Blind Source Separation Using ICA

ECSE 444: Microprocessors

Jacob Shnaidman
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Canada
jacob.shnaidman.mcgill.ca

Charbel El Hachem
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Canada
charbel.elhachem@mail.mcgill.ca

Ege Ozer
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Canada
ege.ozer@mail.mcgill.ca

Jeremy Davis
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Canada
jeremy.davis@mail.mcgill.ca

***Abstract*—The first part of the report contains material from the first experiment. This includes generation, mixing and storing of sine wave. This is accomplished using CMSIS DSP library, which is stored in flash using QSPI, and played back using DMA with a DAC. The results showed that any combination of sine waves below 8000Hz could be stored and played back with good quality. The generation and mixing of sine waves is of interest in applications built for BSS using the ICA algorithm. In the other half of the report, the focus is on the second part of the experiment. More specifically, The implementation of FastICA and testing via separating the mixed sine wave are described in the other half of the report.**

I. Introduction

The objective of this experiment is to build an audio application that can perform Blind Source Separation (BSS) using the Fast Independent Components Analysis (ICA) algorithm. When sounds coming from two different sources are recorded with microphones, it is often desirable to separate the mixed sound coming from both these sources back into their original signals. This can be achieved by generating sine waves of different frequencies, and mixing them; this will produce two mixed sine wave signals. By measuring the difference in the mixed signals, audio separation can be achieved through the ICA algorithm, re-obtaining the two original source signals.

II. System Overview

In order to generate the mixed signals, the program generates angular values to pass to the DSP sin wave generation function, for two sine waves of different frequencies. The sine waves are then normalized and mixed using a 2x2 matrix providing the weights for each signal. Then, the BSP QSPI library is used to write the mixed sine waves to external flash as a contiguous list of floats. After this completes, the signals are read from flash multiple times to construct the ICA filter matrix. After the ICA filter matrix is created, the first 2000 samples are read from flash, unmixed using the ICA filter matrix, converted to a 12 bit integer and stored in a buffer for each DAC channel. Once these buffers are full, the DAC is started in DMA mode which transfers the unmixed signals to separate DAC output registers from these buffers. The DAC playback speed is controlled by configuring the DAC to use the TIM6 timer as an external trigger to perform DMA transfers. After the DMA has transferred an entire buffer, the DMA for the DAC triggers an interrupt which executes the ISR which calls our callback function at the end of the ISR. This callback function will read 2000 samples from flash and perform the unmixing operation, convert the values to 12 bit integers and fill the buffers for each DAC with the unmixed values. Although this implementation has a race condition between the DMA transfer speed and the unmixing operations that are filling the DAC's buffer that the DMA is using to transfer values to the DAC's output register, the relative speed of the processor compared to the output rate of the DAC is so large, that it's easily neglected. Figure 1 shows the hardware component interaction .

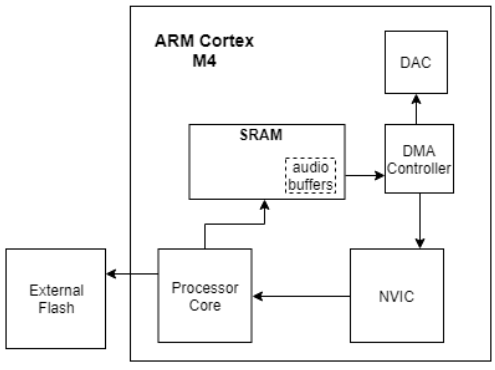

Fig. 1. Block diagram describing the relationship between the different hardware components

III. System Components and Parameters

A. DAC

The DAC (Digital to Analog) converter is what allows us to playback sine waves stored in memory. In this experiment, there are 2 DAC channels used: one for the left audio channel, and the other for the right audio channel. The DAC reads the values of the sine wave in memory, and converts these values to an approximate analog value. This element was chosen because it is necessary to generate an audio signal. The DAC was configured to use 12 bit resolution in order to reconstruct the signal as accurately as possible. The DAC was evaluated by looking at the waveform on the oscilloscope.

B. DMA

DMA (Direct Memory Access) is used to feed the DAC's data output register without using the CPU. There are two DMA handlers, one for each DMA channel. The DMA is set up to transfer halfwords from SRAM into the DAC's data output register 16000 times a second, triggered by the TIM6 timer. We chose to use DMA for the DAC since it reduces the load of the program on the CPU by delegating the task of moving output data to the DAC's data output register to the DMA controller. Once configured in circular mode, it only has to be started once and will always continue to run in the background. The DAC was configured to use halfword transfers, since the DAC values were 12 bits each. The DMA was configured to automatically increment the memory address, since all the samples in memory that were to be sent to the DAC were stored contiguously. This way, the processor did not need to be used as all in the transfer process using DMA. This was evaluated by verifying that the DMA interrupt was triggering interrupts and that the values in the buffer holding the 2000 samples of the mixed sine wave was changing. We also could see the data output register of the DAC continuously changing (in the debugger).

C. QSPI/Flash

The QSPI (Quad SPI) protocol was used to send the mixed sine waves to flash after the values were generated. QSPI was necessary to write and read to the external flash. Since the on-chip SRAM only contains 128KBytes of memory, and on-chip flash only 1MB, storing the entire signal can take up a lot of memory. In order to store 2 seconds of a sine wave sampled at 16,000Hz, this would require 32,000 audio samples,which at 12bits/sample, would require 64,000 bytes of memory for each signal. The advantage of using the external flash and QSPI is that it allows us to have access to much more memory.

The QSPI was configured with 0 as a prescaler for the best performance, since we had no power requirements. A FIFO threshold of 2 proved to be sufficient for this lab, . No sample shifting was needed, as we didn't need to account for any signal delay. The flash size that was set was 22, since this corresponds to 2^{22+1} megabytes in the external flash. We kept the chip select high time to 1, since this is all that the flash seemed to need in order to work. The clock mode was set to LOW so that the clock remains low between QSPI commands. The QSPI system was evaluated by seeing that the values that were written to flash were the same when they were read and put into the buffer holding 2000 samples of the mixed sine wave.

D. DSP Sine Wave Generation

In order to generate the sine waves, the CMSIS-DSP software library was used. In particular, we used the arm_sin_f32() function, which takes an input angle in 32 bit float, and returns a sine value in this same format. We used this sine wave generation function since it was provided by the CMSIS library, which we know is already optimized in assembly from lab 0. Since this was a library function, we didn't explicitly evaluate it.

E. Timers

We used the TIM6 timer to generate update events to trigger DMA transfers from memory to the DAC. This was necessary to ensure that audio is being output at 16kHz. We chose to use TIM6 since it shares the same interrupt with the DAC and the reference manual specifies TIM6 and TIM7 to be used specifically for the DAC. The count period of the timer was set to 5000, because we wanted to output a sample 16000 times a second with a 80MHz clock. Because it takes 5000 cycles to generate an update event to trigger a DMA transfer to the DAC, and we have 80 million cycles per second, we effectively had $\frac{80\text{MHz}}{5000} = 16\text{KHz}$ as our output rate. We evaluated this by verifying that the DAC DMA ISR was being triggered and by reading the frequency of the output on an oscilloscope.

F. Flash Storage Layout

We chose to store the generated sine wave in flash in the transposed form of what we would have preferred mathematically because it reduced how many reads from flash that we had to do and simplified reading from flash. Since reading from flash is really slow, and many transposes needed to be calculated anyways, we decided it was worth having to perform one extra transpose while calculating the weight for ICA rather than reading from flash from two different locations and complicating how the signals are stored and retrieved. This made the code much simpler since we did need to constantly read from flash to calculate the ICA filter, and the math we needed to perform already made the code complicated enough.

IV. Fast Independent Component Analysis

Blind Source Separation (BSS) is the process of separating the source signals from the mixed signals. This can be achieved using Fast Independent Component Analysis (FastICA). FastICA is an algorithm used for independent component analysis. It manipulates de-whitened data, through recursive gradient ascent. For this lab, the FastICA is used for finding the inverse matrix

V. FastICA Implementation

A. Signal Centering

First, the data are centered by subtracting the mean of each column of the data matrix. To do this, we have to calculate the means of each signal. The mean of each signal is calculated by iterating through the flash in 8000 byte chunks which is equivalent to 2000 floats and adding each value to the corresponding mean. At the end of this operation, we divide the two sums by 32000, which is the number of samples in each signal.

In the case of this experiment where the mixed signals are the sum of two sine waves, the average value of each mixed signal is approximately 0.

B. Mixed Signal Covariance Matrix

The covariance matrix of the matrix containing the mixed signals is calculated in chunks, since there is not enough SRAM memory to operate on the mixed signal matrix (stored in Flash) in SRAM. Therefore, in order to calculate the covariance matrix, the 32000 x 2 mixed signal matrix (2 signals sampled at 16000 samples per second for two seconds) is split up into 32 1000 x 2 matrices. This can be done, since the end result is a 2 x 2 covariance matrix, where each entry is a large sum created by multiplying a 2x32000 matrix by a 32000x2 matrix. One at a time, each chunk is loaded into SRAM, and a partial covariance matrix is computed. This is done by subtracting the mean from the chunk using arm_mat_sub_f32(), taking the transpose of the result using arm_mat_trans_f32(), multiplying the transposed matrix by the non transposed matrix using arm_mat_mult_f32(), and adding the resulting matrix to a partial sum matrix named the covarianceMatrix. After this process is done for all 32 chunks, the covarianceMatrix is divided by the number of audio samples minus 1 (31999) to obtain the final covarianceMatrix.

C. Eigenvalues and Eigenvectors

The next step requires solving a quadratic equation in order to get the eigenvalues and eigenvectors of the covariance matrix which are needed to calculate the whitening matrix. First, each element of the A matrix is loaded with the elements of the covariance matrix buffer. Then the trace, determinant and the root is calculated. Finally, because of the quadratic property, we have two eighten values. One from $(tr+root)/2$ and the other $(tr-root)/2$ are computed and stored in a matrix called eigenValueMatrix. We use these eigenvalues to calculate the eigenvectors, which are stored in a variable called eigenVectorMatrix. Each column of the eigenVectorMatrix is then normalized by their euclidean norms.

D. Whitening Matrix

The whitening matrix is calculated after finding the eigenvalues. This is first done by taking the square root of the eigenvalues stored in the eigValueMatrixBuffer[] array using arm_sqrt_f32(). The zeroth element of the eigValueMatrixBuffer[] array is the square root of the first eigenvalue, while the 3rd is the square root of the second eigenvalue. This is because the matrix is stored row by row in the buffer, where the first element is the top left entry, and the 3rd element is the bottom right entry. The resulting matrix is then inversed using arm_mat_inverse_f32(), and multiplied by the transpose of the eigenvector matrix. This gives us the final whitening matrix, as described by figure 2.

$$whitening\ matrix = \begin{bmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{bmatrix}^{-1} \times E^T$$

Fig. 2. Whitening matrix calculation, where λ_1 and λ_2 are the eigenvalues of the variance-covariance matrix.

E. Retrieving Original signals

After the whitening matrix has been calculated, the final step of the ICA algorithm begins. This final step involves using the whitening matrix along with a newly defined weight matrix to calculate the final ICA matrix that will be used to retrieve the original sine waves. The first step is to initialize the weight, which we will denote as w (lower case).

The weightMatrix matrix is determined in an iterative manner until the weights have converged or until there have been 1000 iterations. Convergence is checked by comparing the previous weight to the new weight, and checking if the difference is smaller than 0.00000001.

In each iteration, the mixed signals are read chunk by chunk, 8000 bytes at a time. Each chunk is first centered by subtracting the means from each mixed signal. The whitening matrix is then multiplied with this chunk to create a whitened chunk. With this whitened chunk, call it W, a partial weight is then calculated by computing $W \times (W^T w)^3$

The partial weights for each chunk are then summed together, and divided by 32000. Three times the old weight is then subtracted from the result to yield the total un-normalized weight for that iteration. Finally, the weight is normalized by its Euclidean norm. This entire process is repeated until the weight matrix converges.

After the converged weight matrix has been computed, a basis set is created using the coefficients of this matrix. If the weight matrix is:

$$\begin{bmatrix} a \\ b \end{bmatrix}$$

then the basis set is:

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

because

$$\begin{bmatrix} -b \\ a \end{bmatrix}$$

is a line perpendicular to the weight. The ICA filter matrix is then calculated as the product of the basis set transpose and the whitenening matrix. In the HAL_DAC_ConvCpltCallbackCh1 callback function, the ICA filter matrix is used to convert the mixed signals back into their original sine waves.

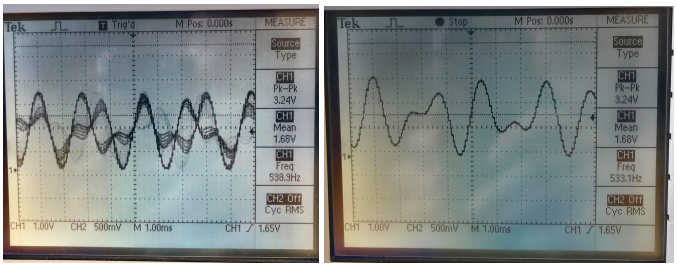

Fig. 3. Two figures showing the mixed sine wave generated by the mixing matrix described in equation 1 using sine waves of $\sin(2\pi*400t)$ and $\sin(2\pi*700t)$ respectively. The first screenshot is a screenshot of the signal while it is playing mixed samples, the second screenshot is the stilled waveform at a particular instant while the oscilloscope has stopped capturing.

Fig. 4. Two figures show the sine waves after they have been separated from the mixed signals by the ICA algorithm. The original equation of each wave was $\sin(2\pi * 400t)$ (left) and $\sin(2\pi * 700t)$ (right).

VI. Results

There were primarily two methods used to evaluate the waves outputted by the DAC. First and foremost, an oscilloscope was used to probe the GPIO pins that the DAC uses to output the waves. This method allowed the observation of the waveform shapes, the exact frequency of the wave, and the amplitude as measured by the peak-to-peak voltage. The quality of the wave could also be quantitatively measured by counting the number of samples that were taken per period of the wave. Another method used to evaluate the results were to connect speakers to the GPIO pins. This allowed the audio quality of the wave to be measured qualitatively.

The waves measured using the oscilloscope in figure 3 demonstrate the output of the mixed signals using the following mixing matrix:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{1}$$

The first sine wave shown in figure 3 is a bit messy and the frequency that was shown was constantly fluctuating. Firstly, we should note that the actual period of the resulting mixed sine wave is $\frac{1}{153560}$, since 153560 is the lowest common multiple of 440 and 698. The oscilloscope, however, thinks that the frequency is merely 533Hz, and doesn't seem to be capturing the full period. Since the oscilloscope is not capturing the full period of this periodic function, and only a local window, it appears as if the local window is constantly fluctuating. This makes sense since the full period of this waveform might be too large to see on the screen of the oscilloscope, and probably exceeds the limitations of what this oscilloscope is capable of capturing. We should also point out that since we are only taking 16,000 samples per second, frequencies above 8000Hz experience aliasing, since this goes beyond the maximum frequency we can produce according to the Nyquist Sampling Theorem with a 16,000 sample rate.

The sine waves in figure 4 demonstrate the output of the ICA-separated mixed signals.

On the figure on the right, a slight discontinuity in phase can be observed inside the red box drawn. This phenomenon is caused due to the buffer size of the DAC not being a multiple of the number of samples per sine wave period. The 700Hz sample is being output at 16000 samples per second, which means that the 700Hz signal will only have 22 full periods every second. After 22 periods, $22 \times 700 = 15400$ samples will have been output, and the last 600 samples will be output without completing a full period (700 samples). Since the DAC's DMA is in circular mode, it replays from the beginning of the buffer which is discontinuous from the value at the end of the buffer. On the left figure, this doesn't happened since 16000 fits the 400hz signal perfectly into 40 periods per second.

VII. Conclusion

Two sine signals were generated using the CMSIS DSP library, mixed and stored in flash. ICA proved challenging because the matrix operations that needed to be performed for ICA required us to understand which operations could be split up into chunks. Ultimately, we were able to reproduce the original signals successfully.

VIII. Post-Mortem

When we began, we used DMA for the QSPI transfer. However, when we ran into bugs, we thought it may have been because of the DMA. As a result, we ended up taking out the DMA implementation for QSPI. If we were to restart the project over, we would avoid using DMA until we have the basic functionality working.

We had a lot of bugs related to the matrix structs not being of the correct dimensions or not having the correct buffer size which lead to hard faults. This could have been avoided if we did error checking.

Although we wanted to save RAM by reusing as many structs and buffers as we could, in hindsight this made the code very hard to debug. We should have used as many structs and buffers as possible to keep the code clean until we had it working, and then optimize for memory afterwards.