



**汇编与接口 课 程 设 计**

团队实验报告

|  |  |
| --- | --- |
| 学 院 | 计算机学院 |
| 专 业 | 计算机科学与技术 |
| 指导老师 | 黄永刚 |
| 组 长 |  |
| 组 员 |  |
| 组长联系方式 |  |

二O二四年 九 月

目 录

[第一章 项目简述 1](#_Toc176979672)

[第二章 组员分工 1](#_Toc176979673)

[第三章 设计目的 2](#_Toc176979674)

[第四章 设计环境 2](#_Toc176979675)

[第五章 设计原理及内容 2](#_Toc176979676)

[5.1 整体架构 2](#_Toc176979677)

[5.1.1 数据通路 3](#_Toc176979678)

[5.1.2 控制逻辑 7](#_Toc176979679)

[5.2 LCD显示 7](#_Toc176979680)

[5.2.1 LCD模块的参数 7](#_Toc176979681)

[5.2.2 显示图像的主要过程 9](#_Toc176979682)

[5.2.3 生成图像数据 14](#_Toc176979683)

[5.2.4 其他 17](#_Toc176979684)

[5.3 蜂鸣器播放音乐 18](#_Toc176979685)

[5.3.1 蜂鸣器发声原理 18](#_Toc176979686)

[5.3.2 音乐数据处理 19](#_Toc176979687)

[5.4 内存映射I/O 20](#_Toc176979688)

[5.4.1 工作原理 20](#_Toc176979689)

[5.4.2 优点 20](#_Toc176979690)

[5.5 C语言环境搭建 21](#_Toc176979691)

[5.5.1使用交叉编译工具 21](#_Toc176979692)

[5.5.2设置栈来支持函数调用 22](#_Toc176979693)

[5.5.3使用链接脚本调整程序布局 23](#_Toc176979694)

[5.5.4 Makefile文件 23](#_Toc176979695)

[第六章 设计与实现 25](#_Toc176979696)

[6.1 蜂鸣器播放音乐模块实现 25](#_Toc176979697)

[6.2 内存映射I/O模块实现 27](#_Toc176979698)

[6.2.1 数据存储器和外设寄存器的映射 28](#_Toc176979699)

[6.2.2 时序控制逻辑 28](#_Toc176979700)

[6.2.3 数据读写逻辑 29](#_Toc176979701)

[6.3 软件代码实现 30](#_Toc176979702)

[6.3.1 模块概述 30](#_Toc176979703)

[6.3.2 main程序实现 31](#_Toc176979704)

[6.3.3 LCD模块 33](#_Toc176979705)

[6.3.4 Base API 模块 34](#_Toc176979706)

[第七章 测试 37](#_Toc176979707)

[仿真测试用例、结果以及分析 37](#_Toc176979708)

[ILA 技术在开发板上的调试与仿真 38](#_Toc176979709)

[第八章 问题及解决方法 39](#_Toc176979710)

[1. 内存读写对齐问题 39](#_Toc176979711)

[2. 时序控制问题 40](#_Toc176979712)

[3. Verilog 编码规范问题 40](#_Toc176979713)

[第九章 心得体会及总结 41](#_Toc176979714)

[第十章 参考文献有价值的资源推荐 41](#_Toc176979715)

# 项目简述

在CPU设计的基础上，项目进一步扩展到外设接口的实现，特别是针对一个“音乐播放器”的外设功能。该播放器集成了LCD显示屏、蜂鸣器、LED灯和数码管，通过按钮进行操作控制，实现了音乐播放、状态显示和视觉反馈等功能。整个系统的设计和实现涉及了Verilog编程、流水线CPU架构、外设接口控制以及C语言交叉编译等多个技术层面，最终通过硬件仿真和实体下板测试验证了系统的功能和性能。

# 组员分工

小组分工见表2.1 小组分工。

表2.1 小组分工

|  |  |
| --- | --- |
| **组员** | **分工任务** |
| 组长 | * 负责CPU总体设计实现 * 外设部分的总体设计实现 * LCD模块设计实现 * CPU及外设联合设计编码 * 仿真及下板测试等 |
| 组员1 | * 负责CPU数据冒险处理 * CPU数据通路扩充 * 仿真及下板测试 * 结构流程图绘制 |
| 组员2 | * 负责CPU控制冒险处理 * 蜂鸣器音乐模块 * 仿真及下板测试 * PPT制作 |
| 组员3 | * 负责CPU指令扩充 * CPU逻辑功能扩充 * 功能仿真 * 文档报告撰写 |

# 设计目的

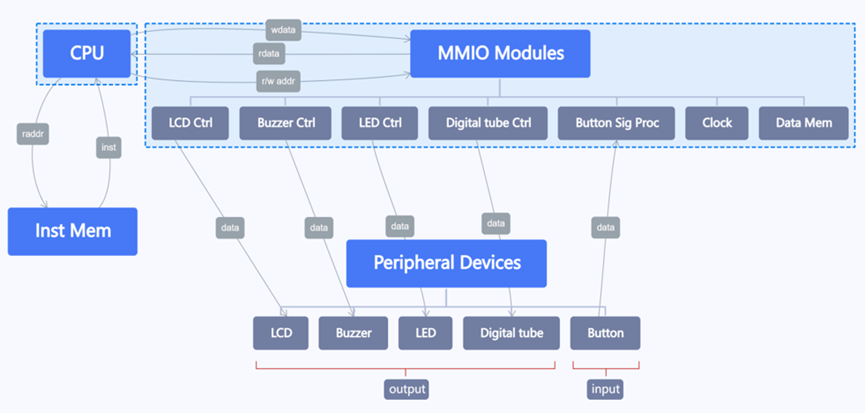
实现一个简易的“音乐播放器”。LCD显示播放状态和图片、蜂鸣器用于播放音乐、LED用于根据音乐进行闪烁、按键用于控制暂停/继续、上一首、下一首，数码管显示用于CPU下板和调试。

# 设计环境

|  |  |
| --- | --- |
| 操作系统 | Windows11 |
| 编程语言 | Verilog |
| EDA工具 | Vivado2022.2 |
| 汇编语言 | riscv32i |
| 汇编程序编辑器 | VSCode |

# 设计原理及内容

## 整体架构

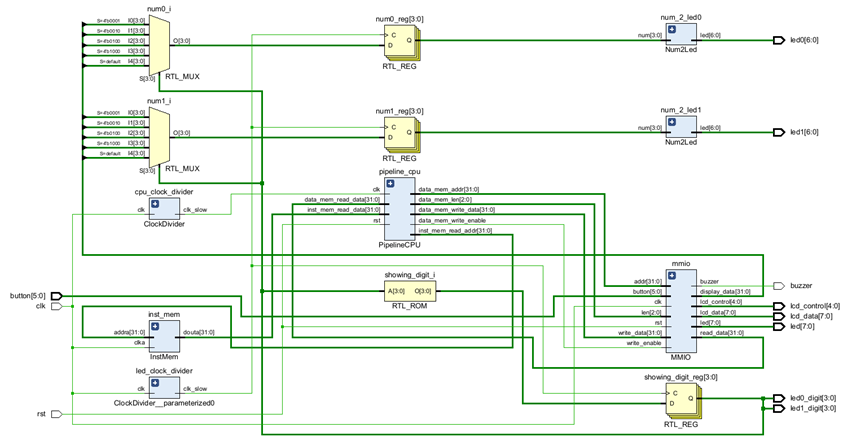


5.1-1整体架构图

### 数据通路

系统的整体架构可以分为以下几个关键部分：

* **CPU核心：**作为系统的核心处理单元，CPU负责执行指令、处理数据和控制程序的运行。
* **指令存储器（InstMem）：**用于存储CPU执行的指令代码。这些指令通过指令存储器提供给CPU，以供执行。
* **数据存储器（DataMem）：**存储CPU在执行过程中需要读写的数据。数据存储器与CPU之间的数据交换是通过地址总线和数据总线进行的。
* **外设接口：**包括LCD显示屏、蜂鸣器、LED灯和数码管等。这些外设通过特定的接口控制器与CPU相连，实现了数据的输入输出和用户交互。
* **时钟分频器（ClockDivider）：**用于生成不同频率的时钟信号，以满足不同外设的工作需求。例如，蜂鸣器需要特定频率的信号来播放音乐，而LED灯需要较低频率的信号来控制闪烁。
* **内存控制器（MMIO）：**实现了内存映射I/O，使得CPU可以通过读写内存的方式间接控制外设。CPU的数据输入输出只和MMIO交互，由MMIO判断读写的地址。如果读写的地址为外设地址则由MMIO控制直接从外设读写，否则对实际数据内存进行读写。通过这种方式实现了外设和内存的统一编址，控制外设和读写内存对CPU没有区别。这种方式简化了CPU与外设之间的通信机制。



5.1.1-1整体数据通路图示

接下来依次描述各主要模块:

1. **MMIO模块**的主要功能是将内存地址空间的一部分映射到特定的外设寄存器上，从而使得CPU可以通过读写这些内存地址来实现对外部设备的控制。模块的功能和信号描述如下：

* **addr**：32位地址总线，用于指定当前操作的内存地址或外设寄存器地址。
* **len**：3位长度指示信号，用于指定读写操作的数据长度。
* **write\_data**：32位写数据总线，用于向外设寄存器写入数据。
* **read\_data**：32位读数据总线，用于从外设寄存器读取数据。
* **display\_data**：32位输出寄存器，用于存储要显示在七段数码管上的数据。
* **button**：6位输入信号，用于接收外部按钮的输入状态。
* **lcd\_data**：8位输出寄存器，用于存储要显示在LCD上的数据。
* **lcd\_control**：5位输出寄存器，用于控制LCD的各种操作。
* **led**：8位输出寄存器，用于控制LED灯的状态。
* **buzzer**：输出信号，用于控制蜂鸣器。

MMIO模块通过解析地址信号（addr[31:0]）来确定CPU是要访问内存还是进行外设控制。如果地址匹配外设的地址范围，则MMIO模块将执行相应的外设控制操作，如读取按钮状态、控制蜂鸣器发声或更新LCD显示。如果地址匹配内存地址，则MMIO模块将允许CPU进行正常的内存读写操作。这种设计使得CPU能够使用统一的指令集来处理内存和外设，简化了系统架构并提高了灵活性。

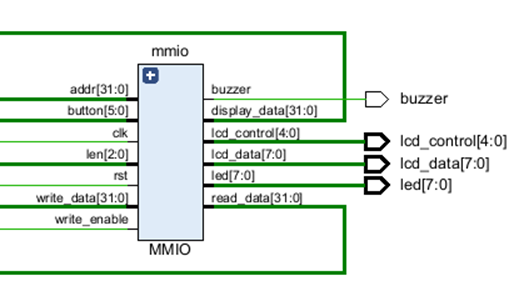


图5.1.1-2 mmio数据通路

1. **DataMem模块**是一个数据存储单元，负责处理来自CPU的数据读写请求。该模块通过其输入输出信号与CPU进行交互，实现数据的存储与提取。具体来说，DataMem模块的功能包括：

* **write\_enable**：写使能信号，当该信号有效时，模块会根据地址将写入数据存储到内存中。
* **addr[31:0]**：32位的地址总线，用于指定要读写的内存地址。
* **len[2:0]**：3位的写入数据长度信号，用于指示写入操作的数据宽度，可以是8位、16位或32位。
* **write\_data[31:0]**：32位的写入数据总线，用于向指定地址写入数据。
* **read\_data[31:0]**：32位的读出数据总线，用于输出从指定地址读取的数据。

在DataMem模块内部，使用了一个双端口RAM（DataBram），它允许同时进行读写操作。通过bram\_wea信号控制写入RAM的字节使能，根据len信号的值来决定是进行8位、16位还是32位的数据写入。当write\_enable为高时，根据len的值设置bram\_wea，以选择正确的数据宽度进行写入；当write\_enable为低时，禁止写入操作。

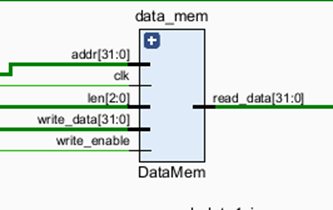


图5.1.1-3 Datamem数据通路

1. **Music\_Player模块**用来播放存储在内存中的音乐信息。该模块通过解析和执行内存中的音乐数据来驱动蜂鸣器产生旋律，从而实现音乐播放的功能。模块的功能和信号描述如下：

* **clk**：时钟信号，用于控制模块内所有时序逻辑和同步操作。
* **rstn**：复位信号，用于初始化或重置音乐播放状态。
* **next\_song** 和 **prev\_song**：控制信号，用于在不同歌曲之间进行切换。当激活next\_song时，模块会跳转到下一首歌；激活prev\_song时，模块会跳转到上一首歌。
* **pause\_resume**：控制信号，用于暂停或恢复当前音乐的播放。
* **buzzer\_out**：输出信号，连接到蜂鸣器模块，用于控制蜂鸣器发声。

音乐播放模块内部使用了一个名为music\_memory的寄存器数组来存储音乐信息，每个音符由8位数据表示，包括八度和音调。模块通过music\_index来跟踪当前播放的音符索引，并根据timer来控制播放速度和节拍。每当timer达到预设的阈值时，模块就会更新当前播放的音符，并输出一拍的音乐。

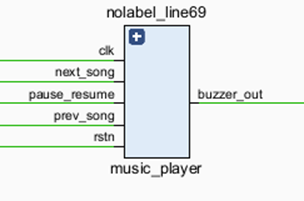


图5.1.1-4musicplayer数据通路

1. **显示数字模块**由两部分组成：Num2Led 模块和控制逻辑。Num2Led 模块负责将输入的数字转换为数码管对应的编码，而控制逻辑则负责根据显示数据选择要显示的数字。

Num2Led 模块是一个简单的数字到七段数码管编码的转换器。它接收一个4位的二进制输入 num，表示0到15之间的数字（包括代表字母A-F的数字），并输出一个7位的信号 led，用于控制数码管的每个段是否点亮。每个数字或字母在数码管上有固定的编码模式。在顶层模块中，Num2Led 模块被实例化为两个实例：num\_2\_led0 和 num\_2\_led1，分别用于控制两组数码管。每组数码管通过 led0 和 led1 信号显示数字，而 led0\_digit 和 led1\_digit 信号则用于控制显示的位数。

控制逻辑部分，使用了一个始终块（always block），它在 led\_clk 的上升沿时钟触发。该逻辑根据 showing\_digit 寄存器的值选择要显示的数字，并通过 display\_data 总线获取要显示的数值。showing\_digit 用于循环显示数码管的不同数字，它按照预设的顺序循环通过数码管的每一位，确保每个数字都能在对应的数码管上显示。

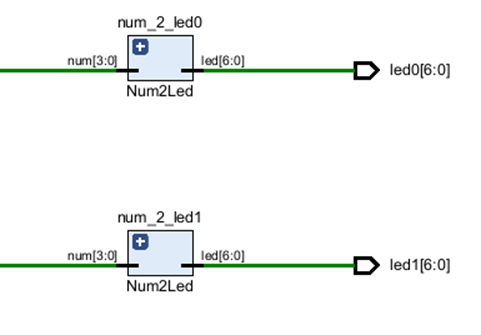


图5.1.1-5 Num2led数据通路

### 控制逻辑

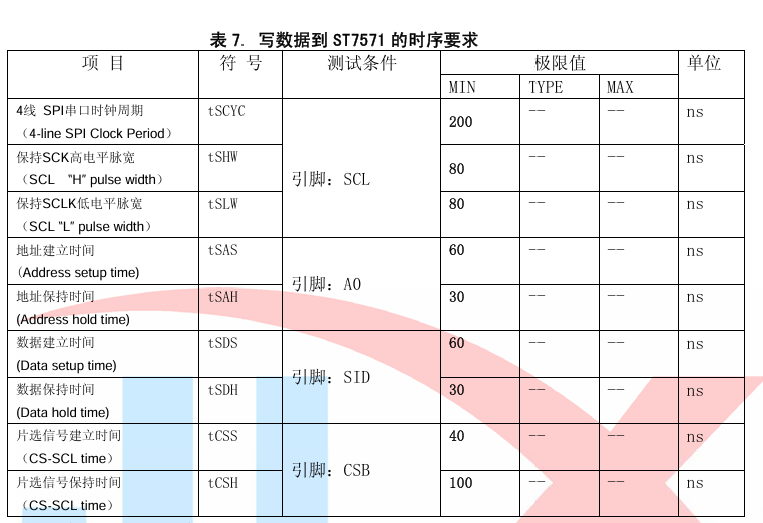
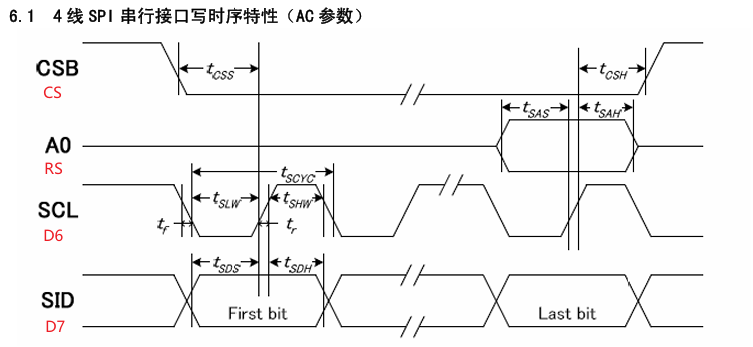
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号 | 位宽 | **输入/输出** | 描述 |
| 1 | button | 6 | Input | 6个游戏按键 |
| 2 | lcd\_data | 8 | Output | 对应LCD的LCD\_D7 ~ LCD\_D0 原理图标号 |
| 3 | lcd\_control | 5 | Output | 对应LCD的LCD\_RS LCD\_RST LCD\_CS LCD\_RD LCD\_WR原理图标号 |
| 4 | led | 8 | Output | 8个led灯，对应 LED7 ~ LED0 原理图标号 |
| 5 | buzzer | 1 | Output | 蜂鸣器 |
| 6 | led0 | 7 | Output | 7段数码管段选信号 |
| 7 | led0\_digit | 4 | Output | 7段数码管位选信号 |
| 8 | led1 | 7 | Output | 7段数码管段选信号 |
| 9 | led1\_digit | 4 | Output | 7段数码管位选信号 |

## LCD显示

### 5.2.1 LCD模块的参数

1. **这款LCD可以显示128×128的单色或4灰度图片。**
2. **引脚的功能介绍：**
   1. **RS（Register Select）**：用于选择寄存器，1为数据寄存器，0为指令寄存器
   2. **RST（Reset）**：低电平复位
   3. **CS（Chip Select）**：低电平片选。当CS信号为低电平时，表示选中了特定的设备或芯片，可以进行数据传输或操作；当CS信号为高电平时，表示设备或芯片处于空闲状态，不进行数据传输
   4. **RD**：使能信号，在串行方式中不用
   5. **RW（Read Write）**：1为读数据，0为写数据，在串行方式中不用
   6. **D7~D0（Data7~Data0）**：在并行方式中表示一字节数据的高位到低位；在串行方式中**D7**用于传输数据，**D6**用于时钟信号，其他位不用
3. **串行SPI方式写入数据的时序要求**

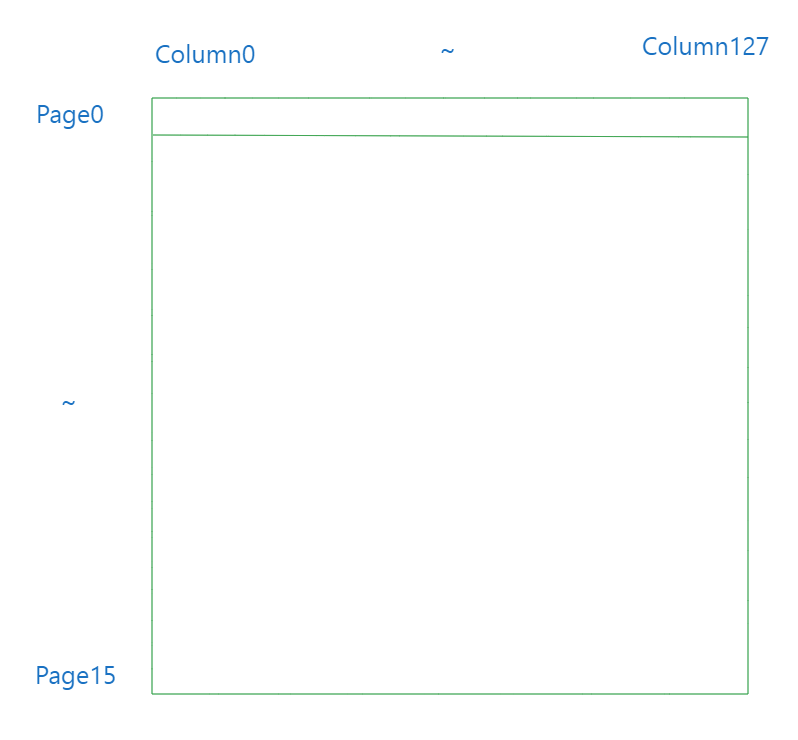
文档中给出的时序要求如下：



关键点如下：

* 写入数据前要置`CS`为0，数据写完后要置`CS`为1
* 在传输数据的最后一位时必须要指定`RS`为1（写入的是数据）还是0（写入的是指令）
* 在传输数据的每一位（`D7`）时，需要保证时钟信号（`D6`）先为低电平后为高电平
* 具体时间要求要看表格中的极限值

1. **page和column**



page将LCD的128行分为16组，每组8行；column对应LCD的128列的每一列。

1. **4灰度显示**

每个像素点需要2bit来决定显示的内容，00~11由白变黑，且黑色程度逐级增加。

### 5.2.2 显示图像的主要过程

这部分主要参考文档第19~28页给出的程序。在实际测试过程中，当以并行方式操作LCD时，它不能显示图像。之后更换了以串行SPI方式操作LCD，结果显示正常。所以实验中以**串行**方式让LCD显示图像。

1. 编写操作管脚的函数：置开发板中与LCD相关的管脚约束为1或0



* SetWR()：置`WR`为1
* ClearWR()：置`WR`为0
* SetRD()：置`RD`为1
* ClearRD()：置`RD`为0
* SetCS()：置`CS`为1
* ClearCS()：置`CS`为0
* SetRST()：置`RST`为1
* ClearRST()：置`RST`为0
* SetRS()：置`RS`为1
* ClearRS()：置`RS`为0
* WriteLcdData(uchar data)：根据data由高位到低位置`D7`~`D0`为1或0

1. 编写向LCD传送数据和命令的函数：按照文档27页要求编写函数

// 向 LCD 传送命令

// 串行版

void TransferCommandToLcd(uchar command) {

ClearCS();ClearRS();

for(int i = 0; i < 8; i++) {

if(command & 0x80) {

WriteLcdData(0b10111111);

// SleepUs(1);

WriteLcdData(0b11111111);

}

else {

WriteLcdData(0b00111111);

// SleepUs(1);

WriteLcdData(0b01111111);

}

// SleepUs(1);

command <<= 1;

}

SetCS();

}

// 向 LCD 传送数据

// 串行版

void TransferDataToLcd(unsigned char data) {

ClearCS();SetRS();

for(int i = 0; i < 8; i++) {

if(data & 0x80) {

WriteLcdData(0b10111111);

// SleepUs(1);

WriteLcdData(0b11111111);

}

else {

WriteLcdData(0b00111111);

// SleepUs(1);

WriteLcdData(0b01111111);

}

// SleepUs(1);

data <<= 1;

}

SetCS();

}

这里解释一下各个语句的含义：

* ClearCS()置片选信号为0，表示开始进行数据或指令传输
* ClearRS()和SetRS()分别表示向指令寄存器和向数据寄存器写入数据
* 之后的for循环表示将要传送的数据或指令从高位到低位逐个传送，并保证时钟先为低电平，后为高电平。以数据`0b10111111`为例：最高位的1表示要传输的数据为1，接着的0表示时钟为低电平，剩余6位在串行方式中不会用到，可随意设置
* 最后调用`SetCS()`表示数据或指令传输完毕
* 文档中要求传输数据时的时钟高电平与低电平需要至少持续一段时间，所以代码在最开始调用了`SleepUs(1)`休眠一微秒。或许由于我们的CPU时钟周期为40ns和插入气泡等因素，我发现不调用该函数LCD也可以正常工作，大家可以根据自己情况是决定是否需要休眠

1. 初始化LCD：该过程必须在让LCD显示图像前调用

// 初始化 Lcd，在使用前必须调用

void InitLcd() {

// 复位，rst保持低电平最少 2us

ClearRST();

SleepUs(500);

SetRST();

SleepUs(100);

// 电源控制

TransferCommandToLcd(0x2c);

SleepUs(200);

TransferCommandToLcd(0x2e);

SleepUs(200);

TransferCommandToLcd(0x2f);

SleepUs(10);

TransferCommandToLcd(0xae); // 显示关

TransferCommandToLcd(0x38); // 模式设置

TransferCommandToLcd(0xb8); // 85 Hz

TransferCommandToLcd(0xc8); // 行扫描顺序

TransferCommandToLcd(0xa0); // 列扫描顺序

TransferCommandToLcd(0x44); // Set initial COMO register

TransferCommandToLcd(0x00);

TransferCommandToLcd(0x40);//Set initial display line register

TransferCommandToLcd(0x00);

TransferCommandToLcd(0xab);

TransferCommandToLcd(0x67);

TransferCommandToLcd(0x26); // 粗调对比度的值， 0x20 ~ 0x27

TransferCommandToLcd(0x81); // 微调对比度

TransferCommandToLcd(0x36); // 微调对比度的值， 0x00 ~ 0x3f

TransferCommandToLcd(0x54); // 1/9 bias

TransferCommandToLcd(0xf3);

TransferCommandToLcd(0x04);

TransferCommandToLcd(0x93);

// TransferCommandToLcd(0x7b); // 扩展指令集 3

// TransferCommandToLcd(0x11); // Gray mode

// TransferCommandToLcd(0x10); // Gray mode

// TransferCommandToLcd(0x00);

TransferCommandToLcd(0xaf); // 显示开

}

这里调用`TransferCommandToLcd()`函数向LCD传送了一系列指令，如果对每条指令的含义有兴趣，可查阅文档的第13~15页。

1. 定位LCD要操作屏幕的位置

// 设置 LCD 要操作的地址

// page 的值从 0 15，colum的值从 0 127

void SetLcdAddress(int page, int column) {

ClearCS();

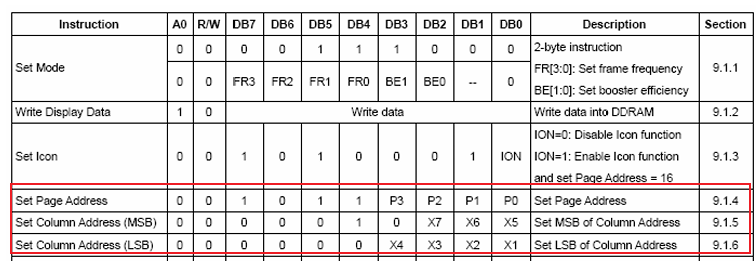
TransferCommandToLcd(0xb0 + page);

TransferCommandToLcd(((column >> 4) & 0x0f) + 0x10);

TransferCommandToLcd(column & 0x0f);

}

这个过程涉及到的指令如下：



1. **向LCD传送数据来显示图像**

// 显示图像

void ShowImageOnLcd(unsigned char\* image) {

for(int page = 0; page < 16; page++) {

SetLcdAddress(page, 0);

for(int column = 0; column < 128; column++) {

TransferDataToLcd(image[page \* 128 + column]);

TransferDataToLcd(image[page \* 128 + column]);

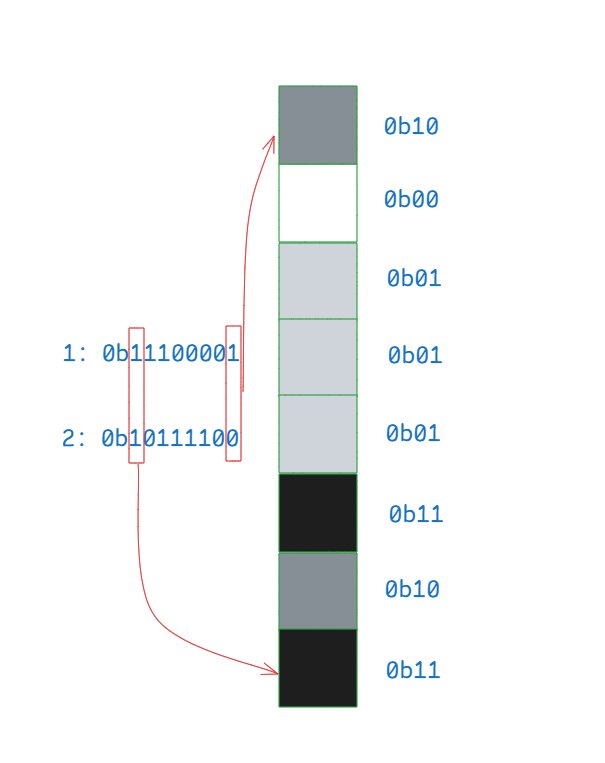
}

}

}

这里需要解释一下这个函数的含义：

* 该函数的主体是一个双层循环，在外层循环的开始调用SetLcdAddress()函数来定位显示位置
* 内层循环每列调用了两次TransferDataToLcd()函数来向LCD传输数据，具体每位的对应关系可以参考下图



* 通过每列两次调用TransferDataToLcd()传送相同的内容可以达到显示单色图像的效果

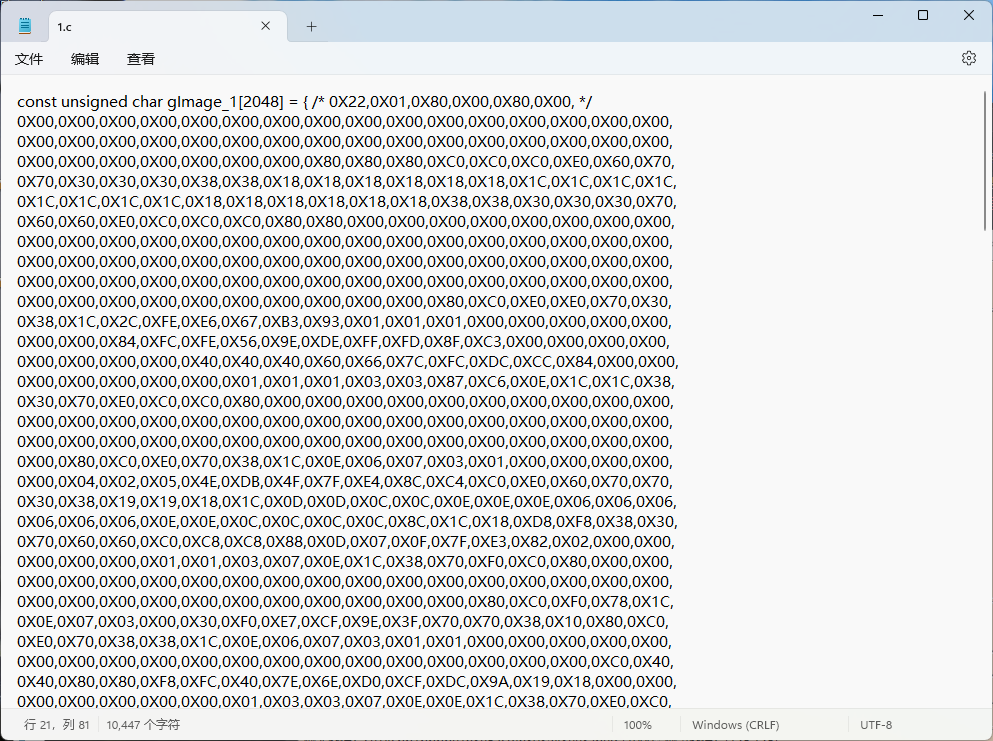
### 5.2.3 生成图像数据

生成图像数据使用**Image2Lcd**软件，这款软件输入图片后可以获得C语言数组形式的图像数据。可以通过此软件生成单色图像和4灰度图像。

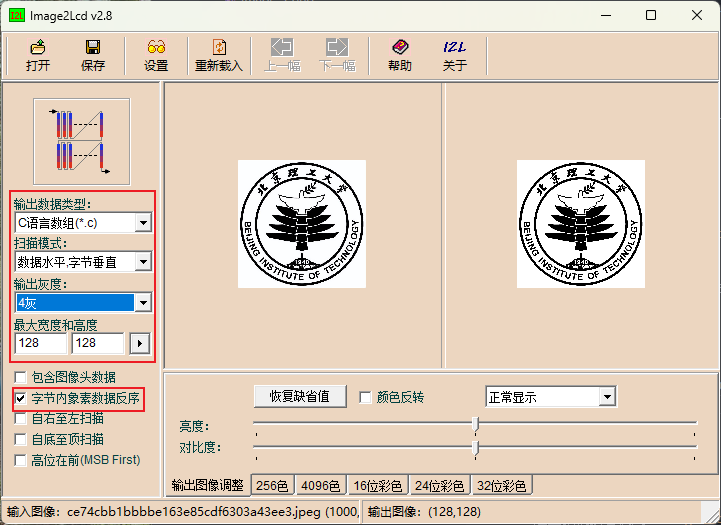
按照图中的配置设置软件，就可以得到可以使LCD正确显示的单色图像数据。



点击**保存**就可以得到C语言数组形式的图像数据。

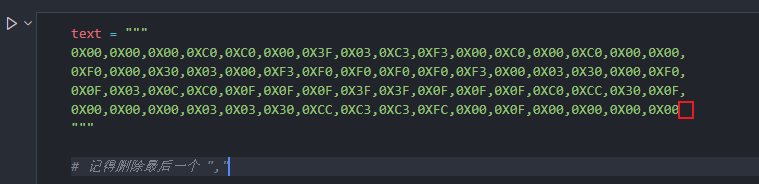


生成4灰度图像需要做如下设置。



需要注意的是，该软件生成的C语言数组形式并不能被我们直接使用，**因为它的格式和我们要求的格式不同**，所以这里编写了一个脚本来进行转换。该脚本名为gen\_4\_grey.ipynb，脚本的使用方法如下：

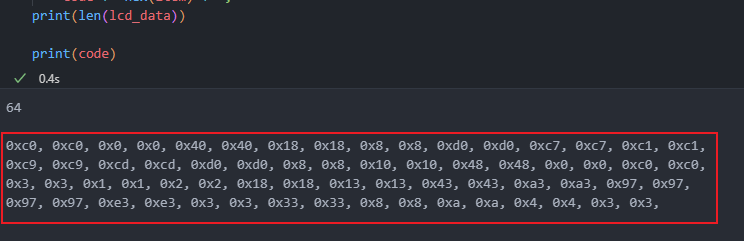
1. 将软件生成的4灰度图像的C语言数组元素内容贴到text变量中，注意要删除最后一个“,”，否则脚本无法正常工作。



1. 指定图像的高度和宽度，要和text所对应。



1. 运行整个脚本，可以在最后得到转换后的C语言数组元素内容。



生成后我们可以调用如下函数来显示4灰度图像：

// 显示4灰度图像

void ShowGreyImageOnLcd(unsigned char\* image) {

for(int page = 0; page < 16; page++) {

SetLcdAddress(page, 0);

for(int column = 0; column < 128; column++) {

TransferDataToLcd(image[page \* 128 + column]);

TransferDataToLcd(image[page \* 128 + column + 1]);

}

}

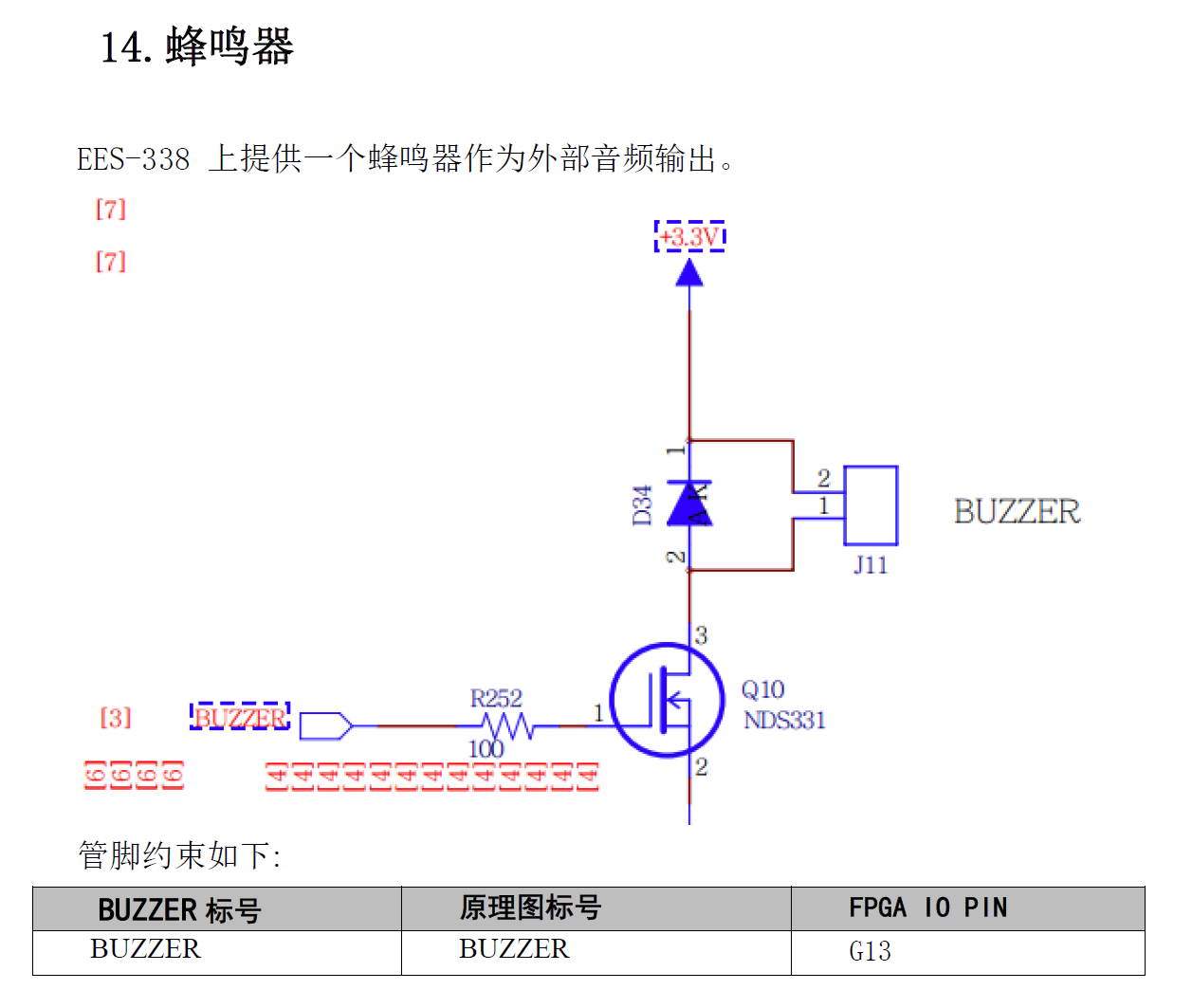
}

### 5.2.4 其他

1. 缩小图像数据大小：从上面可以看到，显示一张128×128的单色图像需要2KB，显示一张128×128的4灰度图像需要4KB。我们可以将屏幕化划分为不变部分和可变部分，每次只修改可变部分，同时可变部分可以区分单色区域和4灰度区域，这样可以显著减少图像占据的空间。
2. 文字：LCD提供了显示文字的功能，但是我没有使用LCD提供的显示文字功能，而是使用图像来显示文字，这样更方便我们排版。

## 蜂鸣器播放音乐

### 5.3.1 蜂鸣器发声原理



使用的EES338开发板自带一个无源蜂鸣器模块。无源蜂鸣器通过输入不同频率的方波信号来发出不同声调的声音，通过控制信号的占空比来控制响度。实验中我们需要进行音乐播放，所以只需要控制输入蜂鸣器的方波频率即可。

每个声调都有自己的频率，我们只需要为蜂鸣器输入该频率的方波信号即可控制蜂鸣器发出相应的声音。根据八度音阶和十二平均律可知：一个八度内几乎平均分为12个音，高八度频率翻倍，低八度频率减半。音调和频率的对照如下表所示。

音调 Hz 音调 Hz 音调 Hz

低音1 262 中音1 523 高音1 1046

低音1# 277 中音1# 554 高音1# 1109

低音2 294 中音2 587 高音2 1175

低音2# 311 中音2# 622 高音2# 1245

低音3 330 中音3 659 高音3 1318

低音4 349 中音4 698 高音4 1397

低音4# 370 中音4# 740 高音4# 1480

低音5 392 中音5 784 高音5 1568

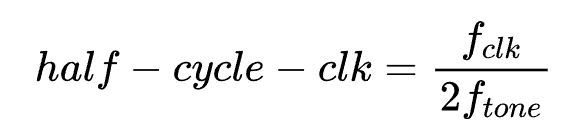
低音5# 415 中音5# 831 高音5# 1661

低音6 440 中音6 880 高音6 1760

低音6# 466 中音6# 932 高音6# 1865

低音7 494 中音7 988 高音7 1976

控制频率可以通过计数器来实现。已知板载时钟信号为100MHz，我们可以为不同的声调设置不同的计数器阈值，当时钟周期数达到设置的声调阈值即可反转输出信号。因此所设置的计数器阈值是所需频率信号的半周期时钟数通过如下公式计算：



实验中实现了三个八度的音调以演奏更加丰富的曲调。高八度和低八度分别使用中音半周期时钟数的一半和两倍来表示，对应频率的翻倍和减半。

同样的，我们也可以通过计数器来控制一个节拍的时间长度。实验中确定一个节拍为0.5秒。

### 5.3.2 音乐数据处理

在设计中，我们使用2位来表示低（01）中（10）高（11）三个八度，使用4位来表示具体的声调，因此表示一个节拍的声调需要6位。同时为了方便存取数据和扩展，使用一个字节（8位）中的低6位来储存一个节拍的声调。在定义中，控制八度的2位为0或控制声调的4位为0均不发出声音。八度内的声调表示如下：

if (level == 2'b01) begin //以低八度为例，此时计数器阈值需要加倍使得频率减半

case (tone)

4'b0001: half\_t\_clk <= C5\*2;

4'b0010: half\_t\_clk <= C5s\*2;

4'b0011: half\_t\_clk <= D5\*2;

4'b0100: half\_t\_clk <= D5s\*2;

4'b0101: half\_t\_clk <= E5\*2;

4'b0110: half\_t\_clk <= F5\*2;

4'b0111: half\_t\_clk <= F5s\*2;

4'b1000: half\_t\_clk <= G5\*2;

4'b1001: half\_t\_clk <= G5s\*2;

4'b1010: half\_t\_clk <= A5\*2;

4'b1011: half\_t\_clk <= A5s\*2;

4'b1100: half\_t\_clk <= B5\*2;

default: half\_t\_clk <= 0;

endcase

为了能够演奏音乐，我们需要将音乐从谱子转化为我们的声调表示方法。按照节拍将曲子转化为节拍数据流并存储在数组中即可。一个音如果需要多个节拍需要按照节拍数重复表示。

## 内存映射I/O

在本实验中，为了实现外设接口，我们采用了内存映射输入输出（Memory-Mapped I/O, MMIO）技术。MMIO是一种通过将外设寄存器映射到CPU的地址空间中的特定地址来实现与外设的交互的技术。在这种模式下，CPU可以像访问内存一样访问外设。

### 5.4.1 工作原理

MMIO的基本原理是将外设的寄存器分配到一块特定的地址空间，这块地址空间与内存空间共享，但其地址范围是专门为外设保留的。当CPU通过总线向特定地址发送读写请求时，MMIO模块会根据地址的范围区分该访问是针对内存还是外设。

* **地址区分**: CPU发送的每一个读写请求都会包含一个内存地址，MMIO模块通过检查该地址是否落在外设的地址范围内来决定是访问内存还是外设寄存器。例如，假设外设的寄存器被映射到0x40000000至0x4000FFFF的地址范围内，当CPU访问这个地址范围时，MMIO模块将请求转发至相应的外设寄存器，否则将请求转发到内存控制器来访问实际的内存。
* **读写操作**: 一旦MMIO模块确定了访问目标是外设寄存器，它会直接与外设的硬件通信，执行相应的读写操作。通过这种方式，CPU不需要专门的指令集来控制外设，而是像访问内存一样与外设进行数据交换。

### 5.4.2 优点

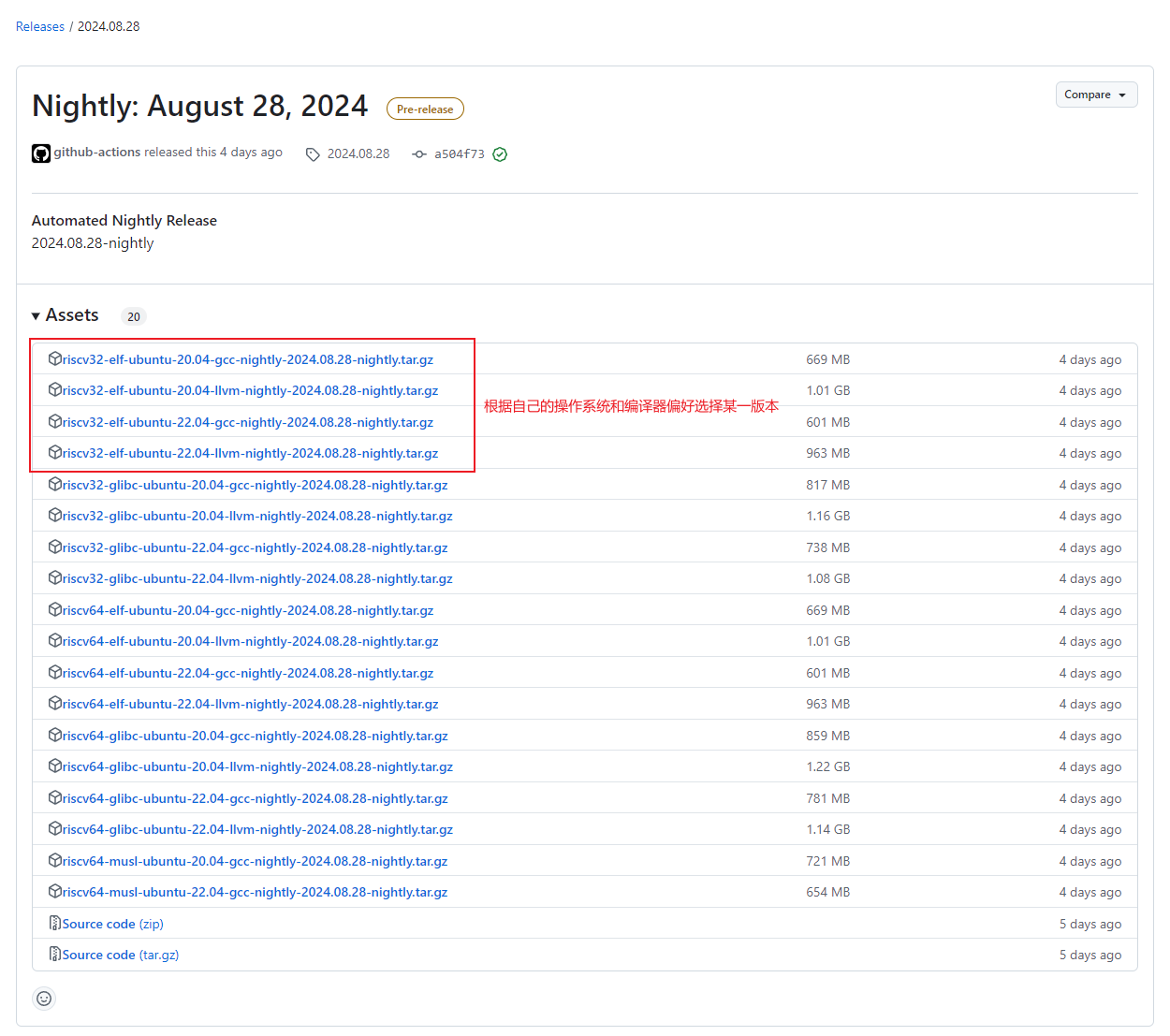
MMIO技术的主要优点是简化了CPU与外设之间的通信。CPU可以使用常规的内存读写指令与外设交互，而无需通过特殊的I/O指令，从而提高了系统的灵活性。此外，MMIO还能与虚拟内存系统相结合，实现更加复杂的内存管理和权限控制。

## 5.5 C语言环境搭建

### 5.5.1使用交叉编译工具

由于我们构建的CPU的指令集为riscv，而我们自己电脑（宿主机）的CPU的指令集为x86\_64或ARM等，默认情况下编译后生成的是和宿主机相同指令集的二进制文件，所以这里需要交叉编译工具。

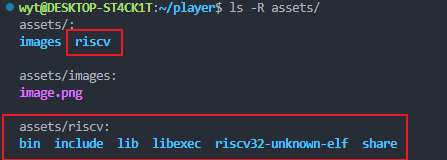
可以从[官网](https://github.com/riscv-collab/riscv-gnu-toolchain)自行下载源码进行编译，也可以从该[链接](https://github.com/riscv-collab/riscv-gnu-toolchain/releases)直接下载已经编译好的二进制文件，这里直接使用已经编译好的二进制文件进行使用。



选择一个任意时间的发行版，然后选择某一版本进行下载。这里选择riscv32-elf-ubuntu-22.04-gcc-nightly-2024.08.28-nightly.tar.gz版本进行下载。它的文件名含义如下：

* riscv32：目标机的指令集为riscv32
* elf：没有标准的运行时库（表明没有任何系统调用的封装支持），但可以生成`ELF`格式的执行程序
* gcc：使用gcc编译器

为了减少修改Makefile文件的地方，我们解压后将其放置到./assets目录下。



### 5.5.2设置栈来支持函数调用

这部分内容位于start.S中。栈一般位于内存的高地址部分，向低地址部分扩展。

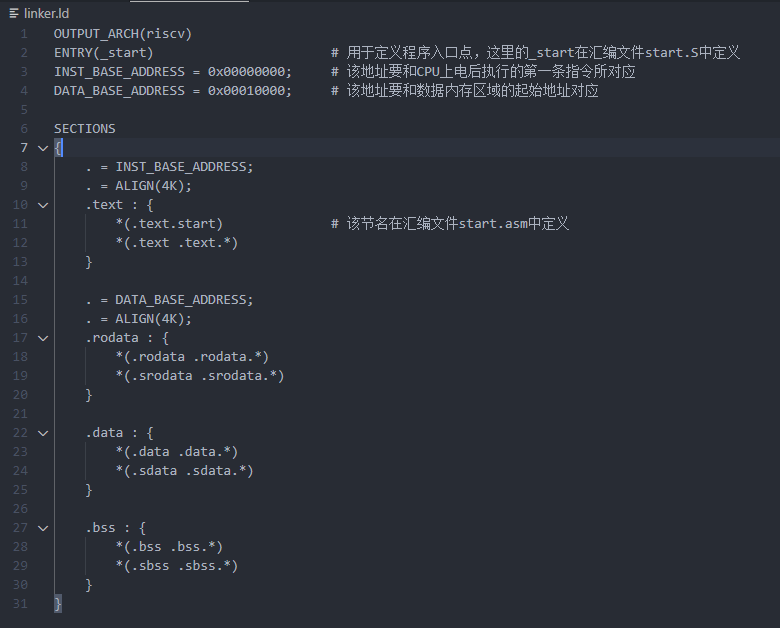


代码的关键点如下：

* 代码第1行声明常量：表示栈底位置为0x00013ff0，由于我们的CPU数据内存映射的地址范围为0x00010000~0x00014000，所以这里我设置了栈地址为0x00013ff0，这个地方**需要根据实际情况修改**。
* 代码第3行表示之后的部分属于名为.text.start节中；代码第4行表示标号\_start可以被其他文件可见。**这些内容会在后续的链接脚本中使用到**。
* 代码第6行设置栈指针；代码第7行调用main函数，这个main函数对应C语言中的main函数，这里main可以更换为其他名字，但要与C语言中的入口函数相互对应。
* 代码第10行的作用是防止main函数调用返回后继续执行非预期指令。

### 5.5.3使用链接脚本调整程序布局

这部分内容位于linker.ld中。主要功能是描述如何将输入文件的节区映射到输出文件的节区中，同时对输出文件的存储布局进行控制。



脚本的关键点如下：

* 脚本第1行指定输出代码的指令集架构为riscv。
* 脚本第2行指定程序入口点为\_start，与start.S中的\_start标号所对应。
* 脚本第3行和第4行定义常量，分别表示代码区（也就是.text部分）的起始位置和数据区（对应于.rodata、.data、.bss部分）的起始位置，**这部分要根据自己的CPU设计进行相应调整**。
* 脚本第11行表示名为.text.start位于代码区的最前面，与start.S中的节名所对应。
* 其他部分感兴趣的可以自行查找资料来理解，这里就不再赘述。

### 5.5.4 Makefile文件

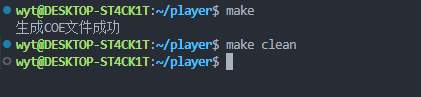
为了方便操作，编写了一个Makefile文件。在使用之前需要进行部分修改来使之正常运行。



* 确保交叉编译工具链的可执行文件路径确实如第5行所示。
* **当C语言源文件增加时，确保被第13行所包含**。
* 第45行指定了python解释器的命令，需要根据自己电脑情况进行修改。
* python需要安装pyelftools库：在命令行中输入pip install pyelftools进行安装。

在命令行中输入如下命令运行Makefile：

* make：生成COE文件
* make clean：清除生成的中间文件和COE文件



# 设计与实现

## 6.1 蜂鸣器播放音乐模块实现

蜂鸣器播放音乐主要通过两个verilog模块实现，分别是buzzer模块和music模块。buzzer模块负责接受输入的音调发出相应的声音，和节拍时序均无关。music模块中实例化了buzzer模块，其中具有音乐的缓存，负责控制音乐的暂停继续和节拍等音乐相关的部分。

buzzer模块的核心部分在于根据确定的半周期时钟数输出相应的频率。下面展示关键的代码，根据信号选择时钟数的部分省略：

module buzzer(

input clk, input rstn, input [1:0]level, input [3:0]tone, output buzzer\_out );

reg pwm; // output of the buzzer

reg state; // state of the buzzer (1: on, 0: off)

reg [31:0]cnt; // counter for the tone

reg [31:0]half\_t\_clk; // half period of the tone

......

**always@**(posedge clk or negedge rstn) begin

if (~rstn) begin

pwm <= 0; cnt <= 0;

end

else if (state) begin

if (cnt >= half\_t\_clk) begin

pwm <= ~pwm; cnt <= 0;

end

else begin

cnt <= cnt + 1;

end

end

else begin

cnt <= 0; pwm <= 0;

end

end

assign buzzer\_out = pwm;

endmodule

music模块的核心在于音乐的缓存和音乐播放设置：

// 内存中存储的音乐信息

reg [7:0] music\_memory [0:128];

// 初始化音乐内存（初始为1首音乐）

initial begin

music\_memory[0] = 8'd64;

music\_memory[1] = 6'b01\_1010; //l6

......//初始化为其他音符

// 控制逻辑

**always @**(posedge clk or negedge rstn) begin

if (~rstn) begin

music\_index <= 1;

song\_start\_index <= 1;

song\_end\_index <= 64;

level <= 0;

tone <= 0;

timer <= 0;

end

else if (restart) begin

song\_end\_index <= music\_memory[0];

music\_index <= song\_start\_index;

end

else if (w\_en) begin

music\_memory[w\_addr] <= w\_data;

end

else if (timer >= 31'd2500\_0000) begin

timer <= 0;

if (playing) begin

if (music\_index > song\_end\_index) begin

music\_index <= song\_start\_index; // 播放结束，循环播放当前歌

end

else begin

level <= music\_memory[music\_index][5:4];

tone <= music\_memory[music\_index][3:0];

music\_index <= music\_index + 1;

end

end

else begin

level <= 0;

tone <= 0;

music\_index <= music\_index;

end

end

else begin

timer <= timer + 1;

end

end

## 6.2 内存映射I/O模块实现

本实验的MMIO模块通过**统一编址**技术实现了外设和内存的访问。通过将外设的寄存器映射到特定的内存地址空间，CPU能够使用标准的`load`和`store`指令直接操作外设，而无需使用专门的I/O指令。以下是MMIO模块的Verilog代码：

module MMIO(

input clk,

input rst,

input write\_enable, // 写使能

input [31:0] addr, // 读写内存地址

input [2:0] len, // 读写的长度 byte0 / half1 / word2 / u\_byte4 / u\_half5

input [31:0] write\_data, // 写入的数据

output reg [31:0] read\_data, // 读出的数据

output reg [31:0] display\_data, // 7 段数码管显示的数据

// 连接外设的端口

input [5:0] button, // 6个游戏按键

output reg [7:0] lcd\_data, // LCD数据

output reg [4:0] lcd\_control, // LCD控制信号

output reg [7:0] led, // 8个LED灯

output buzzer // 蜂鸣器

);

### 6.2.1 数据存储器和外设寄存器的映射

在代码中，我们通过定义内存映射的起始和结束地址来区分数据存储器和外设寄存器。例如，数据存储器的地址范围从`0x0000\_0000`到`0x0000\_4000`，而外设的地址则被映射到更高的地址范围。以下是部分内存映射地址的定义：

// 内存映射

parameter DATA\_MEM\_START = 32'h0000\_0000;

parameter DATA\_MEM\_END = 32'h0000\_4000;

parameter LCD\_DATA\_ADDR = 32'hffff\_0000;

parameter LCD\_CONTROL\_ADDR = 32'hffff\_0004;

parameter LED\_ADDR = 32'hffff\_1000;

parameter BUTTON\_ADDR = 32'hffff\_2000;

parameter CLK\_ADDR = 32'hffff\_3000;

parameter DISPLAY\_ADDR = 32'hffff\_4000;

parameter MUSIC\_CONTROL\_ADDR = 32'hffff\_5000;

### 6.2.2 时序控制逻辑

在时钟信号`clk`的上升沿或复位信号`rst`的下降沿，MMIO模块会更新外设的状态。具体的操作逻辑如下：

**always @**(posedge clk or negedge rst) begin

if(!rst) begin

lcd\_data <= 8'h0;

lcd\_control <= 5'h0;

led <= 8'h0;

clk\_counter <= 32'h0;

display\_data <= 32'h0;

music\_control\_reg <= 3'h0;

end else begin

clk\_counter <= clk\_counter + 1'b1;

if(write\_enable) begin

// 外设寄存器映射的写操作

if (addr == LCD\_DATA\_ADDR) begin

lcd\_data <= write\_data[7:0];

end else if(addr == LCD\_CONTROL\_ADDR) begin

lcd\_data <= write\_data[4:0];

end

else if (addr == LED\_ADDR) begin

led <= write\_data[7:0];

end

else if (addr == DISPLAY\_ADDR) begin

display\_data <= write\_data;

end

else if (addr == MUSIC\_CONTROL\_ADDR) begin

music\_control\_reg <= write\_data[2:0];

end

end

end

end

在上面的代码中，当写使能信号`write\_enable`有效时，根据访问的地址决定将写入的数据发送到相应的外设寄存器。例如，向地址`LCD\_DATA\_ADDR`写入数据，将更新LCD显示屏的数据；向地址`LED\_ADDR`写入数据，将更新LED灯的状态。

### 6.2.3 数据读写逻辑

MMIO模块通过地址解码区分内存和外设的读写操作。对于不同的地址范围，模块执行不同的操作。例如，读取按键状态时，访问的是`BUTTON\_ADDR`地址；而读取当前时钟周期数时，访问的是`CLK\_ADDR`地址。以下是部分读操作的代码实现：

**always @**(\*) begin

if(addr >= DATA\_MEM\_START && addr < DATA\_MEM\_END) begin

read\_data <= data\_mem\_read\_data;

end

else if (addr == LCD\_DATA\_ADDR) begin

read\_data <= {24'b0, lcd\_data};

end

else if(addr == LCD\_CONTROL\_ADDR) begin

read\_data <= {27'b0, lcd\_control};

end

else if (addr == LED\_ADDR) begin

read\_data <= {24'b0, led};

end

else if (addr == BUTTON\_ADDR) begin

read\_data <= {26'b0, button};

end

else if (addr == CLK\_ADDR) begin

read\_data <= clk\_counter;

end

else if (addr == DISPLAY\_ADDR) begin

read\_data <= display\_data;

end

else if (addr == MUSIC\_CONTROL\_ADDR) begin

read\_data <= {29'b0, music\_control\_reg};

end

else begin

read\_data <= 32'h0;

end

end

在上面的代码中，不同的外设地址对应不同的读取方式。例如，从`BUTTON\_ADDR`读取按键状态，从`CLK\_ADDR`读取时钟计数器的值。

通过代码的展示，我们可以清晰地看到MMIO模块如何通过地址解码，将CPU的读写请求分配到不同的外设寄存器或数据存储器。这样，CPU可以直接使用标准的`load`和`store`指令与外设交互，实现对外设的高效控制。

## 6.3 软件代码实现

本项目实现了一个音乐播放器，除了上述关键外设控制器使用verilog编写外，总体控制部分使用C语言编程。其中`main`程序负责调用各个模块的接口，协调程序的整体运行逻辑。主程序通过调用`show`、`music`和`base\_api`模块来实现启动页面的绘制、音乐的播放与控制、以及基础外设的操作（例如LCD、LED和按键检测）。

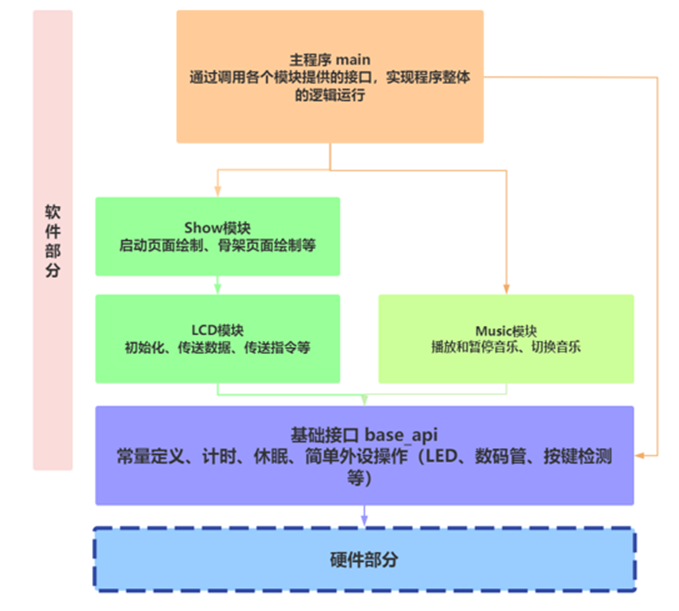
### 6.3.1 模块概述

1. **`show`模块**：负责启动页面和骨架页面的绘制，通过调用`lcd`模块实现具体的显示功能。`lcd`模块负责初始化LCD、传送数据和指令等操作。

2. **`music`模块**：负责音乐的播放、暂停和切换，同样依赖于`base\_api`模块提供的底层接口，来实现音乐的控制和外设交互。

3. **`base\_api`模块**：提供了基础的硬件操作接口，如按键检测、LED控制、定时、休眠等功能。`base\_api`模块使得上层模块可以轻松与硬件交互，而无需关心底层细节。

模块如下图所示



### 6.3.2 main程序实现

主程序首先通过调用`InitLcd()`来初始化LCD显示屏，接着通过`ShowLaunchImage()`函数在LCD上展示启动页面。启动页面绘制完毕后，程序进入一个主循环，等待用户的输入并根据输入执行相应的操作。

#include "base\_api.h"

#include "lcd.h"

uchar gImage\_test[2048] = { / 启动页面图像数据 / };

#define PLAY\_PAUSE\_BTN\_MASK (0b00000001)

#define LEFT\_BTN\_MASK (0b00000010)

#define RIGHT\_BTN\_MASK (0b00010000)

#define MAX\_MUSIC\_INDEX (5)

void ShowLaunchImage() {

ShowImageOnLcd(gImage\_test);

SleepMs(1000); // 显示启动页面1秒

}

部分代码通过调用`ShowImageOnLcd()`函数将图片数据传递给LCD进行显示，随后调用`SleepMs(1000)`使程序暂停1秒，以便用户能够看到启动页面。

在启动页面展示完毕后，主程序进入主循环。程序通过`CheckButtonPress()`函数来检测按键输入，并根据输入的不同控制音乐播放或暂停、切换音乐等操作。

void player\_main() {

InitLcd(); // 初始化LCD

ShowLaunchImage(); // 显示启动页面

int is\_playing = 0; // 记录当前播放状态

int music\_index = 0; // 当前音乐索引

while (1)

{

// 检测播放/暂停按钮按下

if (CheckButtonPress(PLAY\_PAUSE\_BTN\_MASK)) {

if (is\_playing) {

PauseMusic(); // 暂停音乐

is\_playing = 0;

} else {

PlayMusic(); // 播放音乐

is\_playing = 1;

}

}

// 检测左键按下（切换到上一首音乐）

if (CheckButtonPress(LEFT\_BTN\_MASK)) {

// 实现切换音乐的逻辑

}

// 检测右键按下（切换到下一首音乐）

if (CheckButtonPress(RIGHT\_BTN\_MASK)) {

// 实现切换音乐的逻辑

}

}

return;

}

1. **播放与暂停**：程序通过检测`PLAY\_PAUSE\_BTN\_MASK`按键来决定是否播放或暂停音乐。按键检测通过调用`CheckButtonPress()`函数完成，当检测到按键被按下时，程序会根据当前播放状态决定是调用`PlayMusic()`还是`PauseMusic()`。

2. **音乐切换**：主程序还会检测左键和右键（分别对应`LEFT\_BTN\_MASK`和`RIGHT\_BTN\_MASK`）是否被按下，用于实现音乐的切换。切换逻辑将在后续实现中进行完善。

### 6.3.3 LCD模块

在音乐播放器项目中，LCD 模块负责初始化 LCD 显示屏、绘制图像、清屏、满屏操作等基本功能。该模块与`base\_api`模块交互，提供了对 LCD 硬件的控制接口，以确保其他模块能够有效使用显示功能。LCD 模块的核心功能包括命令和数据的传输、LCD 的初始化以及图像显示控制。

#### 6.3.3.1 功能概述

1. **初始化 LCD (`InitLcd`)**：初始化是使用 LCD 的前提操作，程序通过一系列命令设置 LCD 的工作模式、对比度、电源控制等，并确保显示功能能够正常运作。

2. **命令传输 (`TransferCommandToLcd`)**：该函数通过串行通信将控制命令发送到 LCD，用于调整显示的模式、扫描顺序等。

3. **数据传输 (`TransferDataToLcd`)**：该函数用于向 LCD 传送像素数据，以实现图像或其他显示内容的更新。

4. **地址设置 (`SetLcdAddress`)**：用于设置当前要操作的 LCD 页（Page）和列（Column），从而控制像素数据的显示位置。

5. **清空屏幕 (`ClearLcdScreen`)**：通过发送空数据（0x00）实现整个 LCD 屏幕的清空。

6. **满屏显示 (`FullLcdScreen`)**：该功能用于将 LCD 屏幕填满，通常用于测试或特定界面的显示。

7. **显示图像 (`ShowImageOnLcd`)**：负责将指定的图像数据按照页和列的地址绘制在 LCD 上。

#### 6.3.3.2 实现细节

1. **初始化过程**

LCD 初始化通过发送多个指令来配置显示屏。该过程涉及重置硬件、设置显示模式、调整对比度等。

void InitLcd() {

ClearRST();

SleepUs(500); // 确保 RST 信号的持续时间

SetRST();

SleepUs(100); // 等待硬件复位

TransferCommandToLcd(0xae); // 关闭显示

TransferCommandToLcd(0x38); // 设置显示模式

TransferCommandToLcd(0xaf); // 开启显示

}

2. **命令与数据传输**

数据和命令的传输采用串行通信方式。每次传输 8 位数据，通过移位操作依次将数据传递给 LCD。

void TransferCommandToLcd(uchar command) {

ClearCS();

ClearRS();

for (int i = 0; i < 8; i++) {

WriteLcdData((command & 0x80) ? 0xff : 0x00);

command <<= 1;

}

SetCS();

}

3. **显示图像**

`ShowImageOnLcd`函数将图像数据按页和列的方式分块传输到 LCD，并最终显示在屏幕上。该函数通过循环处理 16 页的图像数据并将其传递给显示器。

void ShowImageOnLcd(uchar\* image\_data) {

for (int page = 0; page < 16; page++) {

SetLcdAddress(page, 0);

for (int column = 0; column < 128; column++) {

TransferDataToLcd(image\_data[page \* 128 + column]);

}

}

}

### 6.3.4 Base API 模块

在音乐播放器项目中，Base API 模块负责对硬件外设的底层操作提供接口支持，包括 LCD 数据传输、按键检测、LED 控制、音乐播放控制等功能。通过这一模块，主程序和其他上层模块能够与硬件外设进行通信与控制，使系统整体功能得以实现。

#### 6.3.4.1 功能概述

1. **时间控制**

* `SleepUs` 和 `SleepMs` 函数提供微秒级和毫秒级的延迟控制，确保在硬件操作过程中适当的时序。

2. **外设操作**

* 提供了一系列函数用于与 LCD、LED、按钮、音乐控制器等硬件外设的交互，如 `WriteLcdData`、`WriteLed`、`CheckButtonPress`、`PlayMusic` 等。

3. **硬件寄存器操作**

* 基于内存映射 I/O 机制，Base API 模块通过直接读写特定的内存地址实现对外设寄存器的操作。例如通过向特定地址写入数据来控制 LCD 显示、LED 状态和数码管输出等。

#### 6.3.4.2 实现细节

1. **时间控制**

`SleepUs` 和 `SleepMs` 函数用于延迟指定的时间，在 LCD 初始化、按键检测等功能中，确保硬件操作的时序精度。`SleepUs` 函数通过读取 CPU 时钟计数器 (`CLK\_COUNTER`) 并计算延迟的时间来实现微秒级别的休眠。

void SleepUs(int us)

{

uint start\_counter = ((uint )CLK\_COUNTER);

uint end\_counter = start\_counter + us \* CPU\_TIMES\_PER\_US;

while (1)

{

uint curr\_counter = ((uint )CLK\_COUNTER);

if (curr\_counter >= end\_counter || curr\_counter < start\_counter)

{

break;

}

}

}

**LCD 操作**

`WriteLcdData` 函数通过向 `LCD\_DATA\_ADDR` 地址写入数据，实现向 LCD 传送显示数据。这是 LCD 操作的核心步骤，负责将像素信息传递给显示屏。

inline void WriteLcdData(uchar data)

{

((uchar )LCD\_DATA\_ADDR) = data;

}

**按键检测**

通过 `CheckButtonPress` 函数，系统可以检测特定按键是否被按下。该函数读取 `BUTTON\_ADDR` 地址的值，根据传入的 `mask` 参数进行按键状态检测。

int CheckButtonPress(uchar mask) {

uchar btn = ((uchar )BUTTON\_ADDR);

if (btn & mask) {

((uchar )BUTTON\_ADDR) = btn & ~mask; // 清除按键状态

return 1;

}

return 0;

}

**音乐控制**

Base API 提供了简单的音乐控制功能。通过写入 `MUSIC\_CONTROL\_ADDR` 地址，可以开始或暂停音乐播放。

inline void PlayMusic() {

((uint ) MUSIC\_CONTROL\_ADDR) = 0x1; // 播放音乐

}

inline void PauseMusic() {

((uint ) MUSIC\_CONTROL\_ADDR) = 0x0; // 暂停音乐

}

**其他外设控制**

其他外设如 LED、数码管的控制通过直接向特定地址写入数据来实现。以下是控制 LED 和显示调试信息的示例：

inline void WriteLed(uchar led)

{

((uchar )LED\_ADDR) = led; // 控制LED灯状态

}

inline void PrintData(uint data)

{

((uint )DISPLAY\_ADDR) = data; // 在数码管上显示数据

}

Base API 模块为上层模块与底层硬件的交互提供了关键支持，通过定义统一的接口，简化了与各类外设的交互操作。尤其在音乐播放、按键控制、显示控制等功能上，Base API 模块通过内存映射 I/O 操作，实现了对外设的灵活控制，使整个系统能够高效运行。

# 测试

## 仿真测试用例、结果以及分析

在设计和实现外设控制模块后，我们进行了两次仿真测试，以确保外设接口的正确性以及CPU与外设的交互无误。

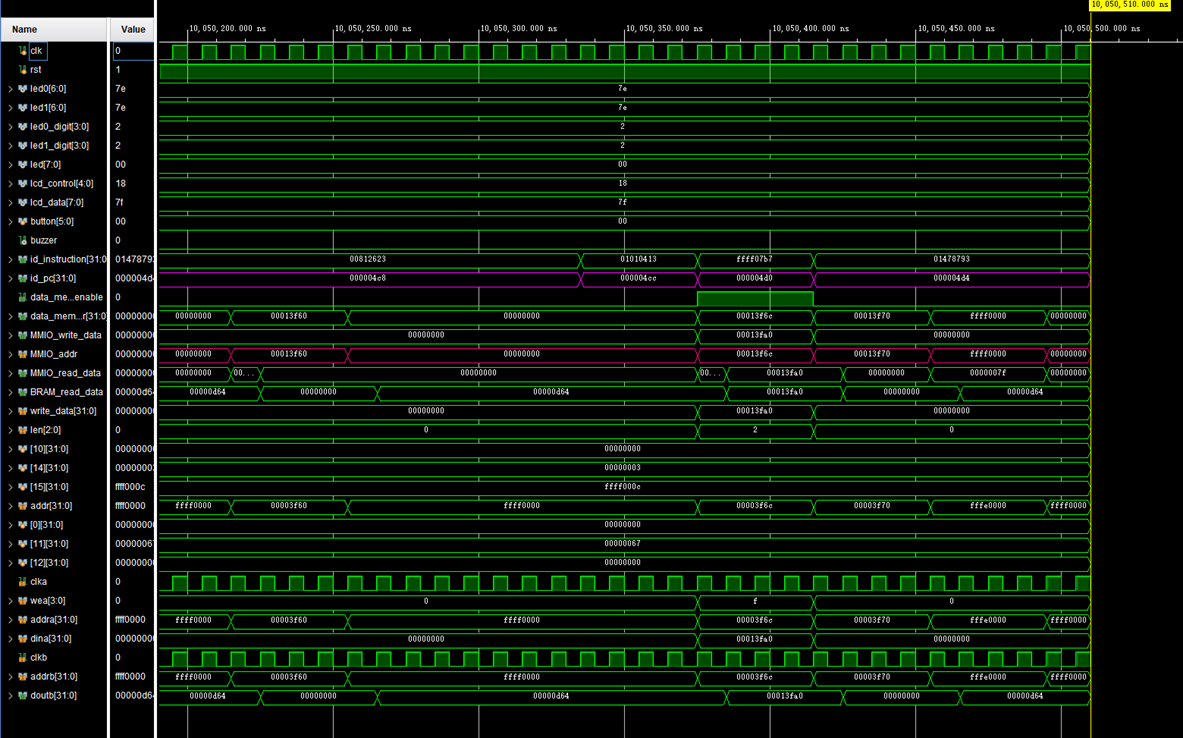
**1. 外设模块的单独仿真**

我们首先对蜂鸣器和LCD这两个外设模块进行了独立仿真测试。通过单独仿真，我们验证了每个模块的基本功能是否正常，例如蜂鸣器的声音控制和LCD的显示更新。仿真中，我们精确控制输入信号，并观察外设模块的响应，确保其与预期一致。这一步的仿真确保了外设模块在独立运行时的正确性，为后续与CPU的整合打下了基础。

**2. 音乐播放器的行为仿真**

在完成外设与CPU的整合后，我们进行了最终音乐播放器的行为仿真。在这一阶段，我们通过播放音乐的节拍控制、暂停与继续操作、LCD界面更新等多项功能来验证外设与CPU的交互是否正常。仿真结果表明，CPU能够准确控制外设的操作，外设也能及时响应来自CPU的指令。这一仿真验证了整个系统，包括外设接口和CPU控制逻辑的正确性。

通过这两次仿真，我们确保了外设与CPU的无缝集成，为整个音乐播放器的正常运行提供了保障。



## ILA 技术在开发板上的调试与仿真

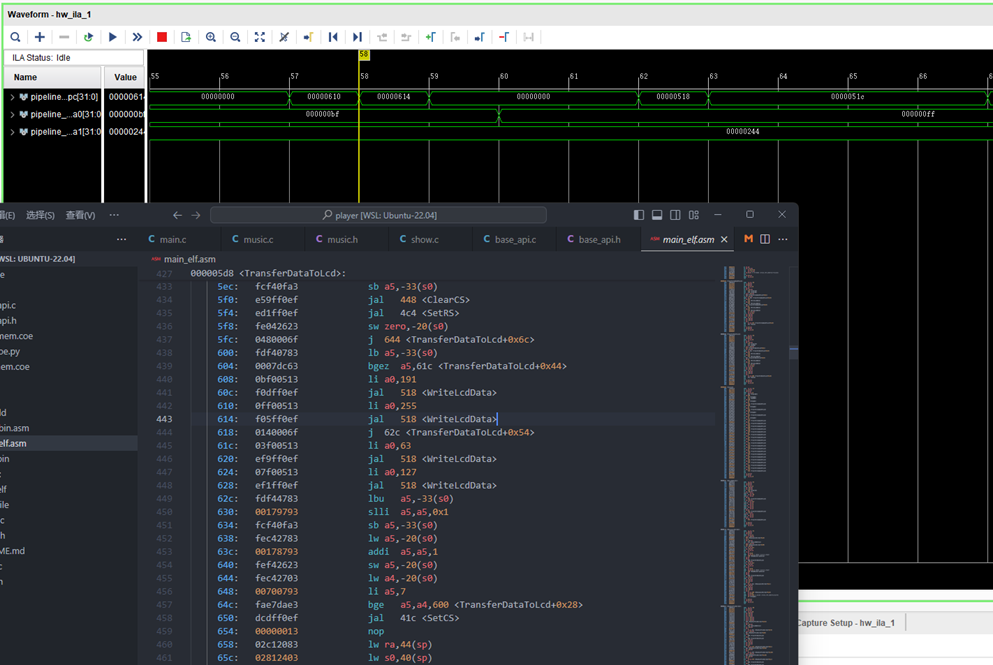
在完成了仿真测试之后，我们进一步通过在开发板上使用 Vivado 的 Integrated Logic Analyzer (ILA) 技术，对整个系统进行实时调试与跟踪。这种调试方式能够帮助我们深入分析系统运行中的具体情况，特别是在函数调用过程中的数据传递和执行路径。

**1. ILA 技术调试概述**

Vivado ILA 是一种嵌入式调试工具，允许我们在硬件运行时监控 FPGA 中信号的状态。我们将 ILA 核心集成到硬件设计中，并通过预先设置的触发条件，对特定时刻的信号进行捕捉和分析。通过这种方法，可以实时追踪系统在不同指令执行时的具体行为，并在开发板上直接进行调试。

**2. 函数调用过程跟踪**

在调试过程中，我们重点对关键函数 `WriteLcdData` 进行跟踪，以验证其调用过程是否正确。该函数主要负责向 LCD 模块传输数据。在具体的调试中，我们跟踪了 CPU 执行地址 `0x614` 处的指令，该指令会调用 `WriteLcdData`，并传递参数 255 (即 0xFF)。通过设置 ILA 的触发条件，当 CPU 执行该指令时，系统会捕获并显示相关信号信息。

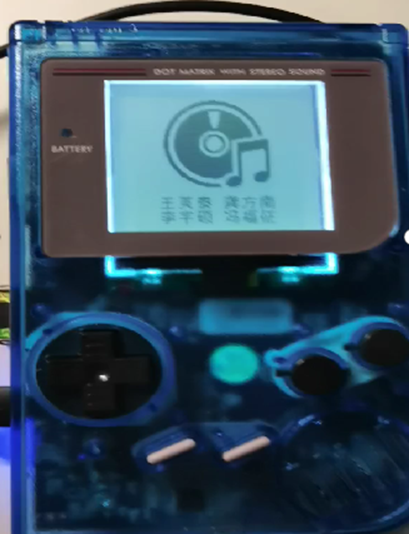
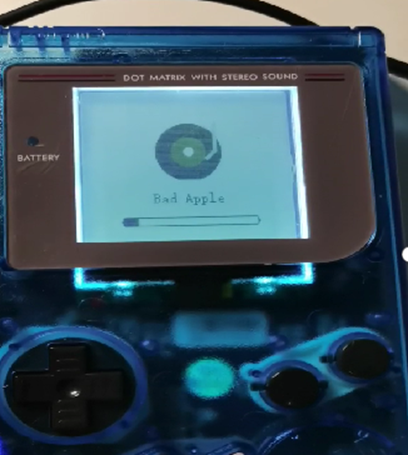


**3. 结果与分析**

通过 Vivado 的 ILA 窗口，我们能够看到 CPU 在执行 `0x614` 处指令时，成功调用了 `WriteLcdData` 函数，并将 255 作为参数传递给该函数。Vivado 显示窗口正确显示了此次函数调用的具体细节，特别是在数据总线上，观察到传入参数 `0xFF` 被准确地发送到 LCD 模块。通过这种实时调试，我们确认了函数调用链的正确性，并验证了系统在实际硬件上的运行情况。

总体而言，使用 Vivado 的 ILA 技术进行调试，极大地提升了我们对系统行为的把控能力，帮助我们快速发现问题并修复，确保了系统的稳定性与正确性。

初始界面： 音乐播放界面：

# 问题及解决方法

在实现外设控制器的过程中，我们遇到了一些关键问题，主要集中在**内存读写对齐**、**时序控制**、以及**Verilog编码规范**等方面。通过不断的调试与改进，最终成功解决了这些问题，保证了外设控制器的正确运行。下面是问题的详细分析与解决方案。

### 1. 内存读写对齐问题

内存读写对齐是一个非常关键但又容易被忽视的问题。在实验中，我们使用了板载 BRAM（Block RAM）作为存储设备，该 BRAM 的读写是 **4 字节对齐**（即按字编址）。然而，在最初的设计中，我们以字节为单位进行地址访问，这就导致了多个操作可能试图读写同一内存块，从而引发了数据冲突和内存访问错误。

解决这个问题的关键是确保内存读写的操作按字对齐，而不是以字节为单位进行访问。通过修正地址的计算方式，避免了误读误写，确保了存储器的访问正确性。

### 2. 时序控制问题

时序控制是整个项目的核心难点之一，尤其在内存读写以及 CPU 与外设交互时，时序问题尤为突出。最初设计中，由于忽略了内存的读写周期，导致了数据读取错误。BRAM 的读写操作并非瞬时完成，而需要多个时钟周期。在我们初始设计中，CPU 的指令周期与内存操作未能很好地同步，进而引发了数据访问的失败。

为了解决这个问题，我们将 **CPU 的指令周期分频**，使其比原始时钟周期延长了数倍。这样就确保了在内存操作完成之前，CPU 不会继续执行下一条指令，保证了时序的正确性。此外，外设与 CPU 之间的交互，如音乐播放中的节拍控制，也依赖于时钟的精确控制。我们为此设计了时钟分频机制，确保外设工作在正确的时序下。

### 3. Verilog 编码规范问题

在 Verilog 编码过程中，不规范的代码是导致调试困难的重要原因之一。Verilog 编译器对许多编码问题并不会报错，这增加了调试的难度，尤其是当代码执行不符合预期时。因此，遵循 Verilog 的编码规范对于确保正确性和调试效率至关重要。

例如，变量必须 **先定义再引用**，以避免未定义的行为。我们还发现，在 Verilog 中同一个 `reg` 信号不能在多个 `always` 块中赋值，否则会导致综合工具无法推断出正确的逻辑电路。此外，在 `always` 块中应尽量使用 **非阻塞赋值** (`<=`)，以确保信号的时序行为正确。非阻塞赋值能够避免在同一时钟周期内由于赋值顺序不同引发的竞争条件。

通过遵循这些编码规范，我们确保了模块的正确性，避免了综合和实现阶段的逻辑丢失，同时提升了代码的可读性和调试效率。

以上问题及其解决过程为我们在设计外设控制器时提供了宝贵的经验。内存读写对齐、时序控制的精确把握、以及 Verilog 编码的规范性，不仅是这次项目中的挑战，也是我们未来数字电路设计中必须时刻注意的关键点。通过解决这些问题，我们不仅提升了代码的稳定性，也加深了对数字系统设计的理解。

# 心得体会及总结

在进行汇编语言与接口课程设计中，我们实现了一个简易的“音乐播放器”，完成了LCD显示播放状态和图片、蜂鸣器播放音乐、LED根据音乐进行闪烁、按键控制暂停/继续、上一首、下一首，数码显示等功能。实验中，我们实现了多个外设接口控制器，包括verilog实现的蜂鸣器buzzer模块和music模块以及C语言实现的LCD显示模块。最终，我们使用C语言对整体计算机系统进行编程并实现了全部的音乐播放功能。

同时，我们在实验中也遇到了很多问题导致仿真波形不符合预期以及下板失败等。我们通过波形对比调试，开会讨论以及不断的仿真和调试后终于完成了全部的工作，前前后后花费了10天左右的时间。实际上，我们大部分的时间都花费在了调试和下板的环节，然而最终解决问题时才发现可能是一个简单的编码错误或者是忽略了微小但重要的细节。

通过这次实验，我们不仅对FPGA编程、外设控制器设计实现乃至于计算机系统实现的相关知识有了更深刻的理解，我们也对我们的开发流程和分工进行了总结和反思。对于一个工程项目，我们需要密切的团队配合和细致整体的设计讨论。将问题在出现之前解决能够大幅减少调试的时间。在未来，我们应该更加注重整体设计和细节考量，才能在更大更复杂的任务中表现的更好。

最后，感谢王娟老师对我们实验的指导和耐心细致的答疑。在您的讲解下，我们明确了任务的划分和实现的层次，对实验要求有了更加明确的理解，也最终实现了完整的计算机系统。再次感谢您的指导！

# 参考文献有价值的资源推荐

1. 乐学资源EES338 lab01-10
2. JLX128128G-81202-PN中文说明书
3. EES-338\_UserManual\_v1.0
4. RISCVGreenCardv8-20151013（riscv指令格式手册）