

清华大学
计算机组成原理

THCO MIPS16e

计算机系统设计与实现

报告

作者

计 XX XXX 201XXXXXXX
计 XX XXX 201XXXXXXX

目录

1 概述	1
1.1 系统概要	1
1.2 组内分工	2
2 设计与实现	3
2.1 总体架构	3
2.2 硬件部分	3
2.2.1 内部总线设计	4
2.2.2 处理器	4
2.2.3 SRAM 与 UART	4
2.2.4 显示控制器	5
2.2.5 PS/2 控制器	5
2.2.6 SD 卡控制器	5
2.2.7 GPIO	10
2.3 软件部分	10
2.3.1 汇编器	10
2.3.2 PS/2 键盘驱动	10
2.3.3 上电自检程序	10
2.3.4 shell	10
2.3.5 2048 小游戏	11
2.3.6 BadApple!!动画播放	12
2.3.7 BadApple!!的弹幕	12
2.3.8 内存转储	12
2.4 内存地址分配	12

3 展望	13
4 参考文献	14

第 1 部分

概述

1.1 系统概要

我们实现了一个支持中断与异常、五级流水线、兼容 THCO MIPS16e 指令集架构的计算机系统。

具体而言，在硬件方面，我们的计算机系统具有如下特性：

- **处理器** 最高可以运行在 40MHz，可达 40MIPS，CPI=1.01。字长 16 位。
- **内存** 由于 16 位地址总线宽度限制，内存寻址只支持 64K 字，即 128KiB。
- **中断** TODO
- **显示控制器** 支持 640x480 @ 60Hz（工业标准）的 VGA 信号输出，可以显示 80x30 个 ASCII 字符，每个字符支持 4 种前景色和 4 种背景色组合。
- **PS/2 接口** 该接口可以配合驱动程序来支持 PS/2 键盘和鼠标。
- **SD 卡存储** 这一次，我们突破了以往对于 SD 卡使用的限制，成功实现了用市面流行的 SDHC 储存卡存储数据的功能。支持 DMA，减轻处理器压力。
- **GPIO** 结合驱动程序，可以实现 SPI 总线协议、I2C 总线协议、软件串口和点亮数码管等功能，扩展性较好。

在软件方面，我们的计算机系统有如下软件设施：

- **汇编器** 实验材料中提供的汇编器不易于使用，我们重新用 Python 3 实现了一个。支持伪指令（如 la 以及 li），以及立即数大小检查和报错。
- **PS/2 键盘驱动** 目前，我们只实现了 PS/2 键盘的驱动程序。能够接收键盘发来的数据，并识别通码和断码，转换为 ASCII 码。
- **shell** 支持从键盘读取命令然后执行相应的代码段。
- **2048 小游戏** 通过字符画的形式实现了一个类似于 2048 的小游戏，支持保存和加载游戏局面。
- **BadApple!!动画播放** 通过读取 SD 卡中存储的每一帧图像信息并显示到屏幕来播放动画，用来验证 SD 卡和显示控制器的功能。
- **弹幕** 支持在播放动画的时候显示弹幕，用时钟中断实现，用来验证处理器的中断和异常功能是否实现正确。

- **内存转储** 支持将内存转储到 SD 卡，进行进一步的 debug 工作，也用来验证 SD 卡的写入功能。
- **上电自检程序** 开机的时候会被执行，主要用来检测内存是否有错误，或者系统时钟频率是否过高（导致内存访问出错）。

1.2 组内分工

组内二人的分工如下：

谭闻德

- 数据通路的绘制
- 完整流水线搭建
- 前期系统仿真
- Store After Load 优化的构思
- 内部总线设计
- GPIO 控制器设计与实现
- VGA 控制器设计与实现
- SD 卡控制器设计与实现
- 汇编器的设计与实现
- POST (上电自检) 程序的汇编实现
- PS/2 驱动程序的汇编实现
- BadApple!!动画播放程序的汇编实现
- shell 的汇编实现

刘明华

- 数据通路的绘制
- 控制信号表的总结
- 指令译码模块的实现
- Store After Load 优化的实现
- VGA 控制器的实现
- PS/2 控制器设计与实现
- PS/2 驱动程序、读取和显示字符串函数的汇编实现
- 2048 游戏的汇编实现
- BadApple!!的弹幕程序的汇编实现

第2部分

设计与实现

2.1 总体架构

2.2 硬件部分

整体数据通路如图2.1所示。

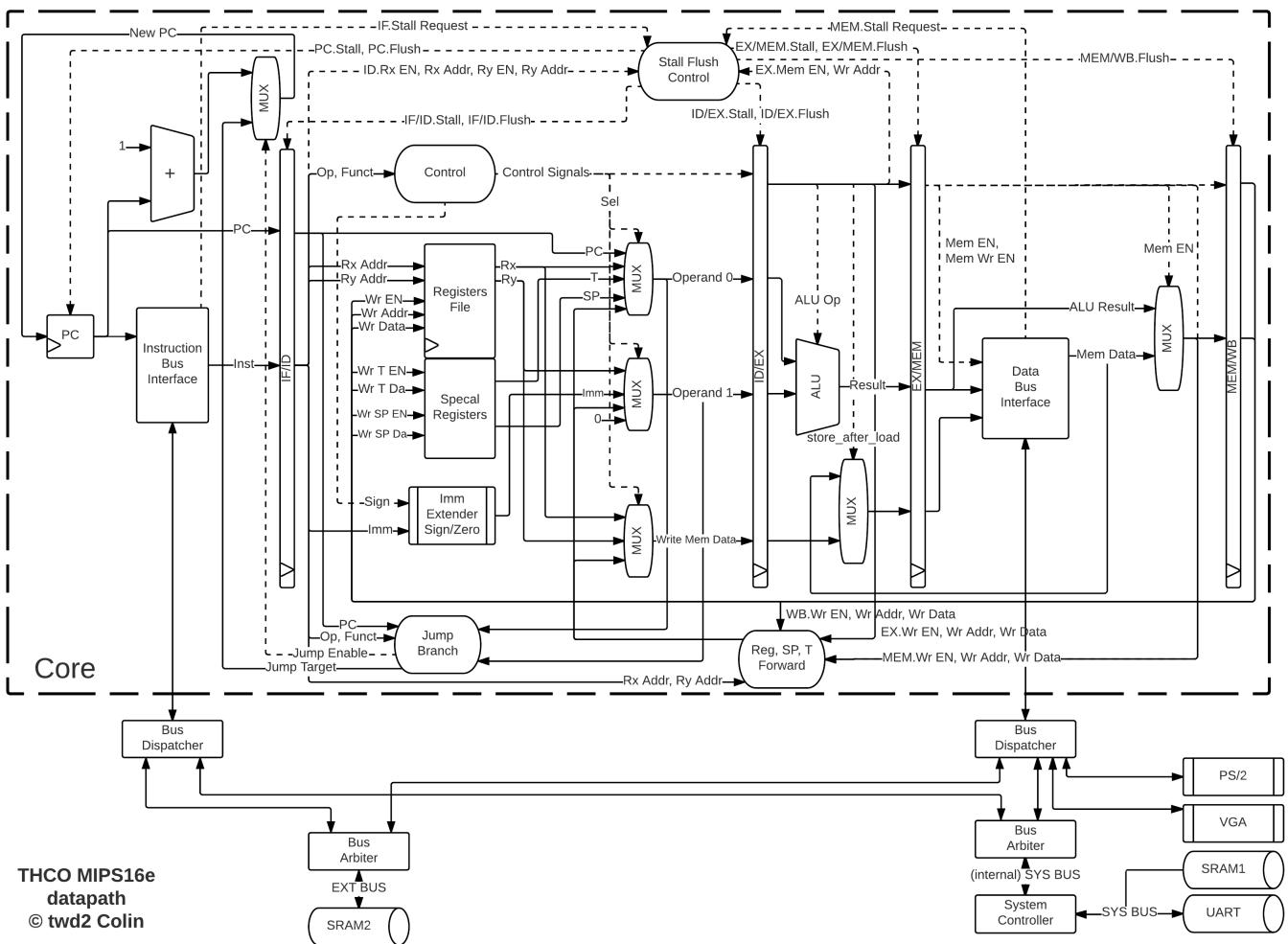


图 2.1: 数据通路

	op	fun_c_p_o_s	func	sub_f_u_n_e_nc_pos	sub_f_u_nc	imm_pos	imm_e_xtend	rea_d_a_ddr_0_buf_f	rea_d_a_ddr_1_buf_ff	rea_d_e_n_0_buf_ff	rea_d_e_n_1_buf_ff	alu_op	operand_0	operand_1	wri_te_en	wri_te_r	mem_en	mem_wr_en	write_mem_data	t_w_e_en	sp_e_en	hi_lo_en	BRANCH_EN	BRANCH_PC	IS_L_OAD	cp0_rea_d_addr	cp0_rea_d_en
NOP	00001							rx	ry	1	1	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	0
B	00010					10:0	sign	rx	ry	0	0	X	X	X	0	X	0	X	X	0	0	X	X	1	b_pc	0	X
BEQZ	00100					7:0	sign	rx	ry	1	0	X	X	X	0	X	0	X	X	0	0	X	X	reg_0_eq_0	cb_pc	0	X
BNEZ	00101					7:0	sign	rx	ry	1	0	X	X	X	0	X	0	X	X	0	0	X	X	not reg_0_eq_0	cb_pc	0	X
SLL	00110	1:0 00				4:2	shift	rx	ry	0	1	alu_sll	READ_DATA_1	shamt_buff	1	rx	0	X	X	0	0	X	X	0	X	0	X
SRA	00110	1:0 11				4:2	shift	rx	ry	0	1	alu_sra	READ_DATA_1	shamt_buff	1	rx	0	X	X	0	0	X	X	0	X	0	X
ADDIU3	01000					3:0	sign	rx	ry	1	0	alu_addu	READ_DATA_0	imm4se	1	ry	0	X	X	0	0	X	X	0	X	0	X
ADDIU	01001					7:0	sign	rx	ry	1	0	alu_addu	READ_DATA_0	imm8se	1	rx	0	X	X	0	0	X	X	0	X	0	X
ADDSP	01100	10:8 011				7:0	sign	rx	ry	0	0	alu_addu	SP	imm8se	0	X	0	X	X	0	1	X	X	0	X	0	X
BTEQZ	01100	10:8 000				7:0	sign	rx	ry	0	0	X	X	X	0	X	0	X	X	0	0	X	X	not T	cb_pc	0	X
MTSP	01100	10:8 100						rx	ry	0	1	alu_or	READ_DATA_1	zero_word	0	X	0	X	X	0	1	X	X	0	X	0	X
LI	01101					7:0	zero	rx	ry	0	0	alu_or	zero_word	imm8ze	1	rx	0	X	X	0	0	X	X	0	X	0	X
CMPI	01110					7:0	sign	rx	ry	1	0	alu_cmp	READ_DATA_0	imm8se	1	X	0	X	X	0	0	X	X	0	X	0	X
MOVE	01111							rx	ry	0	1	alu_or	READ_DATA_1	zero_word	1	rx	0	X	X	0	0	X	X	0	X	0	X
LW_SP	10010					7:0	sign	rx	ry	0	0	alu_addu	SP	imm8se	1	rx	1	0	X	0	0	X	X	0	X	1	X
LW	10011					4:0	sign	rx	ry	1	0	alu_addu	READ_DATA_0	imm5se	1	ry	1	0	X	0	0	X	X	0	X	1	X
SW_SP	11010					7:0	sign	rx	ry	1	0	alu_addu	SP	imm8se	0	X	1	1	READ_DATA_0	0	0	X	X	0	X	0	X
SW	11011					4:0	sign	rx	ry	1	1	alu_addu	READ_DATA_0	imm5se	0	X	1	1	READ_DATA_1	0	0	X	X	0	X	0	X
ADDU	11100	1:0 01						rx	ry	1	1	alu_addu	READ_DATA_0	READ_DATA_1	1	rz	0	X	X	0	0	X	X	0	X	0	X
SUBU	11100	1:0 11						rx	ry	1	1	alu_subu	READ_DATA_0	READ_DATA_1	1	rz	0	X	X	0	0	X	X	0	X	0	X
AND	11101	4:0 01100						rx	ry	1	1	alu_and	READ_DATA_0	READ_DATA_1	1	rx	0	X	X	0	0	X	X	0	X	0	X
OR	11101	4:0 01101						rx	ry	1	1	alu_or	READ_DATA_0	READ_DATA_1	1	rx	0	X	X	0	0	X	X	0	X	0	X
NOT	11101	4:0 01111						rx	ry	0	1	alu_nor	READ_DATA_1	zero_word	1	rx	0	X	X	0	0	X	X	0	X	0	X
CMP	11101	4:0 01010						rx	ry	1	1	alu_cmp	READ_DATA_0	READ_DATA_1	0	X	0	X	X	1	0	X	X	0	X	0	X
JR	11101	4:0 00000	7:5 000					rx	ry	1	0	X	X	X	0	X	0	X	X	0	0	X	X	1	READ_DATA_0	0	X
MFPC	11101	4:0 00000	7:5 010					rx	ry	0	0	alu_or	PC	zero_word	1	rx	0	X	X	0	0	X	X	0	X	0	X
SLLV	11101	4:0 00100						rx	ry	1	1	alu_sll	READ_DATA_1	READ_DATA_0	1	ry	0	X	X	0	0	X	X	0	X	0	X
SRAV	11101	4:0 00111						rx	ry	1	1	alu_sra	READ_DATA_1	READ_DATA_0	1	ry	0	X	X	0	0	X	X	0	X	0	X
MTIH	11110	4:0 00001						rx	ry	1	1	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	0
MFIH	11110	4:0 00000						rx	ry	1	1	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	0
MFC0	11110	4:0 00000						rx	ry	1	0	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	imm 7:5 1
MTC0	11110	4:0 00001						rx	ry	1	0	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	imm 7:5 0

图 2.2: 控制信号表

2.2.1 内部总线设计

TODO。

2.2.2 处理器

TODO。

2.2.3 SRAM 与 UART

TODO。

2.2.4 显示控制器

TODO。

2.2.5 PS/2 控制器

PS/2 控制器主要用于接受键盘传来的信号，将其进行解析后响应总线发来的读取请求。其中地址 0xE003 映射到 PS/2 的控制信号，程序可以通过访问此地址得知当前键盘是否有未读取的新数据；地址 0xE002 映射到 PS/2 的数据，程序可以通过访问此地址获取未读取过的键盘扫描码。

为了过滤 P2/2 信号中的毛刺，我们对信号进行了抽样，其中抽样频率为系统的主频。并通过“打两拍”的方法得到滤波后的 PS/2 时钟信号和数据信号。

我们使用了一个六个状态的状态机来对滤波后的信号进行解析，分别读取其起始位、八位数据位、校验位和结束位。并在每个状态设置了一个计时器，当一个状态停留过久时，会自动回到初始的等待状态，以避免信号丢包带来的状态错乱。

2.2.6 SD 卡控制器

我们的系统选择 SD 卡作为外部的块存储设备，所有和 SD 卡访问相关的电路都实现在这个模块中。这个模块实现的重点在于 SD 卡的存取。

SD 卡支持两种接口，SDIO 和 SPI。其中，SDIO 接口比较复杂且封闭，SPI 则比较自由且资料相对较多，所以我们主要研究的是通过 SPI 来存取 SD 卡中的信息。SD 卡分为标准容量的 SDSC 卡（小于等于 2GB）、大容量的 SDHC 卡（2GB~32GB）以及更大容量的 SDXC 卡（32GB~2TB），我们尝试全部支持。

SPI 是一种同步串行总线，它根据时钟相位、锁存数据的时机可以分为 $2 \times 2 = 4$ 种模式。SD 卡支持的是模式 0，即时钟上升沿锁存数据，下降沿发送数据。此外，SPI 收发数据时，**最高有效位 (MSB)** 先被处理。

SPI 使用了四条数据线：

- SCLK 时钟，主机产生
- SS 或 CS 从机选择，一般是低有效
- MOSI 或 DI 主机发送，从机接收
- MISO 或 DO 从机发送，主机接收，需要上拉电阻

与 SD 卡管脚的对应关系如图2.3*所示。

本次实现中，我们使用状态机来产生 SPI 总线的时钟、控制信号以及收发数据，为了方便状态机的设计，我们将产生时钟以及发送和接收 SPI 总线的数据作为子过程（subroutine），方便其他状态“调用”。

* 图片来自 http://elm-chan.org/docs/mmc/mmc_e.html

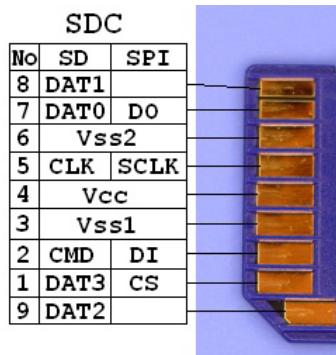


图 2.3: SD 卡接口

为了访问 SD 卡中的数据，首先要对 SD 卡进行上电初始化，图2.4[†]是 SD 卡规范中提供的初始化流程，本次实现我们也是严格按照这个流程图实现的，不过虚框中的可选项我们没有实现。该流程图已经十分详细，不过还需要详细说明的是“Power-on”过程、CMD 的发送和响应的接收。

“Power-on”过程 最开始的上电过程需要等待 1ms，等 SD 卡电压达到所需电压，然后在 \overline{CS} 拉高的情况下， $SCLK$ 产生至少 74 个时钟周期用于 SD 卡上电初始化。接着，才可以进行 CMD 命令的发送。

CMD 的发送 CMD 命令长度固定为 6 个字节，格式如图2.5[‡]，发送时只需要正确构造即可。SPI 模式下，SD 卡不会验证命令的 CRC，但相应的位还需要控制器发送，具体内容随意即可。但是，SD 卡接收 CMD0 命令时还没有进入 SPI 模式，所以 CMD0 命令需要设置正确的 CRC（硬编码即可）；SD 卡规范中规定 CMD8 命令的 CRC 校验一直打开，故 CMD8 命令也需要设置正确的 CRC（同样，硬编码即可）。本模块用到的命令见表2.1，命令的编号为十进制数。

响应的接收 在 CMD 被发送之后，需要等待若干个时钟周期，直到响应到来，表现为 MISO 被拉低（之前被上拉电阻或 SD 卡拉高）。然后，即可根据之前发送的 CMD 类型，接收不同长度的响应数据。本模块使用到的命令对应的响应有三种：1 字节的 R1、5 字节的 R3 和 5 字节的 R7。一般命令的响应为 R1，具体内容如图2.6[§]所示。CMD58 由于需要返回 OCR (Operation Conditions Register) 的内容，所以响应为 R3；CMD8 命令也需要返回额外的信息，所以响应为 R7。事实上，R3 就是 R1 后紧随一个 OCR 字段，R7 就是 R1 后紧随 CMD8 的响应数据。注意，当 SD 卡收到了无法识别的命令时，响应一律为 R1。

[†]图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-2

[‡]图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Table 7-1

[§]图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-9

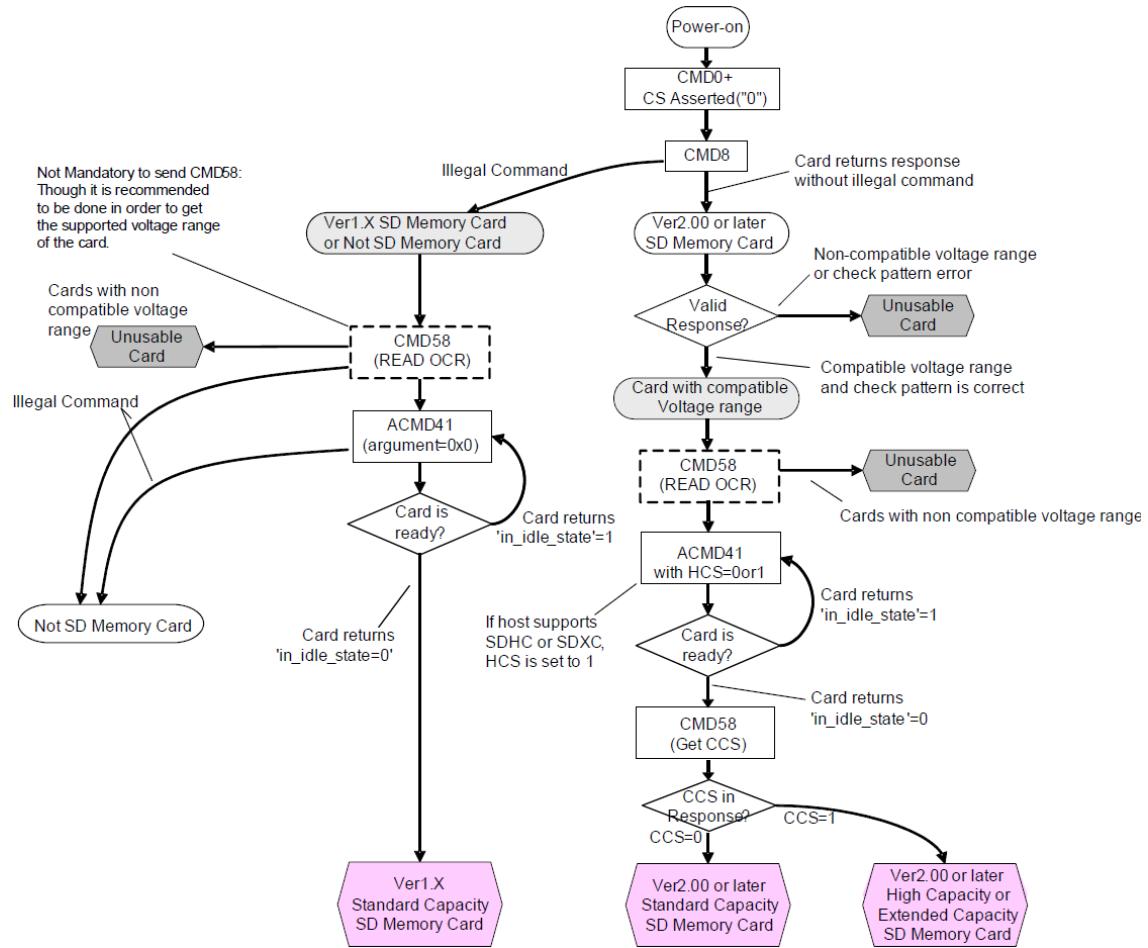


图 2.4: SPI 模式初始化流程

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

图 2.5: CMD 命令格式

成功初始化之后，可以开始进行读写扇区等操作。为了确保每次操作的块恰好是一个扇区（这方便之后的使用），我们在读写操作之前还发送了CMD16命令，设置块大小就为一个扇区的大小，512字节。注意，事实上只有SDSC卡需要这个命令，SDHC或SDXC卡读写操作的块大小固定为512字节。

读扇区使用CMD17命令，参数包含一个地址。值得注意的是，在初始化流程中，如果检测出SD卡为SDSC，则寻址方式为按字节寻址；如果SD卡为SDHC或SDXC，则按照块（或扇区）寻址（从0开始），每块大小512字节，这是在SD卡规范一个表格的注

命令	说明
CMD0	上电初始化
CMD8	检查电压情况
CMD55	所有ACMD的前序命令
ACMD41	进行初始化，获得是否已经初始化好
CMD58	读取 OCR (寄存器) 内容
CMD16	设置块长度
CMD17	读取单个块
CMD25	写入多个块

表 2.1: SD 卡命令

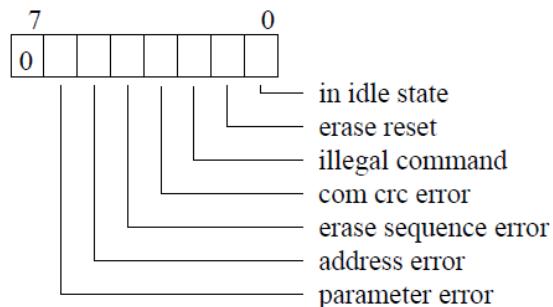


图 2.6: R1 具体内容

释中说明的[¶]。

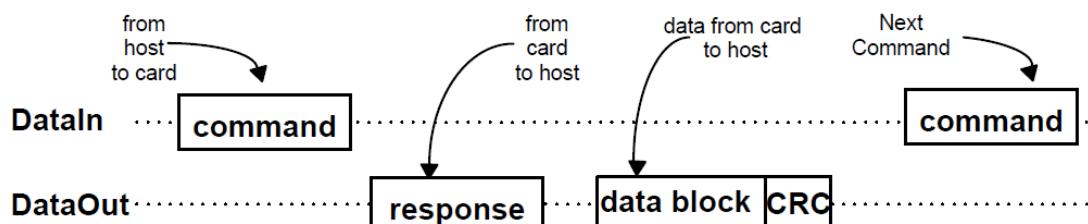


图 2.7: CMD17 读扇区流程

图2.7[¶]清晰地展示了 CMD17 命令的时序，CMD17 命令的响应除了一个字节的 R1 响应以外，一段时间后还会发送一个 data block 以及相应的 CRC。

data block 由一个字节的 token 和后续 1~2048 个字节的数据组成。状态机可以检测 token 来得知 data block 的到来，token 分为两种：数据 token (11111110) 和错误 token

[¶]具体出现在 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Table 7-3

[¶]图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-3

(000 后跟 5 位错误码)。当检测到数据 token，说明数据正常读取，可以进行接收了；当检测到错误 token 就要做相应的错误处理，我们的实现是直接停机。两个字节的 CRC 紧随 data block 之后，由于 SPI 模式中没有开启 CRC 校验，状态机只需要读取这两个字节并丢弃即可。

考虑到优化寄存器资源的使用，我们没有读完一整个扇区再进行内存的写操作，因为那样需要使用 512 字节的寄存器（共 4096 位）作为缓冲区。我们的实现中，每读到 2 个字节就写内存一次，因为内存数据总线宽度为 16 位，就是 2 个字节。有一点需要注意，就是**字节序**的问题，即连续读到的 2 个字节中哪一个作为内存中一个字的高字节，哪一个作为低字节。我们使用**小端序**，即 SD 卡低地址的字节，亦即先被读取到的字节，对应内存中一个字更低的字节。

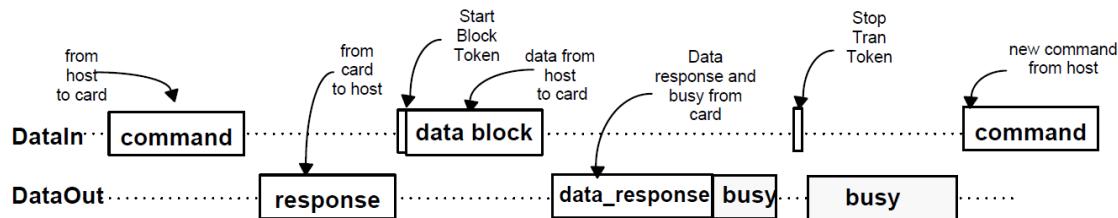


图 2.8: CMD25 连续写扇区流程

对于写入操作而言，我们使用了连续写入命令 CMD25 来优化写入速度，图2.8**清晰地展示了 CMD25 命令的时序。这个命令同样包含一个地址参数，寻址方式和读取命令一样，也区分字节寻址以及块（或扇区）寻址。发送命令后需要等待 R1 响应，若响应正常，发送 start block token（实际上就是 8 个 1）后即可开始发送 data block 来写入。CMD25 的 data block 格式和 CMD17 的完全类似，区别仅在于发送的时候 token 取为写入数据 token (11111100)。每发送一个 data block 后需要等待一个 8 位的数据响应，如果响应的后 5 位为 00101 则说明数据被接受，之后 MISO 会被 SD 卡持续拉低，等到 MISO 恢复为高则可以继续发送下一个 data block。当所有需要写入的 data block 发送完毕并接收到响应之后，需要发送 stop token (11111101 11111111) TODO

本模块整体状态机十分复杂，这里就不展示了。

经过实际测试，本模块确保可以支持这些 SD 卡：

- (SDHC) Kingston 8GB Class 4 Micro SD
- (SDHC) Kingston 16GB Class 4 Micro SD
- (SDHC) SONY UHS-1 8G Class 10 microSDHC
- (SDHC) SONY UHS-1 16G Class 10 microSDHC
- (SDSC) 某品牌 2GB 存储卡

**图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-7

- (SDSC) SanDisk 1GB Micro SD

但是，特别是对于 SanDisk 某些卡，无法支持。由于时间和材料限制，我们没有测试 SDXC 卡，不过 SDXC 卡与 SDHC 卡差别不大，理论上也应该能够很好的支持。

注：Micro SD 卡与普通 SD 卡区别仅仅在于形状。

本模块可以算是本项目中实现、调试较有挑战性的一部分，我们在参考了互联网大量资料的情况下，利用周末累计调试了两天半（包括休息）才完成。

此外，由于我们使用了易于其他广泛使用的计算机系统读写和易于替换的 SD 卡，我们未来可以通过软件实现文件系统的处理，从而更好的支持和其他计算机系统的文件交换。

2.2.7 GPIO

TODO。

2.3 软件部分

2.3.1 汇编器

TODO。

2.3.2 PS/2 键盘驱动

在项目中，我们用汇编语言实现了`getchar`和`gets`函数，用于阻塞地从键盘读取字符或字符串。

其中`getchar`从键盘读入一个字符，并通过查表的方式将读取到的键盘扫描码转换为对应字符的 ascii 码，并将其存入 r4 寄存器后进行返回。

我们的`gets`用于从键盘读入一个以换行符结尾的字符串，存入指定内存地址，并回显到屏幕。其中读入时支持回退，显示时还支持换行、回车和滚屏。

具体来说，读入时我们通过 r0 寄存器得到读取字符串的存储地址，并不断阻塞地从键盘读取字符，若读到的是回退符，则进行相应的回退处理，否则将其查表转换为 ascii 码后存入指定内存并进行显示，并在读取到换行符时结束读取。显示时，我们首先判断当前字符是换行、回车还是普通字符，分别对其进行处理，显示完一个字符后再检查是否需要自动换行和滚屏处理。其中光标的位置和闪烁频率，作为全局变量存在内存中。

2.3.3 上电自检程序

TODO。

2.3.4 shell

TODO。

2.3.5 2048 小游戏



图 2.9: 2048 小游戏

2048 是一款休闲益智游戏。玩家每次控制所有方块向同一个方向运动，两个种类相同的方块撞在一起之后合并成为更高级的块。每次操作之后会在空白的方格处随机生成一个较弱的块。游戏胜利的条件是得到一个叫做“lss”的方块，而如果 16 个格子全部填满并且相邻的格子都不相同，也就是无法移动的话，那么恭喜你，gameover。

在本项目中我们用汇编语言实现了一个 2048 游戏，支持随机开始新局面、从 SD 卡读盘、存盘到 SD 卡并退出等功能。具体来说一个游戏局面由 16 个方块对应的状态所构成，每次游戏开始时我们从 SD 卡加载游戏局面到内存。而在游戏过程中，程序将循环执行渲染局面、阻塞地等待并读取用户输入、根据用户的输入执行相应的处理。

若用户输入的命令为 **N**，我们则随机生成一个包含若干最弱方块的新局面。其中随机数由 PS/2 驱动程序的计数器提供。

若用户输入的命令 **W**、**S**、**A**、**D**，我们则根据用户的输入进行方块移动和合并的处理，之后再随机在空余位置生成最弱的方块。

若用户输入的命令 **Esc**，我们则将当前游戏局面存盘到 SD 卡，再结束程序。

而渲染一个局面时，我们需要显示欢迎语、游戏说明以及每一个方块的边框、文字，并进行着色。由于我们的显存存储的是 30×80 个 ascii 字符及其对应的前景色和背景色，故在游戏中每一个方块由 7×14 个字符构成。其中每一种方块的背景色、文字均作为常量存储在内存中。在程序中我们只需循环计算每个方块的位置，并在显存中写入其所对应的 7×14 个字符的内容和颜色即可。

值得一提的是，汇编程序不仅在编写上反人类，而且缺乏有效的调试手段。在调试过程中，一个困扰我们许久的 bug 是程序跳转指令的立即数长度有限，而我们的编译器又没有进行溢出检查，导致程序跳到奇怪的地方。

2.3.6 BadApple!!动画播放

TODO。

2.3.7 BadApple!!的弹幕

在本项目中，我们还实现了视频的弹幕功能，即在 BadApple!!的动画播放过程中，用户可随时键入评论，并以换行符结束，用户键入的评论将随即从屏幕中飘过。

弹幕功能基于的是时钟中断，每隔 5ms 系统将触发一次时钟中断。若中断处理程序发现当前进程为 BadApple!!动画的播放程序，则会执行我们的弹幕程序。弹幕程序主要执行两个功能：读取用户的键盘输入；当读取到换行符时将用户评论写入显存，并不断更新其显示位置以实现从屏幕中飘过的效果。

但是由于动画播放要求是连续流畅的且中断处理程序不支持被二次中断，故弹幕程序中的读取用户的键盘输入，以及循环显示评论都不能简单地使用堵塞的实现方式。具体来说我们将读取字符串操作变成了很多次中断来实现，每次先检查缓存区是否有新输入，如果有则调用 `getchar` 函数读取一个字符，并将其存入内存；若当前没有新输入，则直接结束中断。而在显示评论时，原来的循环刷新也要被改写成每次只刷新一次。因为被分成若干次调用，程序原来存放在寄存器中的计数器以及循环变量等都需要存放在内存中，并需记录程序当前执行到哪一个步骤。

2.3.8 内存转储

TODO。

2.4 内存地址分配

内存地址的分配需要软件和硬件配合来完成，我们的计算机系统中的内存分 TODO

起始地址	结束地址	存储内容
0x0000	0x3FFF	内核代码
0x4000	0x7FFF	用户代码

表 2.2: 内存地址映射

第 3 部分

展望

TODO。

第 4 部分

参考文献

1234