40 nm CMOS Ultra-Low-Power Keyword Spotting Hardware Design

Peihao Sun, Tingyi Fan

Department of Electrical and Information Technology
Lund University

Advisor: Jonas Skeppstedt

Examiner: Pietro Andreani

June 11, 2026

Printed in Sweden
E-huset, Lund, 2026

Abstract

Keyword spotting (KWS) is an essential function in always-on voice-interface systems, where a low-power detector continuously monitors audio streams and activates subsequent speech-processing modules only when a target keyword is detected. This thesis presents the design and evaluation of a 40 nm CMOS ultra-low-power KWS accelerator that integrates a Mel-Frequency Cepstral Coefficient (MFCC) front-end with a Depthwise Separable Convolutional Neural Network (DSCNN) back-end. To reduce hardware overhead, we optimize not only the front-end processing, the back-end neural network individually but also co-design the whole processing. The proposed architecture adopts fixed-point arithmetic, shared multiplier resources, finite-state-machine-based scheduling, and on-chip SRAM buffering.

The design is implemented and evaluated using a 40 nm HVT CMOS technology under a 1.1 V supply voltage. Post-synthesis results (TT, 1.1 V, 25°C) show that the complete KWS accelerator achieves an active-window average power of $P_{\text{avg}} = 1220.44\ \mu$W during MFCC extraction and DSCNN inference. After incorporating duty-cycled execution over a 100 MHz reference clock, the resulting long-term average power reduces to $P_{\text{total}} = 32.757\ \mu$W, while the total synthesized cell area is 132,731.94 μm^2. SRAM macros occupy 117,658.50 μm^2 (88.6% of the total cell area). The proposed design demonstrates the feasibility and practical value of integrating MFCC feature extraction and DSCNN classification into a compact, ultra-low-power ASIC for always-on keyword spotting.

Contents

1 **Introduction** 2
1.1 Motivation 2
1.2 Research Questions 3
1.3 Proposed Approach 3
1.4 Main Contributions 3
1.5 Thesis Organization 4
1.6 Division of Work 4

2 **Background and Related Work** 6
2.1 Keyword Spotting (KWS) Overview 6
2.2 Audio Front-End and Feature Representation 6
2.3 Common Model Families for KWS 7
2.4 Considerations for Hardware Implementation 9
2.4.1 Time-Domain Pipelining and Arithmetic Resource Reuse 10
2.4.2 Hierarchical Memory Organization and Seamless Interfacing 10
2.4.3 Customized Datapath Design and Arithmetic Trade-offs 10
2.4.4 Full-Pipeline Fixed-Point Quantization 11

3 **Algorithm Design of KWS System** 12
3.1 Design Considerations for Front-end Preprocess 12
3.1.1 MFCC Algorithm Decomposition 12
3.1.2 Operator Analysis and Comparison 14
3.2 Design Considerations for Back-end Neural Network 18
3.2.1 DSCNN Architecture 19
3.2.2 TC-ResNet8 Architecture 19
3.2.3 BC-ResNet Architecture 20
3.2.4 Hardware-Oriented Model Comparison 21

4 **Hardware Implementation and Optimization** 23
4.1 Overall Hardware Architecture 23
4.1.1 MFCC Hardware Architecture 24
4.1.2 DSCNN Hardware Architecure 25
4.2 Processing Elements and Datapath Design 26
4.2.1 MFCC Datapath Design 26

4.2.2 DSCNN Datapath Design 29
4.3 Memory Hierarchy and Data Reuse 31
4.4 MFCC Optimization Strategies 32
4.4.1 FFT Size Optimization 32
4.4.2 Memory Size and Organization Trade-off 33
4.4.3 FFT Algorithm Optimization 34
4.4.4 Mel Filter Optimization 35
4.4.5 Magnitude and Filtering Fusion 36
4.4.6 Summary od MFCC Optimization 36
4.5 DSCNN Optimization Strategies 36
4.5.1 Comparison of Memory-Access Optimization Schemes 38
4.5.2 SRAM Power Evaluation and Technology Selection 40
4.5.3 Weight SRAM Splitting Exploration 41
4.5.4 SRAM-Level Power and Area Comparison 42

5 Results and Discussion 43
5.1 Power Consumption 43
5.2 Area 46
5.3 Timing 47
5.4 Accuracy 48

6 Conclusion 51
6.1 Summary 51
6.2 Research Questions Revisited 52
6.3 Advantages 53
6.4 Limitations 53
6.5 Future Work 54

Acknowledgements 55

Bibliography 56

A Appendix 58

Chapter 1

Introduction

1.1 Motivation

The motivation of this work comes from the need for an ultra-low-power KWS detector suitable for always-on edge applications. In such scenarios, executing the entire speech-processing workload on a general-purpose processor may lead to excessive energy consumption. Sending audio data to the cloud is also undesirable because of communication latency, network dependency, and privacy concerns. A dedicated hardware accelerator can provide a more efficient solution by tailoring the computation and memory organization to the fixed KWS workload. However, designing a low-power KWS accelerator is not straightforward. The front-end feature extractor and the back-end neural network have different computational characteristics. The MFCC front-end contains operations such as windowing, FFT, Mel-filterbank accumulation, logarithmic compression, and DCT. The neural-network back-end mainly consists of convolution, accumulation, activation, pooling, and final classification. If each function is implemented with independent hardware resources, the overall area and power consumption may become too large for an always-on detector. A key observation in this work is that many operations in both the MFCC front-end and the neural-network back-end can be mapped to a shared multiplication-based computation framework. By precomputing deterministic coefficients and storing them in on-chip memory, transcendental or complex operations can be transformed into regular multiply-and-accumulate operations. This makes it possible to reuse arithmetic resources and simplify control. At the same time, because SRAM access and storage occupy a significant portion of the total hardware cost, the memory hierarchy and dataflow must be carefully designed. Therefore, this thesis focuses on a hardware-oriented KWS design methodology. Instead of optimizing only the recognition model, the work considers model structure, data movement, arithmetic reuse, SRAM organization, and post-synthesis implementation results together. The goal is to build a complete and practical KWS accelerator architecture that can serve as a baseline for further low-power optimization.

1.2 Research Questions

The main objective of this thesis is to design and evaluate a 40 nm ultra-low-power KWS accelerator for always-on keyword detection. To guide the design and evaluation, this thesis addresses the following research questions:

RQ1 How can an ultra-low-power keyword spotting (KWS) accelerator be achieved, and which system-level techniques are most effective for minimizing energy consumption?

RQ2 What are the key hardware-architecture trade-offs when implementing an KWS accelerator?

RQ3 Which parts of the system dominate power and silicon area from the final implementation results?

1.3 Proposed Approach

The proposed KWS system consists of two major processing stages: an MFCC front-end and a DSCNN back-end. The MFCC front-end converts the raw input audio stream into a compact time-frequency feature representation. It performs frame-based processing, windowing, FFT, magnitude or energy computation, Mel-filterbank accumulation, logarithmic approximation, and DCT. These operations are optimized via mathematical strategies and implemented using a multiplier-centric approach, where fixed coefficients are stored in on-chip memories and reused during the computation. The back-end classifier adopts a DSCNN architecture. Compared with standard convolution, depthwise separable convolution decomposes the operation into depthwise convolution and pointwise convolution, which reduces the number of multiplications and parameters. This structure is suitable for ASIC implementation because it provides regular data access patterns and allows the computation engine to be reused across layers. In this thesis, DSCNN is selected over other candidate models because it provides a balanced trade-off among memory footprint, computation cost, dataflow regularity, and hardware simplicity. At the hardware level, the system is organized around shared computation and memory resources. On-chip SRAMs are used to store activations, weights, intermediate results, and precomputed coefficients. A finite-state-machine-based control structure schedules memory access, computation, and write-back operations. The architecture emphasizes serialized or partially pipelined execution rather than excessive parallelism, because the target KWS application does not require very high throughput. This allows the design to trade unused performance margin for reduced area and power consumption.

1.4 Main Contributions

The main contributions of this thesis are summarized as follows:

- A complete ASIC-oriented KWS accelerator architecture is designed, integrating MFCC front-end feature extraction and DSCNN back-end classification.
- A multiplier-centric MFCC implementation strategy is adopted to replace more complex iterative function units, enabling arithmetic resource reuse and reducing hardware complexity.
- A hardware-oriented comparison of candidate KWS models is conducted, and DSCNN is selected as the final back-end model due to its balanced memory, computation, and dataflow characteristics.
- A memory-aware hardware architecture is developed using SRAM-based buffering, data reuse, and shared computation resources to support low-power operation.
- The complete design is synthesized and evaluated in a 40 nm HVT CMOS technology. The post-synthesis results show a total power consumption of approximately 1.18 mW and a total synthesized cell area of 132,731.94, with SRAM macros dominating the area and memory/clock activity dominating power consumption.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the background and related work on keyword spotting, audio feature extraction, neural-network models for KWS, and hardware implementation considerations. Chapter 3 presents the algorithm design. Chapter 4 introduces hardware architecture of the proposed KWS system, including the MFCC front-end, candidate neural-network comparison, DSCNN accelerator design, memory hierarchy, and optimization strategies. Chapter 5 reports the implementation results and discusses the design in terms of power consumption, silicon area, timing, and accuracy. Chapter 6 concludes the thesis by summarizing the main findings, discussing the advantages and limitations of the proposed design, and outlining possible directions for future work.

1.6 Division of Work

Throughout the course of this thesis, the work was carried out collaboratively by both authors. The overall system specification, integration strategy, and verification methodology were discussed jointly to ensure a consistent design approach across the complete KWS accelerator.

Regarding the division of responsibilities, Tingyi was responsible for the MFCC front-end processing part, including algorithm optimization, hardware architecture design, RTL implementation, and power analysis. Peihao was responsible for the neural-network back-end processing part, including neural-network model selection, hardware architecture design, RTL implementation, and power analysis.

While the main implementation tasks were divided according to the front-end and back-end modules, all major design decisions, experimental results, and report contents were reviewed and agreed upon jointly by both authors.

Chapter 2

Background and Related Work

This section reviews the technical background and representative prior work underpinning a 40 nm CMOS ultra-low-power keyword spotting (KWS) detector. The discussion starts with the KWS task formulation and system-level constraints, then surveys common KWS model families and illustrative implementations, and finally summarizes key considerations for ultra-low-power on-device inference. By organizing these topics around energy, latency, and accuracy trade-offs, this background motivates the design choices made in the subsequent sections.

2.1 Keyword Spotting (KWS) Overview

Keyword spotting aims to detect a predefined wake word or a small set of keywords from streaming audio. In always-on edge devices, KWS typically serves as a front-end trigger that activates higher-power speech processing only when a keyword is present, so the detector must operate continuously under strict energy and latency constraints.

At the algorithm level, KWS can be formulated as a streaming sequence classification problem, where the system must detect short target patterns within continuous speech while controlling false triggers. Early neural approaches to online word spotting employed recurrent neural networks (RNNs) to model temporal context directly from the input stream and to enable low-latency detection decisions [2].

2.2 Audio Front-End and Feature Representation

A typical always-on speech recognition system consists of an audio front-end, a feature extractor, and a lightweight classifier. The front-end generally includes sampling, optional pre-emphasis, and frame/window processing to generate short-time analysis frames. Due to the high data rate and channel sensitivity of raw waveform signals, many embedded speech recognition systems operate on compact time–frequency representations instead.

Common feature choices include log Mel filterbank energies and Mel-Frequency Cepstral Coefficients (MFCC)[6], computed over sliding windows (typically on the

order of tens of milliseconds). The resulting feature representations can be interpreted either as small images for convolutional models or as short temporal sequences for recurrent or temporal convolutional models. From a hardware perspective, the feature representation directly affects input dimensionality, buffering requirements, and data reuse opportunities in subsequent processing stages.

Among these representations, MFCC is widely adopted due to its ability to approximate human auditory perception while achieving significant dimensionality reduction.[12]. The Mel-scale filterbank emphasizes perceptually relevant frequency bands, while logarithmic compression and discrete cosine transform (DCT) further decorrelate spectral components. As a result, MFCC provides a compact and informative representation that is well suited for resource-constrained embedded systems.

2.3 Common Model Families for KWS

Keyword spotting (KWS) is a small-footprint speech recognition task whose objective is not to transcribe an entire utterance, but to detect whether a short target word or command is present in an audio stream. Because KWS is often used as an always-on front end for voice interaction, the model must satisfy several constraints at the same time: high detection accuracy, low false alarm rate, short response latency, low memory footprint, and low computational cost. These constraints make the choice of neural network architecture particularly important. A model that performs well on a server may be unsuitable for embedded or hardware implementation if it requires excessive multiply–accumulate operations, large activation buffers, or irregular control flow. Therefore, before selecting a final model for hardware realization, it is necessary to review the main neural network families used in KWS and compare their characteristics from both algorithmic and implementation perspectives.

Early neural-network-based KWS systems commonly used *fully connected feed-forward networks*—often referred to in the KWS literature as "DNN" models, and equivalent to multilayer perceptrons (MLPs). In these architectures, acoustic features such as MFCCs or log-mel filter-bank energies are flattened or concatenated over a temporal context window and then passed through several fully connected layers. The advantage of such fully connected DNN/MLP baselines is their regular structure: matrix–vector multiplication, bias addition, and nonlinear activation are easy to describe, train, and map to hardware. This regularity makes them useful baselines in both research and implementation. However, fully connected layers do not explicitly exploit the local time–frequency structure of speech features. To capture variations in pronunciation, speaker, and background noise, these models often require many parameters, which increases memory usage and data movement; for resource-constrained KWS systems, this parameter-heavy nature can become a major limitation [16].

Convolutional neural networks (CNNs) were introduced to KWS to explicitly exploit local patterns in time–frequency features. While a CNN is also a type of deep neural network in the broad sense, here it is distinguished from the above "DNN/MLP" baselines by its use of convolutional layers with local receptive fields

and weight sharing. A spectrogram or mel feature map can be treated similarly to a two-dimensional image, where convolutional kernels learn local acoustic patterns such as formant transitions, short-term energy changes, or frequency-band structures. Compared with fully connected networks, CNNs share weights across positions, so they can achieve better robustness with fewer parameters. Sainath and Parada showed that CNN-based small-footprint KWS models can reduce false rejection compared with fully connected DNN baselines while satisfying limits on parameters or multiplication count [15]. From a hardware viewpoint, CNNs are also attractive because convolution can be parallelized and pipelined. Nevertheless, standard two-dimensional convolution may still require substantial computation, especially when feature maps have many channels or when several convolutional layers are stacked. Therefore, CNNs improve the accuracy–efficiency trade-off, but their hardware cost still needs careful estimation.

Recurrent neural networks (RNNs), including long short-term memory (LSTM) and gated recurrent unit (GRU) models, provide another way to model temporal dependency in speech. Since spoken keywords unfold over time, recurrent layers can accumulate context and learn the sequential structure of an utterance. Convolutional recurrent neural networks (CRNNs) combine convolutional layers for local feature extraction with recurrent layers for long-range temporal modeling, and have been reported to achieve high accuracy with a relatively small number of parameters [1]. However, recurrent computation is inherently sequential: the hidden state at one time step depends on the previous state. This dependency limits parallelism and may increase latency on hardware accelerators. RNN, LSTM, and GRU models also include gate operations and intermediate state storage, which can complicate fixed-point implementation and control logic. As a result, recurrent models are useful candidates for comparison but are not always the easiest choice for low-latency hardware deployment.

Depthwise separable CNNs (DS-CNNs) are especially important for embedded KWS. A standard convolution jointly performs spatial filtering and channel mixing, while depthwise separable convolution decomposes this operation into a depthwise convolution followed by a pointwise convolution. This factorization can greatly reduce the number of operations and parameters. In the "Hello Edge" study, Zhang et al. evaluated several KWS architectures for microcontrollers and showed that DS-CNN achieved strong accuracy under strict memory and computation constraints [19]. For hardware implementation, DS-CNN is appealing because the depthwise part reduces arithmetic cost and the pointwise part can be implemented as regular 1×1 convolution or matrix multiplication. However, DS-CNN also changes the balance between computation and memory access. Depthwise layers may have lower arithmetic intensity, meaning that memory bandwidth and data reuse become important design issues. Therefore, DS-CNN is a strong candidate for KWS hardware, but its accelerator must be designed to avoid inefficient memory movement.

Temporal convolution and residual architectures have also been proposed to improve KWS efficiency. TC-ResNet is a representative example that focuses on temporal convolution for real-time KWS on mobile devices [4]. Instead of relying mainly on two-dimensional convolution over both time and frequency, temporal convolution emphasizes the time dimension and uses a compact residual structure

to improve feature learning. This design reflects an important observation: for KWS, latency and actual device execution time can be as important as theoretical operation count. A model with fewer operations is not necessarily faster if its operations are poorly supported by the target hardware. Conversely, a model with regular memory access and high parallel utilization may run efficiently even if its nominal computation is not minimal. For this reason, TC-ResNet-like models are useful in a model-selection chapter because they connect algorithm design directly with real-time inference constraints.

Attention-based and Transformer-like models have also appeared in small-footprint KWS research. Attention mechanisms can help a model focus on the most informative frames instead of treating all frames equally, which may improve robustness when the keyword occupies only part of the input window or when background noise is present. However, attention and Transformer models often introduce additional matrix multiplications, softmax operations, sequence buffering, and memory-access patterns that are less straightforward than convolution. Lightweight attention can be beneficial in software or when accuracy is prioritized, but it must be evaluated carefully before hardware implementation. In a hardware-oriented KWS thesis, these models can be introduced as advanced candidates rather than assumed to be the best choice. The final decision should depend on measured accuracy, parameter size, operation count, memory footprint, latency, and the degree to which the model maps naturally onto the selected hardware platform [14].

Overall, neural network models for KWS present different trade-offs. DNNs are simple and regular but parameter intensive. CNNs exploit local acoustic patterns and improve accuracy with shared weights. RNNs and CRNNs model temporal context but introduce sequential dependencies. DS-CNNs reduce computation and are highly relevant to embedded deployment, while TC-ResNet and related temporal models emphasize real-time latency. Attention-based models provide stronger temporal selection but may be less hardware friendly. This diversity justifies a later model-selection stage. Instead of choosing a model only by recognition accuracy, a hardware-oriented KWS system should compare candidate networks using a multi-objective evaluation that includes accuracy, model size, arithmetic complexity, memory traffic, latency, and implementation regularity.

2.4 Considerations for Hardware Implementation

The stringent deployment requirements of Keyword Spotting (KWS) deeply drive its overall hardware implementation. Since an always-on voice wake-up detector must continuously monitor audio inputs for hours or days, executing the entire computational workload on general-purpose processors would incur prohibitive energy overhead. Meanwhile, uploading raw audio to the cloud introduces communication costs, network latencies, and serious privacy risks. Consequently, customized hardware acceleration has become an inevitable paradigm.

In an ASIC-oriented design, the entire KWS processing system—spanning from front-end MFCC feature extraction to the back-end neural network classifier—is mapped onto a highly integrated hardware framework. Rather than treat-

ing the front-end and back-end as isolated stages, this framework conceptualizes them as a continuous dataflow, leveraging customized datapaths, parallel processing, and highly optimized memory organizations to jointly mitigate power and latency bottlenecks. Compared to programmable processors, ASICs offer lower post-fabrication flexibility but deliver significantly higher energy efficiency, lower latency, and smaller silicon area for fixed workloads. This makes them ideal for always-on KWS systems operating with well-defined models and application scenarios.

Within this unified hardware acceleration framework, four core design considerations dictate the architecture.

2.4.1 Time-Domain Pipelining and Arithmetic Resource Reuse

Pipelining allows distinct computational stages to operate concurrently. For streaming KWS workloads, audio data arrives sequentially over time. This sequential arrival not only enables window-based neural network processing but also introduces an inherent time-staggering effect, as different processing phases (such as front-end windowing/FFT computation and back-end layer-by-layer inference) do not execute simultaneously. Consequently, hardware utilization can be drastically improved by time-multiplexing the fundamental arithmetic units across both front-end and back-end pipelines, substantially shrinking the overall hardware footprint.

2.4.2 Hierarchical Memory Organization and Seamless Interfacing

In many KWS neural networks, data movement dominates the overall execution time and energy dissipation. Therefore, memory access patterns, data lifetimes, and storage hierarchies are critical factors. Hardware designs heavily rely on on-chip buffering, tiling, line buffers, and double-buffering techniques to maximize weight reuse and cache intermediate activations. Crucially, when both feature extraction and neural inference are implemented on-chip, the memory hierarchy must seamlessly bridge the two components. This allows the extracted MFCC features to be streamed directly into the input buffers of the neural network, entirely eliminating redundant off-chip memory transactions and format conversions.

2.4.3 Customized Datapath Design and Arithmetic Trade-offs

Both the neural network and front-end stages must be mapped onto dedicated arithmetic units. The design of these hardware structures demands a strict compromise among numerical precision, latency, and power consumption. Wider datapaths or complex computing blocks (such as complex multipliers in FFT or multi-channel convolution engines in NNs) can enhance classification accuracy. However, they simultaneously increase switching activity and propagation delays, thereby inflating silicon area and power budgets.

2.4.4 Full-Pipeline Fixed-Point Quantization

Although algorithmic training typically occurs in the floating-point domain, floating-point operations are prohibitively expensive within a compact ASIC footprint. Therefore, full-pipeline fixed-point quantization stands out as the most effective strategy to minimize hardware overhead. On one hand, acoustic features processed by the MFCC stage inherently exhibit a bounded dynamic range, making it highly feasible to represent front-end characteristics using low-bit-width fixed-point numbers. On the other hand, the back-end neural network can quantize weights and activations to lower precisions (e.g., 16-bit fixed-point or 8-bit integer) or even more aggressive binary/ternary formats. Reducing the bit-width across the entire pipeline dramatically shrinks on-chip memory occupancy and simplifies arithmetic logic. By carefully selecting bit-widths through quantization-aware training or calibration, designers can achieve an optimal trade-off between hardware budget and system recognition accuracy.

Chapter 3

Algorithm Design of KWS System

This section introduces the detailed considerations for both front-end preprocessing and neural network designs.

3.1 Design Considerations for Front-end Preprocess

For the front-end MFCC pre-processing, this section provides an overall algorithmic perspective on the MFCC front-end used in the proposed KWS system.It first decomposes the fundamental principles of MFCC processing and then discusses the major design considerations for hardware implementation.In particular, the analysis focuses on how different operator choices affect system response speed, resource occupation, and power consumption. By comparing alternative implementation strategies from these perspectives, this section explains the rationale behind the final hardware-oriented algorithm selection.

3.1.1 MFCC Algorithm Decomposition

The MFCC processing pipeline in the proposed KWS front-end consists of a sequence of stages, including input preprocessing and framing, windowing, Fourier transform, magnitude or power spectrum computation, Mel filter bank processing, logarithmic compression, and discrete cosine transform (DCT). Each stage performs a well-defined mathematical operation that progressively transforms the raw speech waveform into a compact and perceptually meaningful feature representation. The complete algorithmic decomposition is described as follows.

1) Input Preprocessing and Framing For a discrete-time speech signal $x[n]$, the continuous audio stream is first segmented into short-time frames, since speech can be approximated as stationary only over a short duration. Let the frame length be N and the frame shift be H. The m-th frame is defined as

$$x_m[n] = x[n + mH], \quad 0 \leq n \leq N - 1. \tag{3.1}$$

This operation converts the long speech sequence into locally stationary segments, enabling effective spectral analysis while preserving short-time temporal structure.

2) Windowing Each frame is multiplied by a window function to reduce spectral leakage caused by discontinuities at frame boundaries. A commonly used choice is the Hamming window:

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \leq n \leq N-1. \tag{3.2}$$

The windowed signal is given by

$$s_m[n] = x_m[n] \cdot w[n]. \tag{3.3}$$

This step smooths the frame edges and improves spectral energy concentration.

3) Fourier Transform The short-time spectrum is obtained using the discrete Fourier transform (DFT), typically implemented via FFT:

$$S_m[k] = \sum_{n=0}^{N-1} s_m[n]\, e^{-j\frac{2\pi kn}{N}}, \quad 0 \leq k \leq N-1. \tag{3.4}$$

This transformation maps the signal from the time domain to the frequency domain, exposing spectral characteristics such as formants and harmonics.

4) Magnitude Since perceptual features are primarily related to spectral energy, the complex spectrum is converted into magnitude or power form:

$$|S_m[k]| = \sqrt{\mathrm{Re}\{S_m[k]\}^2 + \mathrm{Im}\{S_m[k]\}^2}, \tag{3.5}$$

$$P_m[k] = \frac{1}{N}|S_m[k]|^2. \tag{3.6}$$

This step removes phase information and retains the spectral intensity relevant for perception.

5) Mel Filter Bank Processing The power spectrum is processed by a bank of triangular filters distributed on the Mel scale, which approximates human auditory perception:

$$\mathrm{mel}(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right). \tag{3.7}$$

Let $H_i[k]$ denote the i-th Mel filter. The filter output energy is

$$E_m[i] = \sum_k P_m[k]\, H_i[k]. \tag{3.8}$$

This step aggregates spectral energy into perceptually meaningful frequency bands.

6) Logarithmic Compression Logarithmic compression is applied to the filter-bank energies:

$$L_m[i] = \log(E_m[i]). \tag{3.9}$$

This operation converts multiplicative variations into additive ones and better matches the logarithmic nature of human loudness perception.

7) Discrete Cosine Transform (DCT) Finally, DCT is applied to obtain cepstral coefficients:

$$C_m[r] = \sum_{i=0}^{M-1} L_m[i] \cos\left(\frac{\pi r(i+0.5)}{M}\right), \quad 0 \le r \le R-1, \tag{3.10}$$

where M is the number of Mel filters and R is the number of retained coefficients. This step decorrelates the features and concentrates information into a compact representation.

Overall, the MFCC algorithm transforms raw speech into a compact representation by progressively introducing short-time stationarity, spectral analysis, perceptual frequency warping, logarithmic compression, and decorrelation. This decomposition not only clarifies the role of each processing stage but also provides a clear foundation for subsequent hardware-oriented optimization.

From an algorithmic perspective, MFCC extraction involves computationally intensive operations such as exponential evaluation, trigonometric functions, logarithmic functions, multiplications, and multiply-accumulate operations. A naive implementation using dedicated hardware for each function would lead to high complexity and low resource utilization. Therefore, an important design objective is to identify unified computational primitives that can be reused across multiple stages through reconfiguration, enabling efficient hardware implementation.

3.1.2 Operator Analysis and Comparison

Based on the decomposition of the MFCC algorithm, the overall processing pipeline involves multiple types of operations, including trigonometric evaluation, logarithmic transformation, multiplication, and accumulation. A naive implementation using dedicated hardware blocks for each function would lead to high hardware complexity, low resource utilization, and increased power consumption. Therefore, a key design objective is to identify a unified computational framework and select optimal operator primitives that can support multiple algorithmic stages through reconfiguration or coefficient reuse, tightly coupling the front-end features with the back-end infrastructure.

1) Operator Implementation Options From a hardware mapping perspective, two major paradigms exist for handling the transcendental and multiplicative operations required by MFCC:

- **CORDIC-Based Approach:** The coordinate rotation digital computer (CORDIC) algorithm is a strong candidate for a unified operator. By selecting different operating modes and parameter configurations, a single CORDIC engine can be reconfigured to perform various functions. The general iterative equations are given by:

$$x_{i+1} = x_i - \mu \cdot d_i\, y_i\, 2^{-i}, \tag{3.11}$$

$$y_{i+1} = y_i + d_i\, x_i\, 2^{-i}, \tag{3.12}$$

$$z_{i+1} = z_i - d_i\, \alpha_i, \tag{3.13}$$

where $\mu \in \{-1, 0, +1\}$ dictates the coordinate system, $d_i \in \{-1, +1\}$ represents the rotation direction, and α_i denotes the precomputed elementary angle or constant table.

Specifically, when $\mu = +1$, the system operates in the circular coordinate system; in rotation mode ($z \to 0$), it iteratively evaluates sine and cosine functions, whereas in vectoring mode ($y \to 0$), it computes the magnitude and phase, which can be applied to the FFT magnitude extraction. When $\mu = -1$, the system switches to the hyperbolic coordinate system, enabling the calculation of hyperbolic functions and, through vectoring mode, the natural logarithmic transformation ($\ln(x)$) required after Mel-filtering. When $\mu = 0$, the CORDIC structure degenerates into a linear coordinate system, where the shift-add datapath performs basic multiplication and division operations without requiring dedicated hardware multipliers.[17]

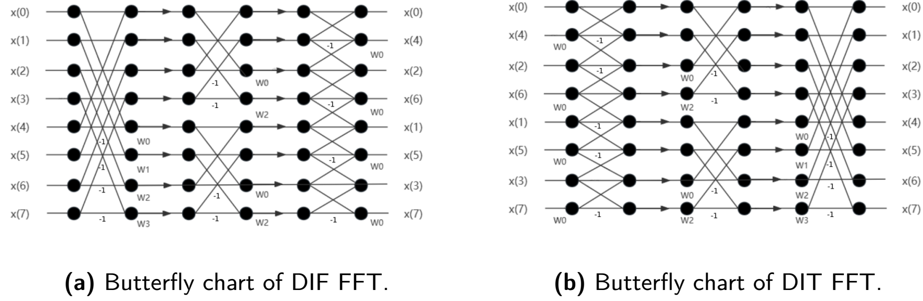
- **Multiplier-Centric Approach:** An alternative paradigm exploits the deterministic nature of the MFCC pipeline, converting transcendental operations into regular arithmetic stages driven by a shared multiplication core. Under this approach, the specific computational details for each MFCC stage are mapped as follows:

  1. *Windowing:* The incoming speech samples are multiplied by fixed pre-emphasis coefficients $(1-\alpha z^{-1})$ and subsequently multiplied with precomputed Hamming or Hanning window coefficients fetched sequentially from an on-chip ROM.
  2. *FFT Acceleration:* The trigonometric terms in the butterfly stages are completely bypassed. Since the frame length is fixed, the twiddle factors ($\mathcal{W}_N^k = \cos(2\pi k/N) - j\sin(2\pi k/N)$) are deterministic. The butterfly complex multiplications are resolved through address indexing and a sequence of real-number multiplications using the core multiplier.
  3. *Mel-Scale Filterbank Energy Accumulation:* The continuous spectrum magnitude is multiplied by the frequency response weights(always 1 for rectangle filter) of the Mel-filterbank. These filter weights are stored as sparse matrix coefficients, reducing the filtering process to a sequence of multiply-accumulate (MAC) operations.
  4. *Logarithmic Energy Fitting:* The non-linear logarithmic transformation is approximated using a piecewise linear fitting scheme. For the q-th segment:

$$\log(x) \approx a_q x + b_q, \quad x \in [t_q, t_{q+1}), \tag{3.14}$$

     where the segment index is quickly determined via range comparison or leading-zero counting, and the final logarithmic value is computed using one coefficient multiplication ($a_q \cdot x$) followed by an addition ($+b_q$).

2) FFT Hardware Choices The Fast Fourier Transform (FFT) is the most computationally intensive block [5] in the front-end processor. In this design, a



(a) Butterfly chart of DIF FFT.

(b) Butterfly chart of DIT FFT.

Figure 3.1: Dataflow comparison of 8-point FFT using DIF and DIT topologies.

radix-2 architecture is adopted rather than higher-radix variants (such as radix-4). Radix-2 FFT provides a highly regular butterfly structure and simpler control logic. For always-on KWS hardware, minimizing arithmetic operations via complex radices often introduces complicated data routing, address generation, and control scheduling that outweigh the power savings. Thus, radix-2 provides a practical trade-off maximizing structural regularity.

Beyond radix selection, the core architectural choice lies in data ordering, specifically between Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF) topologies:

- **Decimation-in-Time (DIT)[5]:** The DIT structure requires the input time-domain sequence to be reorganized into bit-reversed (shuffled) order prior to the butterfly computations. Implementing DIT requires incorporating complex bit-reversal logic or shuffled addressing control at the very beginning of the FFT stage to rearrange the incoming streaming data.
- **Decimation-in-Frequency (DIF)[7]:** Conversely, the DIF structure accepts the input sequence in perfect natural (sequential) order and performs butterfly operations stage by stage. The first stage of butterfly open-loop computation inherently takes two sequential blocks of data separated by $N/2$. However, its final frequency-domain output emerges in bit-reversed order, requiring a bit-reversal mapping when writing to or reading from the final spectrum buffer.

Figure 3.1a and 3.1b illustrate the data processing flow of 8-points FFT using butterfly chart, we can tell that they are in mirror structure.

3) Comparative Analysis and Final Selection To establish the optimal architectural configuration for the target low-power KWS hardware, a rigorous trade-off analysis was conducted for both operator primitives and FFT topologies.

First, regarding the arithmetic primitive layer, a rigorous trade-off analysis was conducted between the CORDIC-based engine and the proposed multiplier-centric approach. Table 3.1 summarizes the main trade-offs between the two solutions.

The CORDIC algorithm evaluates trigonometric functions, vector rotations, and linear scaling through purely shift-and-add operations, thereby eliminating the

Table 3.1: Comparison between CORDIC-based and multiplier-centric MFCC implementations

Implementation	Latency	Complexity	Parameter Storage	Reusability	Suitability
CORDIC-based	High	High	High	High	Moderate
Multiplier-centric	Low	Low	Moderate	High	High

need for dedicated hardware multipliers—a property that favored its widespread adoption in early resource-constrained systems. However, CORDIC inherently relies on multi-cycle iterative computation, where the execution latency scales proportionally with the target numerical precision. If an unrolled, fully-pipelined CORDIC architecture is deployed to enhance throughput, a massive number of pipeline registers must be inserted at each iterative stage[9]. This inevitably inflates the silicon area and significantly increases the clock-tree power dissipation due to heavy switching activities. Furthermore, a practical CORDIC engine demands substantial control complexity, requiring auxiliary angle lookup tables, iteration steering logic, and scaling-factor compensation modules.

In contrast, since an always-on KWS system inherently incorporates a back-end neural network inference engine, heavy multiply-accumulate (MAC) units are already natively available on-chip. Consequently, adopting a lookup-table (LUT) approach combined with a shared multiplier framework introduces a profound advantage: the MFCC front-end can directly time-multiplex the existing multiplier resources inside the neural network's computing array. This high-level hardware sharing completely eliminates the silicon overhead of constructing an isolated CORDIC structure. For deterministic algorithmic parameters—including the FFT twiddle factors, Mel-filterbank weights, and DCT coefficients—the values can be pre-quantized offline and stored in compact on-chip ROMs, converting transcendental evaluations into straightforward, single-cycle MAC operations. This multiplier-centric scheme drastically curtails the register inflation and control logic overhead imposed by CORDIC, while significantly simplifying the fixed-point quantization flow and timing closure. **Therefore, considering silicon area, dynamic power, control complexity, and hardware reusability, this work rejects the iterative CORDIC structure and finally adopts the shared multiplier-centric architecture.**

Second, regarding the FFT decimation strategy, DIT and DIF are mathematically equivalent, sharing identical butterfly arithmetic and computational complexity. The critical hardware trade-off lies strictly in whether the bit-reversal (shuffling) overhead is incurred at the input or the output stage, and how well the dataflow aligns with the memory access scheduling.

The DIF dataflow perfectly aligns with an optimized in-place computation and pipelined multiplication scheduling. As soon as the control logic fetches two required operands from the Data SRAMs, the butterfly addition and subtraction can be evaluated immediately. The newly computed intermediate values can be written back to the same SRAM addresses, while the subtraction result is simultaneously routed to the shared multiplier for the subsequent twiddle-factor multiplication.

This tight interleaving of memory access, in-place writeback, and multiplication effectively hides the multiplier's data-waiting latency and accelerates the pipeline execution, yielding a noticeable temporal benefit.

Although the final output spectrum of the DIF topology emerges in bit-reversed order, this is easily resolved via an address reverse function when transferring data to the subsequent magnitude stage by address wire recombination, which cost trivial hardware cost. **Therefore, the radix-2 DIF architecture is selected for the FFT module**, leveraging its natural-order input and hardware-friendly dataflow to maximize throughput and minimize control overhead. (A detailed architectural breakdown of this pipelined SRAM-multiplier execution will be provided in Chapter 4).

3.2 Design Considerations for Back-end Neural Network

For the back-end neural network, this section will introduce the three candidate keyword spotting (KWS) networks considered in this work: DSCNN, TC-ResNet8, and BC-ResNet. Instead of describing training, quantization, and inference as separate generic steps, this section focuses on the algorithmic structures that determine hardware implementation cost. The purpose is to explain why these three models are selected as candidates and how their layer organization, data dependency, memory footprint, and operation count influence ASIC-oriented implementation.

For always-on KWS hardware, the target network should not only provide acceptable recognition accuracy, but also be compatible with low-power inference. In an ASIC implementation, the main cost usually comes from arithmetic units, on-chip SRAM, data movement, and control complexity. Therefore, a candidate model should be evaluated using both algorithmic metrics and hardware-oriented metrics. In this work, DSCNN, TC-ResNet8, and BC-ResNet are selected as representative candidates because they reflect three different design philosophies. DSCNN uses depthwise separable convolution to reduce computation. TC-ResNet8 uses temporal convolution and residual connections to improve real-time performance. BC-ResNet introduces broadcasted residual structures to enhance accuracy, but its data reshaping and normalization operations make the implementation more complex.

The comparison does not directly assume that the most accurate or deepest model is the best choice. For the intended low-power hardware implementation, the preferred model should have regular layer structures, limited memory requirement, low multiplication count, and sufficient frame processing speed. Since the current application only requires a low frame-rate margin, the power and area costs become more important than peak throughput. In this situation, reducing SRAM size and memory access can be more valuable than increasing the theoretical frame rate.

3.2.1 DSCNN Architecture

DSCNN is based on depthwise separable convolution. A standard convolution mixes spatial filtering and channel mixing in the same operation, while depthwise separable convolution decomposes it into a depthwise convolution followed by a pointwise convolution. In the depthwise part, each input channel is processed independently by its own spatial kernel. In the pointwise part, a 1×1 convolution is used to combine information across channels. This decomposition significantly reduces the number of multiplications compared with standard convolution, especially when the number of channels is large.

As Figure 3.2 illustrates, the depthwise stage performs per-channel spatial filtering, and the subsequent pointwise stage mixes channels through a 1×1 convolution.

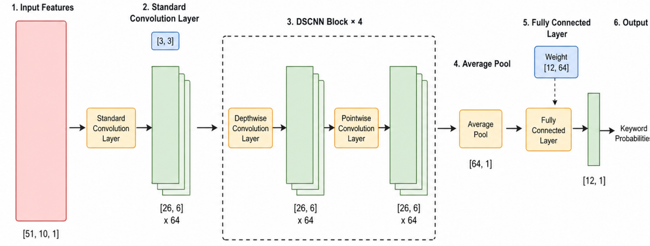


Figure 3.2: Illustration of a depthwise separable convolution block in DSCNN, where a depthwise convolution is followed by a pointwise (1×1) convolution for channel mixing.

In the candidate DSCNN structure, most blocks have consistent dimensions, which simplifies scheduling and hardware reuse. The depthwise layers have weak inter-channel dependency because each channel can be processed independently. The pointwise layers have 1×1 kernels, so the spatial dependency is limited and the computation is mainly channel-wise accumulation. These properties make DSCNN friendly to ASIC implementation: the same processing element can be reused across similar blocks, and the control logic can remain relatively simple.

3.2.2 TC-ResNet8 Architecture

TC-ResNet8 uses temporal convolution and residual connections for KWS. Compared with two-dimensional convolution over both time and frequency, temporal convolution mainly emphasizes the time dimension. This is reasonable for KWS because the target keyword evolves over time, and temporal features are closely related to the final decision. The residual connections help information flow through the network and improve feature representation.

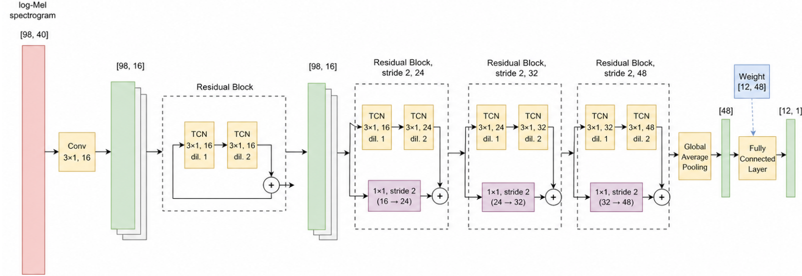


Figure 3.3: Overview of the TC-ResNet8 architecture used as a candidate KWS network.

As Figure 3.3 illustrates, TC-ResNet8 organizes computation around temporal convolutions and residual connections to emphasize time-domain context.

From the hardware perspective, TC-ResNet8 has two important characteristics. First, it contains several stride-2 operations, which reduce the temporal resolution and improve the frame processing speed. Second, its channel numbers are relatively large, such as 32 and 48 channels in the intermediate layers. This improves model capacity but also increases weight storage and arithmetic resources. According to the collected metrics, TC-ResNet8 provides the highest estimated FPS among the three models, but it also has the largest total bit count. Since the target application does not require a very high FPS, this throughput advantage may not fully compensate for the larger memory footprint.

3.2.3 BC-ResNet Architecture

BC-ResNet introduces broadcasted residual blocks to improve the representation ability of compact KWS models. Its structure contains transition blocks and broadcasted blocks. Compared with DSCNN and TC-ResNet8, BC-ResNet includes more data reshape operations, reduce-mean operations, normalization, sigmoid, multiplication, and residual addition. These operations may improve algorithmic performance, but they also increase the difficulty of hardware mapping.

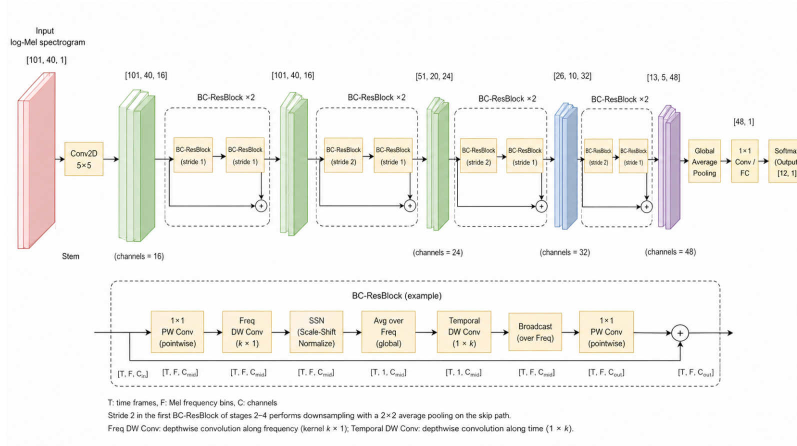


Figure 3.4: Overview of the BC-ResNet architecture used as a candidate KWS network.

As Figure 3.4 illustrates, BC-ResNet builds on broadcasted residual blocks, which improves representation ability but introduces additional control and dataflow complexity.

The main challenge of BC-ResNet is not only the number of multiplications, but also the irregular dataflow. Data reshaping changes the memory access pattern. Reduce-mean and normalization require extra reduction and scaling operations. Different blocks may also have inconsistent parameter sizes, which reduces the possibility of reusing a single simple datapath. According to the collected metrics, BC-ResNet has the smallest total bit count among the three models, but it has the largest multiplication count and much higher read/write cycles. Therefore, although BC-ResNet is attractive from the model-compression perspective, its control and memory access complexity are less suitable for a first-stage low-power ASIC implementation.

3.2.4 Hardware-Oriented Model Comparison

The following table summarizes the current hardware-oriented metrics of the three candidate models. The weight bit and activation bit determine the lower bound of SRAM capacity and influence both static and dynamic power. The multiplication count affects the switching activity of arithmetic units and the upper bound of memory access demand. Read and write cycles reflect data movement, which is often a major contributor to energy consumption in neural network hardware. FPS indicates the available processing margin, but in this work the required FPS is low, so FPS is treated as a constraint rather than the main optimization target.

To make the comparison clearer, Table 3.3 gives a qualitative ranking. DSCNN provides the most balanced result: its total memory requirement is close to BC-ResNet, its multiplication count is far lower than BC-ResNet, and its structure is more regular than both residual-based alternatives. TC-ResNet8 is fast, but its total bit count is high. BC-ResNet has a small total bit count, but its multiplication

Table 3.2: Hardware-oriented comparison of candidate KWS networks

Model	Weight bits	Activation bits	Mult.	Read cycles	Write cycles	FPS	Total bits
DSCNN	176 128	159 744	3 315 456	80 446	89 868	30.0	335 872
TC-ResNet8	513 024	140 000	1 950 528	30 224	25 816	51.0	655 232
BC-ResNet	171 264	258 560	13 217 648	1 205 568	645 200	7.5	332 752

count and memory access cycles are much larger.

Table 3.3: Qualitative hardware suitability of candidate networks

Model	Memory footprint	Multiplication cost	Dataflow regularity	ASIC suitability
DSCNN	Medium	Medium	High	High
TC-ResNet8	High	Low	Medium	Medium
BC-ResNet	Low	High	Low	Low–Medium

Based on this comparison, DSCNN is the most suitable candidate for the current low-power ASIC-oriented implementation. The main reason is not that DSCNN is optimal in every single metric, but that it offers the best balance among memory cost, computation cost, and implementation regularity. Its repeated depthwise-pointwise blocks allow processing element reuse, and its limited inter-channel dependency in depthwise layers simplifies scheduling. Since the target FPS requirement is not strict, the higher FPS of TC-ResNet8 is less important than its larger memory footprint. Similarly, although BC-ResNet has a small total bit count, its high multiplication count and irregular operations may lead to higher dynamic power and more complex control.

Chapter 4

Hardware Implementation and Optimization

This part presents the hardware implementation of the proposed MFCC front-end and selected KWS model. The corresponding optimization methods for low-power KWS applications -based on the previous algorithmic comparison, DSCNN is treated as the main back-end implementation target because it provides a balanced trade-off among memory footprint, multiplication count, and dataflow regularity. The discussion is organized into four parts: overall hardware architecture, processing element and datapath, memory hierarchy and data reuse, and optimization strategies.

4.1 Overall Hardware Architecture

The overall hardware architecture is illustrated in Fig .4.1 The architectural design of the proposed always-on Keyword Spotting (KWS) system is driven by practical commercial deployment specifications. The overall computational pipeline is partitioned into two core processing domains: the front-end Mel-Frequency Cepstral Coefficients (MFCC) pre-processing pipeline and the back-end Depthwise Separable Convolutional Neural Network (DSCNN) classification engine. The front-end hardware processor takes these time-domain speech frames and sequentially passes them through windowing and the previously discussed radix-2 DIF FFT module to extract the short-term power spectrum. To compress the spectral dimensionality while mimicking human auditory perception, the computed spectrum is mapped onto a Mel-scale filterbank containing exactly 40 discrete frequency bins. The log-energy outputs of these 40 channels are subsequently processed by a Discrete Cosine Transform (DCT) block for de-correlation. To optimize the memory footprint and arithmetic load of the subsequent neural network without sacrificing key acoustic cues, the DCT engine discards higher-order coefficients and extracts exactly 10 localized feature values per speech frame.

The interface between the front-end feature extractor and the back-end neural classifier is data sram. Each extracted 10-dimensional feature vector represents the acoustic signature of a single localized frame. Let M denote the frame rate parameter (the total number of historical frames within a target classification window). The back-end DSCNN classification engine remains in a low-power standby state until a complete 2D feature matrix of size $M \times 10$ is fully assembled in the shared SRAM buffer.

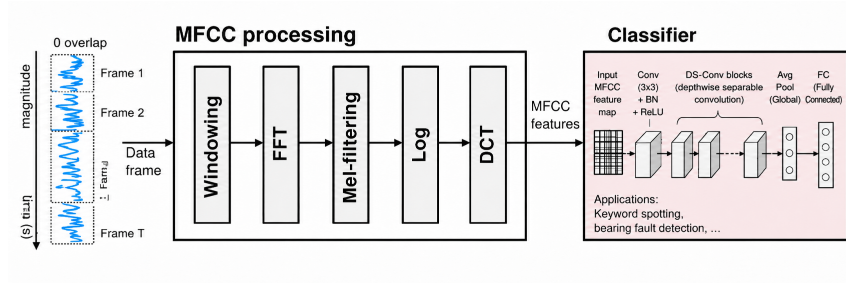


Figure 4.1: Overall architecture of the KWS hardware design.

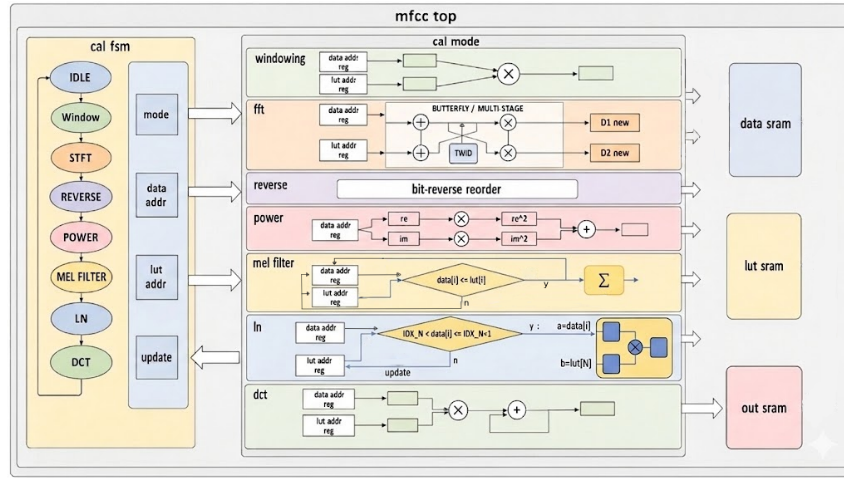


Figure 4.2: Overall architecture of the proposed MFCC hardware design.

Once the $M \times 10$ acoustic feature map is compiled, the DSCNN engine is triggered to execute high-performance inference, mapping the temporal-spectral features through unrolled depthwise and pointwise convolutional layers to generate the final keyword classification probabilities. MFCC processing and the classifier shares the same Multiplier unit governed by a multiplier arbiter.

4.1.1 MFCC Hardware Architecture

Based on the operator-selection discussion in the previous section, the proposed MFCC hardware architecture is organized into two major parts: a control path and a computation path, together with a set of basic processing resources. The control path is responsible for state scheduling and address generation, while the computation path performs algorithm-specific operations according to the current state and the generated addresses. At the lowest level, the basic processing resources mainly consist of two SRAM blocks and one multiplier unit, which together form the fundamental computation engine of the architecture.

The overall architecture of MFCC is illustrated in Fig. 4.2.

The state control module (cal fsm module) determines the execution stage of the MFCC pipeline and generates the corresponding data indexing addresses. According to the current state and incoming control information, the computation module (cal mode module) fetches the required data from the SRAMs in the basic unit, performs the designated arithmetic operations, and writes the results back to memory. In this way, the overall implementation follows a state-driven processing flow, in which memory access and arithmetic execution are tightly coordinated.

A serial architecture is adopted in this design primarily due to the available timing margin under representative operating conditions, as well as strict hardware resource constraints. As a reference configuration given by software department, a clock frequency of 100 MHz, a sampling rate of 8 kHz, and a frame length of 256 samples are considered.

Under this operating point, hardware evaluation shows that the complete MFCC processing flow requires approximately 1.3×10^4 clock cycles to process one frame, given the main computation frequency is 100MHz, this processing time is really small. This result indicates that the architecture provides sufficient timing margin to meet real-time processing requirements of the target application. Therefore, a portion of execution latency can be traded for reduced hardware cost and improved resource efficiency.

Based on this observation, and considering both the coordination with the neural network backend and the resource constraints imposed by the target platform, the design adopts a single shared multiplier to perform all arithmetic operations. Since only one multiplier is available and strong data dependencies exist among different MFCC stages, a multi-state parallel architecture would offer limited performance benefits while significantly increasing hardware complexity and silicon area. Consequently, a serial execution scheme is selected to trade execution speed for reduced resource usage.

It should be noted that the above sampling rate and frame length are used only as representative reference points for architectural analysis, rather than fixed design constraints. Even if these parameters are modified in future configurations, the current evaluation still provides a meaningful baseline for understanding the trade-off between timing performance and hardware resources.

Meanwhile, the top control module, state control module, computation module, and basic processing unit are connected in a cascaded pipeline structure. This pipelined organization is introduced not to maximize parallelism, but to reduce unnecessary idle cycles and improve execution efficiency under the constraint of a single multiplier. In other words, although the overall architecture remains fundamentally serial, pipeline structuring helps recover part of the performance loss and shortens the total processing time without compromising the low-resource design objective.

4.1.2 DSCNN Hardware Architecure

The proposed design mainly consists of three parts. The first part is the top-level finite-state machine, which determines the currently active neural-network layer and sends the corresponding configuration parameters to other modules so that the subsequent computation can be executed correctly. The second part is the

computation engine, which is responsible for the main operations of standard convolution, the depthwise and pointwise stages in depthwise separable convolution, and the fully connected layer. The third part is the average pooling module, which is used to perform data reduction and output generation during the pooling stage.

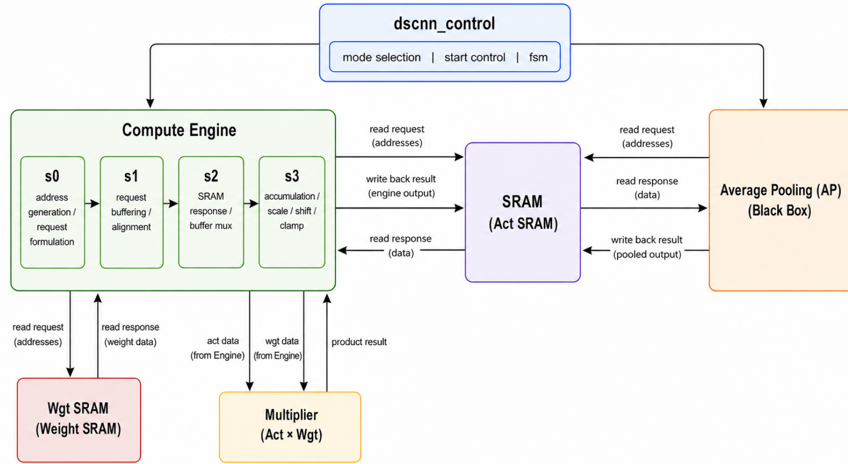


Figure 4.3: Top-level hardware structure of the proposed DSCNN accelerator, highlighting the control FSM, computation engine, and average pooling module.

As Figure 4.3 illustrates, the top-level FSM coordinates the computation engine and pooling module, while shared SRAM macros provide weight and activation storage.

4.2 Processing Elements and Datapath Design

4.2.1 MFCC Datapath Design

This part describes the detailed functionality of each hardware module and the organization of the datapath.

1) Top Control Module The top control module serves as the system-level controller and the primary interface between the MFCC processor and external signals. It integrates all submodules and orchestrates the overall input/output data flow. After system reset, the module receives external input samples and writes framed speech data into the data SRAM, while initializing the parameter lookup-table SRAM. Once initialization is complete, the processing flow is triggered by an external control signal.

During operation, the top module supervises frame-level execution. Upon completion of each frame, it generates a completion signal to notify the external system that the architecture is ready to process the next frame. Meanwhile, the

extracted MFCC features are written to an external SRAM, which serves as the input buffer for the subsequent neural network backend. In this way, the external SRAM naturally bridges the front-end feature extractor and the back-end neural network accelerator.

It is worth noting that the data SRAM is updated on a per-frame basis as new speech input arrives, whereas the parameter SRAM remains constant after initialization under a fixed configuration. This is because the parameter SRAM stores precomputed coefficients determined by system parameters such as sampling rate and frame length. Once these parameters are fixed, the parameter SRAM does not require frequent updates.

2) State Control Module The state control module is responsible for managing the MFCC processing flow and generating the corresponding memory access addresses. It consists of a global finite-state machine (FSM) and an address generator.

The FSM partitions the MFCC computation into multiple processing states, including the idle state, windowing state, fast Fourier transform (FFT) state, bit-reversal reordering state, magnitude computation state, Mel filtering state, logarithmic transformation state, DCT state, and done state. Among these, the bit-reversal reordering state is introduced as a post-processing step associated with the decimation-in-frequency FFT implementation.

The address generator produces address indices based on the current processing state. Since different algorithm stages require distinct memory access patterns, the addressing mode is dynamically selected according to the FSM state. Address updates are driven by progress signals returned from the downstream computation module, ensuring that the control path remains synchronized with the execution status of the datapath.

When the system is in the idle state and completes the handshake with the upstream start signal, the FSM transitions from the IDLE state to the corresponding active state. In each active state, the FSM configures the addressing mode, generates the required addresses, and forwards both the current state and address information to the downstream computation module.

To simplify the control interface, three address outputs are provided between the control module and the computation module. This design is primarily motivated by the FFT stage, where up to three addresses may be required simultaneously, including two data addresses and one twiddle-factor address. In other processing stages where fewer addresses are needed, the unused address outputs are forced to zero, which helps reduce unnecessary signal switching.

3) Computation Module The computation module serves as the core processing unit of the MFCC hardware architecture. Based on the operator analysis discussed previously, multiplication is selected as the fundamental arithmetic primitive. Accordingly, the overall computation flow can be abstracted into a sequence of operations, including address acquisition from the upstream control module, memory access through SRAM, arithmetic processing using the shared multiplier, and result write-back to data SRAM or output to the top-level interface. These operations are executed in a serialized manner.

To support this execution model, the computation module is organized using a hierarchical finite-state machine (FSM) structure, consisting of a global (coarse-grained) FSM and multiple local (fine-grained) FSMs. The global FSM corresponds to the processing states received from the upstream state control module, thereby selecting the appropriate computation mode for different MFCC stages. For each global state, a corresponding local FSM is activated to manage detailed operations such as data fetching, arithmetic execution, and result storage.

During operation, the computation module receives the current processing state and data address indices from the state control module through a handshake interface. Based on the received state, the module transitions into the corresponding local FSM loop. Within this loop, data exchange with SRAM and the multiplier unit is performed through handshake-based control, ensuring correct synchronization among modules.

For each group of input addresses, once the associated computation sequence is completed, the computation module generates a progress pulse and sends it back to the upstream state control module. This pulse indicates that the current data processing step has finished. The state control module then updates the address indices accordingly and provides the next set of addresses to the computation module.

This process repeats iteratively until all data associated with the current processing state are completed. In this way, the computation module maintains tight synchronization with the control path while enabling efficient serialized execution under strict hardware resource constraints.

4) Basic Processing Unit The basic processing unit is located at the downstream end of the computation module pipeline and serves as the fundamental execution backend of the datapath. It consists of two SRAM blocks and a shared multiplier, which together support all data storage and arithmetic operations in the MFCC processing flow.

The two SRAM blocks include a data SRAM and a parameter SRAM. The data SRAM is used to store intermediate computation results and frame-level data, while the parameter SRAM stores precomputed coefficients required by different processing stages. Both SRAM modules receive address indices from the upstream computation module through a handshake interface and return the corresponding data upon successful memory access. This handshake-based interaction ensures correct synchronization between the control path and the memory subsystem.

The multiplier is a shared arithmetic resource that is also reused by the neural network backend. During operation, the computation module provides two operands to the multiplier via a handshake protocol. After the multiplication is completed, the result is returned to the computation module through the same handshake mechanism. This shared usage significantly reduces hardware cost while maintaining sufficient computational capability for both the MFCC front-end and the neural network backend.

From a datapath perspective, the basic processing unit operates in a tightly coupled manner with the computation module. For each processing step, address indices are first issued to the SRAM modules to fetch the required data or coefficients. The retrieved values are then forwarded to the multiplier for arithmetic

processing. The computed results are subsequently written back to the data SRAM or forwarded to the output interface, depending on the current processing stage.

This organization enables a unified "memory-access + multiplication" execution pattern across different MFCC stages, which simplifies datapath design and maximizes hardware reuse under strict resource constraints.

4.2.2 DSCNN Datapath Design

Within the computation engine, the pipeline is divided into four logical stages, namely `s0`, `s1`, `s2`, and `s3`. Stage `s0` generates request addresses and determines the semantic type of the current operation, including whether the current activation should be treated as zero, treated as constant one, fetched from a reuse path, or considered the final accumulation step of the current output point. Stage `s1` introduces a first-level buffer for the request side so that the address and control metadata are time-aligned before entering the external memory-access logic, while also absorbing downstream backpressure. Stage `s2` receives data returned from SRAM and decides, according to the current mode, whether the data should be written into a shared buffer for later reuse or directly forwarded as the operand of the current cycle to the multiplier. Stage `s3` accumulates the multiplication results and performs the remaining post-processing steps, including scaling, shifting, and truncation, before generating the final write-back result and its corresponding address.

From a timing perspective, this staged design places address generation, response reception, data reuse, and numerical post-processing into different pipeline stages, thereby preventing all logic from being compressed into a single clock cycle. Because registers isolate each stage, the critical path is significantly shortened and the clock frequency becomes easier to improve; meanwhile, since the actual computation is not reduced, the throughput per unit time does not decrease. In other words, the structure is not merely a decomposition of computation into several steps, but a timing-aware organization that allows address generation, data movement, multiplier feeding, and result emission to form a stable pipeline rhythm [10].

Besides the four explicit pipeline stages, the system also contains two hidden pipeline layers. The first hidden layer is SRAM, whose role is to exchange data between the computation engine and memory, including reading activations or weights from SRAM and writing final results back to SRAM. SRAM is treated as a hidden pipeline layer because, although it is not part of the engine's explicit computation stages, it determines when data can enter the engine and when results can leave the engine; therefore, it must be considered separately in timing analysis. The second hidden layer is the multiplier, which performs multiplication and is shared with the front-end preprocessing logic. Since the multiplier is a scarce computational resource that may be contended by multiple data streams, treating it as a hidden pipeline layer makes it easier to distinguish request generation, data preparation, and multiplication execution as temporally different processes, thereby clarifying the overall dataflow behavior of the system.

To further reduce contention for the shared multiplier, the design also applies a bypass optimization for inputs equal to 0 or 1. When an input operand is identified

as 0, the multiplication result can be directly determined as 0 without invoking the multiplier. When an input operand is identified as 1, the multiplication result can be directly forwarded from the other operand, again without consuming multiplier resources. In this way, several trivial operations that would otherwise occupy the multiplier are eliminated early in the datapath, which effectively reduces the probability that two different data streams compete for the multiplier at the same time and also lowers the timing pressure introduced by shared resources. For an edge DS-CNN accelerator, such an optimization may appear simple in function, but it can significantly improve resource utilization and scheduling flexibility under concurrent dataflow conditions.

In the post-processing stage, the design introduces coefficient scaling and shift operations to map the accumulated intermediate result back to the quantized output range expected by the model. This step is not merely intended to make the result numerically smaller; rather, it aligns the hardware output with the numeric distribution established during training or offline quantization. In other words, the product and accumulation results are internal intermediate representations, while the values consumed by later layers must follow the same quantization mapping as the software model. Without this stage, the hardware output would systematically drift from the model definition, and the final classification accuracy would be difficult to preserve even if the MAC datapath were correct.

More specifically, let the accumulated output be A, the scaling coefficient be C, the shift amount be S, and the final output be Y. Then the quantized post-processing can be written as:

$$Y = \mathrm{clip}\left((A \times C) \gg S\right), \tag{4.1}$$

where $\mathrm{clip}(\cdot)$ denotes saturation, i.e., restricting the result to the range representable by the target data width. In this design, the final output is truncated to 8 bits, so the output range is typically limited to $[0, 255]$ or an equivalent signed representation depending on the selected convention. This process can be understood as the standard hardware implementation of converting a high-precision intermediate value into the low-precision network output required by the next layer.

From a design-motivation perspective, this stage also separates "network accuracy requirements" from "hardware implementation constraints." Although convolution, depthwise convolution, pointwise convolution, and fully connected layers all mathematically reduce to multiply–accumulate operations, their outputs cannot be directly fed into the next layer in hardware; they must be aligned through quantization to remain consistent with the trained model. Therefore, coefficient scaling and shifting are not optional add-ons but necessary parts of the quantized inference pipeline. They convert high-precision intermediate values into bounded-width activation values, allowing the accelerator to maintain reasonable inference accuracy even on resource-constrained platforms.

In terms of runtime behavior, the proposed design computes only one neural-network layer at a time. After the current layer finishes, its output result is first written back to the activation SRAM, and the next layer is then started. This process continues until the complete forward inference of the entire network is finished. Although this sequential execution strategy does not pursue inter-layer

parallelism, it greatly simplifies inter-layer dependencies and memory organization. The output of one layer naturally becomes the input of the next layer, thus forming a clear and easily verifiable layer-by-layer execution chain. For resource-constrained edge hardware platforms, this approach makes it possible to complete the full neural-network inference task while keeping control complexity and implementation cost manageable.

4.3 Memory Hierarchy and Data Reuse

The memory hierarchy and data reuse strategy are critical components of the proposed MFCC hardware architecture, as they directly impact both performance and energy efficiency.

At the top level, the SRAM modules and the shared multiplier are organized as parallel components alongside the control and computation modules. This architectural choice enables the SRAMs to support both internal accesses from the computation module and external accesses for data initialization and configuration. In particular, the top-level connectivity allows the data SRAM to be initialized with input speech frames and the parameter SRAM to be preloaded with coefficients directly from external sources. Similarly, the shared multiplier can also be externally accessed or configured when required, enabling flexible integration with the neural network backend.

Two separate SRAM blocks are adopted, namely a data SRAM and a parameter SRAM, to store intermediate data and precomputed coefficients, respectively. This separation provides several advantages. First, it allows simultaneous access to both data and parameters within the same clock cycle. As a result, the two operands required for multiplication can be fetched in parallel and directly forwarded to the multiplier in the next cycle, eliminating the need for additional register buffering of one operand. This effectively reduces both latency and register overhead.

Second, separating data and parameter storage simplifies memory access patterns and improves scheduling efficiency. The data SRAM primarily follows dynamic and state-dependent access patterns, while the parameter SRAM is accessed in a more regular and predictable manner due to its static, precomputed contents. By decoupling these two types of memory accesses, contention is avoided and the overall datapath becomes more regular and easier to control.

Third, the parameter SRAM remains unchanged after initialization under fixed system configurations, such as sampling rate and frame length. This read-only characteristic enables further optimization opportunities, such as simplified control logic, reduced write activity, and lower dynamic power consumption.

From a data reuse perspective, the architecture is designed to maximize reuse of both input data and coefficients within each processing stage. Intermediate results stored in the data SRAM are reused across multiple computation steps, while coefficients stored in the parameter SRAM are repeatedly accessed across frames without recomputation. This reuse strategy significantly reduces redundant memory accesses and arithmetic operations, contributing to improved energy efficiency.

Overall, the proposed memory hierarchy, combined with explicit data and parameter separation, supports efficient operand delivery, reduces unnecessary buffering, and enhances both performance and hardware utilization under strict resource constraints.

4.4 MFCC Optimization Strategies

To achieve an efficient balance among hardware cost, power consumption, and real-time performance, several optimization strategies are applied at both the algorithm and architecture levels. These optimizations target different aspects of the MFCC processing flow, including FFT configuration, memory organization, arithmetic simplification, and feature extraction efficiency.

4.4.1 FFT Size Optimization

One important optimization concerns the selection of frame length and FFT size. Originally, the commercial baseline reference mandated a sampling rate (f_s) of 16 kHz, a frame length of 40 ms, and a frame rate of 25 frames per second. Under these specification constraints, the number of discrete acoustic samples contained within a single speech frame is calculated as:

$$N_{\text{sample}} = 16\,\text{kHz} \times 40\,\text{ms} = 640. \tag{4.2}$$

Crucially, a frame size of 640 is not a power of two. From a hardware mapping perspective, implementing an FFT engine tailored for a non-power-of-two sequence introduces severe microarchitectural bottlenecks. Designers are forced to either zero-pad the sequence up to a 1024-point boundary—which incurs substantial waste in on-chip SRAM storage and unnecessary clock cycles—or construct highly complex, non-regular mixed-radix conversion pipelines. Arbitrary FFT sizes necessitate multi-bit index wrapping operations that rely on modulo computations ($\pmod N$). In hardware, modulo operations introduce heavy timing paths, as they require iterative division-like arrays or multiple conditional subtraction structures, both of which severely inflate the silicon area, complicate the control paths, and throttle the operating frequency.

To eliminate these hardware overheads at the source, this work proposes an algorithm-hardware co-optimization that redefines the front-end frame parameters. Through comprehensive collaboration with the algorithmic verification team, the baseline specifications were carefully adjusted. The optimized framework supports dual sampling rates of 8 kHz and 16 kHz, but standardizes the frame length to 32 ms and the frame rate to 32 frames per second. Consequently, the points per frame are cleanly reconfigured to exactly 256 (for 8 kHz sampling) or 512 (for 16 kHz sampling), both of which are perfect powers of two (2^8 and 2^9, respectively).

By ensuring a strict power-of-two constraint across all operational modes, the expensive modulo index computations are entirely bypassed. Furthermore, the standardized radix-2 DIF FFT algorithm can be directly applied to the hardware datapath without any irregular routing control. Supported by extensive algorithmic simulation and fixed-point bit-true verification given by software department

confirmed that this parametric adjustment introduces negligible recognition accuracy loss to the back-end DSCNN classifier. Ultimately, this hardware-driven parametric optimization radically simplifies the control finite state machine (FSM), reduces arithmetic complexity, eliminates SRAM space fragmentation, and improves the overall timing efficiency of the KWS system.

This optimization simplifies control logic, reduces arithmetic complexity, and improves overall timing efficiency.

4.4.2 Memory Size and Organization Trade-off

Another critical optimization involves the selection of SRAM size and organization, which directly impacts storage cost, timing efficiency, and power consumption.

In the proposed design, all data are represented with a fixed bit-width of N. To prevent excessive growth in numerical range, both multiplication and accumulation results are truncated, and the final results written back to SRAM are constrained to N bits. During FFT computation, complex-valued operations are required, meaning that both real and imaginary components must be stored, accessed, and processed. This introduces a trade-off among computation speed, memory access frequency, and storage area.

For the parameter SRAM, although complex coefficients are required in the FFT stage, this portion represents only a small fraction of the total parameter set. Therefore, expanding the parameter SRAM from N bits to $2N$ bits would result in inefficient memory utilization. As a result, the parameter SRAM maintains an N-bit width, and complex coefficients are handled through multiple sequential accesses when necessary.

For the data SRAM, storing complex data introduces additional storage requirements. For a maximum frame length of M, an additional $M \times N$ bits are required to store the imaginary components. This leads to two possible architectural configurations:

- **Option 1:** An N-bit width configuration with a depth of $2M$.
- **Option 2:** A $2N$-bit width configuration with a depth of M.

In Option 1, the real and imaginary parts must be accessed sequentially through separate clock cycles, resulting in a higher total memory access count and increased dynamic switching activity. In Option 2, the wordline width is doubled to $2N$ bits, allowing a complete complex data pair (real and imaginary) to be fetched or stored within a single clock cycle. However, this wider memory configuration increases the static power dissipation due to the larger bit-cell arrays and heavier peripheral circuitry, such as sense amplifiers and wordline drivers.

To determine the optimal configuration, a quantitative evaluation was conducted based on the system parameters $N = 16$ and $M = 512$, as summarized in Table 4.1.

As indicated by the experimental results in Table 4.1, although the $2N$-bit wide SRAM configuration (Option 2) exhibits a higher standby leakage current and larger static power ($0.408\,\mu$W compared to $0.311\,\mu$W), it dramatically slashes the overall execution cycle count by over 93% (from $9,552,448$ down to $665,696$ cycles). This massive cycle reduction enables a highly efficient "race-to-sleep"

Table 4.1: SRAM power comparison between 16-bit × 1024 and 32-bit × 512 options

SRAM	Standby	Read	Write	Access count	Area (μm^2)	Active power (μW)	Total power (μW)
16×1024	0.311	1.586	1.552	9 552 448	8815.2660	0.03653025	1.479967994
32×512	0.408	2.581	2.373	665 696	10 513.9092	0.33283220	1.473063050
40 nm HVT single-port SRAM							1.1 V

mechanism, allowing the hardware accelerator to finish the processing pipeline significantly faster and remain in a low-power standby state for a longer duration. Consequently, despite the higher leakage per cycle, the integrated total energy consumption of the $2N$-bit configuration ($1.473\,\mu$J) is slightly lower than that of the N-bit configuration ($1.480\,\mu$J). Therefore, considering the identical total power level but orders-of-magnitude faster execution velocity, the $2N$-bit dual-component data SRAM organization (Option 2) is definitively selected for this design, ensuring a superior throughput-to-energy metric for always-on edge deployment.

4.4.3 FFT Algorithm Optimization

In the FFT computation, the complex multiplication between the butterfly intermediate subtraction result and the twiddle factor constitutes the primary arithmetic bottleneck. Let the intermediate complex operand and the deterministic twiddle factor be represented as:

$$X_B = a + bj, \tag{4.3}$$

$$W_N^k = \cos\left(\frac{2\pi k}{N}\right) - j\sin\left(\frac{2\pi k}{N}\right). \tag{4.4}$$

A conventional complex multiplication expands directly into four real-number multiplications and two real-number additions:

$$\mathrm{Re} = a\cos\left(\frac{2\pi k}{N}\right) + b\sin\left(\frac{2\pi k}{N}\right), \tag{4.5}$$

$$\mathrm{Im} = b\cos\left(\frac{2\pi k}{N}\right) - a\sin\left(\frac{2\pi k}{N}\right). \tag{4.6}$$

To minimize the precious silicon area under a strict shared single-multiplier constraint, the algebraic formulation is re-mapped into a three-multiplier structure by introducing precomputed twiddle combinations:

$$\mathrm{Re} = (a+b)\cos\left(\frac{2\pi k}{N}\right) + b\left[\sin\left(\frac{2\pi k}{N}\right) - \cos\left(\frac{2\pi k}{N}\right)\right], \tag{4.7}$$

$$\mathrm{Im} = (a+b)\cos\left(\frac{2\pi k}{N}\right) - a\left[\sin\left(\frac{2\pi k}{N}\right) + \cos\left(\frac{2\pi k}{N}\right)\right]. \tag{4.8}$$

This mathematical transformation effectively reduces the number of real multipliers from four to three. The trade-off introduces additional pre-butterfly additions, which are exceptionally cheap in terms of hardware gating logic, yielding a significant net reduction in dynamic power and routing congestion.

Crucially, this operator-level reduction is tightly coupled with a microarchitectural pipelining optimization enabled by the selected Radix-2 DIF topology. In the proposed DIF datapath, after fetching two target operands (D_A and D_B) from the data SRAM, the butterfly node simultaneously evaluates the addition ($D_A + D_B$) and the subtraction ($D_A - D_B$). Because the hardware instantiates separated, decoupled execution pathways for memory writeback and arithmetic multiplication, these two distinct branches can be processed concurrently:

1. **Addition Writeback Pathway:** The newly evaluated addition value requires no further scaling or twiddle rotation. It is routed directly to the memory interface control logic and written back into the original SRAM address to implement in-place computation.
2. **Subtraction Multiplication Pathway:** Concurrently, during the exact same clock cycles where the addition value is performing its SRAM writeback transaction, the subtraction result components (a and b) are streamed directly into the shared multiplier array to execute the three-multiplier complex calculation with W_N^k.

By fully overlapping the memory writeback latency of the addition branch with the active computation cycles of the subtraction-twiddle multiplier, the datapath successfully hides the multiplication overhead. Consequently, for each individual butterfly group, the execution latency is compressed by exactly 1 clock cycle (clk). Across a full 512-point FFT execution framework which spans $\log_2(512) = 9$ cascaded hardware stages, each stage contains 256 independent butterfly groups. Therefore, this co-designed pipeline scheduling successfully slashes the overall processing latency by:

$$\Delta T_{\text{latency}} = 256\,\text{butterflies/stage} \times 9\,\text{stages} \times 1\,\text{clk/butterfly} = 2304\,\text{clk}. \quad (4.9)$$

This architectural optimization guarantees that the hardware achieves a superior throughput and an ultra-low execution timeline, allowing the KWS front-end processor to minimize its active energy profile and enter standby mode significantly earlier.

4.4.4 Mel Filter Optimization

In conventional MFCC processing, Mel filtering is implemented using triangular filters:

$$E_m[i] = \sum_k P_m[k] \cdot H_i[k]. \quad (4.10)$$

This requires multiplication for each coefficient. In the proposed design, a simplified scheme is adopted by directly aggregating energy within Mel-scale bands:

$$E_m[i] = \sum_{k \in \mathcal{B}_i} P_m[k]. \quad (4.11)$$

Since the Mel scale already defines the perceptual frequency partitioning, the triangular weighting is not strictly necessary[8]. This approximation removes multiplication operations and reduces the filtering stage to addition-only computation, significantly lowering hardware complexity.

Software evaluation shows that this simplification introduces negligible accuracy degradation.

4.4.5 Magnitude and Filtering Fusion

The magnitude spectrum is conventionally computed as:

$$|S_m[k]| = \sqrt{\mathrm{Re}\{S_m[k]\}^2 + \mathrm{Im}\{S_m[k]\}^2}. \tag{4.12}$$

However, subsequent processing operates on the squared magnitude:

$$P_m[k] = |S_m[k]|^2 = \mathrm{Re}\{S_m[k]\}^2 + \mathrm{Im}\{S_m[k]\}^2. \tag{4.13}$$

Therefore, the square-root operation is redundant and can be eliminated. The filtering stage can directly operate on $P_m[k]$:

$$E_m[i] = \sum_{k \in \mathcal{B}_i} P_m[k]. \tag{4.14}$$

This removes the need for square-root hardware, reducing area and latency. Moreover, it converts a nonlinear operation into a sequence of additions and multiplications, which are more suitable for fixed-point hardware implementation.

4.4.6 Summary od MFCC Optimization

Overall, the proposed optimization strategies systematically reduce arithmetic complexity, memory overhead, and power consumption. By transforming multiplication- and nonlinear-intensive operations into addition-dominated computations and leveraging memory-efficient data organization, the design achieves a highly efficient hardware implementation of MFCC suitable for always-on KWS systems.

4.5 DSCNN Optimization Strategies

1) layer-reusable computation engine The first optimization method is to use a layer-reusable computation engine instead of assigning a fully independent hardware block to each layer. Prior accelerator studies usually discuss two implementation styles: a streaming architecture, where each layer has dedicated hardware, and a single computation engine, where different layers share the same computation resources. For small always-on KWS ASICs, a shared computation engine is more suitable because the FPS requirement is relaxed and area/power are the primary objectives. The same PE array or MAC datapath can be configured for standard convolution, depthwise convolution, pointwise convolution, average pooling, and the final fully connected layer. This reduces duplicated arithmetic units and improves area efficiency.

2) separate dataflow modes In depthwise convolution, each channel is independent, so the accelerator can process one channel at a time or a small group of channels with simple address generation. The main goal is to reuse the local spatial window and avoid unnecessary channel mixing. In pointwise convolution, the kernel size is 1×1, so there is no spatial window reuse in the same way; the computation becomes a dot product across channels for each output position. Therefore, pointwise convolution benefits more from input-channel tiling, output-channel tiling, and accumulator reuse. A good DSCNN accelerator should not use exactly the same schedule for both operations. Instead, it should switch between depthwise mode and pointwise mode with different loop orders and buffer reuse strategies.

3) reduce memory traffic through local buffers Several MobileNet accelerator works show that depthwise separable networks are often limited by memory bandwidth rather than pure computation. This is especially important for ASIC design because SRAM area and memory access energy may dominate the total cost [3]. Weight SRAM, input activation buffer, output activation buffer, and partial-sum buffer should be organized to maximize reuse. Ping-pong buffering can overlap data loading, computation, and result writing, while tiling allows the accelerator to process a feature map in smaller blocks that fit into local SRAM. For DSCNN-based KWS, feature maps are much smaller than image-classification MobileNet, so an ASIC design can exploit this advantage by keeping most intermediate activations on chip and avoiding frequent off-chip access.

4) low-precision fixed-point computation MobileNet and DSCNN accelerators commonly use 8-bit or 16-bit integer/fixed-point arithmetic instead of floating point to reduce multiplier area, SRAM size, and switching activity. Recent MobileNet accelerator work reports that 8-bit quantization can preserve accuracy with only a small loss while enabling hardware-friendly fixed-point operations [11]. For this thesis, the exact bit width should be determined by the quantization experiment of the selected DSCNN model. The hardware description should therefore keep weight bit width, activation bit width, and accumulator bit width as explicit design parameters.

5) pipeline adjacent operations The fifth optimization method is to fuse or pipeline adjacent operations when the data dependency allows it. In DSCNN, a depthwise layer is usually followed by a pointwise layer. If the intermediate feature map between them is written completely to SRAM and then read back, memory traffic increases. A more efficient schedule can stream partial results from the depthwise stage to the pointwise stage or store only a small tile of intermediate data. For a low-throughput KWS ASIC, full inter-layer pipelining may not be necessary, but partial fusion inside a depthwise-pointwise block can reduce read/write cycles and dynamic power.

4.5.1 Comparison of Memory-Access Optimization Schemes

Because the proposed accelerator is deployed at the edge, the most important hardware target is low power. The PPA priority of this work is therefore different from high-throughput accelerators. Performance is treated as a sufficient-margin constraint: according to the algorithmic analysis in Chapter 2, the target FPS can be satisfied even with only one multiplier, so performance is not the main bottleneck at this stage. Power must be emphasized because the KWS module is expected to operate in an always-on manner. Area is also important, and in this design a large portion of the area is occupied by SRAM. Therefore, reducing activation SRAM size and unnecessary SRAM access can reduce both area and power.

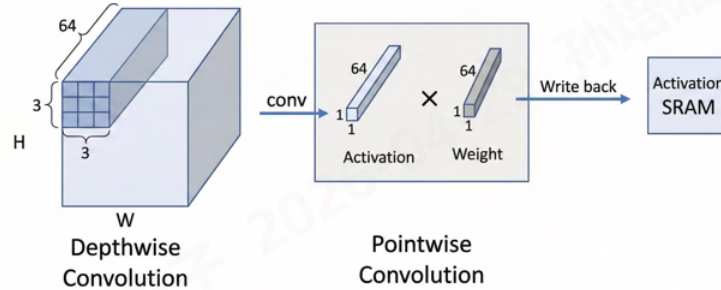


Figure 4.4: Layer fusion scheme: the intermediate activation between two adjacent layers is kept in registers and directly forwarded, reducing activation SRAM traffic.

Based on this objective, two memory-access optimization schemes are considered: layer fusion and SRAM resource reuse with line buffering. Layer fusion attempts to keep the output of the first layer in registers and directly use it as the input of the next layer. In this way, the design avoids writing the intermediate activation of the first layer into SRAM and then reading it back for the second layer. The basic idea follows the general conclusion of layer-fusion research: keeping intermediate data closer to the computation unit can reduce expensive memory movement. In this work, however, the target is more aggressive. Instead of using on-chip SRAM to replace off-chip memory traffic, the design investigates whether register storage can replace part of the on-chip SRAM activity.

As Figure 4.4 illustrates, the intermediate activation between two adjacent layers is forwarded through registers, avoiding a write–read round trip to activation SRAM.

The second scheme is SRAM resource reuse with line buffering. Instead of allocating separate activation SRAM regions for all intermediate tensors, the output activation is written back to SRAM locations that are no longer used by the input activation. This overwriting strategy reduces the required activation SRAM capacity.

In addition, line buffers are introduced to reuse recently generated data and

reduce repeated SRAM access. For depthwise convolution, a small output line buffer can delay write-back and support local reuse of neighboring spatial results.

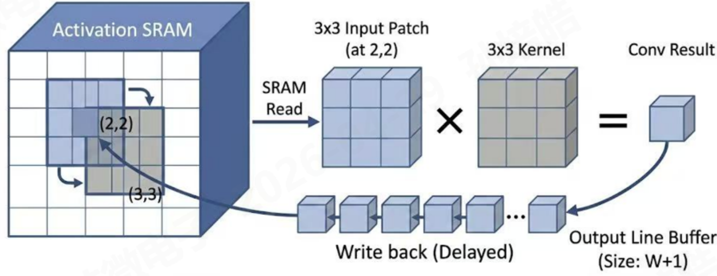


Figure 4.5: Depthwise-convolution line-buffer scheme for delaying write-back and enabling local reuse of neighboring spatial results.

As Figure 4.5 illustrates, a small depthwise line buffer delays write-back and enables local reuse of neighboring spatial results.

For pointwise convolution, a register file can temporarily hold the $1 \times 1 \times C$ activation vector, so that multiple 1×1 kernels can reuse the same input activation without repeatedly reading it from SRAM.

As Figure 4.6 illustrates, holding the $1 \times 1 \times C$ activation vector in a small register file allows multiple pointwise kernels to reuse the same input activation with fewer SRAM reads.

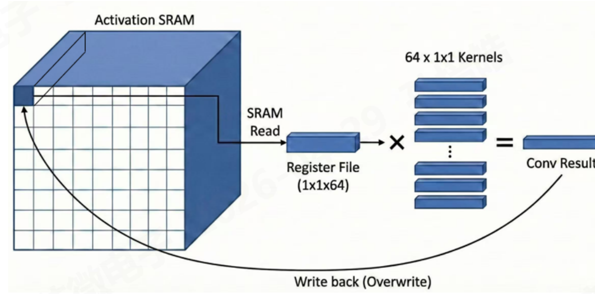


Figure 4.6: Pointwise-convolution register-buffer scheme for reusing the $1 \times 1 \times C$ activation vector across multiple 1×1 kernels.

Table 4.2 compares the access counts and activation SRAM size of the two schemes. The layer-fusion scheme reduces activation SRAM write count because intermediate results can be passed through registers. However, it introduces a large number of register reads and still requires a larger activation SRAM capacity. The SRAM reuse with line buffering scheme reduces weight SRAM reads, register reads, and activation SRAM size. Although it has more activation SRAM reads/writes

and slightly more register writes, the reduction in SRAM capacity is important for this low-power ASIC design because SRAM occupies a large area and contributes to both static and dynamic power.

Table 4.2: Comparison of memory-access optimization schemes

Metric	Layer fusion	SRAM reuse + line buffer
Weight SRAM read	728064	639552
Activation SRAM read	89856	99840
Activation SRAM write	9984	19968
Register read	1288512	718848
Register write	648960	708864
Activation SRAM size	20478	10494

Overall, SRAM reuse with line buffering is more suitable for the current design. The reason is that the proposed accelerator does not need to maximize throughput, but must reduce power and area. The line-buffer scheme reduces the activation SRAM size from 20478 to 10494, which directly supports the area objective. It also reduces weight SRAM reads and register reads, which helps lower access energy. Therefore, the following hardware implementation uses SRAM resource reuse as the main memory organization strategy and uses small register or line buffers only where they provide clear data reuse.

4.5.2 SRAM Power Evaluation and Technology Selection

Since a large portion of the system power is related to SRAM, the SRAM power can be used as an early indicator for evaluating the overall chip power trend. This is reasonable for the current design because the computation core contains only one INT8 multiplier, while weight and activation storage still require relatively large SRAM arrays. Therefore, before final layout-level power estimation, this work first compares the SRAM power under different process options and memory organization schemes.

The first comparison is between 40 nm and 28 nm SRAM implementations. Although 28 nm SRAM has a smaller area, the evaluated 28 nm HVT SRAM shows much higher standby and leakage-related power. As a result, the total SRAM power in 28 nm is higher than that in 40 nm. For the considered activation and weight SRAM configuration, the total SRAM power is about 40.01 μW in 40 nm, while the 28 nm case reaches about 85.38 μW. Therefore, the following hardware design adopts the 40 nm process option.

After selecting 40 nm, the activation SRAM power and area of the two memory-access schemes are compared. The layer-fusion scheme uses an activation SRAM with address depth 20480 and area 79313.58. The line-buffer scheme reduces the required activation SRAM address depth to 11264 and area to 41046. It also reduces standby, read, and write power per MHz. This confirms that the line-buffer scheme is more aligned with the low-power and small-area objective.

To include register activity in the scheme-level power estimation, the register power is estimated using the standard-cell register power data. Assuming the

Table 4.3: SRAM power comparison between 40 nm and 28 nm options

Process	SRAM	Standby	Read	Write	Area	Total power
40 nm HVT	Activation	3.206	4.419	4.561	79 313.58	20.1640
40 nm HVT	Weight	3.350	4.493	4.647	84 774.99	19.8485
28 nm HVT	Activation	29.327	3.261	3.423	39 953.23	41.8468
28 nm HVT	Weight	31.207	3.357	3.528	42 331.13	43.5354
Total power, 40 nm						40.0125 μW
Total power, 28 nm						85.3822 μW

Table 4.4: 40 nm activation SRAM comparison for different memory schemes

Scheme	Volt	Standby	DC	Read	Write	ADDR	Area
Layer fusion	1.1	3.206	3.476	4.419	4.561	20480	79313.58
Line buffer	1.1	1.447	1.748	2.593	2.593	11264	41046.00

read/write toggle rate is 0.5, the total power of n 8-bit registers is estimated as

$$W_{\text{reg}} = n \times 0.005 + 1.19r + 1.99w \quad (\mu\text{W}), \tag{4.15}$$

where n is the number of 8-bit registers, r is the normalized read activity, and w is the normalized write activity. With one frame as the evaluation unit, they are calculated as

$$r = \frac{\text{read cycles} \times \text{frame}}{10^8}, \qquad w = \frac{\text{write cycles} \times \text{frame}}{10^8}. \tag{4.16}$$

Using this estimation, the total power of the two schemes is shown in Table 4.5. The line-buffer scheme reduces total power from 23.2529 μW to 18.9406 μW. Therefore, it is selected as the preferred memory-access scheme.

Table 4.5: Power comparison of layer fusion and line-buffer schemes

Scheme	Register power	SRAM power	Total power
Layer fusion	0.7529	22.50	23.2529
Line buffer	0.4906	18.45	18.9406

4.5.3 Weight SRAM Splitting Exploration

Another observation is that the high SRAM dynamic power is partly caused by long address lines and large monolithic SRAM arrays. If one large SRAM is divided

into several smaller SRAM blocks, the access capacitance and address-line activity of each access may be reduced [13]. Therefore, this work further evaluates several weight SRAM splitting strategies. The original weight SRAM has a word depth of 22016. The evaluated options include keeping it as one SRAM, splitting it into two 11008-depth SRAMs, or using combinations of 8192, 5632, and other smaller SRAMs.

The evaluation shows that splitting the original 22016-depth weight SRAM into two 8192-depth SRAMs and one 5632-depth SRAM gives the lowest estimated power, about 8.7328 μW. However, this configuration increases the total SRAM area to about 91027.74, which is around 20000 higher than the original single-SRAM area. This area increase may become more significant after back-end implementation because additional spacing, isolation, or routing overhead may be required around multiple SRAM macros. Therefore, the weight SRAM splitting scheme is not fixed at this stage. It will be further considered during the back-end design stage.

Table 4.6: Weight SRAM splitting exploration

Configuration	Power (μW)	Area (μm^2)	Comment
1×22016	14.0094	73 020.17	Baseline single SRAM
2×11008	9.9989	80 584.00	Lower power, larger area
$1 \times 16384 + 1 \times 5632$	11.0024	80 490.76	Intermediate option
$2 \times 8192 + 1 \times 5632$	8.7328	91 027.74	Lowest power, area increases

4.5.4 SRAM-Level Power and Area Comparison

Finally, the pure SRAM read/write power before and after the proposed memory optimization is evaluated. This comparison only considers the data-movement-related SRAM power and area. The multiplier, control logic, and other peripheral circuits are not included in this estimation. Therefore, the result should be interpreted as the SRAM-level optimization effect rather than the final full-chip power. Even under this limited scope, the improvement is significant.

Before optimization, the worst-case SRAM configuration has an area of 164087 μm^2 and an estimated power of 40.012 μW. After applying the selected memory optimization strategy, including SRAM resource reuse, line buffering, and SRAM splitting consideration, the SRAM-related area is reduced to 134798 μm^2 and the estimated SRAM power is reduced to 11.327 μW. This corresponds to an area reduction of about 18% and a power reduction of about 71%. These results confirm that memory organization is one of the most important factors in the proposed low-power KWS ASIC.

Chapter 5

Results and Discussion

This section presents the comprehensive hardware implementation details and experimental evaluation of the proposed unified KWS accelerator. In this work, the development roadmap initiates with a comprehensive architectural exploration, follows with systematic algorithmic mathematical modeling, and ultimately culminates in physical hardware implementation. Consequently, it should be noted that the baseline algorithmic validation and precision comparison results discussed below, which do not represent the final hardware bit-true metrics of this design, are provided by the cooperative algorithmic engineering department. The physical hardware is implemented in a standard 40 nm HVT CMOS process under a supply voltage of 1.1 V. The evaluation is categorized into four primary design dimensions: power consumption, silicon area, execution latency, and algorithmic classification accuracy. Finally, a state-of-the-art (SOTA) comparison table is provided to highlight the overall engineering efficiency of this work.

5.1 Power Consumption

By evaluating the full KWS accelerator profile in Table 5.2 and 5.1, the clear power characteristics of front-end design can be accurately extracted. The active execution power of the pure MFCC front-end is evaluated at 645.50 μW.

The specific power distributions inside the front-end processing elements are driven by two main components:

1. **Combinational Power Exploration:** The combinational logic power of the pure front-end core stands at 196.80 μW. This computational weight is directly tied to the high-density arithmetic requirements of the Radix-2 DIF FFT stages and the subsequent logarithmic piecewise linear fitting loops. Because the system utilizes a multiplier-centric shared paradigm to multiplex the back-end multipliers, this combinational overhead has been drastically optimized. It represents the optimized net scaling footprint after rejecting the unrolled fully-pipelined CORDIC layout, avoiding heavy logic congestion.
2. **Memory Access:** The memory-related power dissipation of the pure pre-processing module stands at 149.90 μW. This represents one of the most prominent dynamic sectors inside the MFCC core. This power behavior

perfectly validates our design choice described in the memory architecture selection: the streaming short-term Fourier transform (STFT) demands frequent, continuous data fetching and in-place writeback operations inside the dual-port frame buffer. Thanks to the standardization of the word width to a $2N$-bit layout, the memory access frequency is heavily compressed, ensuring that while the instant dynamic read/write power is noticeable, the long-term accumulated energy remains strictly minimal.

Additionally, the leakage standby power of the front-end core consumes only a minor fraction of 5.48 μW (accounting for just 0.41% of the total power). This exceptionally tiny static footprint directly demonstrates the massive architectural advantage of selecting a 40 nm HVT cell library, completely choking off the subthreshold leakage current during idle intervals and securing a prolonged battery lifecycle for continuous, always-on voice wake-up deployment.

Table 5.1: Post-Synthesis Power Summary of the Isolated MFCC Front-end Core

Power Component	Power (μW)	Percentage
Cell internal power	404.10	62.61%
Net switching power	236.20	36.59%
Cell leakage power	5.18	0.80%
Total power	645.50	100.00%

Table 5.2: Power Distribution by Power Group for MFCC Front-end Core (μW)

Power Group	Internal	Switching	Leakage	Total Power
Clock network	182.40	107.70	0.00001	290.10
Register	4.10	4.26	0.07	8.42
Combinational logic	72.56	124.00	0.21	196.80
Memory	145.10	0.00014	4.52	149.80
Black box	0.00	0.00	0.37	0.37
Total	404.10	236.20	5.18	645.50

Tables 5.3 and 5.4 report the post-synthesis power breakdown of the isolated back-end DSCNN core. The total active power is 1324.00 μW, dominated by dynamic components (cell internal and net switching), while leakage remains negligible. The group-level distribution further shows that the clock network, combinational logic, and SRAM-related memory activity account for the majority of the back-end power budget.

From a micro-architectural perspective, the back-end DSCNN stage is expected to be power-intensive due to two key factors. First, the back-end SRAM is configured with a much deeper storage depth, so memory access and peripheral activity

occupy a substantial fraction of the power budget, consistent with the prominent *Memory* contribution observed in Table 5.4. Second, convolution introduces relatively complex address generation and data scheduling, increasing the depth and toggling of the combinational path and thus raising logic power, which aligns with the dominant *Combinational logic* term in Table 5.4. Moreover, compared with MFCC (where some stages are lightweight), the neural network is more compute-intensive and is typically implemented with a pipelined datapath that keeps multiple stages concurrently active, leading to higher sustained dynamic power.

Table 5.3: Post-Synthesis Power Summary of the Isolated Back-end DSCNN Core

Power Component	Power (μW)	Percentage
Cell internal power	887.30	67.04%
Net switching power	430.80	32.55%
Cell leakage power	5.48	0.41%
Total power	1324.00	100.00%

Table 5.4: Power Distribution by Power Group for Back-end DSCNN Core (μW)

Power Group	Internal	Switching	Leakage	Total Power
Clock network	273.50	135.90	0.00001	409.50
Register	7.81	8.96	0.07	16.84
Combinational logic	180.70	285.70	0.21	466.60
Memory	425.20	0.00024	4.82	430.30
Black box	0.00	0.00	0.37	0.37
Total	887.30	430.80	5.48	1324.00

To account for duty cycling, P_{avg} is computed as a cycle-weighted average between the back-end-only phase ($P_{\text{back}} = 645.50\,\mu\text{W}$) and the full pipeline phase ($P_{\text{full}} = 1324.00\,\mu\text{W}$), using the measured cycle counts over one execution window. P_{total} is then derived by scaling P_{avg} with the total runtime factor in (5.3).

$$P_{\text{avg}} = P_{\text{back}} \times \left(\frac{\text{MFCC cal cycle}}{\text{Total cal cycle}}\right) + P_{\text{full}} \times \left(\frac{\text{DSCNN cal cycle}}{\text{Total cal cycle}}\right) \tag{5.1}$$

$$P_{\text{avg}} = 645.50 \times \left(\frac{409668}{2684020}\right) + 1324.00 \times \left(\frac{2684020 - 409668}{2684020}\right) = 1220.44\,\mu\text{W} \tag{5.2}$$

$$P_{\text{total}} = P_{\text{avg}} \times \frac{2684020}{100 \times 10^6} = 32.757\,\mu\text{W} \tag{5.3}$$

This corresponds to an active-window average power of $P_{\text{avg}} = 1220.44\,\mu$W while the accelerator is performing MFCC extraction and DSCNN inference. After incorporating the duty-cycled execution over a 100 MHz reference clock, the resulting long-term average power reduces to $P_{\text{total}} = 32.757\,\mu$W. In other words, the design spends only a small fraction of time in the high-activity computation mode and remains mostly in low-activity/idle states, making the overall energy budget well within the practical constraints of always-on, battery-powered keyword-spotting nodes.

5.2 Area

Table 5.5 summarizes the post-synthesis area report of the proposed KWS accelerator. The total synthesized cell area is 132,731.94, including both standard-cell logic and SRAM macro area. The result shows that the overall area is strongly dominated by memory macros rather than computation logic.

Table 5.5: Post-synthesis area summary of the KWS accelerator

Component	Area	Percentage
Total cell area	132,731.94	100.0%
Combinational logic area	9,572.54	7.2%
Sequential logic area	5,500.91	4.1%
SRAM macro / black-box area	117,658.50	88.6%
DSCNN top module	83,289.74	62.8%
MFCC top module	22,452.31	16.9%
DSCNN control module	7,013.41	5.3%
Multiplier module	1,225.01	0.9%
SRAM arbiter	34.07	<0.1%

As shown in Table 5.5, the total synthesized cell area of the complete KWS design is 132,731.94. Among all components, SRAM macros occupy the largest portion of the area, with a total macro area of 117,658.50, corresponding to approximately 88.6% of the total cell area. This indicates that the area cost of the proposed accelerator is mainly determined by the on-chip memory storage rather than the arithmetic datapath.

At the module level, the DSCNN top module is the largest contributor, occupying 83,289.74, or 62.8% of the total area. This is mainly caused by the weight SRAM inside the DSCNN module. In comparison, the multiplier module only occupies 1,225.01, which accounts for less than 1% of the total area. Therefore, the multiplier and control logic are not the dominant area bottlenecks in this design.

The MFCC front-end module occupies 22,452.31, corresponding to 16.9% of the total area. Its area is also partly contributed by internal SRAMs used for data buffering and lookup-table storage. The DSCNN control module occupies 7,013.41, or 5.3% of the total area, which is acceptable considering that it contains

the finite-state control logic, mode selection, and scheduling logic for the DSCNN computation.

5.3 Timing

Table 5.6: KWS latency breakdown for each computation stage

Stage	Description	Latency (clk)
Overall KWS	From simulation start to dscnn_finish	2684020
Initial	Write data into srams	2539
state_idle	Waiting to start to state_stft	1
state_stft	STFT computation	7720
state_window	Windowing operation	473
state_magnitude	Magnitude computation	767
state_filter	Filter bank operation	432
state_ln	Logarithm/normalization stage	279
state_dct	Activation preprocessing stage	2001
state_done	Waiting before CL starts	409668
CL	Convolution layer	272135
DWCL_0	Depthwise convolution layer 0	70728
PWCL_0	Pointwise convolution layer 0	424428
DWCL_1	Depthwise convolution layer 1	70728
PWCL_1	Pointwise convolution layer 1	424428
DWCL_2	Depthwise convolution layer 2	70728
PWCL_2	Pointwise convolution layer 2	424428
DWCL_3	Depthwise convolution layer 3	70728
PWCL_3	Pointwise convolution layer 3	424428
AP	Average pooling layer	6596
FCL	Fully connected layer	786

According to Table 5.6, the complete KWS inference requires 2,684,020 clock cycles from the beginning of preprocessing to `dscnn_finish`. Assuming a clock frequency of 100 MHz, the clock period is

$$T_{clk} = \frac{1}{100 \text{ MHz}} = 10 \text{ ns}.$$

Therefore, the latency of one complete inference is

$$T_{\text{frame}} = 2,684,020 \times 10 \text{ ns} = 26,840,200 \text{ ns} = 26.84 \text{ ms}.$$

The maximum frame rate can be estimated as

$$FPS_{\max} = \frac{1}{T_{\text{frame}}} = \frac{100,000,000}{2,684,020} \approx 37.26 \text{ frames/s}.$$

Thus, at 100 MHz, the proposed KWS design can process one frame in approximately 26.84 ms, corresponding to a maximum throughput of about 37.3 FPS.

From the latency breakdown, the pointwise convolution layers dominate the total computation time. Each PWCL stage requires 424,428 clock cycles, which is significantly larger than the corresponding DWCL stage of 70,728 clock cycles. This indicates that the pointwise convolution is the major computational bottleneck of the DSCNN inference. Among the neural network computation stages, the FCL stage has the smallest latency, requiring only 786 clock cycles. For the preprocessing pipeline, the `state_stft` stage accounts for the largest portion, while `state_ln` has the smallest latency among the explicitly listed preprocessing computation states. Overall, the latency is mainly determined by the repeated PWCL stages and the waiting period before the CL stage begins.

5.4 Accuracy

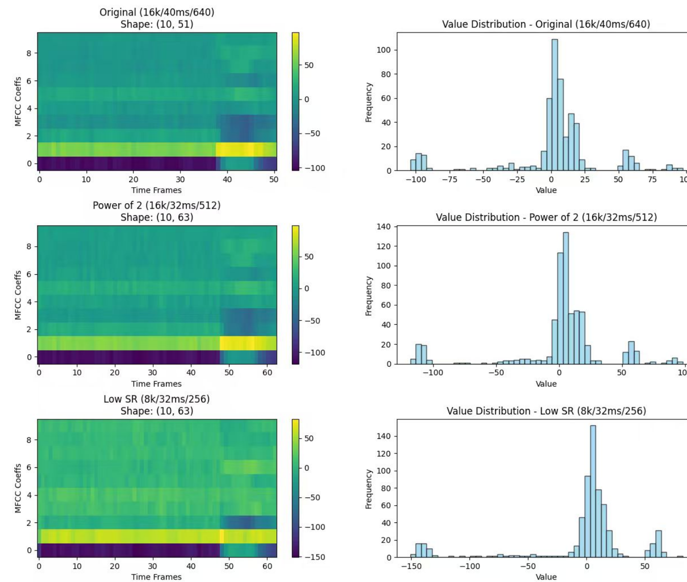


Figure 5.1: Mel Spectrogram and Value Distribution Comparison Across Different Sampling Rate and Frame Length

To ensure that the aggressive hardware-level simplifications and hardware-algorithm co-optimizations do not compromise the keyword recognition capabilities, a rigorous bit-true verification was conducted using a baseline voice dataset[18].

Once our back-end DSCNN model was finalized—which natively exhibits strong

noise robustness—the primary task shifted to ensuring that the various architectural simplifications introduced in the MFCC front-end would not trigger an accuracy crash at the classifier's output. Specifically, we needed to verify that the combined effects of altering the sampling rate/frame length, switching the Mel-filterbank from triangular to square waves, and applying piecewise linear log approximations would not cause the final keyword recognition rate to dive.

Figure 5.1(left column) illustrates the Mel-spectrograms generated by the algorithmic modeling across different acoustic sampling rates(8k and 16k) and framing parameters(52 and 64). Visually, minor differences and graphical discrepancies are noticeble between these spectrograms due to the parametric variations. However, as demonstrated by the quantitative data in Table 5.7, these visual micro-variations have negligible impact once processed through the deep layers of the DSCNN pipeline; the classification accuracy consistently holds strong at around 90%.

More significantly, when the Mel-filterbank shape is swapped from a classic triangular wave to a hardware-friendly square wave, it fundamentally slashes the dynamic power of the core multiplier array. Yet, the tracking accuracy only experience a microscopic nudge, shifting slightly from 92.66% down to 92.58%. In practical ASIC engineering, exchanging a mere 0.08% accuracy drop for such massive power savings is an exceptionally high-value trade-off.

Table 5.7: KWS Accuracy Comparison Across Different Window and Stride Configurations

Sampling Rate	Window Size (ms)	Window Stride (ms)	Test Accuracy (%)
16 kHz	40	40	90.70
16 kHz	32	32	88.71
8 kHz	32	32	92.68

Table 5.8: KWS Accuracy Comparison Across Different Mel Filter Mode (frame stride = 256)

Filter mode	Multiplier Usage (times)	Adder Usage (times)	Test Accuracy (%)
Triangle	128	40	92.66
Rectangle	0	40	92.58

Regarding the logarithmic piecewise approximation, we investigated the impacts of different partitioning strategies and segment counts. As summarized in Table 5.9, the non-linear quantile partitioning generally maintains a tighter accuracy envelope than the uniform linear alternative, with the final scores demonstrating slight fluctuations as the number of segments changes. To maximize design flexibility, this work implements a fully parameterized, custom-defined segment allocation framework. Silicon designers can dynamically configure the hardware to run on 4, 6, or 8 segments depending on the specific application scenario. Upon system boot or reconfiguration, the control logic automatically initializes

the lookup table (LUT) data inside the parameter SRAM based on the selected segment count, achieving a highly versatile and adaptive front-end execution.

Table 5.9: KWS Accuracy Comparison Across Different In Mode(frame stride=256)

Ln mode	Section Number (times)	Test Accuracy (%)
Uniform Section	4	89.66
Uniform Section	6	90.35
Uniform Section	8	91.39
Non-uniform Section	4	91.49
Non-uniform Section	6	92.58
Non-uniform Section	8	92.66

In addition to the hardware-implemented implementation analysis, the accuracy of different neural network back-end models is also evaluated under the same front-end preprocessing configuration. As shown in Table 5.10, all candidate models satisfy the target accuracy requirement, with test accuracy higher than 92%. These results are obtained before quantization during the front-end preprocessing stage, so the final accuracy is relatively high. Among them, TC-ResNet8 achieves the highest test accuracy of 95.09%, while DSCNN also maintains a comparable test accuracy of 94.59%.

Table 5.10: Comparison of candidate neural network models under the same preprocessing configuration

Model	Train Acc.	Valid Acc.	Test Acc.	MACs	Params
DSCNN	0.9725	0.9435	0.9459	3,684,928	23,180
TC-ResNet8	0.9827	0.9545	0.9509	1,596,096	65,148
BC-ResNet1	0.9351	0.9400	0.9402	3,340,252	9,160

From the perspective of computational complexity and model size, BC-ResNet1 has the smallest number of parameters, with only 9,160 parameters, and therefore appears attractive in terms of storage cost. However, further hardware-oriented analysis shows that BC-ResNet1 introduces more frequent intermediate feature-map access and read-write interaction during layer execution. This increases memory-access overhead and makes the scheduling and data reuse more complicated in the proposed hardware architecture.

Although DSCNN has a larger MAC count than the other two candidates, its depthwise separable convolution structure provides a more regular computation pattern and is easier to map onto the designed datapath and memory hierarchy. Therefore, considering the balance among accuracy, implementation regularity, memory-access behavior, and hardware scheduling complexity, DSCNN is selected as the final back-end neural network architecture in this work.

Chapter 6

Conclusion

6.1 Summary

This thesis presented the design and hardware-oriented optimization of a 40 nm ultra-low-power keyword spotting (KWS) system. The proposed solution combines an MFCC-based front-end with a depthwise separable convolutional neural network (DSCNN) back-end and targets always-on edge inference under strict power, area, and latency constraints. Rather than optimizing the algorithm in isolation, the work focused on co-designing the signal-processing pipeline, neural-network structure, and memory organization so that the complete system can be mapped efficiently onto a compact ASIC implementation.

The overall design flow starts from algorithmic modeling, then moves to hardware-aware optimization, and finally evaluates the full pipeline under a 40 nm HVT CMOS process. On the front-end, MFCC was selected because it provides a compact and perceptually meaningful representation while remaining amenable to fixed-point implementation. On the back-end, DSCNN was chosen after comparing several candidate KWS models, including TC-ResNet8 and BC-ResNet1. Although TC-ResNet8 achieved slightly higher recognition accuracy, DSCNN offered a better balance between accuracy, regularity, and hardware efficiency.

The implementation results show that the proposed design can meet the real-time requirement while maintaining competitive recognition performance. At 100 MHz, the full KWS pipeline completes one inference in 26.84 ms, corresponding to a maximum throughput of approximately 37.3 frames/s. The accuracy evaluation indicates that the MFCC simplifications do not cause a significant accuracy drop, and the selected back-end model achieves 92% test accuracy. In terms of power, the active-window average power during MFCC extraction and DSCNN inference is $P_{\text{avg}} = 1220.44\ \mu\text{W}$, and after incorporating duty-cycled execution over a 100 MHz reference clock, the long-term average power reduces to $P_{\text{total}} = 32.757\ \mu\text{W}$. These results demonstrate that a fully integrated MFCC+DSCNN KWS pipeline can be realized as a compact ASIC with both strong recognition quality and extremely low long-term power, making the design well-suited for always-on, battery-powered edge devices.

6.2 Research Questions Revisited

This section revisits the research questions introduced in Section 1.2 and summarizes how the thesis addresses them through the proposed 40 nm ultra-low-power keyword spotting (KWS) accelerator.

RQ1: How can an ultra-low-power keyword spotting accelerator be achieved, and which system-level techniques are most effective for minimizing energy consumption? An ultra-low-power KWS accelerator cannot be achieved by relying on a single circuit-level optimization. Instead, energy reduction must be treated as an end-to-end system objective across the algorithm, architecture, and circuit levels. At the algorithm level, a small and lightweight KWS model is the first step, because it directly reduces the amount of computation required by the following hardware and therefore lowers the total power consumption. At the architecture and circuit levels, the modeling and analysis in this thesis show that one multiplier is sufficient for the target workload. Based on this result, the accelerator can be built around a single-multiplier data path, and the surrounding hardware units can be minimized as much as possible. In addition, system-level techniques such as duty-cycle control and clock gating are highly effective for an always-on KWS system, because they suppress unnecessary switching activity when the accelerator is idle or only partially active.

RQ2: What are the key hardware-architecture trade-offs when implementing a KWS accelerator? The main hardware-architecture trade-offs in this work are among energy efficiency, area, and flexibility. Latency is not treated as an independent optimization target once the operating frequency and frame-rate requirement are defined; the hardware only needs to satisfy the required timing constraint. Similarly, accuracy is mainly determined by the front-end algorithm and model selection, so it is not the primary hardware trade-off in this design. From the hardware perspective, the key question is whether to use more silicon area and parallel hardware to reduce execution time, or to use a smaller architecture that meets the frame-rate requirement with lower area and energy. When the timing requirement is already satisfied, the preferred choice is to reduce unnecessary hardware resources and improve energy efficiency.

Several concrete architectural decisions reflect this trade-off. Increasing the capacity of on-chip registers can reduce costly SRAM accesses, but it also increases chip area and static power. A carefully designed data path can improve data reuse and reduce redundant data movement, which directly improves energy efficiency. Flexibility introduces another important trade-off: a more flexible architecture usually requires redundant control logic or data-path support for other model structures, which increases area and power. However, this flexibility also makes it easier to iterate the accelerator when the front-end algorithm is changed. In this design, the RTL is parameterized so that the accelerator can be quickly reconfigured for different convolutional neural network structures, provided that the target model remains within the CNN-based KWS design scope.

RQ3: Which parts of the system are most critical to the final efficiency of the KWS accelerator? The most critical part of the system is first the front-end algorithm, because it directly determines the amount of computation that the hardware must execute. For example, if the size and parameter count of a neural network are reduced to one-half of the original scale, the total computation can be reduced to roughly one-quarter in many convolutional layers. This reduction is closely related to the final power consumption of the accelerator. Therefore, algorithm-level model compression and compact network design are fundamental to the overall efficiency of the KWS system.

The second critical part is the SRAM organization. A deeper SRAM requires a longer address, and the address decoding and access energy increase with the memory scale. As a result, reducing SRAM storage demand is highly effective for lowering power consumption. If part of the storage space can be reused across different computation stages, both memory area and access energy can be reduced. In addition, the design should identify common computations in the processing flow and implement them as reusable operators, so that multiple computation nodes can share the same hardware resources instead of duplicating similar logic. Finally, a pipelined architecture is preferred because it can maintain continuous operation while satisfying the critical-path constraint. This avoids throughput degradation caused by long combinational paths and improves the utilization of the accelerator during sequential KWS processing.

6.3 Advantages

The main advantage of this work is that it treats KWS as a system-level hardware problem rather than only an algorithm-selection problem. By jointly considering feature extraction, neural-network structure, datapath design, and memory hierarchy, the final architecture is more suitable for always-on edge deployment than a software-only solution.

Another advantage is the emphasis on regularity and reuse. MFCC provides a compact input representation, DSCNN offers a structured computation pattern, and the memory subsystem is optimized to reduce unnecessary read/write activity. This makes the design easier to map onto ASIC resources and improves the balance between power, area, and latency.

A further strength is that the design preserves practical performance. The system still satisfies the target latency and keeps competitive accuracy, which shows that low-power optimization does not necessarily require a large sacrifice in recognition quality. The measured improvements in SRAM power and area also demonstrate that memory organization is one of the most important optimization levers in this type of accelerator.

6.4 Limitations

The proposed design still has several limitations. First, unlike an NPU, this project is still an ASIC chip. Although the RTL-level implementation allows the network

parameters and the internal organization to be adjusted with some flexibility, once the chip is fabricated, only the model parameters can be changed in practice. This means that the design is less flexible than a reconfigurable NPU platform, and any structural change to the network would require a new chip implementation.

Second, although the average power consumption is already low, the peak power is still above 1 mW. This is mainly caused by the post-processing part of the network. In this design, the SRAM scale and the model size have an approximately quadratic relationship under the current parameter setting, so the model can still create a relatively large instantaneous power demand during peak operation. For a formal product implementation, the algorithm side must further reduce the model size in order to lower the peak power.

Third, the reported power and area improvements are mainly derived from architectural and SRAM-level analysis. A complete post-layout implementation could introduce additional routing, control, and integration overhead, so the final chip-level numbers may differ from the pre-layout estimates.

6.5 Future Work

Future work can extend this design in several directions. First, the relationship between model size and power consumption should be studied more systematically. Based on the current parameter set, the next step is to estimate how changes in model size affect power, and then perform a more explicit trade-off analysis to identify an optimal algorithm-hardware design that satisfies all target constraints.

Second, the current project is still a small-scale verification and power-estimation version. For a practical product implementation, the design needs to be integrated with an actual IIC bus interface, and additional registers should be added so that the operational state of the KWS module can be explicitly monitored and controlled.

Third, the current power analysis assumes an output rate of one frame per second. In a real deployment, this assumption may not hold, so the operating condition and corresponding power model may need to be further adjusted. A more realistic workload model will help make the final power estimate closer to actual product behavior.

Acknowledgements

From January to June, we completed our master thesis project, and throughout this journey we received support and kindness from many people.

We would first like to sincerely thank the mentors at SI company for their patient guidance and valuable technical support during the project. Their advice and experience greatly helped us throughout the development process.

We are also deeply grateful to our colleagues in the "Juxin Lund team" for their friendship and companionship. Beyond the intense work on our thesis, we shared meals, spent time together, and created many enjoyable memories that brought warmth and color to our daily lives.

Special thanks also go to our supervisor Jonas Skeppstedt, for his careful feedback and thoughtful suggestions on our thesis during the final stage of the project.

Finally, we would like to thank Lund University for giving us two memorable years of master's study. Although the winters often made us feel constantly sleepy and the bright summers sometimes made it difficult to sleep, and while some courses were excellent whereas others were less useful, it has nevertheless been a welcoming and friendly place for international students. We truly enjoyed a peaceful and meaningful period of student life here, and these experiences will remain valuable memories for us in the future.

Bibliography

[1] Sercan O. Arik, Anjuli Kannan, Yoshua Bengio, A. Haque, Paulius Micikevicius, H. Pham, D. Song, and Pieter Abbeel. Convolutional recurrent neural networks for small-footprint keyword spotting. In *Proceedings of Interspeech*, 2017.

[2] Pallavi Baljekar, Jill Fain Lehman, and Rita Singh. Online word-spotting in continuous speech with recurrent neural networks. In *2014 IEEE Spoken Language Technology Workshop (SLT)*, pages 536–541, 2014.

[3] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 1 2017.

[4] I-Yun Chung, Seungwoo Choi, Seokjun Seo, Hyunwoo Moon, Jongwon Jung, Byungjin Lee, Sunghyun Park, and Yoonsoo Kim. Temporal convolution for real-time keyword spotting on mobile devices. In *Proceedings of Interspeech*, pages 3372–3376, 2019. arXiv:1904.03814.

[5] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[6] S. Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken se. 1980.

[7] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[8] Wei Han, Cheong-Fat Chan, Chiu-Sing Choy, and Kong-Pang Pun. An efficient mfcc extraction method in speech recognition. In *2006 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4–pp. IEEE, 2006.

[9] Puli Anil Kumar. Fpga implementation of the trigonometric functions using the cordic algorithm. In *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*, pages 894–900, 2019.

[10] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 6 1991.

[11] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[12] Vinh Vu Ngoc, James Whittington, and John Devlin. Real-time hardware feature extraction with embedded signal enhancement for automatic speech recognition. *Speech Technologies, Intech*, pages 29–54, 2011.

[13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, CA, USA, 5 edition, 2014.

[14] Oleg Rybakov, Igor Kononenko, Naveen Subrahmanya, Mark Visontai, Seokjun Seo, Seungwoo Choi, Oleksii Sidorov, Rogier Dalen, Peter Chua, Byungjin Lee, and Arnab Ghoshal. Streaming keyword spotting on mobile devices. In *Proceedings of Interspeech*, pages 2277–2281, 2020.

[15] Tara N. Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Proceedings of Interspeech*, pages 1478–1482, 2015.

[16] Ruiming Tang and Jimmy Lin. Deep residual learning for small-footprint keyword spotting. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5484–5488, Calgary, AB, Canada, 2018.

[17] Jack E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959.

[18] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[19] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

Appendix A
Appendix

```
Project structure
├── rtl
│   ├── ap.sv
│   ├── cal_fsm.sv
│   ├── cal_mode.sv
│   ├── dscnn_control.sv
│   ├── dscnn_layer_engine.sv
│   ├── dscnn_top.sv
│   ├── kws_top.sv
│   ├── kws_top.v
│   ├── mfcc_top.sv
│   ├── mfcc_window_ref_pkg.sv
│   ├── multiplier_arbiter.sv
│   ├── multiplier_wrapper.sv
│   ├── multiplier.sv
│   ├── sram_arbiter.sv
│   ├── sram_control_data.sv
│   ├── sram_control_lut.sv
│   ├── sram_wrapper.sv
│   ├── TMHDPS040EBA_V1L0B1R0_256X32M8W1F1.v
│   ├── TMHDPS040EBA_V1L0B1R0_992X16M8W1F1.v
│   ├── TMHDPS040EBA_V1L0B2R0_7296X8M16W1F1.v
│   └── TMHDPS040EBA_V1L0B4R0_23040X8M32W1F1.v
├── simulation
│   ├── regression
│   ├── results
│   ├── run
│   ├── back_end_wave.vcd
│   └── front_end_wave.vcd
├── testbench
│   ├── env
│   └── tb
└── testcase
    ├── dscnn_test
    ├── kws_multi_pic_test
    ├── kws_test
    └── tc_base.sv
```

Figure A.1: Project directory architecture

B. Additional tables and figures

Average Pooling waveform

Figure A.2: Average pooling waveform.

Cal FSM waveform

Figure A.3: Cal FSM waveform.

Cal Mode waveform

Figure A.4: Cal mode waveform.

DSCNN control waveform

Figure A.5: DSCNN control waveform.

DSCNN layer engine waveform

Figure A.6: DSCNN layer engine waveform.

MFCC top waveform

Figure A.7: MFCC top waveform.

SRAM wrapper waveform

Figure A.8: SRAM wrapper waveform.

Multiplier wrapper waveform

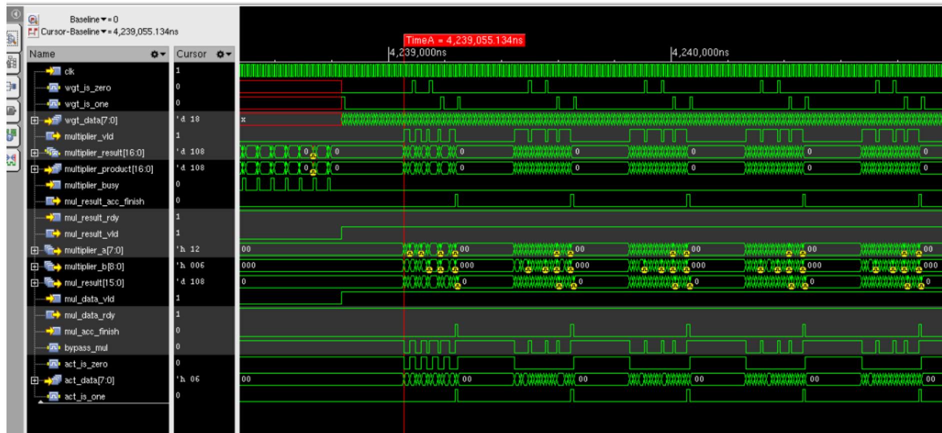


Figure A.9: Multiplier wrapper waveform.