同济大学计算机科学与技术系

计算机系统结构

三级存储提升实验报告

作业项目 三级存储提升实验

学　　号 1751740

姓　　名 刘鲲

专　　业 计算机科学与技术

授课老师 秦国锋

日　　期 2019/12/06

目录

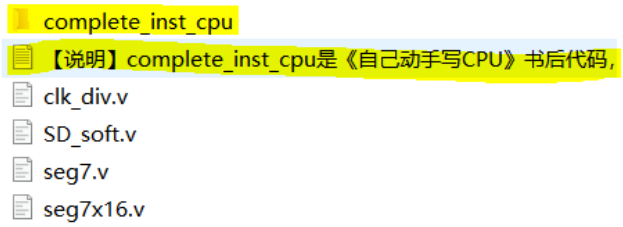
1. 声明

在软件实现 bootloader 方式中，我使用了《自己动手写 CPU》书后代码，技术原因如下：

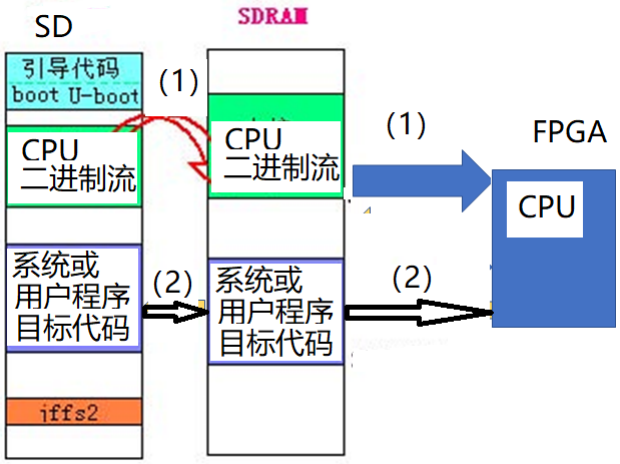
要求的 54 条指令不是完整的 mips 指令，导致编译结果的部分指令无法执行；

由于是软件实现对 SD 卡的控制，SD 卡的频率必定小于 CPU 频率。由于我写的 CPU 频率太低，导致 SD 卡的频率过低，甚至无法初始化。

考虑到临近期末，时间有限，且已完成纯硬件实现的三级存储，故将重点放在跑通软件编写的 BootLoader 上，希望老师谅解！



2. 总体框架



3. 要求

SD 卡中存放流水线 CPU 的二进制流，以及用户程序，N4 板上电自动完成如下的任务：

采用跳线的方式，FPGA 从 SD 卡中获取流水线 CPU 二进制流,并运行该二进制流，FPGA 成为 CPU。

CPU 再按照三级存储方式访问 SDRAM，再由 SDRAM 从 SD 卡中把用户程序目标代码调入到 SDRAM，再由 CPU 把 SDRAM 中用户程序目标代码调入到片内 CACHE 运行。

在本次实验中，我实际完成了两个工程项目：

1. 硬件实现三级存储

2. 软件实现 Bootloader（但没有使用 sdram）

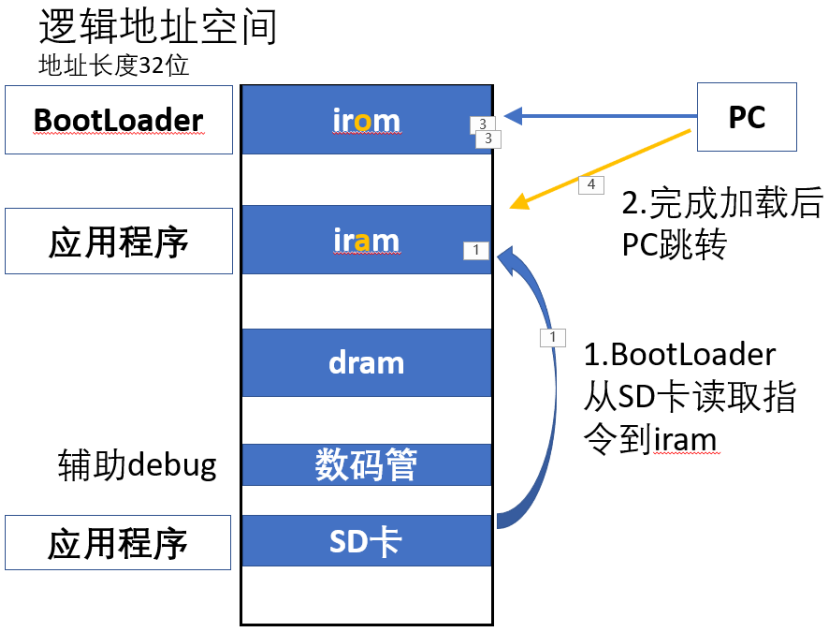
我将分别进行实现方法的介绍。

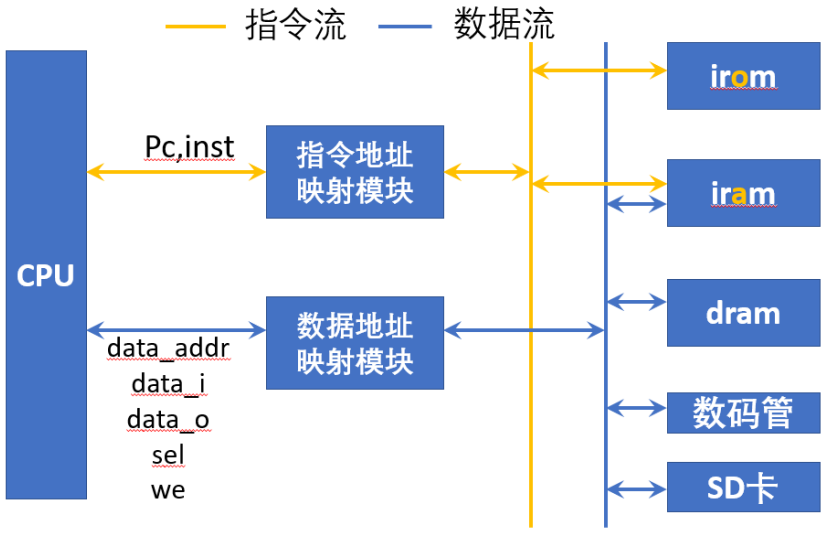
4. 交叉编译实现 BootLoader 实现方法

4.1. 主要思路与模块设计

不使用 Verilog 代码实现 SD 卡的读写，而是将 BootLoader 程序用汇编语言和 c 语言开发，gcc 交叉编译后将其加载到 irom 里，FPGA 加载.bit 以后，执行 irom 里的程序，然后将 SD 卡中的程序搬迁进板内，然后跳转到该地址，执行搬来的程序。

主要框图如下：

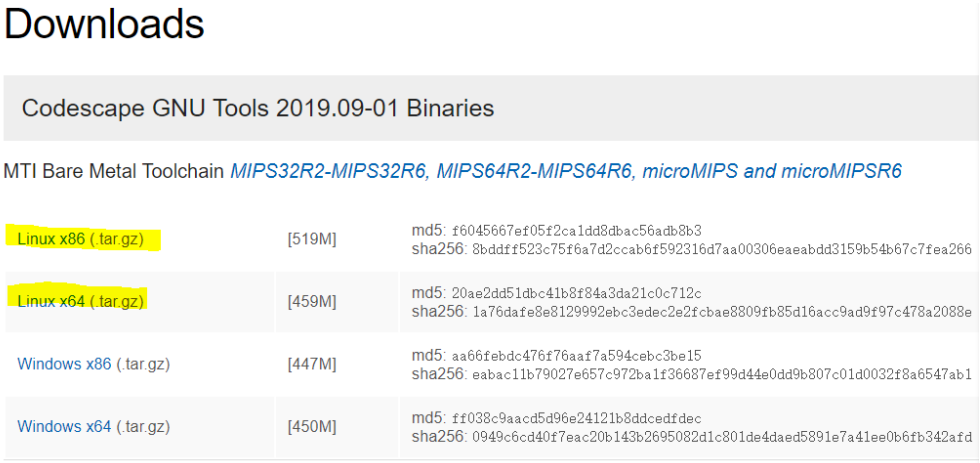




4.2. 交叉编译环境搭建

4.2.1. 交叉编译步骤

首先安装 linux 环境，我的电脑是 Win10 系统，已经有 Ubuntu 18 的 WSL 了，可在 Microsoft Store 下载。然后下载 gcc 编译工具，我使用的是：https://codescape.mips.com/components/toolchain/2019.09-01/downloads.html，根据自己 Linux 系统的位数选择下载，我是 64 位，选择 x64.



Downloads

Codescape GNU Tools 2019.09-01 Binaries

MTI Bare Metal Toolchain *MIPS32R2-MIPS32R6, MIPS64R2-MIPS64R6, microMIPS and microMIPSR6*

Linux x86 (.tar.gz)	[519M]	md5: f6045667ef05f2ca1dd8dbac56adb8b3 sha256: 8bddff523c75f6a7d2ccab6f592316d7aa00306eaeabdd3159b54b67c7fea266
Linux x64 (.tar.gz)	[459M]	md5: 20ae2dd51dbc41b8f84a3da21c0c712c sha256: 1a76dafe8e8129992ebc3edec2e2fcbae8809fb85d16acc9ad9f97c478a2088e
Windows x86 (.tar.gz)	[447M]	md5: aa66febdc476f76aaf7a594cebc3be15 sha256: eabac11b79027e657c972ba1f36687ef99d44e0dd9b807c01d0032f8a6547ab1
Windows x64 (.tar.gz)	[450M]	md5: ff038c9aacd5d96e24121b8ddcedfdec sha256: 0949c6cd40f7eac20b143b2695082d1c801de4daed5891e7a41ee0b6fb342afd

注意，网上有很多推荐装 mips-linux-gnu 进行交叉编译的，但是这个编译器是适用于 Linux 应用的，可能和 c 语言的逻辑不相符，不推荐使用。我使用的是上面的 Bare Metal。

下载到 linux 后，cp 到/opt 目录下解压：tar xvf FileName.tar.gz，FileName 是下载下来的文件名，很长一串。

然后，cd 到/home/username（你的用户名），添加到当前用户的环境变量：用 vim 编辑隐藏文件.bashrc，在末尾添加一条语句：

export PATH="$PATH:/opt/mips-mti-elf（解压成功后的文件夹名）/bin"

使用下列命令使其生效：

source ./.bashrc

然后终端键入 mips-mti-elf-，两次 TAB 会自动补全，会显示能用什么命令，即说明安装成功。

4.2.2. Makefile 书写

这部分需要额外补充 Makefile 语法和 mips-mti-elf-的使用方法，参考文献【】给出连接，《自己动手写 CPU》的第 4 章 4.4 节亦可参考，本报告不再赘述。仅列出 Makefile 代码并给出主要说明，代码已附在附录中。

```
.PHONY:clean
CROSS_COMPILE = mips-mti-elf
CC = ${CROSS_COMPILE}-gcc
AS = ${CROSS_COMPILE}-as
LD = ${CROSS_COMPILE}-ld
OBJCOPY = ${CROSS_COMPILE}-objcopy
#OBJDUMP = mips-linux-gnu-objdump
OBJDUMP =  ${CROSS_COMPILE}-objdump
RM = del
COE = BOOT.coe

all: BOOT.c init.s
#-mno-abicalls BOOT.c
    $(CC) -c -std=c99 -mips32 BOOT.c
    $(AS) init.s -o init.o
    $(LD) -T ram.ld init.o BOOT.o -o BOOT
    $(OBJCOPY) -j ".text" -O binary BOOT BOOT.bin
    $(OBJDUMP) -b binary -m mips -D BOOT.bin -EB
    #./Bin2Mem.exe -f BOOT.bin -o ${COE}
    echo 'memory_initialization_radix = 16;' > ${COE}
    echo 'memory_initialization_vector =' >> ${COE}
    hexdump -v -e '4/1 "%02x" "\n"' BOOT.bin >> ${COE}

clean:
    -$(RM) BOOT.o init.o BOOT BOOT.bin
```

有一些非 Makefile 语法的部分说明：

1.上图中的-EB 是指用大端模式进行反编译，目的是看一下我们的指令有没有问题。如果这里不写-EB 的话，那么就会用默认的小端模式进行反编译，出来的东西肯定是错的。

2.上图中三行与 COE 相关的语句，是将二进制文件 BOOT.bin 转成.coe 可直接供我们加载到 rom 里的格式。

反编译执行完以后，效果如下：

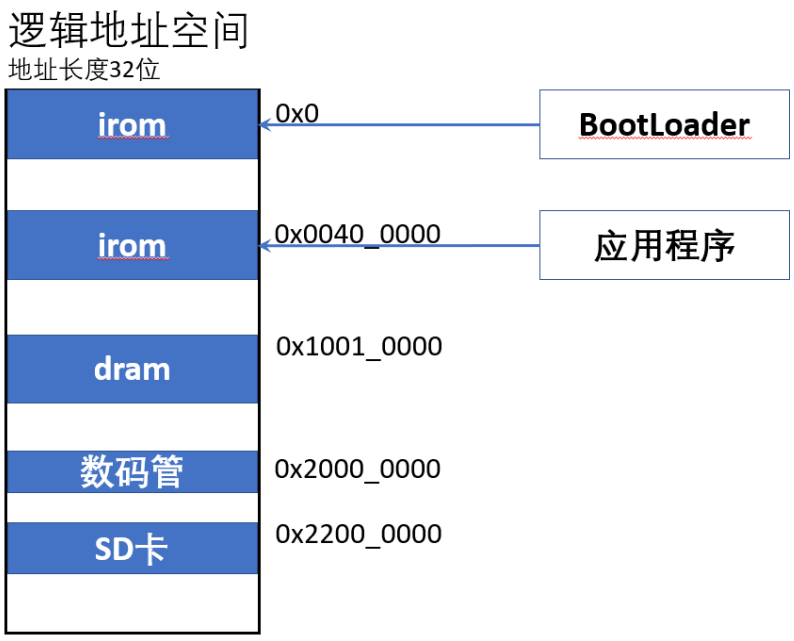
```
 8e4:    afc20014        sw      v0,20(s8)
 8e8:    afc00020        sw      zero,32(s8)
 8ec:    10000004        b       0x900
 8f0:    00000000        nop
 8f4:    8fc20020        lw      v0,32(s8)
 8f8:    24420001        addiu   v0,v0,1
 8fc:    afc20020        sw      v0,32(s8)
 900:    8fc20020        lw      v0,32(s8)
 904:    28422710        slti    v0,v0,10000
 908:    1440fffa        bnez    v0,0x8f4
 90c:    00000000        nop
 910:    8fc2001c        lw      v0,28(s8)
 914:    24420001        addiu   v0,v0,1
 918:    afc2001c        sw      v0,28(s8)
 91c:    8fc2001c        lw      v0,28(s8)
 920:    28420080        slti    v0,v0,128
 924:    1440ffdb        bnez    v0,0x894
 928:    00000000        nop
 92c:    00001025        move    v0,zero
 930:    03c0e825        move    sp,s8
 934:    8fbf022c        lw      ra,556(sp)
 938:    8fbe0228        lw      s8,552(sp)
 93c:    27bd0230        addiu   sp,sp,560
 940:    03e00008        jr      ra
 944:    00000000        nop
#./Bin2Mem.exe -f BOOT.bin -o BOOT.coe
echo 'memory_initialization_radix = 16;' > BOOT.coe
echo 'memory_initialization_vector =' >> BOOT.coe
hexdump -v -e '4/1 "%02x" "\n"' BOOT.bin >> BOOT.coe
lk@DESKTOP-K4AJDO:/mnt/c/Users/95223/Desktop/test_cross_compile$
```

可以看到，每行指令都会找到它对应的解释，这能够帮助我们 debug 以及分析问题。

4.2.3. 代码段地址、存储器地址统一编址

这一步是第一步，也是很重要的一步。核心思想是，所有的代码、数据和存储器都在逻辑地址空间中有一个 32 位的地址，C 语言编写的就是对这些地址的操作。

以下给出我的地址编排方案：



4.2.4. Bootloader 后指令跳转

由上图可得，在执行完 bootloader 后，需要将 pc 改为 0x0040 0000，于是我们在编译时需要在 0x0000 0000 附近加上一段跳转指令到 bootloader 的 main 函数，以及返回时跳转回 0x0040 0000：

```
.org 0x00000000
.global _start
.set noat
_start:
    lui $at,0x1000
    ori $at,$at,0x1F00
    add $sp,$zero,$at
    jal main
    nop
    lui $at,0x40    # 0040 0000
    jr $at
    nop
```

4.2.5. 实现 C 代码控制管脚电平高低

根据前面逻辑地址的排布表，我们通过宏定义编写控制管脚高低的宏函数：

```
#define SD_CS 0x22000000
#define SD_CLK 0x22000001
#define SD_DATAIN 0x22000002
#define SD_DATAOUT 0x22000003
```

```
#define SD_HALF_CLK_LEN 0 //延时，这里是0，否则跟不上频率
// 宏定义
#define SD_CLK_UP()  *((uchar *)SD_CLK) = 1;

#define SD_CLK_DOWN()  *((uchar *)SD_CLK) = 0;

#define SD_DATAIN_UP()  *((uchar *)SD_DATAIN) = 1;

#define SD_DATAIN_DOWN()  *((uchar *)SD_DATAOUT) = 0;

#define SD_CS_UP()  *((uchar *)SD_CS) = 1;

#define SD_CS_DOWN() *((uchar *)SD_CS) = 0;
```

如此，我们就可以用软件方式一一对应地写 c 语言代码，实现 SD 卡的协议。以时钟为例：

```
void SD_send_clk()
{
    SD_CLK_DOWN();
    DELAY_HALF_CLK();
    SD_CLK_UP();
    DELAY_HALF_CLK();
}
```

```
for (int i = 0; i < 80; i++)
    SD_send_clk();
```

4.2.6. SD 卡接口

但仅仅是这样还不够。我们是将对存储单元写入 0/1 转换成高低电平的信号，这种转换还需要一个接口：

Verilog 代码如下：

```
module SD_soft(
    input clk,  //写入时钟
    input rst,
    input we,   //写使能
    input [3:0] sel_i, // 位选信号
    input [31:0] data_i, //写入数据
    output [31:0] data_o, // 将它视作一个4字节的
    // sd相关
    output reg SD_cs, //片选, addr = 0
    output reg SD_clk, //时钟, addr = 1
    output reg SD_datain, //数据输入, addr = 2
    input SD_dataout //数据输出, addr = 3
);
```

```
assign data_o = {7'b0, SD_cs, 7'b0, SD_clk, 7'b0, SD_datain, 7'b0, SD_dataout};
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        SD_cs = 1'b0;
        SD_clk = 1'b0;
        SD_datain = 1'b0;
    end
    else if(we) begin
        if (sel_i[3] == 1'b1) begin
            SD_cs <= (data_i[31:24] != 8'b0);
        end
        if (sel_i[2] == 1'b1) begin
            SD_clk <= (data_i[23:16] != 8'b0);
        end
        if (sel_i[1] == 1'b1) begin
            SD_datain <= (data_i[15:8] != 8'b0);
        end
    end
end
```

除了存储单元的名字以外，与一个 32 位（4 字节）存储单元没有区别！换言之，对 CPU 来说，接口是存储单元，对 FPGA 来说，是管脚信号。

4.2.7. 注意大小端的问题

CPU 有大端、小端模式，而且就算是小端模式，也就是低地址放低位，但是交叉编译器想得不一定和预期的一样，用同样的编译工具，反编译以后就是混乱的。在完成大作业的过程中，我也并没有找到比较好的方法。主要还是靠尝试和波形图。

如我的 makefile：

```
.PHONY:clean
CROSS_COMPILE = mips-mti-elf
CC = ${CROSS_COMPILE}-gcc
AS = ${CROSS_COMPILE}-as
LD = ${CROSS_COMPILE}-ld
OBJCOPY = ${CROSS_COMPILE}-objcopy
OBJDUMP =  ${CROSS_COMPILE}-objdump
RM = del
COE = BOOT.coe

all: BOOT.c init.s
    $(CC) -c -std=c99 -mips32 BOOT.c
    $(AS) init.s -o init.o
    $(LD) -T ram.ld init.o BOOT.o -o BOOT
    $(OBJCOPY) -j ".text" -O binary BOOT BOOT.bin
    $(OBJDUMP) -b binary -m mips -D BOOT.bin -EB

    echo 'memory_initialization_radix = 16;' > ${COE}
    echo 'memory_initialization_vector =' >> ${COE}
    hexdump -v -e '4/1 "%02x" "\n"' BOOT.bin >> ${COE}

clean:
    -$(RM) BOOT.o init.o BOOT BOOT.bin
```

在反编译的时候要求使用大端模式(黄色荧光处)，出来才是正常结果，其它情况下默认小端-EL 编译。但是这明明是一套工具，为什么要看正常结果反而要加-EB？这个问题我暂无时间探究。

再如发送指令时：

```
// cmd
#define CMD0 0x400000000095ULL  // 6个Byte
```

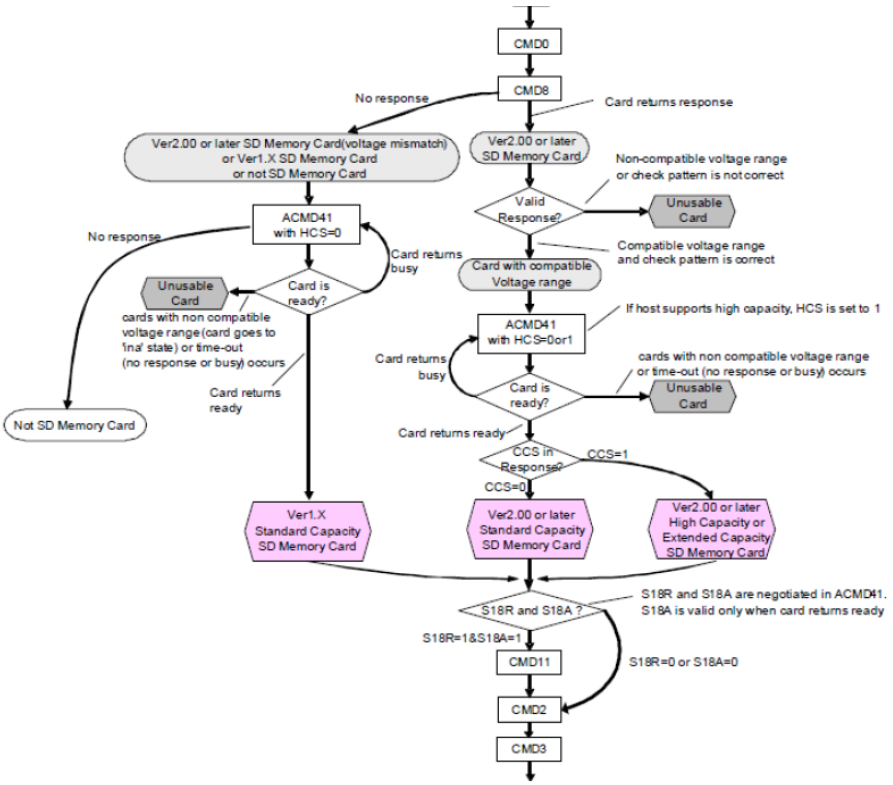
发送指令时这么做：

```
uchar *p = (uchar *)(&cmd) + 2; //无解!
for (int i = 0; i < 6; i++){
    SD_send_byte(*(p + i));
}
```

荧光处的+2 是我反反复复看波形图才发现的，我仍然没搞明白宏定义的数据为什么会变成大端模式（低地址存高位），而明明编译的时候是小端模式？这个问题同样因为时间问题暂无时间深究。

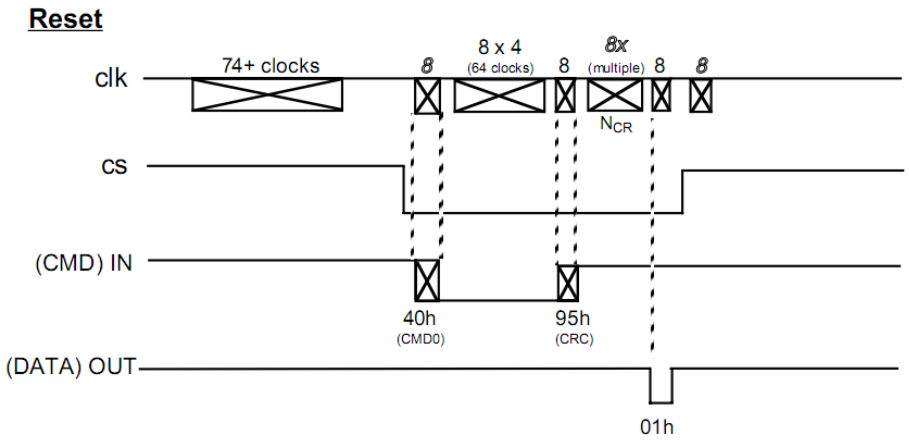
4.3. SD 卡时序说明以及软件编写

4.3.1. SD 卡初始化时序说明



本实验使用的是 SDHC。故以下选区的指令以 SDHC 为准。

4.3.2. 初始化 CMD0



我们将步骤和软件代码一一对应即可，函数名字都很容易理解。

```c
void write_CMD(ullong cmd, uchar *res, int n)
{
    SD_CS_DOWN();
    // 为什么宏定义的单元编译后是大端模式？
    // 高位在低地址？不懂，测试死我了
    uchar *p = (uchar *)(&cmd) + 2;
    for (int i = 0; i < 6; i++){
        SD_send_byte(*(p + i));
    }
    SD_DATAIN_UP();

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

    for (int i = 0; i < n; i++){
        res[i] = SD_get_byte();
    }

    if (*p != 0x51 && *p != 0x58) {
        SD_CS_UP();
        for (int i = 0; i < 8; i++)
            SD_send_clk();
    }
    //*((int *)SEG_DA) = *p; // debug
}
```

其它功能如接收、发送 byte 类似，可看源代码。

4.3.3. 发送 CMD8

为了区别 SD 卡是 2.0 还是 1.0，发送 SD2.0 独有命令 CMD8：

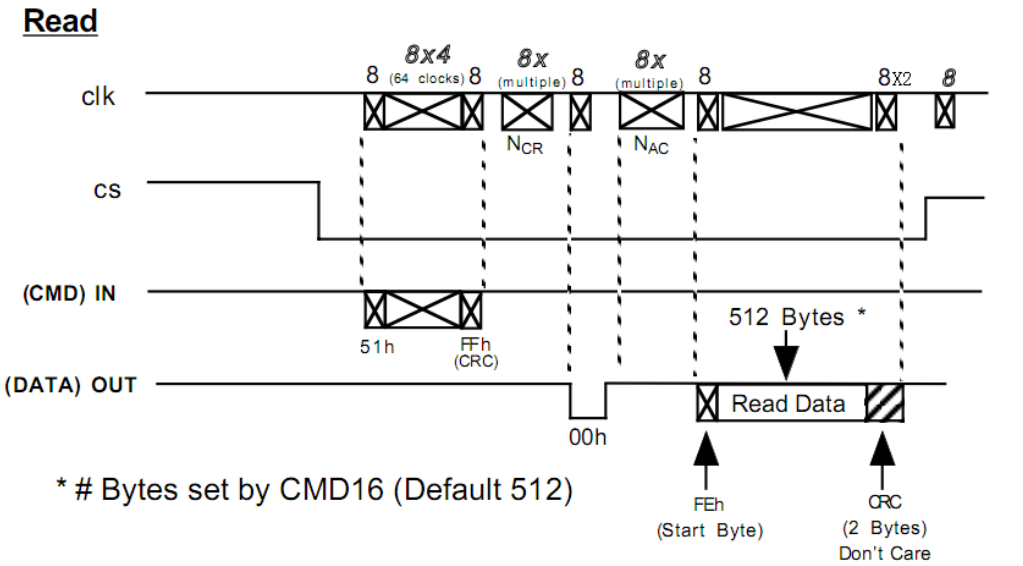
CMD8={8'h48, 8'h00, 8'h00, 8'h01, 8'haa, 8'h87}

返回 0x01，初步判断为 2.0 卡。

4.3.4. 循环发送 CMD55 和 ACMD41

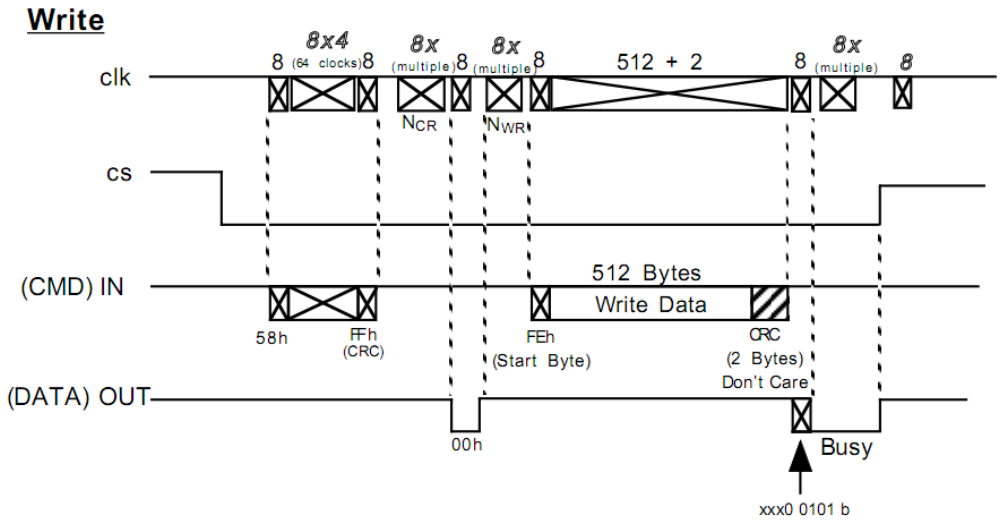
进一步发送命令循环发送 CMD55={8'h77, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff}和 ACMD41={8'h69, 8'h40, 8'h00, 8'h00, 8'h00, 8'hff}，直到返回 0x00，完成初始化。

4.3.5. 读操作



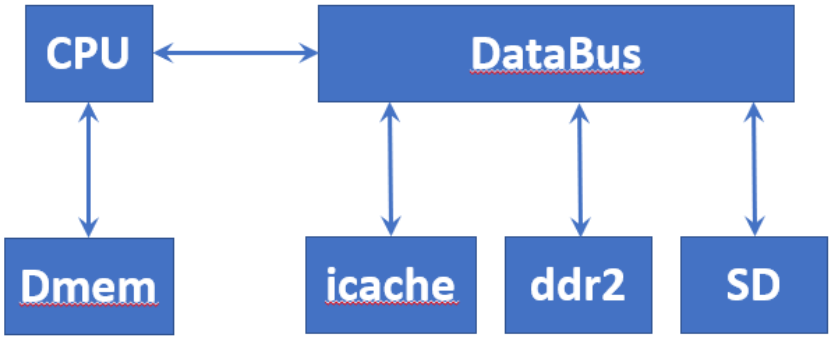
SD 卡提供连续读指令，但我们这里只使用读单块指令，连续多次读单块就是连续读了。

4.3.6. 写操作



5. 硬件三级存储实现方法

5.1. 系统模块设计



将 DataBus 专门作为指令的调度单元，也就是相当于一个顶层的状态机，icache、ddr2、SD 卡之间的数据由 DataBus 进行控制。

5.2. DDR2 时序说明

5.2.1. 写操作时序

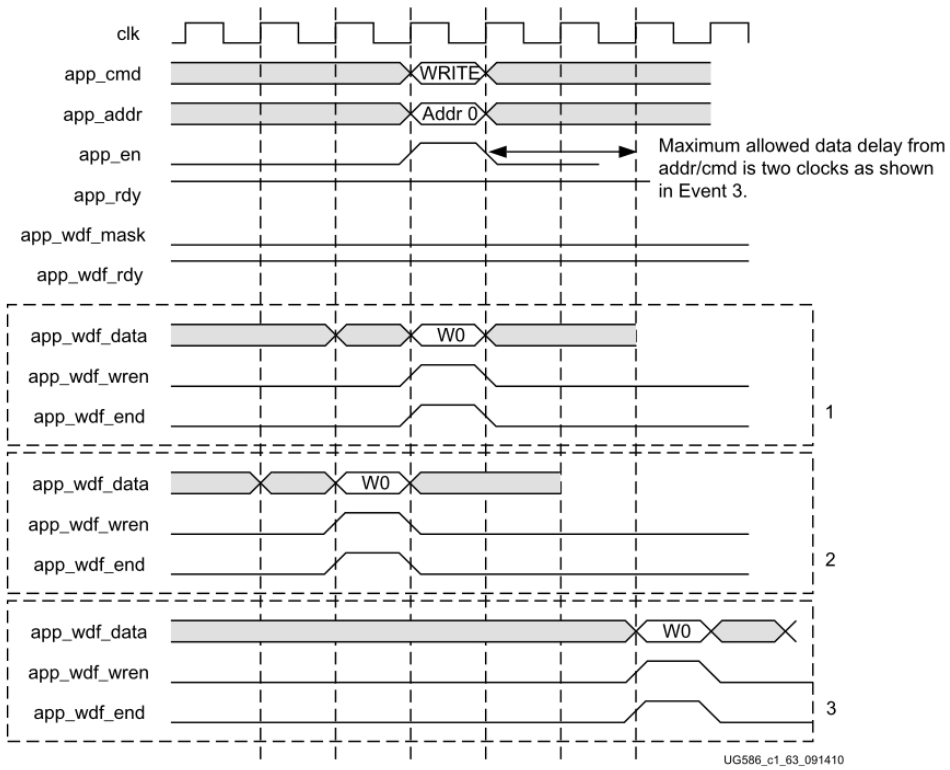


Figure 1-75: **4:1 Mode UI Interface Write Timing Diagram (Memory Burst Type = BL8)**

5.2.2. 读操作时序

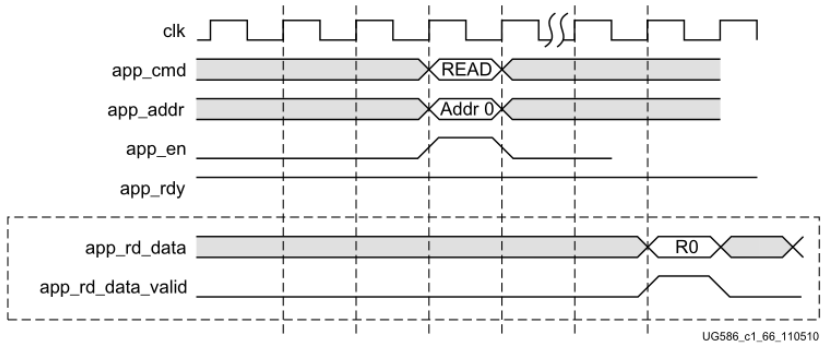


Figure 1-81: **4:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL8)**

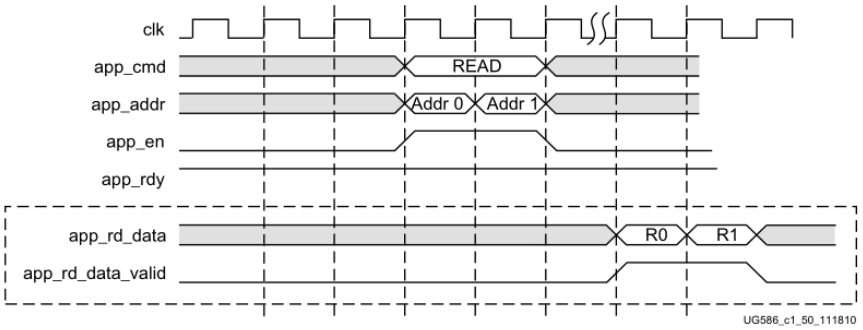


Figure 1-82: **2:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL4 or BL8)**

5.2.3. MIG 二次封装

DDR2 信号非常复杂，但有 IP Core：MIG，能简化信号：

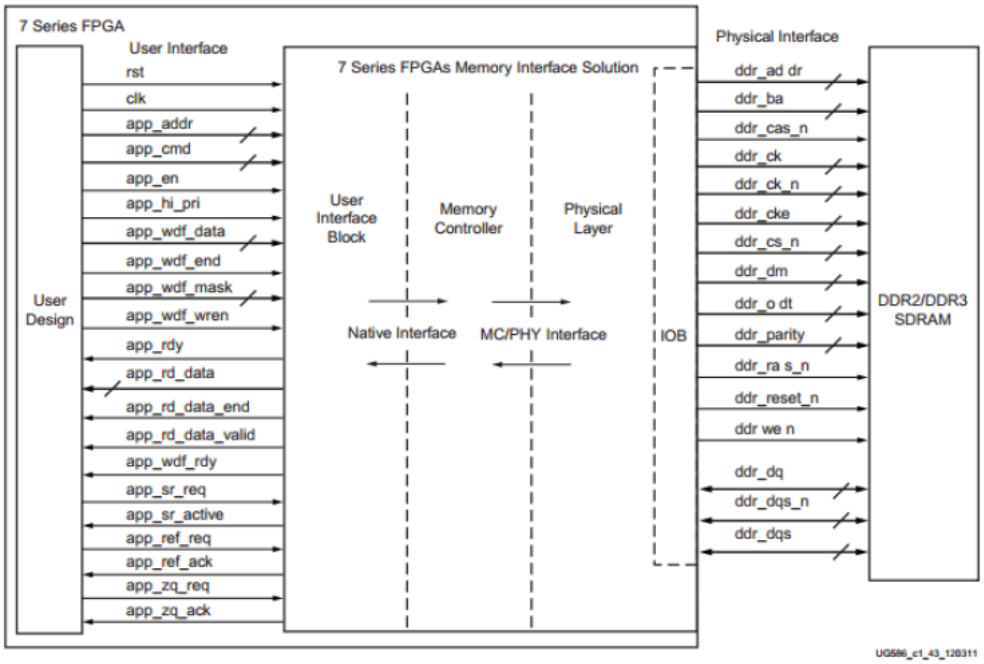


Figure 1-31: **7 Series FPGAs Memory Interface Solution**

但是仍然很复杂，所以我们还需要进一步封装。这个我在下一个小结进行解释。

这里我们需要注意，不同于 Ram 的 IP core，DDR2 的读写都大于一个周期，所以必须要有请求 req 和答复 ack 信号，与 DDR2 控制器进行交互。

我参考了实验参考书的 sealedDDR，书中二次封装的接口设计非常好，如下：

```
module sealedDDR (
    input clk100mhz,
    input rst,
    input [31:0] addr_to_DDR,
    input [31:0] data_to_DDR,
    input read_write,
    output [31:0] data_from_DDR,
    output reg busy,
    output reg done,
    output ddr_start_ready,
```

其它还有 ddr2 的相关信号，略去不写，解释说明如下：

其中最基本的，也就是无论是什么样的存储器都至少有的信号：

信号名	含义
addr_to_DDR	读写地址线
data_to_DDR	写数据线
data_from_DDR	读数据线
read_write	读写使能

这四个信号是***最最基本***的信号。

5.3. 对于存储器，什么样的封装是好封装？

这里我们需要明确，对于一个存储器，什么样的封装是好的封装？

我的结论是：**对于一个在一个边沿只能做读或者写的单口存储器，需要读、写数据线、写使能、地址线；对于一个读写需要多个周期的，需要 req、ack 信号。**

根据实验指导书中的参考代码，对三级存储的三个器件进行说明：

5.3.1. Cache

注意，我将 cache 的标志位和映射地址的记录表放到了 DataBus 里，在数据总线中进行缓存命中不命中的判断。

Cache 这里我们使用 ram 的 IP core 进行模拟：

```
// 第一级 cache
cache U_cache(
  .clk(clk100mhz),
  .a(cache_addr_toIMEM[8:0]),
  .d(cache_data_toIMEM),
  .we(cache_read_write),
  .spo(cache_data_fromIMEM)
);
```

其中最基本的，也就是无论是什么样的存储器都至少有的信号：

信号名	含义
cache_addr_toIMEM	**读写地址线**
cache_data_toIMEM	**写数据线**
cache_data_fromIMEM	**读数据线**
cache_read_write	**读写使能**

IP core 的 ram 是**在一个时钟周期里能读写完毕的部件**，所以很简单。

5.3.2. SD 卡封装

参考实验指导书中提供的代码：

```
module SD (
    input clk100mhz,
    input rst,

    // sd卡的读写信号
    input sd_mem_read_write,//useless 但
    input [31:0] sd_mem_addr,
    input [31:0] sd_mem_data_toSDMEM, //
    output reg sd_mem_ready,
    output [31:0] sd_mem_data_fromSDMEM,
```

其中最基本的，也就是无论是什么样的存储器都至少有的信号：

信号名	含义
sd_mem_addr	**读写地址线**
sd_mem_data_toSDMEM	**写数据线**

sd_mem_data_fromSDMEM	**读数据线**
sd_mem_read_write	**读写使能**

然后在这四个基本信号的基础上加 sd_mem_ready 这一个表示加载指令完成的标志。数据放在 SD 卡的缓存里。

5.3.3. DDR2 封装

实验指导书中同样提供了代码：

```
module sealedDDR (
    input clk100mhz,
    input rst,
    input [31:0] addr_to_DDR,
    input [31:0] data_to_DDR,
    input read_write,
    output [31:0] data_from_DDR,
    output reg busy,
    output reg done,
    output ddr_start_ready,
```

省略 ddr2 的其它一些信号

基本信号：

信号名	含义
addr_to_DDR	**读写地址线**
data_to_DDR	**写数据线**
data_from_DDR	**读数据线**
read_write	**读写使能**

根据我之前得出的结论，由于 ddr2 不是一个周期就能读写的，所以一定要有 req、ack 信号，尤其是 ack。在书上的代码的设计中，加上了如下信号：

信号名	含义
busy	**忙**
done	**上一个操作已完成**
ddr_start_ready	**初始化完成**

当一个存储器需要多个时钟周期读写时，会有以下两个问题：

正在进行读写时，其它的读写请求到达，要阻塞 -> busy

结束读写后，通知取数据方拿数据 -> done

对于这两种需求，可以通过 busy 和 done 信号进行处理。在状态机中进行阻塞，如：

```
WAIT_DDR_DONE:
if (!done)
    state <= WAIT_DDR_DONE;
```

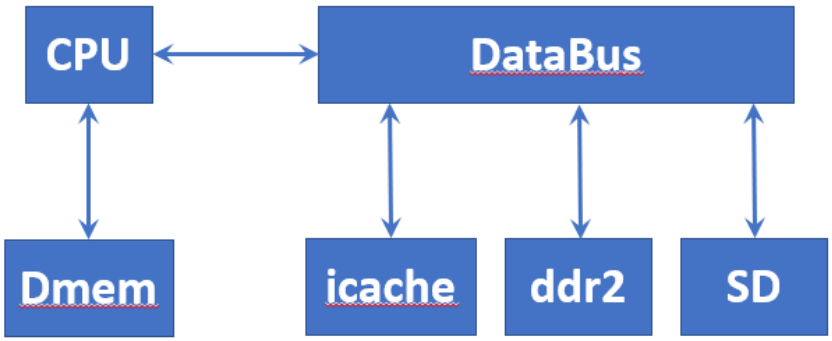
等等。

5.4. 数据总线

本项目参考了实验指导书中给出的 DataBus 模块，对此进行分析：

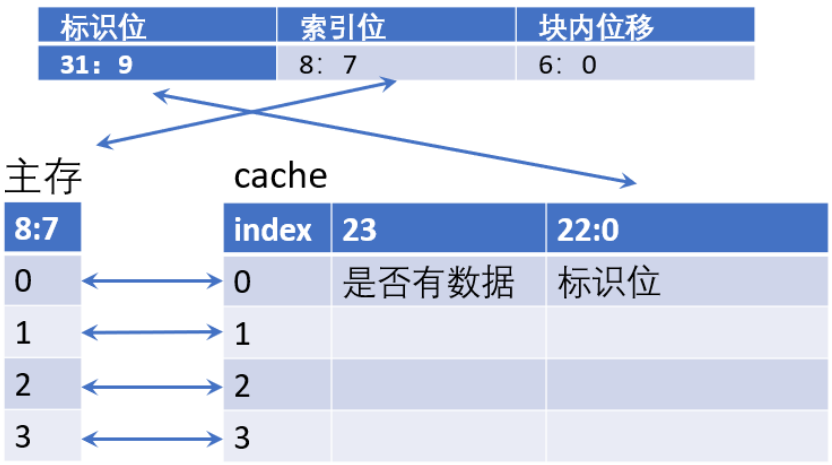
```
module DataBus (
    input clk100mhz,
    input rst,
    input [31:0] cpu_addr,
    input cpu_read_write,
    output reg bootdone,

    // 与CPU的交互信号
    output [31:0] cpu_data_toCPU,
    output reg DataBus_busy,
    output reg DataBus_done,

    // 与cache的交互信号
    input [31:0]cache_data_fromIMEM,
    output  [31:0]cache_addr_toIMEM,
    output  [31:0]cache_data_toIMEM,
    output reg cache_read_write,

    // 与DDR的交互信号
    input [31:0]ddr_data_fromDDR,
    input ddr_busy,
    input ddr_done,
    output [31:0]ddr_addr_toDDR,
    output [31:0]ddr_data_toDDR,
    output reg ddr_read_write,
    input ddr_start_ready,

    // 与SD卡的交互信号
    output reg sd_mem_read_write,
    output  [31:0]sd_mem_addr,
    input  [31:0]sd_mem_data_fromSDMEM,
    output reg [31:0]sd_mem_data_toSDMEM,
    input sd_mem_ready,
```

可以看到，上一小结封装的模块的所有端口都和 DataBus 相连，其中有一组端口与 CPU 相连。

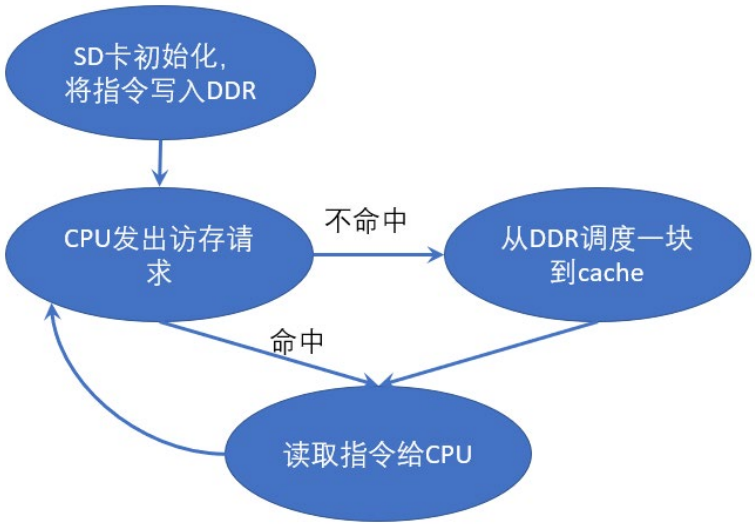


5.4.1. Cache 映射关系

按照参考书的指导，我们采用 4 块 cache，并使用比较好实现的直接映射：



5.4.2. 状态机

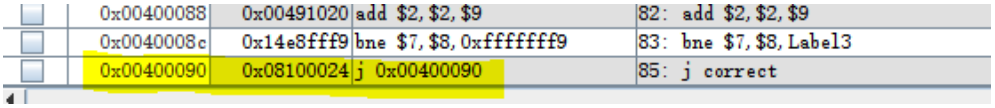


6. 实际运行与验证

本次验证分为两个，一个是硬件实现三级存储，一个是软件实现 Bootloader。由于临近期末，时间关系，无法将两个实验合并，真正达成用软件实现三级存储。

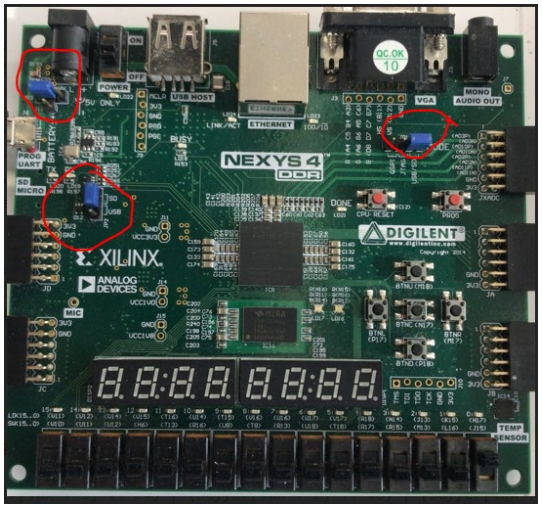
6.1. 硬件实现三级存储结果

这里我们使用流水线 CPU 的测试程序，可见附件。该程序最后一条是死循环语句：

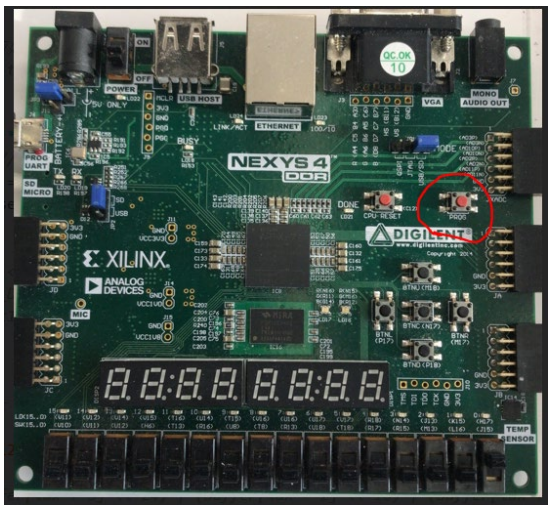


将汇编文件转成十六进制文件以后，全选复制；将.bit 放到 SD 卡中，然后用 WinHex 在逻辑 80 扇区（我这张卡对应的物理扇区是 8272）写入指令（右键，从剪贴板写入，写

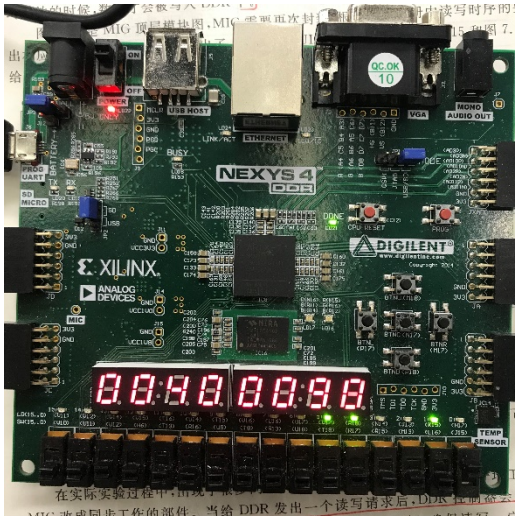
入方式为 16 进制）。然后放置好跳线（也就是蓝色的小帽子）：



之后 UART 口连接上电源，可以同实验指导书一样连接上充电宝。如果连上电脑供电的话，记得按 PROG 按钮（下图红圈）：

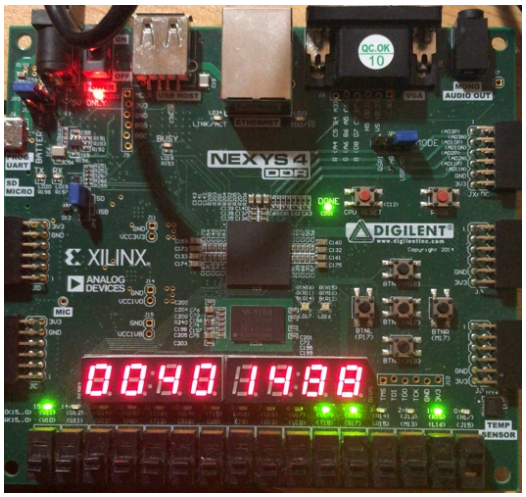


然后稍等一会，就能看到七段数码管显示读到的指令：



0x40_0090，与预期一致。

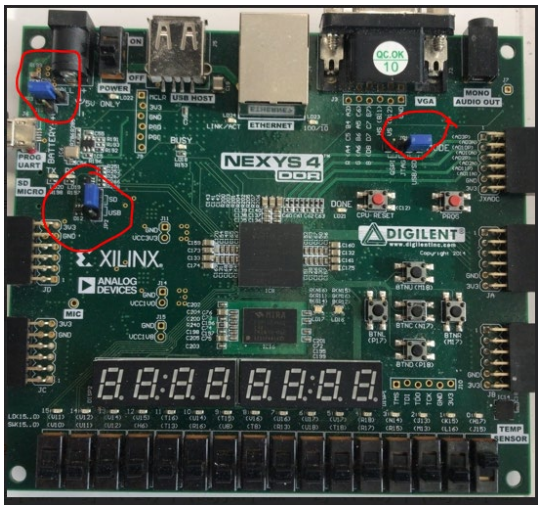
然后换一张 sd 卡，其中逻辑 80 扇区是上学期计组的测试文件，结果如下：



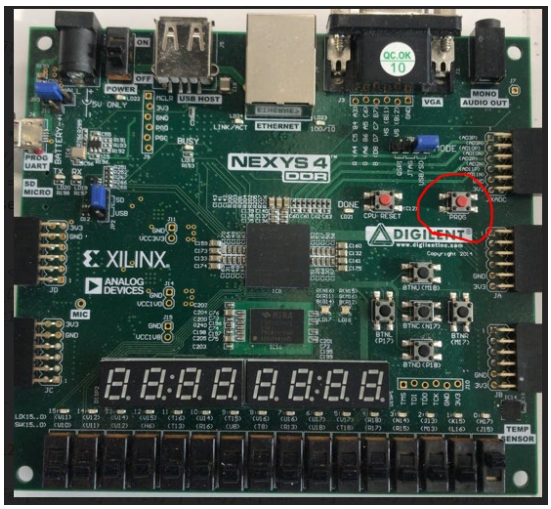
测试成功！可以看到，pc-0x0040 0000 后的值超过了 0x200，说明**读到了多个扇区，运行了多个扇区的指令。**

6.2. 软件实现 Bootloader 结果

将.bit 放到 SD 卡中，然后用 WinHex 在逻辑 80 扇区（我这张卡对应的物理扇区是 8272）写入指令（右键，从剪贴板写入，写入方式为 16 进制）。然后放置好跳线（也就是蓝色的小帽子）：



之后 UART 口连接上电源，可以同实验指导书一样连接上充电宝。如果连上电脑供电的话，记得按 PROG 按钮（下图红圈）：



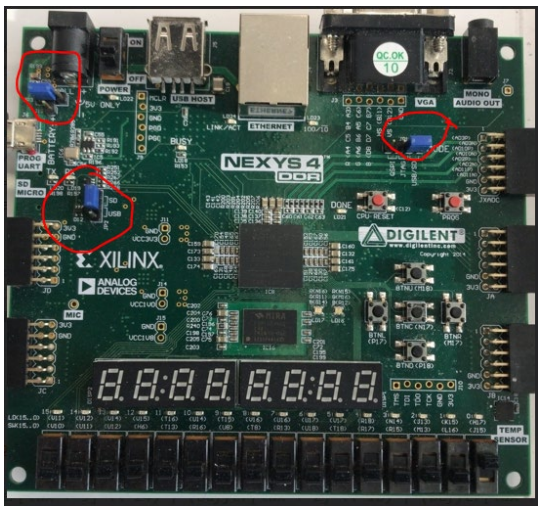
然后稍等一会，就能看到七段数码管显示读到的指令：

Drive I:	100% free
File system:	FAT32
Default Edit Mode State:	original
Undo level:	0
Undo reverses:	n/a
Alloc. of visible drive space:	
Cluster No.:	n/a
	Reserved sector
Snapshot taken	4 hours ago
Logical sector No.:	80
Physical sector No.:	8,272
Used space:	16.0 KB
	16,384 bytes
Free space:	7.2 GB
	7,723,794,432 bytes
Total capacity:	7.2 GB
	7,740,588,032 bytes
Bytes per cluster:	4,096
Free clusters:	1,885,692
Total clusters:	1,885,696

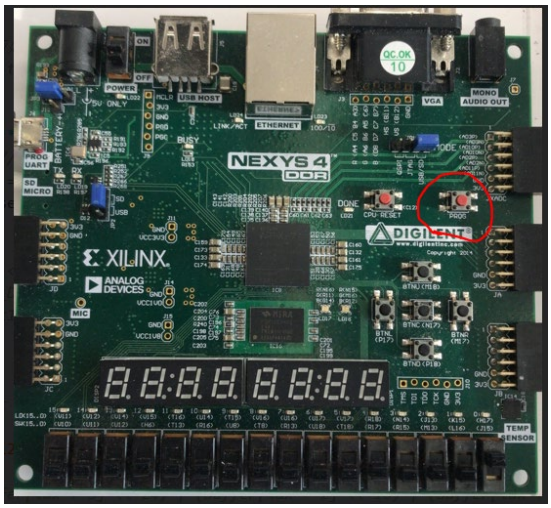
然后在 bootloader 中，指明物理扇区号（注意一定是物理）是 8272.

```
uint code_sector_cur = 8272;
SD_read_sector(code_sector_cur, buffer);  // 32'h0040_A000
```

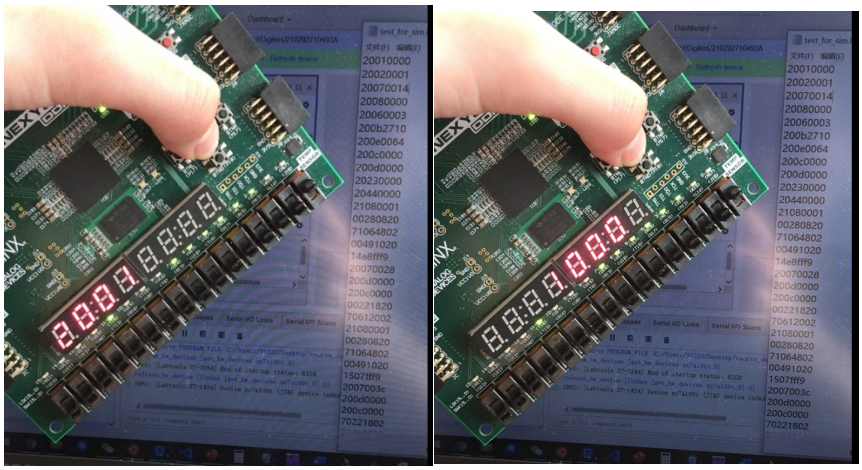
Make 生成新的 coe，加载到 irom 中，生成比特流，放到 sd 卡中，然后放置好跳线（也就是蓝色的小帽子）：



之后 UART 口连接上电源，可以同实验指导书一样连接上充电宝。如果连上电脑供电的话，记得按 PROG 按钮（下图红圈）：



然后稍等一会，就能看到七段数码管显示读到的指令：



将读到的指令和 SD 卡中的比对，是 20010000，和第一条指令一致，之后可以继续比对下去。

以上两张图是从录像中截取的，由于手机录像频率和七段数码管频率的关系，数码管闪烁较为厉害，录像效果较差（录像作为附件提交），直接下板肉眼观察更清晰。

```
uint idx = 0;
for (int j = 0; j < 128; j++) //一个扇区128个字(512/4)，一个字4 Byte
{
    //*p_text = (buffer[idx + 3] << 24) | (buffer[idx + 2] << 16) | (buffer[idx + 1] << 8) | buffer[idx];
    *p_text = (buffer[idx] << 24) | (buffer[idx + 1] << 16) | (buffer[idx + 2] << 8) | buffer[idx + 3];
    *((uint *)SEG_DA) = *p_text;
    for(int k=0;k<1000000;k++) ///debug 可以看到具体指令
        ;
    p_text++;
    idx = idx + 4;
}

return 0;
```

在荧光处可以调整指令在数码管上的显示时间，make 后生成新的 coe，再重新生成 irom 重复以上步骤即可。

在 Bootloader 中，我设置了开关控制 led：

```
assign led = sw[0] ?
        (sw[1] ? {debug_once_was, debug_i_data[14:0]} : inst_addr[15:0]): inst_addr[31:16];
```

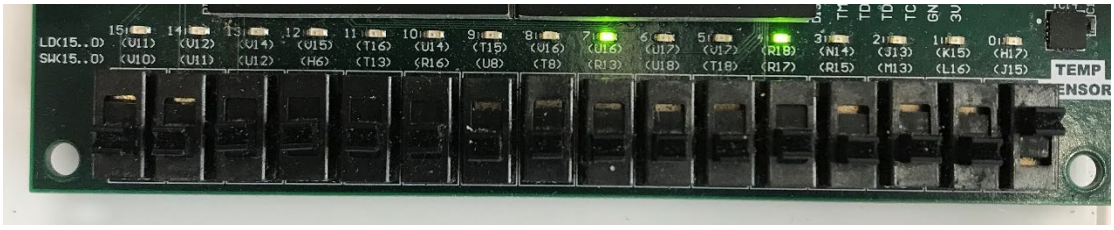
Sw[1:0]==3 时，显示 pc 低 16 位；==1 时显示 pc 高 16 位。

测试程序末尾加入无条件跳转：

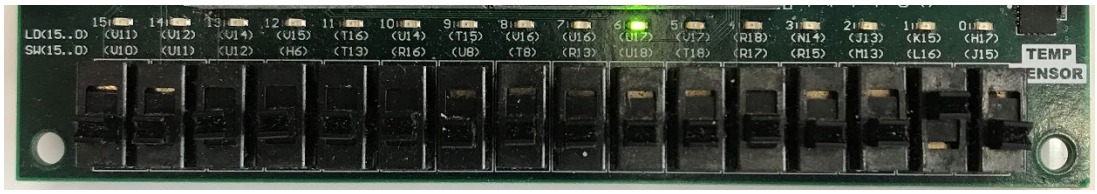
```
0x00400088  0x00491020  add $2, $2, $9          82:  add $2, $2, $9
0x0040008c  0x14e8fff9  bne $7, $8, 0xfffffff9  83:  bne $7, $8, Label3
0x00400090  0x08100024  j 0x00400090            85:  j correct
```

下板后，两种开关下数码管显示如下：

Sw[0]=1, sw[1]=0 如下，说明 pc[15:0]==0b0000_0000_1001_0000=0x0090;



Sw[0]=0, sw[1]=1，如下，说明 pc[31:16]==0b0000_0000_0100_0000=0x0040;



两者拼凑起来就是 0x0040_0090，也就是 j 指令。

测试成功！

7. 遇到的问题与解决

这一部分写一些上面几个小结没有囊括的问题，统一在这里说明。

7.1. 换一张 SD 卡就好

在测试过程中，我曾经遇到仅仅是更换 SD 卡，bug 就消失的情况。这也是硬件 debug 最令人头疼的地方。网上一个博客也提及了这个问题，所以有时候跑不通可能是 SD 卡的问题。我算是运气比较好，手头还有一张卡，不然我就没了。

7.2. 注意 SD 卡的版本

不同 SD 卡版本，初始化方法不同，这个一定要注意！

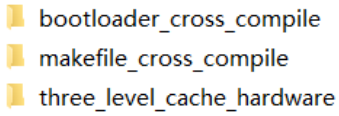
7.3. 重新编译库，就可以用 ip core 仿真

在前仿真的时候遇到了 ip 核无法仿真的问题，后来发现是忘记在 simulation setting 里设置编译库了。库编译的方法网上有。

8. 源程序代码

因为本实验其实由两个大项目组成，为避免报告过长，所有代码见附件。

src 结构如下：



bootloader_cross_compile：软件实现 bootloader

makefile_cross_compile：交叉编译源材料

three_level_cache_hardware：纯硬件实现三级存储

9. 心得体会

通过这次提升实验，我较好地利用了以前的成果，再加上一些新的复杂的老师上课没有讲过的，这部分比较消耗时间。

同时我发现，各个商家生产的 SD 卡可能各有各的区别。在测试过程中，我曾经遇到仅仅是更换 SD 卡，bug 就消失的情况。这也是硬件 debug 最令人头疼的地方。在软件实现交叉编译 bootloader 过程中，我使用上了七段数码管，仅仅是一句赋值语句就可以将数据显示在七段数码管上，非常方便，大大加快了我 debug 的速度。

做这个项目的时候临近期末，事务繁多，而且硬件 debug 很慢，有时候完全是硬着头皮在 debug，呼吸困难，头晕发懵，后面还有一堆 ddl 追着我，实在是身心俱疲。尤其是当我发现 bug 无法解释的时候，更是身心俱疲了。不过好在运气很好，最关键的 bug 在于更换 sd 卡的那一步。在那一步之前，反复看代码，就是搞不懂哪里 sd 卡初始化没有成

功，幸好那一次突破带动了我整个项目，不然就真的凉了。

使用软件方法完成了 bootloader，路子就会宽很多，有很多东西用软件实现速度会快很多。

10. 参考文献

C/C++ 编译为 mips 汇编代码
https://blog.csdn.net/qq_41595874/article/details/88754760

Vivado 中 MIG 核中 DDR 的读写控制
https://blog.csdn.net/wordwarwordwar/article/details/79539049

Micro SD 卡（TF 卡） spi 模式实现方法
https://blog.csdn.net/ming1006/article/details/7281597

v2.0 版 SD 卡协议中命令 CMD8 的使用详解
https://wenku.baidu.com/view/ce5042052af90242a895e582.html

FPGA 初始化 SD 卡及其仿真
https://blog.csdn.net/zengaliang/article/details/76944664

简单的 ld 链接脚本学习 https://www.jianshu.com/p/42823b3b7c8e

Mips GNU 工具链简介 https://blog.csdn.net/alex_xhl/article/details/8137844

hexdump 命令 https://man.linuxde.net/hexdump

【write a toy cpu】环境搭建
https://aojueliuyun.github.io/2018/04/30/2018.4.30--write-a-toy-cpu--environment/