基于 FPGA 设计的简易游戏引擎
——设计文档&实验报告

计 05 陈可卿 2010011347
计 05 梁缘 2010011355
计 05 孙艺瀚 2010011356

摘要

游戏引擎 CPU GPU 存储模块 输入模块 VGA SD 卡 FPGA VHDL 指令集 时序逻辑 组合逻辑

1 主要设计内容描述

1.1 设计背景

在计算机高度发展的今天，各种五花八门的游戏层出不穷，回味经典感受老式游戏机带来的趣味反而成为了人们的追求。

我们小组意在利用 FPGA 设计出可以根据程序的不同，来运行各种不同小游戏的游戏机。由于其主要针对游戏，所有更像一个游戏引擎。

1.2 设计内容描述

从功能性角度来讲，我们希望将机器可以从 SD 卡读入程序，然后执行。根据程序的不同，通过键盘操作，在屏幕上展现不同的游戏。

从模块化的角度来讲，我们首先需要一个从 SD 卡将程序读入的模块，并且给程序分配好地址，存储模块等等内容。之后需要一个 CPU 来按照程序的指令来运行程序。在运行周期中，通过键盘的输入，根据游戏内部逻辑将不同的内容通过简易 GPU 显示在屏幕上。

1.3 设计内容细分

我们将程序大致分成以下几个模块，并对每个模块做了一些初步的定义。并且对于程序指令集和程序存储方式做了一定的定义。

1.3.1 程序存储方式

程序和数据部分将存储在内存的两个不相交的模块中。CPU 将扫描程序模块，依次按照指令来执行命令（涉及读操作），而对于存储模块只会做读写操作。这两个模块的内存地址相互独立，互不干涉。

注：对于栈，我们考虑在写程序时，不使用任何与栈相关的东西，如果有必要使用数组等方式实现。

1.3.2 SD 卡读入模块（未实现）

此模块用于从 SD 卡读入程序，也就是在 CPU 运行之前，会先将程序复制到程序内存中。

注：程序数据的初始化，由程序自己完成。

1.3.3 CPU 模块

此模块按照我们定义的简易指令集依次执行，每次从程序内存中读取一个指令，并执行这个指令。为了简化设计，我们不仅将指令集简化，并且也只用单周期来完成指令，总体思想为在一个周期的上升沿，完成所需要读入数据的读取，在周期的高电平期间完成之中需要做的所有组合逻辑计算（故周期可能会较长），在下降沿时，将需要写入的数据写入寄存器或者内存中。

而对于指令的读取，一个周期的上升沿完成计数器加 1 的工作，然后通过组合逻辑将程序存储模块中的指令读取出来。

对于跳转语句对于计数器 PC 的修改，需要同步置数，即计数器 PC 根据输入不同决定在上升沿时是加 1 还是置数。

且 CPU 有一个 PC 计数器，用于统计时间，存于一个寄存器。

1.3.4 指令集

(Rn 均为寄存器)

```
ADD R1 R2 :R1+=R2
SUB R1 R2 :R1-=R2

EQL R1 R2 :R1=R1==R2? 1:0
GRT R1 R2 :R1=R1>R2? 1:0
SML R1 R2 :R1=R1<R2? 1:0
```

(逻辑判断直接采用整型最后一位)

```
AND R1 R2 :R1&=R2
OR  R1 R2 :R1|=R2
XOR R1 R2 :R1^=R2
NOT R1 : R1=!R1
SHR R1 R2: R1>>=R2
SHL R1 R2: R1<<=R2

JMP R1 R2 :if (R1) goto R2.addr
JMP R1   :goto R1.addr

READ R1 R2: R1=R2.addr
CPY R1 R2: R1=R2
WRT R1 R2:R2.addr=R1
```

RH R1 VAL:把 Val 从高位开始写入 R1

RL R1 VAL:把 Val 从低位开始写入 R1

GPU R1 R2:把 R1 写入 R2.addr 对应的显存

1.3.5 GPU 模块

此模块需要一个显存，显存需要可以被 CPU 输入，且被 GPU 实时读取，这一读一存的操作需要互不干涉，可以同时进行。

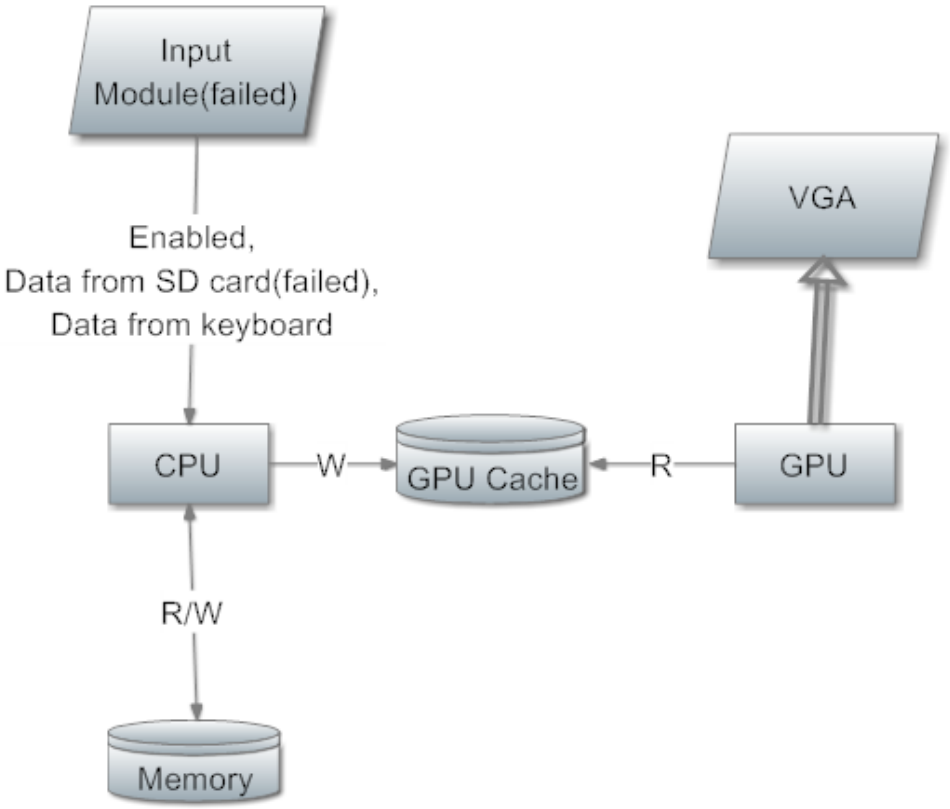
于此同时，GPU 接受来自 VGA 的请求，通过计算以及向显存索取的数据，输入到一个存储了图片信息的 ROM 中，从 ROM 中读出像素点颜色信息，并返还给 VGA，以完成画面显示的操作。

1.3.6 输入模块

对于我们设计的游戏，暂时只考虑四个输入，即上下左右。为了在输入时不需要考虑 CPU 中断问题，所以我们将问题简化为，在 PC 上升沿，将对应的输入（上下左右）写到对应的寄存器中，故这个寄存器仅用于保存四个按键是否被按下（其对应的 4 位表示是否被按下）。

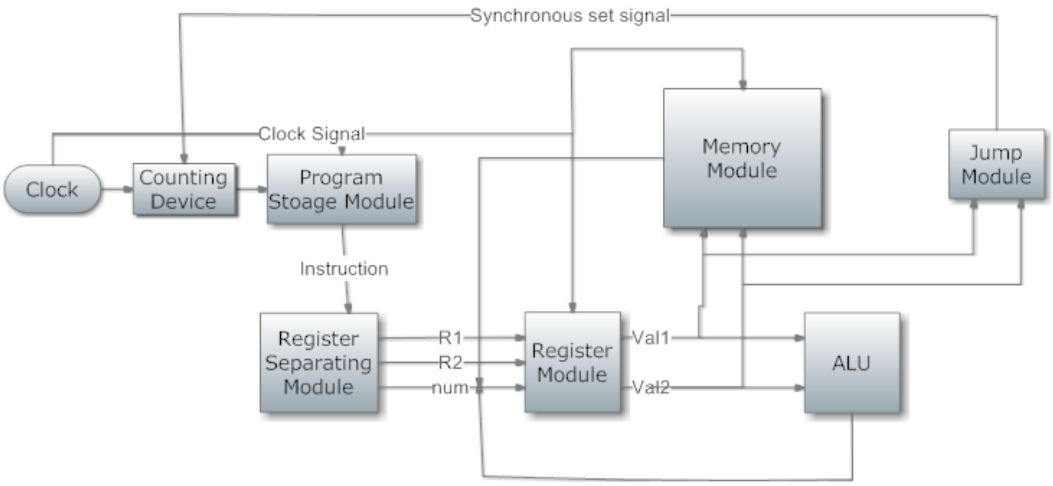
最后的程序中，我们使用键盘的 WSAD 四个键进行上下左右的输入。

2 总体框架图

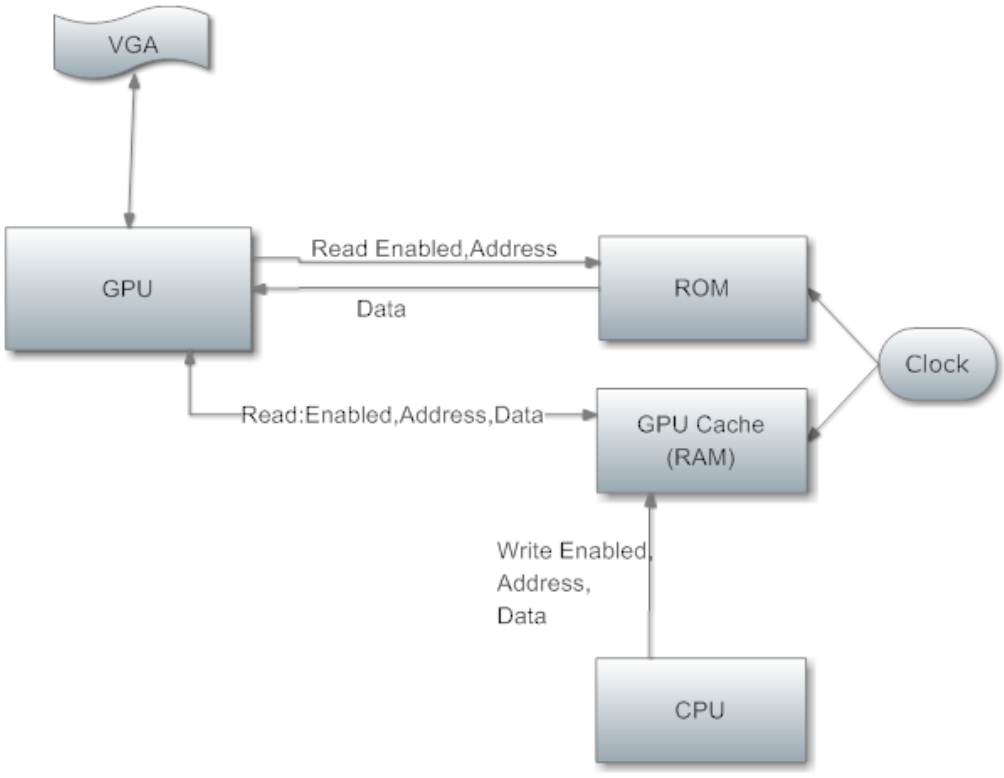

Input
Module(failed)
Enabled,
Data from SD card(failed),
Data from keyboard
VGA
CPU
W
GPU Cache
R
GPU
R/W
Memory

3 内部模块划分

3.1 CPU 模块

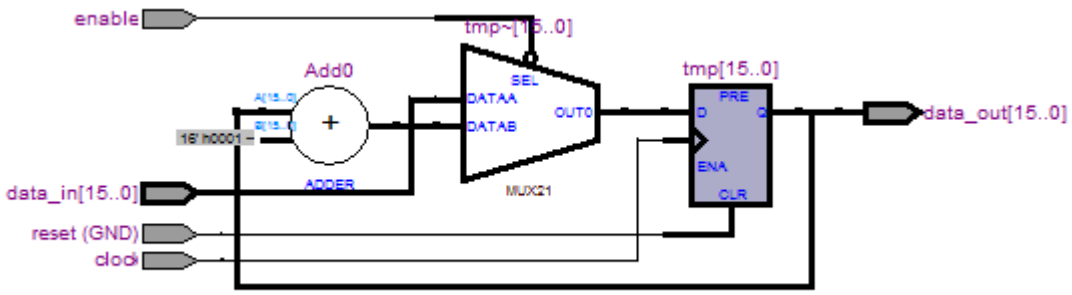


3.2 GPU 模块



4 CPU 模块设计

4.1 计数器模块设计



计数器模块为 CPU 中的重要模块，其作用在于控制 CPU 选择在内存中的模块，并且起到循环转跳的作用。

4.1.1 计数器模块设计

其通过指令集，以及时钟来控制。

在正常情况下在时钟的上升沿计数器加1，但是如果上一个指令为转跳(jump)指令的话，则需要在上升沿时同步置数，其所置数由 data_in 决定，并且输出的指令行号由 data_out 表示。

4.1.2 计数器模块接口

Clock 为时钟
Reset 重置信号
Enable 置数使能
Data_in 置数
Data_out 输出

4.2 寄存器分离模块

此模块由我自己凭空想象出来，其实可以作为寄存器模块的一个子模块。
其功能为分析指令，将指令中的寄存器单独分离出来。

4.2.1 寄存器分离模块接口

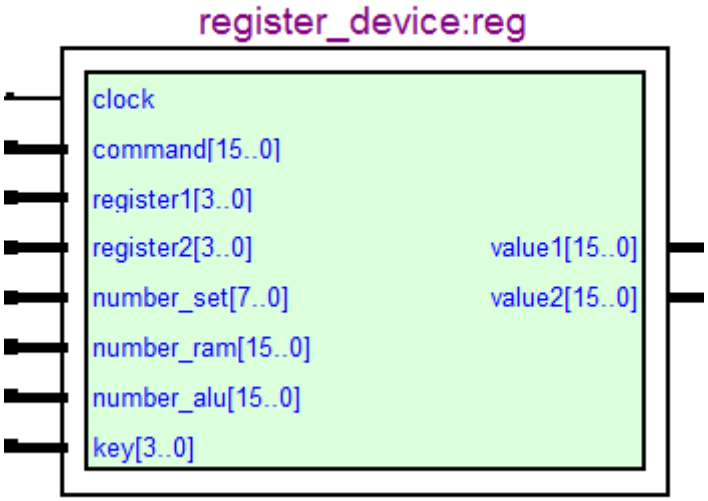
Command 输入指令

register1，register2 输出的两个寄存器，如果只有 1 个，则只输出 register1
number 在指令 RH 和 RL 中，需要对寄存器置数，这个就是所置 8 位数

4.2.2 设计重点

整个模块为一个组合逻辑，无太多难点，就是几个数据选择其构成。

4.3 寄存器模块



这个模块主要功能就在于处理寄存器相关的任何事件。包括从寄存器中读书，将数值写入寄存器，或者从其他任何地方写入寄存器。

其中还包括了，两个特殊寄存器。Reg0 是一个计数器，按照时钟完成加 1，做循环计数用。Reg1 则为一个键盘映射寄存器，将键盘的按键情况映射到这个寄存器中，以方便程序查看。

4.3.1 寄存器模块功能

其需要输入一个或者两个寄存器编号，并且输出其所储存的值。

需要根据指令的不同，将来自内存、alu 或者寄存器分离模块的不同的数据写入寄存器中。这个写入过程默认都在下降沿完成。

故寄存器模块是一个时序相关的模块，但是其内部存储信息与运算无关，运算完全由指令控制。

4.3.2 寄存器模块接口

Command 指令
Clock 时钟

register1, register2 两个寄存器的编号
number_set RH 和 RL 指令中的置数信号
number_ram 来自内存的置数信号
number_alu 来自 ALU 的置数信号
key 来自键盘的映射信号
value1, value2 输出的寄存器值

4.3.3 设计重点难点

对于寄存器的选择，其实是一个较大的数据选择器。

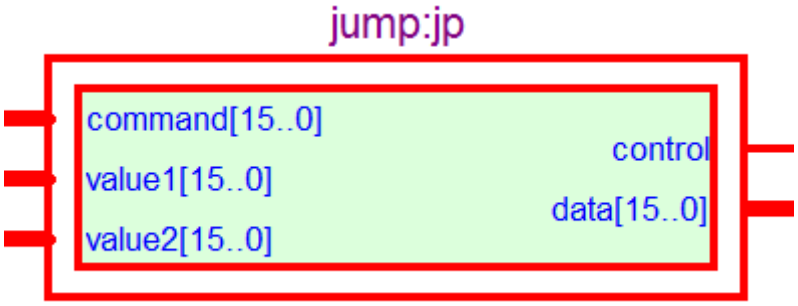
而对于寄存器的置数则相对较为麻烦，首先在设计指令时，我使得所有指令都只能对 register1 中的内容赋值。

之后我们可以通过三态门来完成置数，但是考虑到来自内存和 ALU 的置数信号都是 16 位的，而来自寄存器分离模块的置数信号是 8 位的，这样的线与写起来并不方便。所以我最后使用了同样是数据选择器的方法，通过对 command 指令分析，在上述三个数据中做出选择。并在时钟下降沿完成对寄存器的赋值。

其中值得注意的是，reg0 和 reg1 这两个寄存器不参与赋值。

而如果此次运算中不需要对任何寄存器赋值的话，我默认将其本来的值赋值给自己。这样可以减少设计时的麻烦。

4.4 跳转模块



这是一个看似很简单，但是十分重要的模块。这个模块用于分析代码，然后决定是否让计数器跳转，包括了对于计数器使能的控制，以及对跳转置数的分析。

4.4.1 跳转模块的功能

模块通过对指令的分析判断，决定是否跳转。

如果语句为 jump reg1 则无条件跳转，否则需要判断 reg2 中最末位是否为 1，如果是则跳转。

4.4.2 跳转模块接口

Value1, value2 这两个位来自寄存器模块的输出信号，为 reg1 和 reg2 的两个

值

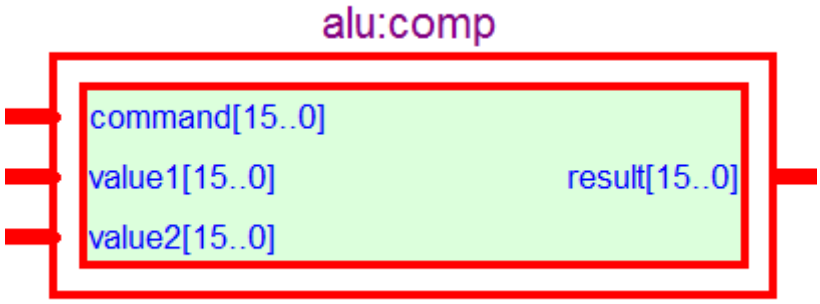
Command 指令
Control 计数器的使能控制
Data 为计数器所置的数

4.4.3 设计重点

同样，这是一个纯组合逻辑，在编写时使用了一些简单的逻辑分析技巧。有如下代码：

```
with c1 select
    control <=
        (value2(0) or c2(0)) when "0010",
        '0' when others;
```

4.5 ALU 模块



这个模块主要用于计算，输入两个量，通过指令控制，完成单目或者双目计算。

4.5.1 ALU 模块接口

Command 指令
Value1 value2 为两个输入的寄存器的值
Result 为输出的值，即计算结果

4.5.2 ALU 模块设计思路及方法

在设计时，我发现有两个指令难以实现。一个是整除，一个是取余。这两个指令难以在一个时钟周期里面完成，且其使用价值极小，所以将其放弃。

其中在所有运算中
加减乘：可以通过 ieee.std_logic_arith.all 中的自带模块完成
与或非以及亦或：可以直接通过逻辑门电路完成
大于小于等于：可以通过数据比较器完成

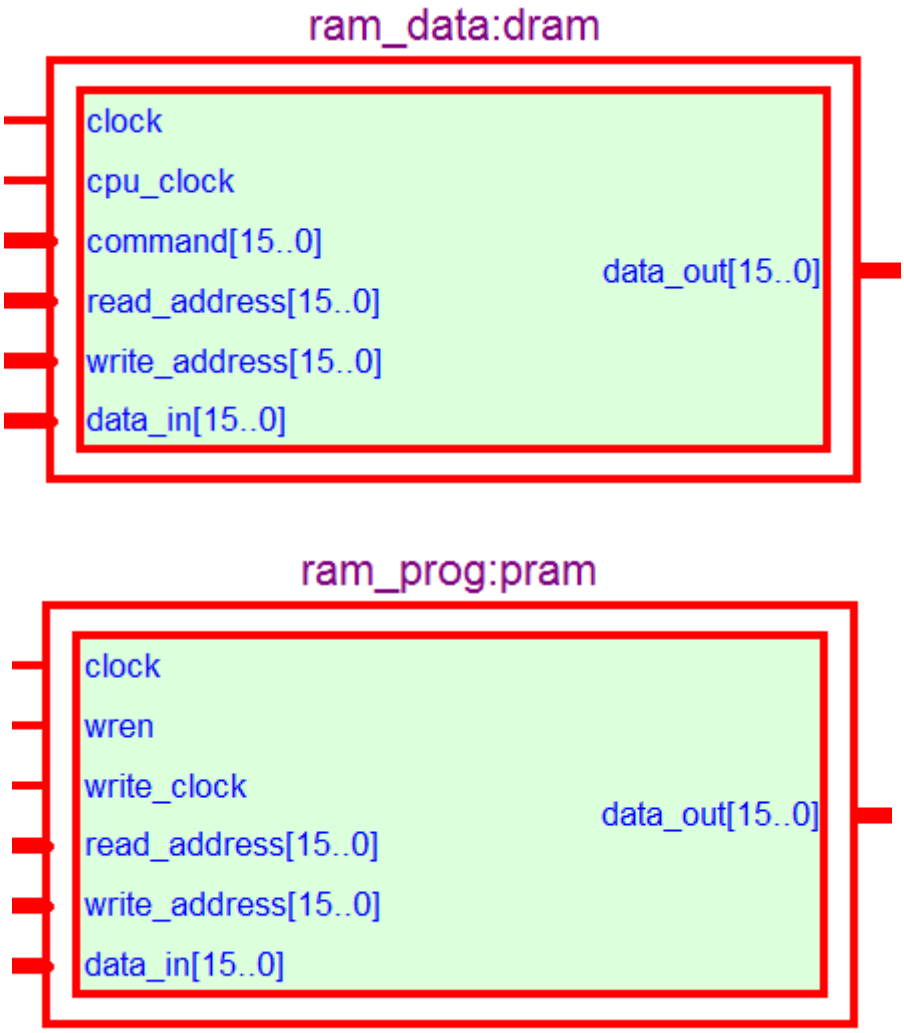
但是位移则不这么方便，最后我们使用了如下语法：

to_stdlogicvector(to_bitvector(value1) srl conv_integer(value2))

to_stdlogicvector(to_bitvector(value1) sll conv_integer(value2))

完成了位移操作。

4.6 内存模块



这个模块其实有两部分组成，一个是程序存储模块，一个是程序运行时内存模块。在一般的计算机中这两个模块是在一起的，但是为了简化设计，我将这两个模块独立开来。从接口看我们发现者两个模块功能基本相同。以下做简要分析。

4.6.1 内存模块接口

Clock 为 ram 内部输入输出锁存所需要的时钟，默认给一个 100M 的时钟

Command wren 这两个其实都是读写使能

Cpu_clock write_clock 输入的控制时钟，下面会做分析

Read_address 读取数据的地址

Write_address 写入数据的地址
Data_in 写入数据内容
Data_out 输出数据的内容

4.6.2 内存模块设计分析

做了简要分析之后，我们觉得我们所编写的程序不大可能会超过 2000 行，而所需要使用的内存，也在 10K 之内就够用了，所以在设计时我们使用了片内 ram。

而片内 ram 的使用则有一定难点。

我们知道，当读写使能变成写的时候，数据就会开始被写入。这就会出现一个问题，如果在使能端变成写之后，数据或者地址还没有到位，则写入的信息就会十分混乱。于是这里就有一个要求：

在写使能变成“写”之前，所有的地址数据，写入数据必须到位。

在写使能从“写”变成“读”之前，所有的数据地址和写入数据必须保持不变。

这其实就是一个十分简单的竞争与冒险，我通过以下代码来实现：

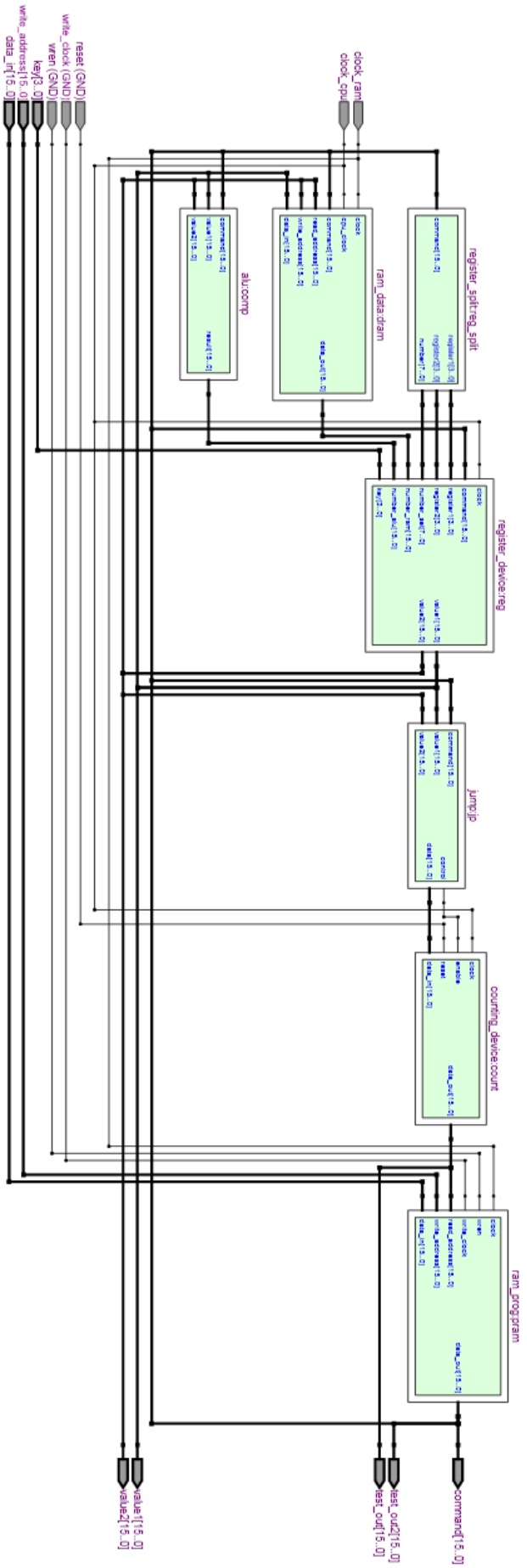
```
address <= write_address(12 downto 0) when (c1 ="00110001")
            else read_address(12 downto 0);
data <= data_in when (c1 ="00110001")
            else "0000000000000000";
    wren <= '1' when (address = write_address(12 downto 0)) and (data = data_in)
and (c1 ="00110001") and (cpu_clock = '0')
            else '0';
```

同样的在显存模块也有同样的问题，下面会做一些简单的分析。

4.7 CPU 模块组合

此模块无太多信息量，主要将上述模块做一个组合。

具体逻辑结构见下图。


reset (GND)
write_clock (GND)
wren (GND)
key[3..0]
write_address[15..0]
data_in[15..0]
clock_ram
clock_cpu
altcomp
ram_data:dram
register_split:reg_split
register_device:reg
jump:jp
counting_device:count
ram_prog:pram
value1[15..0]
value2[15..0]
test_out[15..0]
test_out2[15..0]
command[15..0]

5 VGA 模块

为了简化问题，我们把游戏界面设定为20 × 20的分块，每一个分块有16 × 16个像素点。程序执行过程中，GPU 需要和 ROM 和一个 RAM 进行交互。其中，RAM 用来和 CPU 交互，每一个 RAM 里的地址对应游戏界面上的一个分块，相应地址中存有该分模块需要显示的图案在 ROM 中的地址。对于 ROM，每一个地址存有一个 block 上所有像素点的 RGB 值，工程中设计了 64 种不同的图案，包括 0-9 数字图案、26 个小写字母、常用色块等。

程序中采用了片内的 RAM 和 ROM。

具体程序执行的过程中，对于每一个 block 看做一个整体，从 ROM 里读取预存的图案，输出到屏幕上。

5.1 VGA 模块接口

GPU 的工作流程大致是：对于当前扫描到的屏幕上的像素点，计算出其属于游戏界面中的哪一个分块。由此计算得到相应位置的图案在 RAM 里的地址，同时 CPU 在执行游戏代码的过程中，向 RAM 写入各分块待显示图案在 ROM 里的地址，CPU 写入完成后，关闭 RAM 的写使能端，使 GPU 读取工作可以正常进行。GPU 读取 RAM 中的数据，据此在 ROM 里寻址，得到该分块上每一个像素点的 RGB 值，由此通过 VGA 输出至屏幕。完成对一个点的扫描工作。如此继续扫描下去，以一定频率刷新屏幕即可。

主要接口及交互格式说明如下（略去了时钟和重置接口）：

顶端模块：vga_rom（见最后的大图，不在这里贴出）

输入接口		输出接口	
In_command	RAM 的写操作命令	Hs,vs	行、场同步信号
In_write_addr	RAM 的写入地址	R,g,b	显示 r/g/b 信号
In_data_in	RAM 的写入数据		

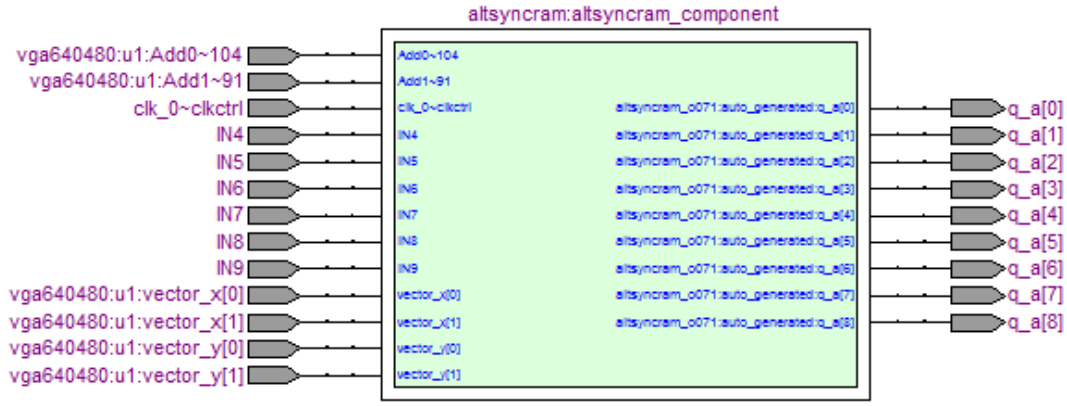
GPU 模块：vga_640480：

输入接口		输出接口	
ram_data	从 RAM 得到的图案在 ROM 中的地址	address	对 RAM 寻址
q	从 ROM 读入的 RGB 值	rom_address	对 ROM 寻址
		hs,vs	行、场同步信号
		r,g,b	显示 r/g/b 信号

ROM 模块：v_rom

输入接口		输出接口	
address	待读取地址	Q	输出数据

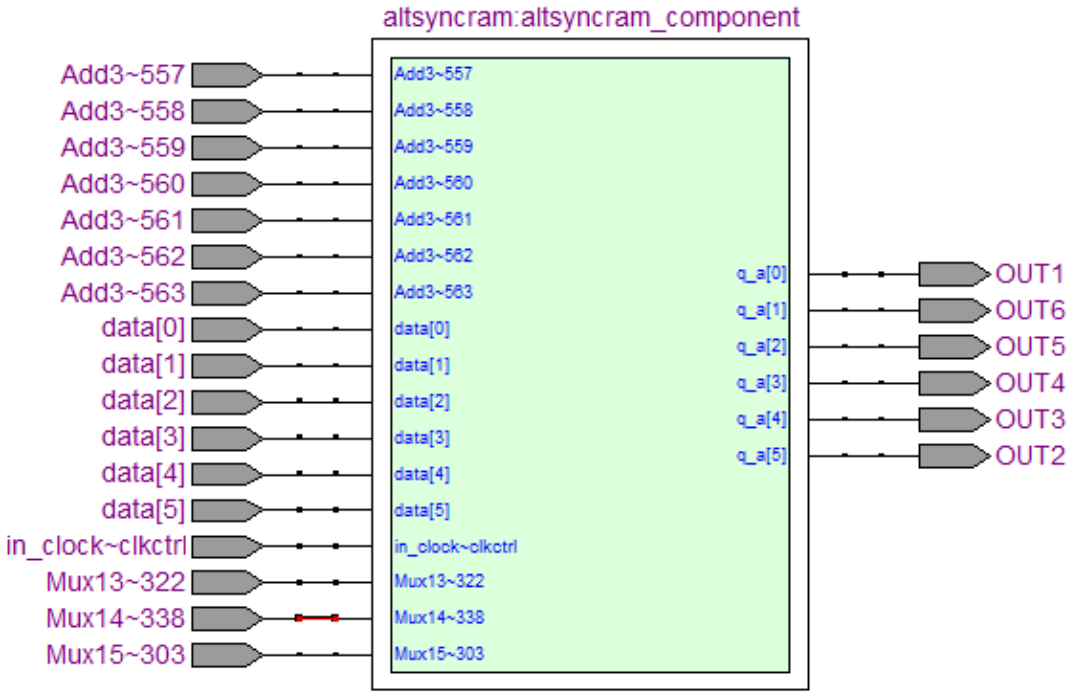
逻辑图如下：



RAM 模块：ram_gpu_d

输入接口		输出接口	
Command	读/写命令	Data_out	读取数据
Read_address	读取地址		
Write_address	写入地址		
Data_in	写入数据		
Data_out	读取数据		

逻辑图如下：



其余关于 vga 所用的 RAM 的设计在后面的显存模块设计中有说明，这里不再赘述。

5.2 遇到的问题和解决方案

GPU 的设计过程中，对于 GPU 与 CPU 的交互方式、显示方式和 RAM 和 ROM 的寻址和组织方式等问题上，曾经多次有疑问，对于如何高效、稳定地交互，通过与组内同学的多次讨论，最终得到了现在的结构和组织方式。由于开始讨论得较为完备，代码的编写中基本上完全按照初始的设计，没有遇到特别大的问题，也没有做什么修改，效率很高，其实最大的难点就在于与 CPU 通过 RAM 的交互，列举主要问题和解决方案如下：

GPU 的主体代码参考老师所给的示例代码，这里提出两点需要注意的问题：

1、 示例代码所给的 vga 时钟是对 100M 时钟分频一次，即 50M 时钟实现的，实际上时钟应该是 25M，需对其再一次分频得到。

2、 在消隐区部分，需要对扫描的 x、y 进行判断，而非行场同步信号。

对于乘法和左移运算，通过网络找到了 ieee.std_logic_arith 中的运算符号。

此外，在 GPU 运行过程中，发现门延迟非常大，有时候会延迟 2-3 个 block，后来将所有的时序逻辑全部写成了组合逻辑，就避免了这个问题。这个过程让我更加深刻地体会了时序逻辑和组合逻辑的区别，让我知道了在设计过程中，要尽量避免时序逻辑的连续大量使用。也充分体会了硬件编程的特点。

还有就是在 GPU 读取 RAM 的过程中，由于 CPU 在写入时，不能同时读出数据，因此屏幕上会有一段的黑色信号。经过计算，该信号持续时间只会在屏幕上形成一道高为一个像素的黑线，不会对人的视觉感官造成比较明显的影响，因此没有着力去解决这个问题。在测试过程中发现确实会偶尔闪过一道黑线，但是不影响整体效果。

另外，由于 VGA 模块设计相对简单，我在最后帮助编写了一部分游戏测试程序，体会了汇编代码的编写和运行过程，在这一门课里同时初步体验了汇编、编译原理、计算机组成原理等各个方面，对整个计算机有了全新的认识，有非常大的收获。

6 显存模块

这是一个需要 CPU 模块和 GPU 模块做结合的模块。他需要有不停读出数据的功能，以及写入数据的功能。

由于 VGA 需要不停的读取数据，而 CPU 需要在特定的时候写入数据到显存中。这时就会有问题产生，显存需要能够同时写入数据以及读取数据，这就和 ram 有很大的区别。

需要实现这样的功能，需要有特殊的硬件模块，或者使用大量寄存器，以及超大规模的数据选择器来实现。

不过在和老师讨论之后，老师告知做短暂的数据写入，并不会太影响输出时的显示效果。所以我们决定使用 ram 来实现显存。也就是说在 CPU 写入的时候，VGA 将无法读取（读取到乱码），这一点在前面的 VGA 模块中已经写到，不再赘述。

5.1 显存模块结构

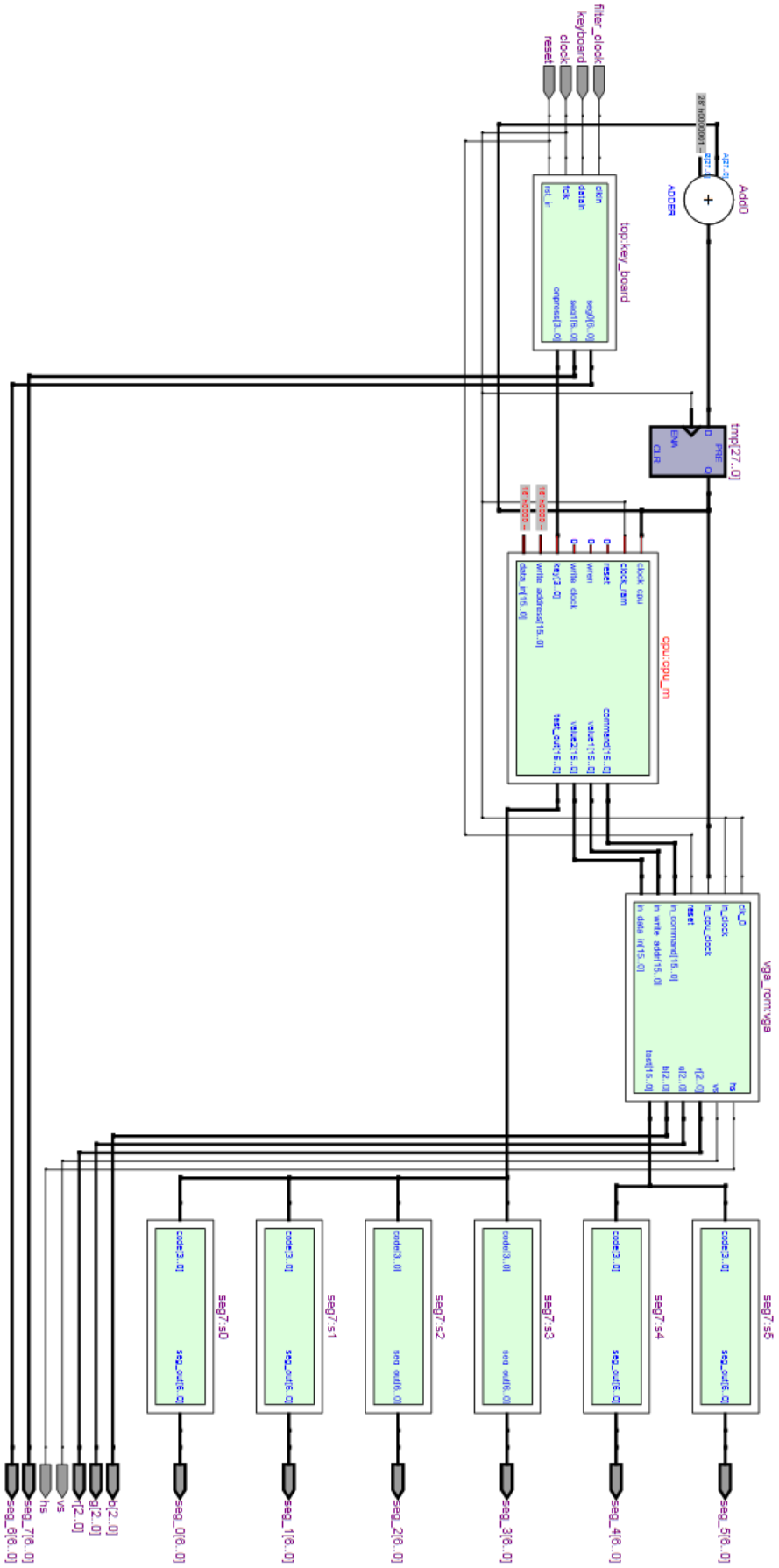
接口同内存模块，其实无太多可以表述的，这里就不重复说明了。

5.2 显存模块难点

VGA 的读取速度其实是非常快的，有 25MHz 的速度。但是 CPU 的速度却没有这么快，只有 1MHz 左右。读取速度的不同就成了一个问题。

但在反复思考之后，我使用了如同上述 ram 设计的方法（其实首先在显存实现时想出，之后再在 ram 中使用）。通过简单的竞争与冒险完成的这个模块的设计。

VGA 及显存模块总体结构图如下：


filter_clock
keyboard
clock
reset
Add0
ADDER
top:key_board
tmp[27..0]
cpu:cpu_m
vga_ram:vga
seg7:s0
seg7:s1
seg7:s2
seg7:s3
seg7:s4
seg7:s5
seg_0[6..0]
seg_1[6..0]
seg_2[6..0]
seg_3[6..0]
seg_4[6..0]
seg_5[6..0]
seg_7[6..0]
hs
vs
r[2..0]
g[2..0]
b[2..0]

6 输入模块设计

该部分包括 SD 卡读入模块，SD 卡模块经过长时间尝试有了一些成果，但最终未能成功。

6.1 SD 卡读入模块

在 SD 模式与 SPI 模式之间斟酌再三选用了 SPI 模式。相比于 SD 模式它牺牲了部分性能，但是代码量（看上去？）要小上很多。

SD 卡在上电后默认进入 SD 模式。如果 CS 标志在接受复位指令(CMD0)期间为低，它将进入 SPI 模式并且处于空闲状态。如果 SD 卡识别到需要保持 SD 模式，它不会对指令作出任何反应并且保持在 SD 模式中。如果需要 SPI 模式，SD 卡将转到 SPI 模式并且进行 SPI 模式响应。而由 SPI 模式回到 SD 模式则必须重新上电。在 SPI 模式下，SD 卡遵守部分协议系统。支持 SPI 模式的 SD 卡指令始终有效。对于 SD 卡读入的过程，SD 卡协议每次从卡内通过 CMD17 或者 CMD18 读取一个或多个块(Block)，每个块的长度由 CMD16 预先指定（我这里取的默认值 512byte）。每个数据块后面跟随着一个由 the standard CCITT polynomial($x^{16}+x^{12}+x^5+1$)确定的 16bit 的 CRC。具体流程图如下（图片来自 SD Specifications, Copyright 2001-2006 SD Group (MEI, SanDisk, Toshiba) and SD Card Association）：

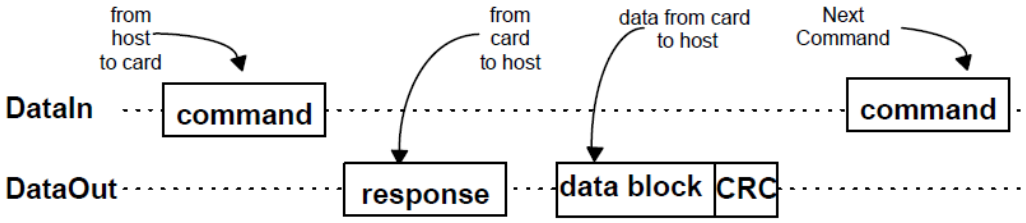


Figure 7-2: Single Block Read operation

而串行发送指令的格式如下表所示：

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

Table 7-1: Command Format

例如发送 ACMD41(一个同步信号)时，发送的命令序列就是 x"690000000001"，其中的”69”为(x”01”)&(x”101001”)。再例如读取单个 block(CMD17)时，拼接的命令序列就是 x"51" & address & x"FF"。

几条比较重要的指令：

CMD0(GO_IDLE_STATE)：重置。

CMD55(APP_CMD)：声明下一条指令是一个 ACMD 拓展而非通用的 CMD。

ACMD41：一个同步信号。
CMD16(SET_BLOCKLEN)：设置单个 block 的长度。
CMD17(READ_SINGLE_BLOCK)：从指定地址处读取单个 block。
CMD18(READ_SINGLE_BLOCK)：从指定地址处连续 block 直至被中断。
CMD12(STOP_TRANSMISSION)：中断由 CMD18 所启动的连续读取过程。

SPI 串行通信整体实现使用一个状态机实现。其中的状态框架我没有自己设计，而是从 StackOverflow 上扒的一个状态框架，随后自己在框架下补的实现。最终实现效果并不理想——成功实现了从某一位置起连续读入数据，但是“某一位置”究竟是哪里我也不知道，调试几天未果，fail 掉。

6.2 键盘输入模块

展示前两天开始调键盘，在示例代码框架的基础上稍作修改——由于端码的后缀即为通码，干扰了对按键状态的侦测，使用一个小号状态机稍作过滤即可。由于已经有了写 SD 卡那个较大状态机的经验，这一小部分写得轻松加愉快。

7 程序设计语言

这个在整个 FPGA 实验中其实是“微不足道”的东西，但是考虑到设计的完整性，这里略提一二。

首先我们更具指令集，可以将简单的汇编代码直接通过简单的分析之后编写成机器语言。

然后我通过汇编语言，重新编写了一个较为高级的代码，其完成了一些变量定义，数组的内存读写，字符集赋值等等的功能，以及 label 可以直接转跳。

具体程序编写并不是这次实验的重点，这里就不多说了。附件中有我所编写的源码，其中 ncp.cpp 为汇编到机器码的转化，scp.cpp 为高级语言到汇编的转化。

为了完成展示我还编写了一个大概 700 行的贪吃蛇程序，详见附件 snake_0.txt。

8 实验收获与总结

这次实验中，我们小组的三个人通过合作，完成了一个小型计算机的设计。在这一过程中，我们更深刻地体会到了硬件编程的原理，也初步体验到了计算机的内部组成，在硬件层面上理解了计算机的运行流程。实验中结合和实践了数设理论课上的很多知识，也亲手编写了 VHDL 代码，收获很大。

在整个实验的创意提出和实现过程中，由于难度较大，我们多次与老师进行过讨论，老师就选题和具体的实现给出了很多建议和帮助，在这里感谢老师的耐心指导。在具体实验过程中，我们也得到过很多其他组的帮助，同样表示感谢。