Contents

1 Abstract

Automatic Speech Recognition (ASR) has become very prominent in recent years for Internet of Things (IoT) devices. One of the steps in many ASR strategies is to extract Mel filter bank features from the input audio stream. In this project we implement the extraction of these features on a Field Programmable Gate Array (FPGA). Our implementation is on par with similar previous implementations in terms of the amount of gates required, but it is capable of operating at a clock frequency of $200MHz$, which is higher than any previous implementation we are aware of. In terms of latency, it requires only 1526 clock cycles for a frame of 512 samples, which takes $7.63/mus$ at $200MHz$. This makes it an improvement by 2 orders of magnitude compared to extracting the Mel features with a CPU.

2 Background

In ASR tasks, Recurrent Neural Networks (RNN) are often used, because they have shown great promise in processing sequential data[6]. A DeltaRNN (DRNN) is a modification to a standard RNN that only updates neurons when their activations change more than a certain delta threshold[7]. This saves computes, making it useful in tasks where hardware or power is limited, or if having low latency is important.

When the input is in the form of a constant audio stream, some pre-processing is required before feeding it to the neural network. A popular choice for the input representation is Mel Frequency Components[8]. These are basically filter banks applied to Short Time Fourier Transforms (STFT). The filter banks are spread out in a way to reflect human sound perception. The reasoning behind this is that human speech and hearing co-evolved, so the information in speech would be spread across frequencies in such a way that human hearing would accurately detect it.

Although extracting these Mel features requires many operations, which could lead to a high latency, but in speech recognition tasks it is usually desired to have the system respond very quickly.

FPGAs are often used to implement processes in which latency is to be minimized. An FPGA is a chip that has already been fabricated, but the hardware can be further defined using a Hardware Description Language (HDL). Although FPGAs are more time consuming to program, they tend to be much faster during run time than a processor running code written in a higher level language, such as C.

3 Introduction

In previous work, a DRNN system that does spoken digit recognition has been implemented on a Xilinx Zynq-7100 FPGA [4], although the extraction of the Mel filter bank features was implemented on a ARM CPU.

In this project, we aim to implement the Mel feature extraction on an FPGA using a low level HDL in order to improve upon the efficiency and latency of the DRNN system. We did this by separating the calculation of the Mel features into various steps and creating modules in System Verilog for each one. Then we combined all these submodules into one big module. The target frequency is $200MHz$ and the target board is the miniZed.

In Section 4 we describe the steps necessary to calculate the Mel features, how they are implemented and how to use it. In Section 5 we describe our results and compare it with similar work. Finally in Section 6 we talk about possible improvements and conclude our work.

4 Methods

In order to calculate the Mel features from a continuous audio input stream, a few steps have to be taken. Section 4.1 explains what these steps are and Section 4.2 explains how they are implemented in modules for the FPGA. Section 4.3 explains some considerations a user must take when using these modules.

4.1 Theory

4.1.1 Pre-emphasis

The first processing that is done to the input audio stream is to apply a pre-emphasis filter. Specifically the filter used here attenuates the low frequencies and emphasizes high frequencies. The purpose is to balance out the frequency spectrum, because human speech tends to contain more low frequencies. We use the finite impulse response(FIR) high pass filter shown in Equation 1.

$$x_{filtered}(n) = x(n) - \alpha x(n-1) \tag{1}$$

where x is the input signal, $x_{filtered}$ is the filtered signal, n is the sample index and α is the filter constant for which we used a value of 0.97.

4.1.2 Framing

In a speech signal, the interesting information is contained in how the frequency contours change over time. We do not want to know the spectrum over the entire signal, but rather the spectra over short snippets of the signal which we assume to be stationary. For a speech signal, these are typically around $25ms$. If we use a sample rate of $20kHz$, that corresponds to 500 samples. If the frame length is a power of 2 the design could exploit this and more efficient hardware could be designed. Therefore we choose a default frame length of $N = 512$.

We overlap these frames in order to increase the time resolution of the spectrogram. We use a stride of 256 samples to get an overlap of 50% between frames. Again, this value has to be a power of 2 in order to design for more efficient hardware.

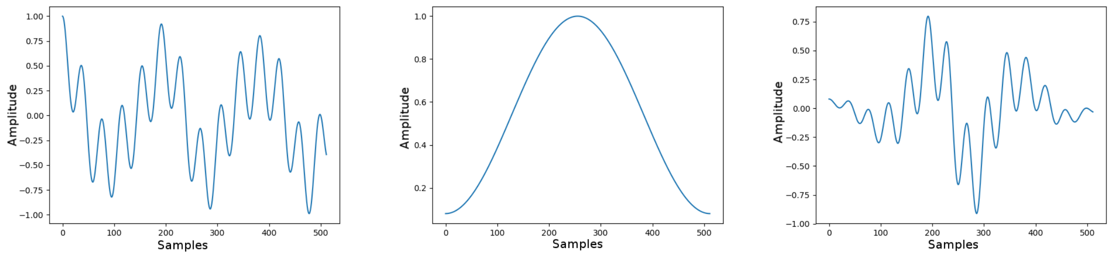
4.1.3 Hamming Window

When a Discrete Fourier Transform (DFT) is done on one of these frames, it assumes that the signal is stationary. If the last sample in this frame is very different to the first sample, there is a jump in the signal that manifests as unwanted high frequencies in the spectrum. To mitigate this problem we apply a Hamming window to the frame. This means multiplying every sample in the frame with the corresponding sample in the window, as shown in Figure 1. The Hamming window is described by Equation 2 and how it is applied to a signal is described by equation 3.

$$w(n) = a_0 - (1 - a_0)cos(\frac{2\pi n}{N-1}) \tag{2}$$

$$x_{windowed}(n) = x_{filtered}(n) \times w(n) \tag{3}$$

where w is the hamming window, a_0 is set to 0.54, N is the number of samples per frame and $x_{windowed}$ is the windowed signal.



(a) Signal of single frame (b) Hamming window (c) Product of frame and window

Figure 1: Applying a Hamming window to a frame

4.1.4 Power Spectrum

In order to calculate the power spectrum, one first takes a DFT of the windowed frame to get the frequency spectrum of that frame. Then one takes the square of the magnitude spectrum, and divides it by the number of samples per frame. The entire process is shown in Equations 4 and 5.

$$X = FFT(x_{windowed}) \tag{4}$$

$$P(f) = \frac{X_r(f)^2 + X_i(f)^2}{N} \tag{5}$$

where X_r and X_i are the real and imaginary parts of the frequency spectrum respectively, and P is the power spectrum.

Since the power spectrum is symmetrical about the index $\frac{N}{2}$, we only require the lower half of the spectrum. So only $\frac{N}{2}$ values are sent to the filter bank.

4.1.5 Filter Banks

The next step is to apply a Mel filter bank to the power spectrum. In a Mel filter bank the center frequencies of the filters are arranged to correspond to the resolution of human hearing. Filtering a signal in this way is useful for speech recognition tasks, because human speech and human hearing co-evolved to use the same frequencies. Converting from Hertz(f) to Mel(m) and vice versa is done using equations 6 and 7 respectively.

$$m = 2595log_{10}(1 + \frac{f}{700}) \tag{6}$$

$$f = 700(10^{m/2595} - 1) \tag{7}$$

The Mel filter bank consists of triangular filters with a gain of 1 at their center frequencies, and a gain of 0 at the center frequencies of the two adjacent filters. The full filter bank for 40 filters with a sampling rate of $20kHz$ and a frame size of $N = 512$ is shown in Figure 2.

The final step once we have the Mel features is to convert these values to decibels. Converting to decibels is done with Equation 8.

$$z_{dB} = 20 \times log_{10}(z) \tag{8}$$

where z is one of the Mel features.

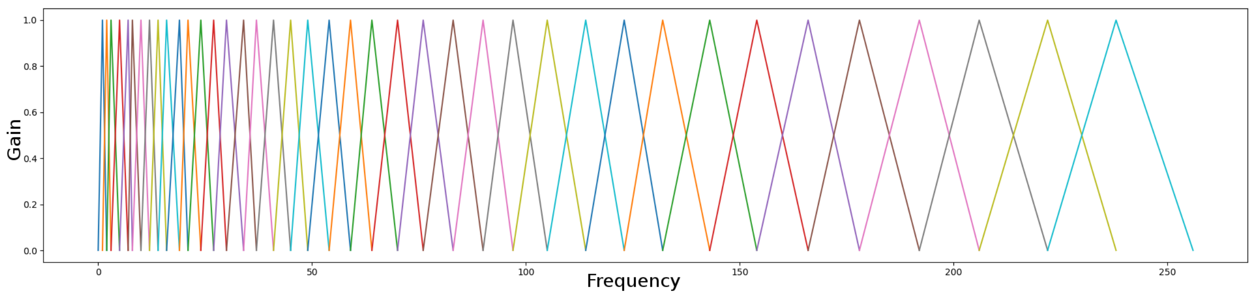


Figure 2: Mel filter bank

4.2 FPGA Modules

4.2.1 Prepare Input

The first module, *prepare_input*, prepares the input by applying the pre-emphasis filter and slicing the input into frames. The input to this module is a buffered sequence of 16 bit numbers representing the samples of audio data.

A simplified block diagram of this module is shown in Figure 3. The multiplier is used to apply the pre-emphasis filter by multiplying the input signal by a_0 as shown in Equation 1. The red line represents a register and is used to delay that signal by one clock cycle. Subtracting this signal from the current input gives the output of the pre-emphasis filter.

Then there is a shift register that holds the samples of the frame. When a valid input is provided, it is pushed into the shift register, and when a frame is being put out, the shift register rotates N times while the output is assigned the last value in the shift register. After N rotations, the shift register is back where it started.

Knowing when to provide an output is what the counters are for. We constrained the stride length to be a power of 2. The *stride_counter* is of size $log_2(STRIDE)$ and is incremented every time a valid input is received. Therefore once a stride length number of inputs are received, *stride_counter* is reset to 0. This means that we want to output a frame every time the stride counter becomes 0.

The purpose of *out_counter* is to make sure that the output frame is of length N. It starts at 0 when a frame is being put out, increments for every output sample, and once it reaches N it stops the output.

4.2.2 Hamming Window

The *hamming* module simply consists of a shift register of size N containing all the values of the Hamming window and a multiplier to multiply the sample passing through the module with the current output of the shift register. For each sample, the shift register rotates once, so after one frame it rotated N times and is back where it started.

Since the Hamming window is symmetrical, we tried using a shift register of half the size while rotating it in one direction $\frac{N}{2}$ times and then in the other direction $\frac{N}{2}$ times, but this turned out to be more expensive than just having a full size shift register.

Since we kept with the full size register, a user can implement a non symmetrical window function if they choose. All one has to do to use a different window function is replace the values inside the shift register. More details of how to do that will be given in Section 4.3.

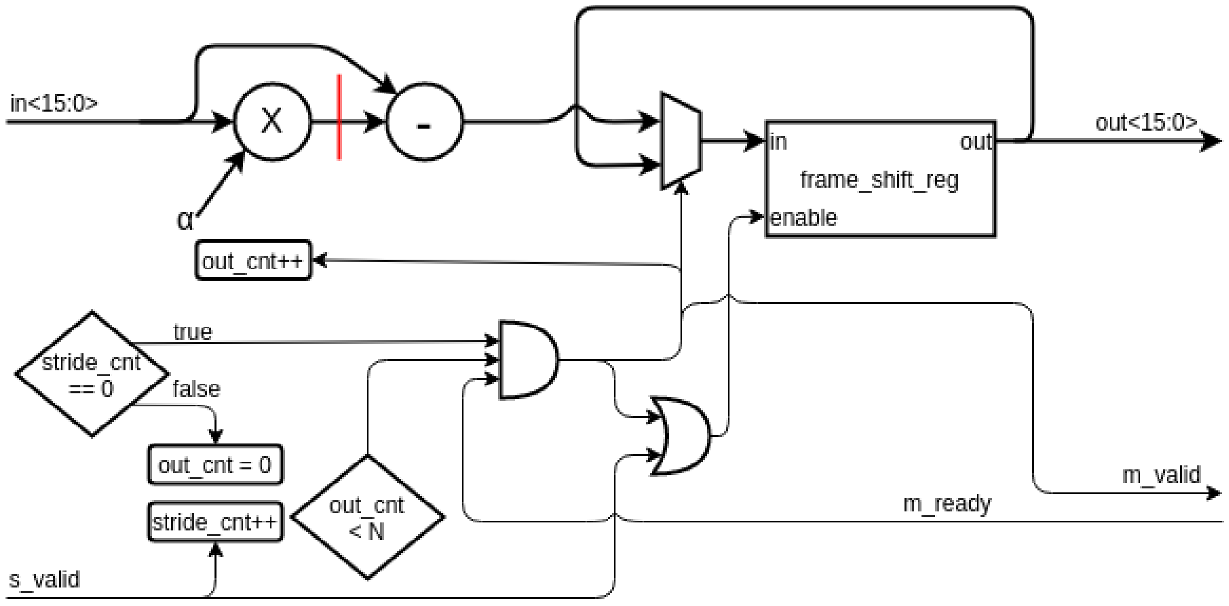


Figure 3: Simplified block diagram of the module that prepares the input.

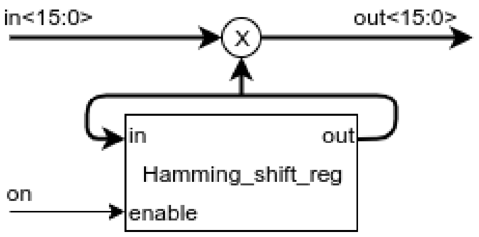


Figure 4: Simplified block diagram of the module that applies the Hamming window.

4.2.3 Power Spectrum

The first step in order to calculate the power spectrum is to do an FFT, for which we use an IP. We then square the real and imaginary output of the FFT using multipliers and sum these together. Since N is a power of 2, dividing by N is simply done by shifting the binary number right $log_2(N)$ times. The red line represents a register that is used to meet the timing constraints.

As explained earlier, we are only interested in the first half of the power spectrum. In order to stop the FFT IP from putting out the second half, we reset it after $\frac{N}{2}$ valid outputs have been given. This is what the counter is used for.

4.2.4 Filter Bank

The next module, *filter_bank*, is the one that calculates the Mel features. A shortcut that can be taken here is to realise that every point in the spectrum that is not at the center frequency of a filter contributes to two filters. And the gain of one of the filters at that point is equal to 1 minus the gain of the other. This means

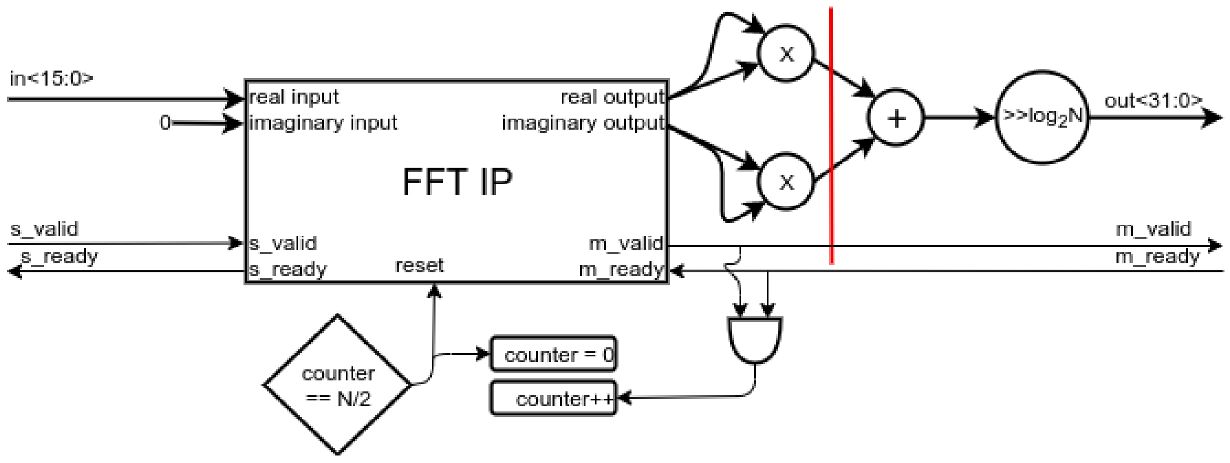


Figure 5: Simplified block diagram of the module that calculates the power spectrum.

that only one multiplication is necessary to calculate the contribution of one frequency to two filters.

The module contains a shift register, called *filt_shift_reg*, which contains the values that have to be multiplied with the input. These values are shown in Figure 6, and they are simply the ascending sides of all the filters shown in Figure 2. This shift register rotates once for every valid input and the signal at its output is called *filt*.

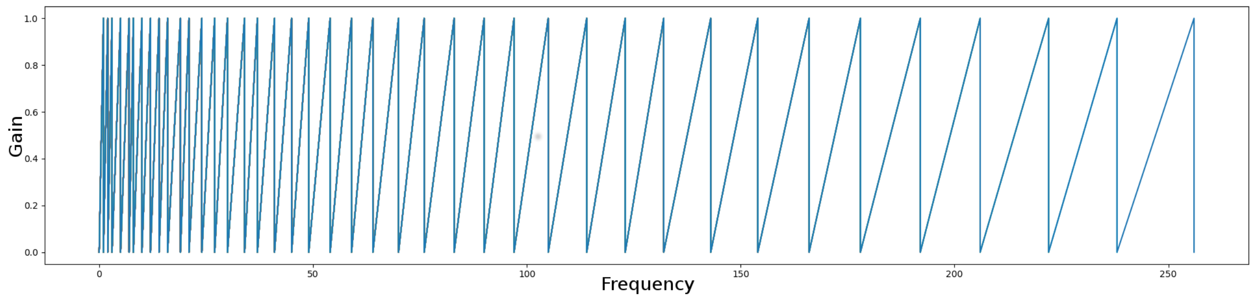


Figure 6: Values in *filt_shift_reg*

The contribution of each input frequency to the filter of which it is in the ascending side (less than the center frequency) is given by the product of *filt* and *in*. In Figure 7 we call this signal *product*. The contribution of the input to the filter of which it is in the descending side (greater than the center frequency) is given by the difference of *in* and *product*. This is the signal labeled *inverse*.

In Figure 7 there are two loops, one containing the *ascending* signal and one containing the *descending* signal. These calculate the contributions to the two filters respectively. When *filt* corresponds to a gain of one, it means the current input frequency corresponds to the center of a certain filter. At this point, four things happen.

Firstly, since we are at the center of a certain filter, we are moving from its ascending side to its descending side. Therefore we transfer the value from the ascending loop to the descending loop. This is done by having the *accumulation* signal read by the descending loop rather than the ascending loop.

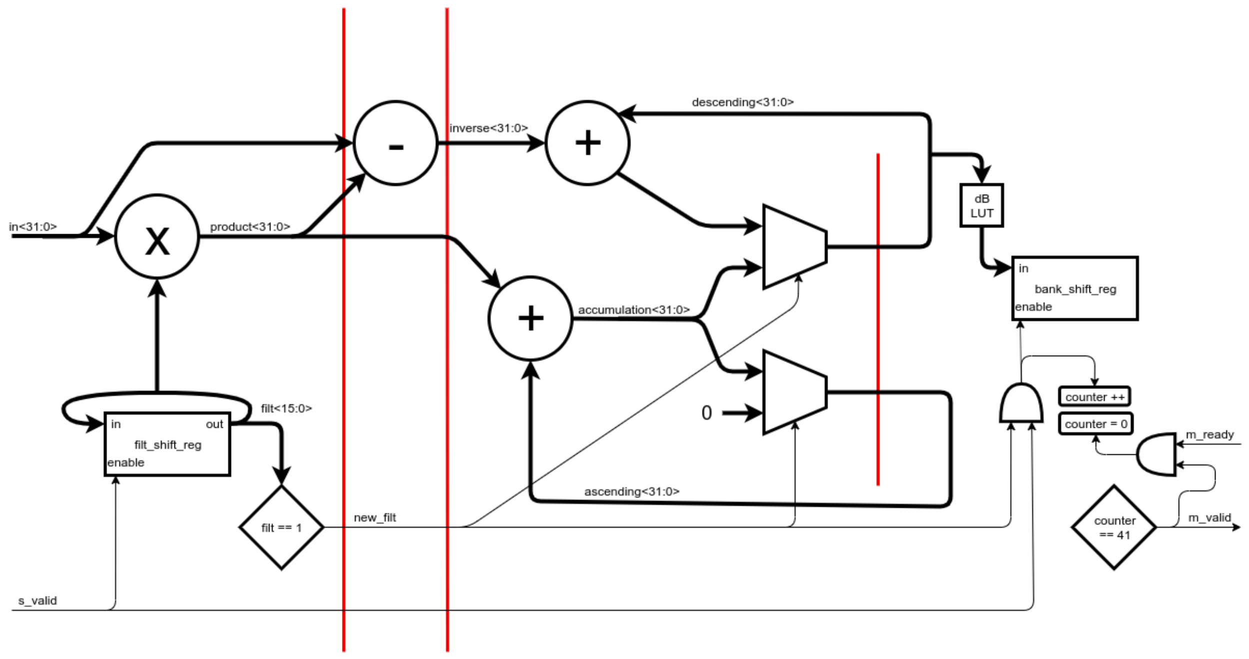


Figure 7: Simplified block diagram of the module that calculates the Mel features.

Secondly, the value of *descending* is the complete energy of the previous filter, since the center of the current filter correspond to the upper corner of the previous filter. Therefore we convert this value to decibels with a lookup table and save it to the shift register called *bank_shift_reg*.

Thirdly, we reset *ascending* to 0 to start the calculation of the next filter.

And fourth, we increment a counter. When this counter reaches 41, we know we have reached the end of the highest filter. At this point we have all the Mel features stored inside *bank_shift_reg*, so we set *m_valid* to 1. When *m_ready* is 1, we reset the counter to 0 and the module is ready to start processing the next input.

4.2.5 Reshape Output

The final module is used to reshape the output, since the 40 parallel 16 bit numbers that the previous module outputs is not ideal in many cases. Figure 8 shows an example for an output width of 4 parallel 16 bit numbers. When *s_ready* and *s_valid* are 1 the values from *bank_shift_reg* from the filter banks module is written to the shift registers of this module.

In this example, since the output width is 4, *shift_reg_0* contains every fourth value starting at 0, *shift_reg_1* contains every fourth value starting at 1 and so forth. Therefore the first output is the values from the indices 0 to 3 inclusive, the second output is from 4 to 7, and so on until 36 to 39 are being put out.

The counter increments for every valid output that is sent, and since $\frac{40}{4} = 10$, that means there is a sequence of 10 valid outputs per frame. So the process stops when the counter reaches 10 and resets when a new valid input is presented.

4.2.6 Full System

Finally, the modules are connected together to create a new module of the full system, as seen in Figure 9. A few pipelines are added to meet timing constraints.

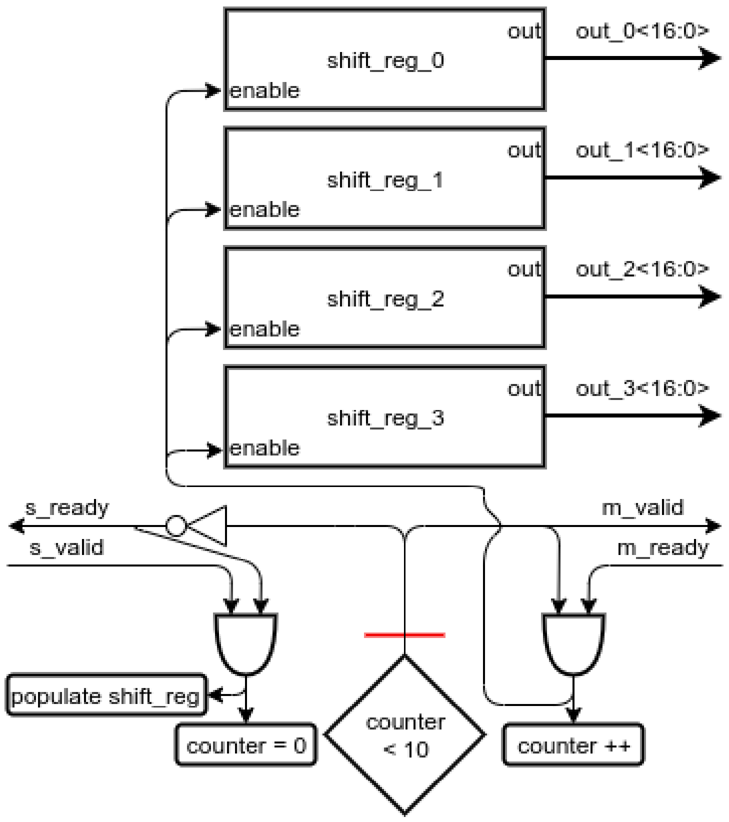


Figure 8: Simplified block diagram of the module that reshapes the output to a user configurable width. For this example the output width is 4 16 bit numbers.

4.3 Using the module

4.3.1 Interface

The AXI Stream protocol is used for the input and output of this module. In this protocol, there are handshake signals in addition to the data signals. A master sets *valid* high when the data that is currently being transmitted is valid, and the slave sets *ready* high to indicate that it is ready to receive data. AXI Stream ports also sometimes contain a *last* signal, which the master sets high during the last clock cycle in which data for a specific packet is being transmitted. In this project we only used a *last* port on the output, because the input audio stream is continuous. There is no separation between packets.

Besides the AXI Stream input and output interfaces, the usual clock and reset signals can also be found.

4.3.2 Parameters

The module has 3 assignable parameters. N sets the frame length, $STRIDE$ sets the stride length and OUT_WIDTH sets the output width. N and $STRIDE$ both have to be powers of 2, and OUT_WIDTH must be a factor of 40.

When N is set to something other than 512, some changes have to be made in the code. The FFT IP has to be specified a new frame length, which can be done in Vivado with a GUI by double clicking on the FFT

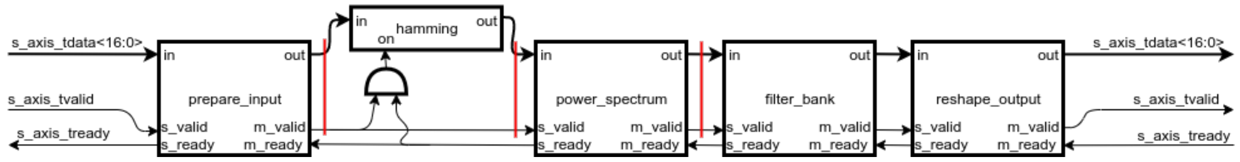


Figure 9: Simplified block diagram of the full system.

module. There are two shift registers that also have to be redefined, one in the *hamming* module, and one in the *filter_bank* module. To generate the new shift registers, run the python functions found in Appendix A named *generate_hamming_shift_reg()* and *generate_filt_bank_shift_reg()* and copy their output into the places indicated in the modules.

If a Hamming window with an a_0 other than 0.54 is to be used, the *generate_hamming_shift_reg()* python function must be run again to generate the relevant shift register.

If a sampling rate other than $20kHz$ is to be used, the *generate_filt_bank_shift_reg()* must be run again, because the locations of the Mel filter banks rely on the sampling rate.

4.3.3 Design Considerations

The pre-emphasis filter could cause overflow if there are very loud very high frequencies present in the audio signal. Specifically, it causes overflow when 2 consecutive samples have opposite signs and are greater than about half the maximum amplitude. To solve this problem, an increased bit depth could be used at the output of this filter, but since such loud high frequencies are not usually present in everyday situations, we decided against it. If the module is to be used in situations where loud high frequencies are expected, we recommend just reducing or disabling the pre-emphasis filter. This can be done by changing the *filt* value in the *prepare_input* module. Setting it to 0 disables the filter.

Asynchronous reset of of the shift registers containing the Hamming window coefficients and the Mel filter banks is very expensive in hardware, therefore it is currently disabled. If the ability to asynchronously reset these shift registers is required, just uncomment the lines indicated in the *hamming* and *filter_bank* modules as shown in Appendix B.

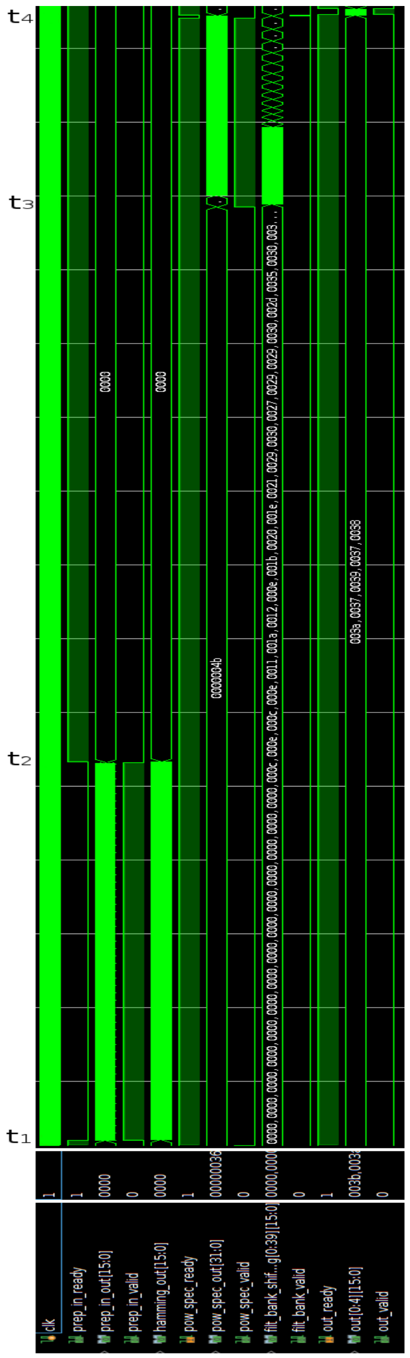
The FFT IP is used in scaled mode, which means after every butterfly (or dragonfly) in the FFT algorithm the numbers are scaled down to prevent overflow. Another option would be to just increase the bit depth after every butterfly, but this is very expensive in hardware. In this project it would be especially expensive, because calculating the power spectrum requires multipliers directly at the output of the FFT module, and multipliers that can handle high bit numbers are very expensive. Specifically, 2 32 bit multipliers are required and, unless pipe-lined, they do not meet the timing constraints at $200MHz$.

5 Results

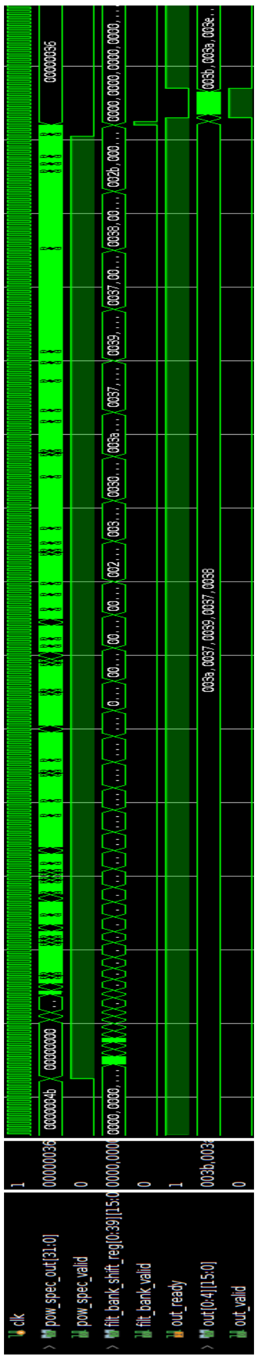
5.1 Behavioral simulation

Figure 10 shows the behavioural simulation for a frame size of 512 and output width of 5. Figure 10a is from the point when the *prepare_input* module is outputting a frame up to when the Mel filter bank features have been calculated for that frame. Figure 10b is from the point when the FFT has been calculated up to the same end as Figure 10a.

In Figure 10a, from t_1 to t_2, the *prepare_input* module is serially putting out one frame. The *hamming* module applies the relevant coefficient to each sample and feeds the windowed frame to the *power_spectrum* module. Then up to t_3, we are waiting for the FFT IP inside the *power_spectrum* module to finish calculating



(a) One frame being calculated.



(b) Filter bank being filled

Figure 10: Behavioural simulation. The opaque green bars show when a signal has a lot of activity.

	clock cycles	cumulative clock cycles	percentage of total
t_1 to t_2	513	513	34%
t_2 to t_3	752	1265	49%
t_3 to t_4	261	1526	17%

Table 1: Clock cycles required for each stage of the calculation.

the FFT. From t_3 to t_4, the *power_spectrum* module is serially putting out the power spectrum. At the same time, the *filter_bank* module is using the power spectrum to calculate the Mel features. After each Mel feature has been calculated it is pushed into *filt_bank_shift_reg*. At t_4, the *reshape_output* module is putting out the contents of that shift register.

Figure 10b shows the same as Figure 10a, but zoomed into the t_3 to t_4 region. Notice how the time between the changes in values stored in *filt_bank_shift_register* increases over time. This shows the logarithmic spacing between frequencies in the Mel scale. Once the *filt_bank_shift_register* is populated, the *filter_bank* module's output is valid for one clock cycle (given that the next module is ready). After receiving this valid input, the *reshape_output* module starts outputting the Mel features. This module stops being ready as it is putting out these values, and is ready again afterwards.

Table 1 shows the clock cycles required by each stage of the calculation. The total latency from t_1 to t_4 is 1526 clock cycles. At a clock frequency of $200MHz$, this corresponds to $7.63\mu s$.

Previously the DRNN system calculated the Mel features with an ARM core programmed in C. We wanted to compare the latency, but could not directly measure the clock cycles used by the ARM core. So we ran the same C code on a desktop computer and used the linux command *perf* to measure the required clock cycles. This test has some caveats, because the latency of C code is different on different platforms and also depends on which other tasks are running at the same time, but it is still useful for showing in which order of magnitude the required clock cycles would be. We measured the latency of calculating Mel features with a CPU to be around $400,000$ clock cycles, which at a typical CPU clock frequency of $667MHz$ would take about $600\mu s$. Our method only requires $\frac{7.63}{600} \approx 1.3\%$ of that, making it an improvement by 2 orders of magnitude.

5.2 Output Values

The results of the behavioural simulation have been compared to results from python code extracting Mel features for one frame of random noise. These results are shown in Figure 11.

When the FFT IP is used in unscaled mode, the results match almost perfectly, the only difference likely being due to the python code using floating point numbers and our system using fixed point numbers. When the FFT IP is used in scaled mode, the spectral shape is preserved, all the values are just scaled down. However the small values become clipped, which would drastically impair performance when the input signal is not sufficiently loud.

5.3 Estimated Efficiency

Vivado provides tools to estimate the power consumption and the timing of the design. The timing shows a worst negative slack of $0.14ns$ and a worst hold slack of $0.024ns$. Since these values are positive, the timing constraints are met.

Figure 12 shows the breakdown of the power consumption. The total power consumption is $0.282W$ which would lead to the FPGA having a junction temperature of 28.3^oC. This means passive cooling would be enough. Although this power consumption and junction temperature is satisfactory, in practice it should

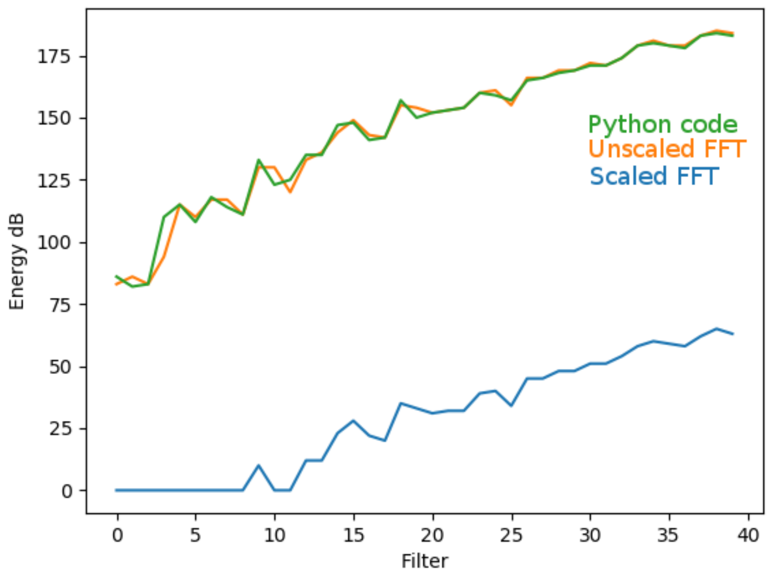


Figure 11: Comparison of our output values with python generated values for Mel feature extraction performed on 1 frame of random noise.

be even more efficient, since with a sample rate of $20kHz$ and a clock of $200MHz$, a valid frame is only provided once every $10,000$ clock cycles on average.

Table 2 shows the utilization of the FPGA hardware resources on a miniZed board.

5.4 Comparison with Previous Work

Table 3 shows how our module compares with previous work in terms of FPGA utilization. Some of the previous work performed a cosine transform on the Mel filter bank features to decorrelate them as well as reduce the output dimensionality. Whether or not this is done affects the hardware utilization, so it is included in the table. We did not implement this transform, because for an RNN it does not make a big difference if the input is correlated or not. Implementing this transform would only cause the design to have a greater latency and require more hardware.

	Utilization	Available	Utilization %
LUT	3895	14400	27
LUTRAM	816	6000	14
FF	3684	28800	13
BRAM	3.5	50	7
DSP	14	66	21
BUFG	2	32	6

Table 2: Hardware utilization of entire design on miniZed board

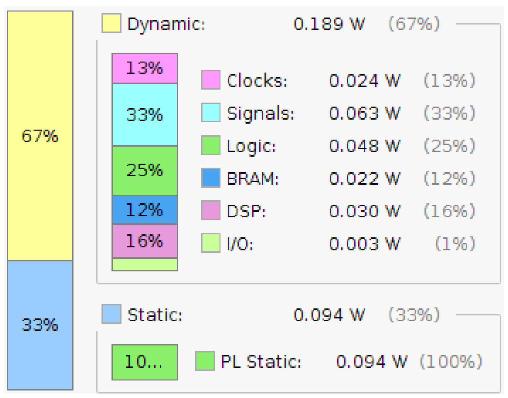


Figure 12: Estimated power consumption of the entire design.

The frame size also affects the hardware utilization, so it is also included in the table. We set it to 256 to match most of the other implementations for a fair comparison.

Comparing the values in Table 3 shows that our design is capable of working at the highest clock frequency. This is because our target frequency was $200MHz$, so we modified our design, mainly through pipelining, until we achieved this. It could also be due to us using a more modern board. The rest of the values in the table show that our design is roughly on par with the best designs out there in terms of hardware utilization.

	frequency (MHz)	DCT	frame size	LUT	FF	BRAM	DSP
M. Bahoura[1]	65.5	yes	1024	14837	13726	4	57
V.L Dao[2] (design 1)	139	yes	256	5753	11567	8	10
V.L Dao[2] (design 2)	52	yes	256	11581	11335	15	11
P. Ehkan[3]	-	yes	256	35104	23251	14	33
D. Ghosh[5]	50	no	-	1833	1906	-	13
J.C. Wang[9]	-	yes	256	5692	2732	-	-
G. Wassi[10]	155	yes	256	8193	11577	44	82
This work	200	no	256	3475	3354	3.5	14

Table 3: FPGA utilization of our design compared to previous work.

6 Discussion

6.1 Possible Improvements

There are a few improvements that could be done. Firstly, under the assumption that the time domain signal is always real valued, the FFT can be improved[9] both in terms of latency and hardware utilization. Since the majority of the latency is currently used to calculate the FFT, as seen in Table 1, this could yield a significant reduction in latency.

Secondly, since the submodules run at different times, some hardware could be reused. For example, the multipliers in the *prepare_input* and the *hamming* modules are only used from t_1 to t_2 as defined in Figure 10a, and the multipliers that square the output of the FFT IP are only used from t_3 to t_4. We can reuse the same 2 multipliers in both cases.

6.2 Conclusion

In this project we implemented Mel feature extraction on an FPGA specifically for use in a DRNN system, but it can be used in many other applications also. Our implementation is on par with previous similar work in terms of hardware utilization, but it is capable of running at higher clock frequencies. It is also an improvement by 2 orders of magnitude in terms of latency compared to extracting the Mel features with a CPU, which is how the DRNN did it previously.

7 Acknowledgements

I would like to thank my Supervisor, Proffesor Shih-Shii Liu, for helping out with the report. I would also like to thank Chang Gao for helping with the project.

Appendices

A Python Functions

```
import numpy as np

def generate_hamming_shift_reg(n=512, a0=0.54):
        print('logic[15:0]␣shift_reg[N]␣=␣{', end = '')
        for i in range(int(n)):
                x = a0-(1-a0)*np.cos(2*np.pi*i/(n-1))
                x = round(2**14*x)
                print('16\'d' + str(x), end = '')
                if(i!=n-1):
                        print(',␣', end = '')
                else:
                        print('};')

def generate_filt_bank_shift_reg(n=512, sample_rate=20000, nfilt=40):
        print("logic[15:0]␣filt_shift_reg[N]␣=␣{", end = '')
        mel_banks(find_mel_pts(n=n,
                                                sample_rate=sample_rate,
                                                nfilt=nfilt),
                          n)

def find_mel_pts(n=512, nfilt=40, sample_rate=20000):
        low_freq_mel = 0
        high_freq_mel = (2595 * np.log10(1 + (sample_rate / 2) / 700))
        mel_points = np.linspace(low_freq_mel, high_freq_mel, nfilt + 2)
        hz_points = (700 * (10**(mel_points / 2595) - 1))
        bins = np.floor((n + 1) * hz_points / sample_rate)
        return(bins)

def mel_banks(bins, n):
        x = 0
        for i in range(1, len(bins)):
                size = int(bins[i]-bins[i-1])
                x = np.hstack([x, np.linspace(0, 1, size+1)[1:]])

        for i in range(1, len(x)):
                z = round(2**15*x[i])
                print("16'd" + str(int(z)), end = '')
                if(i!= n/2):
                        print(',␣', end = '')
                else:
                        print('};')
```

B Enabling Asynchronous Reset of Shift Registers

For asynchronous reset of the Hamming shift register, uncomment these lines in the *hamming* module.

```
always_ff @(posedge clk) begin

    // uncomment the lines below to enable asynchronous reset
//          if (reset) begin
//              shift_reg = {16'd1311, 16'd1311, 16'd1313, 16'd1316...
//          end
//          else begin
                if(on) begin
                    shift_reg[0] <= shift_reg[N-1];
                    shift_reg[1:N-1] <= shift_reg[0:N-2];
                end
//          end
    end
```

For asynchronous reset of the filter bank shift register, uncomment these lines in the *fliter_bank* module.

```
always_ff @(posedge clk or posedge reset) begin
        if(reset) begin
            in_ready <= 0;
            q_in <= 0;
            new_window[1:2] <= {0,0};
            in_valid[1:2] <= {0,0};
            scaled_product[0:1] <= {0,0};
            inverse[1] <= 0;
            q_counter <= 0;
            q_ascending <= 0;
            q_descending <= 0;
            // uncomment the line below to enable asynchronous reset
//              filt_shift_reg = {16'd32768, 16'd32768, 16'd32768...


        end
```

C System Verilog Module Instantiation

```
mfcc #(.N(), .STRIDE(), .OUT_WIDTH()) instance_name
      (.clk(),
       .reset(),
       .s_axis_tdata(),
       .s_axis_tready(),
       .s_axis_tvalid(),
       .m_axis_tdata(),
       .m_axis_tready(),
       .m_axis_tvalid(),
       .m_axis_tlast());
```

References

[1] M. Bahoura and H. Ezzaidi. “Hardware implementation of MFCC feature extraction for respiratory sounds analysis”. In: *WoSSPA* (2013).

[2] V.L Dao et al. “Hardware Implementation of MFCC Feature Extraction for Speech Recognition on FPGA”. In: *Advances in Intelligent Systems and Computing* (2017).

[3] P. Ehkan et al. “Hardware Implementation of MFCC-Based Feature Extraction for Speaker Recognition”. In: *Springer International Publishing Switzerland* (2015).

[4] C. Gao et al. “DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator”. In: *FPGA* (2018).

[5] D. Ghosh, D.S. Debnath, and S. Bose. “A Comparative Study of Performance of FPGA Based Mel Filter Bank & Bark Filter Bank”. In: *IJAIA* (2012).

[6] Y. LeCun, Y. Bengio, and G. Hinton. “Deep Learning”. In: *Nature* 521.436 (2015).

[7] D. Neil et al. “Delta Networks for Optimized Recurrent Network Computation”. In: *ICML* (2017).

[8] B. Shannon and K. Paliwal. “A Comparative Study of Filter Bank Spacing for Speech Recognition”. In: *Microelectronic Engineering Research Conference* (2003).

[9] J.C. Wang, J.F. Wang, and Y.S. Weng. “Chip Design of MFCC Extraction for Speech Recognition”. In: *VLSI* (2002).

[10] G. Wassi et al. “FPGA-based Real-Time MFCC Extraction for Automatic Audio Indexing on FM Broadcast data”. In: *IEEE* (2015).