

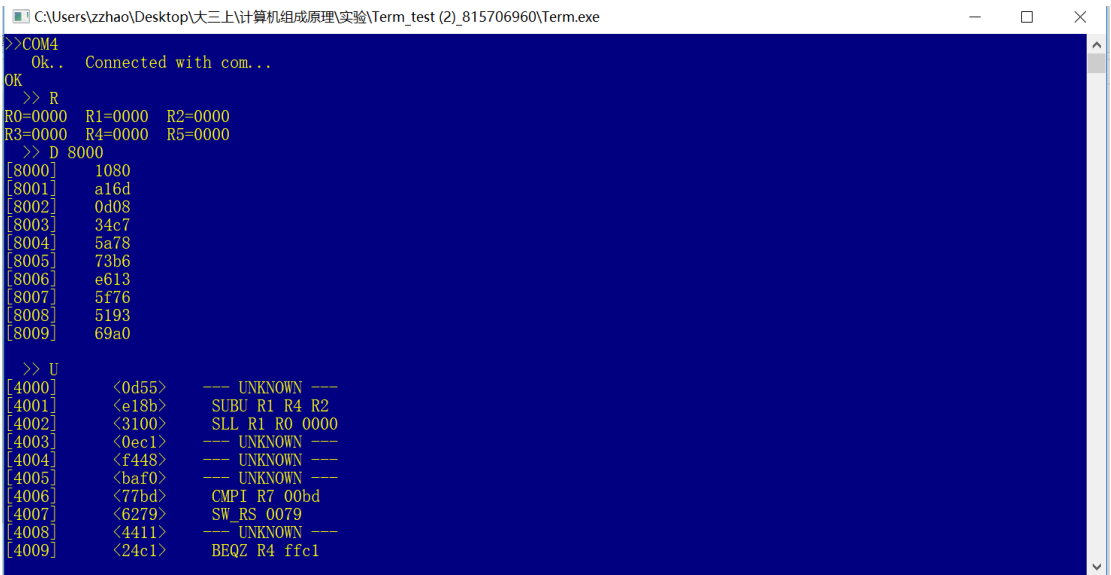
# CPU 实验报告

计 64 徐呈寅  
计 65 章子豪  
计科 60 林一衡

## 简介

我们在 THINPAD 板上基于 VHDL 语言实现了 16 位的流水线 CPU，这个 CPU 可以完整运行监控程序，并可使用监控程序运行写入的汇编代码。

以下为相应截图



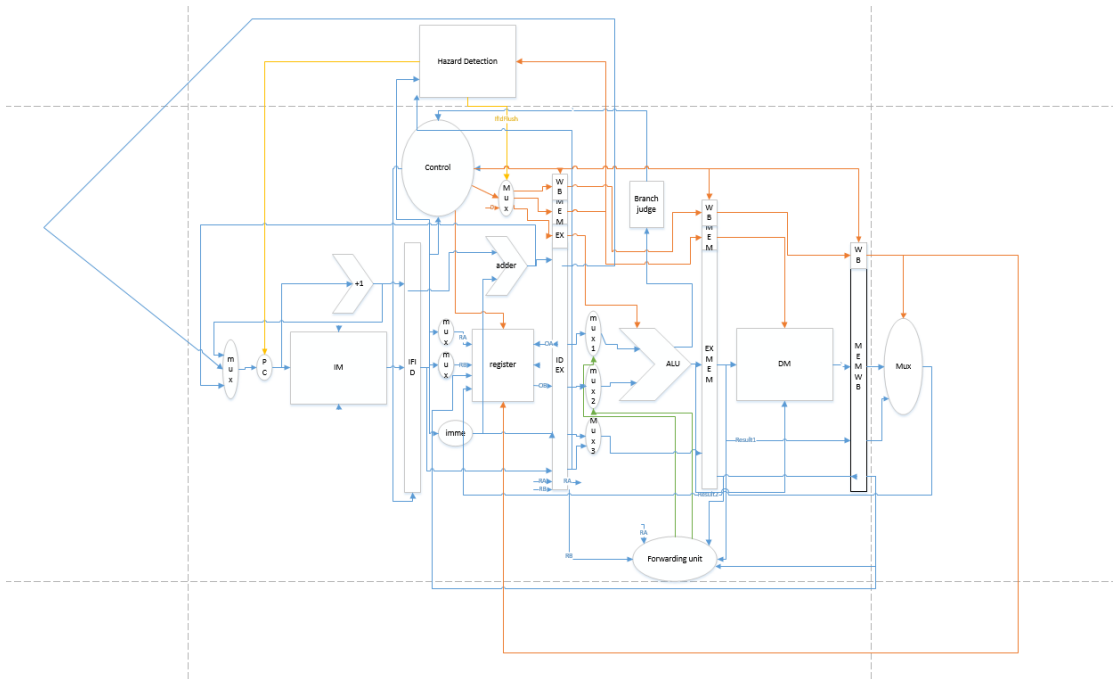
```
>>COM4
Ok.. Connected with com...
OK
>> R
R0=0000 R1=0000 R2=0000
R3=0000 R4=0000 R5=0000
>> D 8000
[8000] 1080
[8001] a16d
[8002] 0d08
[8003] 34c7
[8004] 5a78
[8005] 73b6
[8006] e613
[8007] 5f76
[8008] 5193
[8009] 69a0
>> U
[4000] <0d55> --- UNKNOWN ---
[4001] <e18b> SUBU R1 R4 R2
[4002] <3100> SLL R1 R0 0000
[4003] <0ec1> --- UNKNOWN ---
[4004] <f448> --- UNKNOWN ---
[4005] <baf0> --- UNKNOWN ---
[4006] <77bd> CMPI R7 00bd
[4007] <6279> SW_RS 0079
[4008] <4411> --- UNKNOWN ---
[4009] <24c1> BEQZ R4 ffc1
```

全部测试版代码及数据通路图放在 beta1 文件夹下，其它没有来得及合并入 cpu 的附加功能，包括 VGA, Flash 自启动，贪食蛇游戏汇编程序，键盘，放在“其它功能”文件夹下。

## 实验结果

性能标定：1.5 亿条—18.02s  
运算数据冲突的效率测试：2.25 亿条—24.87s  
控制指令冲突测试：1 亿条—13.57s  
访存数据冲突性能测试：1.5 亿条—72.34s  
读写指令存储器测试：0.75 亿条--20.30s

# 架构介绍



数据通路图

我们数据通路图的设计参考了计 42 徐东亿和陈禹东学长的数据通路图以及陈康老师第 13 讲“控制冲突与异常处理”第 42 页的数据通路图。但由于存储模块的时序不同，我们在他们的基础上做了较大改动。详细说明见各个模块功能的解释。

## 1 IF(取指)阶段

取指阶段包括 PC, PAdder, PCMUX, IM 部分，最后进入 IFID 寄存器。其中 IM 部分放在 MemoryUnit 中统一介绍。

### 1.1 PC

PC 模块（时序，上升沿触发）	
输入	PCMUX 选择出的 PCIn
输出	送往 IM 进行取指的地址 PCOut
控制信号	若 PCKeep 为 1，则在下一个上升沿保持 PCOut 不变

### 1.2 PAdder(用于给 PC 加 1)

PCAdder 模块（组合）	
输入	PC 输出的 PCOut
输出	送往 PCMUX 的 PCOut + 1
控制信号	无

### 1.3 PCMUX（用于选择下一个 PC）

PCMUX 模块（组合）	
输入	PC_AddOne : PCAdder 得到的结果 ID_In : 在 ID 阶段的 Adder 得到的结果 EX_In : 在 IDEX 寄存器上暂存的 Adder 结果

	OA_In : registers 模块取出的结果 OA
输出	送往 PC 的 PCIn
控制信号	控制 4 选 1 即可

#### 1.4 IFIDRegister

IFIDRegister 模块（时序，上升沿触发）	
输入	IM_PC : 指令对应 PC IM_Command : 指令 PCAdder_PC : PCAdder 得到的 PC
输出	IM_Command_Out : 指令 IM_PC_Out : 指令对应 PC PCAdder_PC_Out : PCAdder 得到的 PC
控制信号	KeepSignal: 若为 1, 则保持该寄存器不动

## 2 ID(译码)阶段

译码阶段主要包括 ID\_MUXA, ID\_MUXB, ID\_MUXC, registers, ID\_Adder 模块, 最后进入 IDEX 寄存器。

#### 2.1 ID\_MUXA(用于生成第一个要读的寄存器的编号)

IDMUXA 模块（组合）	
输入	RA_in: IFID 寄存器上 IM_Command_Out 的【10:5】
输出	RA_out: 送往 registers 的要读的寄存器编号。
控制信号	用来控制选择在 RA_in 中截取寄存器编号或生成特殊寄存器的编号。

#### 2.2 ID\_MUXB(与 ID\_MUXA 完全一样, 用于生成第二个要读寄存器的编号)

#### 2.3 ID\_MUXC（用于生成要写回的寄存器的编号）

IDMUXC 模块（组合）	
输入	RC_in: IFID 寄存器上 IM_Command_Out 的【10:2】
输出	RC_out: 送往 IDEX 寄存器的要写回的寄存器编号
控制信号	用来控制选择在 RC_in 中截取寄存器编号或生成特殊寄存器的编号。

#### 2.4 IMME (用于扩展指令中的立即数)

IMME 模块（组合）	
输入	IMME_in: IFID 寄存器上 IM_Command_Out 的【10:0】
输出	IMME_out: 送往 ID_Adder 和 IDEX 寄存器的立即数值
控制信号	需要控制包括写选择哪些位作为立即数, 以及如何扩展 (有符号和无符号)。

#### 2.5 registers

当读数据时是组合逻辑, 写数据时是时序逻辑。如果正在读的寄存器编号与正在等待写入的寄存器编号相同, 则取正在等待写入的数值作为结果。

registers 模块（时序，上升沿触发）	
输入	RA : 要读的第一个寄存器编号; RB : 要读的第二个寄存器编号; WriteReg : 要写入的寄存器编号;

	WriteData：要写入的数据。
输出	OA：读的第一个寄存器的数值； OB：读的第二个寄存器的数值。
控制信号	CWB：控制是否需要写入

## 2.6 ID\_Adder(用于计算跳转用地址)

ID_Adder 模块（组合）	
输入	adderIn1：IFID 寄存器上暂存的 IM_PC_Out; adderIn2：IMME 输出的立即数。
输出	送往 PCmux 和 IDEX 寄存器的 adderOut
控制信号	无

## 2.7 IDEXRegister

IDEXRegister 模块（时序，上升沿触发）	
输入	EX_MuxA_In，EX_MuxB_In, EX_ALU_In: 给到 EX 阶段的控制信号； ME_DM_In: 给到 MEM 阶段的控制信号； WB_Mux_In，WB_CWB_In: 给到 WB 阶段的控制信号。 PC_In：指令附带的 PC； Command_In :指令； adder_In：ID_Adder 的结果； OA_In：第一个读寄存器的结果； OB_In：第二个读寄存器的结果； IMME_In：IMME 单元的结果； ID_MuxC_In：写回寄存器的编号； ID_MuxA_In：第一个被读的寄存器的编号； ID_MuxB_In：第二个被读寄存器的编号。
输出	EX_MuxA_Out，EX_MuxB_Out, EX_ALU_Out: 给到 EX 阶段的控制信号； ME_DM_Out: 给到 MEM 阶段的控制信号； WB_Mux_Out，WB_CWB_Out: 给到 WB 阶段的控制信号。 PC_Out：指令附带的 PC； Command_Out :指令； adder_Out：ID_Adder 的结果； OA_Out：第一个读寄存器的结果； OB_Out：第二个读寄存器的结果； IMME_Out：IMME 单元的结果； ID_MuxC_Out：写回寄存器的编号； ID_MuxA_Out：第一个被读的寄存器的编号； ID_MuxB_Out：第二个被读寄存器的编号。
控制信号	FlushSignal: 控制是否打气泡（即冲掉内容）； KeepSignal: 控制是否保留内容。

### 3 EX(执行)阶段

执行阶段主要包括 EXMuxA, EXMuxB, ALU 模块，最后进入 EXMEM 寄存器。

#### 3.1 EXMuxA(用于生成 ALU 的第一个操作数)

EXMUXA 模块（组合）	
输入	OA_in : IDEX 寄存器上保存的第一个寄存器输出 OB_in : IDEX 寄存器上保存的第二个寄存器输出 IMME_in : IDEX 寄存器上保存的寄存器单元输出 PC_in : IDEX 寄存器上保存的指令对应 PC ALU_MEM_in : EXMEM 寄存器上保存的 ALU 输出 ALU_WBMUX_in : MEMWB 寄存器上保存的 ALU 输出
输出	SA_out: 送往 ALU 的第一个操作数
控制信号	包括 control 模块生成的从 OA_in, OB_in, IMME_in, PC_in 中选择一个的信号；以及 forwardUnit 生成的从 OA_in(OB_in), ALU_MEM_in, ALU_WBMUX_in 中选择一个的信号。

#### 3.2 EXMuxB(与 EXMuxA 完全一样，用于生成 ALU 的第二个操作数)

#### 3.3 ALU

ALU 模块（组合）	
输入	src_A : 第一个操作数 src_B : 第二个操作数
输出	ALU_result: 计算结果 branch_Judge : 跳转指示,'1'代表跳转,'0'代表不跳转
控制信号	ALU_Op: 由 control unit 生成，一共 5 位，后四位负责控制 src_A 与 src_B 的运算，如加减等，第一位负责控制 ALU 部分的运行或暂停。

#### 3.4 EXMEM 寄存器

EXMEM 寄存器模块（时序，上升沿触发）	
输入	ME_DM_In, WB_Mux_In, WB_CWB_In : 继续向前传递的控制信号； PC_In : 指令对应的 PC； Command_In : 当前指令； ALU_In : ALU 的结果； ID_MuxC_In : 要写回的寄存器编号； ID_MuxA_In : 第一个读寄存器的编号； ID_MuxB_In : 第二个读寄存器的编号； ID_OB_In : ID 阶段读出的第二个寄存器的值。
输出	ME_DM_Out, WB_Mux_Out, WB_CWB_Out : 继续向前传递的控制信号； PC_Out : 指令对应的 PC； Command_Out : 当前指令； ALU_Out : ALU 的结果； ID_MuxC_Out : 要写回的寄存器编号； ID_MuxA_Out : 第一个读寄存器的编号； ID_MuxB_Out : 第二个读寄存器的编号；

	ID_OB_Out : ID 阶段读出的第二个寄存器的值。
控制信号	FlushSignal: 控制是否打气泡（即冲掉内容）; KeepSignal: 控制是否保留内容。

## 4 MEM(访存)阶段

访存阶段包括 DM 部分，最后进入 MEMWB 寄存器。其中 DM 部分放在 MemoryUnit 中统一介绍。

### 4.1 MEMWBRegister

MEMWBRegister 模块（时序，上升沿触发）	
输入	WB_Mux_In : WB_Mux 的控制信号; WB_CWB_In : 是否写回的控制信号; PC_In : 当前指令对应的 PC; Command_In : 当前指令内容; DM_In : DM 读出的地址; ALU_In : ALU 的计算结果; ID_MuxC_In : 写回寄存器的编号。
输出	WB_Mux_Out : WB_Mux 的控制信号; WB_CWB_Out : 是否写回的控制信号; PC_Out : 当前指令对应的 PC; Command_Out : 当前指令内容; DM_Out : DM 读出的地址; ALU_Out : ALU 的计算结果; ID_MuxC_Out : 写回寄存器的编号。
控制信号	FlushSignal: 控制是否打气泡（即冲掉内容）; KeepSignal: 控制是否保留内容。

## 5 WB(写回)阶段

写回阶段只有 WBMUX 一个模块。

### 5.1 WBMUX

WBMUX 模块（组合）	
输入	DM_in : 访存读出的结果; ADM_in: 绕过内存直接到达的结果。
输出	WBMUX_out: 用于写回寄存器的结果。
控制信号	用于二选一。

## 6 控制和冲突处理

在最初设计流水线时，为了能够达到 50M 频率，我们希望对于存储模块和其它模块不区分快慢时钟，而统一用一个时钟。这样访存时就至少需要多一个上升沿的时间。所以每次访问 DM 都需要将当前在 EXMEM 寄存器中的指令至少原地暂停一个周期，然后再继续前进。同时进行跳转时，针对 BEQZ (BNEZ, BTEQZ) 指令，如果需要跳转，则需要暂停两个周期来修正；针对 B 和 JR 指令，需要暂停一个周期。所以我们不得不在两个可能的地

方（IFID 和 IDEX 之间，EXMEM 和 MEMWB 之间）插入气泡。而且需要插入气泡的条件也需要根据 IFID 寄存器、IDEX 寄存器、EXMEM 寄存器，以及内存的暂停信号，还有之前的状态来判断。这使得控制模块的任务异常繁重。

为了避免冲突，我们让 control 模块只能看到 IFID 中的内容，并生成 MUX 选择器、ALU、内存、寄存器等通过查看单条指令就可以确定的信号。即 control 只在 ID 阶段工作。

### 6.1 Control

Control 模块（组合）	
输入	CommandIn: 来自 IFID 寄存器的指令
输出	ID_MuxA_Out, ID_MuxB_Out, ID_MuxC_Out, ID_IMME_Out: 给到 ID 阶段的控制信号; EX_MuxA_Out, EX_MuxB_Out, EX_ALU_Out: 给到 EX 阶段的控制信号; ME_DM_Out: 给到 MEM 阶段的控制信号; WB_Mux_Out, WB_CWB_Out: 给到 WB 阶段的控制信号。
控制信号	无

HazardDetection 模块则可以看到所有流水线寄存器中的内容，以及收到存储单元发来的暂停信号，据此进行暂停、打气泡等解决冲突，进行跳转的工作。它控制了所有流水线寄存器的 Flush 和 Keep 信号，还控制了 PCMUX 选择器来决定下一个 PC。

### 6.2 HazardDetection

HazardDetection 模块（时序，下降沿触发）	
输入	IFID_CommandIn : IFID 中的指令; IDEX_CommandIn : IDEX 中的指令; EXMEM_CommandIn : EXMEM 中的指令; MEMWB_CommandIn : MEMWB 中的指令; IM_Stop : IM 暂停信号; DM_Stop : DM 暂停信号; Branch_Judge :分支跳转结果。
输出	所有流水线寄存器的 Keep 和 Flush 信号，以及 PCMUX 的控制信号，还有让 PC 保持不变的信号。
控制信号	无

选择下降沿触发而不是组合逻辑的原因是在实验中发现这种连接了太多不同时序模块输出的组合逻辑模块表现不稳定，因为在上升沿到来时其输入可能不能严格同时变化。在下降沿时，其它时序模块都是稳定，此时进行判断并把结果锁住，在上升沿时就不会被众多输入的突变影响。由于同样的原因，forward\_Unit 也用了同样的方法来促进稳定。

### 6.3 Forward\_Unit

Forward_Unit 模块（时序，下降沿触发）	
输入	ExMemRd : EXMEM 寄存器上的写回寄存器编号; MemWbRd : MEMWB 寄存器上的写回寄存器编号; IdExReg_A : ALU 第一个操作数的源寄存器编号; IdExReg_B : ALU 第二个操作数的源寄存器编号; EX_MuxA_Out : Control 给 EX_MuxA 的控制信号; EX_MuxB_Out :Control 给 EX_MuxB 的控制信号。
输出	Forward_A : 给 EX_MuxA 的控制信号; Forward_B :给 EX_MuxB 的控制信号。



控制信号	无
------	---

这样下降沿触发的设计让实验中一般的跳转和旁路更加稳定，但也会引来一些问题。

由于 forward\_Unit 和 HazardDetection 被改为下降沿触发，在需要生成 Branch\_judge 信号给 HazardDetection 时会无法处理数据冲突。针对这个问题，我们采用了一个临时解决方案，就是让 HazardDetection 在 BEQZ,BNEZ,BTEQZ 跳转指令中承担起原来 forward\_Unit 和 ALU 共同的作用。这样的解决方案并不令人满意。

设计中的另一个缺陷在于对于 SW 指令，由于 ALU 的两个操作数会被 RA 和偏移量占用，没有旁路可以给 RB 解决数据冲突。当时采用的是暂停等待写回的处理方法，进一步增加了 HazardDetection 的负担。如果要重新设计流水线的话，应该增加一条旁路来解决这里可能引起的数据冲突。

## 7 串口内存的访问

我们计划完成一个统一的模块完成内存，串口的访问。这个模块在设计上，包含了数据通路中 IM 模块的功能，DM 模块的功能，访问 DM 的时候，可以根据地址自动的访问串口或相应的内存地址，同时这个模块包含了一定程度的冲突处理。

为了增加访问的效率，我们选择了 RAM1 储存数据，RAM2 储存指令，同样，为了尽可能增加时钟信号的利用率，我们同时使用了上升沿和下降沿触发，具体操作为，在上升沿给地址给控制信号给数据，下降沿取出数据(如果是读的话)，这样的话可以在一个时钟周期内，完成全部的读写操作，可在一个时钟周期内完成串口状态的访问，可在两个时钟周期内完成串口数据的读写。

在冲突处理中，在本模块这样规定：

1.如果当前有人正在占用(串口访问的第二个时钟周期)，则发送相应的暂停信号(req\_stall\_dm)给控制器，保证下一个周期重复得到同一条指令。

2.如果读指令(IM)和写指令(DM)，需要同时访问 ram2，这时候我们将执行 DM 指令，然后发送一个暂停信号(req\_stall\_dm)，保证在下一个周期进行 IM 访问时得到的 PC 值相同。

3.控制器会发送 keep 信号(im\_keep)，如果收到，则不更新 IM 寄存器

在访问中本模块这样规定：

1.如果访问指令地址小于 0x8000，则访问 RAM2，

2.如果访问指令地址为 0xBF00，则访问串口

3.如果访问指令地址为 0xBF01，则返回串口状态

4.否则访问 RAM1

接口说明：

MemoryUnit	
输入	--IM 部分 ImAddr: 指令地址 --DM 部分 dmAddr: 要访问的地址 dmIn: 写内存的输入数据 --内存串口控制信号 data_ready, tbre, tsre 串口的状态信号
输出	--IM 部分 ImOut: 读取的指令输出 Memory_PCOut: 当前访问的 PC 地址



	--DM 部分 dmOut: 读取内存的输出 --内存串口控制信号 ram1_oe, ram1_we, ram1_en: RAM1 控制信号 ram2_oe, ram2_we, ram2_en: RAM2 控制信号 ram1_addr, ram2_addr: 两个 RAM 的地址总线 ram1_data, ram2_data: 两个 RAM 的数据线 wrn,rdn: 串口读写控制信号 req_stall_im: 这个信号标志该周期暂停 IM req_stall_dm: 这个信号标志该周期暂停 IM
控制信号	DmControl: 给出 dm 指令的任务, 000 不做, 001 读 010 写 keep_im: 保持 IM 得到的指令 clk,rst: 时钟信号
调试信号	Display_im, display_dm 这个信号无任何功能仅用于调试

## VGA 模块-附加部分

我们原本希望以贪吃蛇游戏的形式体现附加功能 VGA, 该部分在规定的检查时间内已经独立完成, 但由于时间有限, 未能及时将 VGA 与数据通路合并起来, 因此该部分未经过助教检查。在此也介绍一下已经独立完成的 VGA 附加功能。

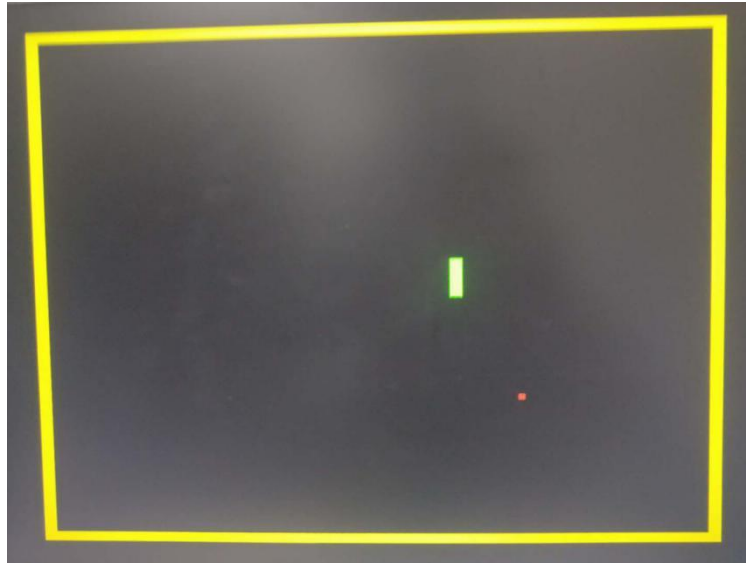
### 6.1 VHDL 部分

VGA 贪吃蛇	
输入	ram_data: ram2 中对应地址的数据, 代表对应位置的颜色 vga_start: vga 开始工作的信号
输出	ram_addr: 输出要访问的 ram2 地址 vga_finish: vga 结束工作的信号 Hs,vs:行同步、场同步信号 oRed: 红色输出 oGreen: 绿色输出 oBlue: 蓝色输出
控制信号	clk_vga:将主频进行分频处理后对 vga 部分的时钟信号

我们将主频分为了 vga、键盘输入、cpu 三个分频, 每个循环周期为 vga 提供了 1000000 个上升沿来绘制 vga 图像, 再将这些上升沿二分频, 一半上升沿负责输出需要访问的 ram2 地址, 一半上升沿负责输入对应地址中的值。我们的 vga 图像分辨率为 525×800,通过场同步、行同步信号, 在每个输入地址中的值的上升沿为对应(x,y)坐标赋三色的值, 并通过 oRed, oGreen, oBlue 这三个输出将对应颜色显示屏幕上。

### 6.2 汇编部分

贪吃蛇的汇编部分代码全部由已实现的指令构成, 并经过独立调试运行, 但由于没能及时与数据通路相结合, 因此无法通过串口读入汇编代码实现最终效果。初始界面测试效果如下:



我们将显示界面分为了  $54 \times 54$  个  $8 \times 8$  方格，每个方格对应一种颜色，将颜色信息储存在  $0x8000 \sim 0x8b63$  的地址中。当 vga 部分工作时，会依次读入这些地址中的值，并显示在屏幕上。此外，我们将果实信息存储在  $0x9100 \sim 0x91ff$ ，蛇身信息储存在  $0x9200 \sim 0x92ff$ ，每次移动方向存储在  $0x9300$  中，通过汇编代码进行维护。

以上为 vga 贪吃蛇部分已经独立实现的任务，我们希望通过键盘输入来修改  $0x9300$  中移动方向的值，再利用串口输入的汇编代码读入移动方向，维护 ram2 中的数据，再通过 vhd1 的输入输出将 ram2 中数据显示在屏幕上。

## 遇到的问题及解决方法

在加入了串口的读写操作和访问数据内存的操作后，用模拟器已经不容易模拟出板上的真实效果，所以我们开始用板子直接调试。我们让板子上的七段数码管输出 PC，让 16 位 LED 灯输出我们想看到的一些内部状态，把时钟信号绑定在手动时钟上。这种调试方法在手动输出 OK 的阶段帮我们解决了很多问题，但是在调试后续的 5 条指令时有些力不从心。因为每次要更换输出都需要重新编译，然后从监控程序的开始一直手按时钟到关注的 PC 处。这样的调试效率很低，加上我们的 RST 键有点松，敲时钟的时候一不小心碰到了就会前功尽弃。

为了克服这种方式的缺点，我们决定采用更高效的调试方法。我们绑定了 11M 的时钟，将 16 个拨码开关输入当作需要检查状态的 PC 值，让程序自动记录该 PC 值处 CPU 的 debug 输出并记录在 LED 灯上。如果需要更换要检查的 PC 值，只需重设拨码开关，按下 RST 键，松手后很快该 PC 处的 debug 输出就会留在 LED 灯上。

同时为了方便锁定死循环的错误，我们将拨码开关全零设计为锁定 PC 输出（在七段数码管显示）的信号。在程序遇到死循环时，将拨码开关置全零，然后把最后一个拨码开关在 0 和 1 之间拨动若干次，便可以大致确定出死循环的 PC 范围。在调试 R 指令时，我们通过这种方法发现死循环的原因是 TestR 函数每次都返回到 B TestR 这条指令之前，于是又会跳到 TestR 函数，导致了死循环。经过对这一反常现象的研究，我们发现了 IM 为每条指令配发其对应 PC 值时的错误。修正了这个错误后，监控程序的绝大部分功能就可以正常运行了。

## 实验总结

这真是一个最让我心累的大作业了，我们最后几天几乎定居在 4 楼的实验室，与 bug 奋战，真的是一次非常不堪回首的经历。最开始我们的设计了很多要完成的工作，最后基本上也就只完成了基础要求，还是刚好卡着时间在 DDL 的最后。即便如此，我们的 CPU 运行也是非常不稳定的，即使是完全同样的代码，在有的时候就会卡死，有的时候就可以正常执行，真的是令人十分抓狂。

虽然这一节的题目叫做实验总结，但我想可能作为实验教训也许更合适一点，我们在实验中走了一些弯路，在这里我归纳了几点。

- 1.首先是设计上的不明确，因为我们的代码是拆分完成的，在开始初期，我们有很多地方的实现不明确，比如数据的访问冲突，虽然我们在数据通路设计时讨论应该怎样去实现，但是在实际设计时，应该由谁了打信号，由谁暂停，什么时候拿到暂停信号却发生了很大的问题，在调试时，我们才发现暂停信号比实际要晚一个周期，导致我们不得不进行大量的改动，诸如此类的改动极大的影响了我们的效率。

- 2.其次是一些不够鲁棒的设计，比如我为了追求效率，在一个模块中同时使用了上升沿触发和下降沿触发，这导致了很大的 bug，还有就是我们在实际的电路设计时使用了大量锁存器的结构，这个实际上是非常不稳定的。很多这样的设计，为我们的电路带来了竞争与冒险，这可能也是为什么我们的电路如此不稳定。

- 3.调试，应该尽早的准备出 VGA，整个过程中，我们调试的工具只有 16 个小灯，这为我们的工作增加了不少的难度。

- 4.附加功能，我们在最初设计的时候只有 CPU 的设计，当我们尝试在这个基础上实现一个贪吃蛇的小程序时，在组合模块的过程中发生了很大的问题，不仅需要补充很多接口，而且整个结构的逻辑都要进行相应的修改，最后也不得不放弃了实现。

以前写了不少的代码，其中也不乏很多奇技淫巧的实现，也有很多先写代码再边写边设计的过程，甚至很多脑袋一热就随便写的代码，然后再 debug，这次大作业算是为这些习惯付出了惨痛的代价，硬件上 debug 总是困难得多，甚至有些即使发现了也不知道是什么的 bug，作为第一次和硬件打交道的我，这三周使我感受颇深。

以上就是我对本次作业的总结，最后，非常感谢李山山老师的指导，也非常感谢各位助教帮助，谢谢！