

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

---

<b>1 概述</b>	<b>1</b>
1.1 系统概要	1
1.2 组内分工	2
<b>2 设计与实现</b>	<b>3</b>
2.1 总体架构	3
2.2 硬件部分	3
2.2.1 内部总线设计	3
2.2.2 处理器	4
2.2.3 SRAM 与串口控制器	4
2.2.4 显示控制器	6
2.2.5 PS/2 控制器	6
2.2.6 SD 卡控制器	6
2.2.7 GPIO	12
2.3 软件部分	12
2.3.1 汇编器	12
2.3.2 汇编程序调用约定	12
2.3.3 PS/2 键盘驱动	12
2.3.4 上电自检程序	13
2.3.5 shell	13
2.3.6 2048 小游戏	14
2.3.7 BadApple!!动画播放	15
2.3.8 BadApple!!的弹幕	15
2.3.9 内存转储	15
2.4 内存地址分配	16

3 未来可能的扩展空间	18
4 致谢	19
5 参考文献	20

## 第 1 部分

---

# 概述

---

### 1.1 系统概要

整体上，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** 字长 16 位，最高可以运行在 40MHz，最高可达 40MIPS，CPI=1.01。
- **内存** 按字编址，每个字的宽度为 16 位。由于 16 位地址总线宽度限制，内存寻址只支持 64K 字，即 128KiB。支持单周期访存。
- **串口** 支持通过 9 针 RS232 串口和其他计算机连接。
- **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，存储的数据量远远大于板载的 Flash，并且 SD 卡易于更换。此外，SD 卡控制器支持 DMA，减轻处理器压力。
- **GPIO** 结合驱动程序，可以实现 SPI 总线协议、 $I^2C$  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，我们用 Python 3 重新实现了一个。支持伪指令（如 la 以及 li）的处理，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。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调用相应的代码段。
- **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** 开机的时候会被首先执行，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）。检测出错误后会在数码管上显示错误码并在屏幕上（如果显示控制器此时工作正常）显示错误信息和红色[FAIL]字样，然后停机。

## 1.2 组内分工

组内二人的分工如下：

### 谭闻德

- 整体数据通路的绘制
- 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
- 显示控制器设计与实现
- 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
- 系统集成、调试与测试
- 整体文档的撰写

### 刘明华

- 整体数据通路的绘制
- 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
- 显示控制器的实现
- PS/2 控制器设计与实现
- PS/2 驱动程序、读取字符串函数的汇编实现
- 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
- 文档的撰写与排版

## 第 2 部分

---

# 设计与实现

---

## 2.1 总体架构

计算机系统是硬件和软件的完美结合，所以我们的系统也包括硬件和软件部分。

硬件和软件由指令集联系在一起，我们实现的指令有（不含汇编器支持的伪指令）：`addiu`、`addiu3`、`addu`、`subu`、`addsp`、`and`、`or`、`not`、`move`、`b`、`beqz`、`bnez`、`bteqz`、`cmp`、`cmpi`、`jr`、`li`、`lw`、`sw`、`lw_sp`、`sw_sp`、`mtsp`、`mtih`、`mfihi`、`mtc0`、`mfc0`、`mfpc`、`sll`、`sra`、`sllv`、`sraw`以及`nop`。

## 2.2 硬件部分

整体数据通路和架构如图2.1所示。

### 2.2.1 内部总线设计

首先，介绍内部总线设计。内部总线是本系统的最关键的部分，因为它连接了处理器、内存以及其他所有模块，允许模块间相互通信、传输数据。

**总线分派器**的功能是解析请求方发来的地址，并选择相应的设备，同时实现了 IO 地址映射。

**总线仲裁器**的功能是根据某个优先级，确定当前设备处理哪一个请求。我们的实现中，为实现简单，采用固定优先级的做法，优先级为：SD 卡控制器 DMA 请求 > 处理器访存阶段请求 > 处理器取指令阶段请求。优先级可能有其他设置，但取指令阶段的请求一定为最低优先级，否则其他部分或全部请求将永远被阻塞。

实际上，总线分派器和总线仲裁器的本质都是数据选择器，总线分派器选择设备的响应并发给请求方，总线仲裁器选择请求并发送给接在总线上的设备。

总线分派器和总线仲裁器组合使用，可以灵活地组成各种结构，我们本次实现将处理器、内存和其他外设组成了交叉互联的结构，允许它们互相访问。

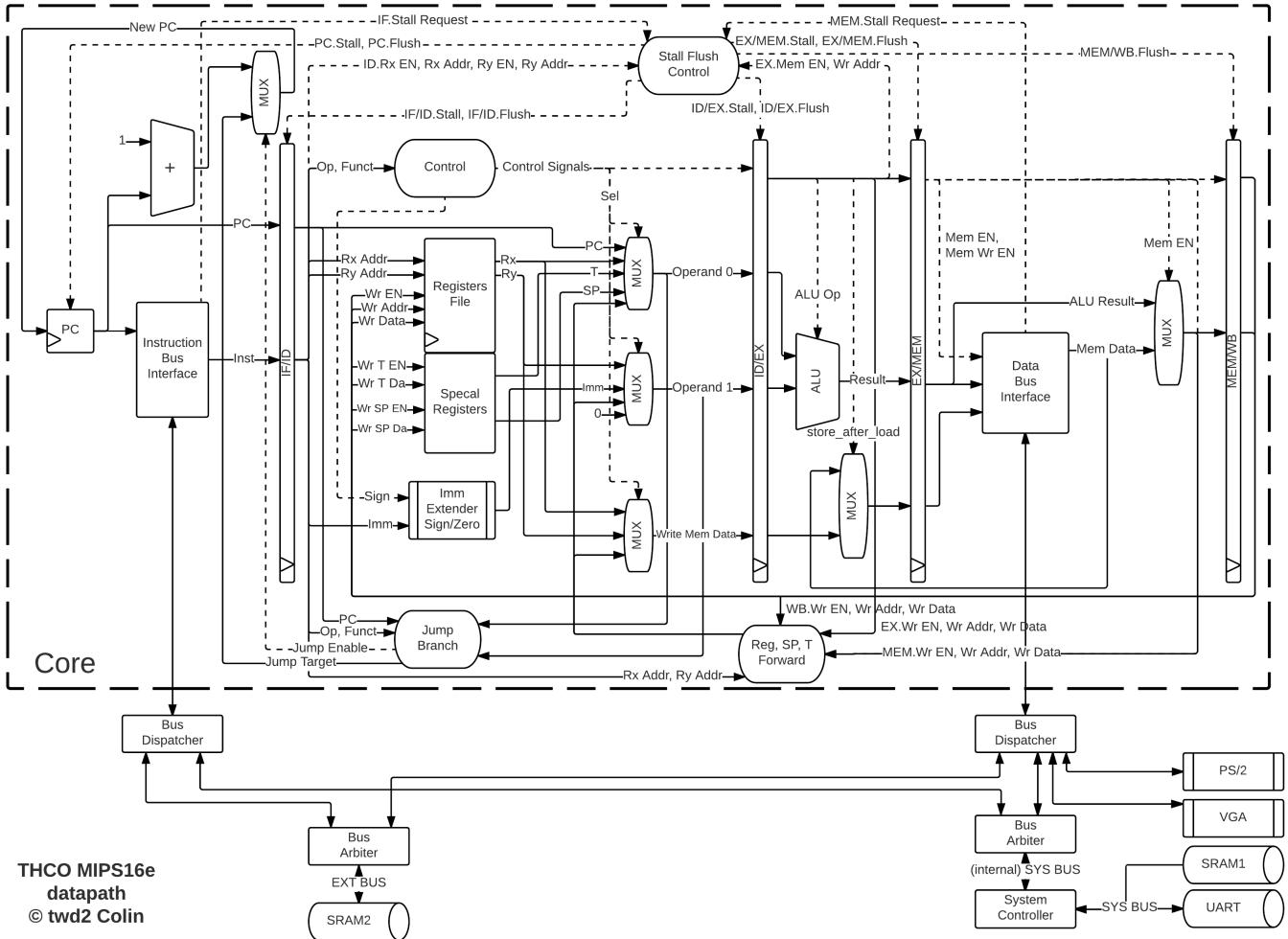


图 2.1: 数据通路和架构图

### 2.2.2 处理器

处理器的内部设计在图2.1中已经有清晰的体现，处理器对外的接口有外部中断请求线6根、指令总线接口和数据总线接口。指令总线接口和数据总线接口都连接到内部总线。控制器中的控制信号的设置如图2.2所示。

### 2.2.3 SRAM 与串口控制器

这个模块的作用是将物理总线包装一层，并和内部总线连接，连接在物理总线上面的设备有（处理器片外的）SRAM 和（处理器片外的）串口控制器。根据地址的不同，选择不同的设备（SRAM 或者串口控制器）访问，SRAM 使用 $\overline{OE}$ 和 $\overline{WE}$ 信号控制，串口使用 $wrn$ 以及 $rdb$ 信号控制。

	op	fun_c_p_o_s	func	sub_fu_nc_pos	sub_fu_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_hi_lo	wri_te_en	wri_te_en	BRANCH_EN	BRANCH_PC	IS_L_OAD	cp0_rea_d_addr	cp0_read_en
NOP	00001							rx	ry	1	1	X	X	X	0	X	0	X	X	0	0	X	X	0	X	0	X	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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
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	0
JR	11101	4:0 00000	7:5 000					rx	ry	1	0	X	X	0	X	0	X	X	0	0	X	X	1	READ_DATA_0	0	X	0	
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	0
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	0
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	0
MTIH	11110	4:0 00001						rx	ry	1	1	X	X	0	X	0	X	X	0	0	X	X	0	0	0	0	0	
MFIH	11110	4:0 00000						rx	ry	1	1	X	X	0	X	0	X	X	0	0	X	X	0	0	0	0	1	
MFC0	11110	4:0 00000						rx	ry	1	0	X	X	0	X	0	X	X	0	0	X	X	0	X	0	imm 7:5 1	0	
MTC0	11110	4:0 00001						rx	ry	1	0	X	X	0	X	0	X	X	0	0	X	X	0	X	0	imm 7:5 0	0	

图 2.2: 控制信号表

若地址为0xBF00，则让串口控制器驱动物理总线，即置wrn或rdn为有效；否则，让SRAM工作，即置OE或WE为有效。

此外，串口控制器有data\_ready、tbre以及tsre控制信号，我们把它们映射到地址0xBF01对应的字，作为串口控制寄存器。值得注意的是，片外串口控制器的data\_ready、tbre以及tsre控制信号所在时钟域和主时钟不同，直接输入到FPGA的触发器中可能会产生跨时钟域相关问题，导致触发器进入亚稳态，从而导致状态机异常甚至卡死，进而导致整个系统设计失败。为了解决这个问题，我们将这三个输入信号用主时钟延迟两个周期再传给需要的地方。

从后文可以看出，其他IO都被映射到了0xE000以上的地址，这里把串口控制器的地址映射到0xBF00和0xBF01是出于兼容性的考虑。

### 2.2.4 显示控制器

此模块分为两个子模块，总线接口部分和 VGA 接口部分。

总线接口部分和内部总线连接，处在主时钟域。

VGA 接口部分连接到 VGA 接口，处在 VGA 的时钟域。

这两个部分通过 FIFO 联系起来，传递像素数据。

TODO。

### 2.2.5 PS/2 控制器

PS/2 控制器主要用于接受 PS/2 接口传来的信号，将其进行解析后存入缓冲器，同时响应总线发来的读取请求。其中，地址 **0xE003** 映射到 PS/2 控制器的控制寄存器，程序可以通过访问此地址得知当前是否有未读取的新数据；地址 **0xE002** 映射到 PS/2 控制器的数据缓冲寄存器，程序可以通过访问此地址获取最后一次读到的数据。

显然，PS/2 的信号来自于不同的时钟域，也会有跨时钟域的问题。这里，我们对信号用主时钟进行了采样，并通过用触发器延迟两个周期的方法得到采样后的稳定的信号。

PS/2 的信号分为时钟信号和数据信号，我们使用了一个有六个状态的状态机来对 PS/2 的信号进行处理，依次检测 PS/2 时钟信号的下降沿并读取其起始位、八个数据位、奇偶校验位和结束位。我们还在每个状态设置了一个计时器，当在一个状态停留过久时，状态机会自动回到初始的等待状态，以避免时钟信号突发错误而带来的状态错乱，提高了 PS/2 控制器的稳定性。

### 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 则比较开放且资料相对较多，所以我们主要研究的是通过 SPI 来访问 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<sup>\*</sup>所示。

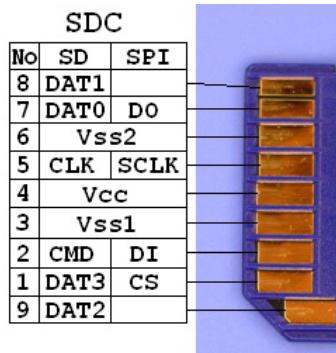


图 2.3: SD 卡接口

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”。

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<sup>†</sup>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 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<sup>‡</sup>，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，

<sup>\*</sup>图片来自 [http://elm-chan.org/docs/mmc/mmc\\_e.html](http://elm-chan.org/docs/mmc/mmc_e.html)

<sup>†</sup>图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-2

<sup>‡</sup>图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Table 7-1

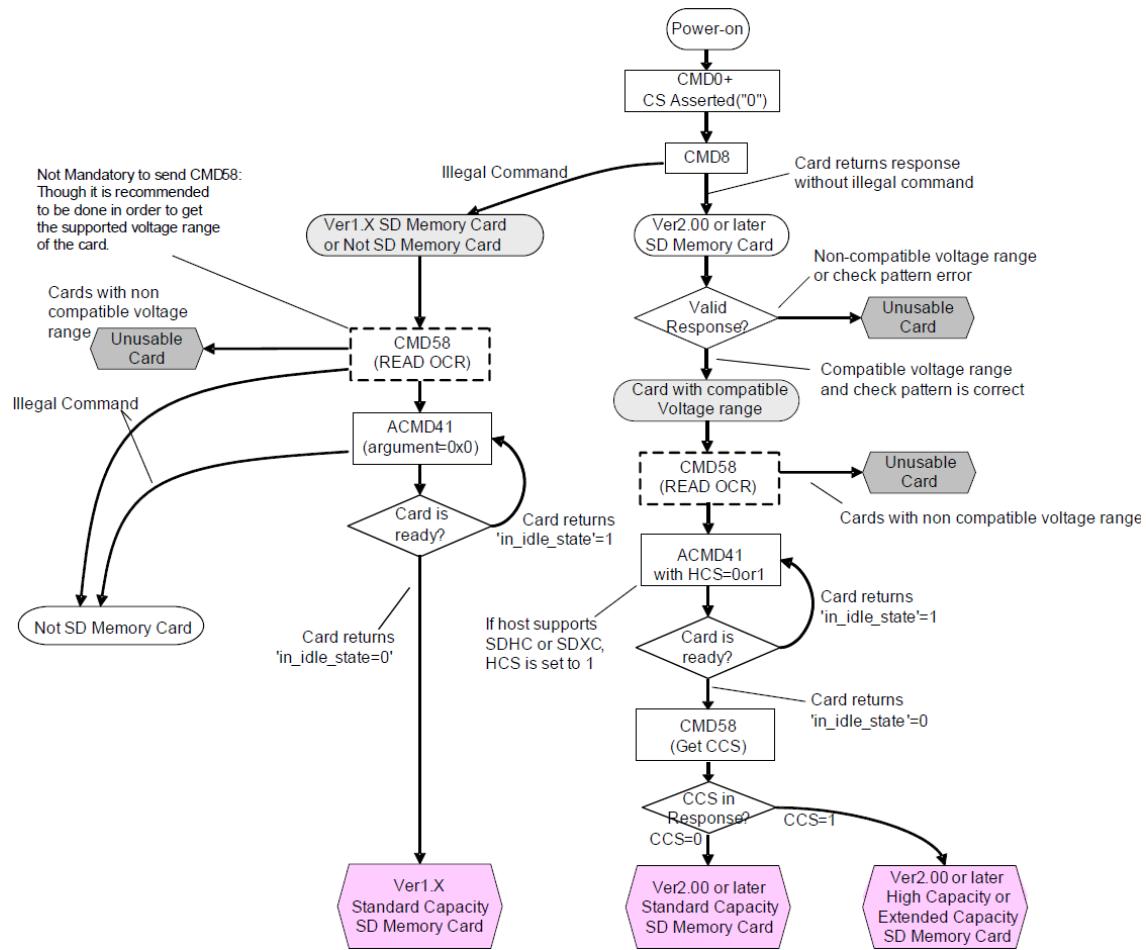


图 2.4: SPI 模式初始化流程

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

图 2.5: CMD 命令格式

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<sup>§</sup> 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。

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

<sup>§</sup> 图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-9

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 (部分)

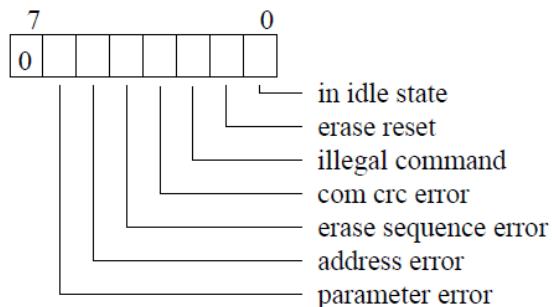


图 2.6: R1 具体内容

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释中说明的<sup>¶</sup>。

图2.7<sup>¶</sup>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 (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

<sup>¶</sup>具体出现在 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Table 7-3

<sup>¶</sup>图片来自于 SD Specifications Part 1 Physical Layer Simplified Specification Version 6.00 Figure 7-3

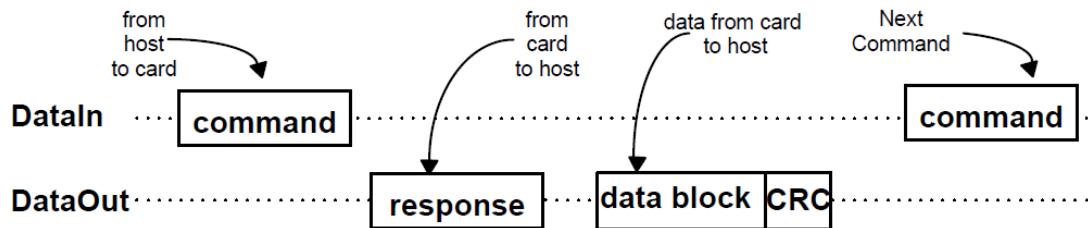


图 2.7: CMD17 读扇区流程

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。**字节序**的问题需要注意，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。

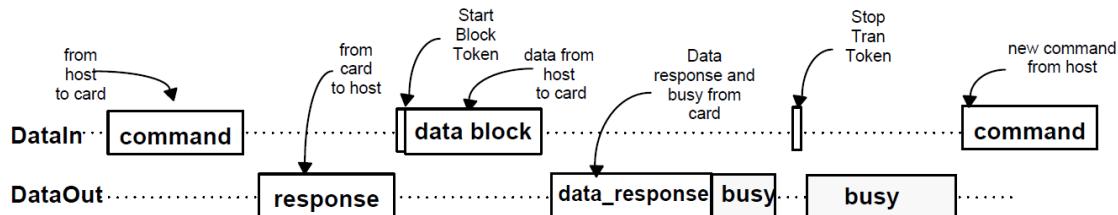


图 2.8: CMD25 连续写扇区流程

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)。此外，发送 data block 时 CRC 字段同样需要发送，但由于 SPI 模式没有开启 CRC 校验，所以发送的内容可以任意。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，可以发送 stop

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

token (11111101) 来停止这次连续写操作。等待 8 个时钟周期后，**MISO**会被 SD 卡持续拉低一段时间，等到**MISO**恢复为高则说明本次写入操作完成。

同样考虑到优化寄存器资源的使用，我们每从内存读取一个字就向 SD 卡发送 2 个字节。

需要注意的是，执行完 CMD25 后还应当检查实际写入的块数（因为有些错误无法立刻发现），但本次实现没有考虑。

SD 卡控制器对处理器的接口表现为几个寄存器：

- **起始扇区号寄存器** 下一次请求要操作的起始扇区号。
- **内存起始地址寄存器** 下一次请求要操作的起始内存地址。
- **扇区数寄存器** 下一次请求要操作的扇区数。
- **响应寄存器** 这个寄存器目前只有一个中断标志位，表明上次的操作是否已经完成。中断标志位写 1 清零。
- **请求寄存器** 这个寄存器不是真实存在的寄存器，而是一个端口。向这个寄存器写入数据就相当于给 SD 卡控制器的状态机发送请求，让其开始执行操作。操作码 0x0001 为读取操作，0x0002 为写入。

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
- (SDSC) SanDisk 1GB Micro SD

遗憾的是，由于不明原因，对于 SanDisk 某些特定型号的存储卡，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，从而更好地支持和其他计算机系统的文件交换。

### 2.2.7 GPIO

这个模块就是将 FPGA 的三态门包装了一层。输出使能接到 GPIO 控制器的方向寄存器，低有效。输入输出数据接到 GPIO 的数据寄存器。如此简单的封装，给予了程序直接访问 FPGA 的 IO 引脚的能力，能够大大提高灵活性。

## 2.3 软件部分

### 2.3.1 汇编器

前文已说明，实验材料中提供的汇编器不方便使用，所以我们重新实现了一个。高级语言十分方便使用，所以我们使用的是 Python 3 语言。但是，这也就导致我们的汇编器无法运行在我们的系统上。

汇编器的本质就是字符串处理，对于每一条指令，解析其指令助记符和操作数，然后根据定义好的翻译规则，生成相应的机器码即可。本次实现，我们实现为多遍扫描：

- 第一遍扫描将文本输入转换为内部数据结构。
- 第二遍扫描完成预处理、伪指令的转换和符号表的建立。
- 第三遍扫描完成符号的解析和立即数字面值常量的解析。
- 第四遍扫描检查操作数合法性、立即数长度并生成机器代码。

事实上，在一般情况下，汇编源程序汇编成机器码之后还需要进行链接，但是在我们的系统中，链接的工作在汇编前手动完成。

顺便指出，此汇编器，特别是其立即数长度检查的功能，方便了我们后续所有汇编程序的开发。

### 2.3.2 汇编程序调用约定

在编写汇编程序时，我们约定：

- 寄存器 r7 存放返回地址
- 寄存器 r0~r7 均由被调用者保存
- r0、r1、r2 以及 r3 分别为第一、第二、第三和第四个参数（如果存在），其余参数通过栈传递
- 如果有返回值，存放在 r4 内，此时 r4 由调用者保存
- 全局符号名不以下划线开始，内部符号名以下划线开始，但汇编器不会检查

### 2.3.3 PS/2 键盘驱动

在本系统中，我们用汇编语言实现了 `getchar` 和 `gets` 函数，支持同步阻塞地从 PS/2 键盘读取字符或字符串。

其中**getchar**从键盘读入一个扫描码，后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 寄存器返回。

**gets**则用于从键盘读入一个以换行符结尾的字符串，存入由 r0 指定的内存地址，并回显到屏幕。其中读入时支持退格。

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。

### 2.3.4 上电自检程序

我们的系统有上电自检功能。在我们的实现中，上电自检主要检查的是内存读写是否正确，因为内存访问的时序要求相对严格，在时钟频率过高的情况下，内存读写的时序可能无法得到满足，从而出现错误。

上电自检程序对于`0x7000~0xBEFF`地址区间内每一个内存单元，首先将一个唯一的数据（例如，地址）写入其内，所有单元全部写完之后进行读取验证，如果读出的内容和写入的完全一致则自检通过，屏幕上显示[OK]，继续开机，否则提示[FAIL]并停机。虽然简单，但是这样确实能够检测出很大一部分问题。注意，是先整体写入一遍，再读取一遍，而不是对于每一个单元分别先写再读。

### 2.3.5 shell

我们用实现好的基础函数**getchar**、**gets**、**putchar**以及**puts**实现了一个类似 Unix 的简易的 shell。shell 通过比对字符串是否和已有字符串相等来得知用户想要执行什么命令，然后跳转到相应的代码段。

shell 显示时还支持回车、换行和滚屏，这都是由**putchar**和**puts**实现的。显示时，我们首先判断当前需要显示的字符是回车、换行还是普通字符，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，存放在显示控制器的寄存器中。

### 2.3.6 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最弱的块（Colin）。游戏胜利的条件是得到一个最强的叫做“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，游戏结束了。此外，游戏胜利后并不会结束，两个“lss”方块合并后会有奇特效果。

在本系统中我们用汇编语言实现了一个上述游戏，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、调用 `getchar` 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，`getchar` 中在等待 PS/2 控制器有新数据到来的时候维护一个计数器，可以用来产生伪随机数，熵由用户敲击键盘的间隔提供。

若用户输入的命令为 **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。

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

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 80x30 个 ASCII 字符及其对应的前景色和背景色，

我们设计在游戏中每一个方块由  $7 \times 14$  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  $7 \times 14$  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需要细心，而且由于我们的系统的简单性，缺乏行之有效的调试手段。在调试过程中，一个困扰我们许久的 bug 是程序分支指令的偏移量立即数长度有限，而我们的汇编器恰好没有对出问题的指令进行立即数溢出检查（现已补上），导致程序行为和预期不一致。

### 2.3.7 BadApple!!动画播放

动画播放的实质就是逐帧显示。我们将动画原始数据存在 SD 卡中 32MB 的位置，在程序中通过循环调用 SD 卡控制器，让 SD 卡控制器将某一帧的数据读出并存放在内存缓冲区中，程序轮询等待 SD 卡控制器操作完成后，在将帧内容再复制到显存中，达到显示的效果。不直接让 SD 卡控制器将帧数据读入显存是因为 SD 卡速度很慢，如果这样做会导致一帧在显示控制器处理的时候发生变化，显示效果变差。

我们的动画数据总计有 51.25MB，用其他片上或板载的存储设备均无法存储，因此这个功能体现出 SD 卡的重要作用并验证了 SD 卡读取功能的正确性。

### 2.3.8 BadApple!!的弹幕

在本系统中，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正在执行的是 BadApple!!动画的播放程序，则会执行弹幕程序。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。

由于动画播放要求连续流畅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阻塞的实现方式。具体来说我们用软件实现了一个状态机，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，这些就是状态机的状态。

### 2.3.9 内存转储

这个程序相对比较简单。由于 SD 卡控制器支持 DMA，所以这个程序只需要向 SD 卡控制器发送请求，把内存初始地址设为 0、扇区数设为  $64K \times 2B / 512B = 256$  个，发送写

入命令后轮询方式等待 SD 卡操作完成即可。

## 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配如表2.2所示。

起始地址	结束地址	存储内容
0x0000	0x3FFF	内核代码
0x4000	0x7FFF	用户代码
0x8000	0xBEFF	数据段
0xBF00	0xBF00	串口控制器 数据寄存器
0xBF01	0xBF01	串口控制器 控制寄存器
0xBF02	0xDFFF	堆和栈
0xE000	0xFFFF	IO 内存映射区域
0xE000	0xE000	GPIO 控制器 数据寄存器
0xE001	0xE001	GPIO 控制器 方向寄存器
0xE002	0xE002	PS/2 控制器 数据寄存器
0xE003	0xE003	PS/2 控制器 控制寄存器
0xE004	0xE004	定时器 保留寄存器 (暂未使用)
0xE005	0xE005	定时器 控制寄存器
0xE006	0xE007	保留, 未映射
0xE008	0xE008	SD 卡控制器 扇区号低 16 位
0xE009	0xE009	SD 卡控制器 扇区号高 16 位
0xE00A	0xE00A	SD 卡控制器 DMA 内存地址低 16 位
0xE00B	0xE00B	SD 卡控制器 DMA 内存地址高 16 位
0xE00C	0xE00C	SD 卡控制器 扇区数低 16 位
0xE00D	0xE00D	SD 卡控制器 扇区数高 16 位
0xE00E	0xE00E	SD 卡控制器 响应寄存器
0xE00F	0xE00F	SD 卡控制器 请求寄存器
0xE010	0xEFFB	保留, 未映射
0xEFFC	0xEFFC	显示存储器 显存偏移地址低 16 位
0xEFFD	0xEFFD	显示存储器 显存偏移地址高 16 位
0xEFFE	0xEFFE	显示存储器 光标位置
0EFFF	0EFFF	显示存储器 光标闪烁计数器溢出值
0xF000	0xFFFF	显示存储器

表 2.2: 内存地址映射

值得说明的是，我们把 IO 都被映射到了 0xE000 以上的地址，这是为了总线分派器译码的时候方便（地址为 0xE000 以上的地址当且仅当其最高三位都为 1）。

## 第 3 部分

---

# 未来可能的扩展空间

---

可以将汇编器用汇编指令重新实现一遍，使其能够运行在系统自身，从而实现自举（bootstrap）。进一步，我们还可以用汇编语言实现 FAT16、ext2 等文件系统，进而支持从外部存储器（SD 卡）动态加载并运行程序。此外，还可以利用时钟中断实现上下文切换，进而实现多线程等现代操作系统所具有的特性。利用处理器对于外部中断的支持以及串口控制器和 SD 卡控制器的中断请求线，还可以实现异步 IO，提高系统性能。此外，对于 SD 卡控制器，还可以加入更多的错误处理机制，而不是遇到错误就停机。

## 第 4 部分

---

# 致谢

---

感谢李山山老师、刘卫东老师和助教们的大力支持！

## 第 5 部分

---

# 参考文献

---

- [1] 计算机硬件系统实验教程
- [2] 自己动手写 CPU
- [3] CPU 自制入门
- [4] SD Specifications Part 1 Physical Layer Simplified Specification: <https://www.sdcard.org/downloads/pls/>
- [5] How to Use MMC/SDC: [http://elm-chan.org/docs/mmc/mmc\\_e.html](http://elm-chan.org/docs/mmc/mmc_e.html)
- [6] VGA Signal Timing: <http://tinyvga.com/vga-timing>
- [7] Computer Organization and Design The Hardware/Software Interface Fifth Edition
- [8] Computer Systems: A Programmer's Perspective Third Edition
- [9] See MIPS Run Linux Second Edition
- [10] FPGA 设计实战演练（高级技巧篇）
- [11] 深入浅出玩转 FPGA 第 3 版
- [12] FPGA 设计实战演练（逻辑篇）