计算机系统结构课程实验

总结报告

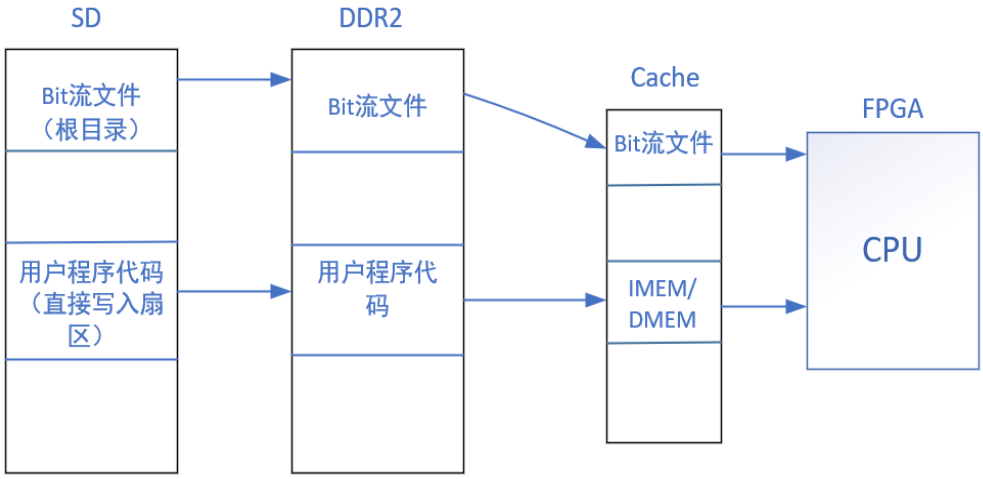
实验题目：三级存储 CPU

一、 实验环境部署与硬件配置说明

Win10　Vivado 2018.2　　8GB SD 卡　Nexys4 开发板

二、 实验描述

对于绝大多数程序而言，所访问的指令和数据在地址上不是均匀的，而是相对聚集的。包含时间局部性和空间局部性。前者指程序将用到的信息很可能就是现在使用的信息。后者指程序将用到的信息很可能与现在正在使用的信息在存储空间上是相邻的。多级存储器之间以块或页面传送数据，其目标是：从 CPU 来看,存储系统速度接近 M1 的速度，而容量和位价格都接近 Mn 的容量和价格。

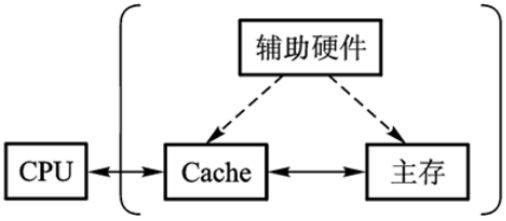


数据流通路：　SD --> DDR2 --> CACHE

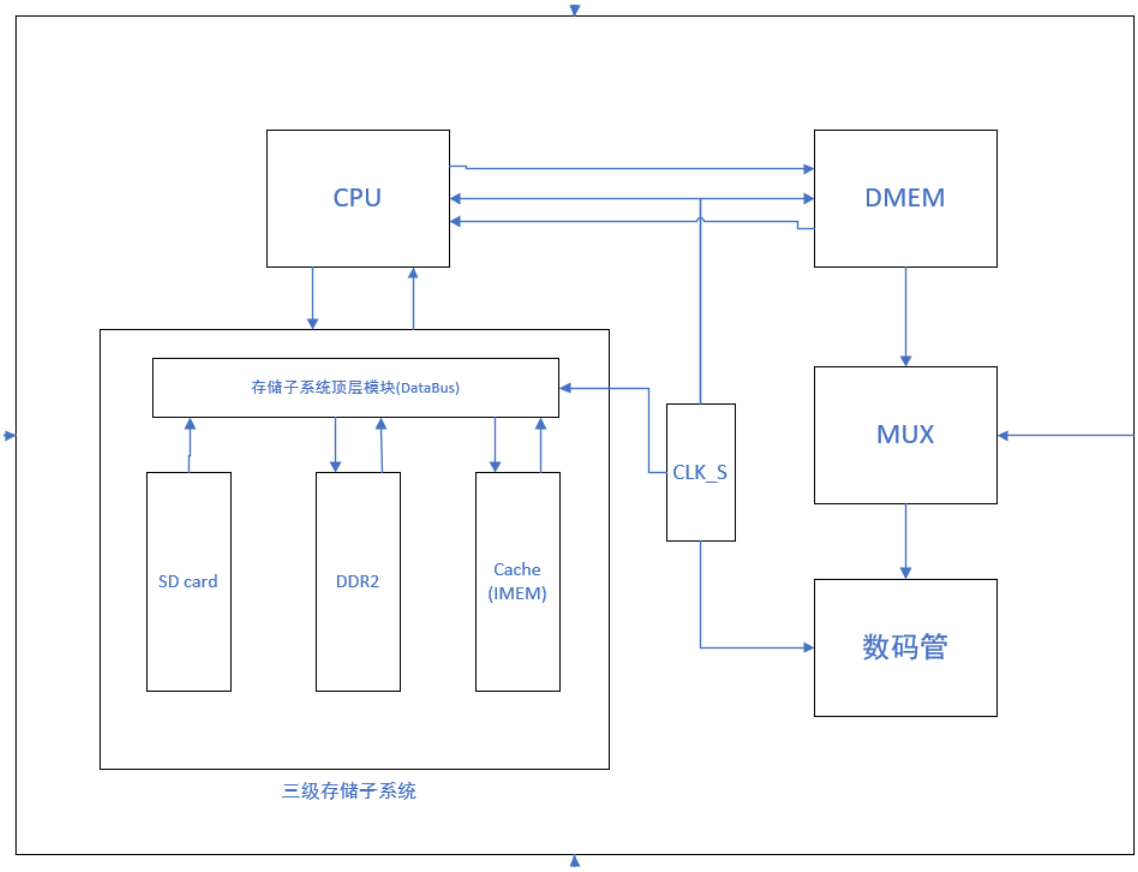
实验要求：
SD 卡中存放流水线 CPU 的二进制流，以及用户程序，N4 板上电自动完成如下的任务：
1） 采用跳线的方式，FPGA 自动从 SD 卡中获取流水线 CPU 的二进制流,并运行该二进制流， 使 FPGA 成为 CPU。
2） CPU 再按照三级存储的方式访问 SDRAM，再由 SDRAM 从 SD 卡中把用户程序的目标代码调入到 SDRAM，再由 CPU 把 SDRAM 中的用户程序目标代码调入到片内 CACHE 加以运行。

三、 整体框架

本实验采用三级存储结构，即 CACHE -> 主存 -> 辅存。借助辅助硬件，使 CACHE 与 主存形成一个有机整体，弥补主存速度不足.由硬件实现,对应用与系统程序员都透明。“Cache－主存”层次：弥补主存速度的不足。“主存－辅存”层次：弥补主存容量的不足。其中，CACHE 使用 IMEM，主存使用 SDRAM DDR2，辅存使用 SD 卡。



“Cachc-主存”层次



根据上图架构，设计代码结构如下：

数据流工作说明

开发板上电后，会自动从 SD 卡中加载比特流文件下板，此时 FPGA 就变成了具有三级存储的 CPU，存放在 SD 卡 0 号扇区的 MIPS 指令被搬运进 DDR 中存储。

三级存储子系统与 CPU 通过 DataBus 模块进行通信，当 CPU 发起一次 Cache 访存时，向存储子系统给出要访问的地址，存储子系统则通知 CPU 其工作状态和取出的指令。在 DataBus 模块中设置 busy 和 done 两个信号，若 DataBus 忙，则 busy 有效，CPU 需要进入阻塞状态等待；读写数据完成时，done 信号有效，CPU 继续运行。

详细的来说，Cache 块的控制器如果发现给定的内存地址不在 Cache 块上时，会发送 busy 信号通知 CPU，使得 CPU 自动进入阻塞的状态，等待 Cache 块的控制器发送 done 信号后才能继续执行。此时 Cache 块的控制器向 DDR 控制器发送读取数据请求，同时将自己陷入等待 DDR 控制器信号的状态。 DDR 进一步向 SD 控制器发送数据请求信号， SD 控制器使能 SD 驱动来读取 SD 卡上指定扇区的数据。等到 SD 卡将指定扇区的数据读取完毕后发送信号通知 DDR 开始读取 SD 卡暂存在 SD_MEM 中的数据。DDR 数据就绪后发送信号通知 Cache

控制器开始读取 DDR 中的数据。Cache 数据就绪后，Cache 块的控制器发送数据就绪信号通知 CPU 进入正常运转状态。

如果在运行中发现 Cache 中的数据块有数据但标识不对应，那么启动替换程序进行数据替换。

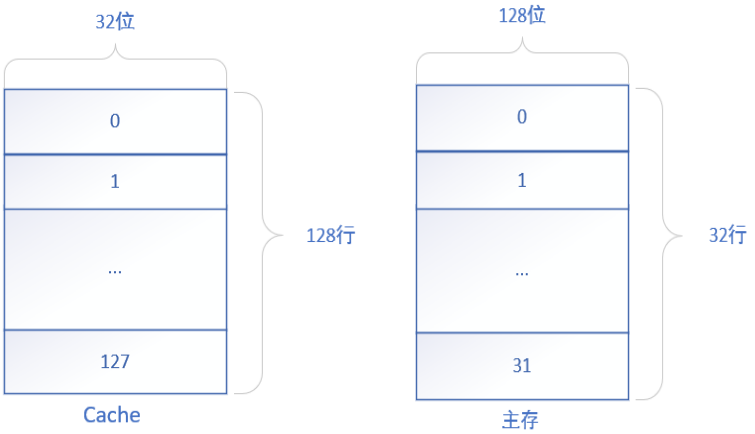
四、 存储系统结构

1. 主存与 Cache 组织结构

(1) Cache 块

1) 每个块为 32*128，每行存一条指令，为 32 位

2) 共四个块，总容量 512 字



(2) 主存块

1) 每个块 128*32，每行存 4 条指令

2) 每块容量大小与 cache 相同

2. 主存与 Cache 映射

采用直接映射，规则如下图，主存四个块为一组进行映射，每个块只能被放置到 Cache 中唯一的一个位置，主存块到 Cache 的对应关系是循环分配的。

地址映射方法：

将主存地址划分为标识位、索引位和块内位移三部分。Cache 中每块大小为 128 字（每个字 32 位），因此块内位移地址位数为 7 位(2^7=128)；Cache 中共有 4 个块，索引引位为 2 位，其余高位作为标志位，实际上也可以理解为组地址。

可以通过主存块地址的索引位选择直接影响 Cache 中的块。可以设置表示存储器来判断 Cache 是否命中。CPU 进行访存时，如果发现标识存储器中对应的单元，其最高位为 0 或者最高位虽然为 1，，但是表示与访存地址的标识不符，则进行替换，将主存 DDR 中的数据块调入 Cache，同时修改标识寄存器，而后即可取得数据。

标识位	索引位	块内位移
31：9	8：7	6：0

	23	22:0
Cache[0]	是否有数据	标识
Cache[1]	是否有数据	标识
Cache[2]	是否有数据	标识
Cache[3]	是否有数据	标识

主存块索引 i , 映射到cache块索引 j, cache共4块

j = i mod 4

为更清楚地描述地址组成，例如：

1） CPU 给主存发地址，总的第 513 条指令，cpu_addr[32:0] = 00000804H =… 0100000000100B，应该在 Cache 的第 1 行，ddr 的第 4 块第 0 行第 1 条指令（第 32 位）。从第 0 块，第 0 行，第 0 条指令开始计数。

2） 右移两位，变为…0100**0000001**B

3） 对于 Cache: 块内位移为[6：0] = 0000001=1，即对应 Cache 的第 1 行。

4） 对于主存，[1:0]=01B，对应 DDR 中行内偏移，为 1，即该行的第一条指令,[6:2]=00000，即为块内的第 0 行。

5） 索引位 [8:7] = 00，对应 Cache 第 0 块。

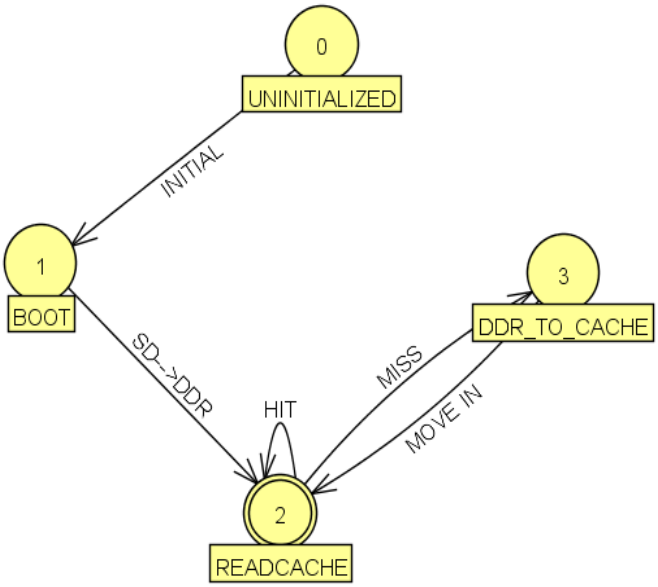
6） 标识位 [31:9] = …01B，为主存的第一组。

3. DataBus

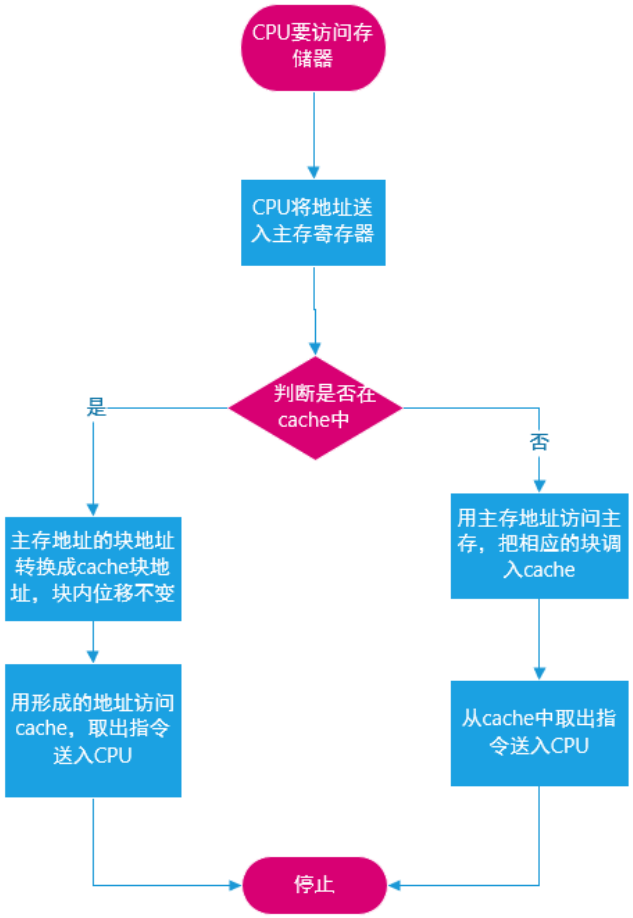
作为控制器控制三级存储子系统，与 CPU 配合同步并给出存储模块的读写信号和数据流向。是一个有限状态自动机，四个状态如下：

名称	含义
UNINITIALIZED	未初始化状态，进行 SD 卡和 DDR 的初始化
BOOT	将 SD 卡数据写入主存 DDR 中，用二重循环将数据块写入 DDR
READCACHE	读 Cache，对比地址和标识寄存器，判断 Cache 是否命中，若命中则将数据送到 CPU，否则转入 DDR_TO_CACHE 状态
DDR_TO_CACHE	Cache 未命中时，将 DDR 的数据块调往 Cache，之后转回 READCACHE 状态

状态机转移图如下：



代码实现流程图：



READCACHE 状态代码：

```
`READCACHE:
begin
    cache_read_write=`READ;
    //read cache
    if(cache_status[cpu_addr[8:7]][22:0]==cpu_addr[31:9]
        &cache_status[cpu_addr[8:7]][23]==1'b1)
    //cache 命中
    begin
        DataBus_busy=`NoBusy;
        Databus_done=`DONE;
        cur_state=`READCACHE;

    end
    else//cache 不命中
    begin
        cur_state=`DDR_TO_CACHE;
        DataBus_busy=`BUSY;
        Databus_done=`UNDONE;
    end
end
```

DDR_TO_CACHE 状态替换实现：

```
if(ddr_done&~is_first_to_imem_count&~ddr_busy)
begin
    if(ddr_to_imem_count==BLOCK_SIZE-1)//写满了一个块
    begin
        loadData_DDR_TO_CACHEDone=`DONE;
    end
    else
    begin
        ddr_to_imem_count=ddr_to_imem_count+1;
    end
end
```

```
assign cache_addr_toIMEM=
            (cur_state== `DDR_TO_CACHE)?
            cpu_addr[8:7]*BLOCK_SIZE+ddr_to_imem_count:cpu_addr[8:0];
//cpu_addr [8:7]-索引 [6:0]-块内位移
assign cache_data_toIMEM=ddr_data_fromDDR;
```

五、 三级存储实现方法

1. SD 卡

使用 SPI 模式进行读写，每次与 SD 卡的传输单位是一个扇区。

手册中的读写状态机：

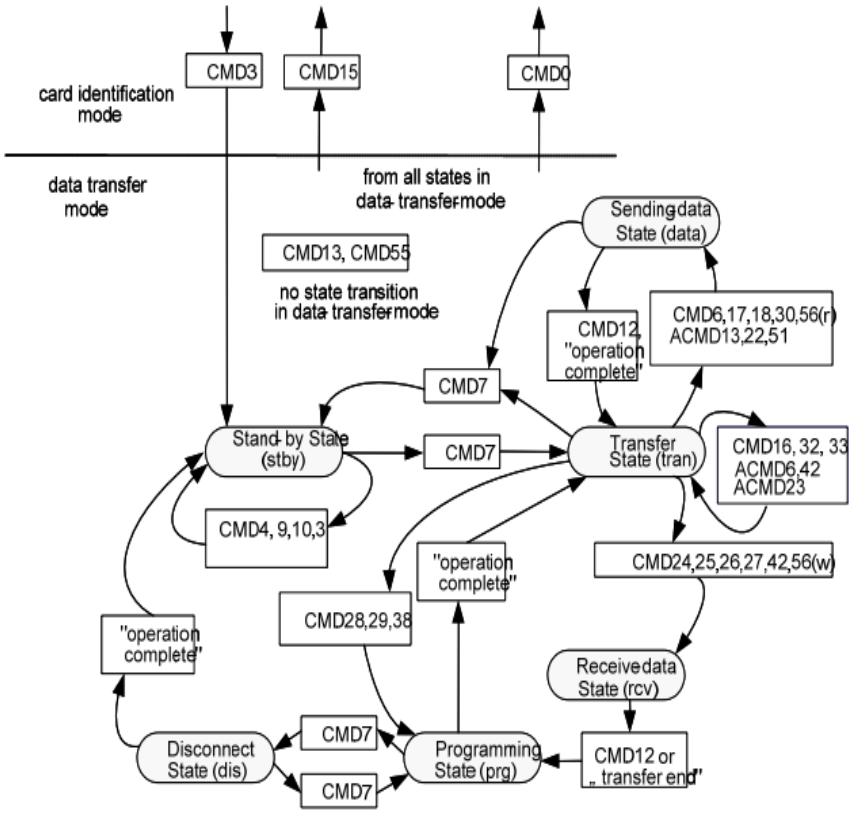


Figure 4-3: SD Memory Card State Diagram (data transfer mode)

SD 卡的初始化，需要先给予至少 74 个 CLK，因为在上电初期，电压的上升过程据 SD 卡组织的计算约合 64 个 CLK 周期才能到达 SD 卡的正常工作电压即 Supply ramp up time，其后的 10 个 CLK 是为了与 SD 卡同步。之后开始 CMD0 的操作再发送 CMD0，CMD0 没有参数，后面发出 3 个字节的 0 即可，再加上校验码 0x95，共六个字节，发送完之后就可以读取 SD 卡的回应码，返回 1 即表明 SD 卡进入空闲状态，接着发送 CMD8, CMD55, CMD41，同样没有参数发送 0 即可，校验码为 0xff，SD 卡即跳出空闲状态，可以接受命令了。

即： CMD0——0x01（SD 卡处于 in-idle-state）

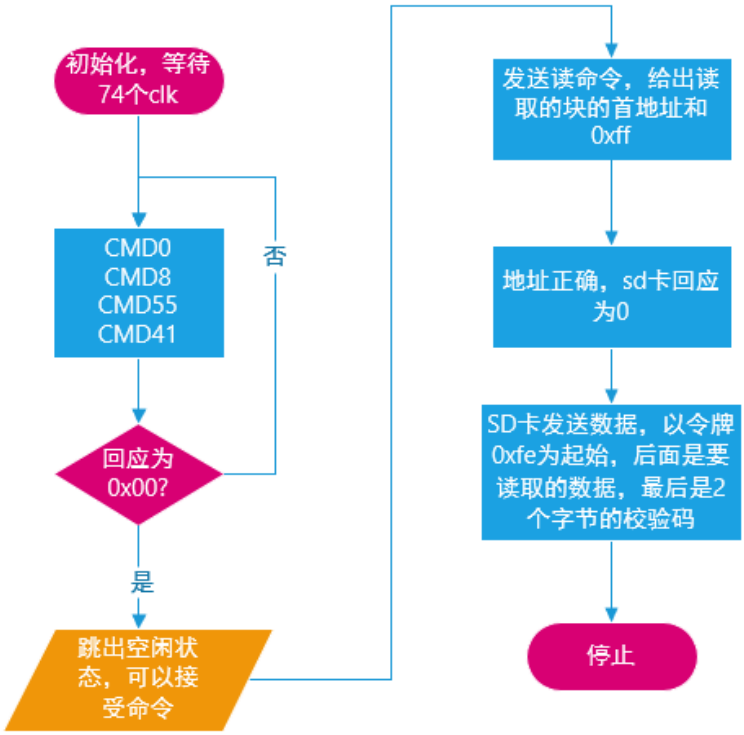
CMD55——0x01（SD 卡处于 in-idle-state）

CMD41——0x00（SD 卡跳出 in-idle-state，完成初始化准备接受下一条指令）

如果最后的回应内容还是 0x01 的话，可以循环发送 CMD55+ACMD41，直到回应的内容 0x00。

SD 卡的读：发送一个读命令，后面的参数是读取的块的首地址和 0xff，SD 会以 R1 回应，如果地址正确，回应应该为 0。接着是 SD 卡发送数据，以令牌 0xfe 为起始，后面是要读取的数据，最后是 2 个字节的校验码。

在自己测试 SD 卡时实现了读写功能，但因为三级存储 CPU 不需要对 SD 卡实现写功能，因此流程图可简化如下：



读取 SD 卡一个扇区的主要代码实现：

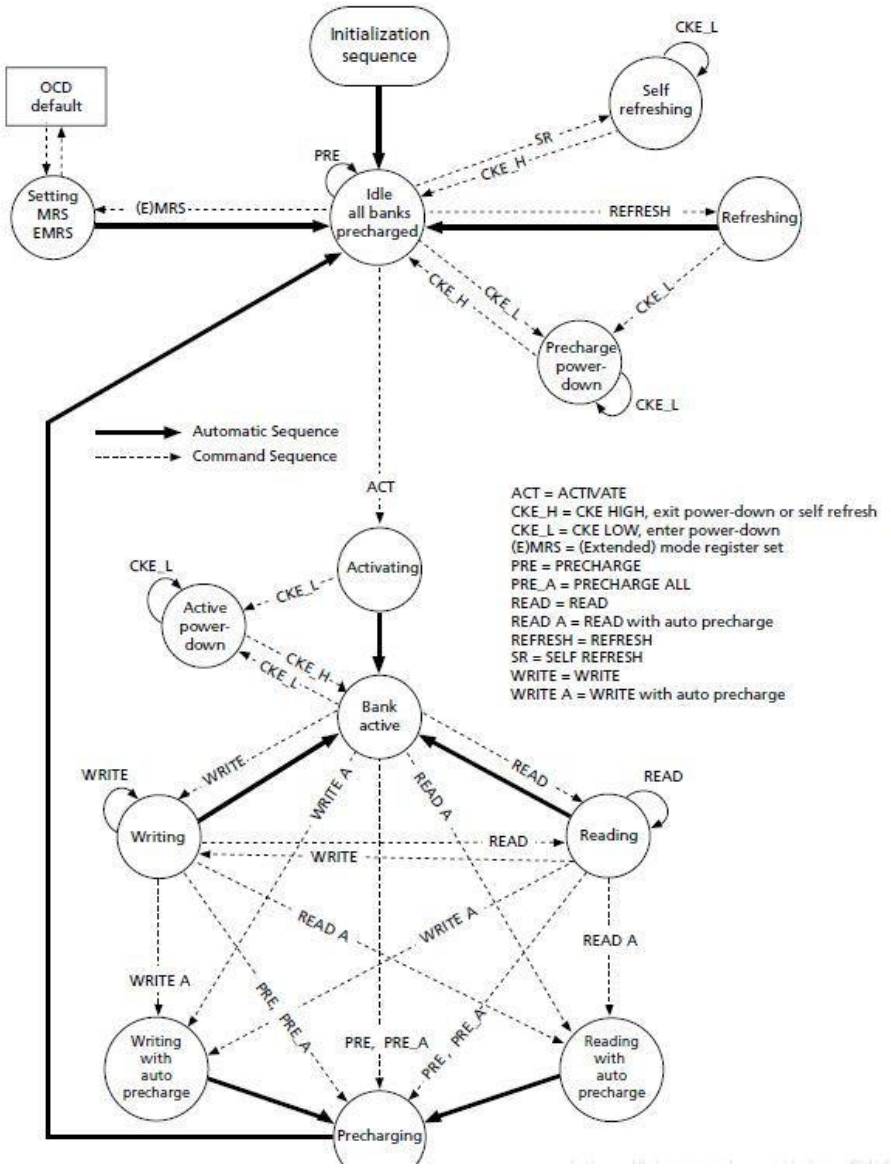
```
S_READ_WAIT:
            begin
                if(spi_wr_ack == 1'b1 && data_recv == 8'hfe)
                begin
                    spi_wr_req <= 1'b0;
                    state <= S_READ;
                    byte_cnt <= 16'd0;
                end
                else
                begin
                    spi_wr_req <= 1'b1;
                    send_data <= 8'hff;
                end
            end
            S_READ:
            begin
                if(spi_wr_ack == 1'b1)
                begin
                    if(byte_cnt == 16'd512)
                    begin
                        state <= S_READ_ACK;
                        spi_wr_req <= 1'b0;
                        byte_cnt <= 16'd0;
                    end
                    else
                    begin
                        byte_cnt <= byte_cnt + 16'd1;
                    end
                end
                else
                begin
                    spi_wr_req <= 1'b1;
                    send_data <= 8'hff;
                end
            end
```

读出的数据格式为：开始令牌 0xfe+正式数据 512B+2B 校验位，根据物理接口，每次读出 8bit，而后可拼接为 32 位的指令。

2. DDR2

DDR 是 N4 板上自带的一个存储资源，这里将其作为主存使用，借助了 Vivado 自带的 MIG 核封装物理信号，这样就可以只需要关注应用信号而无需在意繁琐的物理信号。

手册中 DDR 状态图：



根据手册总结出一些引脚的功能如下，加粗的是重点使用的接口：

接口名称	用途
app_addr[26:0]	数据地址，从高位到低位分别是bank 4位，行地址13位和列地址10位
app_cmd[2:0]	用户指令，3'b001为读，3'b000为写
app_en	指令使能，当app_addr和app_cmd都准备好后，将其拉高来送出指令
app_rdy	指令接收信号，此信号升高表示送入的指令已经被接受。
app_wdf_data [127:0]	写数据，它会先经过app_wdf_mask，将指定部分覆盖为全1后送出
app_wdf_mask [15:0]	写数据mask，16位的mask，每一位对应数据中的8位

接口名称	用途
app_wdf_rdy	写数据准备，此信号升高表示已经准备好接收新数据
app_rd_data [127:0]	读出的数据
app_rd_data_end	读数据末端信号，最后一个读出的信号
app_rd_data_valid	读数据准备，此信号升高表示读数据准备，可以接受数据
app_wdf_wren	写使能，当数据准备好时，将此信号拉高
app_wdf_end	写数据末端信号，数据送完后拉高

DDR 的控制信号如图：

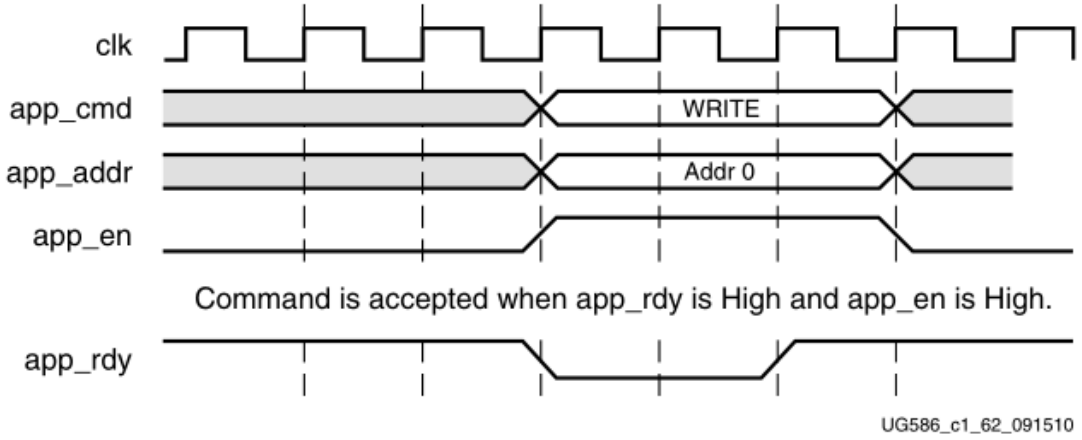


Figure 1-74: **UI Command Timing Diagram with app_rdy Asserted**

可以看出只有当 app_rdy 信号有效时，程序所发出的读写命令才会被控制器接收。

写操作时序如下图：

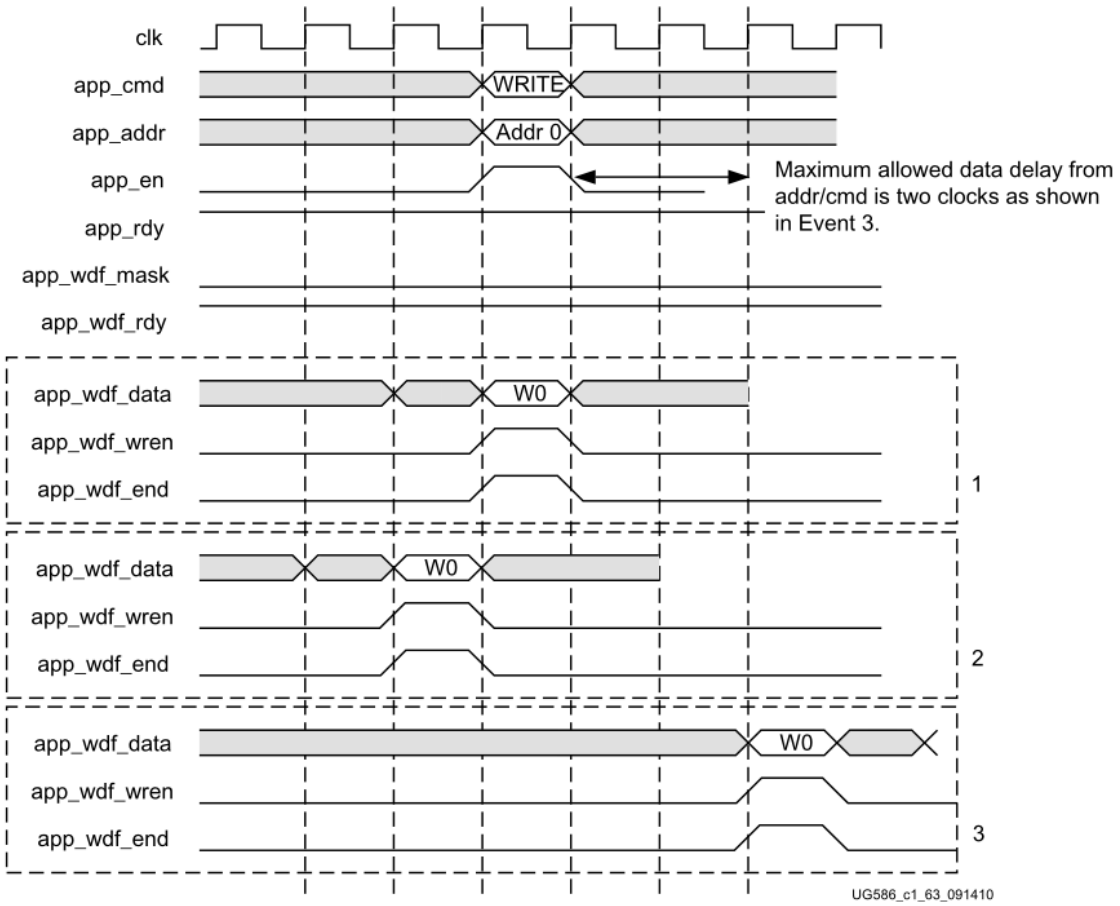


Figure 1-75: **4:1 Mode UI Interface Write Timing Diagram (Memory Burst Type = BL8)**

由图可知，在向 DDR 写数据时，需要提供写命令 app_cmd、地址 app_addr、数据 app_wdf_data 等信号，且写入的数据最多可以比 app_cmd 提前一个时钟周期有效，最迟可以比 app_cmd 晚两个时钟周期有效。在写数据的时候必须检测 app_rdy 和 app_wdf_rdy 信号是否同时有效，否则写入命令无法成功写入到 DDR 控制器的命令 FIFO 中，从而导致写操作失败。

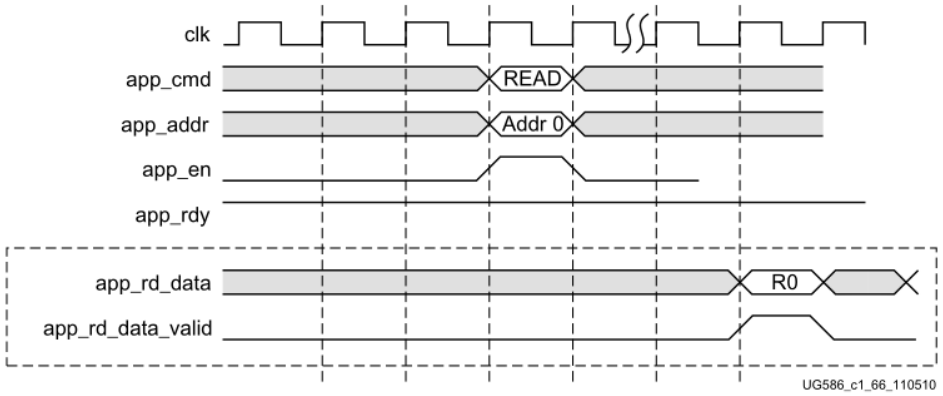


Figure 1-81: **4:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL8)**

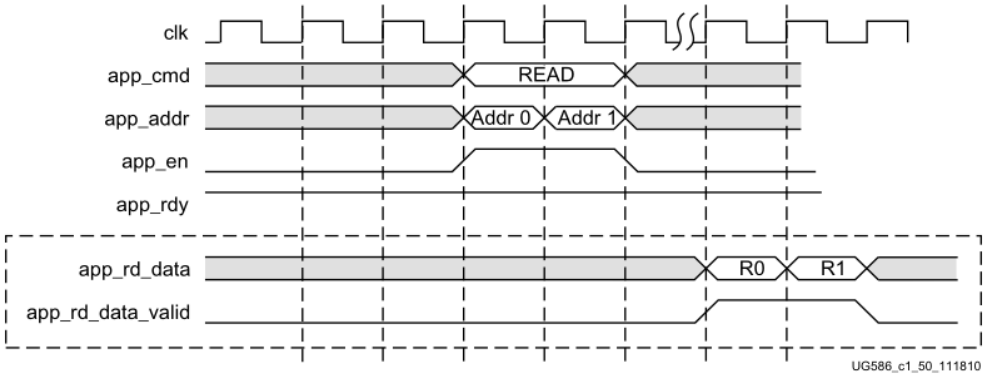
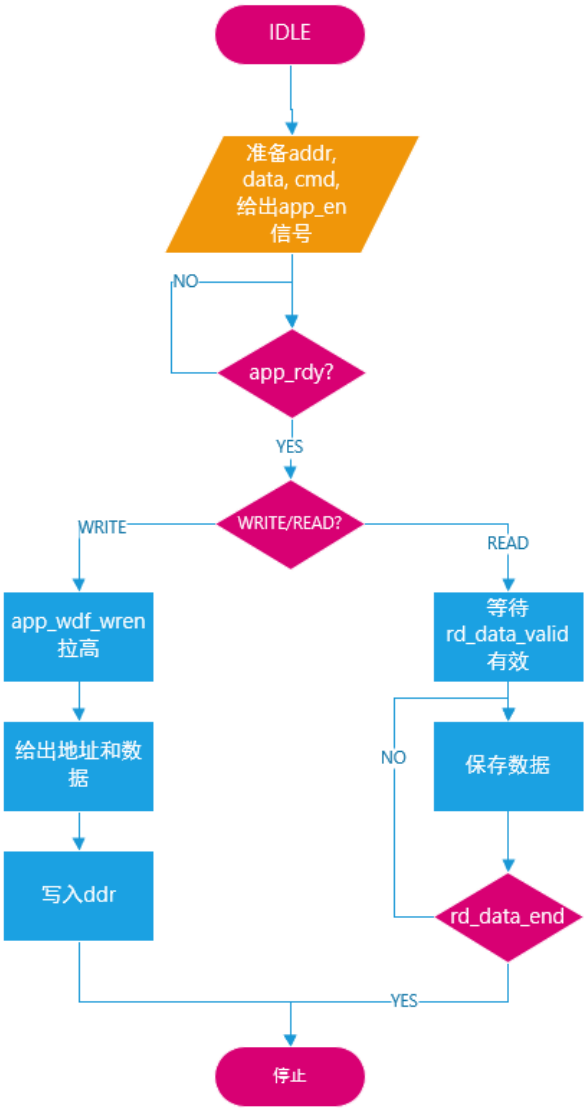


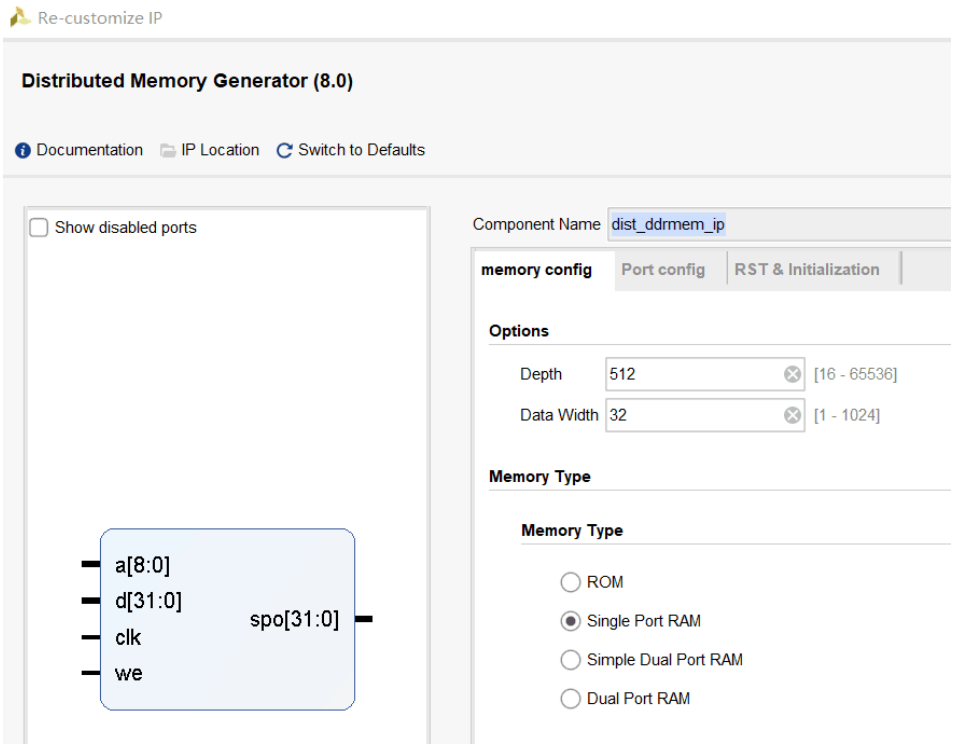
Figure 1-82: **2:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL4 or BL8)**

读时序比写稍微简单一些，最重要的是需要注意 app_rdy 是否有效。
因此可以总结出 DDR 读写流程图：



3. Cache

Cache 共有四块，每块大小 512B，每块中有 128 行，每行为 32 位，可存放一条指令。使用 IP 核实现，可读可写。



六、 实验验算数学模型及算法程序

int a[m],b[m],c[m],d[m];

a[0]=0;

b[0]=1;

a[i]=a[i-1]+i;

b[i]=b[i-1]+3i;

$$c[i]=\begin{cases} a[i], & 0\le i\le 19 \\ a[i]+b[i], & 20\le i\le 39 \\ a[i]*b[i], & 40\le i\le 59 \end{cases}$$

$$d[i]=\begin{cases} b[i], & 0\le i\le 19 \\ a[i]*c[i], & 20\le i\le 39 \\ c[i]*b[i], & 40\le i\le 59 \end{cases}$$

使用 MIPS 汇编程序根据上面的逻辑编写测试代码，将代码转为 16 进制，通过 WinHex 软件写入 SD 卡的第 10000 号扇区。通过板上的开关控制数码管显示数组 C 或 D 的第 n 个元素的值。

汇编指令：

```
1.  main:
```

```
addi    $2,$0,0
addi    $3,$0,1
addi    $4,$0,0
addi    $5,$0,4
addi    $6,$0,1
addi    $7,$0,3
addi    $10,$0,0
addi    $11,$0,240
addi    $12,$0,60
addi    $13,$0,40
addi    $14,$0,20


addi $1, $0, 0x10010000
sw      $2,0($1)    #   stop
sw      $3,240($1)


AB_loop:
addu    $1,$1,$5    #   for data address

add $2,$2,$6    #   A[i]
sw      $2,0($1)    #   stop

add     $3,$3,$7    #   B[i]
sw      $3,240($1)  #   stop

addi    $6,$6,1     #   i
addi    $7,$7,3     #   3i
bne $6, $12, AB_loop

#   reset
nop

addi $1, $0, 0x10010000
andi    $6,$0,0

CD_loop1:

nop

lw      $2,0($1)    #   A
sw      $2,720($1)  #   C

```

```
lw  $3,240($1)  #   B
sw  $3,960($1)  #   D

nop

addi    $1,$1,4     #   for data address
addi    $6,$6,1     #   i

bne $6,$14,CD_loop1
nop

CD_loop2:
nop

lw      $2,0($1)    #   A
lw  $3,240($1)  #   B
add $4,$2,$3    #   C
sw      $4,720($1)

multu   $2,$4
mflo    $7      #   D
sw  $7,960($1)

nop

addi    $1,$1,4 #   for data address
addi    $6,$6,1     #   i

bne $6,$13,CD_loop2

nop

CD_loop3:
nop

lw      $2,0($1)    #   A
lw  $3,240($1)  #   B

    multu   $2,$3
    mflo    $4      #   C

sw      $4,720($1)

multu   $3,$4
```

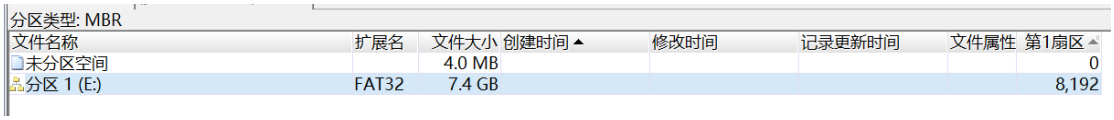
```
90. mflo    $7      #   D
91. sw  $7,960($1)
92.
93. addi    $1,$1,4     #   for data address
94. addi    $6,$6,1     #   i
95.
96. bne $6,$12,CD_loop3
97.
98. endtest:
99. nop
100.
101. endreloop:
102. beq $0,$0,endreloop
```

七、 实验验算程序下板测试过程与实现

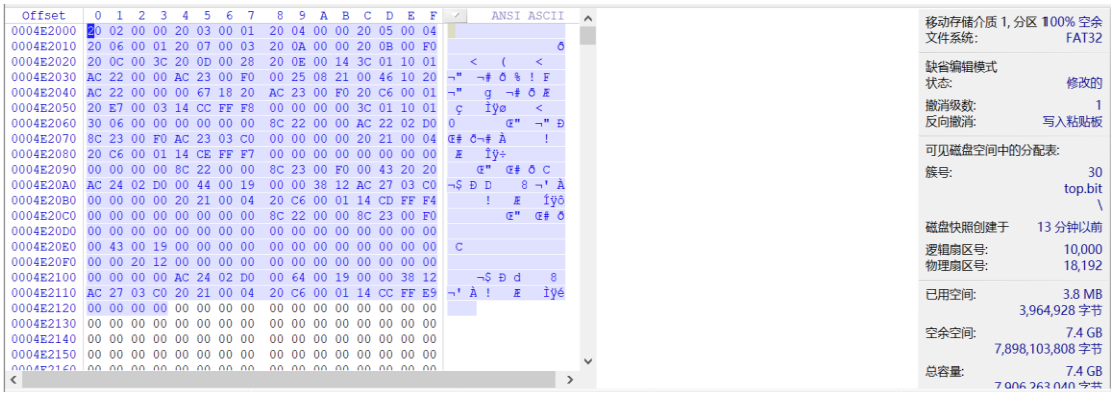
板上最左边开关控制显示数组 D 或数组 C，打开显示 D，关闭显示 C，右侧六个开关控制显示数组元素下标。

将 SD 卡格式化为 FAT32 格式，使用 WinHex 软件打开 SD 卡：

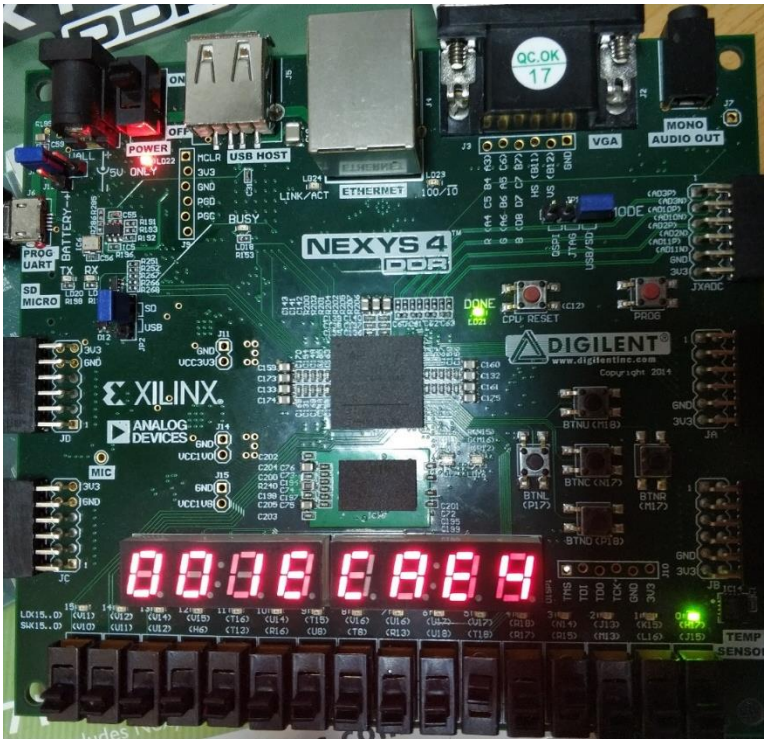
SD 卡结构如下图：有 4MB 的未分区空间，可用空间从第 8192 号扇区开始，因此物理扇区 = 逻辑扇区+8192



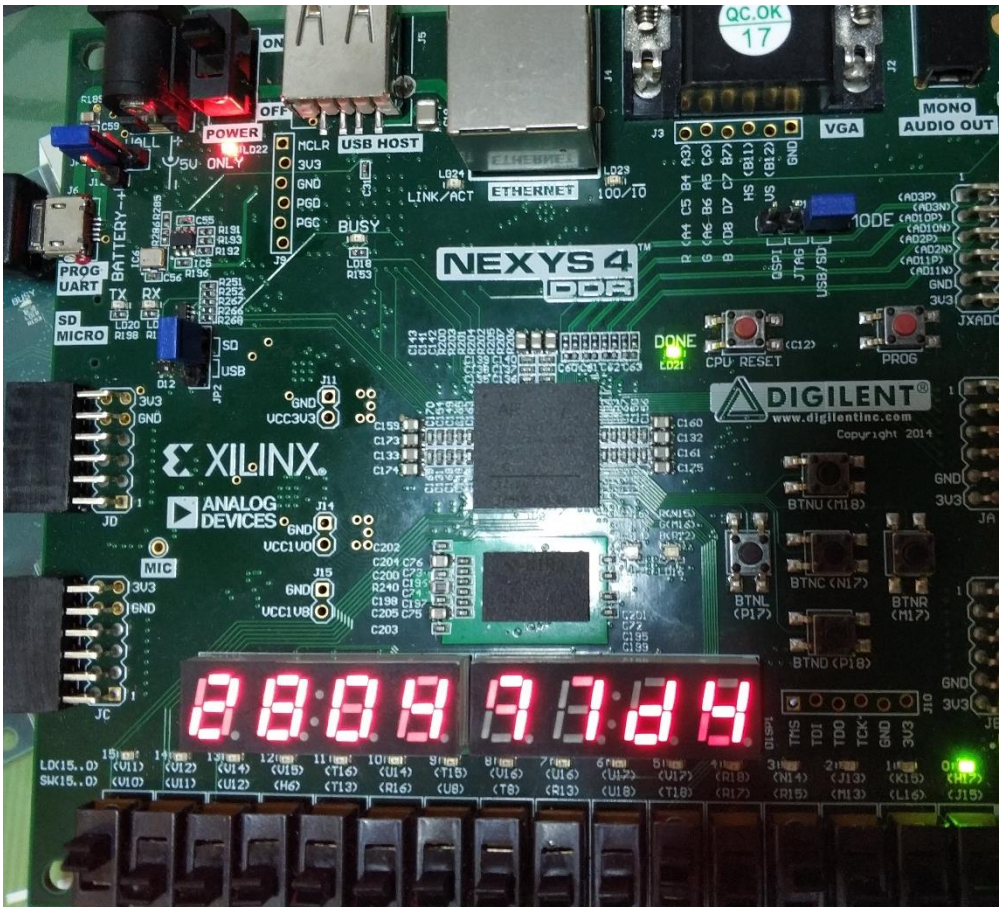
找到写入的物理扇区，在 SD 卡中直接读取该扇区，本程序将测试 MIPS 代码写入逻辑 10000 号扇区，即物理 18192 号扇区（扇区号无特殊意义，只是取整便于找到），并在 Verilog 代码中直接读取该扇区。



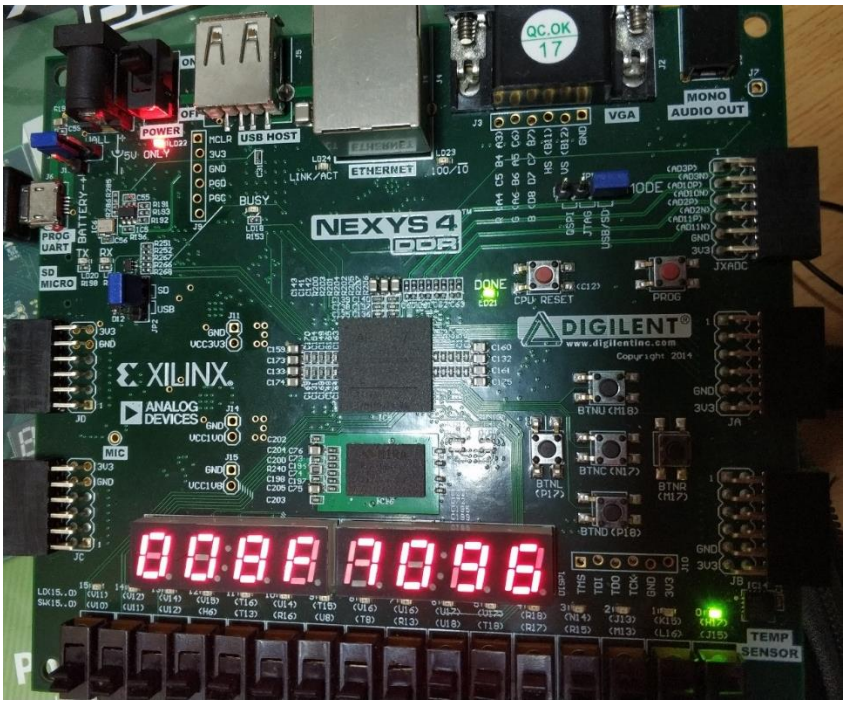
将比特流存入 SD 卡，下板：



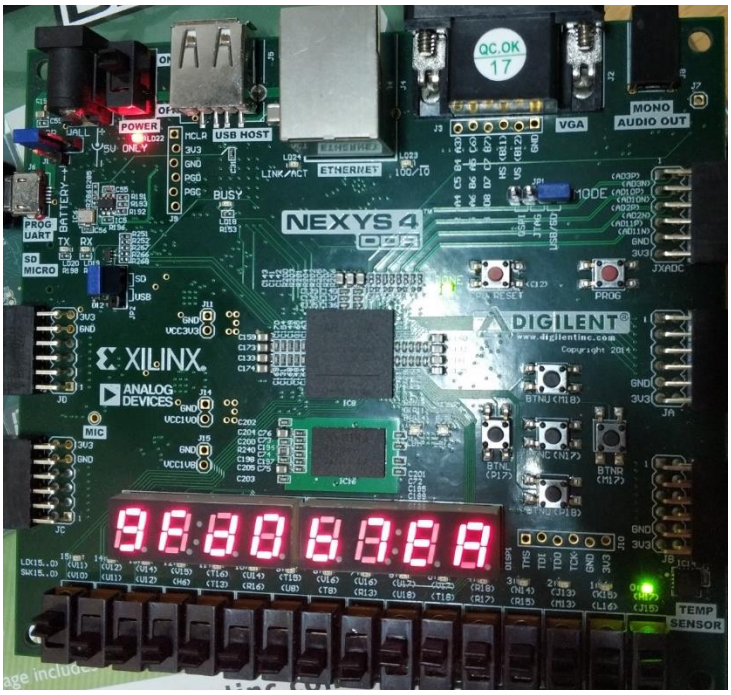
C[40]=001ECAE4H



D[40]=280471D4H



C[59]=008F7096H



D[59]=9FD067EAH

下板结果与实际应得结果一致。

八、 未来优化

1. 使用纯硬件驱动 SD 卡进行 boot

SD卡的初始化工作全部由Verilog实现，没有使用c语言的bootloader，可实现从 SD 卡将 MIPS 程序导入 DDR，但需要用 WinHex 将程序直接写入

扇区。希望以后可以综合 C 语言的启动程序进行交叉编译，实现直接将文件形式的代码放入 SD 卡即可运行。

2. 映射方法较简单

采用直接映射进行对数据块进行替换，在程序较短时劣势体现不明显，程序较长时则效率低，有待改进为组相联。