# CPU实验报告

计64 徐呈寅

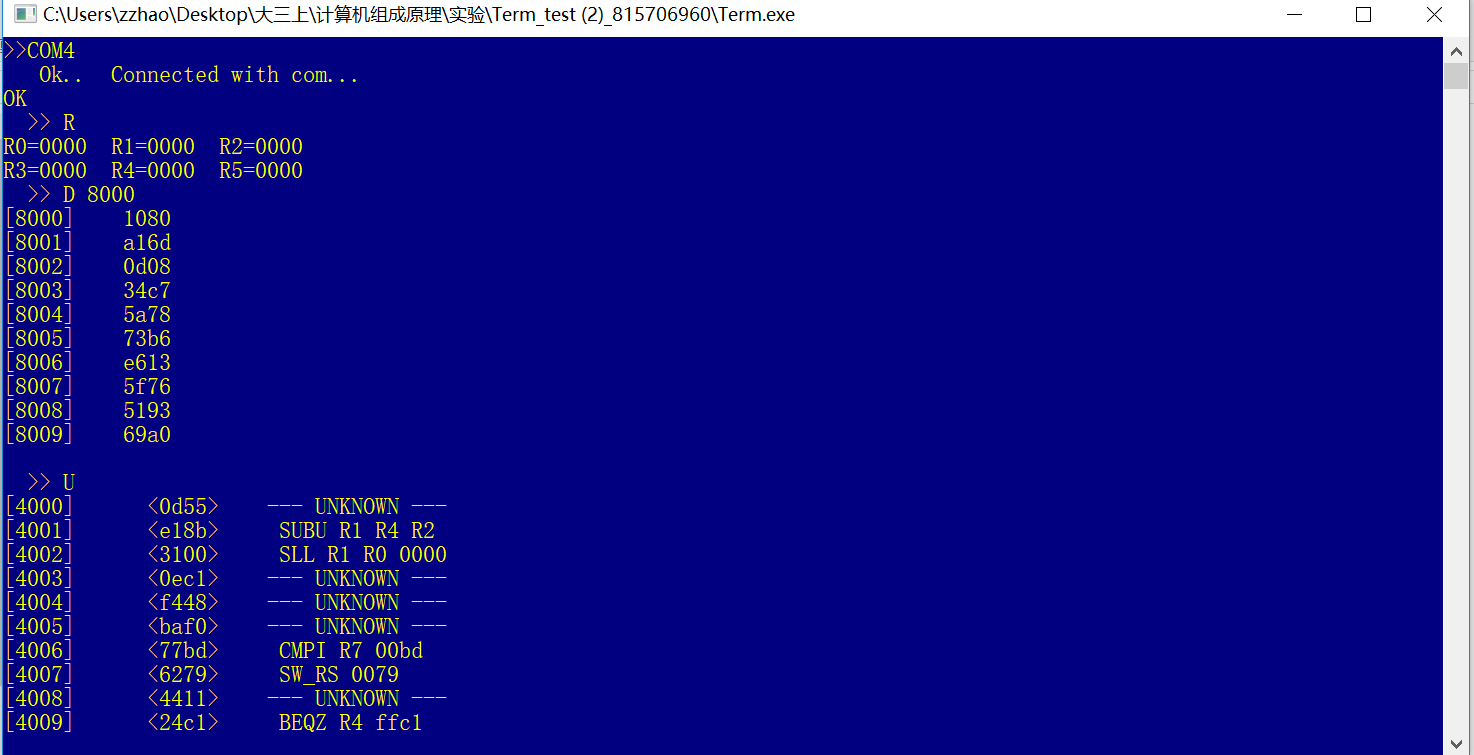
计65 章子豪

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



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

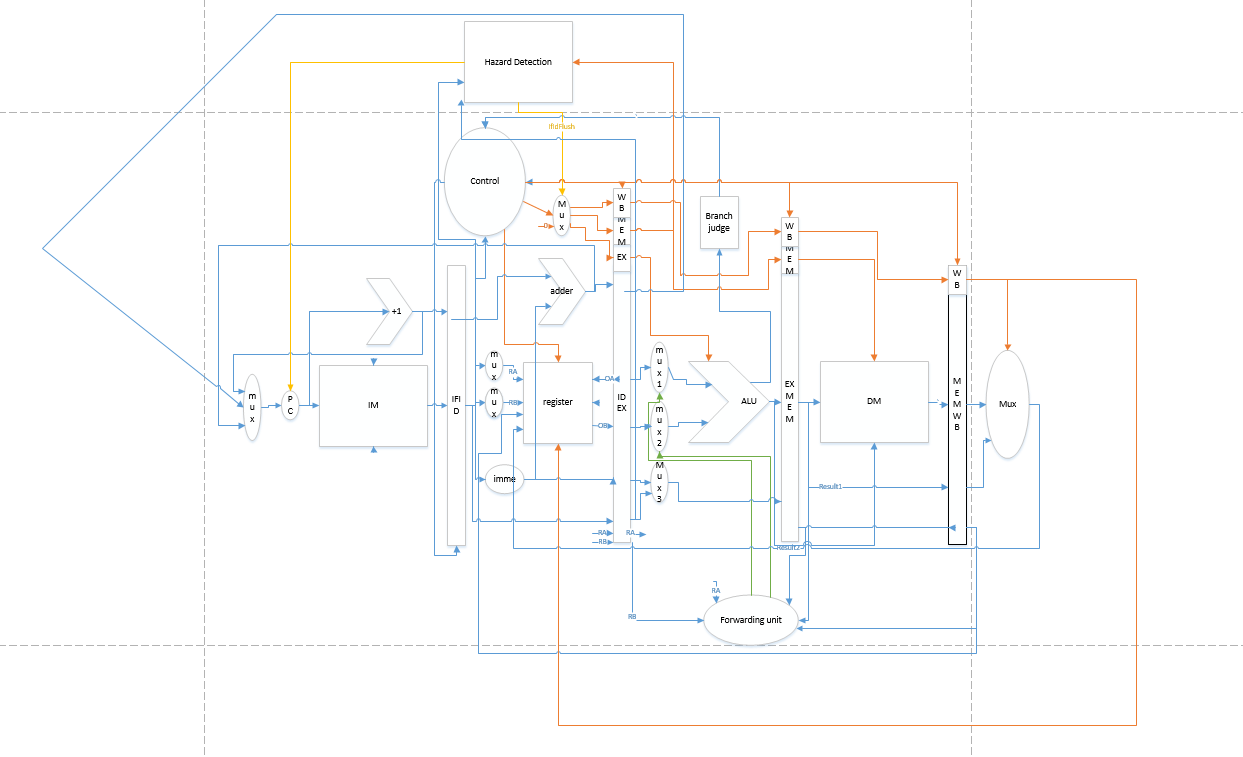
运算数据冲突的效率测试：2.25亿条—24.87s

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，PCAdder, 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 PCAdder(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IDMuxA, IDMuxB, IDMuxC, registers，ID\_Adder模块，最后进入IDEX寄存器。

**2.1 IDMuxA(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 IDMuxB(与IDMuxA完全一样，用于生成第二个要读寄存器的编号)**

**2.3 IDMuxC（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。

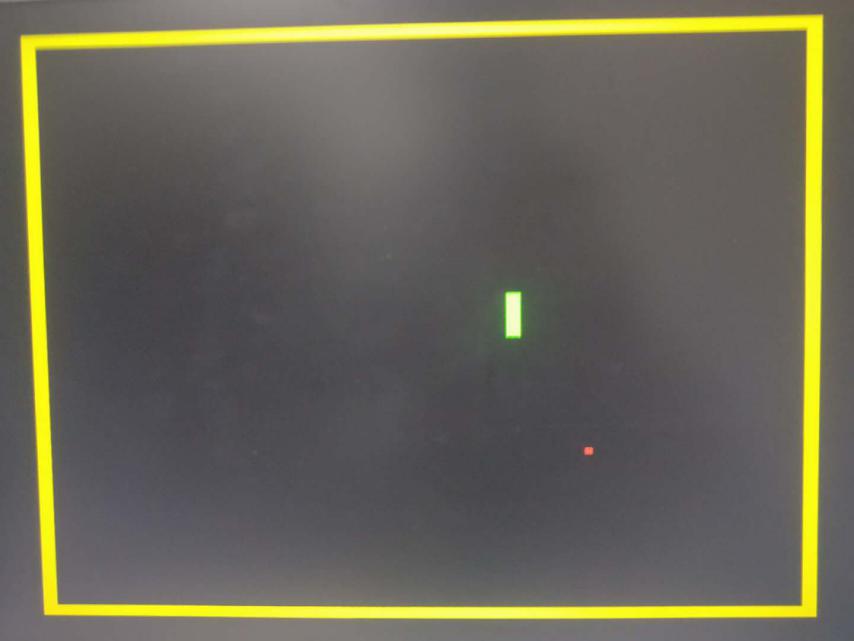
**6.1 VHDL部分**

|  |  |
| --- | --- |
| 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54×54个8×8方格，每个方格对应一种颜色，将颜色信息储存在0x8000～0x8b63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0x9100～0x91ff，蛇身信息储存在0x9200～0x92ff，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vhdl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的帮助，谢谢！