms of real-time factor (RTF). This provides a comparison between our platform (with three different pruning cutoff configurations) and Kaldi.<sup>14</sup> At large beam widths the temporary reductions imposed by the cutoff logic occur more frequently, so the cutoff configuration is more important. Generally, lowering the cutoffs will speed up decoding. A soft cutoff at 2k states (compared to the default of 5k) decreases runtime by up to 37%, almost as much as a hard cutoff, without degrading accuracy. Overall, the word error rate on this task (with a recognizer vocabulary of 145k words) is similar to Kaldi when the same models are used.

The memory bandwidth requirements of this sub-system and its predecessor (under 10 MB/s) are lower than those reported in other work (100+ MB/s). We attribute much of this discrepancy to our use of WFSTs (a single, optimized graph structure) rather than separately modeling intra-word and inter-word transitions. Another reason is our broader use of on-chip memory for search purposes; the active state list is frequently

<sup>14</sup>Don't scrutinize the Kaldi RTF numbers. They were measured by decoding in batch mode with a single job (multi-process pipeline) on an Intel Xeon E5-1670v2 (4 cores at 3.7 GHz). The purpose of including these numbers is to show that an un-optimized research decoder on a modern x86 machine has comparable search throughput to our design (when running at 80 MHz), though it is much faster at acoustic model evaluation.

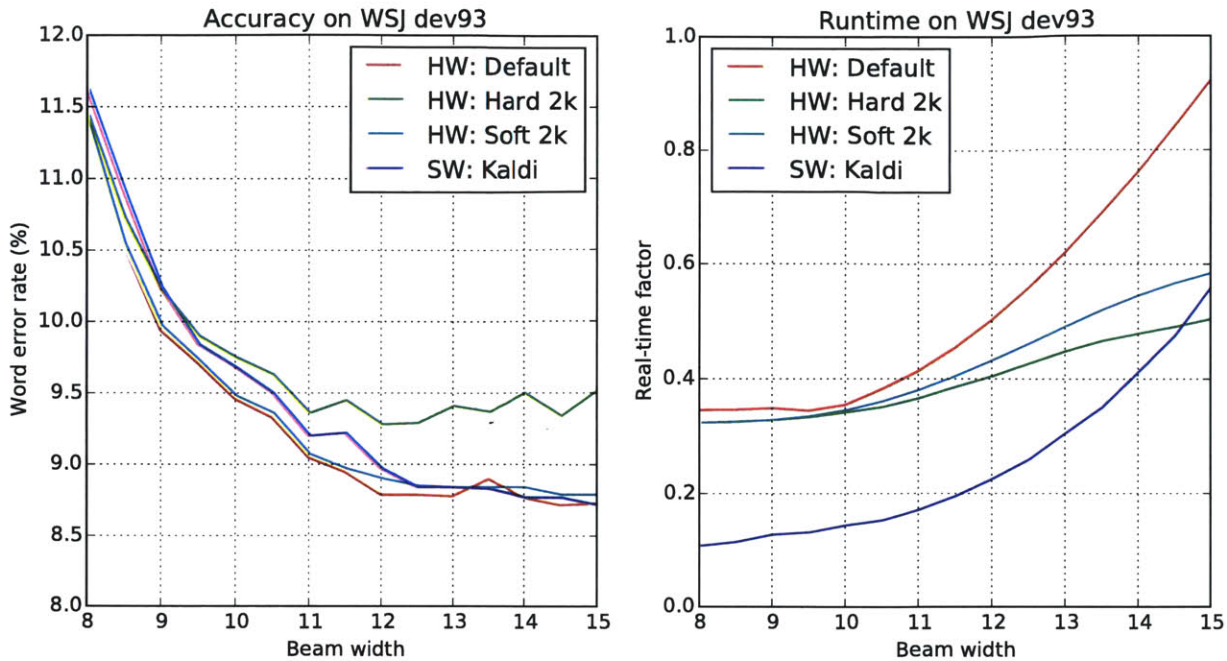


Figure 3-23: WER (left) and RTF (right) for search using varying soft and hard cutoffs, compared to Kaldi. (Hardware recognizer is run at 80 MHz.)

accessed, but our design limits it to on-chip memory. Finally, we may be performing more aggressive pruning while still obtaining reasonable accuracy, thanks to the quality of acoustic models derived from Kaldi. The caching, WFST compression, and word lattice techniques described in this chapter provide further bandwidth reductions from this baseline.

### 3.5 Summary

This chapter developed enhancements to the Viterbi search architecture of [67] that allow us to approach the accuracy of the reference Kaldi recognizer despite limitations on local memory size and external memory bandwidth. Local memory was used to store only the two most recent frames of the state lattice created during search. Data from older frames were filtered to generate a word lattice on-chip, reducing the amount of data that must be written to external memory. WFST compression and caching techniques reduced the volume of memory reads incurred by WFST queries. The resulting architecture evaluates each hypothesis in about 40 clock cycles, performing real-time search for typical ASR tasks at a frequency of 10–40 MHz (250k–1M hypotheses per second). External memory bandwidth was reduced to about 8 bytes per hypothesis.

## Chapter 4

# Voice activity detection

Most people don't talk constantly. Even an enthusiastic ASR engineer has to cope with the reality that his product will not be used at all times. Additional engineering is needed to direct the recognizer's attention to the relevant portions of an audio stream, with a realistic duty cycle of 5% or less. Typical approaches include voice activity detection (VAD), which detects speech; keyword spotting (KWS), which detects a specific word or phrase; and speaker ID, which detects the voice of a particular person. We explored the use of VAD in an embedded speech interface.

### 4.1 Accuracy influences power consumption

The preceding chapters have, in the interest of system-level simplicity and power consumption, de-emphasized the internal area and power of ASR circuitry relative to the burdens placed on off-chip memory. A similar tradeoff (or paradox) occurs with VADs: if what you care about is power consumption, then the power consumption of the VAD should be the last thing on your mind. Accuracy is far more important.

It's important to put the power consumption of different components in perspective. VAD power draw varies based on algorithm and circuit implementation details, but is generally low (on the order of 100  $\mu\text{W}$  or less, for special-purpose hardware). When the VAD detects speech, it will power up a downstream system that is much more power-hungry (it probably includes ASR, a microcontroller, or other application processor)—even if carefully optimized, probably several mW. The time-averaged system power is

$$P_{\text{avg}} = p_{\text{VAD}} + [(1 - p_M)D + p_{FA}(1 - D)] p_{\text{downstream}}$$

where  $D$  is the duty cycle of speech. The coefficient  $(1 - p_M)D + p_{FA}(1 - D)$  reflects how often the

downstream system is enabled—a duty cycle that will be higher than  $D$ , if the false-alarm probability  $p_{FA}$  of the VAD is significant. Differences in the contribution of the downstream system can far exceed the differences in power consumption between different VAD implementations.

A variety of research groups have presented hardware VADs, some with impressively low power consumption [71, 4], but have not adequately addressed their accuracy in low-SNR and nonstationary noise conditions. To investigate, we constructed fixed-point models and designed a chip allowing direct comparison of three VAD algorithms. These algorithms cover the spectrum from simple, elegant, and inaccurate to complex, messy, and accurate. Each of the algorithms and their circuit implementation are discussed below. We conclude by comparing the receiver operating characteristics (ROCs) and modeled system power consumption of each algorithm in different noise scenarios.

## 4.2 Algorithms

Decades of research have produced a large class of algorithms for VAD. Many algorithms are variants of the likelihood ratio test (LRT) popularized by [79]: different ways of estimating the noise spectra [24, 51], weighting different frequencies [82], temporal context [70], adaptive thresholds [23], post-filtering decisions, and more. In our experience these techniques can be a bit fragile, especially if they create positive feedback loops (i.e., updating speech/noise statistics based on decisions), and do not deliver consistently good performance across tasks. We describe three other VAD algorithms, all variants on previously published work, which we used to characterize the tradeoffs in VAD circuit design.

### 4.2.1 Energy-based (EB)

Thresholding a signal based on its short-term energy is a workable VAD approach in low-noise conditions. The energy-based algorithm we implemented is a variation on that idea, except it tracks the changes in energy over time that distinguish speech from many non-speech sounds [83].

These energy differences are weighted by an estimate of SNR, reflecting the decrease in confidence when SNR is low.<sup>1</sup> We compute the SNR using a gross simplification of the minimum statistics approach

---

<sup>1</sup>Use of the square root to compress energy differences was suggested in an unpublished successor to [83].



[51]: the noise energy is estimated as the minimum frame energy over the last 100 frames, plus 3 dB.

$$E_t = 10 \log_{10} \sum_{n=0}^{N-1} x^2[n]$$

$$D_t = \sqrt{|E_t - E_{t-1}| \text{SNR}_{\text{post}}}$$

$$\text{SNR}_{\text{post}} = \max(0, E_t - (\min_{t'=0}^T E_{t-t'} + 3))$$

Figure 4-1 shows these parameters and the resulting decision output for an example waveform. The energy difference timeseries (bottom plot) is itself quite noisy. To obtain clean speech/non-speech decisions, the threshold (dashed line) is applied to a moving average of the scaled energy differences over the previous 21 frames. Estimating a noise floor and SNR (middle plot) allows the VAD to reject the noise burst at the beginning of the waveform.

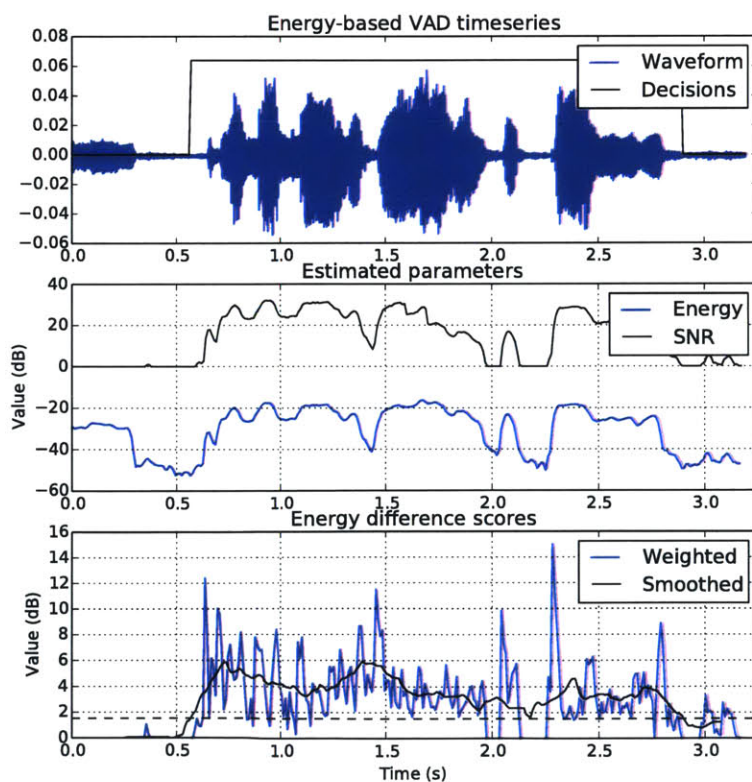


Figure 4-1: Energy-based VAD operation including waveform (top), estimated frame-level energy and SNR (middle), and energy difference scores (bottom).

### 4.2.2 Harmonicity (HM)

A conceptually simple but relatively robust VAD technique is to threshold audio frames based on their harmonicity, or the amplitude ratio between periodic and non-periodic components of the signal. Not all speech is periodic: unvoiced phonemes (some fricatives, affricates and stops) consist of turbulence and silence. However, every utterance (except perhaps whispering) contains a mixture of unvoiced and voiced sounds.

Boersma [10] provided the mathematical underpinnings for accurate estimates of harmonicity. The periodic nature of a signal is visible in its autocorrelation; however, the exact amplitude and period are obscured by windowing. Dividing out the autocorrelation of the window gives us a normalized autocorrelation that allows accurate pitch extraction. Tweaks such as cepstral filtering (as in [19]) can improve performance, but for simplicity we limit the pitch frequency to the range of 80–400 Hz. The harmonicity is expressed on a decibel scale, with 0 dB indicating equally strong periodic and non-periodic components:

$$r'_{xx}(t) = \frac{r_{xx}(t)}{r_{ww}(t)}$$
$$H = 10 \log_{10} \max_{T_1 < t < T_2} \frac{r'_{xx}[t]}{r'_{xx}[0] - r'_{xx}[t]}$$

In our experience a threshold of 6–8 dB detects speech reliably. To account for unvoiced frames at the beginning and end of speech segments, we extend the regions of high harmonicity forward by 350 ms and backward by 150 ms. Figure 4-2 illustrates this behavior. The per-frame harmonicity crosses the threshold several times during the speech segment; applying the “window maximum” gives us a clean segmentation.

### 4.2.3 Modulation frequencies (MF)

The energy-based and harmonicity techniques are, in a sense, rule-based. We can't design rules that cover every possible scenario, and the limitations of these simple schemes are obvious. Stuttering sounds such as railroad noise will fool the energy-based algorithm; beeping sounds and music will fool the harmonicity algorithm. These challenges motivate a supervised learning approach, which requires choosing a representation and a classifier.

Representations that work well for ASR are not ideal for VAD. MFCCs capture short-term spectral characteristics of a sound which can guide HMM search and, over time, discriminate between words. But a VAD doesn't have knowledge of temporal constraints; most people would find it difficult to make speech/non-speech decisions based on a repeated clip of 25 ms. Hence the features need to contain some temporal

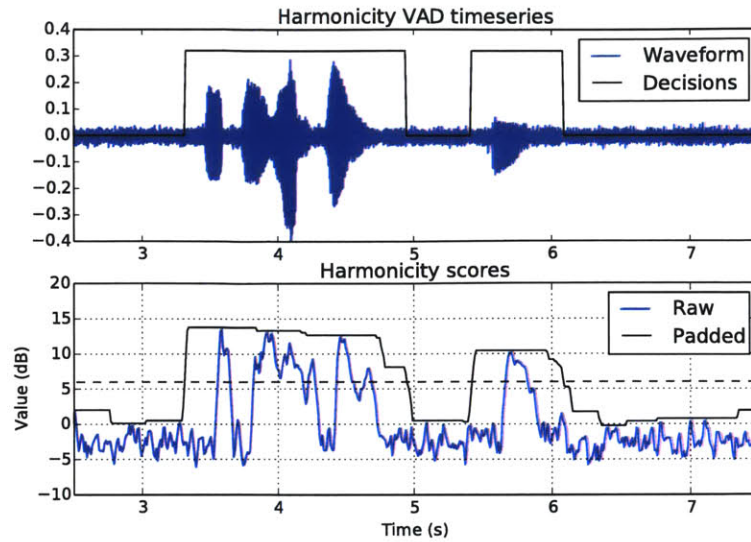


Figure 4-2: Harmonicity VAD operation including waveform (top) and harmonicity scores (bottom).

context. The brute force solution would be to concatenate several MFCCs, as in [14]. Another representation that captures temporal patterns is the modulation frequency (MF) [89], illustrated in Figure 4-3. MF features are calculated in two stages. First, we compute the energy in each frequency band using a short-term FFT. Second, we compute the power spectrum of the amplitude modulation within each band with a set of long-term FFTs (one per band). The first stage is run at a typical frame rate of 100 fps, while the second stage can be run at a reduced rate.

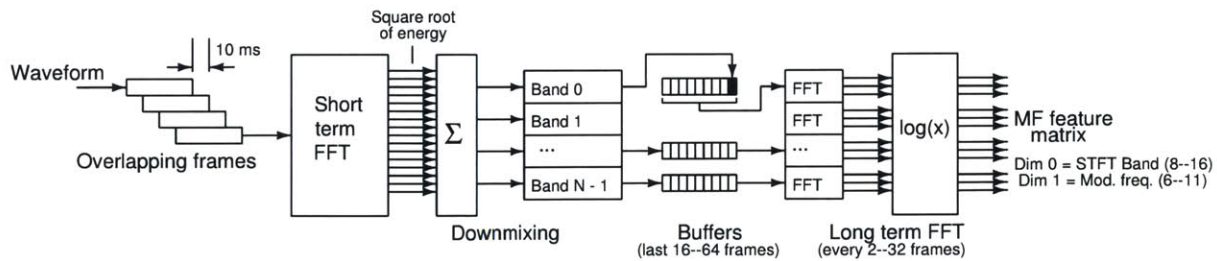


Figure 4-3: Modulation frequency (MF) feature extraction. The use of both short-term and long-term FFTs allows MF features to capture temporal patterns that are unique to speech.

MF features are amazing. The difference between speech and non-speech segments in an MF spectrogram is visible to the human eye even at 0 dB SNR (Figure 4-4), and more subtle distinctions can be captured by simple classifiers. In [19] (on which this work is based), MF features were fed into a group of SVM classifiers whose outputs were fused. We compared GMM, SVM, and NN classifiers on MF features and observed that NNs performed nearly as well with many fewer parameters. The NN classifier is

especially attractive because (like the GMM) the parameters could all be stored in an on-chip memory (e.g., 32 kB), which was not possible with SVMs because of the large numbers of support vectors. The resulting networks are much smaller than those used for speech recognition: 2–4 layers of 32–64 nodes are sufficient.

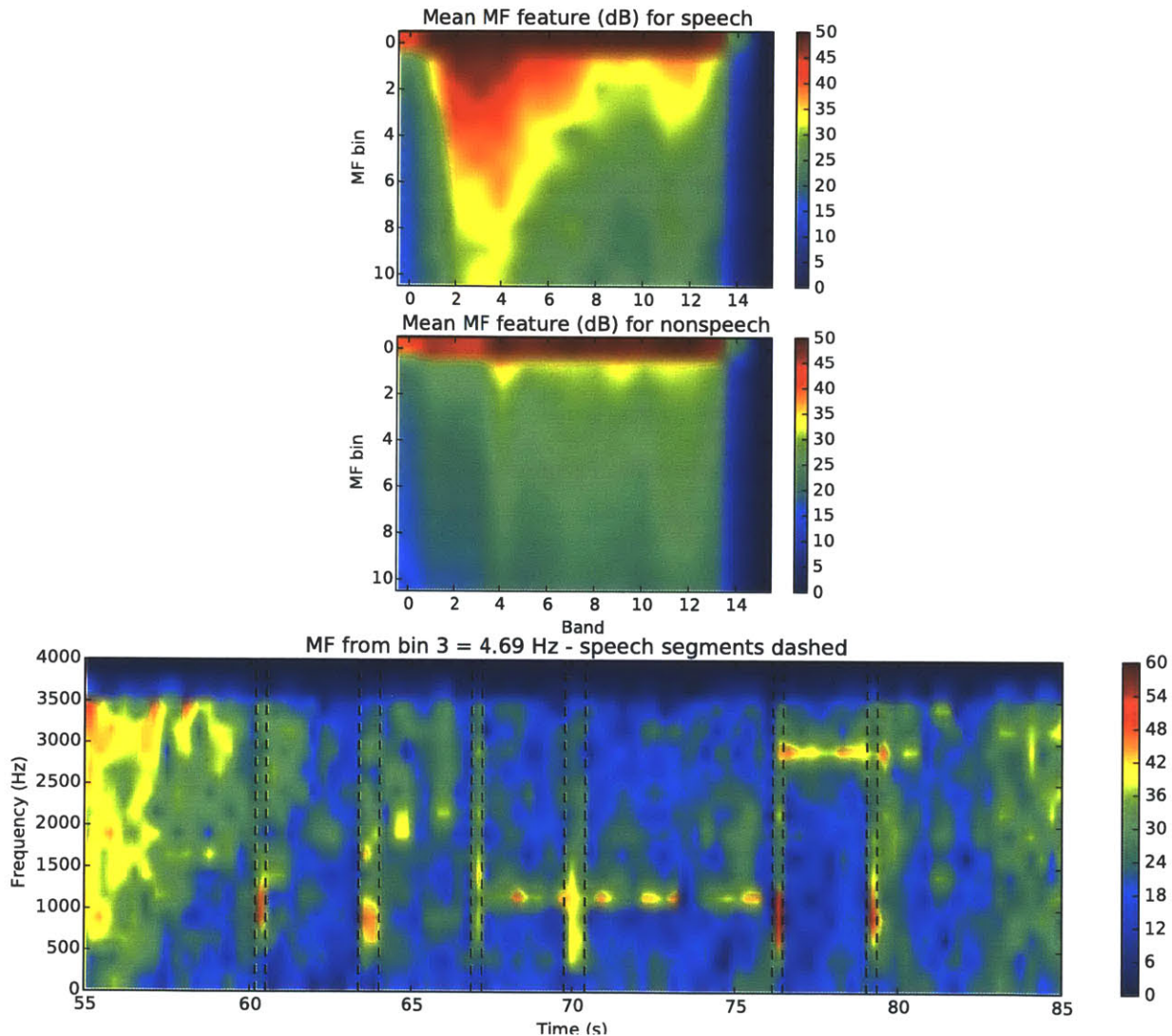


Figure 4-4: Example of mean MF matrix for speech and non-speech (top), and 4 Hz MF spectra (bottom) with speech segments indicated by dashed lines, at 0 dB SNR.

We believe the MF algorithm is useful for VAD in computationally constrained scenarios, even in difficult noise conditions. More robust supervised VADs have been implemented for certain tasks (such as the DARPA RATS program [76]), but these schemes risk diminishing returns in performance while requiring much larger computational resources.

## 4.3 Architecture

### 4.3.1 Top level

Our VAD circuit (implemented as a standalone chip, as well as the combined ASR/VAD chip of Chapter 6) includes the EB, HM and MF algorithms connected to a control module and external interfaces. The control module interfaces to the host over SPI, performs buffering and downsampling of input audio, and allows any subset of the three algorithms to be run. It interfaces to each stage through FIFOs for commands, scores and decisions, as well as an SRAM storing the downsampled audio (which is buffered separately from the input audio).

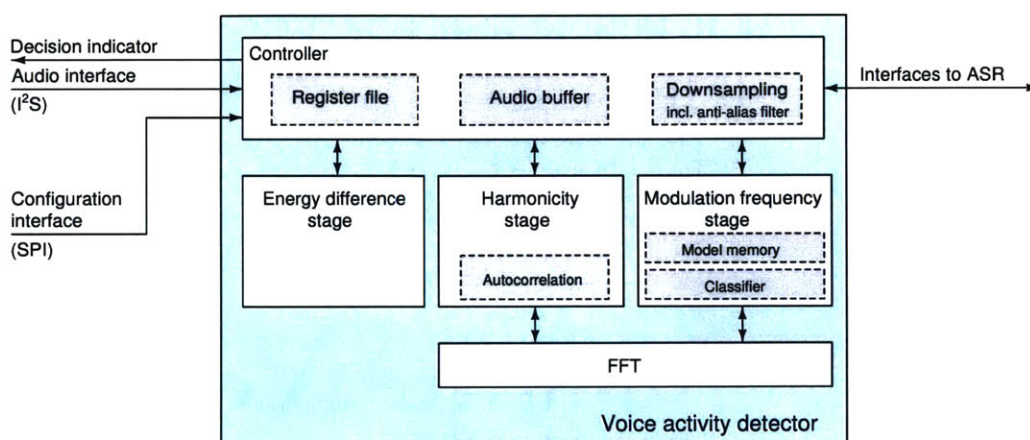


Figure 4-5: Block diagram of VAD module that can selectively enable EB, HM, and MF algorithms.

To save area, the HM and MF algorithms share the use of a single FFT module instantiated at the top level. This is a basic radix-2 FFT with two dual-port memories and a ROM for complex exponential coefficients. It supports variable input length and precision since these two algorithms have different requirements.

The VAD can be operated as a standalone device via the SPI interface, but it also has a set of FIFO interfaces linking it to the ASR (on the same chip, or externally). Updates from the VAD indicate the start or end of a speech segment, as well as the advancing of time within a speech segment. Other devices can request blocks of audio samples from a circular buffer on the VAD that holds 12k samples (0.75 sec at 16 kHz). VAD/ASR interactions must be handled carefully; this will be discussed further in Section 5.1.3.

#### Antialiasing filter

In some scenarios the ASR may use a different sample rate from the VAD algorithms. For example, many speech recognizers are trained on 16 kHz audio, while our VADs run at 8 kHz (there are limited performance

benefits at higher sample rates). We support integer downsampling ratios from 1 to 24. While we assume that the microphone input is anti-aliased, the chip has to perform lowpass filtering before downsampling. A steep filter is required to maintain signal bandwidth close to the Nyquist frequency.

We provide a 65-tap FIR filter with programmable coefficients. The default coefficients are a windowed sinc impulse response, which provide a -3 dB point of 3.5 kHz for downsampling from 16 kHz to 8 kHz.<sup>2</sup> The filter architecture shown in Figure 4-6 is a compromise between area and runtime: it multiplies 8 coefficients per cycle, requiring 9 cycles to obtain the filter output in an accumulator (in addition to the latency of pipelining, which is not shown).

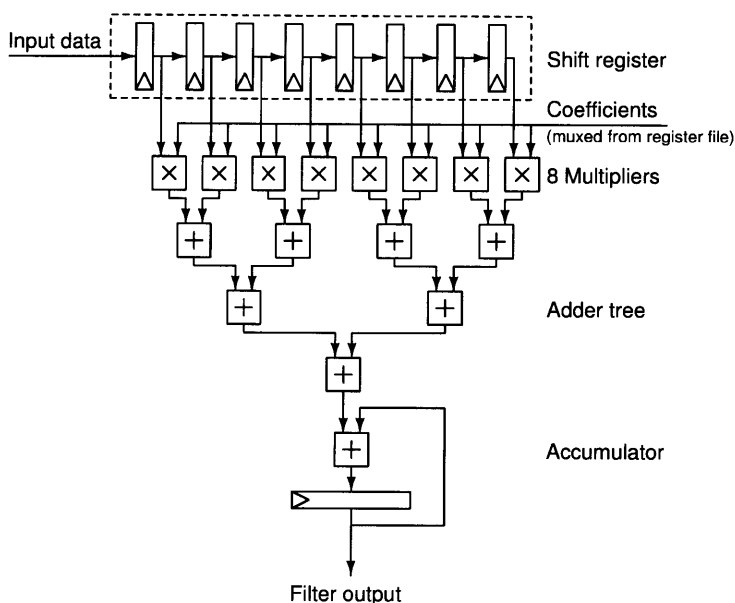


Figure 4-6: The VAD provides a 65-tap antialiasing filter that evaluates 8 taps per cycle.

We did not study numerical precision issues in detail; bit widths for real numbers were chosen conservatively (either 24- or 32-bit) in order to avoid loss of accuracy.

### 4.3.2 Energy-based (EB)

The block diagram of the energy-based VAD is shown in Figure 4-7. Our fixed-point implementation of this VAD operates on Q8.24 data. The logarithm is computed using the shift-and-add algorithm [22] with a 25-element LUT for evaluating  $\ln(1 + 2^{-k})$ . Square roots are computed via successive approximation. (The

<sup>2</sup>An IIR filter with the same performance would require nine 2nd-order biquad sections, with a total of 46 multiplications per sample; it's possible that this would be more efficient than an FIR filter, but the sample rate is so low that efficiency is not very important.

latency of these operations is insignificant because they are performed only once per frame.) The minimum energy of the last 100 frames is computed by grouping samples into bins of 10 frames and computing the minimum over the last 10 bins [51].

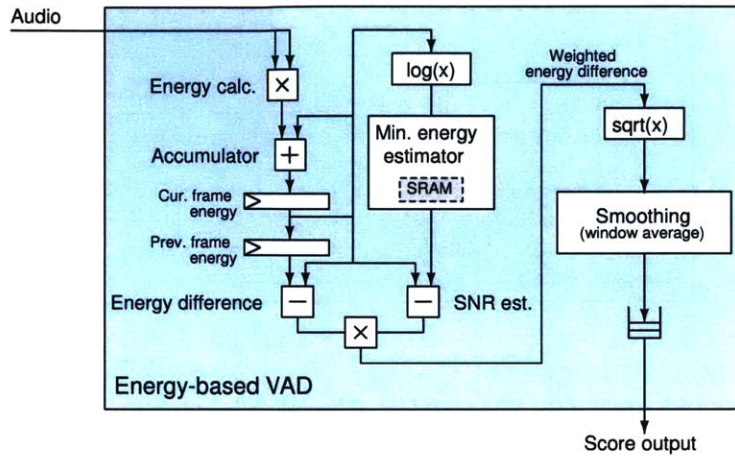


Figure 4-7: Architecture of energy-based VAD module. This module evaluates the difference of energy between two frames and weights it with an estimate of SNR.

The energy-based VAD circuit occupies 51k gates and has 2 kb of SRAM. It requires a clock frequency of 660 kHz to run in real-time on 8 kHz audio.

### 4.3.3 Harmonicity (HM)

The harmonicity VAD shown in Figure 4-8 computes normalized autocorrelation  $r_{xx}[t]$ , finds a peak within the expected pitch range, and translates this peak amplitude to a harmonicity value. The module operates on 512-sample (64 ms) long frames of Q8.16 values with a 10 ms pitch. If the signal level within a frame is below -30 dBFS, we rescale the FFT inputs to improve accuracy. Autocorrelation is computed as the IFFT of the power spectral density. At startup, the autocorrelation of the Hanning window  $w(t) = \frac{1}{2}(1 - \cos \frac{2\pi t}{T})$  is computed and stored. We divide the signal and window autocorrelations using a Taylor series approximation as described by [40]. Recent harmonicity values (up to 64) are stored in order to find the “window maximum” that is used as a score.

Excluding the FFT, this module requires 154k gates and 26 kb of SRAM. Note that this algorithm requires running two FFTs per frame in order to compute autocorrelation.

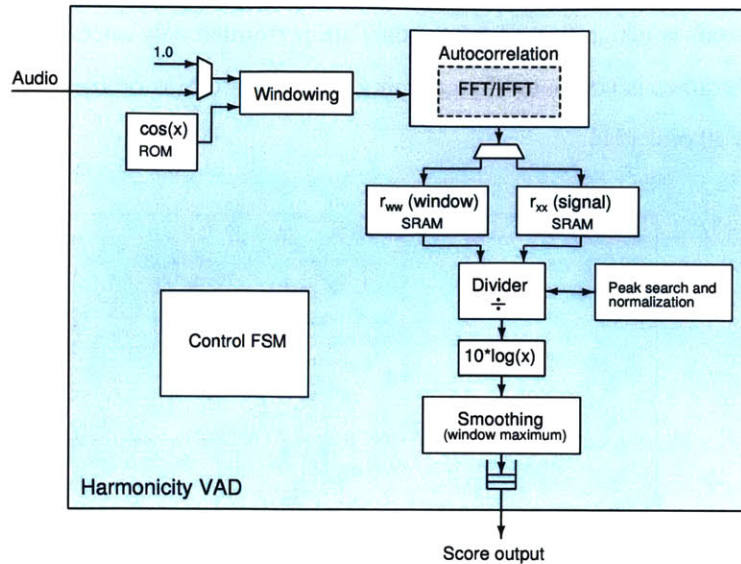


Figure 4-8: Architecture of harmonicity VAD module. This module computes a normalized autocorrelation  $\frac{r_{xx}(t)}{r_{ww}(t)}$  to obtain the amplitude ratio of periodic and non-periodic components, expressed in dB as harmonicity.

#### 4.3.4 Modulation frequencies (MF)

A block diagram of our MF VAD implementation is shown in Figure 4-9. MF features are computed in Q8.24 format. A short-term FFT operates on 512-sample frames with configurable pitch. These FFT outputs are downmixed to a configurable set of frequency bands (typically 16) by adding the square root of the energy from each bin. It is important to use a square root here for best accuracy; the square root is performed using a pipelined 1-select non-restoring algorithm [18]. Then after  $N$  frames, we compute a long-term FFT on the history of energy values from each band. The decibel-scale outputs of the long-term FFT are used as MF features. Only the DC and low-frequency bins (up to 11 Hz) from the LTFT are saved.

The MF algorithm relies on the quantized NN implementation we developed for ASR, with a single execution unit (instead of 32) and sparse matrix support removed to save area. Each MF feature is sent to the neural network for classification as speech/non-speech; the difference between the two last layer outputs can be used as a score. The model is trained using Kaldi on disjoint speech and non-speech segments (no overlap), which improves accuracy even though the audio stream at test time contains overlapped frames. The model is stored entirely in on-chip SRAM; the VAD does not use external memory. A 24 kB capacity is sufficient for 3 layers with 60 hidden nodes; in practice, most of our models are smaller.



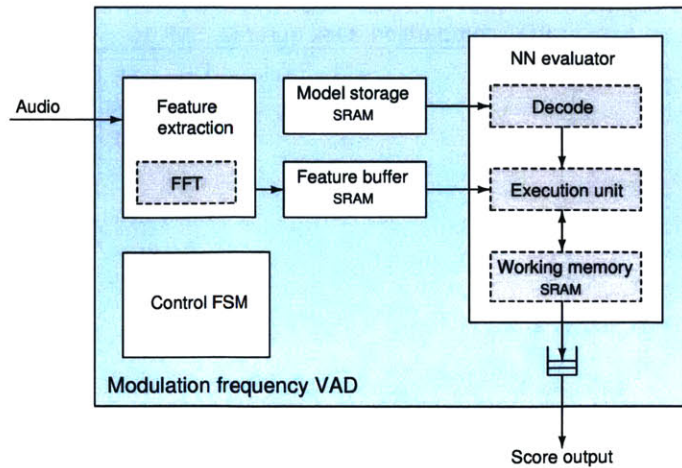


Figure 4-9: Architecture of MF VAD module. This module performs modulation frequency feature extraction and then classification. On-chip memory is used to store a small NN model for classification.

## 4.4 Performance comparisons

We constructed two sets of performance tests for comparing the VADs' detection performance. In each case, noise and speech waveforms were blended in software to create test waveforms with a speech duty cycle of about 20% (lower duty cycles would reflect realistic use, but require increasingly long waveforms). Additional waveforms for training MF models were prepared with a speech duty cycle of 50%. The amplitudes of speech segments in the test waveforms were scaled so the average speech SNRs were uniformly distributed across  $\pm 3$  dB relative to nominal SNR. We trained a single MF model across all SNRs, rather than one model for each nominal SNR. All tests were performed using 8 kHz audio; the antialiasing filter and downsampling were bypassed.

The first set of tests was performed with read digit sequences from TIDIGITS [45], and short (16 min total) noise segments in 8 distinct environments from Aurora2 [36]. Figure 4-10 shows the ROC of each algorithm at 10 dB average SNR. We count frame-level errors by comparing to a reference segmentation (CTM) from a speech recognizer. This method results in higher error rates than other studies, but is necessary in order to estimate power consumption. False-alarm and miss rates up to about 5% can be explained by minor differences in the start and end times of speech segments, even if they are detected correctly.

In Figure 4-11, we use our simple system model to transform the ROC into a plot of power consumption versus miss rate. Dashed lines show educated guesses for the power consumption of the VAD alone. At low miss rates (which are desirable for good ASR performance), average power is dominated by the downstream portion of the system. This is true even with only 1 mW active power.

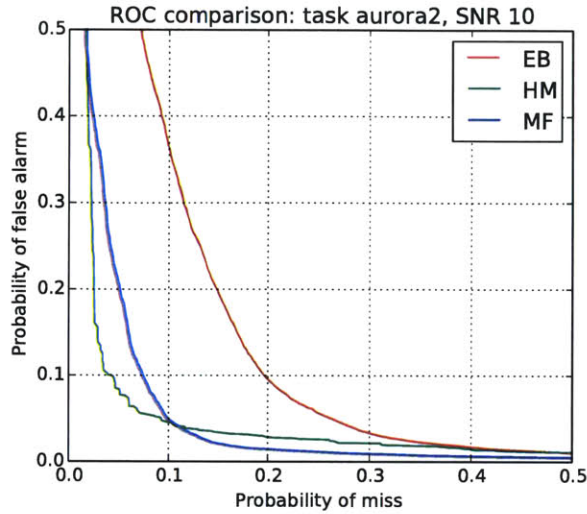


Figure 4-10: Frame-level ROC of each VAD on Aurora2 task at 10 dB average SNR.

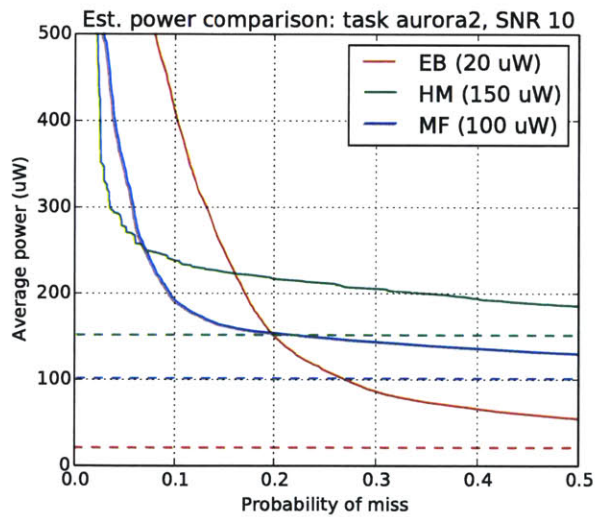


Figure 4-11: Estimated system power, derived from ROC of Figure 4-10, with 1 mW downstream power and 5% speech duty cycle. VAD power figures are educated guesses.

The HM algorithm requires running two FFTs per frame in order to compute autocorrelation. While the MF algorithm is more complex, the long-term FFT and neural network operations are not as energy-intensive as the more frequent short-term FFT. Thus our imprecise estimates show the HM algorithm having slightly higher power consumption than the MF algorithm.

Performance in a given noise condition can be summarized by the equal error rate (EER) where  $p_M =$

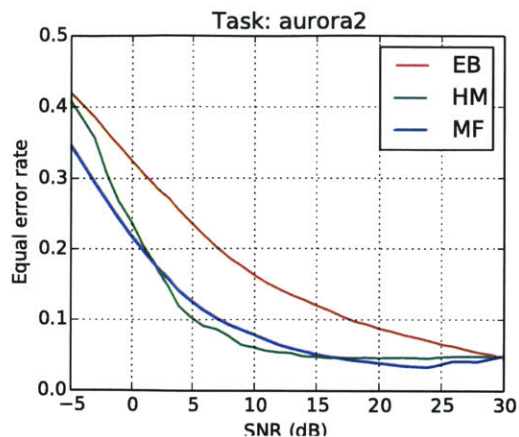


Figure 4-12: EER vs. average SNR on Aurora2 task.

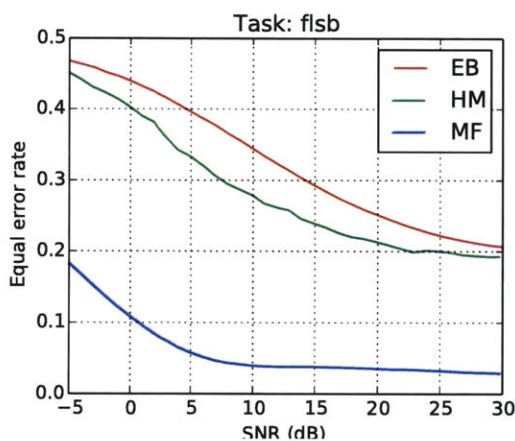


Figure 4-13: EER vs. average SNR on Forklift/Switchboard task.

$p_{FA}$ . Figure 4-12 shows EERs over a range of average SNR from +30 to -5 dB.<sup>3</sup> The HM and MF algorithms offer similar performance, both handling 5–10 dB lower SNR than the EB algorithm. We believe the MF algorithm is underperforming due to insufficient training data.

A more challenging and realistic scenario was constructed using the 53 min of noise waveforms (also in 8 distinct environments) collected for a robotic forklift project [84], and conversational speech waveforms from the Switchboard dataset [25]. For both tasks, training and testing of the MF model used disjoint sets of 4 noise conditions each.<sup>4</sup> This task gives very different results (Figure 4-13). MF performance improves significantly, with little change in EER down to 5 dB SNR. The EB and HM algorithms are essentially

<sup>3</sup>This is not shown, but all three algorithms perform much better when tested on white noise instead of recorded noise.

<sup>4</sup>For Aurora2, the training conditions were [airport, restaurant, car, subway] and test conditions were [babble, train, exhibition, street]. For Forklift/Switchboard, the training conditions were [Vassar, ForkliftBeep, MassAve, WindTunnel] and test conditions were [FortHood, ForbesCafe, ForkliftMotor, LoadingDock]. However, results did not change significantly with different permutations.

useless; we believe this resulted from the more dynamic noise environment and array microphone artifacts. The choice of VAD algorithm should be informed by knowledge of the expected noise conditions and (for the MF algorithm) available training data.

### **4.5 Summary**

This chapter developed and compared digital circuit implementations of three VAD algorithms. As mentioned in Section 4.1, the ROC of a VAD algorithm has a large impact on average system power consumption. This has implications for the design of portable devices. A few extra microwatts spent on the VAD, or a little more effort preparing good models and runtime configurations, can go a long way to reducing unwanted false alarms and extending battery life. We adopted a compact NN classifier for modulation frequency features, providing good noise robustness if sufficient training data is available.

# Chapter 5

## Infrastructure

The acoustic model, search and VAD modules described in earlier chapters contain the primary innovations of this thesis, but are not sufficient to build a hardware speech interface. This chapter details the remaining building blocks that should allow others to reproduce and build on our results, or design other special-purpose hardware more effectively.

### 5.1 Circuits

Figure 5-1 shows a more thorough accounting of the chip's organization. Additional interfaces, control, and signal processing modules are required to tie the major computational modules together. The front-end is designed to compute and buffer MFCC or filterbank features from audio in a configurable fashion. An ASR-specific control module mediates between host (i.e., USB) and VAD interfaces and sequences the operation of all other modules, including startup and shutdown for each utterance. Finally, supervisory logic is necessary to link the VAD and ASR and ensure that the ASR can be safely power gated.

#### 5.1.1 Front-end

The front-end extracts feature vectors such as MFCCs from audio. This accounts for around 1% of the computational workload of ASR [46] and a similar portion of power consumption [67], so it would be reasonable to rely on a software implementation. Nevertheless, we designed a special-purpose front-end to realize our vision of a chip handling audio-to-text transcription, without the complexity or overhead of introducing a general-purpose processor. The front-end shares FFT, logarithm, and divider module designs with the VAD. It is shown in Figure 5-2.

The front-end consists of a series of signal processing stages operating on Q8.24 fixed-point vectors

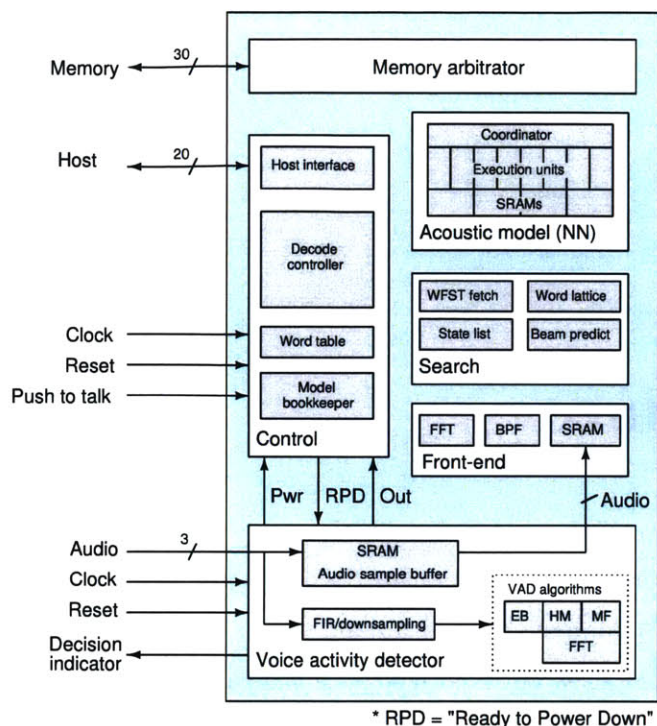


Figure 5-1: Block diagram of the complete IC design, including top-level interfaces and internal ASR/VAD interfaces.

bouncing between two scratch memories. It is meant to be interchangeable with the Kaldi front-end, so that acoustic models trained in Kaldi can be used with little modification to Kaldi. We remove some of Kaldi’s configuration options in order to simplify the implementation: there is no dither, and only the Hanning window is supported. The user retains control of sample rate, number of frequency bands, and whether to perform a DCT (i.e., selecting MFCC vs. filterbank features). Filter bank frequency responses are pre-computed by software and stored in the configuration block of the external memory to be loaded at startup. This simplifies the on-chip logic while adding a small delay to the startup sequence (reading 1 kB of filter coefficients).

A 2048×32 internal memory is used to buffer features, so the front-end can process audio frames as soon as they are received; it takes less memory to buffer features than to buffer the original audio samples.<sup>1</sup>

Some acoustic models expect spliced features (with “before” and/or “after” temporal context), or features with time derivatives (deltas and double-deltas). The front-end supports these transformations, but performs them on-the-fly when sending features to the acoustic model rather than doing so before writing features to the buffer memory. This avoids storing redundant data (i.e., 13-dimensional MFCCs, rather than

<sup>1</sup>Buffering is necessary because the search workload varies, resulting in variable latency for the consumer of the features.

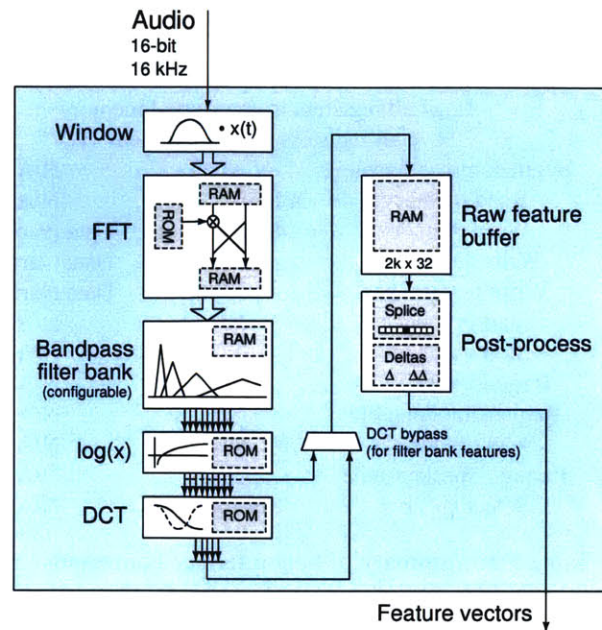


Figure 5-2: Front-end block diagram. The frontend buffers raw features and applies output transformations on-the-fly to save memory.

143-dimensional MFCCs including  $\pm 5$  frames context), allowing the buffer to handle up to 150 frames of search latency.

### 5.1.2 ASR control

The ASR can accept input data and commands from the host interface (a bidirectional 8-bit FIFO) or the VAD, or a combination of both. When using the host interface, the host can supply either audio samples or feature vectors (bypassing the front-end). Statistics and transcriptions are sent through the host interface regardless of the input mode. The protocol for using the host interface is summarized in Tables 5.1 and 5.2. The range of possible use cases, and the real-time nature of local ASR, lead to complex sequencing requirements that need to be centrally controlled.

Control responsibilities are broken down across modules as follows:

- The **decode controller** issues frame-level commands to the front-end, acoustic model, and search based on utterance-level commands from above.
- The **host interface module** acts as an endpoint for the host interface, and handles loading/saving of the primary configuration block from external memory.
- The **word table** converts word labels from integers to strings (using a WFST symbol table stored in

Code	Command	Parameters	Payload
0x10	Read register	Address	N/A
0x11	Write register	Address	Data (32-bit)
0x13	Load all registers from external memory		
0x14	Save all registers to external memory		
0x15	Begin command group	Length	N/A
0x20	Read memory	Address, length	N/A
0x21	Write memory	Address, length	Data (variable)
0x30	Write audio data	Length	Data (variable)
0x31	Write feature data	Length	Data (variable)
0x40	Start utterance	N/A	N/A
0x41	Stop utterance	N/A	N/A
0x42	Request backtrace	Flags	N/A
0x43	Request model table	N/A	N/A
0x45	Select modelset	Modelset ID	N/A
0x46	Request speaker table	N/A	N/A
0x48	Select speaker	Speaker ID	N/A

Table 5.1: Summary of host interface commands.

Code	Response	Parameters	Payload
0x12	Register read data	N/A	Data (32-bit)
0x22	Memory read data	Address, length	Data (variable)
0x32	Backtrace text	Length	Data (variable)
0x33	Word lattice arc	N/A	Data (19 bytes)
0x34	Per-frame statistics	Active stats bitmask	Data (variable)
0x35	Per-utterance statistics	Active stats bitmask	Data (variable)
0x44	Model table data	N/A	Data (variable)
0x47	Speaker table data	N/A	Data (variable)
0x50	Error	N/A	N/A

Table 5.2: Summary of host interface responses.

external memory) at the request of the host interface.

- The **model bookkeeper** supplies information about available modelsets and speakers to the host interface, and allows the desired modelset and speaker to be selected. (A default modelset and speaker, specified in the primary configuration block, are loaded at startup.)

### 5.1.3 Supervisory logic

Supervisory logic is designed to safely shut down and isolate the ASR when it is not needed, power gating the logic and memory supplies off-chip. At the beginning of a speech segment from the VAD, or a wake-up request from the host, the ASR will be powered up and brought out of reset. The sequencing of shutdown and startup are shown in Figure 5-4. This logic runs off the VAD clock, with appropriate synchronizers for ASR-related signals; isolation features make it possible to disable the ASR clock off-chip while the ASR



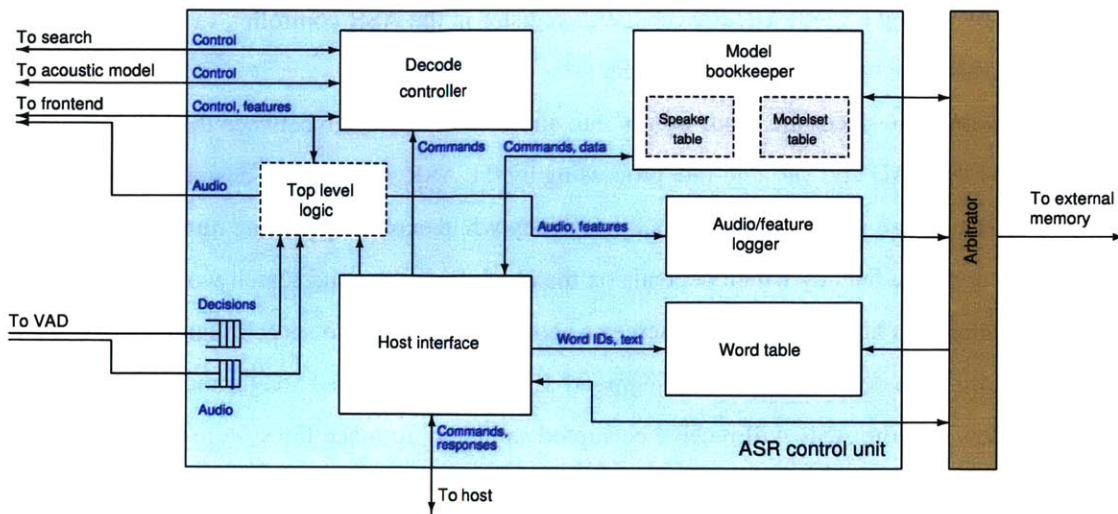


Figure 5-3: Block diagram of ASR controller. The decode controller module coordinates the operation of front-end, acoustic model, and search; interface responsibilities such as audio/feature logging are delegated to other modules.

power supplies are shut down.

State		Control signal outputs
State	Transition rules	iso reset clk pwr
SV_SHUTDOWN	sv_begin_powerup == 1	1 1 0 0
SV_EN_PWR		1 1 0 1
SV_EN_CLK	asr_pg == 1	1 1 1 1
SV_DIS_RST	Delay: 10 cycles	1 0 1 1
SV_ENABLED	Delay: 10 cycles	0 0 1 1
SV_EN_ISO	sv_begin_shutdown_sync && !sv_begin_powerup	1 0 1 1
SV_EN_RST	Delay: 10 cycles	1 1 1 1
SV_DIS_CLK	Delay: 10 cycles	1 1 0 1

Delay: 10 cycles

Figure 5-4: State transition rules for supervisory logic controlling isolation, reset, clock, and power state of ASR.

The ASR and VAD interact through four asynchronous FIFO interfaces shown in Figure 5-5. The VAD sends an update message at the beginning and end of speech segments, as well as periodically (every 10 ms by default). The ASR can request audio samples from a buffer on the VAD side; this enables it to catch up on samples captured during the VAD’s latency, while the ASR was powered down. It is possible to disable

these interfaces by setting a CONFIG\_IGNORE\_VAD register in the ASR controller, in which case the host can be used to mediate between the VAD and ASR.

When the internal interfaces are used, we run into an inherent mismatch between the fixed-rate processing performed by the VAD and variable-rate processing by the ASR. Our ASR is designed to buffer feature vectors, so that search can run at constant clock frequency while exploring varying numbers of hypotheses per frame. This creates a latency which depends on the clock frequency and search workload. If latency at the end of an utterance is high enough, and there is a short gap between two speech segments, the ASR may be delayed in starting to decode the second segment. If this delay is large enough, the audio buffer in the VAD will overflow and the ASR will receive corrupted samples. To make the system robust to these scenarios, ASR decoding is stopped immediately when the end of a speech segment is detected; this effectively clips an unknown portion of the end of the utterance. In the future, it may be possible to design rules that reduce the amount of clipping while preserving robustness (or simply increase the size of the audio buffer). Performing VAD/ASR mediation on the host (assuming it is always on, and has a relatively large memory) resolves these issues at the expense of latency.

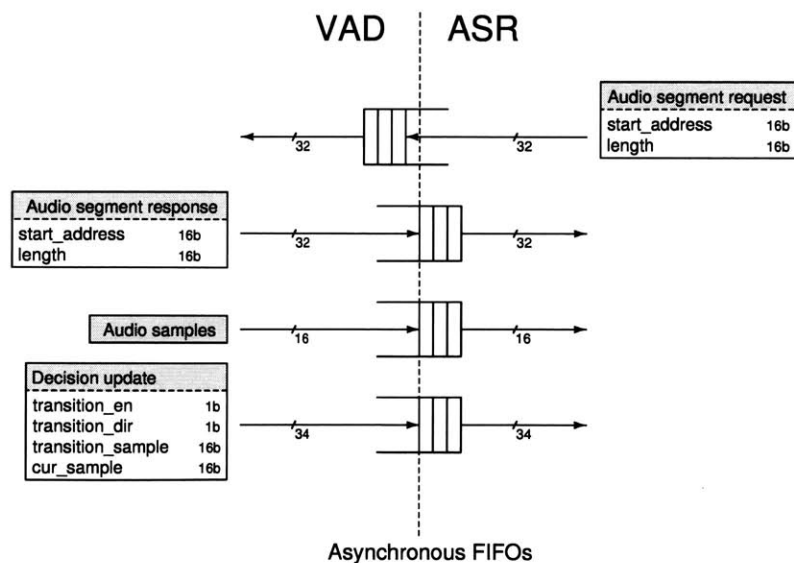


Figure 5-5: VAD/ASR interfaces allow the ASR to retrieve earlier audio samples after it is woken up by the VAD.

## 5.2 Design tools

A recurring challenge in digital system design is propagating design and verification ideas back and forth between the software and hardware domains. This section presents two tools we developed to facilitate hardware/software co-design: binary-backed data structures and hardware-oriented unit testing techniques.

### 5.2.1 Binary-backed data structures

In software design, objects can be serialized for storage or network transmission, and deserialized to construct in-memory data structures. In hardware design, data structures are normally designed “ad hoc,” application specific, and relatively simple.<sup>2</sup> Software includes more elaborate components that are often reused and composed hierarchically. When designing hardware, it’s often necessary to write additional software that converts back and forth between formats.

The middle ground that we established for our off-chip data structures is a “binary-backed” tree that is always stored in serialized form (i.e., a binary blob), but whose leaves can be accessed easily by either hardware or software. Figure 5-6 shows this framework, which is used for prototyping and simulation. The same sequence of bytes is accessible through a C++ API or digital logic.

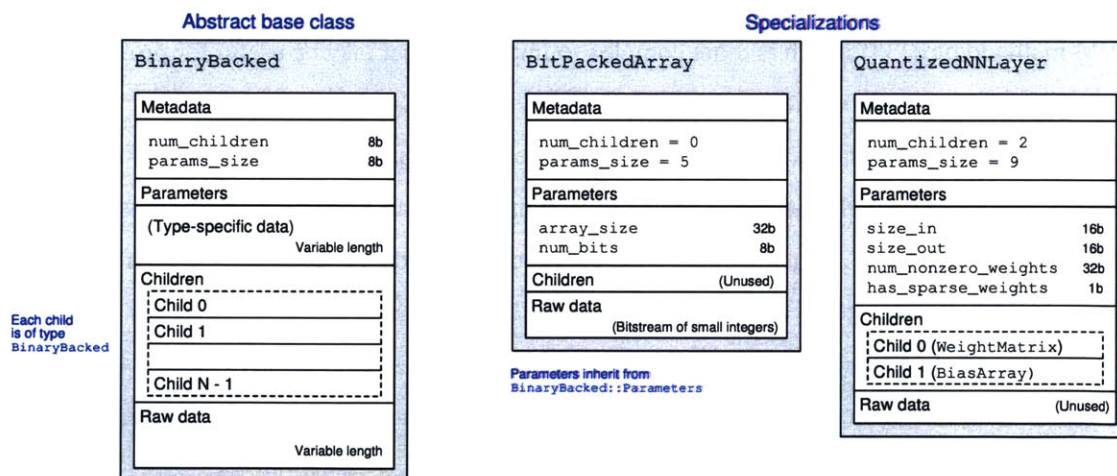


Figure 5-6: Hierarchical binary-backed data structure: base class (left) and example implementations (right).

We wrote classes for building blocks of the models we use, including:

- Arrays of byte/word-aligned and bit-packed values.
- Quantized real-valued arrays (32-bit lookup table followed by bit-packed indices).

<sup>2</sup>You may have noticed that most information stored on-chip in our ASR and VAD architecture has been boiled down to arrays and hash tables.

- Run-length encodings and sparse matrices.

Hierarchical combinations of these building blocks are used to construct the WFST and DNN models used by our system, as well as incidentals such as configuration and word symbol tables. At the top level, the ASR’s external memory stores a blob of type `ModelSuite`. Each model suite contains arrays of WFSTs and DNNs, as well as a “modelset table” showing which combinations of those models can be selected to establish a recognizer. (A WFST and DNN are compatible if the input labels on the WFST correspond to indices in the output vector of the DNN. It is possible for the same acoustic model to be paired with different lexicons or language models.) This makes it possible to prepare a single memory image with multiple recognizers, and switch between those recognizers at runtime (between utterances).

### 5.2.2 Unit testing with hardware/software equivalence

Not only do our hardware and software implementations rely on the same data format, they also have the same architecture and interfaces between modules. This adds some overhead and complexity to the software, and constrains the interfaces that can be used in hardware. In exchange, it allows simulations that mix and match hardware and software modules. The tools we developed rely on the `import` and `export` feature of SystemVerilog (SV) to construct RTL-level unit tests driven by software.<sup>3</sup> This provides an alternative to the UVM methodology [38] which is more complex to implement, but may be useful for projects that have a lot of software infrastructure (such as this one).

The principles of this approach are laid out in Figure 5-7. A high-level description of each module, including its interfaces and submodules, is specified in Python. Available interface types include clock/reset, FIFO, request/response, SRAM, and simple wire connections; the data transmitted by each interface is a packed struct, and all data interfaces are synchronized with one of the clocks. Python scripts generate C++ and SV templates for each module; the designer then adds implementation details. The C++ version uses a callback function for incoming data on each interface, and API functions to perform transactions on the other interfaces.

The software/hardware link is established by a SV “placeholder” version of each module that is automatically generated and instantiated in the RTL simulation. The placeholder does not implement any functionality, except to monitor the interfaces and call C++ functions that have been imported from the software implementation of the same module. Likewise, the software implementation calls SV tasks that have been exported from the placeholder to actuate the interfaces. Whether a module is simulated in hardware or

---

<sup>3</sup>We use the GoogleTest framework, but other unit testing frameworks in C++ and other languages could be used with the appropriate plumbing.

software depends on what is provided in the RTL simulator’s compile command: the path to the placeholder SV file, or the path to the actual SV implementation. All of the C++ modules are compiled into a shared library so they can be called as needed by the RTL simulator.

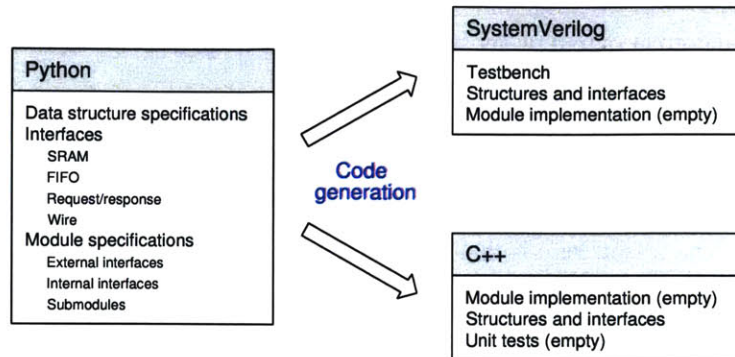


Figure 5-7: Tools for generating simulation models with hardware/software equivalence.

To speed things up, there is a “software only” mode which routes messages directly through the module tree. This allows simulations to be run on an all-software design without RTL simulation licenses.

We implemented 166 unit tests against 33 modules, as well as other non-unit tests, as part of the verification process. The software-driven framework made many of our tests easier to construct and maintain. For example:

- We construct several small WFSTs in OpenFST format, compress them, and run Viterbi search (with a preordained schedule of acoustic model likelihoods) while verifying the hypotheses after each frame. This allows us to check proper handling of specific WFST characteristics such as self-loops, final weights, and high in-degree. It also ensures that our WFST compression and decompression are consistent.
- With ASR partitioned into front-end, acoustic model, and search, we evaluated the performance of each by substituting it into a Kaldi decoder pipeline. For example, we would use Kaldi’s front-end and search with our (fixed-point and quantized) acoustic model. This helped isolate bugs that didn’t cause obvious problems in simulation of individual utterances, but degraded accuracy when tested against many utterances.
- We perform numerical consistency checks between the hardware and software versions of each module.
- We compute error statistics of our fixed-point numerical functions (square root, divider, sigmoid, etc.) against their C++ STL implementations.

Despite these efforts, verification remained a major challenge for this design. For example, unit tests for the hash table checked that individual operations such as insertions, reads, and deletes were performed correctly. However, they did not thoroughly explore sequences of different operations. Bugs in the implementation of dynamic resizing caused keys to occasionally be stored in an inconsistent order, leading future operations to produce incorrect output or get stuck in an infinite loop. This led to unexplained lockups and incorrect recognizer output. Due to the long timescale (sometimes millions of cycles) between the occurrence of the bug and its detection, it was difficult to diagnose these problems through simulation or with the FPGA’s internal logic analyzer. After careful analysis it was possible to write tests that triggered these bugs, leading to fixes. The lesson here is that unit testing never provides complete coverage; it should be used alongside other tools such as constrained random verification or formal proofs. We briefly discuss the changing demands of verification in Section 7.2.1.

## 5.3 Software

We created tools for deploying software recognizers on our hardware platform. These tools should be accessible to anyone experienced with Kaldi.

### 5.3.1 Kaldi modifications

Kaldi provides “recipes” (scripts) for training and testing recognizers with several publicly available datasets. Our recognizers were constructed with these and additional recipes that we created for our datasets (Jupiter and nutrition). However, some features enabled by default in these recipes don’t apply to our use cases. We modified the scripts to disable these features:

- Cepstral mean and variance normalization (CMVN) cannot be performed on a per-utterance level in real-time decoding. Global feature whitening is performed before the first layer of the NN acoustic model.
- All of our models are speaker-independent, resulting in lower accuracy than the speaker-dependent models trained in Kaldi. (In some recipes Kaldi performs two-pass decoding to estimate an fM-LLR transform for each utterance, which is far from real-time.) Our architecture supports speaker-dependent decoding if appropriate models are available. We prepare a “feature transform” NN to be chained in front of the “acoustic model” NN. Either of these NNs can be changed at runtime; for example, the feature transform could be an affine layer with no nonlinearity that performs fMLLR for a specific speaker.

To evaluate sparse NN acoustic models within Kaldi, we added a program to truncate weights smaller than a certain threshold, and modified the existing Kaldi binary `nnet-train-firmshuff` to support retraining with a fixed sparsity pattern (following [93]).

### 5.3.2 Model suite preparation

Information about the ASR configuration is read from a variety of sources specified in an XML file, as shown in Figure 5-8. Our `prepare_model_suite` program transforms the XML specification, and a disparate collection of model files created by Kaldi, into a single binary blob (the `ModelSuite`) that can be loaded into external memory over the host interface.

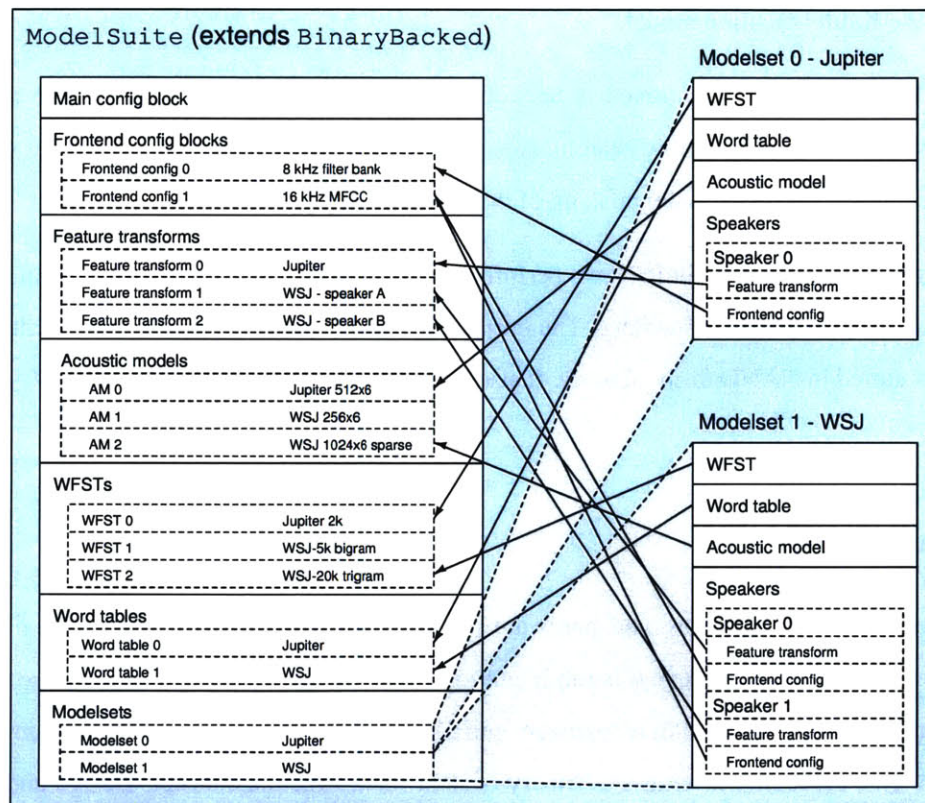


Figure 5-8: The `ModelSuite` data structure allows multiple Kaldi recognizers to be compiled into a single binary blob. This example shows a configuration supporting weather information (Jupiter) and business news (WSJ) tasks.

Before converting the models into compressed format and organizing them into modelsets, we apply the following transformations:

1. **Set up front-end and feature transform:** In Kaldi, the binaries `splice-feats` or `add-deltas`

add temporal context or time derivatives to the feature vectors as part of a decoding pipeline; `transform-feats` can apply a linear transformation such as LDA. Kaldi NN recipes often add a splice such as  $\pm 5$  frames as an NN component (layer). Regardless of how it's being performed in Kaldi, feature splicing in our system is performed by the front-end (rather than the NN). The XML file specifies splice and delta values which are stored in the model suite's configuration block. Other feature normalization (and optionally LDA/fMLLR) is pushed into the feature transform NN, to be applied immediately before the acoustic model.

2. **Replace WFST input labels:** Kaldi uses transition-ids as WFST input labels. We replace these transition-ids with pdf-ids so the index of the NN output vector equals the input label minus 1. We discard the Kaldi transition model.
3. **Add WFST  $\epsilon$  arcs:** As mentioned in Section 3.1 and [16], we add “bypass” arcs to the WFST to avoid recursion in the  $\epsilon$  phase of search. This process may add compound labels to the word symbol table, depending on earlier label pushing of the WFST.
4. **Reorder WFST states:** As in [67], we perform a breadth-first search to cluster sequentially accessed states close to each other in memory. The purpose of this is to reduce page access penalties when the WFST is stored in NAND flash memory. It also improves the effectiveness of our WFST compression scheme.

## 5.4 Summary

This chapter explained the software and hardware infrastructure needed to deploy ASR on a chip with VAD-based power gating. We believe it is important to streamline the deployment process, so we provided software to compile one or more Kaldi recognizers into a single-file “model suite” configuration for the hardware platform. This procedure leverages a “binary-backed” data structure concept. Design and verification of the many circuit blocks required for the chip was facilitated by a code generation scheme that produces hardware and software modules having equivalent interfaces. These two techniques were presented in the hope that they will be useful for other researchers and engineers.



## Chapter 6

# IC Implementation

### 6.1 Overview

This chapter shows how the RTL-level design from preceding chapters were transformed into an ASIC that could be characterized and demonstrated, and provides early measurement results. This process includes physical design of the IC layout, and design of the test setup (spanning PCB design, FPGA firmware, and software). While the same general procedure is used for most digital design projects, we highlight design patterns that may be helpful for other researchers in this area. This is a multi-voltage design with a large amount of on-chip memory.

Figure 6-1 shows the logical interfaces of the chip. The VAD relies on industry-standard SPI and I<sup>2</sup>S interfaces for control and audio input, respectively. The ASR has generic 8-bit FIFO interfaces for communicating with the host and with an external memory.

### 6.2 Synthesis flow information

This chip was designed for the TSMC 65 nm low-power logic process, using third-party IP libraries for standard cells, I/O, and memory compilers.<sup>1</sup> There was no custom circuit design or layout. The low-power process was helpful to this design because of the large area and generally low activity factors. Much of the scalability in the design comes from varying the clock frequency; the higher leakage of a high-performance process would blunt the benefit of a slower clock.

There are 87 memories of 30 distinct types in the design, including single-port SRAM, dual-port SRAM,

---

<sup>1</sup>We are grateful to TSMC for fabricating these chips through the University Shuttle Program, and to EDA vendors Synopsys, Mentor Graphics, and Cadence for supplying tool licenses and documentation.

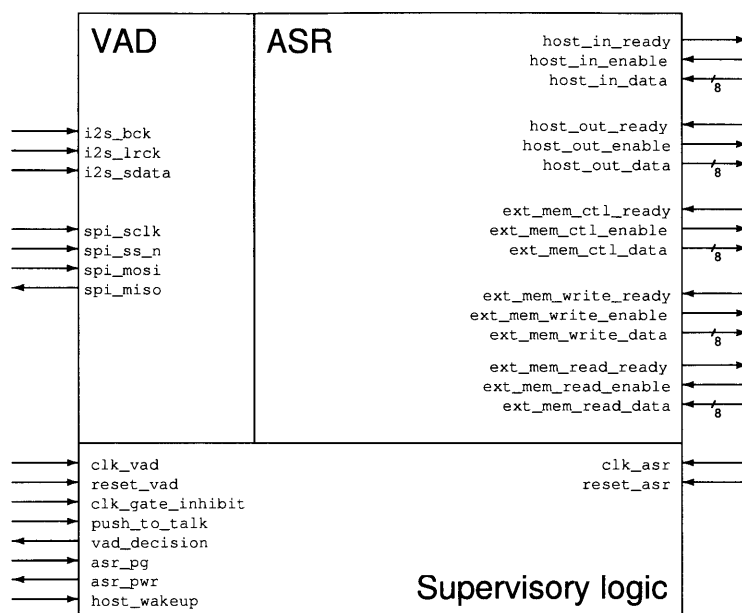


Figure 6-1: Top level pinout partitioned into ASR, VAD, and supervisory regions.

and ROM. To speed up library preparation, we automated the following procedures:

- For SRAMs:

1. We start with behavioral Verilog implementations of single-port and dual-port SRAMs.<sup>2</sup> These use width and depth module parameters to control the dimensions of a register array. (SV modules throughout the design are parameterized, with the dimensions of RAMs depending on the values of those parameters.) The internals of the module can be replaced with dummy logic using a ``ELABORATION_ONLY` preprocessor directive.
2. The synthesis tool is used to elaborate the chip design with ``ELABORATION_ONLY` defined (in order to avoid instantiating millions of registers). The tool generates a report of all unique modules after elaboration, which includes the module names and parameters.
3. A Python script parses the elaboration report and generates configuration files for the memory compiler. It determines the most area-efficient column muxing for each RAM and when RAMs need to be tiled in width and/or depth.

- For ROMs:

<sup>2</sup>These implementations (whether SRAM or ROM) are structured for automatic inference as synchronous block RAM by FPGA synthesis tools, so that the same source files are used throughout the design process.

1. A Python script computes the values to be stored. It generates a behavioral Verilog implementation of each ROM using a `case` statement; this Verilog is used for RTL development and simulation. It also generates an equivalent Intel HEX file to be read by the memory compiler.
- For both SRAMs and ROMs:
    1. The appropriate memory compiler is invoked to generate a GDS layout, SPICE netlist, Verilog model, LEF file, and timing/power library.
    2. For each distinct ROM type, we generate a Verilog wrapper that conditionally instantiates the behavioral or the compiled module, based on a `use_compiled_mem` module parameter. This way the same source files can be used for RTL simulation (with either behavioral or compiled memories), FPGA synthesis, ASIC synthesis, and gate-level simulation.
    3. Once for all single-port SRAMs, and once for all dual-port SRAMs, we generate a Verilog wrapper that conditionally instantiates the behavioral module or one of many compiled modules, based on the `use_compiled_mem`, `width`, and `depth` module parameters.
    4. The timing/power libraries are compiled for selected operating points (best, typical, and worst case).
    5. Blockage, pin, and via (BPV) extraction is run to generate a physical library from the LEF file and GDS layout.

Figure 6-2 depicts the synthesis and verification flow from RTL sources to the final layout. Memory library preparation was run as necessary to synchronize with changes in on-chip memory requirements. The remainder of the process took 36–40 hr with the available compute resources in our lab; the critical path is through synthesis (2 hr), placement and routing (18–24 hr), and gate-level simulation (8–12 hr). It may have been possible to speed up placement and routing by using a hierarchical flow; at 736k instances, design complexity was close to the limit of our tools. A hierarchical flow should be considered for designs with more than 500k instances.

Digital designs rely on a “chain of equivalence” for verification (Figure 6-3), because the outcome of the synthesis flow (a GDS layout) is too complex to be simulated directly. The chain ends at a level of abstraction where complete functional verification is possible. For small designs, it may be possible to thoroughly verify the gate-level netlist. For this design, it was not possible to simulate complete datasets with gate-level or RTL designs, so the chain extends to FPGA and software implementations. In theory, it’s possible to establish exact equivalence between different versions of the design; in practice, some of the links in the chain have uncertain quality. We tried to leave sufficient redundancy between these branches to

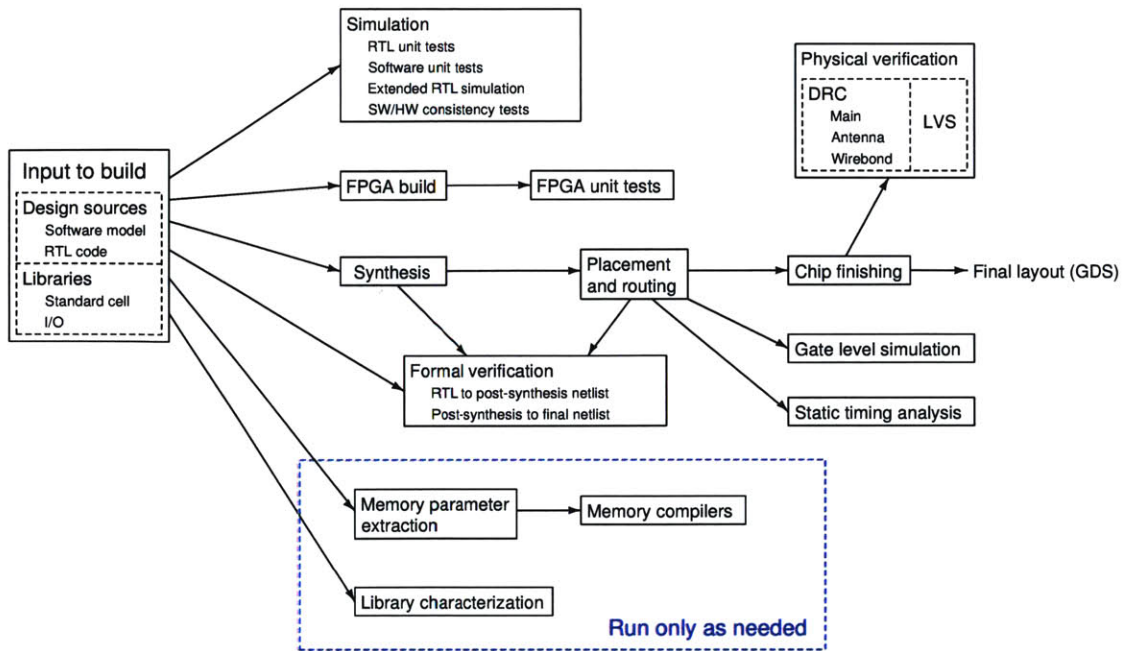


Figure 6-2: Physical design flowchart illustrating the dependencies between different steps of IC implementation.

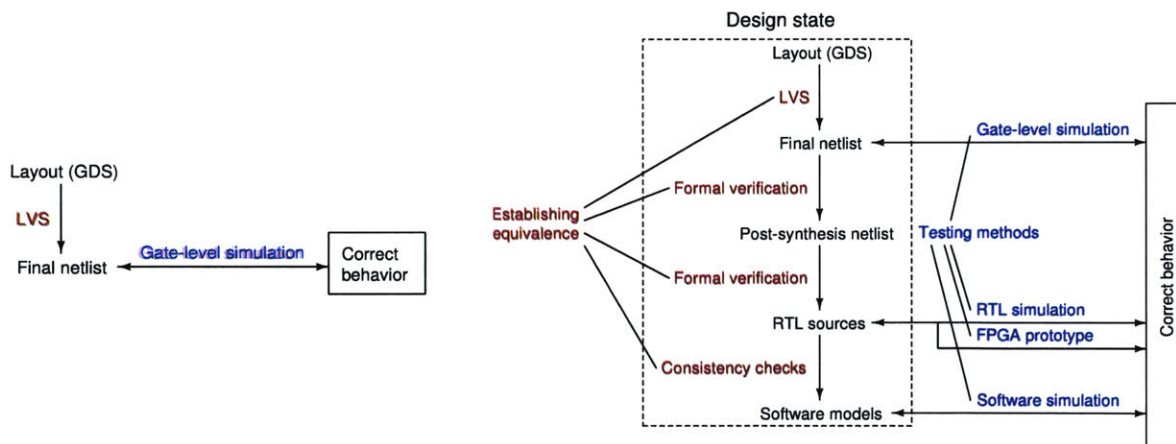


Figure 6-3: Verification chains: ideal (left) and real (right).

account for imperfections. For example, we added gate-level simulation coverage for 32-bit arithmetic in the front-end and VAD that resulted in hard (inconclusive) formal verification. We exhaustively simulated the compiled memory wrappers to make sure that tiling and connectivity were correct.

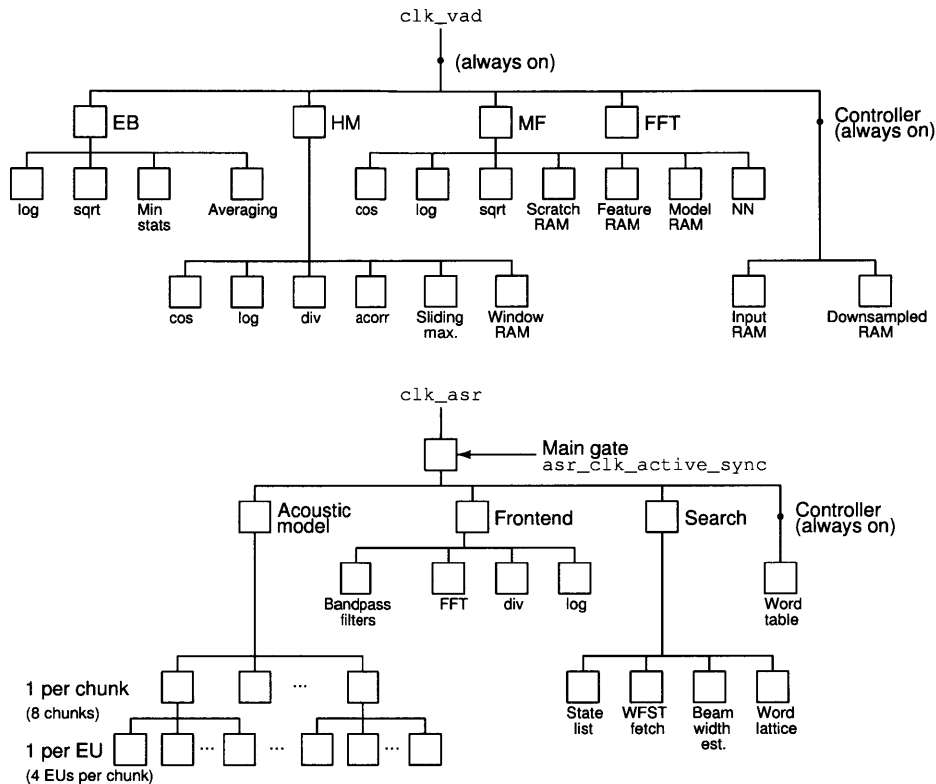


Figure 6-4: Clock gating hierarchy: VAD clock domain (top), ASR clock domain (bottom).

### 6.2.1 Clock gating: explicit and implicit

Clock gating is an important power reduction tool, especially for designs like this that have low or highly variable switching activity. Modern synthesis tools automatically insert clock gates on groups of registers, and our design implicitly assumes this optimization is being performed. However, at higher levels of the clock tree the tools cannot infer simple rules for the control signals; this requires knowledge of design intent, which we supply explicitly.

To save power in higher levels of the clock tree, our explicit clock-gating scheme partitions the design as in Figure 6-4. Modules that employ clock gating generate a synchronous enable signal for each of their child clock domains. The clock gating is used both at high levels (enabling the front-end, acoustic model, and search only when needed) and lower levels (enabling a subset of NN execution units, or arithmetic blocks in the VAD) of the module hierarchy. This is facilitated by embedding the standard latch-based clock gate in our SV clock/reset interface. At most connection points the enable signal is always high, and the synthesis tools optimize away the clock-gating cell.

Because we were unable to evaluate our clock gating scheme on the FPGA, and simulations provide

insufficient coverage, there was some risk of functional failure. We added a `clk_gate_inhibit` input to the chip which would bypass all of the explicit clock gates. During chip testing, explicit clock gating did not affect functionality; we analyze its impact on power consumption in Section 6.4.

We opted not to implement on-chip power gating for the ASR logic and memory voltage areas. Instead, we provide a power control signal `asr_pwr` to off-chip power gates. To wake up the ASR, supervisory logic enabled `asr_pwr` and does not proceed until the “power good” signal `asr_pg` is high. To disable power gating, the control signal can be ignored and `asr_pg` can be held high.

## 6.2.2 Floorplan

Our synthesis flow employed the IEEE 1801 UPF standard for low-power design, so that the tools could automatically insert appropriate level shifters and perform supply-aware simulation and timing analysis.

Floorplanning is the process of determining the size and shape of each hierarchical block, and each voltage area within each block. We partitioned the design into 5 voltage areas: 2 each for VAD and ASR (logic and memory), and one “top-level” area for supervisory logic and external interfaces. For purposes of level shifting and timing analysis, we assume the following voltage relationships:

1.  $V_{ASR\_MEM} \geq V_{ASR\_LOGIC}$  and  $V_{VAD\_MEM} \geq V_{VAD\_LOGIC}$

Memory is powered from a higher supply voltage than logic.

2.  $V_{TOP} \geq V_{ASR\_LOGIC}$  and  $V_{TOP} \geq V_{VAD\_LOGIC}$

Supervisory logic is powered from a higher supply voltage than VAD or ASR logic.

The architecture allows logic/memory connections within the VAD and ASR, but all communication between the VAD and ASR modules (and to/from the I/O pads) goes through the supervisory voltage area (Figure 6-5). If the above assumptions are met, every connection that crosses a voltage area boundary is always either low-high or high-low; the direction of voltage change isn’t affected by any adjustments to the supply voltages. One set of level shifters is sufficient for any allowed scenario.

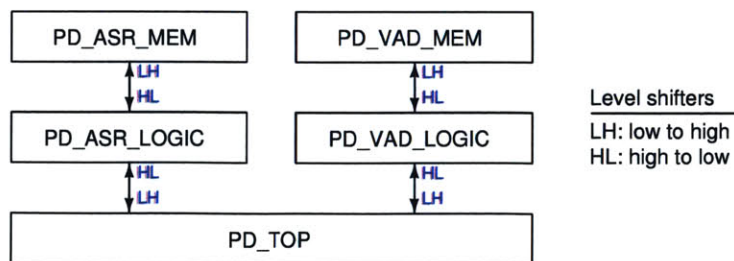


Figure 6-5: Interaction between voltage areas is designed so that voltage differences are consistent with level shifter orientation.

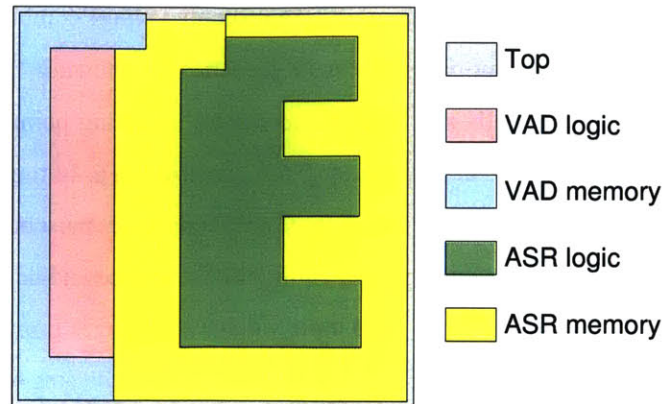


Figure 6-6: The ASR/VAD chip floorplan groups logic into contiguous regions to simplify optimization by the placement and routing tool.

The floorplan is shown in Figure 6-6. Memory macros were placed manually; it is technically possible to place them automatically, but we were not happy with the density and quality of results (especially in the multi-voltage case). The voltage area shapes must be specified manually within the constraints of the tool (for example, no closed “ring” shapes). Supervisory logic is placed near the core boundary and interfaces with the I/O cells. Memories are generally grouped together around a contiguous region of logic in the center. Revisions to the floorplan, such as altered memory placements and dedicated channels for routing, were needed to address congestion and timing problems arising from paths traversing the memory voltage area.

Table 6.1 summarizes the post-layout area requirements of each part of the floorplan.

	<b>Logic</b>	<b>Memory</b>
<b>ASR</b>	1562k gates	5174 kb
<b>VAD</b>	477k gates	625 kb
<b>Supervisory</b>	49k gates	None
<b>Total</b>	2088k gates	5840 kb

Table 6.1: Summary of ASR/VAD chip area.

### 6.2.3 Timing considerations

One of our goals is to explore the limits of scalability and minimize power consumption on simpler ASR and VAD tasks. This requires scaling to supply voltages significantly below the nominal  $V_{DD}$  of the process technology. However, logic and memory performance strongly depends on supply voltage, with longer and more uncertain delays at lower voltages [69]. The challenge is to ensure timing closure (no hold violations

and reasonable clock frequencies) across a range of different supply voltage combinations.

We checked timing using post-layout parasitics in 12 scenarios, as shown in Table 6.2. We performed library characterization for standard cells and level shifters at all operating points not supplied by the IP vendor. The inverter gate delay in the slowest scenario (0.6 V, worst-case) is 18 times longer than the fastest scenario (nominal  $V_{DD}$ , best-case). Paths that cross voltage area boundaries must be designed conservatively to account for highly variable clock tree latencies depending on the source and destination voltages. In the typical process corner, clock tree latency in the VAD logic voltage area ranges from 1.98–2.30 ns at nominal  $V_{DD}$ ; this increases to 8.63–15.34 ns at 0.6 V. This voltage area communicates with the top level voltage area, which is at nominal  $V_{DD}$  in both scenarios and therefore has fixed clock latency (7.05–7.66 ns). Buffering and routing to satisfy hold time constraints in both scenarios will inevitably lengthen the clock period; fortunately, we don’t need high clock frequencies.<sup>3</sup> In multi-voltage designs with wide operating ranges, it might make sense to use asynchronous FIFOs even for synchronous voltage area crossings to remove these challenging constraints.

Voltages VAD/ASR/Top	Hold margin (ns)		
	WC	TC	BC
0.6/0.8/Nom	2.41	1.25	0.90
0.6/Red/Nom	1.64	0.88	0.49
Red/Red/Nom	1.64	0.88	0.49
Nom/Nom/Nom	0.57	0.47	0.35

Voltages VAD/ASR/Top	ASR clock frequency (MHz)		
	WC	TC	BC
0.6/0.8/Nom	49.7	62.9	83.4
0.6/Red/Nom	64.9	109.3	164.2
Red/Red/Nom	64.9	109.3	164.2
Nom/Nom/Nom	72.1	112.9	164.2

Table 6.2: Static timing verification scenarios covering the space of expected supply voltage combinations. “Nom” indicates the nominal  $V_{DD}$  of the standard cell library and “Red” indicates a reduced  $V_{DD}$ .

We satisfied these constraints by using a multi-scenario optimization in the placement and routing tool, with two scenarios representing the extreme cases (Table 6.3). In the low-voltage scenario, we modeled on-chip variation using the global slow/slow corner for all “slow” paths and the fast/fast corner with 1.5x derate for all “fast” paths.<sup>4</sup> We found this to provide reasonable variation robustness given that statistical

<sup>3</sup>The VAD can run in real-time with a clock frequency of 4 MHz or lower. The ASR clock is faster, but there is less variation to deal with since the ASR logic is only operated down to 0.8 V in these scenarios.

<sup>4</sup>When evaluating setup constraints, the destination clock is treated as “fast” while the source clock and data chains are treated as “slow;” these roles switch when evaluating hold constraints.



models were not available. 0.3 ns clock uncertainty was added as a margin. An additional round of hold fixing was required for the nominal voltage, worst-case process scenario.

Parameter	Scenario 1: nominal		Scenario 2: const supply	
	Max	Min	Max	Min
VAD operating conditions	WC 0.6	BC 0.6	WC Nom	BC Nom
VAD clock frequency	8 MHz		25 MHz	
ASR operating conditions	WC 0.8	BC 0.8	WC Nom	BC Nom
ASR clock frequency	40 MHz		66 MHz	
Top operating conditions	WC Nom	BC Nom	WC Nom	BC Nom
Timing analysis type	On-chip variation		Best-case/worst-case	
Timing derate factor	1.0	1.5	1.05	0.95
Clock uncertainty	1.0 ns		0.3 ns	

Table 6.3: Timing optimization scenarios cover the extremes of possible multivoltage operating points. “Nom” refers to the nominal  $V_{DD}$  of the standard cell library.

Assuming typical process conditions, this library and constraint configuration allows 62 MHz ASR operation at 0.8 V; this is faster than real-time with any of the recognizers we tested. The 1.25 ns hold margin should account for local variation at 0.8 V and possibly lower voltages.

The final layout of the chip is shown in Figure 6-7. Outer dimensions are 3.63 x 3.63 mm, with a core size of 3.1 x 3.1 mm. There are 132 I/O and supply pads, intended for wirebonding in an 88-pin QFN package with 2 pads per supply pin and all grounds downbonded to an exposed pad. Note that 62% of the active core area is occupied by SRAM.

### 6.3 Test setup

This section describes PCB, FPGA, and software design considerations for a test setup requiring no specialized lab equipment.

A typical test setup is illustrated in Figure 6-8. A pattern generator provides I/O data and clock; a logic analyzer is used to verify that the outputs are consistent with expected behavior. Additional power supplies and multimeters are used to control and monitor the power supplies. The equipment can be manually operated or, with additional setup work, sequenced over GPIB.

For improved flexibility and portability, and reduced testing time, we centralize much of this functionality in 2 PCBs that connect to a host computer. One is a custom PCB carrying the chip, and one is a commercial FPGA board.

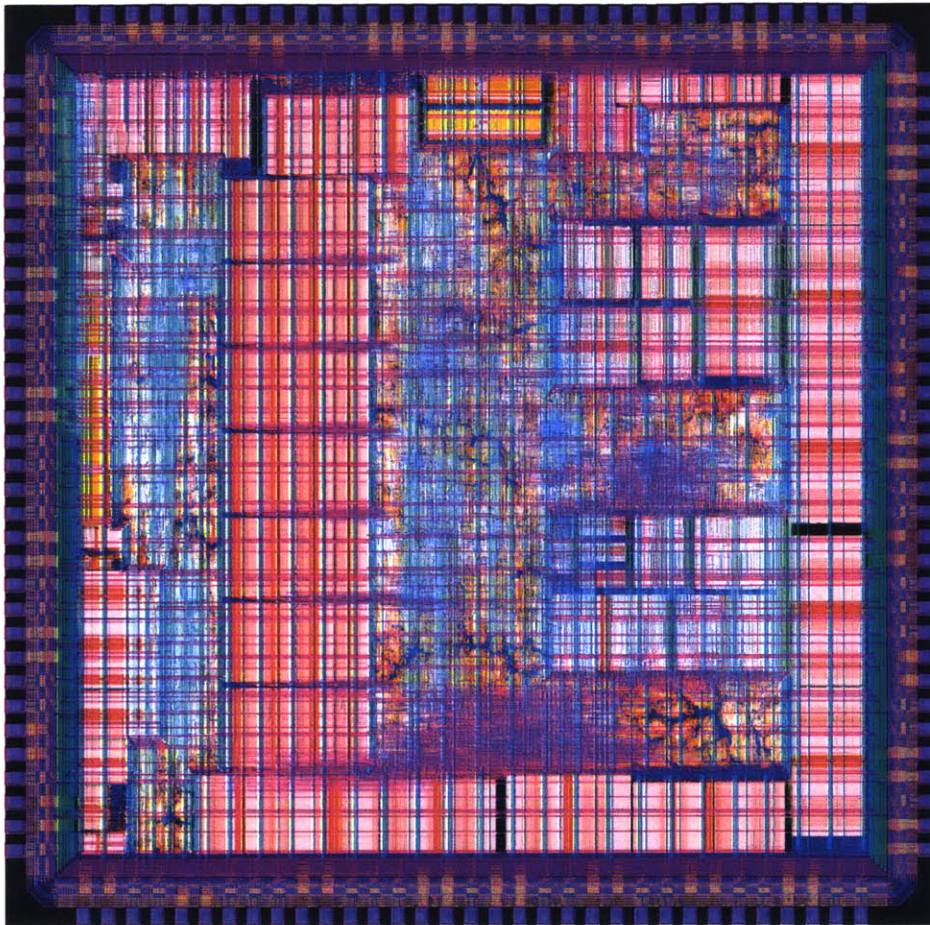


Figure 6-7: ASR/VAD chip layout.

### 6.3.1 PCB

Our custom PCB is illustrated in Figure 6-9. The INMP441 MEMS microphone connects to the VAD portion of the chip directly over the I<sup>2</sup>S bus.

The ASR/VAD chip has 6 power supplies which are independently controlled and monitored. We implemented onboard power supply regulators using SPI DACs and ADCs. Figure 6-10 shows the schematic of one power supply channel. To allow accurate current measurements, the current sense resistor is sized to create about 1 V of drop at full-scale,<sup>5</sup> while remaining inside the feedback loop for good load regulation. Lead compensation ensures that the loop is stable despite the large capacitive load on the op amp. Output impedance peaks at about 0.5  $\Omega$  at 30 kHz. The lowpass filter formed by R1 and C1 prevents overshoot when responding to steps in target voltage from the DAC. Note that the 10  $\mu$ F bulk decoupling capacitor has

<sup>5</sup>The op amp has an output current limit of 50 mA.

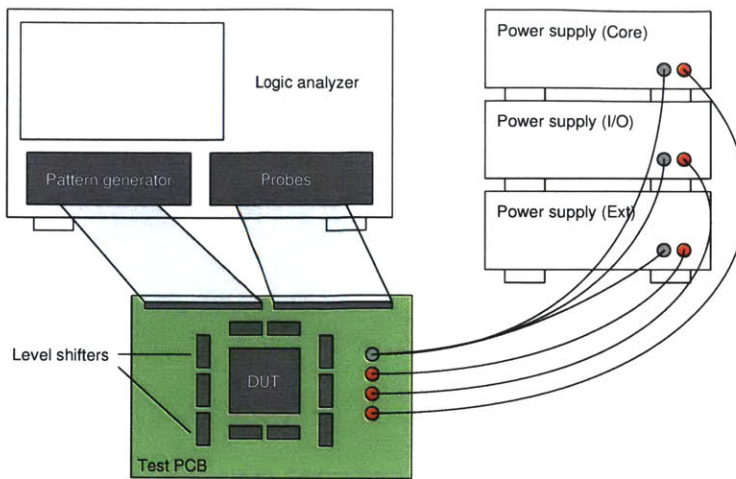


Figure 6-8: A conventional digital IC test setup requires the coordination of several pieces of lab equipment.

been chosen jointly with the feedback network; larger is not better. The current sense resistor and decoupling capacitor can be varied jointly (maintaining the same  $RC$  product) without adjusting the rest of the feedback network.

Linear Technology LTC6903 clock generators supply independent ASR and VAD clocks at frequencies specified over the SPI bus. A Total Phase Cheetah SPI adapter interfaces all of these peripherals (and the VAD itself) to the host computer; the number of slave-select lines is expanded by a 74LVC138 3-to-8 one-hot decoder. We also successfully used a Silicon Labs CP2130 USB/SPI bridge, so that a single USB port handled all of the test setup's data and power needs. (The VAD has no external memory, so no FPGA is necessary to use it.) Test results are reported with the Cheetah, due to its lower and more consistent latency than the CP2130.<sup>6</sup>

### 6.3.2 FPGA

We use a commercial FPGA board to translate from the chip's generic host and memory interfaces to USB and DDR3, respectively. A block diagram of the FPGA firmware and related board-level components is shown in Figure 6-11.

DDR3 memory access is provided by a Xilinx memory interface generator (MIG) and an interface adapter allowing bitwise addressing and arbitrary length read/writes. To create a USB host interface, we wrap our own bitwise protocol around the 32-bit block pipes provided by the FPGA board vendor. Wire I/Os allow the host computer to monitor control outputs and actuate control inputs. (Memory access

<sup>6</sup>Both SPI adapters have an adjustable SCLK frequency; we had to reduce SCLK to 3 MHz for the lowest supply voltages.

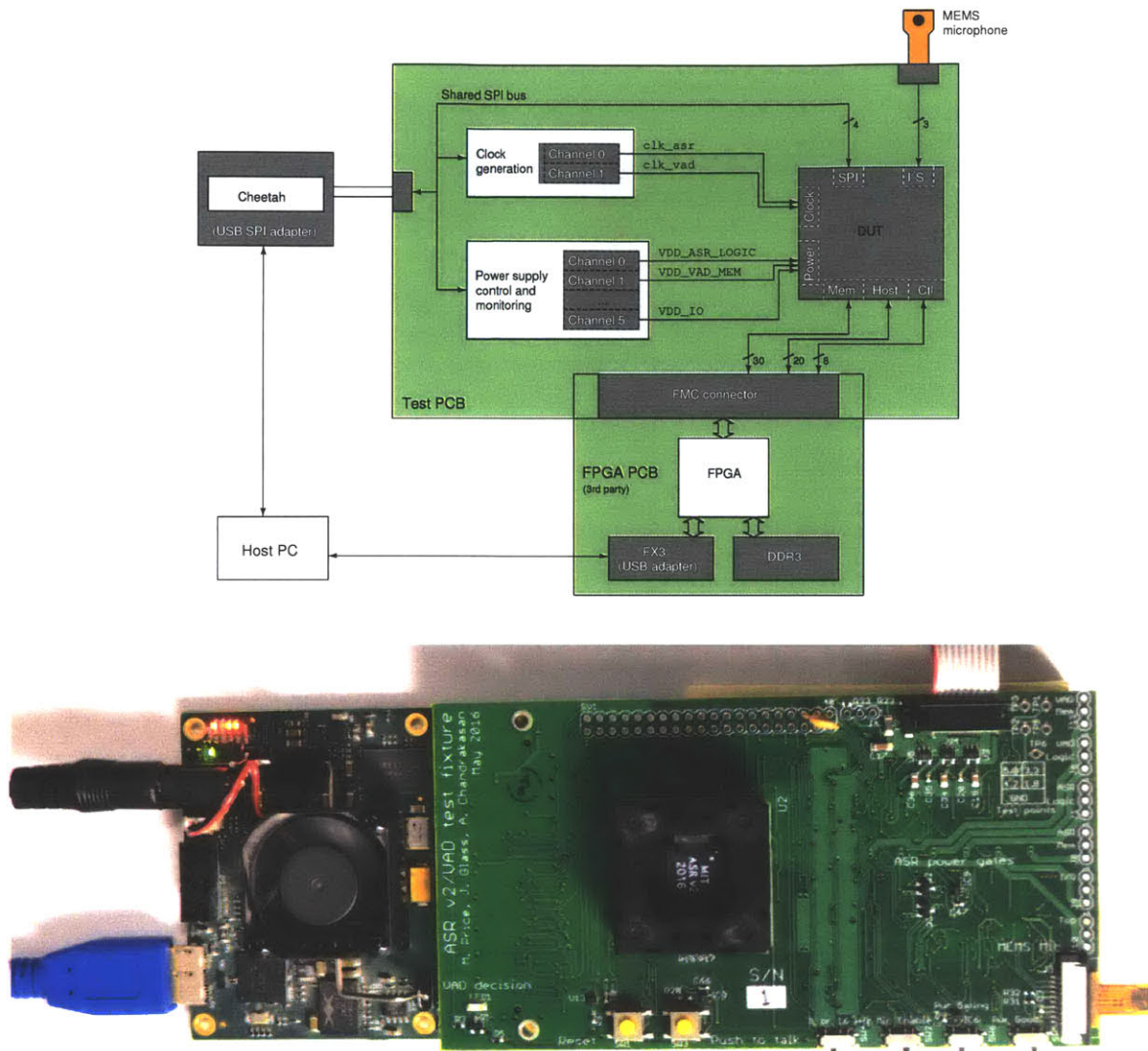


Figure 6-9: Block diagram (top) and photo (bottom) of test PCB stack integrating power supplies and functional test interfaces.

statistics are collected on-chip.) The FPGA is also optionally used to generate a higher frequency ASR clock by multiplying the LTC6903 output (which tops out at 68 MHz) by 2x or 3x.

### 6.3.3 Software

ASR, VAD, and test functionality is exposed through a modular Python API. Programs written to use this API generated all of the measured performance plots you see in this thesis. The software components and key subroutines are illustrated in Figure 6-12.

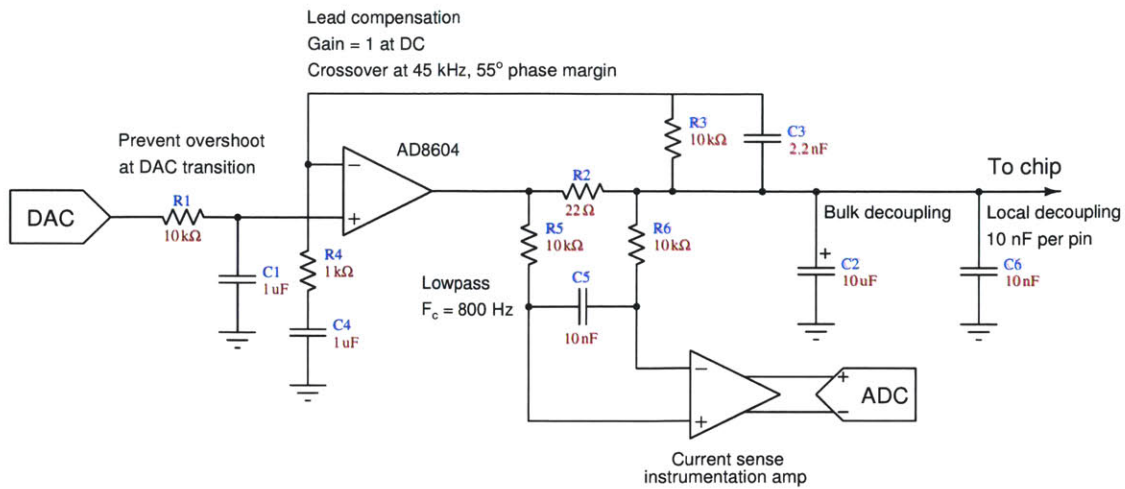


Figure 6-10: Onboard programmable power supply regulator with current sense resistor inside the feedback loop.

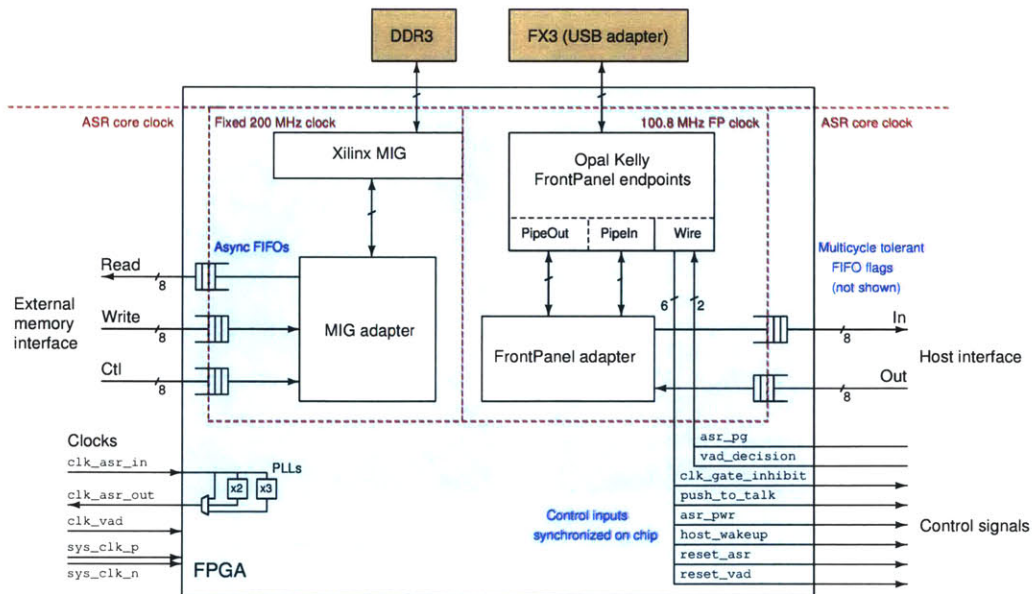


Figure 6-11: FPGA-based adapter for generic host, memory, and control interfaces.

## 6.4 Results

As of this writing, the chip is operational with the same ASR and VAD accuracy as the FPGA prototype that provided results for earlier chapters. This section provides a baseline characterization of the chip, focusing on voltage/frequency scaling and the efficiency of different ASR components. Additional measurements (e.g., more aggressive voltage scaling) will be provided in a future publication.

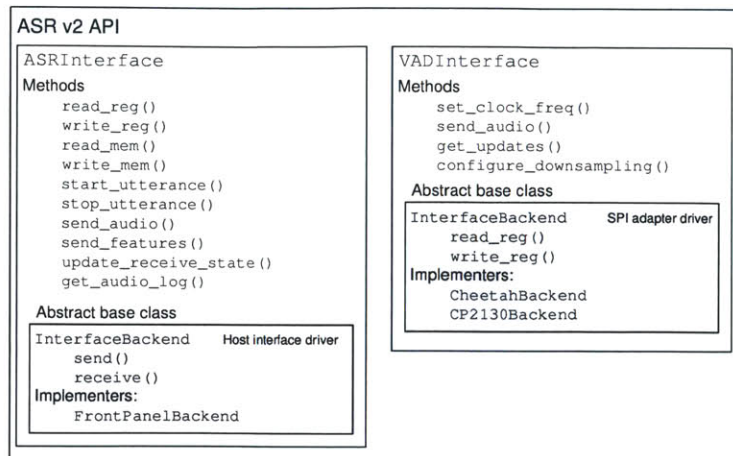


Figure 6-12: Software modules used for ASR/VAD testing and demonstration.

Photos of the bare and packaged silicon die are shown in figure 6-13.

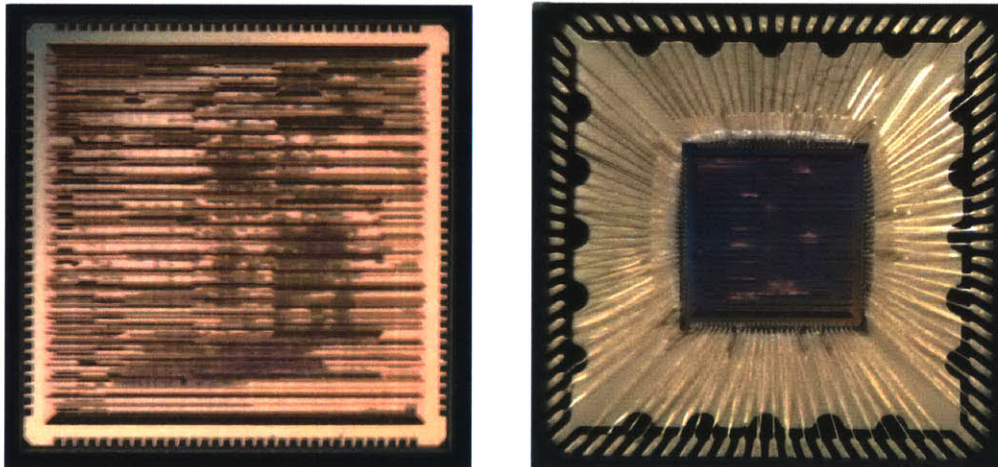


Figure 6-13: ASR/VAD chip photos: bare die (left) and 88-pin QFN package (right).

### 6.4.1 Voltage/frequency scaling

The chip has 6 power supplies, as summarized in Table 6.4.  $V_{IO}$  was fixed at 1.8 V for all tests. The relationship between the five core supply voltages is constrained by level shifters (Section 6.2.2) and timing constraints (Section 6.2.3). For testing the ASR module, the  $V_{VAD\_LOGIC}$  and  $V_{VAD\_MEM}$  supplies were fixed at 0.70 and 0.90 V respectively. We held  $V_{ASR\_MEM}$  at 0.15 V above  $V_{ASR\_LOGIC}$ , up to a limit of 1.2 V.

Sweeping  $V_{ASR\_LOGIC}$  as an independent variable, we identified the maximum ASR clock frequency in

Net Name	Description	Nominal voltage
$V_{IO}$	I/O supply	2.5 V
$V_{TOP}$	Supervisory logic and top-level interfaces	1.2 V
$V_{ASR\_LOGIC}$	Core logic for ASR module	1.2 V
$V_{ASR\_MEM}$	Core SRAM/ROM for ASR module	1.2 V
$V_{VAD\_LOGIC}$	Core logic for VAD module	1.2 V
$V_{VAD\_MEM}$	Core SRAM/ROM for VAD module	1.2 V

Table 6.4: ASR/VAD chip power supplies.

each voltage configuration via binary search. This is shown in Figure 6-14. The primary test (performed at steps of 10 mV) was to correctly decode one utterance using the TIDIGITS digit recognizer (about 4M cycles). Correct ASR functionality was obtained from 0.60 V (10.2 MHz) to 1.20 V (86.8 MHz). Cross-checks with a 5,000 word WSJ recognizer (about 100M cycles) identified very similar maximum frequencies.

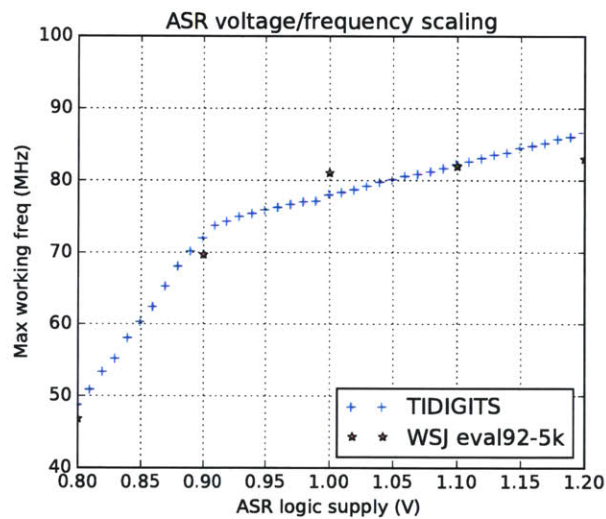


Figure 6-14: Maximum working frequency for ASR as a function of logic supply voltage.

### 6.4.2 ASR power consumption

**Individual utterances:** Figure 6-15 shows current waveforms measured during decoding of one utterance from the nutrition task. The figure includes  $V_{TOP}$ ,  $V_{ASR\_LOGIC}$ , and  $V_{ASR\_MEM}$ ; currents are also measured in real-time for the other three supplies.

Some insight into the chip's operation is apparent from these waveforms:

- The large peaks in logic and memory current come from acoustic model (DNN) evaluation, which is

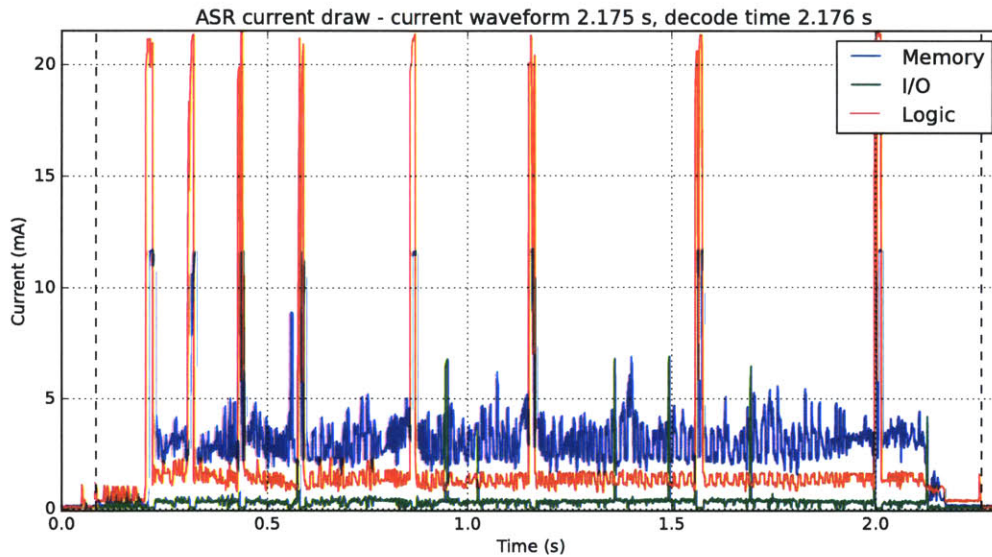


Figure 6-15: Waveforms of selected power supply currents during ASR decoding.

performed on 32 frames in parallel.<sup>7</sup>

- The relatively long periods between acoustic model evaluations are used for search (Viterbi updates), which sequentially processes each of the 32 frames. The search workload varies throughout the utterance, as described in Section 3.2.4.
- The acoustic model is more logic-intensive (due to the large number of multiply/accumulate and sigmoid operations), and search is more memory-intensive (due to the large hash tables used for storing hypotheses).
- Word lattice snapshots (Section 3.2.5) cause 50–60 kB to be written to the external memory, resulting in a spike in I/O power. At all other times the I/O power is relatively low; most output activity comes from requests for external memory reads.

**Task-averaged power:** We evaluated each of our five recognizers on the ASR v2 platform. We have not exhaustively searched for the optimal operating point for each recognizer. Instead, we ran each recognizer, with a beam width of 12, at a sufficiently high clock speed to be faster than real-time (on average). Supply voltages were chosen conservatively. The results are summarized in Table 6.5.

We report the core power consumption (from the  $V_{ASR\_LOGIC}$  and  $V_{ASR\_MEM}$  supplies) and full-chip

<sup>7</sup>In some cases, we observe separate spikes for each of the layers in the DNN. This is easier to see at lower clock frequencies, where we capture more current samples.



power consumption (from all six supplies, including I/O and the unused VAD). I/O power is relatively low (under 1 mW) due to the low clock frequencies, low external memory bandwidth, and read-heavy memory access pattern. Power consumption was averaged over the time used for decoding; because these results are (on average) faster than real-time, average power would be lower if normalized to utterance length.

Task	Clock freq.	$V_{ASR\_LOGIC}$	$V_{ASR\_MEM}$	WER	Runtime	Core power	Chip power
TIDIGITS	1.5 MHz	0.75 V	0.90 V	1.66%	0.92xRT	246 $\mu$ W	326 $\mu$ W
Jupiter	22 MHz	0.75 V	0.90 V	4.38%	0.99xRT	5.30 mW	5.75 mW
Nutrition	48 MHz	0.82 V	0.97 V	8.57%	0.92xRT	4.99 mW	5.97 mW
WSJ-5k	18 MHz	0.75 V	0.90 V	3.12%	0.83xRT	2.57 mW	3.13 mW
WSJ-dev93	48 MHz	0.82 V	0.97 V	8.78%	0.81xRT	10.14 mW	11.27 mW

Table 6.5: Power consumption averaged over each ASR task.

### 6.4.3 ASR subsystem efficiency

The five recognizers we tested represent samples from a large design space. To allow predictions of power consumption on other recognizers, and comparisons with other work, we break out the contributions of acoustic modeling and search with a normalized efficiency metric for each.

**Acoustic modeling:** The chip can evaluate acoustic models (DNNs) with different dimensions and quantization depths. The contribution of the acoustic model to power consumption varies widely, depending on the size of the model (e.g., 64 to 2048 nodes per hidden layer). Because matrix/vector multiplication is the most energy-intensive aspect of DNN evaluation, we normalize the power consumption in terms of energy per neuron weight. (Fully connected DNN models used for ASR typically have between 100k–25M neuron weights, and the model is evaluated against 100 frames per second.) This refers to core power only, and does not include either I/O or external memory power incurred by fetching the parameters.

To estimate acoustic model efficiency, we evaluate the same utterance with two different acoustic models. We choose an utterance from the nutrition test set, and connect the same lexicon and 3-gram language model to both 256x6 and 512x6 NNs. Both NNs use the dense weight storage format for all layers. A narrow beam width of 6.0 is used to minimize the contribution of search. The difference in energy consumption is divided by the difference in the number of neuron weights applied throughout the utterance. An example of this calculation is shown in Table 6.6.

We estimated acoustic modeling efficiency from energy measurements at different operating points, as shown in Figure 6-16. (Clock frequency was scaled to 95% of the maximum working frequency for each

Model size	256x6	512x6
Num. neuron weights	432k	1.66M
Energy	3.00 mJ	10.14 mJ
$\Delta$ Energy	7.14 mJ	
$\Delta$ Neuron weights	1.23M	
Number of frames	220	
Energy per neuron weight	26.5 pJ	

Table 6.6: Estimation of acoustic model efficiency over a single utterance.

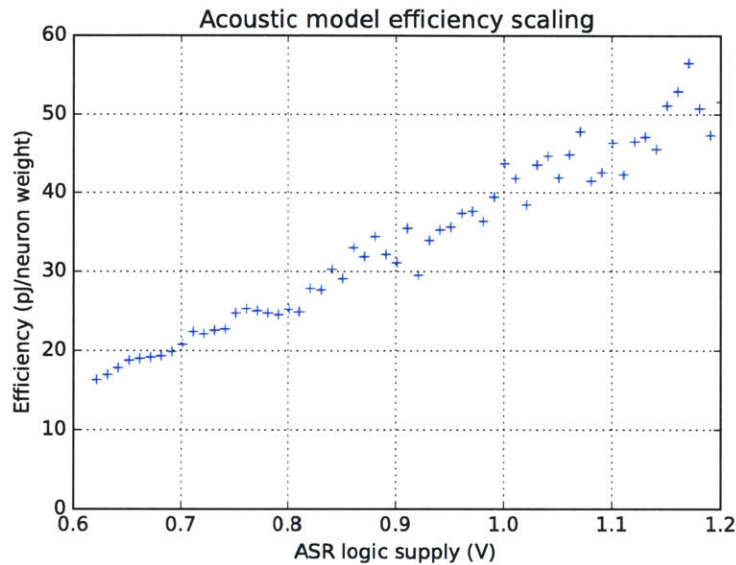


Figure 6-16: Acoustic model efficiency varies by a factor of 3 due to voltage/frequency scaling.

supply voltage.) The efficiency varies from 16 to 56 pJ per neuron weight.<sup>8</sup>

**Search:** Search performance is also scalable. A larger search workload (in terms of the number of hypotheses evaluated) leads to higher power consumption, and usually better ASR accuracy. We define search efficiency in terms of energy per hypothesis.

Estimating search efficiency is straightforward. We vary the beam width, which is a runtime parameter, while repeatedly decoding the same utterance using the same models. Each time, we measure the total core energy consumed, and read the chip’s instrumentation registers to see how many hypotheses were evaluated. Then we construct a linear regression model for energy consumption (Figure 6-17) as a function of the

<sup>8</sup>The measurements are noisier at higher voltages because runtime was very short (and hence the number of samples in the current waveform was relatively small); this would probably be addressed by averaging over more frames of test data.

number of hypotheses. This assumes that the energy is linearly dependent on the number of hypotheses; fortunately, it is ( $r^2 = 0.99$ ).

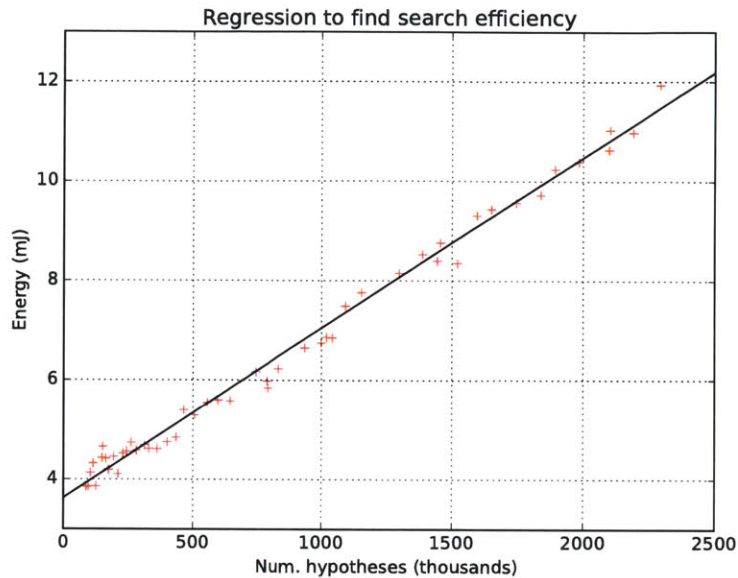


Figure 6-17: Regression to estimate search efficiency from decoding results at different beam widths.

Our estimate for search efficiency is the slope of the regression line (in this example, 3.4 nJ). As with the acoustic model, we evaluated search efficiency across voltage/frequency operating points (Figure 6-18); this gives us a range of 2.5–6.3 nJ per hypothesis.<sup>9</sup> The quadratic trend flattens out around 1.05 V because most of the search-related power is drawn by memory, and the memory supply is 0.15 V higher than logic (up to a limit of 1.2 V).

#### 6.4.4 Explicit clock gating

The measurements we presented above were performed with both explicit and implicit clock gating enabled. (Explicit clock gating was added manually in the RTL-level design, and implicit clock gating was added automatically by the synthesis tool. See Section 6.2.1.) The explicit clock gating can be disabled using a `clk_gate_inhibit` control input, allowing us to see its impact on power consumption in different scenarios.

Explicit clock gating reduces power consumption in both idle and active conditions. It is worth noting

<sup>9</sup>While this compares favorably to the 16 nJ/hypothesis measured in [67], it is important to note that [67] included the energy of on-demand acoustic model evaluation.

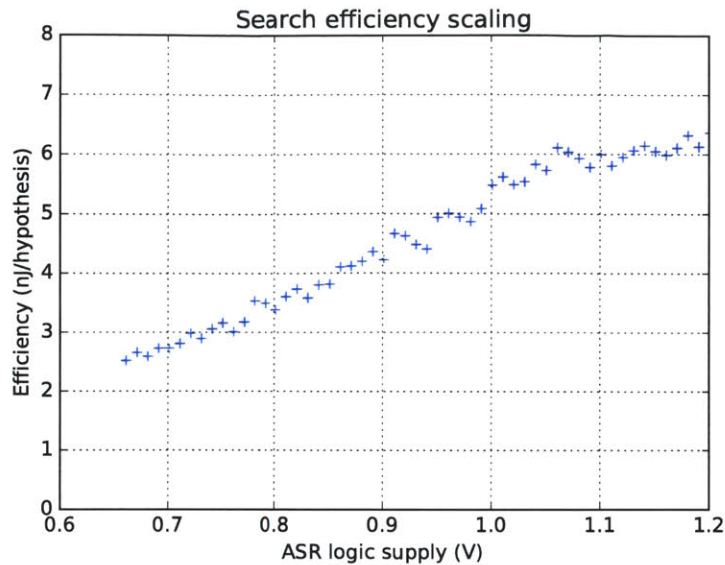


Figure 6-18: Search efficiency scales quadratically with the memory supply voltage.

that the ASR module is normally off (when there is no speech input), and when it is off the clock input can be disabled along with the ASR power supplies. However, during real-time ASR there are idle periods when we are waiting for more frames in order to perform the next acoustic model evaluation. (This will happen unless the clock frequency is not fast enough to keep up with real-time input.) To illustrate the effect, Figure 6-19 shows the average core power for one utterance from the WSJ eval92-5k task with the audio supplied in real-time. Across this range of clock frequencies (20–80 MHz), there is no significant difference in decoding time; latency at the end of the 5.72-second utterance ranged from 215 ms (80 MHz) to 372 ms (20 MHz).

This data suggests that when explicit clock gating is disabled, the clock tree can be responsible for a majority of the chip’s power consumption.<sup>10</sup> Even at 20 MHz (where there are few idle cycles), explicit clock gating reduces core power by up to 40%. In addition, explicit clock gating significantly decreases the impact of clock frequency on power consumption.

### 6.4.5 VAD

The VAD’s behavior is relatively simple because its runtime does not depend on the input data.<sup>11</sup> Table 6.7 shows initial measurements of the VAD, with the clock frequency set for real-time operation with each algorithm. “Core power” is the average power drawn from the  $V_{\text{VAD\_LOGIC}}$  and  $V_{\text{VAD\_MEM}}$  supplies, and

<sup>10</sup>This is also visible in the supply current waveforms (not shown).

<sup>11</sup>It is possible that power consumption will depend on the input data, but we expect this relationship to be weak.

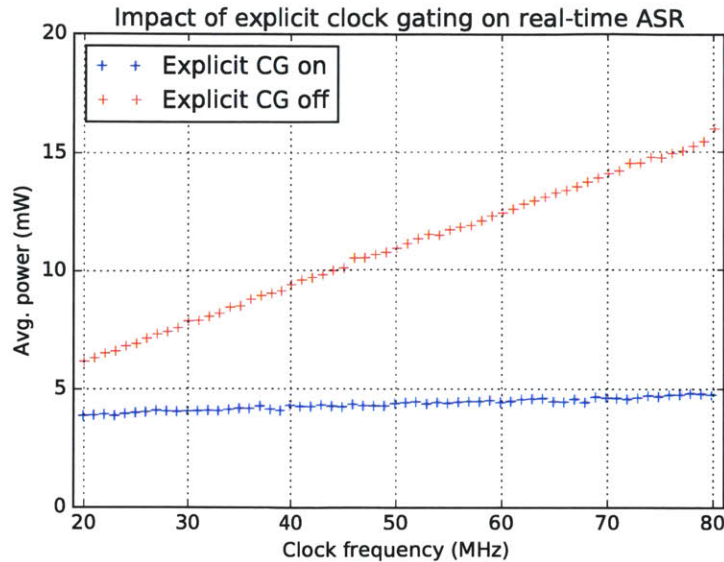


Figure 6-19: Power consumption with and without explicit clock gating.

Algorithm	Clock freq.	$V_{VAD\_LOGIC}$	$V_{VAD\_MEM}$	Core power	Chip power
<b>EB</b> : Energy-based	0.8 MHz	0.53 V	0.70 V	21.1 $\mu$ W	299.0 $\mu$ W
<b>HM</b> : Harmonicity	2.0 MHz	0.53 V	0.70 V	87.7 $\mu$ W	240.7 $\mu$ W
<b>MF</b> : Modulation frequencies	1.8 MHz	0.53 V	0.70 V	54.5 $\mu$ W	187.2 $\mu$ W
All 3 algorithms	3.4 MHz	0.53 V	0.70 V	122.0 $\mu$ W	268.1 $\mu$ W

Table 6.7: Initial measurements of VAD power consumption.

“chip power” is the average power drawn from all 6 supplies (including I/O and the unused ASR module) over a 20-second waveform. The ASR module was not power gated for this test; its logic and memory supplies were set to 0.65 V and 0.80 V respectively.

These early measurements should be interpreted with caution: power supply monitoring was not carefully calibrated (except for a straightforward offset subtraction), the 22  $\Omega$  current sense resistance resulted in low sensitivity (1.7  $\mu$ A per LSB), and I/O power was influenced by the shared SPI bus. We did not optimize the power supply voltages used in each configuration. Despite these limitations, the measurements are consistent with our gate-level simulation results: the MF algorithm requires less power than the HM algorithm because it is only performing one short-term FFT per frame.

Tables 6.5 and 6.7 showed the power consumed by the VAD and ASR individually; we do not report combined VAD/ASR power because we did not evaluate the VAD and ASR with a single (common) set of waveforms. The power consumption of an “always-on” system (using VAD to activate ASR) is ultimately

task-dependent, but it can be predicted using our system power model (Section 4.1) and the ROC of the selected VAD algorithm on the desired task.

## 6.5 Summary

This chapter provided an overview of the concepts needed to prepare and evaluate an IC implementation of the ASR and VAD designs. The physical design and verification flows were extended to handle the complexity of the design and the difficulty of performing thorough simulations and formal verification. We described the voltage and timing constraints needed to support multi-voltage operation with a total of six power supplies and four clock domains. The goal was to minimize power consumption by operating logic at the lowest possible voltage, separate from the high-density SRAM arrays. We also presented key elements of board-level, FPGA, and software design to exercise the functionality of the chip for testing and demonstration purposes. Measurements showed that the chip has similar power consumption to [67] under heavy workloads, but scales to lower voltages and achieves equivalent ASR performance at lower clock frequencies.

# Chapter 7

## Conclusion

The preceding chapters described how and why we designed an accurate, programmable, and scalable ASR/VAD chip. We conclude with a summary of what we achieved and a discussion of what comes next.

### 7.1 Summary

Designers of complex systems have many choices for improving energy efficiency. For digital IC designers, the most accessible levers include voltage/frequency scaling, clock gating, and improved process technology. These are general techniques that can be applied to any computational problem. But as [73] wisely observed, the best place to start looking for efficiency improvements is in the algorithms. Algorithm developers don't always care about efficiency, and may not be aware of the constraints and opportunities of specialized architectures. There is often a tradeoff between performance and computational complexity that remains unknown until someone investigates. Power and area savings from simplified or optimized algorithms are complementary (sometimes multiplicative) with architectural or circuit techniques.

#### 7.1.1 Contributions

We demonstrated that in a portable or embedded system, the entire speech decoding chain from audio to text can be performed on a single chip without a general-purpose processor. To achieve good energy efficiency and scalability, our ASR and VAD designs draw techniques from the disciplines of algorithms, architectures, and circuits.

### Algorithms:

- **Acoustic model selection** (Section 2.2.2)  
The designer can rapidly prototype DNNs in Kaldi and select the smallest model that provides acceptable accuracy for each task.
- **Model compression** (Sections 2.1 and 3.2.3)  
Reduces memory bandwidth and provides accuracy/bandwidth tradeoffs to the designer.
- **Word lattice** (Section 3.2.5)  
Reduces or eliminates memory writes required by search.
- **Predictive beam search** (Section 3.2.4)  
Improves accuracy over previous hardware-oriented pruning approaches.
- **MF/NN VAD** (Section 4.2.3)  
Achieves good accuracy and noise robustness with a model small enough to be stored on-chip.

### Architectures:

- **WFST caching** (Section 3.2.3)  
Reduces memory bandwidth and page access penalties.
- **Resizable hash tables** (Section 3.2.1)  
Improves scalability by requiring fewer clock cycles for light search workloads.
- **Parallel NN evaluation** (Section 2.2.1)  
Reduces memory bandwidth (and to a lesser extent, reduces core power).
- **Two-frame state list** (Section 3.2.2)  
Reduces SRAM area.
- **VAD-based power gating** (Section 5.1.3)  
Reduces time-averaged power consumption; audio buffering by VAD prevents front-end clipping.

### Circuits:

- **Multi-voltage design** (Section 6.2.2)  
Allows lower operating voltages for logic, while using high-density SRAM.
- **Explicit clock gating** (Section 6.2.1)  
Reduces power by identifying additional gating opportunities at higher levels of the clock tree.
- **Variation-tolerant timing** (Section 6.2.3)  
Allows lower operating voltages.



### 7.1.2 Performance and scalability

The decoding performance of our ASR engine on several tasks is summarized in Table 7.1.<sup>1</sup> Comparing with our previous work [67] on the WSJ eval92-5k task, WER went from 13.0% to 2.91%, a 77% reduction, while external memory bandwidth went from 61.6 MB/s to 4.84 MB/s, a 92% reduction. From 11 words to 145k words, runtime scales by a factor of 51x and external memory bandwidth scales by 136x. We believe this scalability is essential to make a single architecture useful in a variety of application scenarios.

<b>Task</b>	<b>Vocabulary</b>	<b>Clock freq (1xRT est.)</b>	<b>Memory BW</b>	<b>WER</b>
TIDIGITS	11 words	0.76 MHz	0.11 MB/s	1.65%
Jupiter	2k words	20.3 MHz	10.1 MB/s	4.38%
Nutrition	7k words	38.6 MHz	9.02 MB/s	8.51%
WSJ (eval92-5k)	5k words	18.6 MHz	4.84 MB/s	2.91%
WSJ (dev93)	145k words	29.8 MHz	15.0 MB/s	9.04%

Table 7.1: ASR performance summary.

The detection performance of our VAD engine on several tasks is summarized in Table 7.2. The MF algorithm achieves 10% frame-level EER at SNRs about 10 dB lower than an energy-based algorithm. We believe this robustness is essential to minimize average power consumption when the draw of the downstream system is taken into account.

<b>Algorithm</b>	<b>SNR for 10% frame level EER</b>		
	White noise	Aurora2	Forklift/Switchboard
Energy-based	-1 dB	18 dB	Fail
Harmonicity	< -5 dB	5 dB	Fail
Modulation frequencies	-2 dB	7 dB	1 dB

Table 7.2: VAD performance summary.

## 7.2 Future work

To some extent, the ideas developed in this thesis are ready for commercial development. There are probably some products for which the performance and system requirements (power, area, and external components) of our WFST/DNN recognizer are manageable and better than existing alternatives. Our prototype was made with tools and technologies readily available to industry: 65 nm CMOS with off-the-shelf IP.

<sup>1</sup>The system is in some cases capable of higher accuracy by setting a higher beam width.

New applications require continued improvements in performance, flexibility, and integration. We envision a single-chip solution to a variety of speech processing problems (Figure 7-1), with enough scalability to be reused in applications from wearables to servers. Realizing this vision requires expanding and refining the set of algorithms we implemented, integrating additional components on-chip, and leveraging technological changes that are affecting the electronics and software industries.

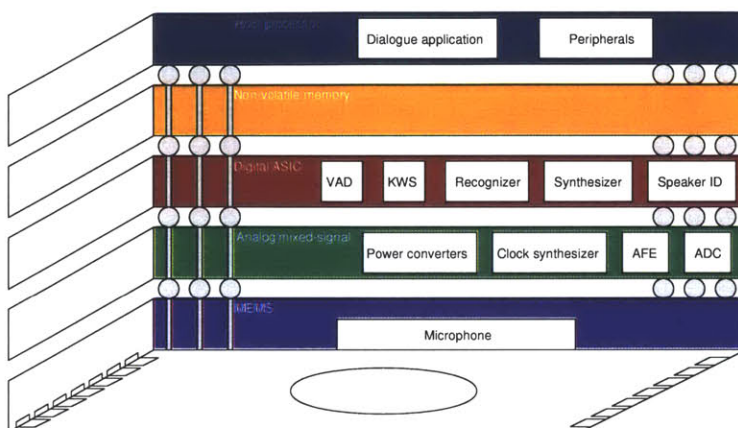


Figure 7-1: Vision for single-chip speech processing solution.

### 7.2.1 Design and application refinement

Within the framework we have established, there are several ways to extend the performance, usability, and efficiency of hardware speech interfaces. We discuss some of these opportunities briefly:

- Improved acoustic models:** Speech recognizers trained with large datasets can benefit from recurrent NN topologies, including variants of the LSTM cell [77]. It is possible to add hardware support for these more complex models. This poses a challenge since the time dependence of recurrent layers prohibits simple parallel evaluation. Time delay neural networks (TDNN [62]) and convolutional neural networks (CNN) may also be useful.
- Better model compression:** Minimizing the size of neural network models is important for many applications; we should incorporate techniques developed for other disciplines such as computer vision [28] to further reduce memory bandwidth and power consumption. There are also promising results from neural networks represented in a logarithmic number system [50], improving compression and simplifying model evaluation.

- **On-chip model storage:** For small tasks (on the order of 100 words), it may be possible to reduce external bandwidth by providing sufficient DNN and WFST model storage on-chip. This would simplify system design and reduce power consumption. Larger models could be supported if embedded DRAM is available. The models would still have to be loaded once at the beginning of each utterance, unless the on-chip memory is non-volatile or not power gated.
- **Dynamic WFST composition:** Large language models lead to large WFSTs that degrade caching performance. To reduce memory requirements, some software ASR systems store the language model  $G$  separately from the other portions ( $HCL$ ) of the WFST, and compose them on-the-fly [2]. This arrangement could be beneficial to a hardware ASR system, though it would significantly increase decoder complexity.
- **Increased search throughput and efficiency:** As mentioned in Section 3.4, the search subsystem handles about 1 hypothesis every 40 clock cycles. Improving the throughput (say, to 10–20 cycles per hypothesis) through more aggressive RTL-level design would allow significant reductions in clock frequency and normalized power consumption. Separate storage for the  $\epsilon$ -input WFST arcs could help, as the current system reads all arcs from each active state in both the non- $\epsilon$  and  $\epsilon$  phases of search. Many states have no outgoing  $\epsilon$ -input arcs and could be skipped entirely during the  $\epsilon$  phase.
- **Power reduction in state list:** Our state list performs an “advance” operation which moves all entries from the destination frame to the source frame. These transactions could be avoided by allowing the source and destination fields to be stored in either order, tracking this ordering in a separate array. This is a minor change that could have a significant power impact, given that the state list occupies 1.82 Mb of SRAM. It would also be possible to use a smaller SRAM or register array as a cache for the larger state list.
- **Alternative wake-up methods:** Our VAD and ASR architectures could be repurposed for keyword spotting (KWS) and speaker ID, but it would make more sense to add circuits dedicated to more suitable algorithms. For example, speaker ID based on the i-vector approach [21] classifies a single vector per utterance (rather than every frame). KWS may be possible with the MF VAD, but would benefit from simple time-dependent modeling (simpler than a full ASR HMM).
- **Audio event detection:** It may be possible to use our MF feature classification technique to label audio segments with classes other than speech and non-speech. If the architecture is changed to allow more than 2 NN outputs, it could detect certain outdoor sounds (vehicles, tools, animals, weather) or

indoor sounds (toilets, falls, fires, appliances) with the appropriate training data. (We don't know how effective the MF representation will be for these problems.)

- **VAD power reduction:** Because our VAD doesn't rely on an external memory, its power consumption could be reduced via advanced process technology. Furthermore, most of its power consumption comes from the FFT logic, which we neglected to optimize; techniques such as split-radix FFTs [68] and data gating [95] could drive down this component. The microphone and its analog front-end would also have to be carefully optimized to drive system power below 100  $\mu$ W. More efficient VADs could also be designed using a hybrid analog/digital approach using analog filters, avoiding the need for an FFT and reducing ADC sample rates [32, 6].
- **Multi-stage VAD:** We showed that there was a general tradeoff between complexity (including circuit power and area) and accuracy among VAD algorithms. It may be possible to combine the outputs of different algorithms in a way that exceeds the performance of any one algorithm. To reduce power, such an approach would need to selectively enable algorithms (for example, an MF VAD that is enabled only when an energy-based VAD detects a signal).
- **Joint ASR/VAD evaluation:** We evaluated ASR and VAD algorithms on disjoint data sets. Using a task designed to exercise VAD and ASR jointly would allow the system power model of Section 4.1 to be tested experimentally. This could be done using a dataset designed for noise-robust ASR such as Aurora4 [35] or one of the CHiME challenges [8].
- **Verification improvements:** The testing framework of Section 5.2.2 was helpful for verifying the ASR and VAD, and can be applied to other digital designs. It can be made more effective by improving its generality, and adding interfaces to other programming languages besides SystemVerilog and C++.

### 7.2.2 System integration

We chose to focus on signal processing aspects of speech interfaces in a purely digital ASIC environment. In light of reduced memory burden and increased chip complexity, we could take advantage of area and efficiency gains from process shrinks (to 28 nm or smaller, if the cost is acceptable). CMOS scaling also provides the opportunity for non-digital components to go on the same chip: clock generation, microphone front-end and ADC, power conversion, and wireless interfaces. Demonstrating this level of integration would require an interdisciplinary mixed-signal design effort, but could lead to products in the form of a

single-package “smart microphone.” This aspect of our vision may be more suited to a commercial development effort than academic research.

### 7.2.3 Leveraging technological change

In both the hardware and software realms, the ground is shifting beneath our feet. Future efforts in hardware-accelerated speech processing will be more productive if they find a way to leverage developments in memory technology and ASR algorithms.

On the hardware side, research and development across the industry promises to deliver non-volatile memories with speed and efficiency approaching DRAM [94, 87]. This would present an even larger opportunity than continued CMOS scaling. We have designed our ASR to work with NAND flash memory;<sup>2</sup> a faster and lower-power memory would allow lower clock frequencies, significantly reduce system power, and allow more complex tasks to be run in real-time. I/O signaling power could become a significant issue, leading to multi-chip packaging and 2.5D or 3D integration.

On the software side, some engineers would like to abandon the HMM framework for ASR and replace it with a more general sequence-to-sequence translation framework, probably based on neural networks. This would eliminate much of the manual effort from creating speech recognizers, especially for less commonly used languages and dialects. It may ultimately result in better accuracy as well. Promising, though not state-of-the-art, results have been obtained by Listen, Attend, and Spell [13] and Eesen [53].

A hardware-accelerated version of one of these recognizers would not need a front-end, and could markedly simplify the Viterbi search architecture we have developed. Most resources would be devoted to NN evaluation, which could present an even larger memory bottleneck than we observed with HMM recognizers. Overall, it would look very different from the system presented in this thesis. This line of research should be watched carefully.

---

<sup>2</sup>You design a system with the memory technology you have, not the memory technology you might want or wish to have at a later time. (With apologies to the former US Secretary of Defense, Donald Rumsfeld.)



# Bibliography

- [1] C. Allauzen, M. Mohri, M. Riley, and B. Roark. A generalized construction of integrated speech recognition transducers. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 1, pages I-761-4 vol.1, 2004.
- [2] Cyril Allauzen, Michael Riley, and Johan Schalkwyk. A generalized composition algorithm for weighted finite-state transducers. In *INTERSPEECH*, pages 1203-1206, 2009.
- [3] L. M. Aubert, R. Woods, S. Fischhaber, and R. Veitch. Optimization of weighted finite state transducer for speech recognition. *IEEE Transactions on Computers*, 62(8):1607-1615, Aug 2013.
- [4] K. M. H. Badami, S. Lauwereins, W. Meert, and M. Verhelst. A 90 nm CMOS, 6  $\mu$ W power-proportional acoustic sensing frontend for voice activity detection. *IEEE Journal of Solid-State Circuits*, 51(1):291-302, Jan 2016.
- [5] Komail Badami, Steven Lauwereins, Wannas Meert, and Marian Verhelst. 24.2 Context-aware hierarchical information-sensing in a 6 $\mu$ W 90nm CMOS voice activity detector. In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, pages 1-3. IEEE, 2015.
- [6] M.W. Baker, T.K.-T. Lu, C.D. Salthouse, Ji-Jon Sit, S. Zhak, and R. Sarpeshkar. A 16-channel analog VLSI processor for bionic ears and speech-recognition front ends. In *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, pages 521-526, 2003.
- [7] O. A. Bapat, P. D. Franzon, and R. M. Fastow. A generic and scalable architecture for a large acoustic model and large vocabulary speech recognition accelerator using logic on memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2701-2712, Dec 2014.
- [8] Jon Barker, Ricard Marxer, Emmanuel Vincent, and Shinji Watanabe. The third CHiME speech separation and recognition challenge: dataset, task and baselines. In *2015 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU 2015)*, 2015.
- [9] E. Bocchieri and D. Blewett. A decoder for LVCSR based on fixed-point arithmetic. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, pages I-I, May 2006.
- [10] Paul Boersma. Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. In *Proceedings of the institute of phonetic sciences*, volume 17, pages 97-110. Amsterdam, 1993.
- [11] Patrick J. Bourke and Rob A. Rutenbar. A high-performance hardware speech recognition system for mobile applications. In *Proceedings of Semiconductor Research Corporation TECHCON*, August 2005.

- [12] Andrew Burstein. *Speech recognition for portable multimedia terminals*. PhD thesis, University of California, Berkeley, CA, USA, 1996.
- [13] William Chan, Navdeep Jaitly, Quoc V Le, and Oriol Vinyals. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.
- [14] Guoguo Chen, C. Parada, and G. Heigold. Small-footprint keyword spotting using deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 4087–4091, May 2014.
- [15] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, Jan 2016.
- [16] Jungwook Choi, Kisun You, and Wonyong Sung. An FPGA implementation of speech recognition with weighted finite state transducers. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 1602–1605, march 2010.
- [17] Youngkyu Choi, Kisun You, Jungwook Choi, and Wonyong Sung. A real-time FPGA-based 20,000-word speech recognizer with optimized DRAM access. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 57(8):2119–2131, aug. 2010.
- [18] Wanming Chu and Yamin Li. Cost/performance tradeoff of n-select square root implementations. In *Computer Architecture Conference, 2000. ACAC 2000. 5th Australasian*, pages 9–16. IEEE, 2000.
- [19] Ekapol Chuangsuwanich and James R Glass. Robust voice activity detector for real world applications using harmonicity and modulation frequency. In *INTERSPEECH*, pages 2645–2648. Citeseer, 2011.
- [20] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, Aug 1980.
- [21] N. Dehak, P. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet. Front-end factor analysis for speaker verification. *Audio, Speech, and Language Processing, IEEE Transactions on*, 19(4):788–798, 2011.
- [22] B DeLugish. *A class of algorithms for automatic evaluation of functions and computations in a digital computer*. PhD thesis, PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, 1970.
- [23] Shiwen Deng, Jiqing Han, Tieran Zheng, and Guibin Zheng. A modified MAP criterion based on hidden Markov model for voice activity detection. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5220–5223. IEEE, 2011.
- [24] Yariv Ephraim and David Malah. Speech enhancement using a minimum-mean square error short-time spectral amplitude estimator. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 32(6):1109–1121, 1984.
- [25] John J Godfrey, Edward C Holliman, and Jane McDaniel. SWITCHBOARD: Telephone speech corpus for research and development. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 1, pages 517–520. IEEE, 1992.
- [26] Joshua T Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.



- [27] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.
- [28] Song Han, Huizi Mao, and William J Dally. Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [29] Guangji He, Y. Miyamoto, K. Matsuda, S. Izumi, H. Kawaguchi, and M. Yoshimoto. A 40-nm 54-mW 3x-real-time VLSI processor for 60-kWord continuous speech recognition. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 147–152, Oct 2013.
- [30] Guangji He, T. Sugahara, S. Izumi, H. Kawaguchi, and M. Yoshimoto. A 40-nm 168-mW 2.4x-real-time VLSI processor for 60-kWord continuous speech recognition. In *Custom Integrated Circuits Conference (CICC), 2012 IEEE*, pages 1–4, 2012.
- [31] Guangji He, T. Sugahara, Y. Miyamoto, T. Fujinaga, H. Noguchi, S. Izumi, H. Kawaguchi, and M. Yoshimoto. A 40 nm 144 mW VLSI processor for real-time 60-kWord continuous speech recognition. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 59(8):1656–1666, 2012.
- [32] Qing He. *An architecture for low-power voice-command recognition systems*. PhD thesis, Massachusetts Institute of Technology, 2016.
- [33] I. Lee Hetherington. PocketSUMMIT: small-footprint continuous speech recognition. In *INTERSPEECH-2007*, pages 1465–1468, 2007.
- [34] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.
- [35] Guenter Hirsch. Experimental framework for the performance evaluation of speech recognition front-ends on a large vocabulary task. *ETSI STQ Aurora DSR Working Group*, 2002.
- [36] Hans-Günter Hirsch and David Pearce. The Aurora experimental framework for the performance evaluation of speech recognition systems under noisy conditions. In *ASR2000-Automatic Speech Recognition: Challenges for the new Millenium ISCA Tutorial and Research Workshop (ITRW)*, 2000.
- [37] Xuedong Huang, Alex Acero, and Hsiao-Wuen Hon. *Spoken language processing: A guide to theory, algorithm, and system development*. Prentice Hall PTR, 2001.
- [38] Accellera Systems Initiative et al. Universal verification methodology (uvm). Available online: <http://www.accellera.org/community/uvm/>.
- [39] Frederick Jelinek. *Statistical methods for speech recognition*. MIT Press, 1997.
- [40] Jong-Chul Jeong, Woo-Chan Park, Woong Jeong, Tack-Don Han, and Moon-Key Lee. A cost-effective pipelined divider with a small lookup table. *IEEE Transactions on Computers*, (4):489–495, 2004.
- [41] J.R. Johnston and R.A. Rutenbar. A high-rate, low-power, ASIC speech decoder using finite state transducers. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 77–85, 2012.

- [42] Kate M Knill, M Gales, and Steve J Young. Use of Gaussian selection in large vocabulary continuous speech recognition using HMMs. In *Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on*, volume 1, pages 470–473. IEEE, 1996.
- [43] Mandy Korpusik, Calvin Huang, Michael Price, and James Glass. Distributional semantics for understanding spoken meal descriptions. To be presented at ICASSP 2016.
- [44] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, 2013.
- [45] R Gary Leonard. A database for speaker-independent digit recognition. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 328–331. IEEE, 1984.
- [46] Edward C. Lin and Rob A. Rutenbar. A multi-fpga 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 83–92, New York, NY, USA, 2009. ACM.
- [47] Edward C Lin, Kai Yu, R Rutenbar, and T Chen. In silico vox: Towards speech recognition in silicon. In *HOTCHIPS*, 2006.
- [48] Edward C. Lin, Kai Yu, Rob A. Rutenbar, and Tsuhan Chen. Moving speech recognition from software to silicon: the In Silico Vox project. In *INTERSPEECH-2006*, 2006.
- [49] Edward C. Lin, Kai Yu, Rob A. Rutenbar, and Tsuhan Chen. A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, FPGA '07, pages 60–68, New York, NY, USA, 2007. ACM.
- [50] Leo Liu. Acoustic models for speech recognition using deep neural networks based on approximate math. Master's thesis, Massachusetts Institute of Technology, 2015.
- [51] Rainer Martin. Noise power spectral density estimation based on optimal smoothing and minimum statistics. *Speech and Audio Processing, IEEE Transactions on*, 9(5):504–512, 2001.
- [52] Joel Max. Quantizing for minimum distortion. *Information Theory, IRE Transactions on*, 6(1):7–12, 1960.
- [53] Yajie Miao, Mohammad Gowayyed, and Florian Metze. EESEN: End-to-end speech recognition using deep RNN models and WFST-based decoding. *arXiv preprint arXiv:1507.08240*, 2015.
- [54] Mehryar Mohri. Weighted finite-state transducer algorithms. An overview. In *Formal Languages and Applications*, pages 551–563. Springer, 2004.
- [55] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook on Speech Processing and Speech Communication*. 2008.
- [56] Preetum Nakkiran, Raziq Alvarez, Rohit Prabhavalkar, and Carolina Parada. Compressing deep neural networks using a rank-constrained topology. In *Proceedings of Annual Conference of the International Speech Communication Association (Interspeech)*, pages 1473–1477, 2015.
- [57] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

- 
- [58] G. Panneerselvam, G.A Jullien, S. Bandyopadhyay, and W.C. Miller. Reconfigurable systolic architectures for hashing. In *Databases, Parallel Architectures and Their Applications, PARBASE-90, International Conference on*, pages 543–, Mar 1990.
- [59] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. J. Yoo. A 1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. In *Solid- State Circuits Conference - (ISSCC), 2015 IEEE International*, pages 1–3, Feb 2015.
- [60] Douglas B. Paul and Janet M. Baker. The design for the Wall Street Journal-based CSR corpus. In *Proceedings of the workshop on Speech and Natural Language, HLT '91*, pages 357–362, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.
- [61] U. Pazhayaveetil, D. Chandra, and P. Franzon. Flexible low power probability density estimation unit for speech recognition. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1117–1120, May 2007.
- [62] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. A time delay neural network architecture for efficient modeling of long temporal contexts. In *Proceedings of INTERSPEECH*, 2015.
- [63] D. Povey, L. Burget, M. Agarwal, P. Akyazi, Kai Feng, A. Ghoshal, O. Glembek, N.K. Goel, M. Karafiat, A. Rastrow, R.C. Rose, P. Schwarz, and S. Thomas. Subspace Gaussian mixture models for speech recognition. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4330–4333, March 2010.
- [64] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi Speech Recognition Toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. IEEE Catalog No.: CFP11SRW-USB.
- [65] Daniel Povey et al. Decoders used in the Kaldi toolkit (Kaldi documentation). Available online: <http://kaldi-asr.org/doc/decoders.html>.
- [66] M. Price, J. Glass, and A.P. Chandrakasan. A 6mW 5K-Word real-time speech recognizer using WFST models. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 454–455, Feb 2014.
- [67] M. Price, J. Glass, and A.P. Chandrakasan. A 6 mW, 5,000-Word Real-Time Speech Recognizer Using WFST Models. *Solid-State Circuits, IEEE Journal of*, 50(1):102–112, Jan 2015.
- [68] Zhuo Qian and Martin Margala. A novel low-power and in-place split-radix FFT processor. In *Proceedings of the 24th edition of the great lakes symposium on VLSI*, pages 81–82. ACM, 2014.
- [69] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits: a design perspective*. Prentice Hall, 2 edition, 2002.
- [70] Javier Ramírez, José C Segura, Carmen Benítez, Luz García, and Antonio Rubio. Statistical voice activity detection using a multiple observation likelihood ratio test. *Signal Processing Letters, IEEE*, 12(10):689–692, 2005.
- [71] A. Raychowdhury, C. Tokunaga, W. Beltman, M. Deisher, J. W. Tschanz, and V. De. A 2.3 nJ/frame voice activity detector-based audio front-end for context-aware system-on-chip applications in 32-nm CMOS. *IEEE Journal of Solid-State Circuits*, 48(8):1963–1969, Aug 2013.

- [72] Michael Riley, Andrej Ljolje, Donald Hindle, and Fernando Pereira. The AT&T 60,000 word speech-to-text system. In *Fourth European Conference on Speech Communication and Technology*, 1995.
- [73] Rahul Jagdish Rithe. *Energy-efficient system design for mobile processing platforms*. PhD thesis, Massachusetts Institute of Technology, 2014.
- [74] T.N. Sainath, O. Vinyals, A. Senior, and H. Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4580–4584, April 2015.
- [75] George Saon, Daniel Povey, and Geoffrey Zweig. Anatomy of an extremely fast LVCSR decoder. In *INTERSPEECH*, pages 549–552, 2005.
- [76] George Saon, Samuel Thomas, Hagen Soltau, Sriram Ganapathy, and Brian Kingsbury. The IBM speech activity detection system for the DARPA RATS program. In *INTERSPEECH*, pages 3497–3501. Citeseer, 2013.
- [77] Andrew Senior, Hasim Sak, Felix de Chaumont Quitry, Tara N. Sainath, and Kanishka Rao. Acoustic modelling with CD-CTC-SMBR LSTM RNNs. In *ASRU*, 2015.
- [78] David Sheffield, Michael J Anderson, Yunsup Lee, and Kurt Keutzer. Hardware/software codesign for mobile speech recognition. In *INTERSPEECH*, pages 627–631, 2013.
- [79] Jongseo Sohn, Nam Soo Kim, and Wonyong Sung. A statistical model-based voice activity detection. *Signal Processing Letters, IEEE*, 6(1):1–3, 1999.
- [80] A. Stolzle, S. Narayanaswamy, H. Murveit, J.M. Rabaey, and R.W. Brodersen. Integrated circuits for a real-time large-vocabulary continuous speech recognition system. *Solid-State Circuits, IEEE Journal of*, 26(1):2–11, jan 1991.
- [81] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. *arXiv preprint arXiv:1511.06488*, 2015.
- [82] Lee Ngee Tan, Bengt J Borgstrom, and Abeer Alwan. Voice activity detection using harmonic frequency components in likelihood ratio test. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4466–4469. IEEE, 2010.
- [83] Zheng-Hua Tan and Børge Lindberg. Low-complexity variable frame rate analysis for speech recognition and voice activity detection. *Selected Topics in Signal Processing, IEEE Journal of*, 4(5):798–807, 2010.
- [84] Seth Teller, Matthew R Walter, Matthew Antone, Andrew Correa, Randall Davis, Luke Fletcher, Emilio Frazzoli, Jim Glass, Jonathan P How, Albert S Huang, et al. A voice-commandable robotic forklift working alongside humans in minimally-prepared outdoor environments. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 526–533. IEEE, 2010.
- [85] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
- [86] Karel Veselý, Arnab Ghoshal, Lukás Burget, and Daniel Povey. Sequence-discriminative training of deep neural networks. In *INTERSPEECH*, pages 2345–2349, 2013.
- [87] Stratis D Viglas. Data management in non-volatile memory. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1707–1711. ACM, 2015.

- 
- [88] Philip C Woodland and Daniel Povey. Large scale discriminative training of hidden Markov models for speech recognition. *Computer Speech & Language*, 16(1):25–47, 2002.
- [89] Hong You and Abeer Alwan. Temporal modulation processing of speech signals for noise robust ASR. In *INTERSPEECH*, pages 36–39, 2009.
- [90] Kisun You, Jungwook Choi, and Wonyong Sung. Flexible and expandable speech recognition hardware with weighted finite state transducers. *Journal of Signal Processing Systems*, 66(3):235–244, 2012.
- [91] Steve J. Young. The HTK hidden Markov model toolkit: Design and philosophy. Technical report, Cambridge University, 1994.
- [92] Steve J Young, Julian J Odell, and Philip C Woodland. Tree-based state tying for high accuracy acoustic modelling. In *Proceedings of the workshop on Human Language Technology*, pages 307–312. Association for Computational Linguistics, 1994.
- [93] Dong Yu, F. Seide, Gang Li, and Li Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4409–4412, March 2012.
- [94] J. Zahurak, K. Miyata, M. Fischer, M. Balakrishnan, S. Chhajed, D. Wells, Hong Li, A. Torsi, Jay Lim, M. Korber, K. Nakazawa, S. Mayuzumi, M. Honda, S. Sills, S. Yasuda, A. Calderoni, B. Cook, G. Damarla, Hai Tran, Bei Wang, C. Cardon, K. Karda, J. Okuno, A. Johnson, T. Kunihiro, Jun Sumino, M. Tsukamoto, K. Aratani, N. Ramaswamy, W. Otsuka, and K. Prall. Process integration of a 27nm, 16Gb Cu ReRAM. In *Electron Devices Meeting (IEDM), 2014 IEEE International*, pages 6.2.1–6.2.4, Dec 2014.
- [95] DongNi Zhang. Low-energy radix-2 serial and parallel FFT designs. Master’s thesis, Massachusetts Institute of Technology, 2013.
- [96] Victor Zue, Stephanie Seneff, James R Glass, Joseph Polifroni, Christine Pao, Timothy J Hazen, and Lee Hetherington. Jupiter: A telephone-based conversational interface for weather information. *Speech and Audio Processing, IEEE Transactions on*, 8(1):85–96, 2000.