

计算机组成原理大实验报告

组长：杨煜

组员：肖迪，张艺庆

2014 年 12 月 22 日

目录

1 实验目标	4
1.1 基本目标	4
1.2 拓展目标	4
1.3 设计细节	4
2 流水线数据通路	5
2.1 【IF段】取指令	5
2.2 【ID段】指令译码与寄存器堆的读写	5
2.3 【EX段】执行或地址计算	6
2.4 【MEM段】存储器访问	6
2.5 【WB段】寄存器写回	6
3 控制信号	7
4 流水线模块实现	9
4.1 主要模块的功能与实现	9
4.1.1 寄存器堆模块	9
4.1.2 符号扩展器模块	10
4.1.3 解码器模块	11
4.1.4 ALU模块	12
4.1.5 RAM控制器模块	13
4.1.6 freezeCalculator模块	14
4.2 冲突处理	15
4.2.1 冒险检测单元	15
4.2.2 转发单元	16
4.3 扩展功能	17
4.3.1 VGA模块的实现	17
4.3.2 PS/2键盘控制器模块的实现	19

5 实验结果测试	20
5.1 系统整体测试	20
5.2 系统运行速度测试	20
5.3 VGA测试	21
5.4 系统编程测试	22
6 实验总结与体会	26
6.1 实验心得概述	26
6.2 合理分工	26
6.3 高效工具的使用	27
6.3.1 绘制电路图	27
6.3.2 代码管理工具	27
6.4 Verilog使用心得	27
6.5 外设调试心得	28
6.5.1 VGA调试心得	28
6.5.2 PS/2键盘调试心得	29
6.5.3 串口Ram调试心得	29
6.6 汇编程序调试心得	29
6.7 Debug的毅力	30
6.8 良好的沟通交流非常重要	30

1 实验目标

1.1 基本目标

- 实现五级指令流水CPU的数据通路和控制信号设计。
- 实现THCO MIPS指令系统，可运行监控程序。
- 实现在监控程序下运行应用程序。
- 实现FPGA端教学计算机与PC端Term终端程序的通信，通过Term操控教学计算机的运行。

1.2 拓展目标

- 实现数据旁路（转发单元），冒险检测处理单元，实现对delayslot的支持。
- 实现VGA模块，在单步模式下显示当前指令，格式为指令的英文单词。
- 实现PS/2键盘控制模块，实现使用PS/2键盘选择当前CPU的工作频率。

1.3 设计细节

- CPU主频为12.5MHz。
- 按照要求，除25条基本指令外，实现了NOT, SLLV, SRAV, CMPI, SLT扩展指令。

2 流水线数据通路

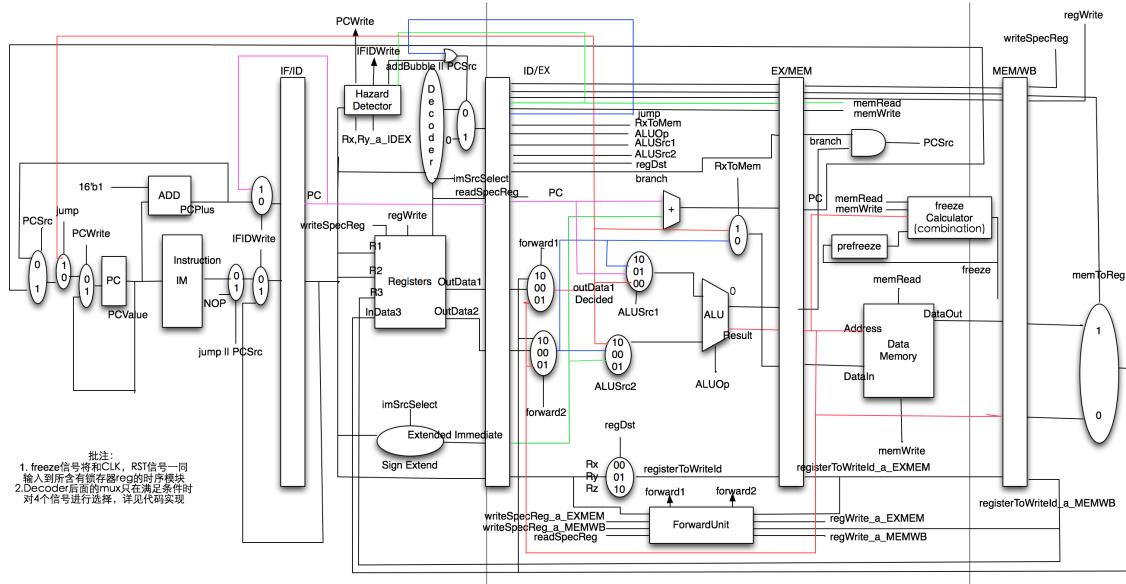


图 1: 数据通路图

2.1 【IF段】取指令

对于线性执行的代码片段，每个周期都将PC加1后的PCPlus值送回PC，同时Instruction Memory模块根据PC的输入值取出对应的指令保存在IF/ID中间寄存器中。同时PCPlus值也保存在IF/ID中间寄存器中，为了将来实现Branch指令（PCPlus加立即数）。

2.2 【ID段】指令译码与寄存器堆的读写

将保存在IF/ID中的指令送入Instruction Decoder进行译码，译码的过程为对于每一条指令生成不同的控制信号的组合输出；译码器Instruction Decoder输入14个控制信号，其中imSrcSelect和readSpecReg两个控制信号在这一阶段已经被使用，其余12个信号暂存到ID/EX中间寄存器当中；这12个信号中memWrite, regWrite, jump, branch, 4条指令首先经过数据选择器才送到ID/EX暂存，用来处理Branch指令后的第二条指令失效、Jump指令后第一条指令失效，强制插入气泡NOP指令时，失效这四个信号的功能。

16位Instruction的[10:8]位定义为R1, [7:5]位定义为R2, [4:2]位定义为R3；将Instruction的各个位输入到Registers中，并根据控制信号readSpecReg, writeSpecReg, regWrite进行寄存器的读写操作，读操作之后从outData2输出从R2读出值，outData1根据控制信号输

出R1读出值或者特殊寄存器的读出值，并暂存到ID/EX中间寄存器中。

各个含有立即数的指令的立即数长度不相同，经过Sign Extend模块可以将这些不同长度的立即数符号扩展为16位的立即数，并暂存到ID/EX中间寄存器中。

2.3 【EX段】执行或地址计算

根据ID/EX中间寄存器暂存的控制信号并结合数据旁路forward1, forward2控制信号，首先决定outData1Decided和outData2Decided；然后根据ALUSrc1和ALUSrc2控制信号进一步选择ALU的两个数据输入，根据ALUOp信号执行相应的运算，ALUResult存储到EX/MEM中间寄存器。

该阶段还用到的RxToMem信号决定将outData1Decided还是outData2Decided送入内存的DataIn接口；regDst信号从R1, R2, R3中选择一个作为写回寄存器；jump控制信号控制下一条PC保存的值。

2.4 【MEM段】存储器访问

根据EX/MEM中间寄存器中的控制信号、地址和数据，进行内存（Ram1）的读/写操作，读入的数据从DataIn输入，输出的数据从DataOut输出。实际工程中我们将Ram2和Ram1的控制模块合并为一个模块，并在Ram1控制模块中添加了对串口访问的支持。在这个阶段设置freezeCalculator模块的原因是为了在读写指令寄存器Ram2时候暂停流水线一个CPU周期，这部分的介绍详见freezeCalculator模块的说明。

2.5 【WB段】寄存器写回

根据MEM/WB中间寄存器的控制信号writeSpecReg, regWrite和memToReg控制信号，将要写回寄存器的数据写回到序号为registerToWriteId的寄存器中。

3 控制信号

在本次实验中，我们一共设计了14个控制信号，来控制数据通路上的十四个多路选择器选择合适的数据，我们设计控制信号和多路选择器的方法是渐进迭代的，首先，我们首先仿照书上设计出一个初版本的数据通路，然后将我们要实现的扩展指令一条一条的模拟在数据通路上的运行过程，并且给出相应的控制信号，如果现有的多路选择器能够满足实现这条指令的要求，那就继续下一条指令，只有当出现一条指令，现有的数据通路实在无法实现的时候，我们才在数据通路上添加一个新的多路选择器来满足新的指令的要求，并且我们力求做到使用最少的硬件资源来完成这件事情。14个控制信号分别是imSelector, ALUSrc2, memWrite, memRead, regDst, branch, regWrite, memToReg, op, readSpecReg, writeSpecReg, jump, ALUSrc1, rxToMem。每条指令所对应的控制信号我们在设计图表文件夹中的controlSignal.xls给出了，而每个控制信号每一位都代表什么含义，是起到什么控制作用的，我们在设计图表文件夹中的controlBitsMeaning.xls给出了。在这里不加赘述。

这里以一条典型指令来说明设计情况：

ADDIU rx immediate

该指令将立即数的值与寄存器rx的值相加，值再存回rx寄存器。它生成的控制信号见表1。

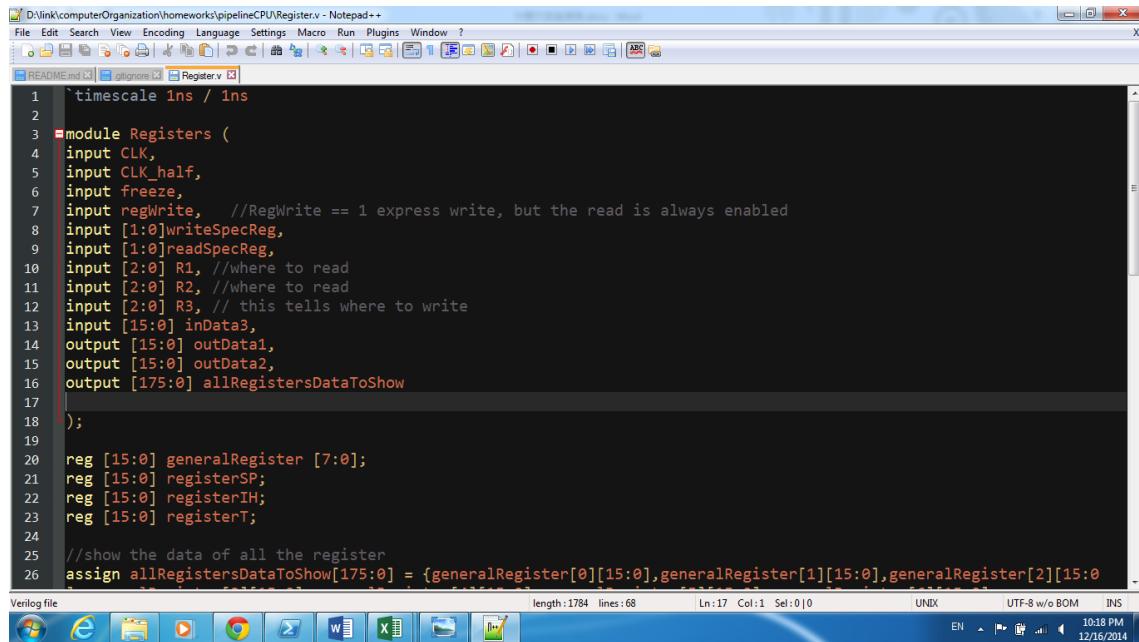
控制信号名	控制信号值	说明
imSelector	1000	该指令是从[7:0]位取出立即数，然后进行符号扩展
ALUSrc2	01	ALU的第二个操作数需要读取经过符号扩展的立即数
memWrite	00	指令不写寄存器，所以为00
memRead	00	该指令不读寄存器，所以为00
regDst	00	该指令写回的寄存器是依靠rx控制的
branch	0	该指令不分支
regWrite	1	该指令需要写寄存器堆
memToReg	0	该指令非装载指令，不需要从内存中读取操作数
Op	0000	加法指令，ALU执行加法操作
readSpecReg	00	该指令读通用寄存器，非特殊寄存器
writeSpecReg	00	该指令写通用寄存器，非特殊寄存器
jump	0	该指令不跳转
ALUSrc1	00	ALU的第一个操作数从正常寄存器堆的第一个输出中读入
rxToMem	x	不需要写内存所以这个控制信号无所谓

表 1: ADDIU指令控制信号

4 流水线模块实现

4.1 主要模块的功能与实现

4.1.1 寄存器堆模块



```

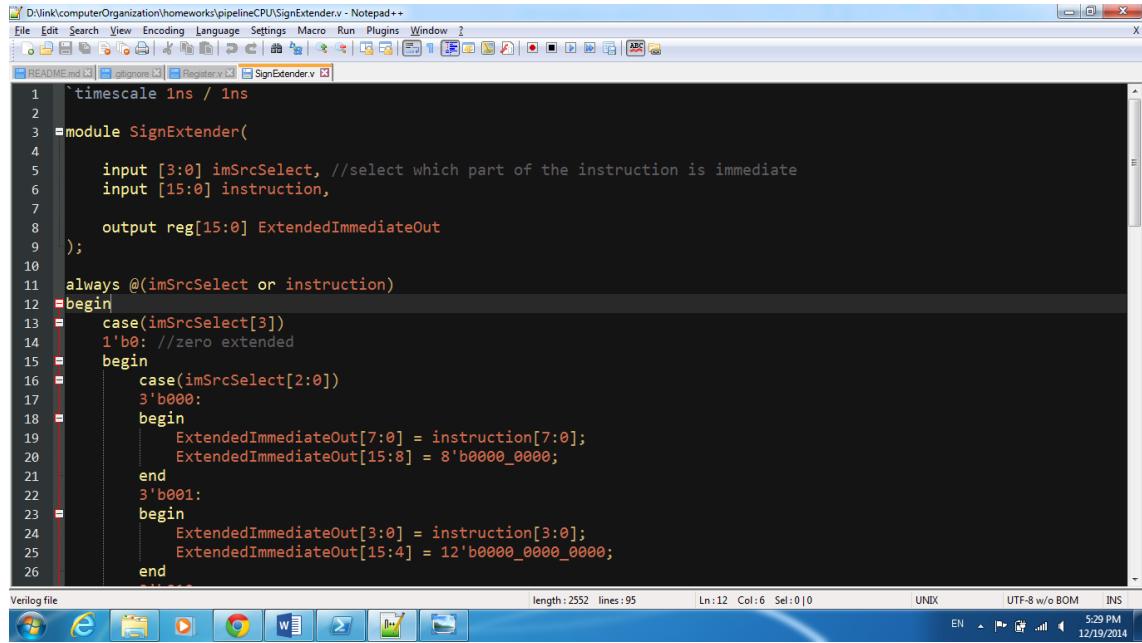
1 `timescale 1ns / 1ns
2
3 module Registers (
4   input CLK,
5   input CLK_half,
6   input freeze,
7   input regWrite, //RegWrite == 1 express write, but the read is always enabled
8   input [1:0]writeSpecReg,
9   input [1:0]readSpecReg,
10  input [2:0] R1, //where to read
11  input [2:0] R2, //where to read
12  input [2:0] R3, // this tells where to write
13  input [15:0] inData3,
14  output [15:0] outData1,
15  output [15:0] outData2,
16  output [175:0] allRegistersDataToShow
17 );
18
19
20 reg [15:0] generalRegister [7:0];
21 reg [15:0] registerSP;
22 reg [15:0] registerIH;
23 reg [15:0] registerT;
24
25 //show the data of all the register
26 assign allRegistersDataToShow[175:0] = {generalRegister[0][15:0],generalRegister[1][15:0],generalRegister[2][15:0]
Verilog file length:1784 lines:68 Ln:17 Col:1 Sel:0|0 UNIX UTF-8 w/o BOM INS

```

图 2: 寄存器堆模块

该模块实现了包括8个通用寄存器和3个特殊寄存器（T， SP， IH），每一个寄存器使用了verilog语言中的16位reg类型实现，电路本质就是触发器，读寄存器的值是通过组合逻辑读出的，所以outData1和outData2实时反应的就是最新的寄存器的值，只不过需要利用R1， R2和readSpecReg的值来确定是否要读取，同时确定读取的是哪个寄存器的值。而写入操作只是发生在主频时钟的下降沿的时候，由regWrite， writeSpecReg来确定是否要写寄存器，由R3确定写入的是哪个通用的寄存器。而读出的数据就通过outData1，和outData2来读出。

4.1.2 符号扩展器模块



```

1 `timescale 1ns / 1ns
2
3 module SignExtender(
4     input [3:0] imSrcSelect, //select which part of the instruction is immediate
5     input [15:0] instruction,
6
7     output reg[15:0] ExtendedImmediateOut
8 );
9
10 always @(imSrcSelect or instruction)
11 begin
12     case(imSrcSelect[3])
13         1'b0: //zero extended
14             begin
15                 case(imSrcSelect[2:0])
16                     3'b000:
17                         begin
18                             ExtendedImmediateOut[7:0] = instruction[7:0];
19                             ExtendedImmediateOut[15:8] = 8'b0000_0000;
20                         end
21                     3'b001:
22                         begin
23                             ExtendedImmediateOut[3:0] = instruction[3:0];
24                             ExtendedImmediateOut[15:4] = 12'b0000_0000_0000;
25                         end
26                 end
27             end
28         endcase
29     end
30 endmodule

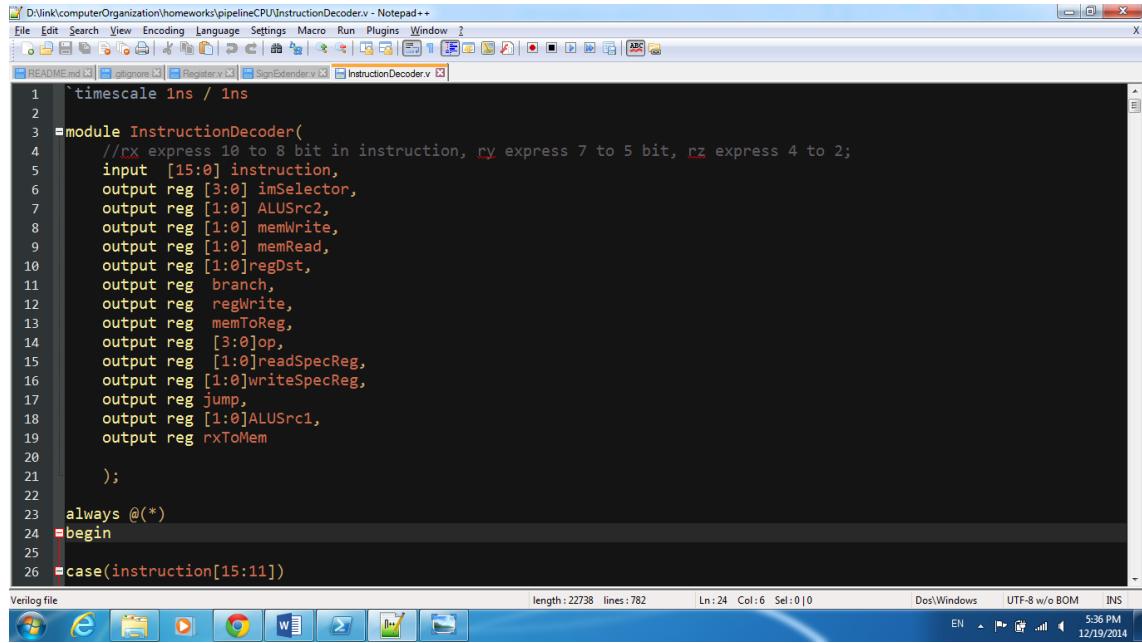
```

The screenshot shows a Notepad+ window displaying Verilog code for a 'SignExtender' module. The code defines a module with an input for immediate source selection (imSrcSelect) and an input for the instruction. It has an output for the extended immediate value. The logic inside the module uses cases based on imSrcSelect[3] to determine the extension method. For imSrcSelect[3]=1'b0, it uses zero extension. For imSrcSelect[3]=3'b000, it extends the lower 8 bits of the instruction to the full 16-bit output. For imSrcSelect[3]=3'b001, it extends the lower 4 bits of the instruction to the full 16-bit output. The code is annotated with comments explaining the logic.

图 3: 符号扩展器模块

由于THUMIPS的指令集中的指令立即数可能出现的位置和位数有很多种可能，符号扩展器相对于标准MIPS来说相对复杂，但也是归根结底还是简单的组合逻辑。该组合逻辑通过解码器产生的控制信号imSrcSelect来选择和扩展指令中可能存在的立即数，扩展的方式由符号扩展和0扩展，模块的输出是16位的经过适当扩展的立即数。控制位相应的意义和对应的功能请详见设计图表/controlBitsMeaning。

4.1.3 解码器模块



```

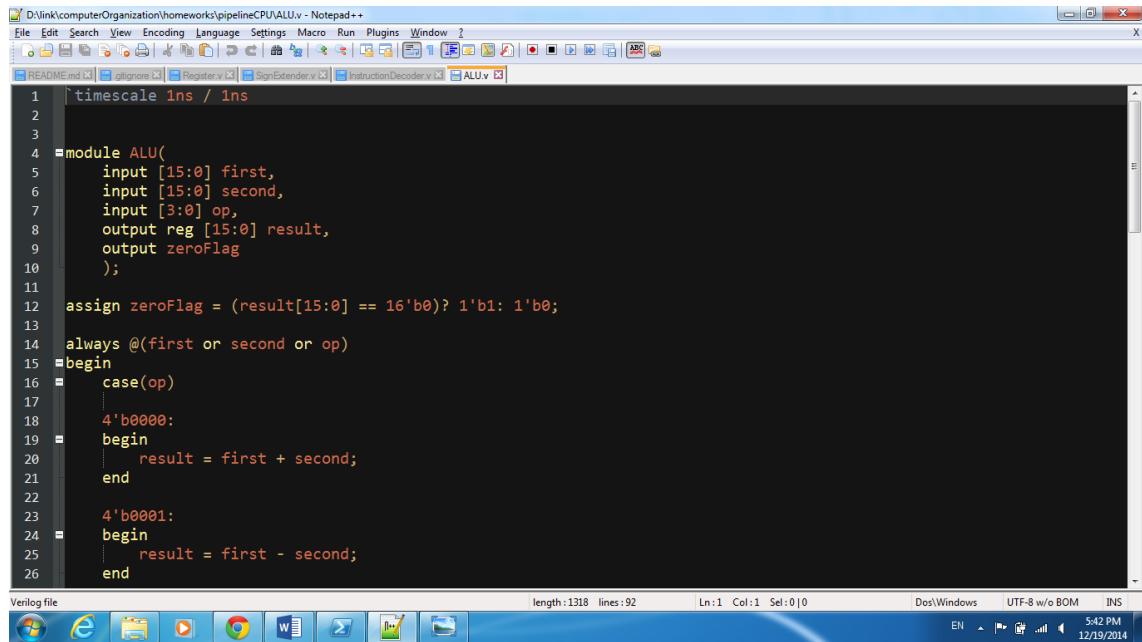
1 `timescale 1ns / 1ns
2
3 module InstructionDecoder(
4     //rx express 10 to 8 bit in instruction, ry express 7 to 5 bit, rz express 4 to 2;
5     input [15:0] instruction,
6     output reg [3:0] imSelector,
7     output reg [1:0] ALUSrc2,
8     output reg [1:0] memWwrite,
9     output reg [1:0] memRead,
10    output reg [1:0] regDst,
11    output reg branch,
12    output reg regWrite,
13    output reg memToReg,
14    output reg [3:0] op,
15    output reg [1:0] readSpecReg,
16    output reg [1:0] writeSpecReg,
17    output reg jump,
18    output reg [1:0] ALUSrc1,
19    output reg rxToMem
20 );
21
22 always @(*)
23 begin
24     case(instruction[15:11])
25
26 endcase
27 end
28
29 endmodule

```

图 4: 解码器模块

解码器模块虽然代码量比较大，但是逻辑十分清楚，是一个组合逻辑，输入是一个16位的指令，输出是该条指令相对应的14个控制信号，我们使用了verilog常用的switch-case语句来实现这个逻辑。相应的每一条指令对应的控制信号可以在设计图表/controlSingal查到。实现这个控制器是一项比较繁杂的工作，我们在实现的时候采用了一个人编码，另外一个人复核的实现策略，果然复核的时候检查出了上一个人的实现中的一些问题。

4.1.4 ALU模块



```

1 `timescale 1ns / 1ns
2
3
4 module ALU(
5     input [15:0] first,
6     input [15:0] second,
7     input [3:0] op,
8     output reg [15:0] result,
9     output zeroFlag
10 );
11
12 assign zeroFlag = (result[15:0] == 16'b0)? 1'b1: 1'b0;
13
14 always @(first or second or op)
15 begin
16     case(op)
17
18         4'b0000:
19             begin
20                 result = first + second;
21             end
22
23         4'b0001:
24             begin
25                 result = first - second;
26             end

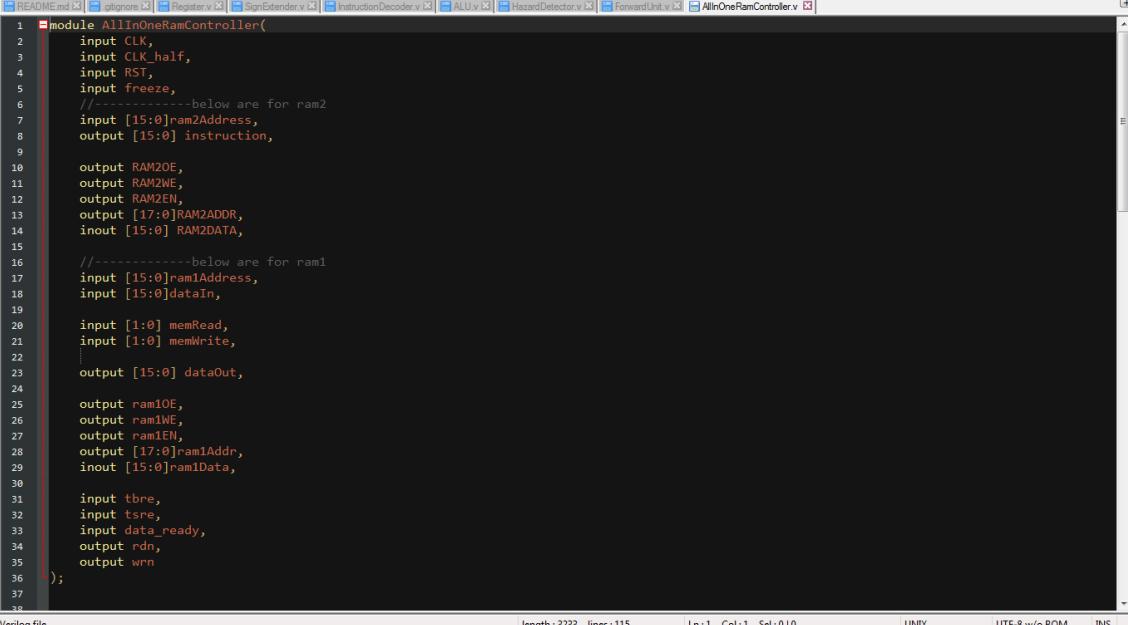
```

The screenshot shows a Notepad++ window displaying Verilog code for an ALU module. The code defines a module with four inputs: first (16-bit), second (16-bit), op (4-bit), and zeroFlag (1-bit). It uses an always block to implement addition and subtraction based on the op code. The zeroFlag output is assigned the value of 1 if the result is 0, otherwise 0. The code is written in a standard Verilog style with indentation.

图 5: ALU模块

ALU是整个CPU的核心，但是实现起来却比较简单，它的功能就是根据操作码以不同的方式对两个操作数first和second进行操作，将结果输出到result里面，同时，若result等于全0，将zeroFlag设为1。在整个数据通路的设计中，我们为了使得总体的设计简单，稍微将ALU的可能进行的操作复杂化了。比如说，ALU中有一项看起来很奇特的操作，将输出的结果设置为第一个操作数first，也就是说操作数直接通过ALU，没有进行任何操作。这样做的目的是简化外部顶层逻辑的设计，因为有些指令，不需要ALU进行操作，本来我们可以在顶层通路的设计中加一条直接链接的线，然后在结果处加一个多路选择器，但是这样会使得顶层逻辑变得复杂，于是我们就没有这么做。这里的设计思想其实是某种平衡，若模块设计适当复杂，顶层逻辑就可以适当简单，所以我们就希望通过子模块来实现顶层模块的功能，来降低顶层模块设计的难度。

4.1.5 RAM控制器模块



```

1 module AllInOneRamController(
2     input CLK,
3     input CLK_half,
4     input RST,
5     input freeze,
6     //-----below are for ram2
7     input [15:0]ram2Address,
8     output [15:0] instruction,
9
10    output RAM2OE,
11    output RAM2WE,
12    output RAM2EN,
13    output [17:0]RAM2ADDR,
14    inout [15:0] RAM2DATA,
15
16    //-----below are for ram1
17    input [15:0]ram1Address,
18    input [15:0]dataIn,
19
20    input [1:0] memRead,
21    input [1:0] memWrite,
22    ...
23    output [15:0] dataOut,
24
25    output ram1OE,
26    output ram1WE,
27    output ram1EN,
28    output [17:0]ram1Addr,
29    inout [15:0]ram1Data,
30
31    input tbre,
32    input tsre,
33    input data_ready,
34    output rdn,
35    output wrn
36 );
37

```

Verilog file length: 3233 lines: 115 Ln:1 Col:1 Sel:0|0 UNIX UTF-8 w/o BOM INS

图 6: RAM模块

RAM模块是整个流水线CPU中的最重要的模块之一，也是开发过程中尤其需要注意时序的模块。正如数据通路图所示，我们的CPU中用到了指令存储器（Instruction Memory）和数据存储器（Data Memory）分别使用了板上的Ram2和Ram1进行存储。同时为了与监控程序通信，我们将串口控制器集成在了Ram1模块中，在遇到读写BF00和BF01两个地址时将要对Ram1进行的读写操作映射成为对串口控制信号和数据信号的发送和接收。以上全部写在了AllInOneRamController.v这个模块中。

我们将25MHz和12.5MHz的时钟分别以CLK和CLK_half送入该模块，CLK_half表示一个CPU时钟周期，也就是说在一个CPU时钟周期内我们可以遇到CLK（25MHz）的4个时钟上升沿或下降沿。我们设计的时序是在每个CLK_half的下降沿，也就是CLK的第二个上升沿，所存前面EX/MEM中间寄存器发送来的数据。组合逻辑的建立也是需要时间的，对于读指令或者写指令，我们在CLK第二个下降沿拉低对应的读信号或者写信号（若地址是BF00或BF01则拉低对应的串口读写信号），被拉低的读信号和写信号一直保持到下一个CPU时钟周期的CLK第一个下降沿，整个保持时间为半个CPU时钟周期，向后面的模块输出控制信号的时间恰好覆盖了CPU的上升沿，能够确保下一个模块在给定的控制信号下正确处理该模块读出的数据（或正确读写串口）。这样做不存在任何的竞争与冒险

的问题，系统的可靠性非常强，但是这样做带来的一个问题是不能实现Ram2指令存储器的读写，关于这一个问题的解决详见下文freezeCalculator模块的说明。

4.1.6 freezeCalculator模块

```
//prefreeze is a register defined ahead;
//freeze is a wire defined ahead; 莱
assign freeze = ( (~prefreeze) && (memWrite_a_EXMEM[1:0] != 2'b0) && (ALUResult_a_EXMEM[15:14] == 2'b01)
&& (memRead_a_EXMEM[1:0] == 2'b0) ) ? 1'b1 : 1'b0;
always @ (posedge button_half)
begin
| prefreeze <= freeze;
end
```

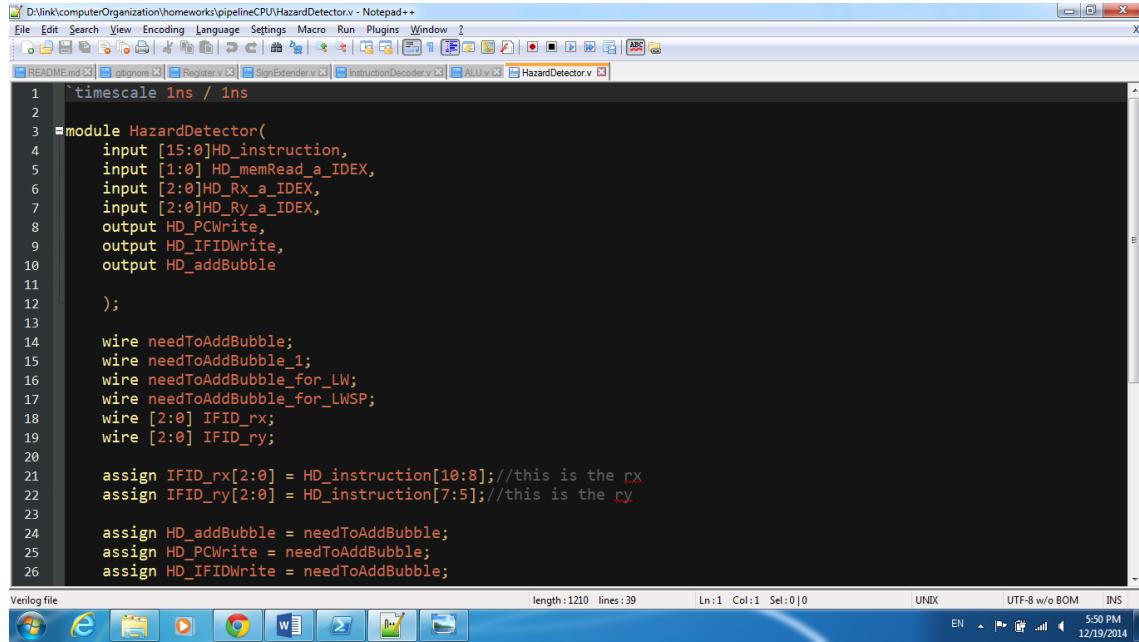
图 7: freeze控制寄存器

正如上文所说，当我们要读写指令寄存器Ram2时，我们采取的方法是暂停流水线：一条指令的控制信号被分析为读写指令而且内存地址在4000到7FFF之间时候，同时上一条指令的freeze信号不为1时（prefreeze寄存器暂存上一个周期的freeze信号）将freeze信号设为1，并将freeze信号送到其余所有含有寄存器的模块。这些模块的实现过程为只有在freeze信号为0时才进行所有的always块正常赋值，否则保持。

严格来讲freezeCalculator并非一个独立的模块，而是为了更好的在数据通路上绘制出而设计的，其本质就是用组合逻辑实现的一个数据选择器，配合prefreeze寄存器，完成暂停流水线的功能，上图为顶层模块PipelineCPU.v中实现freeze功能的代码片段。

4.2 冲突处理

4.2.1 冒险检测单元



```

1 `timescale 1ns / 1ns
2
3 module HazardDetector(
4     input [15:0] HD_instruction,
5     input [1:0] HD_memRead_a_INDEX,
6     input [2:0] HD_Rx_a_INDEX,
7     input [2:0] HD_Ry_a_INDEX,
8     output HD_PWrite,
9     output HD_IFIDWrite,
10    output HD_addBubble
11 );
12
13
14     wire needToAddBubble;
15     wire needToAddBubble_1;
16     wire needToAddBubble_for_LW;
17     wire needToAddBubble_for_LWSP;
18     wire [2:0] IFID_rx;
19     wire [2:0] IFID_ry;
20
21     assign IFID_rx[2:0] = HD_instruction[10:8];//this is the rx
22     assign IFID_ry[2:0] = HD_instruction[7:5];//this is the ry
23
24     assign HD_addBubble = needToAddBubble;
25     assign HD_PWrite = needToAddBubble;
26     assign HD_IFIDWrite = needToAddBubble;

```

The screenshot shows a Notepad++ window displaying Verilog code for a 'HazardDetector' module. The code defines the module with various inputs and outputs, including HD_instruction, HD_memRead_a_INDEX, HD_Rx_a_INDEX, HD_Ry_a_INDEX, HD_PWrite, HD_IFIDWrite, and HD_addBubble. It includes logic to assign IFID_rx and IFID_ry based on HD_instruction, and to set HD_addBubble, HD_PWrite, and HD_IFIDWrite to the value of needToAddBubble. The code is annotated with comments explaining the assignments.

图 8: 冒险检测单元

由于流水线的存在，若前一条是装载指令，后一条是指令需要用到装载指令的目标地址的值，那么这时，装载指令还没有执行，在时间上通过转发解决这个问题是不可能的，所以我们必须通过插入气泡（NOP指令）的方式来在事实上暂停流水线一个周期。形式化的表述这种情况的发生是：

HD_memRead_a_INDEX

&& (HD_Ry_a_INDEX[2:0] == IFID_rx[2:0] || HD_Ry_a_INDEX[2:0] == IFID_ry[2:0])

表示的意义就是（前一条指令是装载指令&& 装载的目的地址==本条指令的任意一个源地址）

插入气泡的方式是通过将新解码出来的指令变为NOP（所有控制信号变为0即可），然后保持PC值和已经读出到IF/ID流水寄存器的的值就可以。其中，HD_PWrite是用来保持的PC的，HD_IFIDWrite是用来保持IF/ID流水寄存器的，HD_addBubble是用来将指令变为NOP的。

4.2.2 转发单元

```

1 `timescale 1ns / 1ns
2
3 module ForwardUnit(
4     input [2:0] Rx_a_INDEX,
5     input [2:0] Ry_a_INDEX,
6     input [2:0] Rz_a_INDEX,
7     input regWrite_a_EXMEM,
8     input regWrite_a_MEMWB,
9     input [2:0] registerToWriteId_a_EXMEM,
10    input [2:0] registerToWriteId_a_MEMWB,
11    input [1:0] writeSpecReg_a_EXMEM,
12    input [1:0] writeSpecReg_a_MEMWB,
13    input [1:0] readSpecReg_a_INDEX,
14
15    output [1:0] forward1,
16    output [1:0] forward2
17 );
18
19 //the wire for the forward 1
20 wire [1:0]forward1_a;
21 wire [1:0]forward1_EXMEM; //regWrite_a_MEMWB == 1'b1 && regWrite_a_EXMEM == 1'b1
22 wire [1:0]forward1_b;
23 wire [1:0]forward1_c;
24 wire [1:0]forward1_EX; //regWrite_a_MEMWB == 1'b0 && regWrite_a_EXMEM == 1'b1
25 wire [1:0]forward1_MEM; //regWrite_a_MEMWB == 1'b1 && regWrite_a_EXMEM == 1'b0
26 wire [1:0]forward1_maybeEX;

```

图 9: 转发单元

转发单元的出现也是由于使用了流水线本身，假如前一（二）条指令是一条需要写寄存器堆的指令，后一（二）条指令刚好要读寄存器堆，如果他们写和读的地址是同一个位置，就会发生问题，因为写指令是在最后一个阶段WB才写的，下一（二）条指令没有办法及时读取到正确的值，所以就需要通过旁路将正确的值及时的转发过去由于转发的情况较多，形式化的描述请见代码，非形式化的描述如下：

如果（前一或者二条指令是需要写寄存器的） $\&\&$ ((前一条指令的写入的地址==后一条指令读取的地址) || ((前一条指令的写入的地址!=后一条指令读取的地址) $\&\&$ (前两条的指令的写入地址==后一条指令的写入的地址)), 那么就使用相应的转发。

之所以有这么多复杂的条件，是因为不但前一条指令满足目的地址等于源地址的条件需要转发，前两条指令也需要转发。而且还会出现前一条和两条指令都满足转发条件的情况，这个时候需要转发最近的也是最新的那条。

转发单元虽然内在逻辑较为复杂，但是本质上还是一个组合逻辑，控制的输出也只有两个，分别命名为forward1，和forward2，分别控制ALU的两个操作数的第一次选择，他们选择ALU的操作数是通过从寄存器堆的结果读取出来还是通过转发得到。

4.3 扩展功能

4.3.1 VGA模块的实现

VGA（指令跳转）相关的各模块含义如下，具体实现详见工程代码

顶层模块 传入参数：指令（instruction）、时钟信号（clk）

模块	模块功能	需调用的模块
<pre>module VGA(input clk, rst, output hs, vs, output [2:0] r, g, b, input [15:0] instruction //input [15:0] PC,);</pre>	VGA实现模块的顶层模块	VGAScanner、 VGAPainter
<pre>module VGAScanner(input clk, rst, output reg hs, vs, output reg [10:0] x, y);</pre>	实现“扫描”的模块	无
<pre>module VGAPainter(input [10:0] x, y, input [15:0] instruction, //input [15:0] pc, output reg [2:0] r, g, b);</pre>	实现“渲染绘制”的模块	paintLogoA~paintLogoZ
<pre>module paintRect(input enable, input [10:0] x, y, input [10:0] cx, cy, input [10:0] width, height, output reg hit);</pre>	实现水平线段和竖直线段 的绘制	无
<pre>module paintPosPara(input enable, input [10:0] x, y, input [10:0] cx, cy, input [10:0] width, height, output reg hit);</pre>	实现对角线的绘制	无
<pre>module paintNegPara(input enable, input [10:0] x, y, input [10:0] cx, cy, input [10:0] width, height, output reg hit);</pre>	实现反对角线的绘制	无
<pre>module paintLogoA(input enable, input [10:0] x, y, input [10:0] delt, output reg hit);</pre>	实现字母绘制	paintRect、 paintNegPara、 paintPosPara

表 2: VGA模块及功能说明

顶层模块调用：VGAPainter和VGAScanner两个模块，进行显示屏上内容的显示。

字母绘制模块 我们发现，本次实验中所有字母都能用三种笔画表示出来，它们分别是：直线、水平或者竖直直线、从右上角到左下角的对角线绘制。在我们的实现模块中，我们调用以下三个模块完成这几种笔画的绘制从而完成字母绘制、数字绘制功能：

paintRect.v: 完成水平或者竖直直线的绘制

paintPosPara.v: 完成从左上角到右下角的对角线绘制

paintNegPara.v: 完成从右上角到左下角的对角线绘制

扫描模块 原理

(1) 由于 $640*480$ 像素分辨率，在帧频率为 60Hz 时需要的像素时钟为 25.18Hz ，所以首先对输入的 50MHz 的时钟信号进行二分频，得到 25MHz 的时钟。

(2) 行区间像素扫描：在包含消隐区的情况下，行像素为800，因此在0~799之间循环扫描。

(3) 场区间像素扫描：当扫描完一行时，场区间的行数加一，在包含消隐区的情况下，场像素为525，因此在0~524之间循环扫描。

(4) 设置行同步信号：根据上文所述，当行扫描像素区间在656~752之间时，行同步信号为低电平，其余时为高电平。

(5) 设置场同步信号：在扫描至490和491行时，场同步信号为低电平，其余行时为高电平。

(6) 输出行场同步信号。

绘图模块 实例化：由于在Verilog语言中，代码描述的是电路的结构，所以我们无法在always语句块中实例化模块，我们只能在always块外面先实例化好模块，并给每个模块增加两个变量，一个控制字母是否显示，一个控制字母的偏移量。

调用模块： paintLogoA paintLogoZ

主要遇到的困难 在实验中我们主要遇到的问题是

语法问题：由于我们使用的是大家都没有使用的一种新语言，所以我们都是自己探索的新语言的用法，这是一个比较艰难的过程。在遇到任何bug的时候，我们没有课上其他同学一起讨论，都只能自己解决遇到的问题，这无疑是颇具有挑战性的。

实例化问题：由于Verilog是硬件描述语言，跟我们平时编程使用的面向软件的语言还是不一样的，比如当我们使用Verilog的时候，我们不能在always块里面实例化对象，

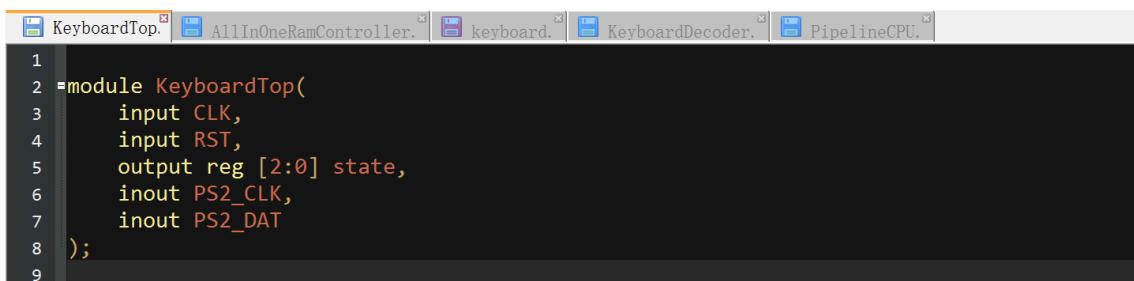
因为实例化的对象都对应着电路中的结构。我们只能通过先在always块外部实例化好了，添加enable变量在always块中控制实例化好的对象出现与否。

代码长度过长，debug比较繁琐：由于我们的字母绘制部分是一个一个字母绘制出来的，所以整个实现指令实时跳转的部分代码文件较多，代码长度较长。我们遇到的bug都是一些很隐蔽的小细节的bug，所以调起bug也是十分困难的。在以后类似的设计中，我们应该想办法换用不同的设计模式，减少代码的长度。

要让显示器在精确位置显示是不容易的：由于我们并不是特别了解使用的VGA的内部构造是什么样的，所以我们如果要让内容显示在显示器的特定位置，我们就只能采用连蒙带猜的方式调整参数，使得内容最终显示在我们想要的位置上，不过这样的方式比较费时费力，不是一个很好地方法。在以后遇到这种情况的时候，说不定我们先大致弄清了VGA显示模块的物理构造，能更快地解决问题。

4.3.2 PS/2键盘控制器模块的实现

PS/2键盘控制器分三个模块，分别为KeyboardTop.v, Keyboard.v和KeyboardDecoder.v文件，其中由KeyboardTop.v提供对外接口，由keyBoardState向外输出。



```

1
2 module KeyboardTop(
3     input CLK,
4     input RST,
5     output reg [2:0] state,
6     inout PS2_CLK,
7     inout PS2_DAT
8 );
9

```

图 10: 键盘控制模块

Keyboard.v文件实现了FPGA与外部PS2键盘通信的过程，每次读到PS/2键盘发送来的数据时候存储到shift_reg寄存器中，cnt和ct分别计数到指定位置的时候，从shift_reg寄存器中取出指定的值，既为该次发送的两位16进制值。

KeyboardDecoder实现了将PS2键盘发送来的通码和断码进行边沿检测，并采用优先编码的方式（后按下的键会覆盖掉前面按下的键）来将按键8, 2, 4, 6, +, Enter映射为单步，1000Hz, 1MHz, 1Hz, 25MHz, 12.5MHz的时钟频率。实际调用过程中这些频率均可以稳定输出。但CPU最大能够跑到12.5MHz稳定运行。

5 实验结果测试

5.1 系统整体测试

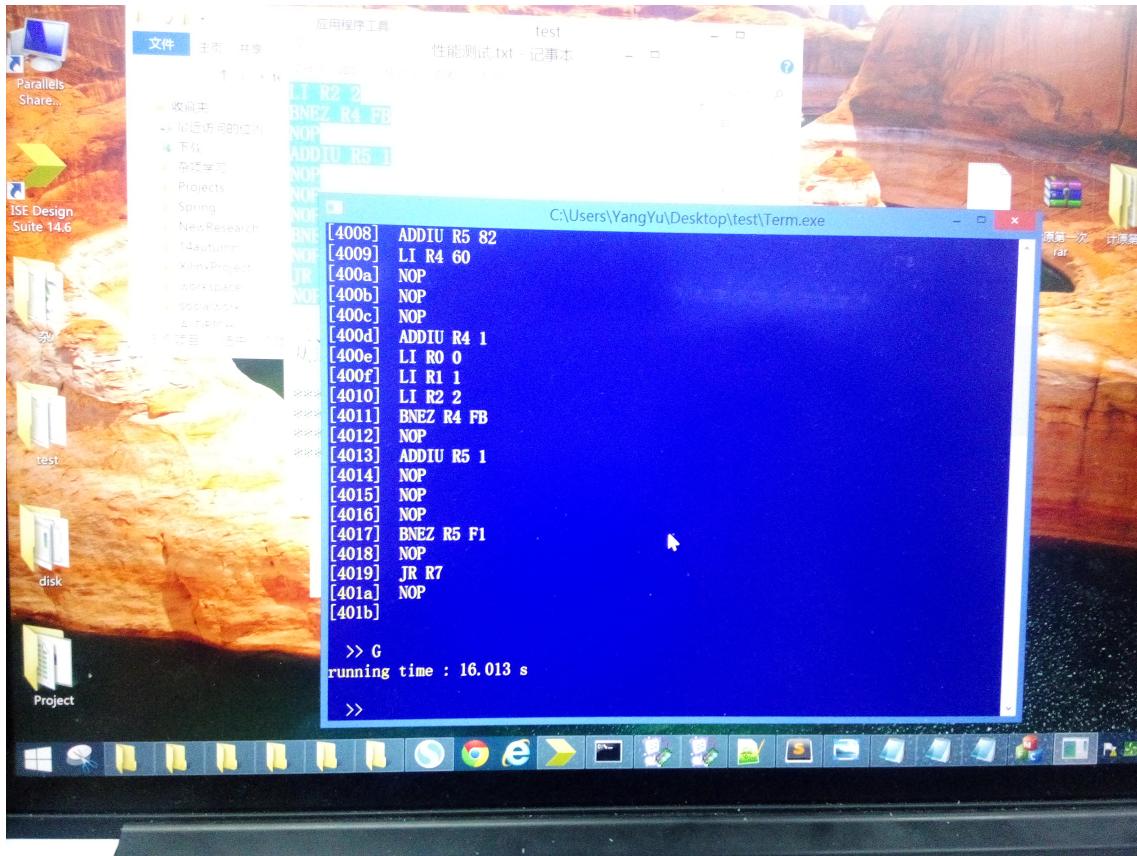


图 11: 整体运行测试

经过测试，系统可以正常的使用，可以在我们做出来的CPU上运行监控程序，监控程序A, R, D, U, G各项命令运行正常。在单步运行的时，通过VGA接口，显示屏幕上可以显示当前刚刚从指令寄存器取出的指令汇编代码（经过解码的），ps2小键盘运行也是正常的，可以通过小键盘的不同的按键选择不同的CPU运行的频率。系统整体通过了助教的检查验收，我们也拍摄了视频录制了系统正常运行的状态。

5.2 系统运行速度测试

我们小组的CPU运行主频是12.5MHz，在这个频率下，运行助教的提供的五段测试代码，运行的时间如下表：

测试代码	运行时间(s)
1	16.0
2	22.0
3	12.0
4	18.0
5	12.0

表 3: 测试代码运行时间

5.3 VGA测试

CPU单步运行的过程中VGA能够正常显示对应指令的英文字母。

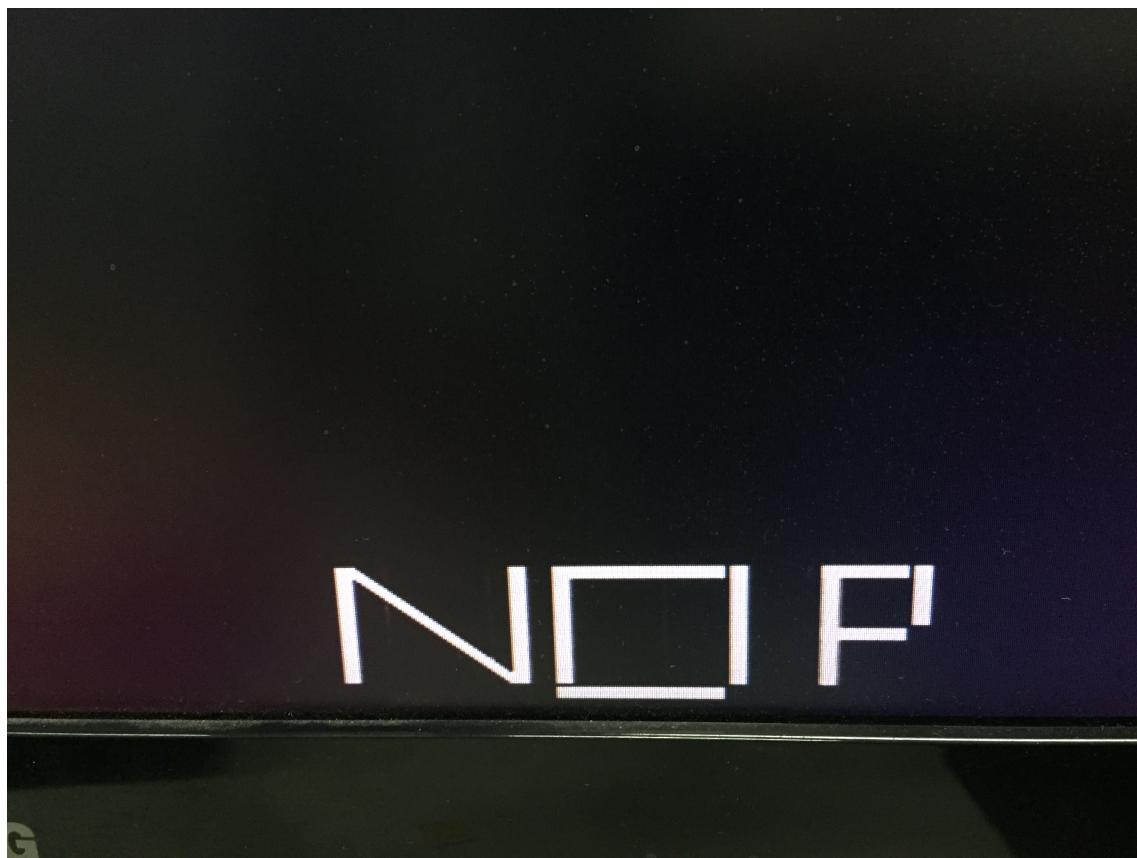


图 12: 单步英文字母显示

下图为指令所使用到的所有英文字母

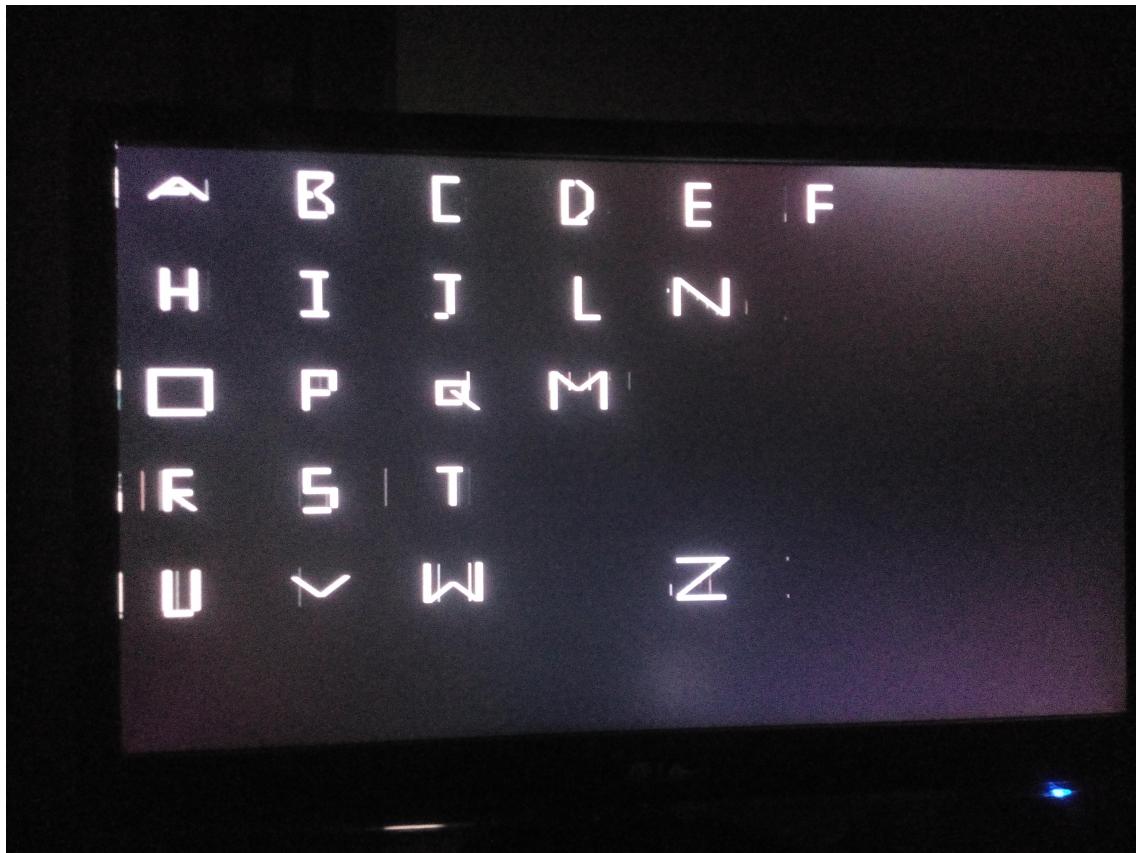


图 13: VGA 测试

5.4 系统编程测试

能够成功的运行监控程序，这就说明系统基本正确，但还有一些可能出现的冲突情况我们没有测试，另外还有监控程序中没有的扩展指令我们没有测试。

```

>> DD
[8000] 000c
[8001] 000d
[8002] 000d
[8003] 000d
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R2 D
[4001] LI R1 80
[4002] SLL R1 R1 0
[4003] SU R1 R2 0
[4004] LW R1 R3 0
[4005] ADDIU R3 1
[4006] SW R1 R3 1
[4007] SU R1 R3 2
[4008] SW R1 R3 3
[4009] JR R7
[400a] NOP
[400b]

>> G
running time : 0.015 s

>> D
[8000] 000d
[8001] 000e
[8002] 000e
[8003] 000e
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

```

图 14: load test

```

>> D
[8000] 000d
[8001] 000e
[8002] 000e
[8003] 000e
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R4 1
[4001] LI R2 A
[4002] LI R1 80
[4003] SLL R1 R1 0
[4004] SU R1 R4 0
[4005] CMPI R2 A
[4006] BTEQZ 3
[4007] ADDIU R4 1
[4008] NOP
[4009] NOP
[400a] SW R1 R4 1
[400b] JR R7
[400c] NOP
[400d]

>> G
running time : 0.015 s

>> D
[8000] 0001
[8001] 0002
[8002] 000e
[8003] 000e
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

```

图 15: forward delayslot test

左图是测试“load”后使用这个数据冲突的，这个数据冲突在流水线中必须要加nop作为气泡，[4004]的load指令的目标地址是R3，然后下一条指令就要读R3，这就会产生一个冲突，从[4006]开始三条指令就是将R3的值写入[8001]开始的三个地址，可见，程序正确处理了这个情况，写入的值是加1之后的新值。

左图是测试转发单元，和延迟槽的代码：若前一条指令的目标寄存器是后一条指令的源寄存器，就会产生数据冲突，必须使用转发，在[4002]和之后一条指令之间就存在这样的关系。延迟槽是分支和跳转指令后面的一条指令，无论分支是否执行，这条指令都要被执行，[4006]这条分支指令之后的一条指令就是延迟槽，虽然这里分支指令是被执行了，但是延迟槽中的指令照样要被执行，R4的值要被加1，所以我们看到[8001]和[8000]内存地址的结果就是不一样的，[8001]的结果被加了1，说明延迟槽是正常工作的。

```

>> D
[8000] 000f
[8001] 000f
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] b1c9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R2 FF
[4001] SLL R2 R2 0
[4002] LI R1 80
[4003] SLL R1 R1 0
[4004] SW R1 R2 0
[4005] NOT R2 R2
[4006] SW R1 R2 1
[4007] JR R7
[4008] NOP
[4009]

>> G
running time : 0.016 s

>> D
[8000] ff00
[8001] 00ff
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] b1c9
[8007] fe36
[8008] b743
[8009] 6b48

```

左图是测试我们302小组的第一条扩展指令NOT的，我们小组第一条扩展指令是NOT，测试代码中，将寄存器R2的值设置为0x00FF，然后取非，可以看到，结果变成了0xFF00，我分别将结果储存在了[8000]开头的内存中，这将是之后测试所将采用的模式。

```

>> D
[8000] ff00
[8001] 00ff
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] b1c9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R2 FF
[4001] LI R1 80
[4002] LI R3 4
[4003] SLL R1 R1 0
[4004] SW R1 R2 0
[4005] SLLU R3 R2
[4006] SW R1 R2 1
[4007] JR R7
[4008] NOP
[4009]

>> G
running time : 0.000 s

>> D
[8000] 00ff
[8001] 0ff0
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] b1c9
[8007] fe36
[8008] b743
[8009] 6b48

```

左图是第二条扩展指令SLLV，测试代码中，将寄存器R2的值设置为0x00FF，然后左移4位，结果变为0xFF0，我分别将结果储存在了[8000]开头的内存中。

图 16: not test

```

>> D
[8000] 00ff
[8001] 0fff
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R2 FF
[4001] LI R1 80
[4002] LI R3 4
[4003] SLL R1 R1 0
[4004] SW R1 R2 0
[4005] SR0U R3 R2
[4006] SW R1 R2 1
[4007] JR R7
[4008] NOP
[4009]

>> G
running time : 0.016 s

>> D
[8000] 00ff
[8001] 000f
[8002] 000f
[8003] 00aa
[8004] 00aa
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

```

左图是测试第三条扩展指令SRAV的，和上一条类似，将寄存器R2的值设置为0x00FF，然后右移4位，结果变为0x000F，我分别将结果储存在了[8000]开头的内存中。

```

>> D
[8000] 000b
[8001] 000d
[8002] 000d
[8003] 000d
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

>> A
[4000] LI R3 C
[4001] LI R2 B
[4002] LI R4 A
[4003] LI R1 80
[4004] SLL R1 R1 0
[4005] SW R1 R2 0
[4006] SLT R2 R4
[4007] BTEQZ 3
[4008] NOP
[4009]
[400a] SW R1 R3 3
[400b] SW R1 R3 4
[400c] SW R1 R3 1
[400d] SW R1 R3 2
[400e] JR R7
[400f] NOP
[4010]

>> G
running time : 0.015 s

>> D
[8000] 000b
[8001] 000c
[8002] 000c
[8003] 000d
[8004] 000d
[8005] 1a06
[8006] bic9
[8007] fe36
[8008] b743
[8009] 6b48

```

类似的，左图是测试扩展指令SLT的，这条指令也是一条根据不同的条件而改变标志寄存器的值。在[4006]的SLT指令比较了两个寄存器R2和R4的值，然后置标志寄存器为0，下一条[4007]的跳转指令就会产生跳转，跳转到的位置是[400b]，跳过了[4009] 和[400a]的两条将内存修改成c的指令，结果只是将[8001]和[8002]两个内存地址修改成了c，而不是不发生跳转的4个。

图 19: slt test

6 实验总结与体会

6.1 实验心得概述

本次计算机组成原理大实验已经结束。回顾这次实验，我们能很清晰地看到本组成员的心路历程，能明显地感受到成员们的成长。

从还没有选修计算机组成原理一课的时候，听到学长学姐描述“三星期造计算机”这个大作业非常恐怖，需要三星期每天熬夜到很晚；到真正上了计算机组成原理，在平时的课堂内容都还没来得及完全消化的时候就要进行大实验了的时候的无奈；再到要分组的时候，满脑子想着怎么找一个大腿；再到真正开始进行大实验，团队开始分工、磨合甚至产生许多摩擦与冲突；但是最后的最后，我们还是在规定时间内把计算机造出来了，并且在基本要之外完成了自我设定的各种扩展功能。

回头看看，一路辛酸却收货满满，非常感谢计算机组成原理一课给我们这样的机会挑战我们的极限。

我们一致认为，在开始代码工作之前的前期准备工作是实验中最重要的环节，因此我们花了大量的时间在前期。

设计数据通路和控制信号是决定实验能否完成和能否快速完成的关键部分。

数据通路的设计是决定后续计划的关键；控制信号的设计影响到不同模块之间的通信，简单、有效、清楚的设计对代码的编写有很大的作用；时序设计也是实验中非常重要的一个部分，在最后的debug时间段，我们主要的错误就出在时序部分，所以时序设计对于完成实验的重要性也是毋庸置疑的。我们以相应数据结构为标准，结合课上的知识，绘制数据通路和控制信号图。也就数据通路和控制信号图与老师、助教和学长进行过多次交流，进行了多次的修改与完善，确保了最终的设计的正确性和合理性；

同时，我们也早早明确了每个人的分工，使得大家能各自高效地完成自己的部分，同时确保能快速将各自代码整合起来；

在听取了同学、助教、老师的意见，权衡了多方利弊之后，我们选择了大部分同学没有选择的Verilog语言作为本次大实验的编程语言。

6.2 合理分工

由于本次试验任务非常繁重，不论是前期的设计，中期的编程，后期的debug都需要很多的精力和时间，所以我们进行了非常合理的分工。

实验前期的数据通路以及控制信号的设计由大家共同讨论完成，保证每个人对于整个CPU有一个大体框架认知，之后主要由一名同学负责搭建CPU的总体框架，主要包括根据数据流图指定的各个模块，定义好其输入输出端口，同时搭建好CPU顶层模块，完成各部分的连线工作。有了这个整体框架后，由另一名同学具体实现大部分的模块。对于Ram这个逻辑比较复杂的模块则是在后期一起讨论实现的。

由另一个人来搭建顶层框架，另两个同学实现具体模块的功能，这样一方面能保证设计的完整性，同时也利于两名同学互相分工合作。此外，试验中的一个经验教训就是变量名的定义一定要统一，我们最后解决的一个bug就是因为变量名命名错误导致的，所以良好的变量命名是一门学问。

外设部分由于和基本模块能分开实现，所以我们安排一名同学专门负责外设的设计，调试与实现。实验最后的结果也证实了我们的想法是正确的，一名同学单独完成了外设的代码部分后，只要进行比较简单的处理就能整合进基本内容。

6.3 高效工具的使用

6.3.1 绘制电路图

我们刚开始尝试了Matlab自带的Simulink仿真软件实现电路图的绘制。后来我们逐渐发现，Simulink强在能进行电路仿真，但是在很多情况下都无法满足我们对绘图定制型较强的需求。所以我们之后选择图形绘制工具OmniGraffle 进行各种图形的绘制，这个工具满足了我们的需求。

6.3.2 代码管理工具

我们使用Github进行代码管理，通过这个分布式的版本控制系统，能很好整合全组代码，协同合作，并很容易回到之前的代码版本。

6.4 Verilog使用心得

从一开始我们就选用Verilog语言进行程序的编写。Verilog是一种更适合工业界的语言，所以在格式方面的要求会比VHDL要轻松一些，所以更利于我们编写代码。并且，Verilog的语法等方面与C语言比较相似，使得我们很好使用以前学过的一些编程思想来编

写我们的程序，更重要的是，在我们编写的时候，发现Verilog拥有非凡的性能和可扩展能力。

由于Verilog与C语言有比较大的相似性，于是Debug的过程不会是那么痛苦的一个过程，使得我们的查错过程比较轻松。

但是由于其他组基本上都是使用的VHDL，所以在语法检查方面我们组也不能请教其他的组，只能自己探索，找出程序的问题所在，并解决问题。

6.5 外设调试心得

6.5.1 VGA调试心得

VGA调试在本次试验中可以说是占掉了很多时间。因为我们要实现实时跳转指令功能，那么首先，我们必须把所有字母在VGA显示器上显示出来。

本来我们准备使用window自带的点阵字体进行表示。但是当我们下载了点阵字体的文件之后，我们只能以二进制文件的形式打开文件，但是我们无法把二进制字母成功映射成为我们需要的那些字母，在对点阵字体文件多次探索无效之后，我们决定自己来把我们需要的字母写出来。

但是把自己需要的所有字母写出来也不是一件非常容易的事情。如果是在表示字体中，用到非常平滑的弯折过度，比如B，那些有弧度的地方就十分难表示，我们需要计算每一个需要点亮的像素在什么时候应该被显示出来。在有限的时间内完成这项工作是不是非常现实的一件事情。

经过小组讨论之后，我们决定，把所有字母的绘制抽象，使得所有的字母都能用水平、竖直直线和斜率为0.25以及斜率为0.75的直线表示出来。这样的抽象使得把所有字母表示出来的复杂度在我们的可控范围内。

在绘制完所需要的字母后，我们按照显示屏显示的原理构建了VGA的扫描和绘图模块。最后将这些部分整合起来，实现了所需要的功能。

调试VGA的最大的心得就是，要有很好的耐心去调试。因为当基本框架搭建起来之后，VGA部分的代码书写就变成了体力活。虽然变成了体力活，但是我们VGA部分3000多行的代码，在调试问题的时候也是非常不容易的，这个时候就很需要耐心。好在是，当调试的同学失去耐心的时候，同组的同学们会立马鼓励，和同学一起度过这样的难关。

6.5.2 PS/2键盘调试心得

在PS/2接口键盘调试中最大的心得就是对PS/2协议的理解和ChipScope板上调试工具的使用时非常重要的。在理解了PS/2协议的基本原理之后才能设计好状态机加以实现。当然对于类似于PS/2这样的独立外设模块的调试，做模拟实际上是没有必要的，很多时候我们需要Xilinx软件提供的ChipScope板上调试工具。在正确编译完成并烧入FPGA之后，当我们按下键盘上的某一个按键，ChipScope工具可以实时采样ps2_clk,ps2_data等信号，辅助我们发现代码中出现的问题。

6.5.3 串口Ram调试心得

AllInOneRamController模块是本次实验中时序最复杂，最后完成调整的一个模块。当监控程序“OK”跑起来，R, D指令正确执行但是A, U指令不正常，后来发现我们必须解决指令存储器的写才能完成监控程序最后的A和U的功能。而按照我们之前的设计，我们对于Ram2内存只能读（读预先使用flashandram.bit文件烧进去的监控程序）不能写监控程序动态执行过程中的用户程序，这最后一处时序的设计困扰了我们很多天，我们请教了很多1子班的学长和我们同年级的同学，发现我们同年级的同学大部分使用了停掉流水线的方式进行指令存储器的写操作，在这样一个想法的直下，我们成功实现了A和U的指令。

调试串口和Ram的最大心得是对于复杂时序的设计是不能靠一遍一遍写代码试就可以试出来的，必须在前期进行充分的设计和鲁棒性考察。自己首先有一套设计方案，然后可以进行小组之间的思路交流和请教有经验的学长指点，确定自己最终使用的设计方案并加以实现。

6.6 汇编程序调试心得

为了确保我们代码的正确性，在调通监控程序可以正常通过A, G指令执行代码之后，我们小组编写了一些汇编程序对程序进行了测试，测试程序在testingFiles中，我们不但测试了我们扩展指令的正确性，我们还测试了我们有没有正确的处理延迟槽，转发等。我们感觉，这样的测试是十分必要的，在测试中我们发现了一些问题，比如说原来我们没有正实现算数右移的指令，因为在verilog语言中，只是使用>>>还不能实现算数右移，还需要我们将寄存器改为signed类型的才可以。通过测试，我们还增强了对我们自己设计的

系统的信心，发现可以在我们设计的系统上跑写出来的几乎所有的程序的时候，当时心情是十分的兴奋。

6.7 Debug的毅力

在本次实验中，我们发现，相比于其他课来说，平心而论，本课的代码量不算很多，但是对于硬件编程来说，debug的问题是比较棘手的，原因有以下几点：

1. 代码逻辑很复杂，导致了每次编译时间都在数分钟左右，眼中影响到了debug效率，常常一个多小时也找不到bug所在；
2. 代码的问题往往原因非常多，比如时序、信号等等，有可能不是代码编写的问题，而是设计的问题
3. 有一些比较奇葩的错误，有的时候根本找不到原因，最后也以奇怪的方法解决了。
4. 由于硬件设计语言的特殊性，有时候没办法进行人脑的模拟执行。

因此我们需要很大决心和毅力去debug，这样才能完成实验。

6.8 良好的沟通交流非常重要

三人一组完成实验其实不是一个简单的过程，前期需要良好的分工，期间需要大家不断配合，不断沟通，一起解决问题，组员遇到困难的时候还得组员之间相互体谅。当我们遇到组内没法解决的问题，我们会主动向其他组的同学询问，沟通交流，讨论解决办法，这种方式让我们能非常迅速地解决遇到的很多问题。

此外，助教和老师的帮助对于我们来说也是非常重要，我们曾多次与助教和老师讨论，得到了助教和老师的帮助，在最后检查的时候，助教老师也是非常认真地耐心等着我们解决偶然碰到的问题，并给出建议，在我们讨论后解决了问题。

图片列表

1	数据通路图	5
2	寄存器堆模块	9
3	符号扩展器模块	10
4	解码器模块	11
5	ALU模块	12
6	RAM模块	13
7	freeze控制寄存器	14
8	冒险检测单元	15
9	转发单元	16
10	键盘控制模块	19
11	整体运行测试	20
12	单步英文字母显示	21
13	VGA测试	22
14	load test	23
15	forward delayslot test	23
16	not test	24
17	sllv test	24
18	not test	25
19	slt test	25

图表列表

1	ADDIU指令控制信号	8
2	VGA模块及功能说明	17
3	测试代码运行时间	21