Chapter 7

Digital Signal Processor

In this chapter, we firstly introduce the hardware architecture and the data type characteristics of the DSP board. Secondly, program execution efficiency optimization guidelines are presented. Subsequently, choices and considerations about the C program implemented on the DSP board are recorded in the third section. Finally, the improvements on previous work are summarized.

7.1 ADSP-BF548 EZ-KIT Lite

This section primarily summarizes the hardware-related information that helps developers write more robust and efficient C code.

7.1.1 Hardware Architecture

Audio Codec in charge of sampling speech and **Memory Hierarchy** that restricts the storage of variables are highly significant to the implementation on the DSP board.

Audio Codec

An Analog Devices AD1980 audio codec is the audio interface of the EZ-KIT Lite. The codec connects to multiple audio connectors (3.5 mm) which allow us to get audio in and out. These connectors can be easily found at the bottom left corner of the board.

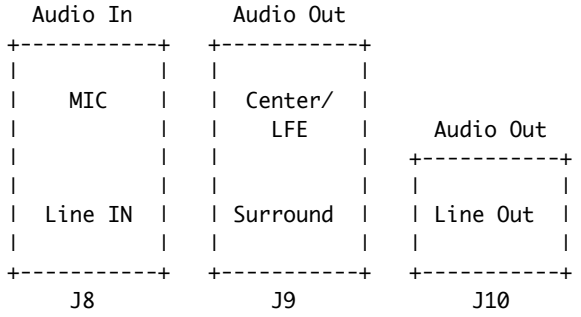

Figure 7.1: Audio Connectors

Fig. 7.1 illustrates that the top location of `J8` is for a stereo microphone and the bottom location is for a stereo line in. According to Fig. 7.2, the difference between `MIC` and `LINE IN` is signals via `MIC` will be pre-amplified. The preamp gain is collaboratively controlled by `MIC` Volume Register (`AD1980_REG_MIC_VOL_CTRL`) and Miscellaneous Control Bit Register (`AD1980_REG_MIC_VOL_CTRL`) [26]. In addition, it can be clearly seen that AD1980 can only sample 2 channels at any given moment due to the RECORD SELECTOR multiplexer. Thus, we decide to input noisy command and noise via the left / right channel of `MIC` respectively.

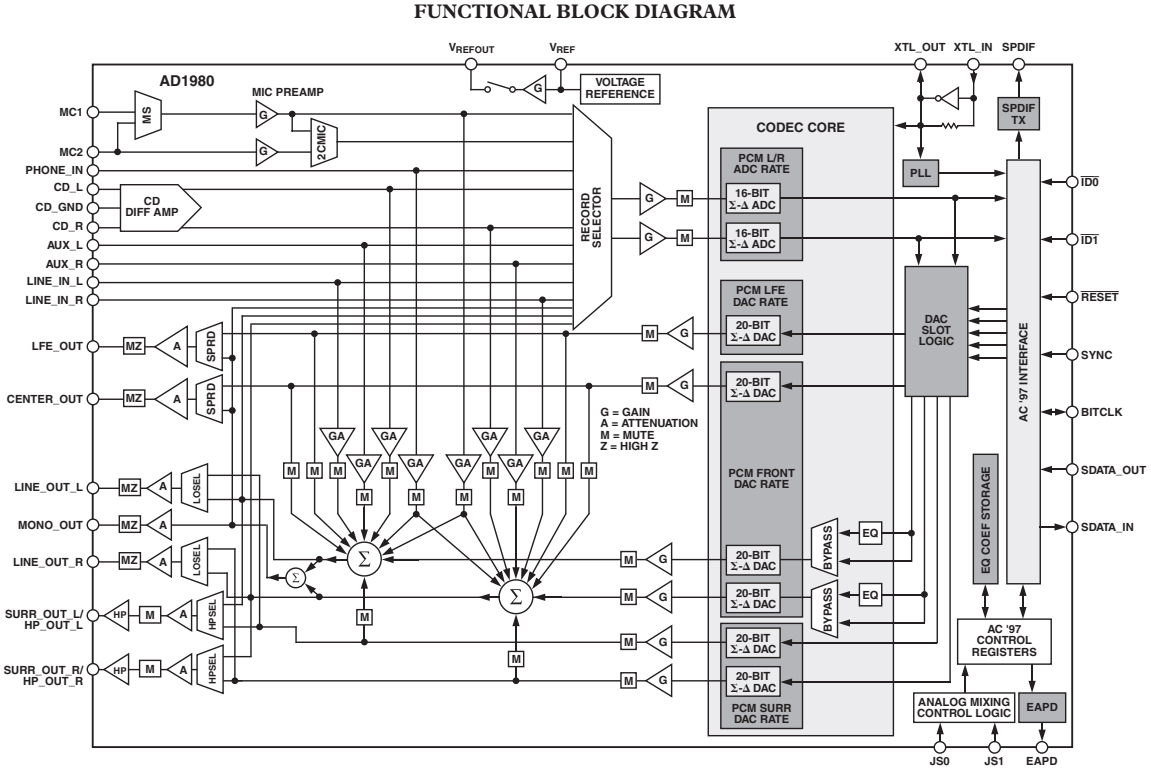

Figure 7.2: AD1980 Functional Block Diagram

Memory Hierarchy

The ADSP-BF548 processor supports a hierarchy of three synchronous memories.

Internal L1 memory offers two 32 KB SRAM banks for data [27]. "L1 memory is the highest-performing memory available to the Blackfin core and can be accessed at core clock speeds."[28]

Internal L2 memory consists of a single 128 KB area of SRAM. "L2 is somewhat lower-performing than L1, requiring two core clock cycles for access." L2 provides larger capacity accompanied by higher latency. [28] Totally, 32768 variables of 32-bit data type (e.g. `float` and `fract32`) can be stored in L2.

External memory is a 64 MB DDR SDRAM that exists external to the processor mounted on the DSP board. "External memory operates synchronously with the processor's system clock rather than the core clock, causing access time to SDRAM to be relatively slower than to L1 or L2 memory."[28] 64 MB (67108864 bytes) is enormous given that each `float` / `fract32` variable occupies 4 bytes. Hence, we will pre-compute reusable coefficients as long as fetching them from external memory is faster than computing them. In fact, L1 & L2 are big enough to store these coefficients, decisions will be elaborated in section 7.3 *C Program*.

7.1.2 Data Types

Directly Supported Data Types

Table A.2 (in appendix on page 98) and Table 7.2 summarize the ten scalar data types directly supported by the complier.

Table 7.1 as a subset of Table A.2 lists main fixed-point data types. Note that `long` is equivalent to `int`. We use 8-bit `unsigned char` when manipulating GPIO (*General Purpose I/O*) registers.

Table 7.1: Main Fixed-Point Data Types

Type	Size	Min	Max
`unsigned char`	8-bit	0	255
`short`	16-bit	-32,768	32,767
`int`	32-bit	-2,147,483,648	2,147,483,647
`long`	32-bit	-2,147,483,648	2,147,483,647

Table 7.2 summarizes properties of floating-point data types. Note that `double` is equivalent to `float`. Value ranges are computed based on the internal representation of single-precision floating-point numbers. Details are provided in appendix on page 98. We have also shown `float` has a precision of $\log_{10}(2^{24}) \approx 7.225$ decimal digits in the appendix. This deduction is significant to exporting model and coefficients from MATLAB.

Table 7.2: Floating-Point Data Types

Type	Size	Range
`float`	32-bit	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
`double`	32-bit	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$

The smallest denormalized number has a marked impact on small probability calculation. For instance, the emission probability can easily become smaller than $e^{-200} = 1.3839 \times 10^{-87}$. Fortunately, all computation involved between probabilities are multiplications, hence we take natural logarithm before conducting multiplications and convert all multiplications into additions based on the property of logarithm (7.1).

$$\ln(xy) = \ln(x) + \ln(y) \tag{7.1}$$

Fractional Data Types

Fractional data types `fract16` and `fract32` can be represented as `short` and `int` respectively. Fig. 7.3 shows the internal representation of `fract16` and `fract32` (from National Instruments). Table 7.3 lists the main properties of fractional data types. `fract32` has higher precision (assuming no overflow or underflow) and higher processing speed (due to fixed-point

arithmetic) than `float`. Hence, `fract32` variables are widely utilized when implementing the algorithms introduced in chapter 3 *Feature Extraction*.

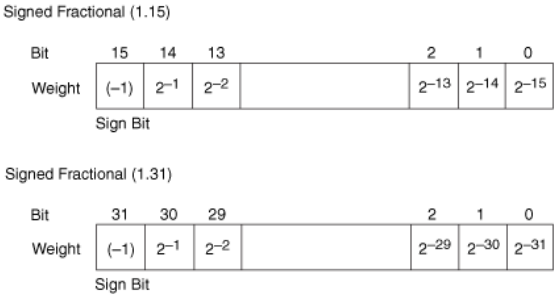

Figure 7.3: Internal Representation of Fractional Data Types

Table 7.3: Fractional Data Types

Type	Size	Range	Resolution
`fract16`	16-bit	-1 to $1 - 2^{-15}$	2^{-15}
`fract32`	32-bit	-1 to $1 - 2^{-31}$	2^{-31}

7.1.3 CrossCore Embedded Studio

During ELEN90058 *Signal Processing* and ELEN90052 *Advanced Signal Processing* workshop sessions, we used CrossCore® Embedded Studio to compile and debug code for Blackfin DSP board. Basing on Eclipse, CCES exploits the ecosystem of third-party tools already available to Eclipse developers. Thus, plenty of convenient functions such as plotting arrays of data are available to users [29][30]. CCES could be an ideal substitution for VisualDSP++.

However, the AD1980 Audio Codec on ADSP-BF548 EZ-Kit Lite Board is not supported with CCES [31]. Analog Devices engineer Craig Gilchrist explained in *EngineerZone* support community that they do not have a driver for the AD1980 Codec with CCES. In addition, he recommended AD1836 Audio Codec instead [32].

Considering the project budget and potential risk, we eventually decided to stick on VisualDSP++ 5.1.2.

7.2 Execution Efficiency Optimization

In addition to the experience gained in ELEN90058 *Signal Processing* workshop sessions, [33] provides official guidance for optimizing C program execution efficiency.

7.2.1 Avoid Integer Division in Loops

Fixed-point arithmetic operations are natively supported by hardware, hence computation on `int` (including `fract32`) variables are executed at high speed. Nevertheless, the hardware does not directly support **integer division**. Division and modulus operations on `int` variables are multicycle operations, this will prevent the optimizer from using a hardware loop for any loops around the division [33]. Hence, it is recommended to avoid divide or modulus operators inside a loop whenever possible.

7.2.2 Avoid Float Arithmetic

Floating-point arithmetic operations are implemented using software emulation and consequently far slower than integer operations. An arithmetic floating-point operation inside a loop will prevent the optimizer from using a hardware loop [33]. As a result, floating-point operations inside a loop should be avoided whenever possible.

7.2.3 Statistical Profiling

VisualDSP++ provides an excellent tool called *Statistical Profiling* helping developers find the bottleneck of the execution efficiency. This tool can be found in the `Tools` → `Statistical Profiling` → `New Profiling` menu.

As is shown in Fig. 7.4, `pre_emphasis()` function and `hamming()` function consume respectively 18.18% and 5.65% of the total processing time to recognize a speech command.

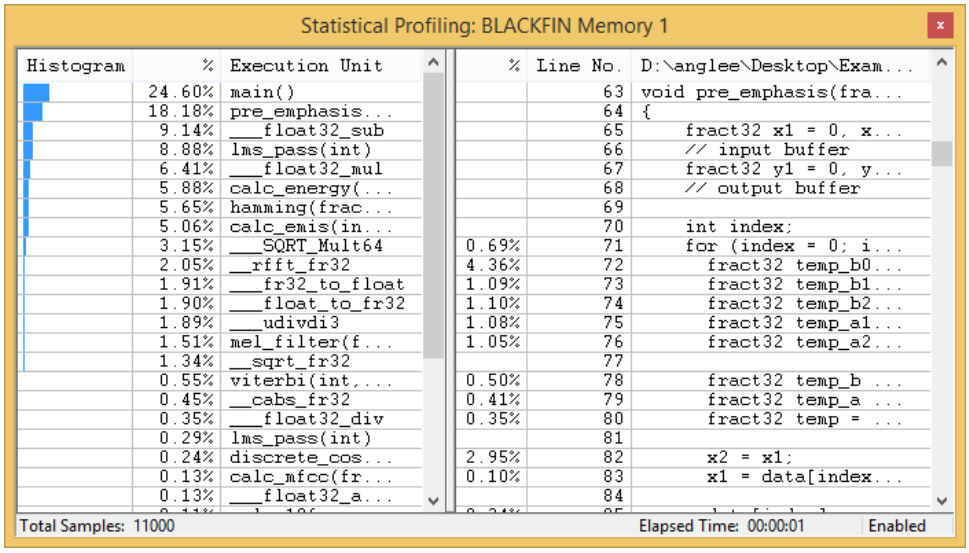

Figure 7.4: Statistical Profiling

7.2.4 The `time.h` Library

The `time.h` header defines several variable types, macros and various functions for manipulating date and time. The variable type `clock_t` is suitable for storing the processor time. The function `clock_t clock(void)` returns the processor clock time used since the beginning of the program. With the aid of the macro `CLOCKS_PER_SEC`, processor clock time can be converted into time in millisecond. In terms of ADSP-BF548, `CLOCKS_PER_SEC` $= 600 \times 10^6$ clocks per second.

```
1 clock_t clocks = clock();
2
3 /* Some operations here */
4
5 clocks = clock() - clocks;
6 float time_elapsed = (float) clocks * 1000 / CLOCKS_PER_SEC;
7
8 printf("Clocks: %d\n", clocks);
9 printf("Time elapsed: %f ms\n", time_elapsed);
```

Preceding code snippet provides the paradigm about measuring code execution time. The `clock()` function in 1st line gets the current timestamp before certain operations. Then, some actions are executed. The `clock()` function in 5th line obtains the timestamp again and calculate the difference between two timestamps. The 6th line converts the time in clocks into

time in millisecond.

7.3 C Program

This section introduces how to conduct real-time processing on the DSP board and the considerations during implementation of feature extraction algorithms.

7.3.1 Audio_Loopback

Analog Devices provides an example named `Audio_Loopback` which simply copies the sampled and quantized audio input to the audio output in real-time. This example is available in the `/VisualDSP 5.1.2/Blackfin/ADSP-BF548 EZ-KIT Lite/Drivers/AudioCodec` folder under VisualDSP++ installation directory. This framework initializes the codec AD1980 via the device manager and will invoke an interrupt callback function when the inbound buffer is full.

This example project helps us spend more time on our main quest instead of configuring hardware environment from scratch. Original `.h` and `.c` files are added to the project. Some codes are added to `main` and `AD1980Callback` function to initialize the recognition system and associate the functions properly.

1. `header.h` defines all constants and global variables.
2. `function.c` includes all original functions.
3. `model.h` contains speech models exported from MATLAB.
4. `mel_filter.h` incorporates precomputed Mel-filter bank gain.

7.3.2 Buffering

We set the length of segment buffer as 6000 data points, i.e. interrupt callback function is called every 6000/16000 = 0.375 seconds. Time-consuming LMS based filter works every interrupt and noise-reduced command is cached in a buffer of 48000 data points. Speech signal processing and recondition routine run once when the eighth interrupt comes.

7.3.3 Pre-emphasis

It is easy to prove that filter characteristic represented by (3.3) will not change if coefficients a_i and b_i are uniformly scaled. Some pre-emphasis filter coefficients (3.4) are beyond the fractional data types domain $[-1, 1]$. Hence, the coefficients are scaled by λ before converting them from `float` to `fract32`.

We choose $\lambda = \frac{1}{4}$ primarily because $\frac{1}{a_0} = \frac{1}{\lambda} = 4$ which is a power of 2. In terms of fixed-point data types, left shifting 2 bits is equivalent to multiplying by $\frac{1}{a_0} = 4$ but left shifting is more efficient. Besides, $\lambda = \frac{1}{2}$ is not small enough while $\lambda = \frac{1}{8}$ leads to loss in precision. $a_i \in [-1, 1]$ and $b_i \in [-1, 1]$ for $\forall i = 0, 1, 2$ when $\lambda = \frac{1}{4}$.

$$\begin{cases} a_0 = \frac{1}{4} = 0.25 \\ a_1 = \frac{-1.523796}{4} = -0.380949 \\ a_2 = \frac{0.649345}{4} = 0.162336 \end{cases} \qquad \begin{cases} b_0 = \frac{1.861856}{4} = 0.465464 \\ b_1 = \frac{-3.102851}{4} = -0.775712 \\ b_2 = \frac{1.366544}{4} = 0.341636 \end{cases}$$

Filter coefficients are converted into `fract32` once by `calc_shelving_coef()` function when initializing the system. Finally, all computations in `pre_emphasis()` function are fixed-point arithmetic.

7.3.4 Hamming Window

For the window length $N = 512$, Hamming window weight $w[n]$ has 512 elements. Table 7.4 summarizes the efficiency of different methods to obtain $w[n]$. $512/2 = 256$ (due to symmetry) `fract32` data points occupy 1 KB. There is no reason to recompute them every time. Considering they are most frequently used (180 times per 3-second recording), we store them in the fastest L1 memory. During the initialization session, `calc_hamming_coef` function precomputes them and converts them into `fract32`. All computations in `hamming()` function are fixed-point arithmetic.

Table 7.4: Efficiency of Approaches to Obtain $w[n]$

Action	clocks	time elapsed
compute	607471	1.012452 ms
fetch from L1	8731	0.014552 ms
fetch from L2	12827	0.021378 ms
fetch from external memory	30878	0.051463 ms

7.3.5 Thresholds

Frame Energy

The energy of frame j is defined in (3.8) on page 22. We have known the amplitude of a data point $-1 \leq s_j[n] \leq 1$, thus the power of this point $(s_j[n])^2 \leq 1$. The energy of a frame is the sum of N power points. We invoke intrinsic function `add_fr1x32()` when performing addition of 32-bit fractional data. Although the energy is possibly larger than 1, because intrinsic functions support saturation arithmetic that prevents overflow [34]. That is, the addition function will return 1 if the original outcome exceeds 1. Given the threshold set by us is smaller than 1, as long as the variable becomes saturated, the frame energy will be definitely higher than the threshold. Hence, saturation will not influence the threshold decision.

7.3.6 MFCC extraction

Power Spectrum

We have known that the DFT of a N-point real sequence can be obtained from one $\frac{N}{2}$-point complex-valued DFT and additional computations (from ELEN90058 *Signal Processing* lecture slides by Erik Weyer). VisualDSP++ has a built-in function called `rfft_fr32()` to exploit this property.

`rfft_fr32()` function takes 7 arguments. The first two are time domain real sequence input and spectrum output. The third one is the twiddle table calculated once by `twidfftrad2_fr32()` function during system initialization. The fourth parameter is the twiddle stride that should be set to 1 because the twiddle table is tailored to this FFT. The fifth argument is the FFT size

which equals the window length $N = 512$. The last two are block exponent and scaling method. We select dynamic scaling mode that will inspect intermediate results and only apply scaling where required to prevent overflow. In this mode, precision can be better preserved than in static scaling method, although inspections leads to an negative impact on performance. The block exponent returned will be the number of times that the function scales the intermediate set of results.

Once the spectrum is obtained, we invoke `cabs_fr32()` function to compute the magnitude of each data point (data type: `complex_fract32`), the product of the magnitude and itself is the power of each point.

Bank Filtering & Log Scaling

Fig. 7.5 depicts the process to compute $X_j[m]$ of bank $m = 9$ based on (3.15) on page 26. In subplot 1, each power data point $\hat{S}_j[k]$ (gold circle $\circ$) is multiplied by the corresponding gain $H_{mel}[m,k]|_{m=9}$ (orange asterisk $*$). Filtered power data points are shown in subplot 2 by orange cross $\times$. Thereby, the sum of all filtered power data points is the energy $X_j[m]$ within filter bank $m = 9$.

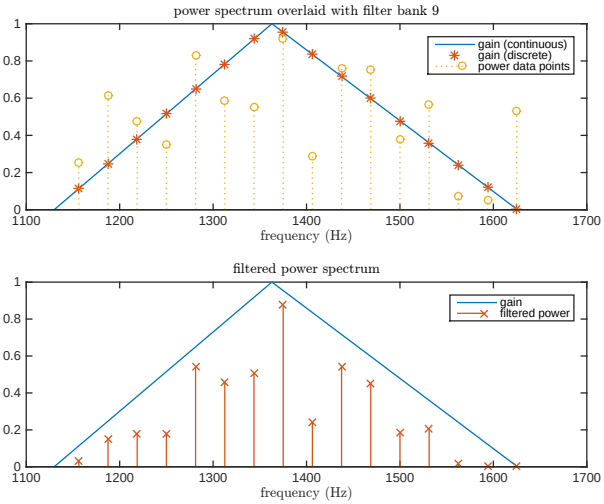

Figure 7.5: Bank Filtering Demonstration

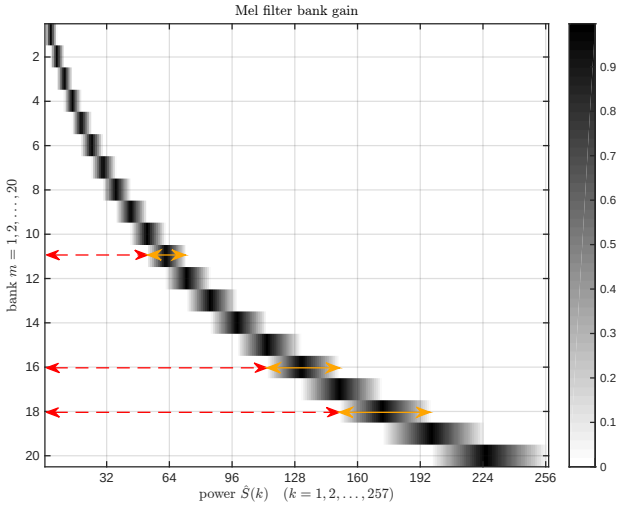

Figure 7.6: Gain of Each Point in Each Bank

Fig. 7.5 also shows that bank $m = 9$ only intersects with 16 data points in power spectrum because gain for other points are all zero. We plot the gain of each data point for each bank in Fig. 7.6, grayscale represents the value of gain. We find the same situation for all banks.

Directly computing $H_{mel}[m,k]$ results in a $M \times \left(\frac{N}{2}+1\right)$ sparse matrix (white blocks stand for zeros).

In order to diminish the processing time by pre-computing and eliminating the useless operations with zeros, we design a special storage structure for this sparse matrix. One array for storing non-zero values in linear shape and one array for storing corresponding offset for each bank. One array for storing at which k gain becomes non-zero (red dashed-line arrows) and one-array for storing the length of each grayscale block (orange solid-line arrows).

Take the following matrix as an example

$$\begin{bmatrix} 0 & 0.1 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.8 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.7 & 0.9 & 0.6 \end{bmatrix}$$

The first array stores the non-zero values.

$$\begin{bmatrix} 0.1 & 0.2 & 0.5 & 0.8 & 0.3 & 0.5 & 0.7 & 0.9 & 0.6 \end{bmatrix}$$

The second array stores the offset of the first element of each row.

$$\begin{bmatrix} 0 & 2 & 5 \end{bmatrix}$$

The third array stores the number of zeros before the first non-zero element of each row (red dashed-line arrows in Fig. 7.6).

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

The forth array stores the length of non-zero sequence of each row (orange solid-line arrows in Fig. 7.6).

$$\begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$$

Gains are converted into `fract32` by `calc_bank_gain()` function during system initialization. Power cepstrum data points are converted back to `float` in case of saturation before summation that evaluates the total power within each bank. We no longer insist on fixed-point arithmetic because

1. only 20 - 50 frames (out of 180 frames) are passed to MFCC;

2. precision error will be accumulated;

3. log scaling in the next step will be done with the aid of `float` type `log()` function;

4. outcome of logarithm ranges widely.

Discrete Cosine Transform

We define

$$H_{dct}[n,m] = \sqrt{\frac{2}{M}} \cos\Big(\frac{\pi}{M}(m-0.5)(n-1)\Big) \tag{7.2}$$

where $n = 1, 2, \ldots, F$ and $m = 1, 2, \ldots, M$.

Thereby, (3.20) becomes

$$\hat{C}_j[n] = \sum_{m=1}^{M} \hat{X}_j[m] H_{dct}[n,m] \quad n = 1, 2, \ldots, F \tag{7.3}$$

It can be proven that for even M and odd F

$$H_{dct}[n,m] = \begin{cases} H_{dct}[n, M-m+1] & n = 1, 3, \ldots, F \\ -H_{dct}[n, M-m+1] & n = 2, 4, \ldots, F-1 \end{cases} \tag{7.4}$$

By utilizing this symmetry, (7.3) can be simplified as

$$\hat{C}_j[n] = \begin{cases} \sum_{m=1}^{M/2} (\hat{X}_j[m] + \hat{X}_j[M-m+1]) H_{dct}[n,m] & n = 1, 3, \ldots, F \\ \sum_{m=1}^{M/2} (\hat{X}_j[m] - \hat{X}_j[M-m+1]) H_{dct}[n,m] & n = 2, 4, \ldots, F-1 \end{cases} \tag{7.5}$$

To calculate $\hat{C}_j[n]$ for each n, (7.3) requires M multiplications and $(M-1)$ additions while (7.5) requires $M/2$ multiplications (reduced by 50%) and $(M-1)$ additions. Furthermore, only a half precomputed coefficients need to be stored in memory.

We pre-compute $H_{dct}[n,m]$ by `calc_dct_coef()` while initializing the system. Table 7.5 summarizes the efficiency of different methods to obtain $H_{dct}[n,m]$. It can be clearly seen that fetching from external memory is even 40 times faster than directly computing.

Table 7.5: Efficiency of Approaches to Obtain $H_{dct}[n, m]$

Action	clocks	time elapsed
compute	172281	0.287135 ms
fetch from L1	3889	0.006482 ms
fetch from L2	4929	0.008215 ms
fetch from external memory	9367	0.015612 ms

7.4 Improvements on previous work

Coefficients that intensively require floating-point computations are precomputed.

Table 7.6: Reusable Coefficients

	variable	equation
Hamming window	$w[n]$	(3.7) on page 20
Mel-bank filter gain	$H_{mel}[m, k]$	(3.16) on page 26
discrete Fourier transform	$H_{dct}[n, m]$	(7.3) on page 86

Take Hamming window as an example, the processing time for each frame is reduced by 0.9979 ms, consequently 179.6 ms are saved for 180 frames. Note that the average total processing time to recognize a command is 95.8 ms (will be further discussed in chapter 8 *Final Implementation*). Without pre-computation of Hamming window weights, the total processing time will increase by 187%.

Besides, frame energy is calculated in fixed-point arithmetic (subsection 7.3.5 *Thresholds*). As is shown in the following table, the total processing time for 180 frames is reduced by 8.173 ms.

Table 7.7: Compute Frame Energy in `fract32` and `float`

	clocks	total processing time (ms)
`fract32`	50325648	83.876
`float`	55229936	92.050
difference	4904288	8.173

Moreover, special data structures are designed for sparse matrices, such as Mel-bank filter gain shown in Fig. 7.6 on page 84 and transition matrices shown in Fig. 5.5 on page 51. As a result, meaningless multiplications with zeros are effectively illuminated.

Symmetry is also adequately taken into consideration. For example, when computing the power spectrum of a real sequence, only the first half of spectrum are calculated because of the conjugate symmetric characteristics.

Table 7.8: Optimization based on FFT Symmetry

	clocks	total processing time (ms)
with optimization	50325648	83.876
without optimization	54965952	91.610
difference	4640304	7.733

The dataset tested has 23 informative frames. Usually, 20-50 frames are passed into MFCC procedure, hence 6.725 to 16.813 ms can be saved by considering the conjugate symmetry of spectrum.