electronics

MDPI

Article

Hardware–Software Co-Design of an Audio Feature Extraction Pipeline for Machine Learning Applications

Jure Vreča [1,2,*], Ratko Pilipović [3] and Anton Biasizzo [1]

[1] Jožef Stefan Institute, 1000 Ljubljana, Slovenia; anton.biasizzo@ijs.si
[2] Jožef Stefan International Postgraduate School (IPS), 1000 Ljubljana, Slovenia
[3] Faculty of Computer and Information Science, University of Ljubljana, 1000 Ljubljana, Slovenia; ratko.pilipovic@fri.uni-lj.si
* Correspondence: jure.vreca@ijs.si

Abstract: Keyword spotting is an important part of modern speech recognition pipelines. Typical contemporary keyword-spotting systems are based on Mel-Frequency Cepstral Coefficient (MFCC) audio features, which are relatively complex to compute. Considering the always-on nature of many keyword-spotting systems, it is prudent to optimize this part of the detection pipeline. We explore the simplifications of the MFCC audio features and derive a simplified version that can be more easily used in embedded applications. Additionally, we implement a hardware generator that generates an appropriate hardware pipeline for the simplified audio feature extraction. Using Chisel4ml framework, we integrate hardware generators into Python-based Keras framework, which facilitates the training process of the machine learning models using our simplified audio features.

Keywords: FPGA; MFCC; keyword spotting; chisel

Citation: Vreča, J.; Pilipović, R.; Biasizzo, A. Hardware–Software Co-Design of an Audio Feature Extraction Pipeline for Machine Learning Applications. *Electronics* **2024**, *13*, 875. https://doi.org/10.3390/electronics13050875

Academic Editor: Chunping Li

Received: 31 January 2024
Revised: 17 February 2024
Accepted: 22 February 2024
Published: 24 February 2024

1. Introduction

The development of deep neural networks has opened up possibilities for applications in diverse areas. One such area is speech recognition, where researchers have been able to show promising results using large transformer-based neural networks [1]. These networks are, however, very computationally expensive and thus are hard to implement on battery-powered devices. This can be overcome using a simpler keyword detection system that merely listens for specific keywords (e.g., Hey Siri) and then wakes up a more powerful system generally implemented in the cloud. In this way, the simpler low-power system acts as an always-on listener, and the more powerful system is used only when needed. Henceforth, we will refer to the first kind of system as the keyword-spotting (KWS) system [2], and the second type as the Large Vocabulary Speech Recognition system.

The keyword-spotting system is an always-on system. This requirement facilitates the need for it to be as energy-efficient as possible. The authors of [3] recognized this problem and explored an integer-only implementation of the MFCC algorithm. They achieved good results. However, they refrained from modifying the MFCC algorithm. Furthermore, they targeted a DSP processing unit. Dedicated hardware circuits could make this process even more energy-efficient. The authors of [4] developed a custom MFCC extraction processing unit. However, they also failed to explore simplifications of the MFCC algorithm and instead focused exclusively on the hardware implementation. The authors of [5] showcased a system that simplifies the MFCC features to a certain degree. While they did obtain satisfactory classification performance, they again focused on the actual hardware implementation and did not show how the MFCC simplification affects the final accuracy of the KWS system. Yet another stream of research explores the optimization of the MFCC with analog circuits. In [6], the authors explored the moving parts of the MFCC pipeline into the analog domain. However, the computations performed in the analog

domain are prone to environmental influence. Our study focuses on digital processing using custom hardware accelerators.

The main contributions of this paper are as follows: First, we formulate a simplified version of the MFCC features. Second, we explore how simplifications of the MFCC algorithm affect the performance achieved by KWS systems. Third, we develop hardware generators and evaluate generated hardware circuits that can compute these features in a resource-efficient manner. And fourth, we connect our hardware generators with the Python-based machine learning framework Keras. This software-based implementation allows for a user-friendly way of training machine learning models that can exploit the aforementioned simplified features.

The rest of this paper is structured as follows. Section 2 explains how the standard MFCC features are calculated, as well as which simplifications we studied to make them more amenable to an efficient hardware implementation. Section 3 details the modules that compute the simplified MFCC features. In Section 4, we discuss Chisel4ml [7]. It is a Python/Chisel-based framework, which we developed, that is the basis of how we connect our hardware generators with the Python ecosystem. In Section 5, we study how various simplifications of the MFCC algorithm affect the accuracy of a KWS system. In Section 6, we give the synthesis results of the hardware circuits generated by our generators. And finally, in Section 7, we conclude and give some indications of future work.

2. Audio Preprocessing

A common way to detect keywords from audio is to use convolutional neural networks on two-dimensional features representing audio. One of the frequently used features is Mel-Frequency Cepstrum Coefficients (MFCCs) [8]. Figure 1 shows how raw audio is converted to obtain MFCC features.

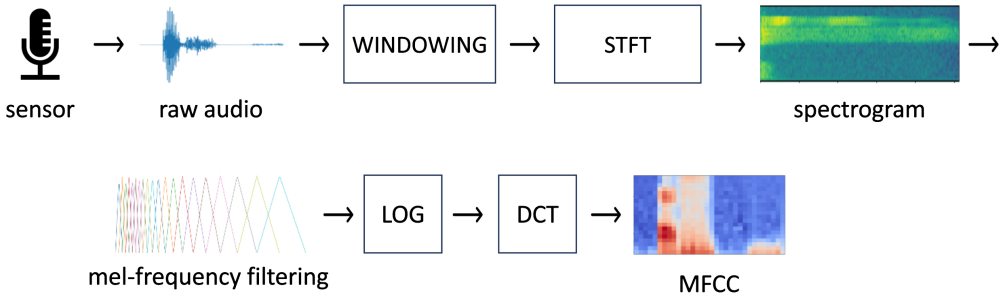


Figure 1. From raw audio to MFCC features.

2.1. Classic MFCC Features

The procedure to compute MFCC features for approximately one second of audio sampled at 16 kHz is as follows:

1. We start by sampling the audio with a microphone and an Analog-to-Digital converter. We take 16,384 samples because it simplifies framing in the next step.

$$X[n], \quad n \in [0, 16{,}383] \tag{1}$$

2. Next, we separate the audio into frames and use a windowing function to reduce the effect of spectral leakage. In the illustrated example, we employ a frame size of 512, thus obtaining 32 frames ($32 \times 512 = 16{,}384$). A commonly used windowing function is the Hamming window shown in Figure 2:

$$X[n] \rightarrow X'[w, n'] \cdot win[n'] = X''[w, n'], \quad w \in [0, 31], \quad n' \in [0, 511] \tag{2}$$

3. We perform the Short-Time Fourier Transform (STFT) [9], which is essentially a Discrete Fourier Transform (DFT) applied to each frame. Due to the symmetry of the

DFT for real signals, it suffices to consider only the first half and one of the frequency bins of the DFT (512/2 + 1 = 257):

$$X''[w, n'] \rightarrow STFT(X'') \rightarrow Y[w, k], \quad w \in [0, 31], \quad k \in [0, 256] \tag{3}$$

4. We filter the power spectrum by a mel-frequency filter bank matrix. The mel-frequency filter bank is a set of triangle filters where the triangle width increases exponentially to mimic the non-linear human ear perception of sound. The number of filters in the bank is a settable parameter. In our example, it is set to 6. Figure 3 shows an example mel-frequency filter bank with 6 filters:

$$Y[w, k] \rightarrow |Y[w, k]| \cdot M = Y', \quad shape(M) = (257, 6), \quad shape(Y') = (32, 6) \tag{4}$$

5. We apply the logarithmic function to each element of matrix Y'.
6. Finally, we compute the Discrete Cosine Transform (DCT) on Y' to obtain the MFCC features.

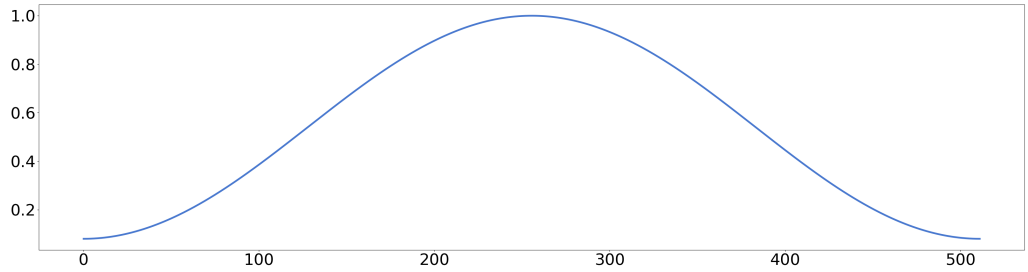


Figure 2. Hamming window.

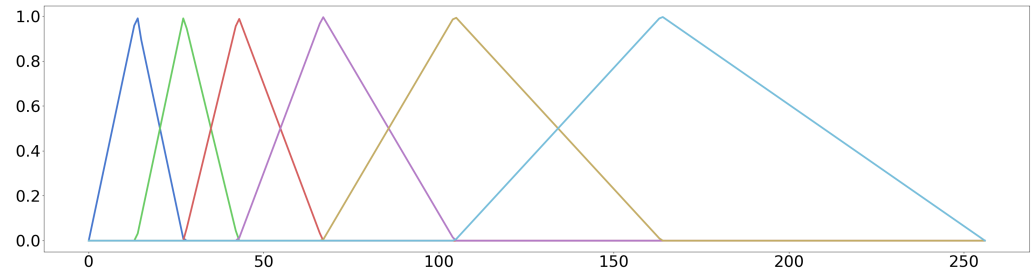


Figure 3. Mel-frequency filter bank with 6 bins. Each color represents a bin.

2.2. Simplified MFCC

The MFCC feature calculation involves some mathematical operations that, in order to implement them in hardware exactly, require substantial hardware resources and larger processing time. Thus, we make the following simplifications:

1. The arithmetic in floating-point number representation requires complex circuits to implement the multiplication and addition operations. Therefore, we use fixed-point arithmetic, which is simpler to implement and is more energy-efficient.
2. Instead of computing the full power spectrum, we use just the real part of the DFT result. We do this because the real part holds most of the information.
3. Instead of the natural logarithm, we use an approximation of the logarithm of number two. Essentially, we take an integer part of the result and determine the position of the leading one bit.
4. We noticed that due to the logarithm approximation, low-amplitude sound intervals were indistinguishable from completely silent intervals. To mitigate this, we added the value 1.0 to each DFT frequency bin, which increased the frequency bin amplitude while preserving their variations.
5. We skip the DCT calculation entirely.

The simplifications above do affect the final accuracy of the KWS system. However, we can still achieve good performance, as we will show in Section 5. We call these simplified features the Log-Mel Filter Bank Real Energy (LMFE) representation.

3. Hardware

We break the steps described in Section 2 into two separate hardware modules. The first module SDF-FFT performs the windowing and the STFT, while the second module Mel-Engine performs the mel-frequency filtering and logarithm calculation. The modules are connected by an AXI4-Stream interface as shown in Figure 4. A real system would additionally need a pre-emphasis filter after the Analog-to-Digital conversion of the sound.

The hardware implementation may vary, depending on the LMFE parameters (frame size, number of frames, number of mel-frequency bins, windowing function). Therefore, we decided to write a hardware generator that is parameterized by the aforementioned parameters. We used the Chisel Hardware Construction Language (HCL) [10] for the implementation. Chisel is a Scala framework for writing reusable generators of synchronous digital circuits. Chisel is a library of special class definitions that correspond to various hardware constructs: registers, multiplexers, adders, etc. When using Chisel, you are writing a Scala program that, when executed, constructs a set of Chisel objects and connects them. The interconnected Chisel objects represent a hardware graph, which can be exported as Verilog.

Figure 4. The preprocessing pipeline.

3.1. SDF-FFT

To calculate the Fast Fourier Transform (FFT), we use a publicly available SDF-FFT Chisel generator [11]. It uses the Single-path Delay Feedback topology to calculate the FFT in a pipelined fashion. SDF-FFT is parameterized by the frame size and the radix of the butterfly (2, 4, or 2^2). We use the radix-2 butterfly and set the frame size according to the parameters of the LMFE features. Since the SDF-FFT module does not properly support windowing, we extend it by implementing the required windowing. The original module is available at the link: https://github.com/milovanovic/sdf-fft (accessed on 30 January 2024).

3.2. Mel-Engine

Mel-Engine is the hardware module that performs the process of extracting LMFE features from the FFT results.

Figure 5 shows the hardware architecture of the Mel-Engine. We obtain the FFT results on an AXI4-Stream input bus, take the real part, and increment it by one to improve low-volume audio signal sensitivity, as described in Section 2.2. Next, we square the intermediate result to obtain the power approximation of the given frequency bin. To efficiently filter the approximate power spectrum with mel-frequency filter banks, we design two parallel multiply-and-accumulate units that simultaneously process two sets of mel-frequency filter parameters. Finally, the accumulated results are passed through the base-2 logarithm approximation unit. This unit outputs the position of the leading one bit of the integer part of the accumulated result.

The key observation that inspired the shown architecture is the property that, at most, two mel-frequency filters are non-zero for each frequency bin. This property can be seen in Figure 3. We store the mel-frequency filter bank parameters in the Mel-Filter ROM (MFROM). The MFROM length is equal to the number of frequency bins. We split each MFROM word into the left and right halfwords, and each halfword holds a parameter stored as a 16-bit fixed-point number. The left halfwords hold all the odd filter parameters, and the right halfwords hold all the even filter parameters.

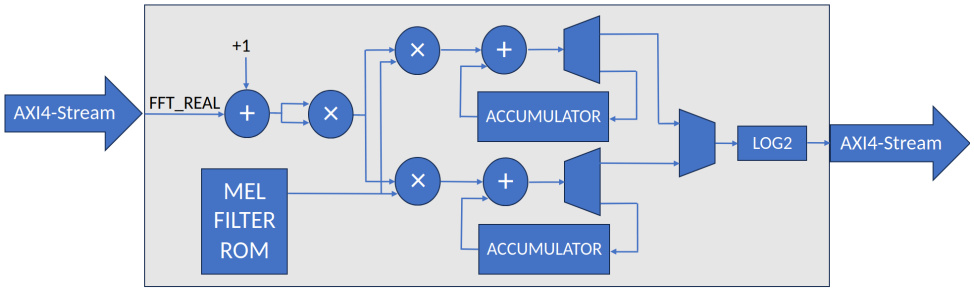


Figure 5. Hardware architecture of the Mel-Engine.

Figure 6 shows how we store the mel-frequency filters in MFROM. The colored triangles on the left represent the mel-frequency filters, and the MFROM is depicted on the right. MFROM addresses correspond to FFT frequency bins. The colored memory cells store the mel-frequency filter parameters of the corresponding triangle filter. An additional data structure is required to distinguish the filter parameters of individual mel-frequency filters. This data structure is implemented by an additional small look-up table. Its size is the number of mel-frequency filters, and it stores the last non-zero element of the corresponding mel-frequency filter. This look-up table is a part of the control logic, which is not depicted in Figure 5, to reduce figure complexity. The source code for the Mel-Engine accelerator is available at the following link: https://github.com/jurevreca12/mel-engine (accessed on 30 January 2024).

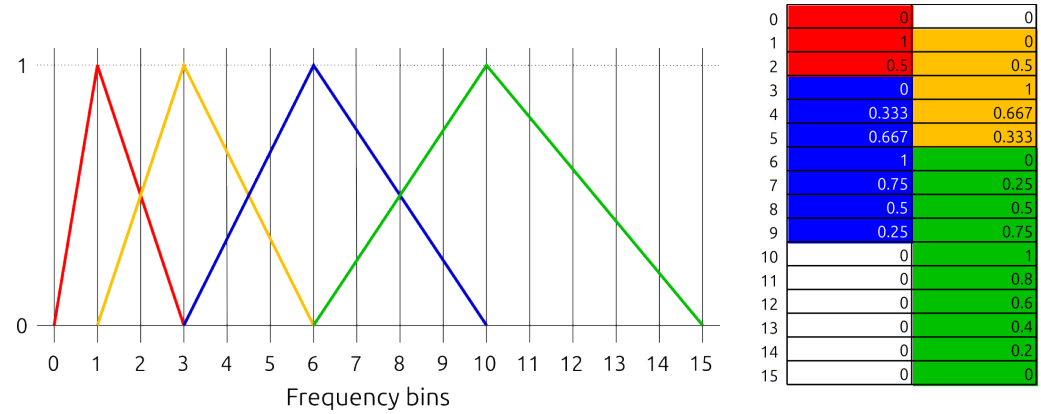
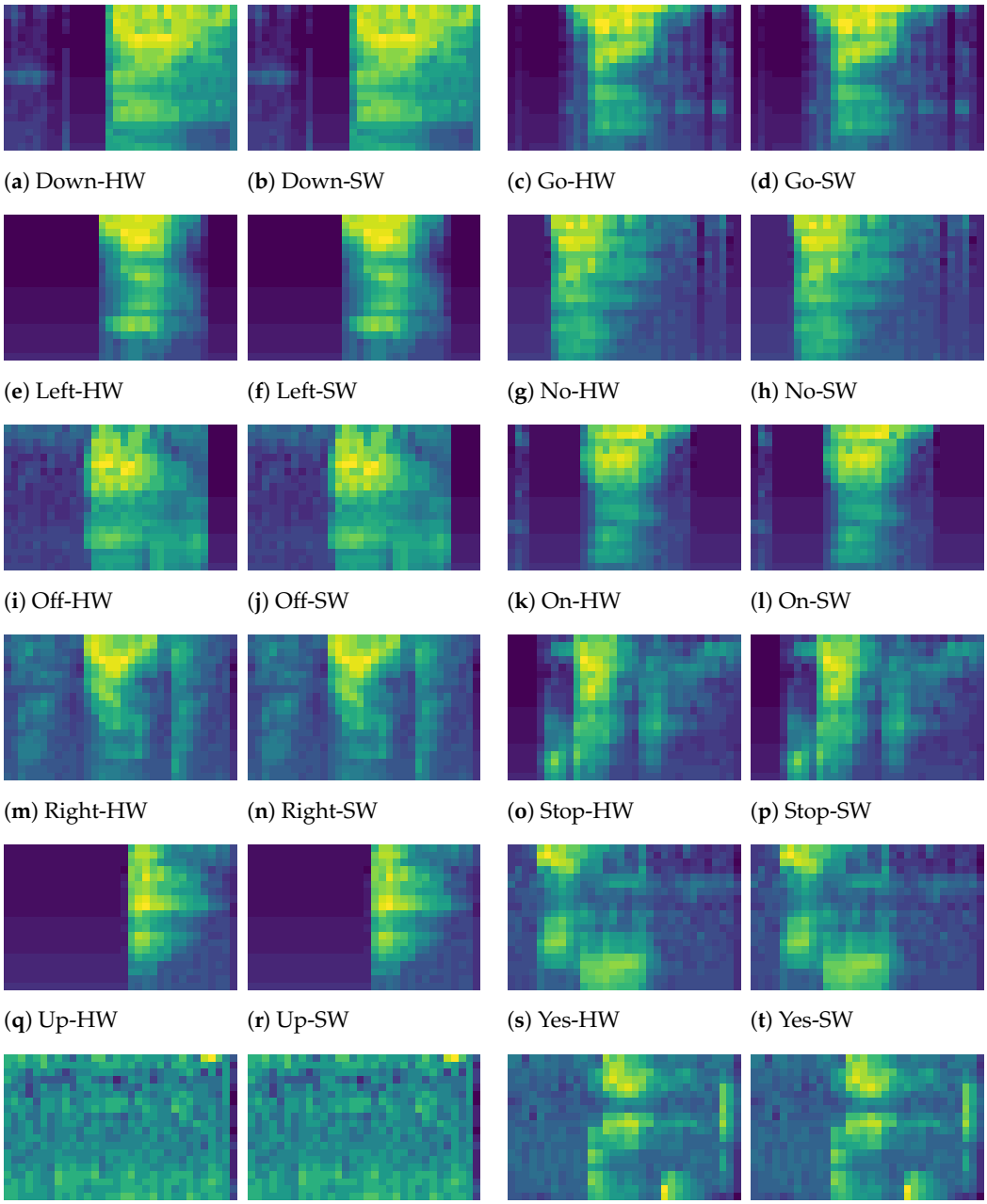


Figure 6. Mel-filter ROM in the Mel-Engine accelerator.

4. Python Integration-Chisel4ml

Following the hardware design, we provide support for the proposed LMFE features in the Keras framework by creating the FFTLayer and LMFELayer Python classes. The created classes mimic the calculations of the hardware modules SDF-FFT and Mel-Engine, which enable efficient training of neural networks that use the proposed LMFE features. The main reason that FFTLayer and LMFELayer classes are not bit-accurate is that they use floating-point arithmetic, while the hardware implementations use fixed-point arithmetic. However, we found that the differences between these two implementations are negligible as is clearly illustrated in Figure 7, which shows a comparison of 12 spoken words from the Google Speech Commands dataset [12]. The spoken words are down, go, left, no, off, on, right, stop, up, and yes. There are also two other classes, "silence" and "unknown". The subfigures with HW in the subcaption show the LMFE features generated via RTL simulation of the hardware, and subfigures with SW in the subcaption are obtained by using the FFTLayer and LMFELayer classes. We chose to implement FFTLayer and LMFELayer with floating-point arithmetic because of the existence of high-performance floating-point libraries and because modern high-performance hardware is optimized for floating-point operations.



(**a**) Down-HW (**b**) Down-SW (**c**) Go-HW (**d**) Go-SW

(**e**) Left-HW (**f**) Left-SW (**g**) No-HW (**h**) No-SW

(**i**) Off-HW (**j**) Off-SW (**k**) On-HW (**l**) On-SW

(**m**) Right-HW (**n**) Right-SW (**o**) Stop-HW (**p**) Stop-SW

(**q**) Up-HW (**r**) Up-SW (**s**) Yes-HW (**t**) Yes-SW

(**u**) Silence-HW (**v**) Silence-SW (**w**) Unknown-HW (**x**) Unknown-SW

Figure 7. LMFE features obtained from hardware and software.

We provide a direct path from the FFTLayer and LMFELayer class definition to generated hardware via our Python/Chisel framework Chisel4ml [7]. The software architecture of Chisel4ml is shown in Figure 8. The Python frontend of Chisel4ml directly maps the configuration of FFTLayer and LMFELayer to SDF-FFT and Mel-Engine generator configuration. It then sends the configuration to the Chisel backend, which generates the hardware. The backend then returns the circuit ID number to the Python frontend, which can be used to drive the RTL simulation of the generated circuit on the backend. This functionality allowed us to evaluate the generated circuits directly from Python. In this work we have used Chisel4ml 0.2.1 running on Python 3.10. The source code of the Chisel4ml framework is available at the following link: https://github.com/cs-jsi/chisel4ml (accessed on 30 January 2024).

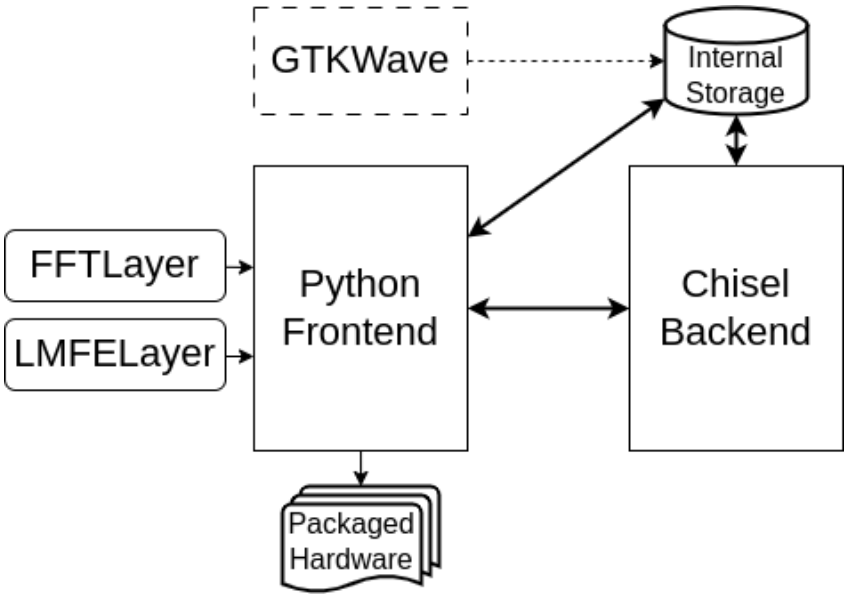


Figure 8. Software architecture of Chisel4ml.

Listing A1 in Appendix A shows an example of a Python script that uses the FFTLayer and LMFELayer classes to generate the hardware and compare the software and hardware results.

5. Case Study: Keyword Spotting on Google Speech Commands

We investigate the impact of MFCC feature configurations on the accuracy of keyword spotting using a Depthwise Convolutional Neural Network (CNN) model. Our experimental design involves comparing the classification accuracy of the baseline, which utilized the traditional MFCC features, against three modified feature sets. Firstly, we examine the performance without the commonly applied Discrete Cosine Transform (DCT) step in MFCC computation. The omission of DCT in calculating MFCC is a common approach to reduce the complexity of the feature extraction stage. Subsequently, we explore the effect of replacing the natural logarithm with base-2 logarithm approximation, which outputs only the integer part of the logarithm. Finally, we evaluate the performance of the proposed LMFE features, which, besides the mentioned modifications, take only the real part of the frequency spectrum into account.

5.1. Depthwise Separable Convolutional Neural Network

We employ a Depthwise Separable Convolutional Neural Network (DS-CNN) [13], a widely recognized and efficient CNN architecture for various tasks, e.g., computer vision [14], audio recognition [2] and many others. The basic building block represents depthwise separable convolution, where each channel within the input feature map undergoes convolution with a dedicated 2-D filter. Then, pointwise convolutions (1×1) are employed to fuse the resulting outputs along the depth dimension. The DS-CNN decomposes 3-D convolutions into sequential 2-D and 1-D convolutions, which reduces the number of parameters and operations and enables the construction of deeper and broader architectures. Significantly, these characteristics make DS-CNN well suited for deployment in microcontroller devices with limited computational resources. The utilized network is depicted in Figure 9. It starts with two convolution layers with filter dimensions 10×4 and 1×1, respectively. The network core represents four depthwise separable convolution layers, comprising 3×3 depthwise convolution and a pointwise (1×1) convolution. The final layer of the network consists of two fully connected layers designed to determine the class of input samples. The first fully connected layer has 512 neurons, while the second has 12 neurons. The convolution and depthwise convolution layers employ the Rectified Linear Unit (ReLU) activation function, contributing to non-linearity in the network's learning

process. In contrast, the activation function for the last fully connected layer is softmax, which outputs probability scores for each class label.

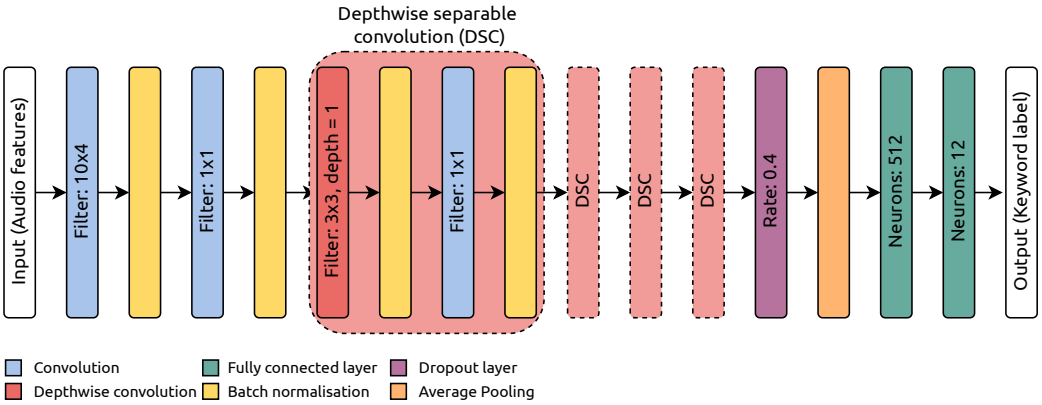


Figure 9. Depthwise separable convolutional network.

5.2. Experimental Settings

For assessing examined MFCC features, we rely on the Google Speech Commands dataset [12] to conduct experiments. This dataset comprises 65,000 1-second-long audio clips featuring 30 different keywords. These audio clips were contributed by thousands of individuals, with each clip containing a singular keyword. The primary task for the employed network models is to classify incoming audio into one of ten specified keywords: "Yes", "No", "Up", "Down", "Left", "Right," "On", "Off", "Stop", and "Go". Additionally, the dataset includes categories for "silence" (indicating no spoken word) and "unknown", encompassing the remaining 20 keywords from the dataset. We partition the dataset into training, validation, and test sets in an 80:10:10 ratio. The DS-CNN model is trained using the Keras framework, employing sparse cross-entropy loss and the Adam optimizer. Training is conducted with a batch size of 128 over 20 epochs, utilizing a learning rate of 5×10^{-3}.

5.3. Keyword-Spotting Results

In this section, we justify the choice of the proposed simplifications of the MFCC algorithm, which we described in Section 2.2. We differentiate the following audio feature representations:

1. *MFCC*—The full floating-point MFCC as described in Section 2.1.
2. *W/O DCT*—Same as *MFCC* but without the DCT at the end.
3. *LOG2APPROX*—Same as *W/O DCT* but using the logarithm base-2 approximation instead of the natural logarithm.
4. *LMFE*—The LMFE features as described in Section 2.2.

The results depicted in Figure 10 offer insight into keyword recognition performance across different audio feature representations and varying numbers of mel-frequency filters. The baseline *MFCC* features do not necessarily lead to the best classification accuracy. The *W/O DCT* and *LOG2APPROX* features deliver similar or even better classification accuracy. The *LMFE* features show a slight decrease in classification performance. However, the difference is in the order of 1% on average. We attribute this to the information lost by discarding the imaginary part of the spectrogram. Although *LMFE* features have the worst classification performance of the four shown versions, they still perform satisfactorily. We consider this to be a worthwhile tradeoff, as *LMFE* features are significantly less computationally expensive. The source code of this experiment is available at the following link: https://github.com/RatkoFri/kws_preprocess_test (accessed on 30 January 2024).

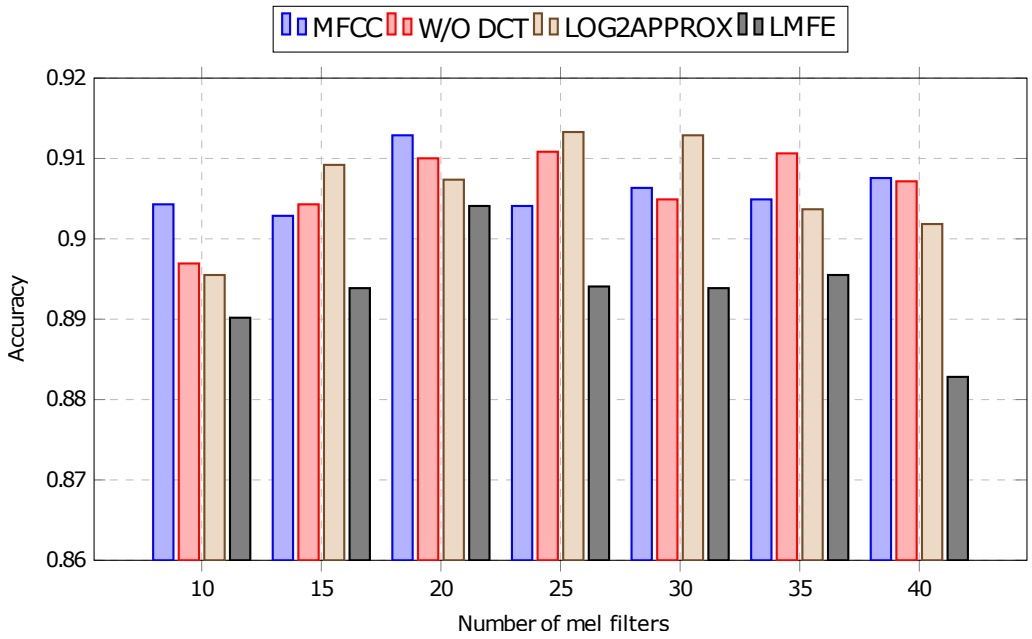


Figure 10. Simplifications effect on final accuracy.

6. Hardware Synthesis Results

In this section, we present the synthesis results on a range of parameter settings for the hardware generators. We vary the following parameters of the LMFE features:

- *frame_size*—The size of the frame in STFT (128, 256, 512, and 1024).
- *num_frames*—Number windows that together make a frame (8, 16, 32, and 64).
- *num_mels*—Number of mel filters (10, 13, 15, 20).

For each of the parameters listed above, we have four different settings. Taking all combinations of them, we generate and characterize a total of 64 different circuits. Table A1 in Appendix B gives the synthesis results for the combined SDF-FFT and Mel-Engine preprocessing modules in different parameterizations. We synthesize the circuits for the Xilinx XC7Z010-1CLG225C FPGA device using Xilinx Vivado 2022.2. The *Parameterization* column gives the trio of parameter values: $frame\_size$, num_frames, num_mels, in that order. The LUT, FF, and DSP columns give the number of look-up tables, flip-flops, and DSP blocks, respectively, used by the combined preprocessing modules. The $CYCLES$ column shows the cycles needed to compute one set of LMFE features. The T column gives the maximum throughput in millions of audio samples per second. The throughput results significantly surpass actual sound processing requirements; thus, the hardware clock can be significantly reduced in practical implementation. Finally, the DP column gives the dynamic power in watts. As the circuits have a similar structure, they all achieved an equal maximum clock of 15.1 ns or the maximum frequency of 66.3 MHz. All circuits use fewer than 4 Block RAM elements.

Figure 11 shows the number of look-up tables and flip-flops consumed by designs with 64 frames (num_frames) and 20 mel-frequency filters (num_mels). The FFT module is the larger component of the generated designs. Its size is only dependent on the $frame\_size$ parameter (from the parameters we varied). In other words, the only changing parameter is the frame size.

Looking at the data from the perspective of the Mel-Engine, its low resource consumption and high throughput make it an efficient addition to the audio preprocessing pipeline, as it significantly lowers the dimensionality of data, and thus also the complexity of the processing needed to be performed by the KWS system.

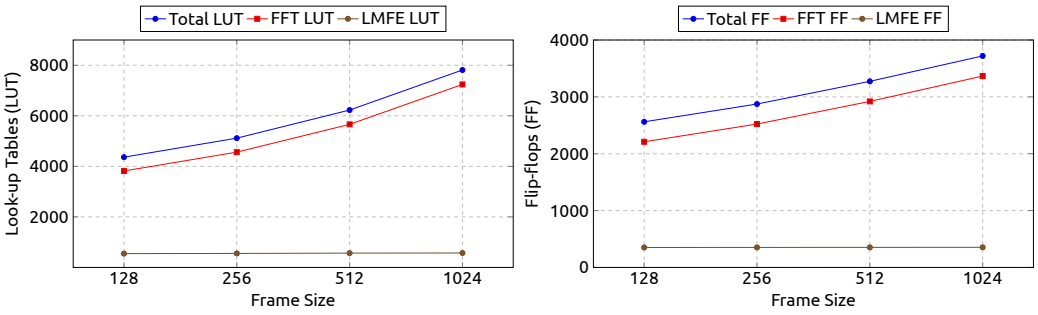


Figure 11. Resource consumption by frame size.

Figure 12 shows our circuits' dynamic, static, and total power consumption by frame size. The static power also remains constant among the designs at 0.093 W. Although the static power consumption remains unchanged, the dynamic power consumption does increase with frame size. Namely, it follows a similar trajectory to the growth in resource consumption.

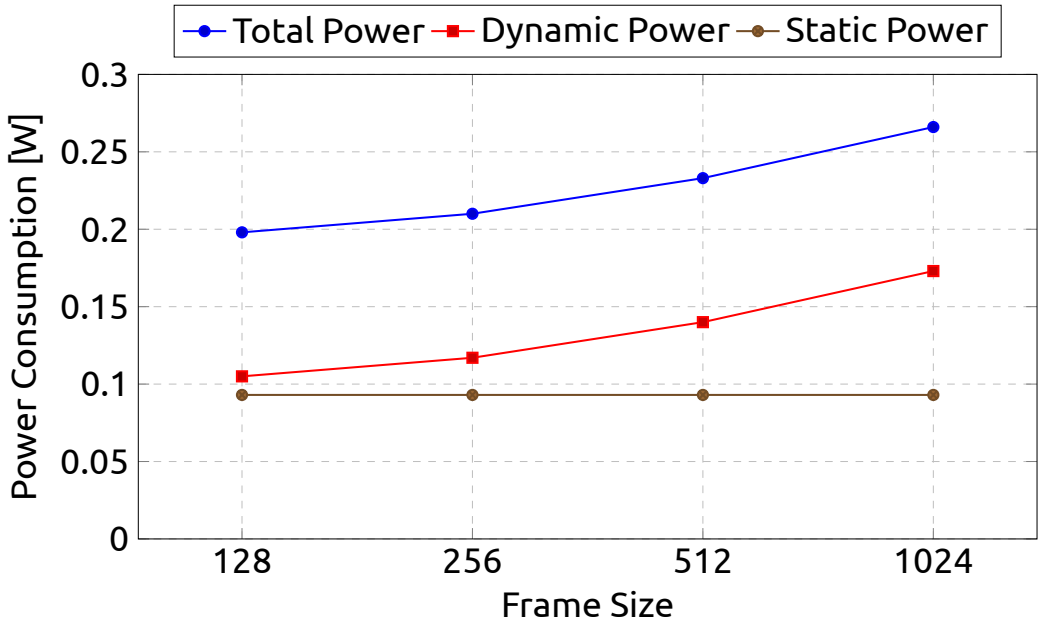


Figure 12. Power consumption by frame size.

Our results are hard to directly compare with previous work because other works target specific parameter designs and do not necessarily simplify the MFCC algorithm. Among the approaches that introduce simplifications in the MFCC algorithm, the authors of [5] obtain similar results for a frame size of 256 (they only provide results for this frame size). Table 1 compares our Mel-Engine (ME) combined with SDF-FFT against the Mel-Processing Unit (MPU) [5]. MPU uses more flip-flops and slightly more look-up tables than our combined SDF-FFT and ME pipeline. However, MPU uses fewer DSP blocks.

Table 1. Comparison of our hardware results with [5].

	MPU-256 [5]	**ME/FFT-256**
LUT	5349	5116
FF	3735	2874
DSP	5	46

7. Conclusions

We explored a hardware–software co-design approach for extracting audio features that can be used for the keyword-spotting task. Slight modifications to the algorithm lead only to a minor decrease in classification accuracy but a substantial decrease in the computing requirements. This is especially relevant for resource-limited embedded systems. Our Mel-Engine processing unit introduces only a minor overhead in terms of FPGA resources but simultaneously reduces the dimensionality of the audio representation. Consequently, the proposed approach relieves the subsequent neural network classifier of unnecessary complexity. By developing custom Python classes, we increase the practical value of the proposed solution. We intend to explore this area further, with possible additional simplifications to both the algorithm and the hardware design. A piece that needs to be added to our pipeline is the actual keyword-spotting system. We intend to upgrade our framework, Chisel4ml, to support neural network topologies such as DS-CNN, enabling the generation of the entire KWS system from Python.

Author Contributions: Methodology, A.B.; Software, J.V. and R.P.; Writing—original draft, J.V.; Writing—review and editing, R.P. and A.B. All authors have read and agreed to the published version of the manuscript.

Funding: The authors acknowledge the financial support from the Slovenian Research and Innovation Agency under grants: No. P2-0098, No. P2-0359, and BI-US/22-24-114. This work is also part of projects that are funded by the ECSEL Joint Undertaking under grant agreements No. 101007273 (DAIS) and No 876038 (InSecTT) .

Data Availability Statement: Publicly available datasets were analyzed in this study. These data can be found here: https://www.tensorflow.org/datasets/catalog/speech_commands (accessed on 30 January 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MFCC	Mel-Frequency Cepstrum Coefficients
LMFE	Log-Mel Filter Bank Real Energy
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
STFT	Short-time Fourier Transform
LUT	Look-up Table
FF	Flip-Flop
DSP	Digital-Signal Processing block
DS-CNN	Depthwise Seperable Convolutional Neural Network
FPGA	Field-Programmable Gate Array
KWS	Keyword Spotting
MFROM	Mel-Filter Read Only Memory
ReLU	Rectified Linear Unit

Appendix A. Using Chisel4ml

Listing A1 shows how to define a Keras model with the FFTLayer and LMFELayer classes.

Listing A1. Using chisel4ml.

```
import tensorflow as tf
import numpy as np
from chisel4ml import generate, FFTConfig, LMFEConfig, FFTLayer, LMFELayer

preproc_model = tf.keras.Sequential()
preproc_model.add(tf.keras.layers.Input(num_frames, frame_length))
preproc_model.add(
    FFTLayer(
        FFTConfig(
            fft_size=frame_length,
            num_frames=num_frames,
            win_fn=np.hamming(frame_length),
        )
    )
)
preproc_model.add(
    LMFELayer(
        LMFEConfig(
            fft_size=frame_length,
            num_frames=num_frames,
            num_mels=num_mels,
        )
    )
)
preproc_circuit = generate.circuit(preproc_model)

sw_res = preproc_model(audio_sample)
hw_res = preproc_circuit(audio_sample)
assert np.allclose(
    sw_res.numpy().flatten(),
    hw_res.flatten(),
    atol=1,
    rtol=0.05
)
```

Appendix B. Synthesis Results

Table A1. Synthesis results.

Parameterization	LUT	FF	DSP	CYCLES	T [Msamples/s]	DP [W]
(128, 8, 10)	4353	2557	40	225	301.72	0.105
(128, 8, 13)	4362	2557	40	225	301.72	0.105
(128, 8, 15)	4351	2557	40	225	301.72	0.106
(128, 8, 20)	4353	2558	40	225	301.72	0.106
(128, 16, 10)	4360	2558	40	225	603.44	0.105
(128, 16, 13)	4358	2558	40	225	603.44	0.106
(128, 16, 15)	4355	2558	40	225	603.44	0.105
(128, 16, 20)	4362	2559	40	225	603.44	0.106
(128, 32, 10)	4349	2559	40	225	1206.87	0.105
(128, 32, 13)	4355	2559	40	225	1206.87	0.105
(128, 32, 15)	4354	2559	40	225	1206.87	0.105
(128, 32, 20)	4361	2560	40	225	1206.87	0.105
(128, 64, 10)	4356	2560	40	225	2413.74	0.106
(128, 64, 13)	4355	2560	40	225	2413.74	0.106
(128, 64, 15)	4357	2560	40	225	2413.74	0.106
(128, 64, 20)	4363	2561	40	225	2413.74	0.105

Table A1. *Cont.*

Parameterization	LUT	FF	DSP	CYCLES	T [Msamples/s]	DP [W]
(256, 8, 10)	5110	2870	46	421	322.50	0.117
(256, 8, 13)	5112	2870	46	421	322.50	0.117
(256, 8, 15)	5114	2870	46	421	322.50	0.117
(256, 8, 20)	5115	2871	46	421	322.50	0.117
(256, 16, 10)	5114	2871	46	421	645.00	0.117
(256, 16, 13)	5114	2871	46	421	645.00	0.117
(256, 16, 15)	5113	2871	46	421	645.00	0.117
(256, 16, 20)	5114	2872	46	421	645.00	0.117
(256, 32, 10)	5107	2872	46	421	1290.00	0.117
(256, 32, 13)	5113	2872	46	421	1290.00	0.117
(256, 32, 15)	5107	2872	46	421	1290.00	0.117
(256, 32, 20)	5112	2873	46	421	1290.00	0.117
(256, 64, 10)	5113	2873	46	421	2580.01	0.117
(256, 64, 13)	5113	2873	46	421	2580.01	0.117
(256, 64, 15)	5115	2873	46	421	2580.01	0.117
(256, 64, 20)	5116	2874	46	421	2580.01	0.117
(512, 8, 10)	6216	3269	52	809	335.66	0.140
(512, 8, 13)	6218	3269	52	809	335.66	0.140
(512, 8, 15)	6221	3269	52	809	335.66	0.140
(512, 8, 20)	6224	3270	52	809	335.66	0.140
(512, 16, 10)	6222	3270	52	809	671.31	0.140
(512, 16, 13)	6217	3270	52	809	671.31	0.140
(512, 16, 15)	6215	3270	52	809	671.31	0.140
(512, 16, 20)	6222	3271	52	809	671.31	0.140
(512, 32, 10)	6221	3271	52	809	1342.63	0.140
(512, 32, 13)	6221	3271	52	809	1342.63	0.140
(512, 32, 15)	6219	3271	52	809	1342.63	0.140
(512, 32, 20)	6226	3272	52	809	1342.63	0.140
(512, 64, 10)	6219	3272	52	809	2685.25	0.140
(512, 64, 13)	6221	3272	52	809	2685.25	0.140
(512, 64, 15)	6222	3272	52	809	2685.25	0.140
(512, 64, 20)	6228	3273	52	809	2685.25	0.140
(1024, 8, 10)	7802	3716	58	1581	343.51	0.173
(1024, 8, 13)	7805	3716	58	1581	343.51	0.173
(1024, 8, 15)	7810	3716	58	1581	343.51	0.173
(1024, 8, 20)	7813	3717	58	1581	343.51	0.173
(1024, 16, 10)	7806	3717	58	1581	687.02	0.173
(1024, 16, 13)	7807	3717	58	1581	687.02	0.173
(1024, 16, 15)	7809	3717	58	1581	687.02	0.173
(1024, 16, 20)	7809	3718	58	1581	687.02	0.173
(1024, 32, 10)	7806	3718	58	1581	1374.05	0.173
(1024, 32, 13)	7809	3718	58	1581	1374.05	0.173
(1024, 32, 15)	7805	3718	58	1581	1374.05	0.173
(1024, 32, 20)	7810	3719	58	1581	1374.05	0.173
(1024, 64, 10)	7807	3719	58	1581	2748.09	0.173
(1024, 64, 13)	7807	3719	58	1581	2748.09	0.173
(1024, 64, 15)	7808	3719	58	1581	2748.09	0.173
(1024, 64, 20)	7813	3720	58	1581	2748.09	0.173

References

1. Baevski, A.; Zhou, Y.; Mohamed, A.; Auli, M. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. In Proceedings of the Advances in Neural Information Processing Systems, Virtual, 6–12 December 2020; Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H., Eds.; Curran Associates, Inc.: Nice, France, 2020; Volume 33, pp. 12449–12460.
2. Zhang, Y.; Suda, N.; Lai, L.; Chandra, V. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv* **2018**, arXiv:cs.SD/1711.07128.
3. Fariselli, M.; Rusci, M.; Cambonie, J.; Flamand, E. Integer-Only Approximated MFCC for Ultra-Low Power Audio NN Processing on Multi-Core MCUs. In Proceedings of the 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS), Washington, DC, USA, 6–9 June 2021; pp. 1–4. [CrossRef]

4. Paul S, B.S.; Glittas, A.X.; Gopalakrishnan, L. A low latency modular-level deeply integrated MFCC feature extraction architecture for speech recognition. *Integration* **2021**, *76*, 69–75. [CrossRef]
5. Bae, S.; Kim, H.; Lee, S.; Jung, Y. FPGA Implementation of Keyword Spotting System Using Depthwise Separable Binarized and Ternarized Neural Networks. *Sensors* **2023**, *23*, 5701. [CrossRef] [PubMed]
6. Zhang, Y.; Qiu, X.; Li, Q.; Qiao, F.; Wei, Q.; Luo, L.; Yang, H. Optimization and Evaluation of Energy-Efficient Mixed-Signal MFCC Feature Extraction Architecture. In Proceedings of the 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Limassol, Cyprus, 6–8 July 2020; pp. 506–511. [CrossRef]
7. Vreča, J.; Biasizzo, A. Towards Deploying Highly Quantized Neural Networks on FPGA Using Chisel. In Proceedings of the 26th Euromicro Conference on Digital System Design (DSD), Durres, Albania, 6–8 September 2023; IEEE: Piscataway, NJ, USA, 2023.
8. Abdul, Z.K.; Al-Talabani, A.K. Mel Frequency Cepstral Coefficient and its Applications: A Review. *IEEE Access* **2022**, *10*, 122136–122158. [CrossRef]
9. Allen, J. Short term spectral analysis, synthesis, and modification by discrete Fourier transform. *IEEE Trans. Acoust. Speech Signal Process.* **1977**, *25*, 235–238. [CrossRef]
10. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. Chisel: Constructing Hardware in a Scala Embedded Language. In Proceedings of the 49th Annual Design Automation Conference, New York, NY, USA, 3–7 June 2012; pp. 1216–1225. [CrossRef]
11. Milovanović, V.M.; Petrović, M.L. A Highly Parametrizable Chisel HCL Generator of Single-Path Delay Feedback FFT Processors. In Proceedings of the 2019 IEEE 31st International Conference on Microelectronics (MIEL), Niš, Serbia, 16–18 September 2019; pp. 247–250. [CrossRef]
12. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv* **2018**, arXiv:cs.CL/1804.03209.
13. Chollet, F. Xception: Deep Learning With Depthwise Separable Convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017.
14. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861.