基于 Cortex-M3 和 MFCC+CNN 的实时离线语音关键词识别系统

设计文档

目录

1. 系统简介和演示

1.1 关键词识别系统简介

随着人工智能的迅猛发展和人机交互需求的不断增大，智能语音技术获得了前所未有突破。语音领域的研究成果，不仅推动了前沿科技的进步,更创造了巨大的市场价值，意义重大。语音识别技术在智能终端、车载系统、智能家居等场景应用越来越广泛。语音识别可以分为在线识别和离线识别，而在一些低端消费产品上，离线识别依然是主流。

离线语音关键词识别可以应用在智能音箱、智能灯具、智能窗帘等等一系列应用上，市场潜力巨大。本设计基于这样一个背景来进行实时离线语音关键词识别系统的开发和设计。

1.2 系统简介

我们实时离线语音关键词识别系统在算法上基于 MFCC 和 CNN 算法，这两个算法都是比较成熟的算法，无论是对实际应用还是参加比赛来说风险都是比较小的。同时算法在整个系统中的运算精度为单精度浮点，保证了算法移植到系统上时运行的精度以及移植算法的简洁性。目前可以识别('five' 'four' 'go' 'left' 'one' 'right' 'stop' 'three' 'two' 'yes')这十个关键词，如果想要识别其他对应的关键词，只要找到对应的训练集重新训练网络就可以了。具体的算法内容可以参考第二章。

硬件系统基于 ARM Cortex-m3 处理器，并设计了一系列浮点加速单元，为了满足算法的实时性，具体硬件的实现可以参考第三章。硬件的开发语言为 Verilog，开发平台为 Vivado 2020.1。我们所采用的 FPGA 开发板为依元素科技公司的 EGO1 开发板，该开发板采用的 FPGA 芯片为 Xilinx Artix-7 系列的 XC7A35T 芯片。除了 FPGA 开发板外，我们只用到了一个语音放大模块 MAX4464，我们的系统成本极低，只需要一个语音传感器，我们的系统可以移植到任意的 FPGA 平台上，比较具有市场竞争力。MFCC 和 CNN 语音识别算法只是一个应用的例子，该系统有能力运行一些的其他信号处理的算法，适合多种嵌入式应用场景。

该系统的软件部分由 C 语言和汇编语言组成，开发环境为 keil IDE，具体软件和系统实现可以参考第四章。本系统示例程序的音频采集频率为 8Khz，每秒钟可以识别 4 个长度约为 1

秒(7808 点)的语音样本，当然这些参数都是可以通过软件的编写调整的。

1.3 系统测试与演示

图 1.1 展示了系统的硬件实物图。语音识别的结果通过串口输出，同时语音识别的结果也可以通过 GPIO 输出以达到控制外设的效果，比如控制灯的开关和电机的转动和停止等等其他的人机交互功能。

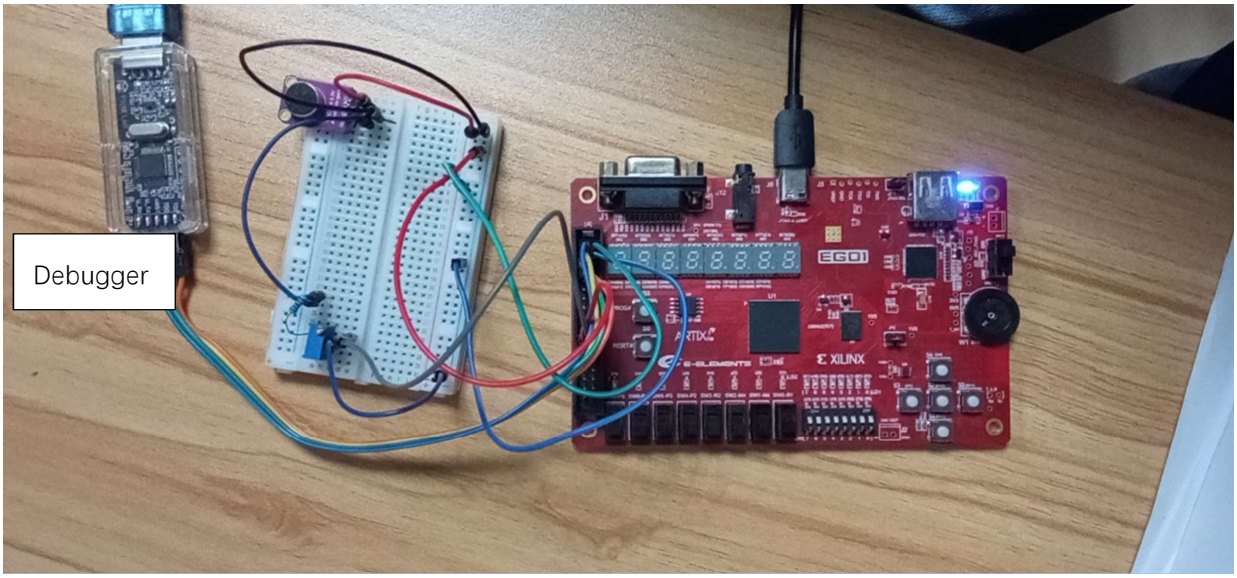

图 1.1 系统实物图

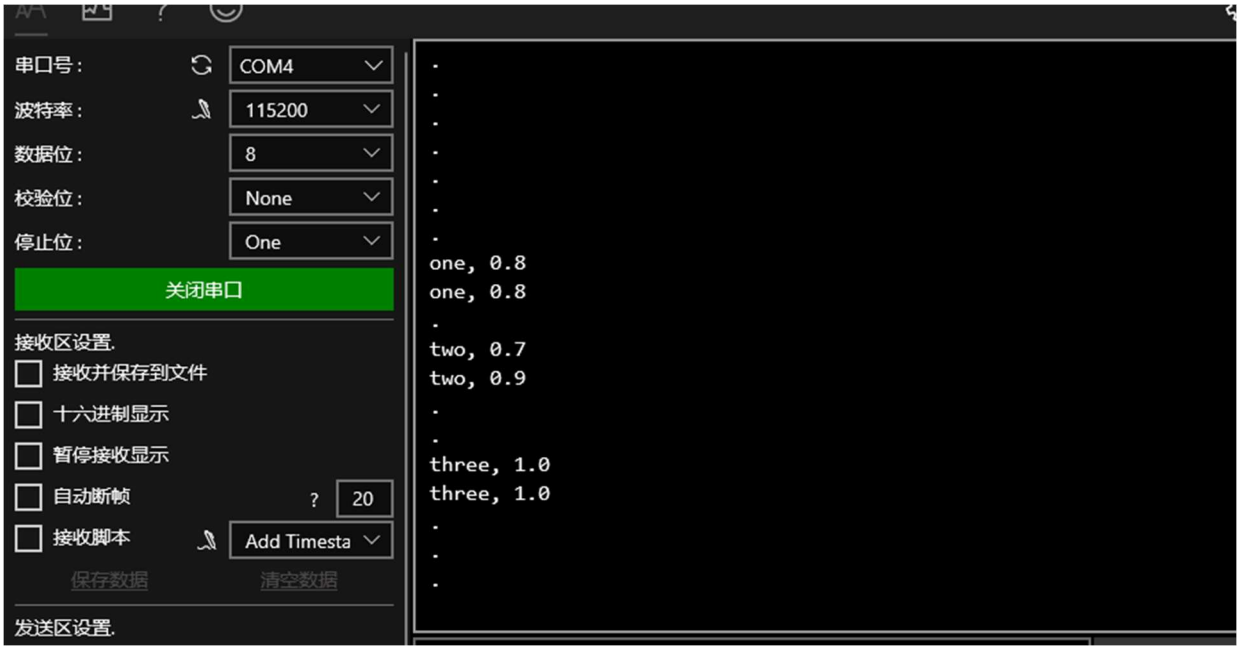

图 1.2 串口输出识别的关键词以及可信度

算法在 PC 机上训练和测试的准确率大约为 0.8。将算法移植到嵌入式系统后，经过作者亲身长时间的语音测试，每个关键词测试 20 次，得到的准确率大约为 0.8，因为算法在嵌入式系统中以单精度浮点数的精度运算(和 PC 机精度保持一致)，所以准确率和在 PC 机上测试时的准确率基本一致。通过一系列硬件浮点运算加速单元的加速，系统可以达到我们所要求的运算速度，进而满足系统的实时性。下面按顺序分别介绍了该系统的算法、硬件、系统的设计细节。

2. 算法

2.1 算法概述

一个典型的 KWS 系统包括特征提取和一个基于神经网络的分类器，如图 2.1 所示。首先，将输入的长度为L的离散语音信号以步长s分割为长度为l的帧，要求每一帧和之前一帧是有重叠的，这就要求$s < l$，一共分为$T = \frac{L-1}{s} + 1$ 帧。对于每一帧语音信号提取F个特征值，这样的话一个语音样本经过特征提取后就可以表示为一个$T \times F$的的矩阵，再将这个矩阵送入分类器进行分类。

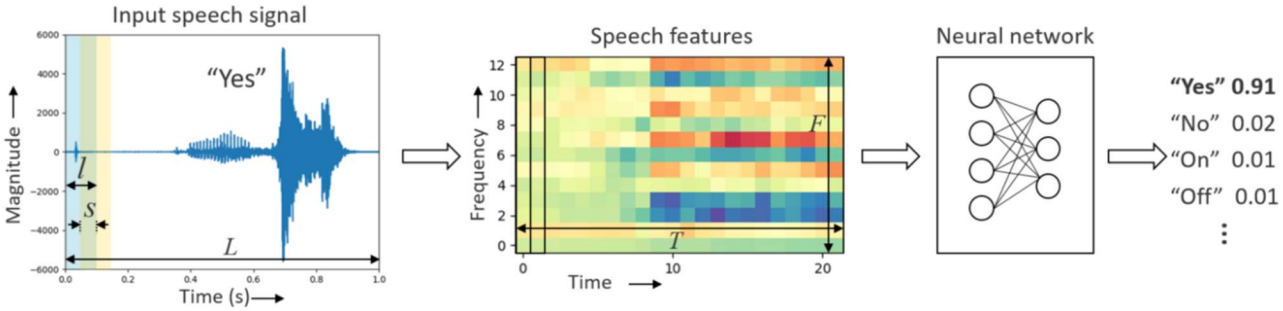

图 2.1 关键词识别系统流水线

这里我们对每一帧信号提取F个梅尔倒频谱系数(Mel-Frequency Cepstral Coefficients, MFCC)，分类器我们使用卷积神经网络(CNN)，下面两节详细介绍在该设计中这两个算法的具体细节。本次设计中我们选取的声音采样频率为 8Khz，每一个声音样本包含$L = 7808$个采样点，步长$s = 128, l = 256$ 。对于每一个长度为 256 的帧提取$F = 13$个 MFCC 系数，每个样本一共提取$T = 60$帧。下面两节详细介绍该设计中的 MFCC 算法和 CNN 算法。注意该设计在算法上，无论是在 PC 机上验证和训练，还是在嵌入式系统上实现，全部采用 32 位单精度浮点数计算。虽然在嵌入式系统上采用浮点运算显得有些愚蠢，但是通过设计我们发现，对于运算复杂度较小的算法，采用浮点运算来实现系统，灵活性很高，移植算法也很快。配合浮点加速运算单元，可以满足简单算法的实时性，采用浮点运算同时降低了开发时间和复杂度。

2.2 MFCC

在信号处理中，梅尔倒频谱（Mel-Frequency Cepstrum, MFC）是一个可用来代表短期音频的频谱，其原理基于用非线性的梅尔刻度（Mel scale）表示的对数频谱及其线性余弦转换（linear cosine transform）上。

MFCC 是一组用来创建梅尔倒频谱的关键系数。由音乐信号当中的片段，可以得到一组足以代表此音乐信号之倒频谱（Cepstrum），而梅尔倒频谱系数即是从这个倒频谱中推得的倒频谱（也就是频谱的频谱）。与一般的倒频谱不同，梅尔倒频谱最大的特色在于，梅尔倒频谱上的频带是均匀分布于梅尔刻度上的，也就是说，这样的频带相较于一般所看到、线性的倒频谱表示方法，和人类非线性的听觉系统更为接近。

MFCC 的提取过程没有严格的规定，一般 MFCC 的提取方式如图 2.2 所示。下面分别来介绍每个步骤的详细信息以及在该设计中的具体实现细节，MFCC 的提取是以每一帧为单位的，在该设计中每一帧的长度选取为 256 个点。

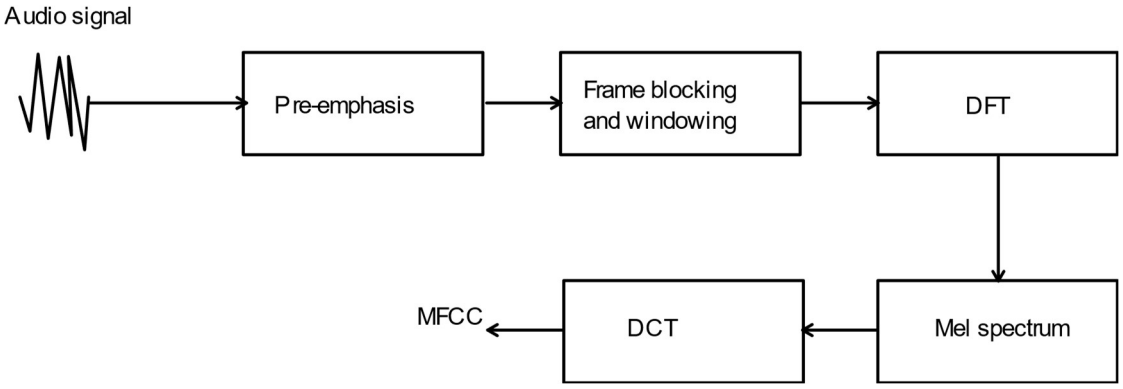

图 2.2 MFCC 提取过程

2.2.1 预加重 Pre-emphasis

预加重处理其实是将语音信号通过一个高通滤波器。预加重的目的是提升高频部分，使信号的频谱变得平坦，保持在低频到高频的整个频带中，能用同样的信噪比求频谱。同时，也是为了消除发生过程中声带和嘴唇的效应，来补偿语音信号受到发音系统所抑制的高频部分，也为了突出高频的共振峰。最常用的滤波器形式为：

$$H(z) = 1 - bz^{-1}$$

其中b的值通常选取区间为 0.4 到 1.0 之间。因为通过实验发现预加重后的特征对于神经网络的识别准确率并无太大帮助，所以该设计在算法上并**没有**对信号进行预加重处理。

2.2.2 分帧和加窗

首先要把语音信号分割成长度值固定的一帧帧数据，注意每帧数据之间是有重叠的，然后再对每一帧信号提取 MFCC。如果信号是实时采集的，就不需要把语音样本专门分成每一帧了，只需要采集够一帧($l = 256$)的数据直接提取 MFCC 参数就好了。因为帧与帧之间是重合的，所以在实时采集信号的情况下每采集到步长($s = 128$)个点，再和之前缓存的（$l - s$）点数据，就可以提取 MFCC 了，这样对内存的要求是很少的。

接下来要对每一帧信号进行加窗以增加帧左端和右端的连续性。加窗的选择有很多种，该设计采用汉宁窗(hanning window)，通过下面的公式得到汉宁窗的系数：

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{N}\right),$$
$$\text{for } n = 0, 1, 2, \ldots, N-1.$$

2.2.3 DFT

由于信号在时域上的变换通常很难看出信号的特性，所以通常将它转换为频域上的能量分布来观察，不同的能量分布，就能代表不同语音的特性。所以在乘上汉宁窗后，每帧还必须再经过离散傅里叶变换以得到在频谱上的能量分布。对分帧加窗后的各帧信号进行快速傅里叶变换得到各帧的频谱。并对语音信号的频谱取模平方得到语音信号的功率谱。该设计的 DFT 是和该下公式保持一致的：

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-j2\pi nk}{N}}; \qquad 0 \le k \le N-1$$

取模平方的运算在下一步。

2.2.4 梅尔谱 Mel spectrum

Mel spectrum 是通过将功率谱通过乘以一组带通滤波器(filter bank)来计算的，这一组带通滤波器叫做 Mel-filter bank。Mel 是一个基于人的听觉系统来衡量频率高低的单位，因为人的听觉系统对频率的敏感程度并不是线性的。它们之间的关系可以大致表示为

$$f_{Mel} = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

f表示真正的频率，f_{Mel}表示人的感知频率。

$X(k)$的 Mel spectrum 可以通过以下公示来求得。

$$s(m) = \sum_{k=0}^{N-1} \left[|X(k)|^2 H_m(k)\right]; \qquad 0 \le m \le M-1$$

注意公式中对$X(K)$进行了取模和平方的操作，通过与不同的$H(k)$进行点积得到不同的系数。这里的$N = \frac{l}{2} + 1$，因为输入信号是实信号，所以功率谱是对称的并且最高频率在中间取得，我们就取前$\frac{l}{2} + 1$点(对应本设计为 129 点)。该设计选择$M = 26$，这样一共可以得到 *26* 个系数，这样的话长度为$l = 256$的一帧就变为了长度为$M = 26$的数据。

系数$H(k)$是通过以下公式得到，m的取值范围为0到$M-1$。$f(m)$为每个滤波器的中心频率对应的 bin。

$$H_m(k) = \begin{cases} 0, & k < f(m-1) \\ \frac{2(k-f(m-1))}{f(m)-f(m-1)}, & f(m-1) \le k \le f(m) \\ \frac{2(f(m+1)-k)}{f(m+1)-f(m)}, & f(m) < k \le f(m+1) \\ 0, & k > f(m+1) \end{cases}$$

令$mel(f) = 2595\ log_{10}(1 + \frac{f}{700})$，该设计中

$$f(m) = int\left(\frac{mel^{-1}\left(mel\left(\frac{sr}{2}\right) \times \frac{m}{M}\right)}{sr/l}\right) \quad m = 0, 1, \ldots, M-1$$

sr为采样频率，sr/l为 DFT 频率的分辨率。画出 26 个滤波器的形状大致如图 2.3 所示。

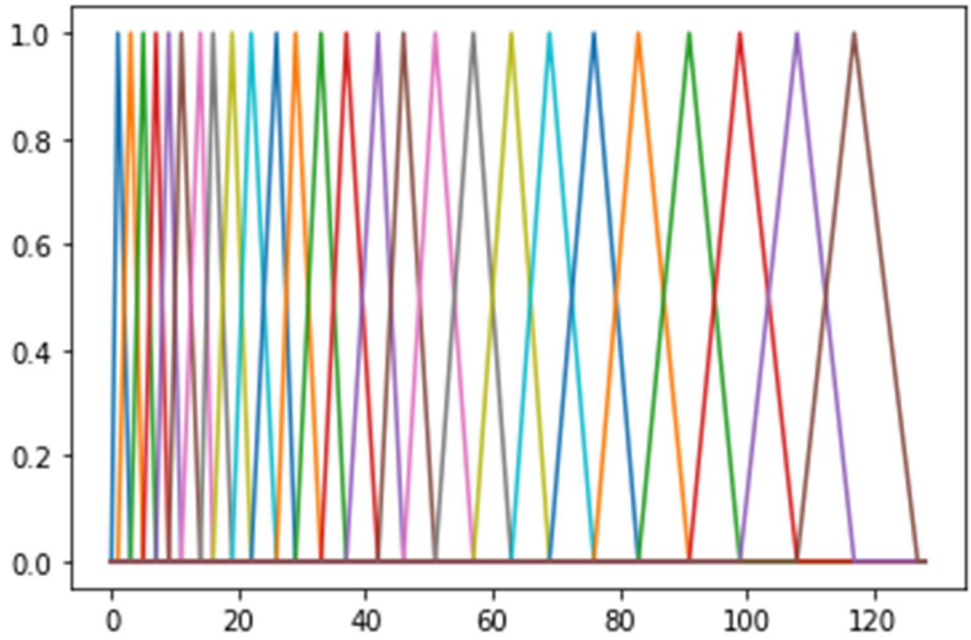

图 2.3 MFCC filter bank

2.2.5 DCT

最后我们对得到的$M = 26$个系数s取ln 并进行 DCT，具体实现和以下公式保持一致

$$y(k) = 2\sum_{m=0}^{M-1} ln(s(m) + 10^{-8})cos(\frac{\pi k(2m+1)}{2M}) \quad M = 26, \quad k = 0, \dots, K-1$$

该设计中$K = 13$，这样就一帧长度为$l = 256$的信号就可以提取出$F = 13$个 MFCCs。一个声音样本可以分割为T帧，这一个样本样特征提取后的数据形状就为$T \times F$的矩阵。图 2.5 是一个语音信号提取 MFCC 之后的例子。

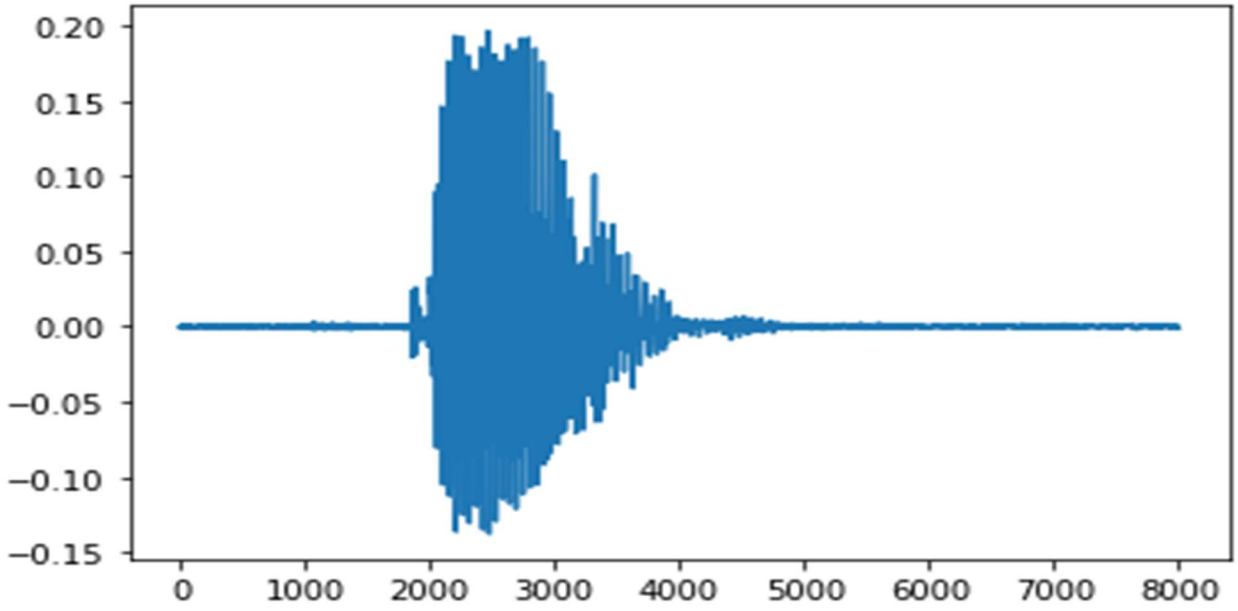

图 2.4 语音信号在时域的波形

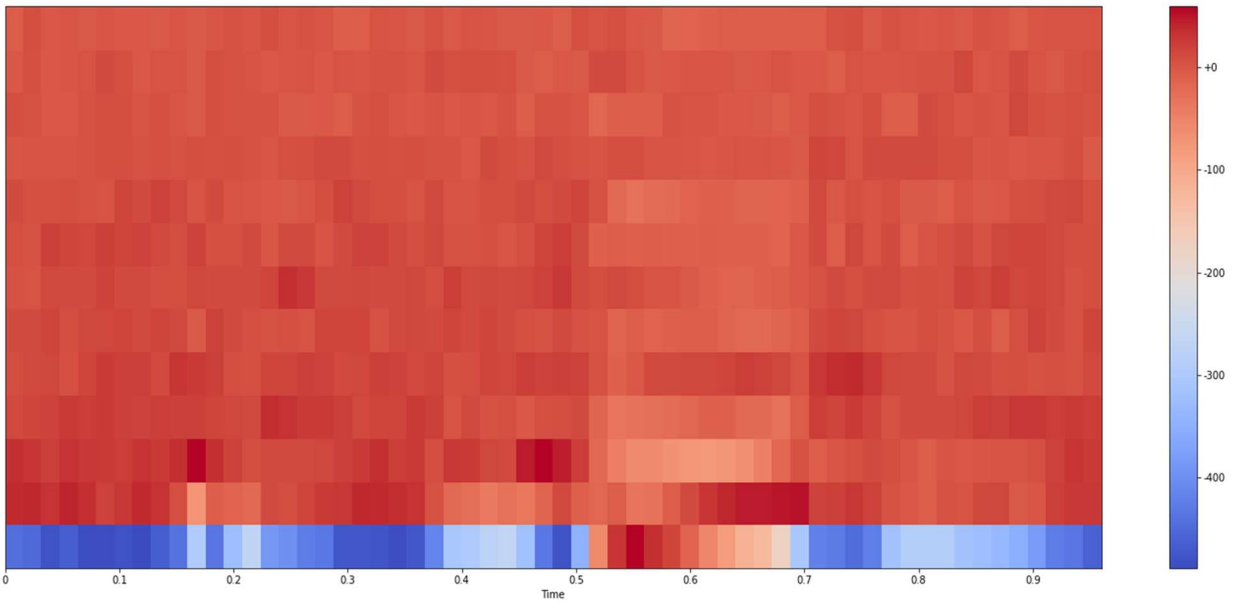

图 2.5 语音信号的 MFCC

2.3 CNN

一个长度为L语音信号经过经过 MFCC 特征提取之后可以得到一个大小为$T \times F$的矩阵，我们可以把这个矩阵看作为语音信号的 Feature map，这就是 CNN 的输入数据。下面给出 CNN 的实现细节。

2.3.1 网络框架和训练

CNN 的网络通过 Tensorflow 的框架来搭建和训练。由于我们最终要在嵌入式系统上来进行推测，而该设计又全部采用 32 位浮点数计算，由于嵌入式系统上内存和计算资源的限制，CNN 的网络结构必须保持在一个很小的规模。图 2.6 给出该设计 CNN 的框架，$input\_shape = T \times F$。

```
cnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=4, kernel_size=(3,3), activation="relu", input_shape=shape, padding="valid",
                           kernel_regularizer=tf.keras.regularizers.l1_l2(l1=1e-5, l2=1e-4)),

    tf.keras.layers.MaxPool2D(pool_size=(2,2),    strides=(2,2), padding="valid"),

    tf.keras.layers.Conv2D(filters=8, kernel_size=(3,3), activation="relu", padding="valid",
                           kernel_regularizer=tf.keras.regularizers.l1_l2(l1=1e-5, l2=1e-4)),
    tf.keras.layers.MaxPool2D(pool_size=(2,3), strides=(2,3), padding="valid"),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation="relu", kernel_regularizer=tf.keras.regularizers.l1_l2(l1=1e-5, l2=1e-4)),
    tf.keras.layers.Dense(10, activation="softmax", kernel_regularizer=tf.keras.regularizers.l1_l2(l1=1e-5, l2=1e-4))
])
```

图 2.6 CNN 网络层细节

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 58, 11, 4)         40
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 29, 5, 4)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 27, 3, 8)          296
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 1, 8)          0
_________________________________________________________________
flatten (Flatten)            (None, 104)               0
_________________________________________________________________
dense (Dense)                (None, 64)                6720
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 7,706
Trainable params: 7,706
Non-trainable params: 0
```

图 2.6 CNN 框架总结

该 CNN 网络一共有 7706 个参数，每个参数(32 位浮点)要占据 4Bytes 的 ROM，这样这些权重参数一共占据约 30Kbytes 的空间，这样的大小是可以接受的。

训练 CNN 的数据集来自 Google 的 Speech_commands_dataset_version_2，该数据集里一共有 105829 个时长为 1s 的包括 35 个不同英文单词的发音。我们在训练时，只选取其中 10 个英文单词('five' 'four' 'go' 'left' 'one' 'right' 'stop' 'three' 'two' 'yes')并且每个单词的出现次数为 3000 次，训练集一共有 30000 个样本。我们把这 30000 个样本分为 train(28000)和 test(2000)。

2.3.2 CNN 网络训练结果

CNN 网络训练结果如图 2.7，其中 Validation 占 Training 数据集的 0.1。

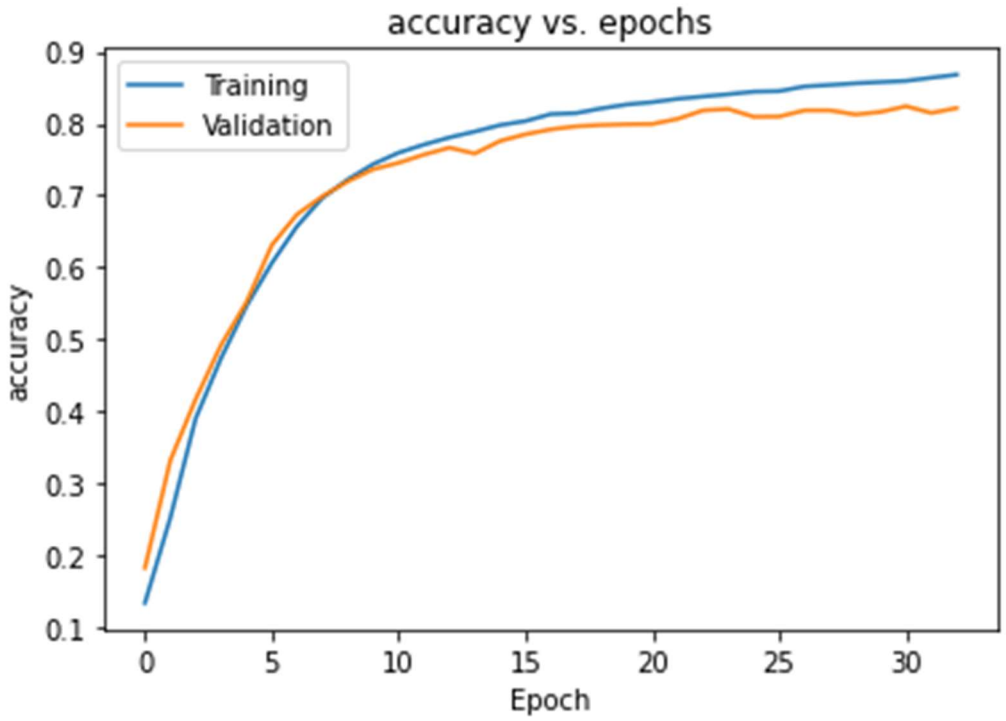

图 2.7 Epoch vs accuracy

最终的在 Test 数据集上测试的准确率大致为 **0.8**，可见网络的规模限制了 CNN 的准确率。虽然 0.8 的准确率表现比较平淡，但由于设计在 FPGA 系统实现的时候所有计算完全使用 32 位浮点数计算，和验证算法的过程完全保持一致，所以算法移植到硬件的时候 CNN 的准确率不会降低。

2.4 参考

[1] P. Warden, “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition,” arXiv:1804.03209 [cs], Apr. 2018, Accessed: May 25, 2021. [Online]. Available: http://arxiv.org/abs/1804.03209

[2] Valerio Velardo - The Sound of AI, Extracting Mel-Frequency Cepstral Coefficients with Python, (Oct. 08, 2020). Accessed: May 25, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=WJI-17MNpdE&ab_channel=ValerioVelardo-TheSoundofAI

[3] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello Edge: Keyword Spotting on Microcontrollers,” p. 14.

[4] “Speech Recognition Using Articulatory and Excitation Source Features | K. Sreenivasa Rao | Springer.” https://www.springer.com/gp/book/9783319492193 (accessed May 25, 2021).

3. 硬件

3.1 硬件系统框图

本章介绍该设计的硬件细节，首先给出系统的整体框图，然后分别给出各个硬件模块的详细说明和寄存器模型。图 3.1 给出了整个硬件系统的框图，本系统总线结构借鉴了参考文献[1]中的示例系统，除了我们设计的硬件模块外，该系统中的模块来源有：1)cmsdk，2)参考文献[1]，3)Xilinx IP(一般用于模块的底层)。

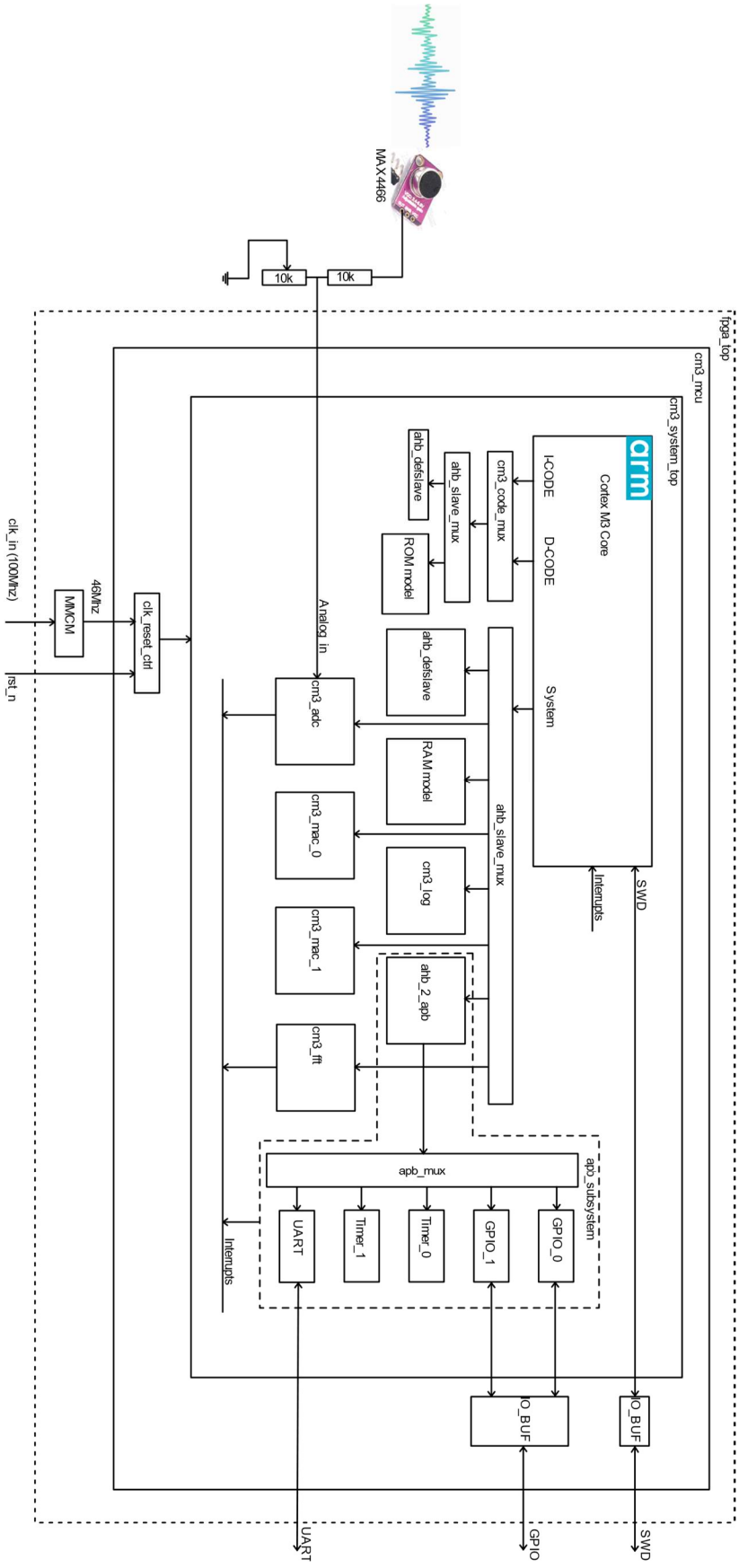
MAX4466
10k
10k
fpga_top
cm3_mcu
cm3_system_top
Cortex M3 Core
I-CODE
D-CODE
System
ahb_defslave
ahb_slave_mux
cm3_code_mux
ROM model
ahb_defslave
RAM model
cm3_adc
cm3_mac_0
cm3_mac_1
cm3_fft
cm3_log
ahb_2_apb
ahb_slave_mux
Interupts
SWD
apb_mux
apb_subsystem
UART
Timer_1
Timer_0
GPIO_1
GPIO_0
Interupts
IO_BUF
IO_BUF
Analog_in
clk_in (100Mhz)
rst_n
MMCM
46Mhz
clk_reset_ctrl
UART
GPIO
SWD

图 3.1 系统框图

本系统的总线结构比较简单，除了 Cortex_M3 核之外没有 AHB master，所有模块全为 slave 接口。与总线搭建相关的模块有 cm3_code_mux、ahb_slave_mux 和 ahb_defslave 均来自 cmsdk，注意图中没有画出地址译码器。

fpga_top 为顶层模块，该设计除了传感器外，全部使用 FPGA 片上资源，大大减少了开发成本。同时专用的浮点运算(cm3_log, cm3_mac_x, cm3_fft, cm3_log)加速单元，保证了实时算法运行的速度，注意文中的浮点数全部代表 32bit 浮点数。该硬件系统除了实现本设计的算法外，还具有通用性，通过软件控制可以实现一些别的数字信号处理的算法。下面以地址从低到高的顺序依次介绍每一个模块。

3.2 ROM model

Name	Base Address	Address size
ROM model	0x00000000	64K

利用 FPGA block ram 实现的系统 ROM，但是为了方便 Debugger 能够通过 SWD 接口把程序写进去，这个 ROM 其实是一个 RAM 并可以通过 D-CODE bus 写数据进去。注意图中没有画出 ahb2sram 模块。

3.3 RAM model

Name	Base Address	Address size
RAM model	0x20000000	64K

利用 FPGA block ram 实现的系统 RAM，注意图中没有画出 ahb2sram 的模块。

3.4 apb_subsystem

Name	Base Address	Address size
apb_subsystem	0x40000000	32K

apb_subsystem 是借用参考文献[1]中的例子，该系统中每个外设地址空间 4K。具体每个外设的寄存器模型可以参考 doc 文件夹中参考文献[1]的 chapter7-8。

3.5 cm3_adc

Name	Base Address	Address size
cm3_adc	0x40010000	64K

cm3_adc 模块是本系统的信号采集模块，用来采集外部的语音信号输入。该模块利用了 Xilinx XADC 实现，同时加上了控制逻辑和 AHB slave 接口以及中断的产生。Xilinx XADC 的精度为 12bit，速度为 1MSPS，完全可以满足语音信号采集的要求。由于 XADC 要求的电压输入信号 pp 值为 1V，而我们的音频采集模块电压信号输出 pp 值为 3.3V，所以信号在外部分压后再输入到 XADC，关于 XADC 的详细内容可以参考[2]。

XADC 每次采集一次信号会产生一个 12bit 的数据，同时转换完成后会发出中断请求，处理器把数据读走后中断会自动清除。因为总线的宽度为 32bit，为了提高总线带宽的利用率，该模块可以配置为每两次采集只发出一次中断，这样处理器可以一次读两个数据(24bit)。

3.5.1 模块框图

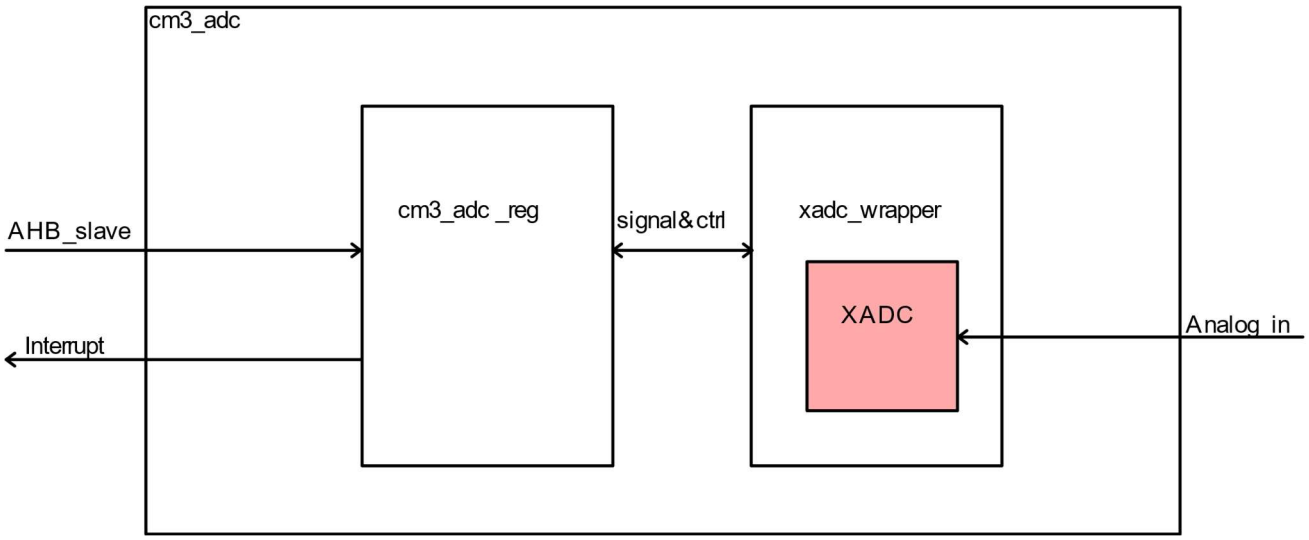

图 3.2 cm3_adc 模块框图

cm3_adc_reg 包含了寄存器以及 AHB 接口的实现和中断控制，下面其他模块中的 xxx_reg 子模块也是基本一样的功能，就不再重复。

3.5.2 寄存器

Register List:

Register Name	Offset	Width	Description
CFGCTL	0x00	32	Configure & Control
DIV	0x04	32	Control ADC sampling rate
DAT	0x08	32	ADC sample data
INT	0x0C	32	Interrupt status

Register Description

CFGCTL: 0X00, RW.

Bits	Name	Default	Description
31:3	Reserved		
2	int_enable	0	1 - Enable interrupt IRQ 0 - Disable interrupt IRQ
1	mode	0	1 - One interrupt per two conversions 0 - One Interrupt per conversion
0	enable	0	1 - ADC active 0 - ADC inactive

DIV: 0X04, RW.

Bits	Name	Default	Description
31:16	Reserved		
15:0	div	0x0c34	ADC conversion rate = clk / (div+1), div must greater than 0x64 to meet XADC minimum conversion time. In our design clk = hclk = 46Mhz.

DAT: 0X08, RO.

Bits	Name	Default	Description
31:28	Reserved		
27:16	data_2	0x000	Conversion data 2, only when mode=1. data_1 precedes data_2 in time.
15:12	Reserved		
11:0	data_1	0x000	Conversion data 1

INT: 0X0C, RW.

Bits	Name	Default	Description
31:1	Reserved		
0	data_rdy_int	0	Data ready IRQ, write 1 or read **DAT** to clear.

3.6 cm3_fft

Name	Base Address	Address size
cm3_fft	0x40020000	64K

cm3_fft 功能有：1)定点数转浮点数，2)加汉宁窗，3)计算 256 浮点傅里叶变换，4)取傅里叶变换后模的平方。该模块的实现利用了 Xilinx 的 Fast Fourier Transform IP[3]和 Floating-Point IP[4]。

因为该设计在算法上采用的是浮点数计算，所以必须把 ADC 采集到的 12bit 定点数转换为**单精度**浮点数再进行运算，软件会通过把无符号的 12bit 整型数减去直流分量转换为 32bit 的有符号整型数。硬件的输入为 32bit 有符号整数，定点数转浮点数的规则为：

$$f = (float)32bit\_signed\_int \times 2^{scale} / 2^{12}$$

$scale$是一个可以通过寄存器配置的参数，除以2^{12}是为了利用定点数转浮点数的单元，顺便实现信号归一化的效果。

对信号加窗的操作是可选的，通过配置寄存器来实现。该模块的基本操作流程为，向该模块对应地址写 256 个数，模块会根据配置进行一系列运算，最终完成运算后向系统发出中断，待运算结果被处理器从该模块取出后，中断会自动清除。该模块对于一般的数字信号处理算法具有通用性。

3.6.1 模块框图

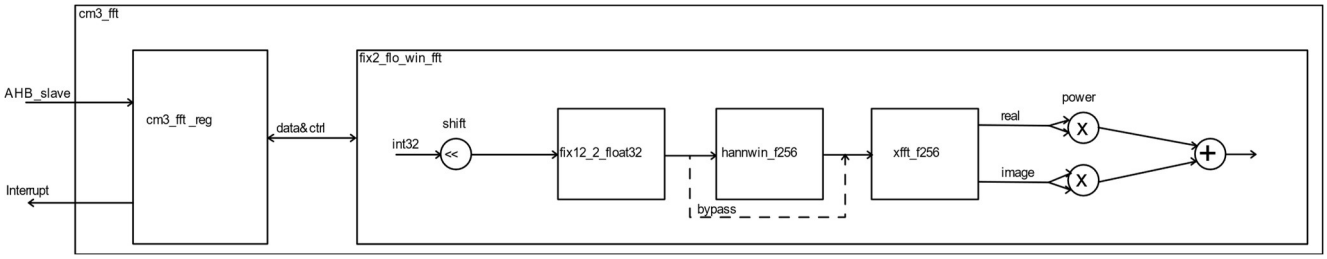

图 3.3 cm3_fft 模块框图

fix12_2_float32 模块用来将有符号定点数(该模块认为定点数的小数位为 12bit)转换为单精度浮点数。**hanwin_f256** 包含浮点乘加单元和 256 点汉宁窗的系数(存储在 ROM table)，对流过的数据进行加窗操作。注意 bypass 路径并没有画出 mux。**xfft_f256** 对浮点数据进行 fft 操作，输出的 fft 结果通过两个浮点乘法单元和一个浮点加法单元得到 fft 模的平方。模块进一步的细节可以参考 RTL 代码。

3.6.2 寄存器

Register List:

Register Name	Offset	Width	Description
CFG	0x00	32	Configure
CTL	0x04	32	Control
DIN	0x08	32	Data in
DOUT	0x0C	32	Data out

Register Description：

CFG: 0X00, RW.

Bits	Name	Default	Description
31:16	Reserved		
15:8	n_need	0xff	
7:4	scale	0	Cover input data in*2**scale/4096 to Float point number.
1	int_enable	0	1 - Enable interrupt IRQ 0 - Disable interrupt IRQ After read **n_need** data from **DOUT**, interrupt clear automatically.
0	window	0	1 - Window before fft 0 - No window before fft

CTL: 0X04, RW.

Bits	Name	Default	Description
31:1	Reserved		
0	fft_enable	0	1 - Enable cm3_fft 0 - Disable cm3_fft

DIN: 0X08, WO.

Bits	Name	Default	Description
31:0	data_in		Input data buffer

DOUT: 0X0C, RO.

Bits	Name	Default	Description
31:0	data_out	0x0000_0000	Output data buffer

3.7 cm3_mac

Name	Base Address	Address size
cm3_mac_0	0x40030000	64K
cm3_mac_1	0x40040000	64K

cm3_mac 为浮点数乘加单元，该模块的实现利用了 Xilinx Floating-Point IP[4]。cortex m3 没有专用的浮点指令，如果利用软件实现浮点运算的话，做一次单精度浮点乘法或者加法大概

要 100 个 cycle，这样的速度完全不能满足实时算法的要求。cm3_mac 单元包含三个寄存器，两个输入寄存器 A，B，和一个输出寄存器 OUT。算法中很多运算都是向量的点积运算，假设有两个浮点数向量 a[10]与 b[10]做点积，只需要把 a[0]…a[9]与 b[0]…b[9]按顺序写到寄存器 A 和寄存器 B，最后读 OUT 寄存器(读完 OUT 后，cm3_mac 的累加和自动清零)得到向量点积的结果，这样的话一次乘加运算就减少为两个写寄存器的时间(最快 2cycle 每次乘加)。在最后读 OUT 寄存器的时候，要距离最后写 A，B 寄存器的时刻至少 2 个 cycle，这意味着如果只计算两个浮点数的乘法的话至少需要 4 个 cycle(2 个 cycle 写寄存器，2 个 cycle 等待结果)，所以点积向量越长，越划算。

如果一个 cm3_mac 正在在主程序中使用，然后突然进来中断也使用这个 cm3_mac 进行不同的运算，这样数据就会遭到完全破坏。所以这里有两个 cm3_mac 单元，其中一个给主程序用，一个给中断服务函数用，这样避免了两段程序用一个 cm3_mac 单元，当然如果有需要可以再加。

3.7.1 模块框图

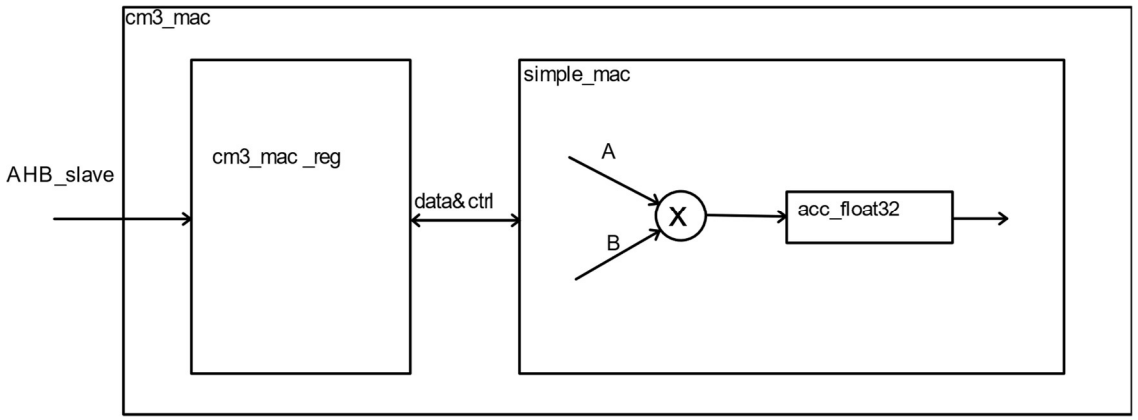

图 3.4 cm3_mac 模块框图

注意该模块没有中断信号。

3.7.2 寄存器

Register List:

Register Name	Offset	Width	Description
DATINA	0x00	32	Data in A
DATINB	0x04	32	Data in B
DOUT	0x08	32	Data result

Register Description

DATINA: 0X00, WO.

Bits	Name	Default	Description
31:0	data_in_a		Input data A

DATINB: 0X04, WO.

Bits	Name	Default	Description
31:0	data_in_b		Input data B

DOUT: 0X08, RO.

Bits	Name	Default	Description
31:0	data_out	unknown	Data out. 2 cycle delay，software need to handle this delay.

3.8 cm3_log

Name	Base Address	Address size
cm3_log	0x40050000	64K

cm3_log 为取 log 操作，log 的底为e，该模块的实现利用了 Xilinx Floating-Point IP[4]。虽然取 log 操作在算法中的比例很小，但是使用软件取 log 需要大约 1000 个 cycle。cm3_log 操作只需要向 DATIN 寄存器写一个浮点数，然后再从 DATOUT 寄存器中读出结果，结果会有 4cycle 的延迟。这样一个 log 操作就只需要 5 个 cycle。

3.8.1 模块框图

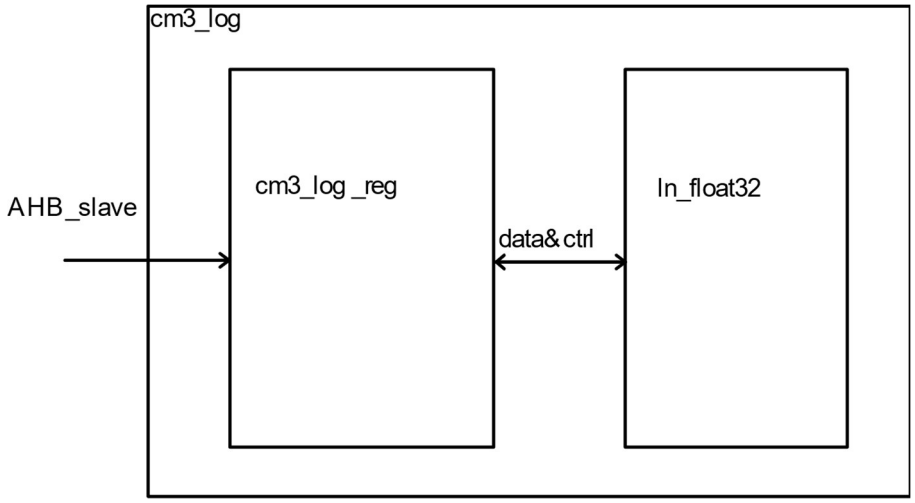

图 3.5 cm3_log 模块框图

ln_float32 为 Xilinx IP 模块。

3.8.2 寄存器

Register List:

Register Name	Offset	Width	Description
DATIN	0x00	32	Data in
DATOUT	0x04	32	Data in out

Register Description

DIN: 0X00, WO.

Bits	Name	Default	Description
31:0	data_in		Input data buffer

DOUT: 0X0, RO.

Bits	Name	Default	Description
31:0	data_out	unknown	Output data buffer

3.9 硬件实现

我们 FPGA 的型号为 Xilinx Artix 7a35tcsg324-1，开发环境为 Vivado 2020.1。经过 implementation 后的资源利用率和 floorplanning 视图为图 3.6 和图 3.7。我们通过优化架构来降低资源的使用，最终可以在低端的 FPGA 上实现系统。如果想要查看更详细的工程信息，可在文件目录/first/hardware/build/vivado/vivado 下找到该系统工程文件打开。/first/hardware/tb 文件夹包含系统的 testbench 和各个子模块的 testbench。/first/hardware/rtl 包含了所有有效的硬件代码，同时也包括了 Xilinx IP(.xci 文件)管理文件夹。/first/hardware/build/vivado/design_test 为测试模块功能所用的 vivado 工程文件夹。

Resource	Utilization	Available	Utilization %
LUT	19427	20800	93.40
LUTRAM	257	9600	2.68
FF	10457	41600	25.14
BRAM	34	50	68.00
DSP	47	90	52.22
IO	24	210	11.43
BUFG	4	32	12.50
MMCM	1	5	20.00

图 3.6 系统资源利用率

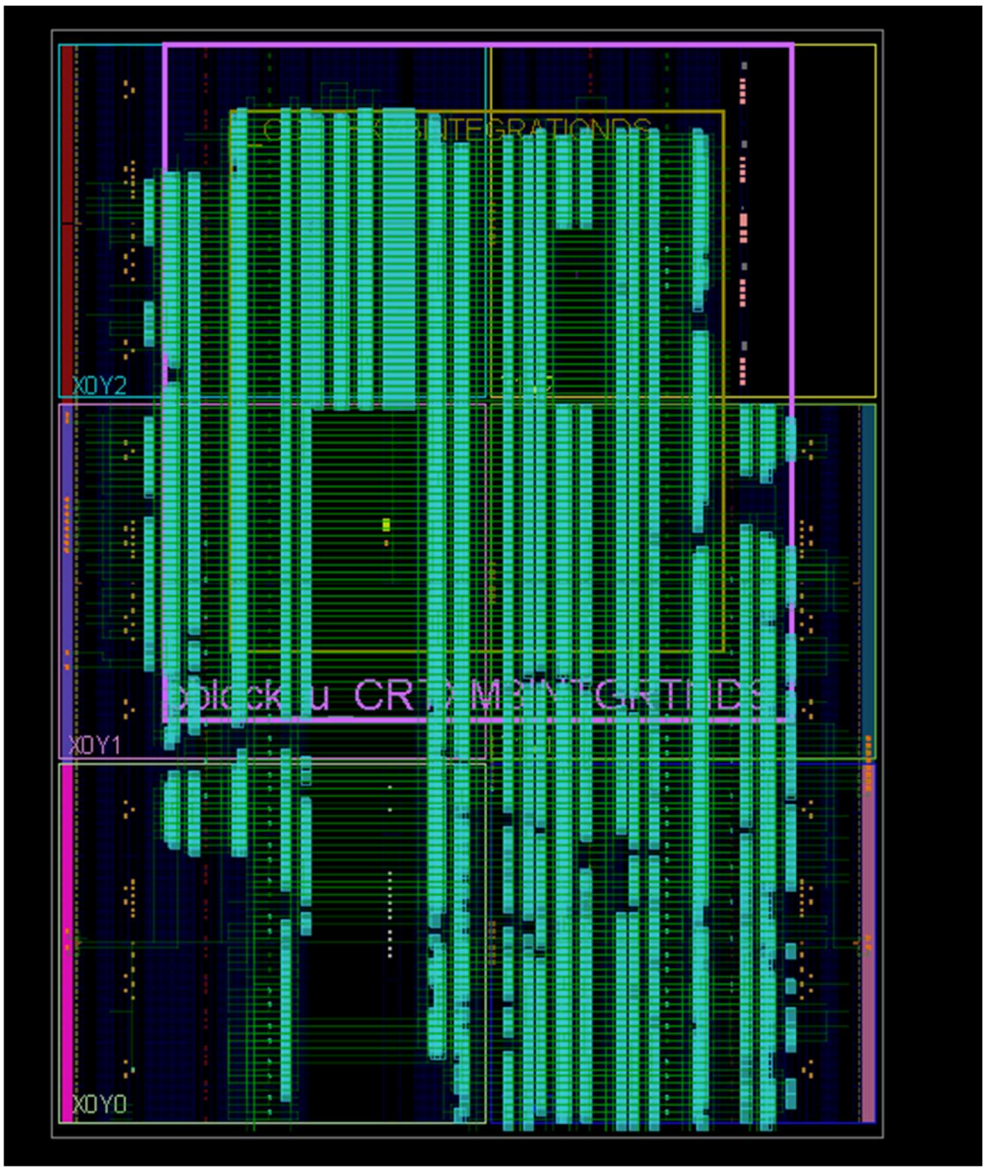

图 3.7 系统 floorplanning view

3.10 参考

[1] A. Ltd, “System-on-Chip Design with Arm Cortex-M – Arm,” *Arm | The Architecture for the Digital World*. https://www.arm.com/resources/ebook/system-on-chip-design (accessed May 30, 2021).

[2] “7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480),” , 2018.

[3] “Fast Fourier Transform v9.1 LogiCORE IP Product Guide,” , 2021.

[4] “Floating-Point Operator v7.1 LogiCORE IP Product Guide,” , 2020.

[5] “Documentation – Arm Developer.” https://developer.arm.com/documentation/ddi0479/c/ (accessed May 30, 2021).

4. 系统实现

4.1 系统概述

本章节讲述整个系统的实现细节，包含软件的实现和软硬件协同设计的内容。软件所有的代码在/first/software/src 目录下，keil 工程在/first/software/build 文件夹下。注意工程利用了 CMSIS Core 和 Compiler IO 包，只拷贝源代码编译是不会通过的，必须要在 keil 中进行正确的配置后才能编译。软件的编写主要分为两个部分，第一部分为音频信号 MFCC 特征的提取，第二部分为利用提取到的音频 MFCC 特征进行 CNN 推理运算。首先介绍软硬件的交互，再分别介绍算法的实现流程。

4.2 软硬件交互

以 cm3_adc 模块为例说明软硬件交互的过程。cm3_adc 模块的基地址为 0x40010000，并且有四个寄存器分别为 CFGCTL、DIV、DAT 和 INT 寄存器。首先在 cm3_mcu.h 文件中定义该模块的结构体：

```
typedef struct
{
  __IO   uint32_t  CFG;          /*!< Offset: 0x000 (R/W) */
  __IO   uint32_t  DIV;          /*!< Offset: 0x004 (R/W) */
  __I    uint32_t  DAT;          /*!< Offset: 0x008  (RO) */
  __IO   uint32_t  INT;          /*!< Offset: 0x00C  (R/Wc) */
} CM3MCU_ADC_TypeDef;
```

然后给出该模块的基地址并且 define 一个指向该地址的该结构体类型的指针：

```
#define CM3MCU_ADC_BASE  (0x40010000UL)

#define CM3MCU_ADC       ((CM3MCU_ADC_TypeDef *) CM3MCU_ADC_BASE )
```

这样就可以通过定义的结构体指针去访问该模块了，几乎每个模块都有对应的.c 文件，包含一些初始化函数和对模块常用的操作函数。比如该模块对应的 adc.c 中的 init 函数：

```
#include "adc.h"
//div : must greater than 100 to meet adc conversion speed
//mode: 0 stands for signle transfer, otherwise double transfer
void adc_init_start(uint32_t div, uint32_t mode)
{
  //config div
    CM3MCU_ADC -> DIV = div;
    /*enable interrupt*/
    //Enable NVIC
    NVIC_EnableIRQ(ADC_IRQn);

    //start adc
    if (mode)
    {
        CM3MCU_ADC -> CFG = 7; //111
    } else
    {
        CM3MCU_ADC -> CFG = 5; //101
    }
}
```

该模块的中断号 ADC_IRQn 根据硬件的连接在 cm3_mcu.h 定义，同时在 startup_cm3_mcu.s 中的中断向量表的对应给出定义对应的中断处理函数地址(函数名)。其他模块的交互方式与该模块类似，具体细节可以参考工程源代码。

4.3 MFCC 特征提取流程

根据算法所描述的 MFCC 提取流程，每一个语音样本长度$L = 7080$，分成每一帧的长度为$l = 256$，每一帧之间的步长$s = 128$，每一帧对应一个长度为$F = 13$的 MFCC 特征，每一个语音样本可以得到60×13大小的特征矩阵，这意味这我们需要一个这么大的 buffer 来存储 MFCC 的结果。

因为步长为$s = 128$，ADC 每采集 128 个点后我们就需要结合上次缓存的 128 个点全部送到 cm3_fft 模块，让 cm3_fft 对这一帧数据进行求功率谱的操作。这些操作在 cm3_adc 模块的中断服务函数 ADC_Handler(位于 int_handlers.c)中实现：

```
void ADC_Handler(void)
{
    uint32_t sample = adc_read_data(); //clear int
    if (adc_buffer.index == HOPSIZE)
    {
        while (adc_buffer.index)
        {
            fft_write(adc_buffer.buffer[adc_buffer.index--]);
        }
    }
    //move current sample to adc_buffer
    adc_buffer.buffer[adc_buffer.index++] = (sample & 0x0000ffff) - ADC
_BIAS;
    adc_buffer.buffer[adc_buffer.index++] = (sample>>16) -  ADC_BIAS;
    //move sample to adc_buffer
    fft_write(adc_buffer.buffer[adc_buffer.index-2]);
    fft_write(adc_buffer.buffer[adc_buffer.index-1]);
    return;
}
```

其中 fft_write 为写数据到 cm3_fft 模块的函数，adc_buffer 为缓存 128 个数据的 buffer，同时该系统中的所有 buffer 均在 buffer.c 文件中定义。

当 cm3_fft 模块接受到 256 点数据后会进行自动进行运算，运算完成后会发起中断，在其中断函数 FFT_Handler 中会对硬件运算完的结果(功率谱)进行后续的计算最终得到 MFCC 特征并存储到 mfccs_buffer 中。下面是 FFT_Handler 的操作流程，具体代码请参考附录 int_handlers.c：

```
void FFT_Handler(void)
{
    从 cm3_fft 模块中读取功率谱结果;

    if(mfccs_buffer is full)
    {
       return;//放弃这一帧数据
    } else
    {
       利用硬件加速单元 cm3_mac 和 cm3_log 计算得到 MFCC 特征;
       将 MFCC 特征存入 mfccs_buffer;
    }
}
```

在进行 MFCC 后续计算的时候需要一些系数，这些系数存储在 weight.c 文件夹中。

4.4 CNN 实现

现在我们说明 cnn 函数(cnn.c)的一些实现细节，CNN 的运算包括点积、relu、maxpool 和 softmax 操作。下面是 cnn 推理函数的顶层实现，注意每一层的实现函数的输入输出参数都为固定大小的数组，这样虽然减小了灵活性，但增加了运算的效率。

```
//cnn flow
uint8_t cnn(const float input [N_FRAMES][N_MFFCCS])
{
    conv1(input, conv2d_W, conv2d_b, conv1_buffer);
    pool1(conv1_buffer, pool1_buffer);
    conv2(pool1_buffer, conv2d_1_W, conv2d_1_b, conv2_buffer);
    pool2(conv2_buffer, pool2_buffer);
    dense1((float *)pool2_buffer, dense_W, dense_b, dense1_buffer);
    linear(dense1_buffer, dense_1_W, dense_1_b, dense2_buffer);
    softmax(dense2_buffer, 10);  //final result is in dense2_buffer
    return argmax(dense2_buffer, 10); // return final index
}
```

点积运算利用该系统的 cm3_mac 单元进行加速运算，具体操作表现为读写 cm3_mac 寄存器。relu 运算最主要是要判断一个浮点数是否大于 0，根据 IEEE 754 标准，我们只需要判断最高位是否为 1 就可以得出结论，所以 relu 操作用软件实现。maxpool 的运算包含浮点数的比较，也是用软件直接进行实现。softmax 包括求 exp 和浮点除法两个运算操作，因为每次预测一个样本只需要进行一次 exp 运算，而除法只需要进行十次，所以这两个运算也只需要软件实现完全可以满足算法的速度。CNN 网络的所有权重存储在 weight.c 文件中，CNN 的具体实现请参考附录 cnn.c 以及 cnn.h，这两个文件还引用了别的函数，由于报告篇幅有限不可能全部列出，有需要可以参考源码。

CNN 预测函数在 main 函数中被调用，一个基本例子程序的运算流程可以如下：

```
void main(void)
{
    while(1)
    {
        while(mfccs_buffer is full);
        result = cnn(mfccs_buffer);
        printf(result);
        利用 result 控制外设等等……;
        flush mfccs_buffer 四分之一的数据;
    }
}
```

因为我们希望每个时间样本之间有重叠，所以在对 mfccs_buffer 中的 60 个 MFCC 特征预测完之后删除其中的 15 个 MFCC 特征，保留其中的 45 个，这样就意味着每 15 个 MFCC 特征就需要进行一次 CNN 的预测，这样我们每秒钟要大约预测 4 个长度为$L = 7808$的样本。这个流程可以根据不同的应用环境来进行改变，并非固定的。

4.5 参考

[1] Y. Zhu, *Embedded Systems with ARM® Cortex-M3 Microcontrollers in Assembly Language and C*. E-Man Press, 2014. Accessed: May 30, 2021. [Online]. Available: https://digitalcommons.library.umaine.edu/fac_monographs/208

[2] *The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors*. Elsevier, 2014. doi: 10.1016/C2012-0-01372-5.

5.总结和提升

5.1 设计亮点

1. 硬件采用浮点计算，实现了小型算法移植的灵活性和方便性，以及软件实现的简洁性。
2. 浮点加速器，保证了算法运行的实时性。
4. 通用的浮点加速器，Soc 平台具有能力实时运行任意小型 dsp 算法
3. 系统架构简单，成本低，并且嵌入式语音识别的应用场景十分广阔，市场潜力巨大。

5.2 正在改进

我们正在努力改进以下几点：

1. 加入具有 AHB master 接口的定点矩阵加速器，给算法实现更多选择，增强 Soc 的通用性。
2. 优化总线结构，加入 DMA 模块，增强 Soc 的完整性和速度。
3. 改进算法，提升算法的抗噪声能力。
4. 给系统的性能进行一个定量的评估。

6.附录

为了展示，这里列出了一小部分软件 c 代码，并不代表完整工程。

int_handlers.c:

```c
#include "adc.h"
#include "fft.h"
#include "cm3_calculate.h"
#include "buffer.h"
#include "weight.h"
#include "stdio.h"

//ADC_Handler priority = 1
void ADC_Handler(void)
{
    uint32_t sample = adc_read_data(); //clear int
    if (adc_buffer.index == HOPSIZE)
    {
        while (adc_buffer.index)
        {
            fft_write(adc_buffer.buffer[adc_buffer.index--]);
        }

    }
    //move current sample to adc_buffer
    adc_buffer.buffer[adc_buffer.index++] = (sample & 0x0000ffff) -  ADC_BIAS;
    adc_buffer.buffer[adc_buffer.index++] = (sample>>16) -  ADC_BIAS;
    //move sample to adc_buffer
    fft_write(adc_buffer.buffer[adc_buffer.index-2]);
    fft_write(adc_buffer.buffer[adc_buffer.index-1]);
    return;
}

//FFT_Handler  priority = 2
void FFT_Handler(void)
{
    uint32_t i;
    for ( i = 0; i < FFT_NEED; i++)
    {
        fft_buffer[i] = fft_read();
    }
```

```
    //if the buffer is full return, abound this frame, wait cnn to read buffer
    if (mfccs_buffer.index == N_FRAMES_PER)
    {
        return;
    }

    //get 26 filters result and compute log
    for ( i = 0; i < N_FILTER_BANKS; i++)
    {
        bank_result_buffer[i] = fmac(CM3MCU_MAC0, (float *)mel_filter_banks_coes_W[i
], fft_buffer+
            mel_filter_banks_start_len[i][0], mel_filter_banks_start_len[i][1]);
        // printf("%.10f,\n", bank_result_buffer[i]);
    }
    //add 10e-8, avoid log(0)
    fadd_bias(CM3MCU_MAC0, bank_result_buffer, 1e-8, N_FILTER_BANKS);
    //log base e
    for ( i = 0; i < N_FILTER_BANKS; i++)
    {
        bank_result_buffer[i] = flog(bank_result_buffer[i]);
    }
    //calculate DCT result
    for ( i = 0; i < N_MFFCCS; i++)
    {
        mfccs_buffer.buffer[mfccs_buffer.index][i] = fmac(CM3MCU_MAC0, bank_result_b
uffer, dct_coe[i], N_FILTER_BANKS);
    }
    mfccs_buffer.index++;
    return;
}
```

cnn. h:

```
#ifndef CNN_H
#define CNN_H
#include "buffer.h"
void conv1(const float input[N_FRAMES][N_MFFCCS], const float W[4][3][3][1], const f
loat b[4], float result[58][11][4]);
void pool1(const float input[58][11][4],  float output[29][5][4]);
```

```
void conv2(const float input[29][5][4], const float W[8][3][3][4], const float b[8],
 float result[27][3][8]);
void pool2(const float input[27][3][8],  float output[13][1][8]);
void dense1(const float * input, const float W[104][64], const float b[64] ,float  o
utput[64] );
void linear(const float * input, const float W[64][10], const float b[10] ,float  ou
tput[10]);
void softmax(float *inout, uint32_t len);
uint8_t argmax(const float * predict, uint8_t len);
uint8_t cnn(const float input [N_FRAMES][N_MFFCCS]);

#endif
```

cnn. c：

```
#include "cm3_calculate.h"
#include "cnn.h"
#include "math.h"
#include "weight.h"
void conv1(const float input[N_FRAMES][N_MFFCCS], const float W[4][3][3][1], const f
loat b[4], float result[58][11][4])
{
    uint32_t k, r, c, i, j;
    float tmp;
    for ( k = 0; k < 4; k++)
    {
        for ( r = 0; r < 58; r++)
        {
            for ( c = 0; c < 11; c++)
            {
                //for one mac, 9
                for (i = r; i < r+3; i++)
                {
                    for ( j = c; j < c+3; j++)
                    {
                        //利用 cm3_mac(乘加单元)进行浮点向量点积运算 *
                        CM3MCU_MAC1 -> DATINA = W[k][i-r][j-c][0];
                        CM3MCU_MAC1 -> DATINB = input[i][j];
                    }

                }
                //add bias
                CM3MCU_MAC1 -> DATINA = b[k];
                CM3MCU_MAC1 -> DATINB = 1.0;
                __nop();
                __nop();
                __nop();
                //fetch result from mac unit
                //RELU
                //MSB = 1 is negtive
                tmp = CM3MCU_MAC1 -> DOUT;

                //Relu operation
                if (*(uint32_t*)(&tmp)>>31) //if tmp < 0.0
                {
                    result[r][c][k] = 0.0;
                }
```

```
                else
                {
                    result[r][c][k] = tmp;
                }


            }

        }

    }

}

//pooling size is 2x2 and stride is 2x2
//max pooling
void pool1(const float input[58][11][4],  float output[29][5][4])
{
    uint32_t k, r, c;
    float tmp;
     for ( k = 0; k < 4; k++)
    {
        for ( r = 0; r < 29; r ++)
        {
            for ( c = 0; c < 5; c ++)
            {   //three compare
                tmp = input[r*2][c*2][k] > input[r*2][c*2+1][k]? input[r*2][c*2][k]:
 input[r*2][c*2+1][k];
                output[r][c][k] = input[r*2+1][c*2][k] > input[r*2+1][c*2+1][k]? inp
ut[r*2+1][c*2][k]: input[r*2+1][c*2+1][k];
                if (tmp > output[r][c][k])
                {
                    output[r][c][k] = tmp;
                }

            }
        }
    }
}


void conv2(const float input[29][5][4], const float W[8][3][3][4], const float b[8],
 float result[27][3][8])
{
```

```
uint32_t k, r, c, i, j, d;
float tmp;
for ( k = 0; k < 8; k++)
{
    for ( r = 0; r < 27; r++)
    {
        for ( c = 0; c < 3; c++)
        {
            //for one mac, 9
            for (i = r; i < r+3; i++)
            {
                for ( j = c; j < c+3; j++)
                {
                    for (d = 0; d < 4; d++)
                    {
                        CM3MCU_MAC1 -> DATINA = W[k][i-r][j-c][d];
                        CM3MCU_MAC1 -> DATINB = input[i][j][d];
                    }
                }

            }
            //add bias
            CM3MCU_MAC1 -> DATINA = b[k];
            CM3MCU_MAC1 -> DATINB = 1.0;
            __nop();
            __nop();
            __nop();
            //fetch result from mac unit
            //RELU
            //MSB = 1 is negtive
            tmp = CM3MCU_MAC1 -> DOUT;

            //Relu operation
            if (*(uint32_t*)(&tmp)>>31) //if tmp < 0.0
            {
                result[r][c][k] = 0.0;
            }
            else
            {
                result[r][c][k] = tmp;
            }


        }
```

```
        }

    }

}


//pooling size is 2x3 and stride is 2x3
void pool2(const float input[27][3][8],  float output[13][1][8])
{
    uint32_t k, r;
    float tmp;
     for ( k = 0; k < 8; k++)
    {
        for ( r = 0; r < 13; r ++)
        {
            tmp = input[r*2][0][k] > input[r*2][1][k]? input[r*2][0][k] : input[r*2]
[1][k];
            tmp = tmp > input[r*2][2][k] ? tmp : input[r*2][2][k];

            output[r][0][k] =  input[r*2+1][0][k] > input[r*2+1][1][k]? input[r*2+1]
[0][k] : input[r*2+1][1][k];
            output[r][0][k] =  output[r][0][k] > input[r*2+1][2][k] ? output[r][0][k
] : input[r*2+1][2][k];
            if (tmp > output[r][0][k])
            {
                output[r][0][k] = tmp;
            }
        }
    }
}

//13*8 -> 64
void dense1(const float * input, const float W[104][64], const float b[64] ,float  o
utput[64] )
{
    uint32_t r, c;
    float tmp;
    for(r = 0; r < 64; r++)
    {
        for(c = 0; c < 104; c++)
        {
            CM3MCU_MAC1 -> DATINA = input[c];
```

```
            CM3MCU_MAC1 -> DATINB = W[c][r];
        }

        //add bias
        CM3MCU_MAC1 -> DATINA = b[r];
        CM3MCU_MAC1 -> DATINB = 1.0;
        __nop();
        __nop();
        __nop();
        //fetch result from mac unit
        //RELU
        //MSB = 1 is negtive
        tmp = CM3MCU_MAC1 -> DOUT;

        //Relu operation
        if (*(uint32_t*)(&tmp)>>31) //if tmp < 0.0
        {
            output[r] = 0.0;
        }
        else
        {
            output[r] = tmp;
        }
    }
}


void linear(const float * input, const float W[64][10], const float b[10] ,float  ou
tput[10])
{
    uint32_t r, c;
    for(r = 0; r < 10; r++)
    {
        for(c = 0; c < 64; c++)
        {
            CM3MCU_MAC1 -> DATINA = input[c];
            CM3MCU_MAC1 -> DATINB = W[c][r];
        }
        CM3MCU_MAC1 -> DATINA = b[r];
        CM3MCU_MAC1 -> DATINB = 1.0;
        __nop();
        __nop();
        __nop();
```

```
        output[r] = CM3MCU_MAC1 -> DOUT;

    }
}

void softmax(float *inout, uint32_t len)
{
    uint32_t i;
    float sum;
    //exp(inout)
    for(i=0; i<len; i++)
    {
        inout[i] = expf(inout[i]);
        CM3MCU_MAC1 -> DATINA = inout[i];
        CM3MCU_MAC1 -> DATINB = 1.0;
    }
        // __nop();
        // __nop();
        // __nop();

    sum = CM3MCU_MAC1 -> DOUT;
    // x/sum
    for(i=0; i<len; i++)
    {
        inout[i] = inout[i] / sum;

    }

}

//get the max index , predict length  max to 128
uint8_t argmax(const float * predict, uint8_t len)
{
    uint8_t index = 0, i;
    for(i=0; i<len-1; i++){
        index = predict[i+1] > predict[index] ? i + 1 : index;
    }
    return index;
}


//cnn flow
uint8_t cnn(const float input [N_FRAMES][N_MFFCCS])
{
```

```
    conv1(input, conv2d_W, conv2d_b, conv1_buffer);
    pool1(conv1_buffer, pool1_buffer);
    conv2(pool1_buffer, conv2d_1_W, conv2d_1_b, conv2_buffer);
    pool2(conv2_buffer, pool2_buffer);
    dense1((float *)pool2_buffer, dense_W, dense_b, dense1_buffer);
    linear(dense1_buffer, dense_1_W, dense_1_b, dense2_buffer);
    softmax(dense2_buffer, 10);  //final result is in dense2_buffer
    return argmax(dense2_buffer, 10); // return final index
}
```