中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2018K8009929046　　姓名：何咏哲　　专业：计算机科学与技术

实验序号：2　　实验名称：单周期处理器设计

注 1：请在实验项目个人本地仓库中创建顶层目录 doc。撰写此 Word 格式实验报告后以 PDF 格式保存在 doc 目录下。文件命名规则：学号-prjN.pdf，其中学号中的字母“K”为大写，“-”为英文连字符，“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：2018K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：学号-prj5-projectname.pdf，例如：2018K8009929000-prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将 doc 目录下的实验报告 PDF 文件添加到本地仓库，并通过 git push 推送提交。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

```
//normal
`define CALCU_I 3'b001 //2*GPR + IMM/OFF
`define BRANCH  4'b0001 //2*GPR + OFF
`define LOAD    3'b100 //2*GPR + OFF
`define STORE   3'b101 //2*GPR + OFF
`define LUI     6'b001111 //1*GPR + IM
`define BLEZ    6'b000110 //1*GPR + OFF
`define JUMP    5'b00001 //instr_index

//special
`define SPECIAL 6'b000000
`define CALCU   3'b100 //3*GPR_H
`define JUMP_R  5'b00100 //2*GPR + hint
`define SLT_X   5'b10101 //3*GPR_H
`define SHIFT   4'b0000 //3*GPR_L
`define SHIFT_V 4'b0001 //3*GPR_H
`define SHIFT_ALL 3'b000
`define MOVE    5'b00101
//regimm
`define REGIMM   6'b000001 //1*GPR + OFF
`define BRANCH_Z 4'b0000  //only 5 bits

`define SHIFT_SIGN  5'b00000
`define JAL 6'b000011
`define JALR 6'b001001//special
```

```
`define J     6'b000010
`define JAL   6'b000011
`define JR    6'b001000
`define JALR  6'b001001
`define BNE   6'b000101
`define BEQ   6'b000100
`define BGEZ  5'b00001
`define BLTZ  5'b00000
`define MOVN  6'b001011
`define MOVZ  6'b001010
`define SB_s  3'b000//the sh
`define SH_s  3'b001
`define SW_s  3'b011
`define SWL_s 3'b010
`define SWR_s 3'b110
`define SB    6'b101000
`define SH    6'b101001
`define SW    6'b101011
`define SWL   6'b101010
`define SWR   6'b101110
```

如图所示，将同一类操作的共同之处采用宏定义的方法表述出来，增强了代码的可读性。比如 LOAD 类型指令的高 3 位都是 100，STORE 类型指令的高 3 位都是 101。而另一些需要进行特判的指令的宏定义也单独列在了下面。

```
always @(posedge clk) begin
    if(rst) begin
        PC = 32'b0;
    end else begin
        PC = Jump ? JumpAddress:((Branch&Branch_eff)?branch_PC:PC_4);
    end
end
```

PC 的赋值采用了时序逻辑，在每个时钟周期的上升沿进行判断，如果复位信号有效就将 PC 复位为 0，否则根据读取的指令对 PC 进行操作，如果是跳转分支类指令，就将 PC 赋为相应的值，否则给输出 PC+4 的值。

```
alu res_cal(
    .A(shift_num),
    .B(ALU_N2),
    .ALUop(ALU_control),
    .Result(ALU_result),
    .Overflow(),
    .CarryOut(),
    .Zero(Zero)
);
```

```
reg_file ren_data_get(
    .clk(clk),
    .rst(rst),
    .waddr(RF_waddr),
    .raddr1(RF_raddr1),
    .raddr2(RF_raddr2),
    .wen(RF_wen),
    .wdata(RF_wdata),
    .rdata1(RF_rdata1),
    .rdata2(RF_rdata2)
);
```

实例化前一个实验中写过的 ALU 和 REG_FILE，来进行相应的存储与计算。

```
assign rs = Instruction[25:21];
assign rt = Instruction[20:16];
assign rd = Instruction[15:11];
assign opcode = Instruction[31:26];
assign func = Instruction[5:0];
assign RegDst = (opcode[5:3]==`CALCU_I||opcode[5:3]==`LOAD)?0:1;
assign Jump = (opcode[5:1]==`JUMP || (opcode==`SPECIAL&&func[5:1]==`JUMP_R)) ?1:0;
assign Branch = (opcode[5:2]==`BRANCH || (opcode==`REGIMM && rt[4:1]==`BRANCH_Z))?1:0;
assign MemRead = (opcode[5:3]==`LOAD)?1:0;
assign MemWrite = (opcode[5:3]==`STORE)?1:0;
assign MemtoReg = (opcode[5:3]==`LOAD)?1:0;
assign RF_wen = (opcode[5:3]==`STORE || opcode==`J || opcode[5:2]==`BRANCH || (opcode==`REGIMM && rt[4:1]==`BRANCH_Z) || (opcode==`SPECIAL && (func==`JR||func==`MOVZ&&(!Zero)||
  func==`MOVN&&Zero)))?0:1;
```

使用模块化的方法，通过译码得到一系列控制信号，并且使用这些控制信号来引导电路结构。

```
assign byte = ALU_result[1:0];
assign load_byte_data = (byte[1]&byte[0])?Read_data[31:24]:
                        ((byte[1]&!byte[0])?Read_data[23:16]:
                        ((!byte[1]&byte[0])?Read_data[15:8]:Read_data[7:0]));
assign load_half_data = (!byte[1]&!byte[0])?Read_data[15:0]:Read_data[31:16];
assign lwl_data = (byte[1]&byte[0])?Read_data[31:0]:
                  ((byte[1]&!byte[0])?{Read_data[23:0],RF_rdata2[7:0]}:
                  ((!byte[1]&byte[0])?{Read_data[15:0],RF_rdata2[15:0]}:
                  {Read_data[7:0],RF_rdata2[23:0]}));
assign lwr_data = (!byte[1]&!byte[0])?Read_data[31:0]:
                  ((!byte[1]&byte[0])?{RF_rdata2[31:24],Read_data[31:8]}:
                  ((byte[1]&!byte[0])?{RF_rdata2[31:16],Read_data[31:16]}:
                  {RF_rdata2[31:8],Read_data[31:24]}));
```

LOAD 类型指令的数据选择，根据指令手册的图示以及小字部分实现。

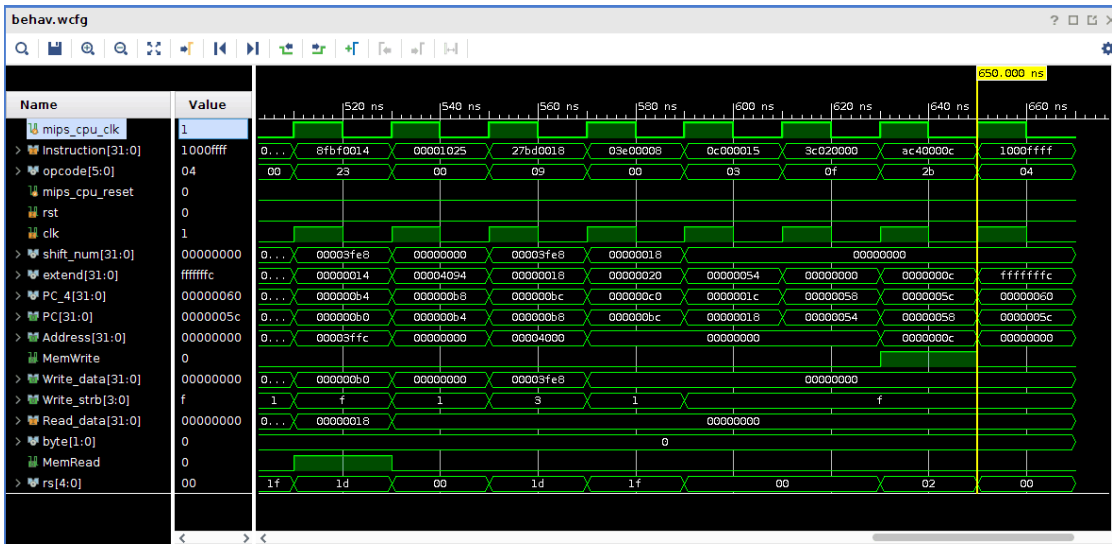
```
assign lui_extend = {Instruction[15:0],16'b0};
assign zero_extend = {16'b0,Instruction[15:0]};
assign sign_extend = Instruction[15] ? {{16{1'b1}},Instruction[15:0]} : {{16{1'b0}},Instruction[15:0]};
assign shift_sign_extend = Instruction[15] ? {{14{1'b1}},Instruction[15:0],2'b00} : {{14{1'b0}},Instruction[15:0],2'b00};
assign sltiu_extend = Instruction[15] ? {{1'b0},{15{1'b1}},Instruction[15:0]} : {{16{1'b0}},Instruction[15:0]};
assign extend = (opcode==`ANDI||opcode==`ORI||opcode==`XORI)?zero_extend:
                ( (opcode==`ADDIU||opcode==`SLTI||opcode[5:3]==`LOAD||opcode[5:3]==`STORE)?sign_extend:
                ( (opcode==`SLTIU)?sltiu_extend:
                ( (opcode==`LUI)?lui_extend : shift_sign_extend)));
```

指令手册中出现的各种扩展信号，先分别扩展，再根据具体指令来进行选择。

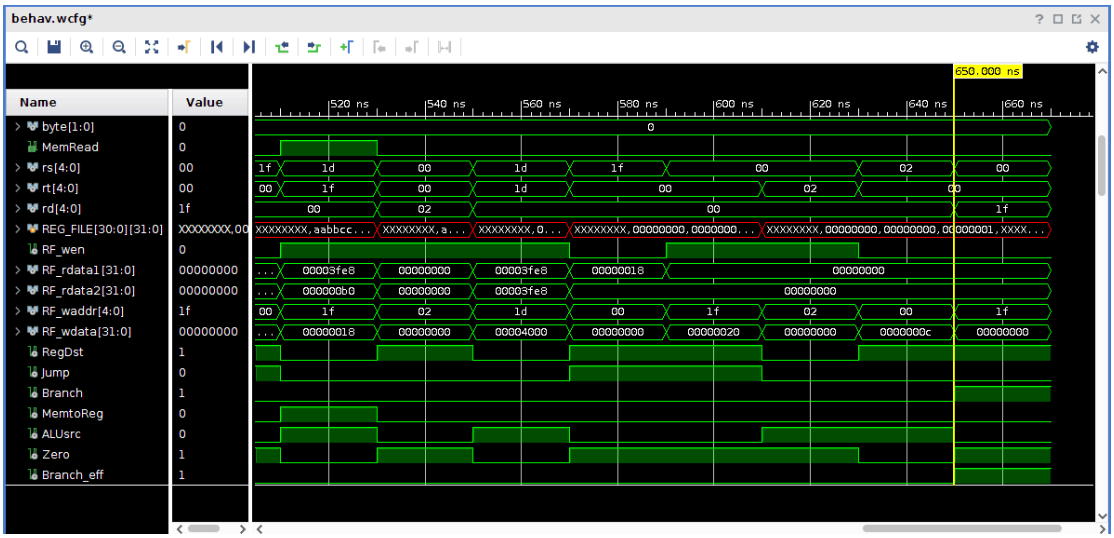
```
assign Address = {ALU_result[31:2],2'b00};
assign ALUsrc = (opcode[5:3]==`CALCU_I || opcode[5:3]==`LOAD || opcode[5:3]==`STORE)?1:0;
assign sb_strb = 4'b1000 >> (~ALU_result[1:0]);
assign sh_strb = {ALU_result[1],ALU_result[1],!ALU_result[1],!ALU_result[1]};
assign sw_strb = 4'b1111;
assign swl_strb = {ALU_result[1]&ALU_result[0],ALU_result[1],ALU_result[1]|ALU_result[0],1'b1};
assign swr_strb = {1'b1,!(ALU_result[1]&ALU_result[0]),!ALU_result[1],(!ALU_result[1])&(!ALU_result[0])};
assign Write_strb = opcode[2:0]==`SB_s?sb_strb:
                    ( opcode[2:0]==`SH_s?sh_strb:
                    ( opcode[2:0]==`SW_s?sw_strb:
                    ( opcode[2:0]==`SWL_s?swl_strb:swr_strb)));
```

```
assign Write_data = (opcode==`SB)? sb_result :
                    ((opcode==`SH)? sh_result :
                    ((opcode==`SWL)? swl_result :
                    ((opcode==`SWR)? swr_result: RF_rdata2)));
assign sb_result = {32{store_byte[0]}} & {{24{1'b0}},RF_rdata2[7:0]} |
                   {32{store_byte[1]}} & {{16{1'b0}},RF_rdata2[7:0],{8{1'b0}}} |
                   {32{store_byte[2]}} & {{8{1'b0}},RF_rdata2[7:0],{16{1'b0}}} |
                   {32{store_byte[3]}} & {RF_rdata2[7:0],{24{1'b0}}};
assign sh_result = {32{store_byte[0]}} & {{16{1'b0}},RF_rdata2[15:0]} |
                   {32{store_byte[2]}} & {RF_rdata2[15:0],{16{1'b0}}};
assign swl_result = {32{store_byte[0]}} & {{24{1'b0}},RF_rdata2[31:24]} |
                    {32{store_byte[1]}} & {{16{1'b0}},RF_rdata2[31:16]} |
                    {32{store_byte[2]}} & {{8{1'b0}},RF_rdata2[31:8]};
assign swr_result = {32{store_byte[1]}} & {RF_rdata2[23:0],{8{1'b0}}} |
                    {32{store_byte[2]}} & {RF_rdata2[15:0],{16{1'b0}}} |
                    {32{store_byte[3]}} & {RF_rdata2[7:0],{24{1'b0}}};
```

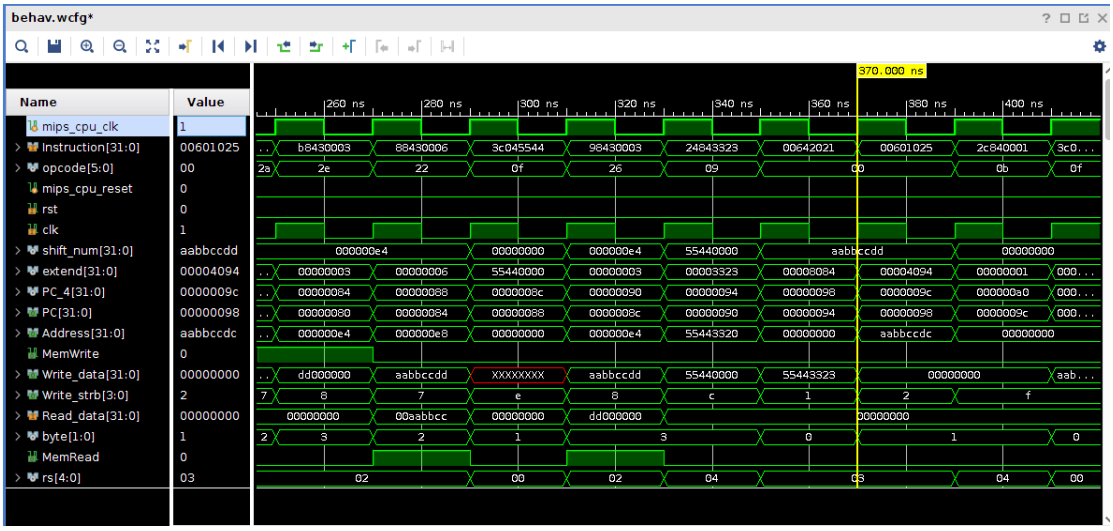
strb 信号的译码以及 STORE 类型指令的数据选择，根据指令手册实现，面向 bug 修改完善，体现了按字节对其的思想。



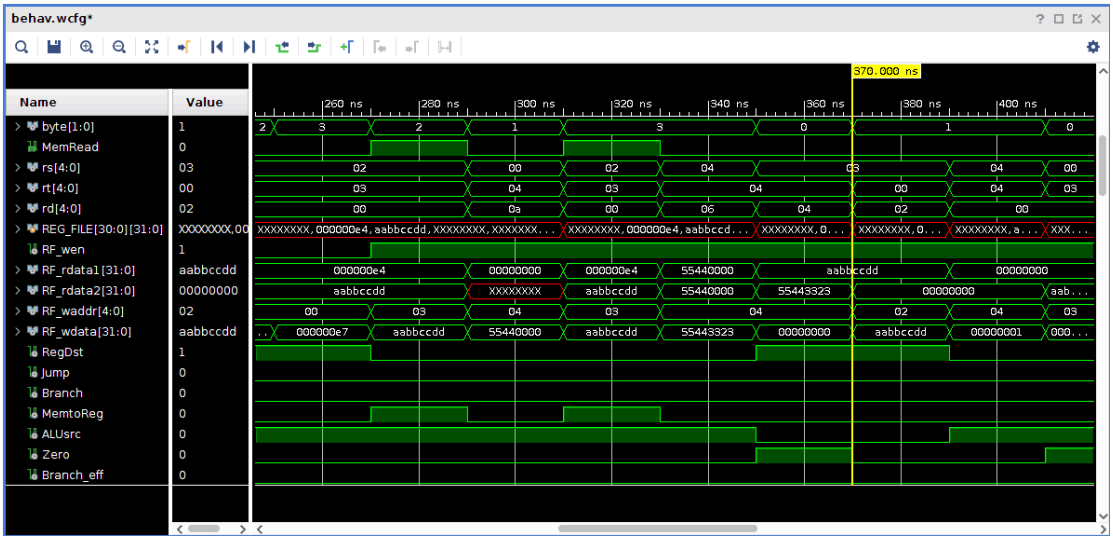
该图为运行 advanced17 仿真结果的波形图，可以看到，在 650ns 处读取到的 Instruction 为 0x1000ffff,转化为二进制得到 1 0000 0000 0000 1111 1111 1111 1111，通过查阅指令手册可以知道这条指令对应的是 BEQ 指令。按照要求，我们需要比较 rs 与 rt 两个通用寄存器的值，若相等则对 PC 进行跳转，转移地址由 offset 后接 00 再进行有符号扩展得到。



此时 rs 和 rt 寄存器的地址都为 0，读出来的数据 RF_data1 和 RF_data2 也均为 0，相等，满足跳转的条件。offset 的值为 16’b1111_1111_1111_1111，经过扩展后的值应该为 32’b1111_1111_1111_1111_1111_1111_1111_1100，从前一张图可以发现 extend 的值正好为 fffffffc，与预期相等，并且此值成功地赋予了 PC。



370ns 时指令为 0x00601025，转化为二进制为 0000 0000 0110 0000 0001 0000 0010 0101，高 6 位为全 0，应该为 SPECIAL 类型，末 6 位为 100101，查阅指令手册得知应该为 OR 操作。此时应该将 rs 与 rt 两个通用寄存器中的值进行按位或，将结果付给 rd 寄存器。



容易验证操作的正确性。此时 PC 应+4。

其余波形图同理。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真、本地上板及云平台调试过程中的难点等）

1、**问题：**一开始没有真正理解“按指令格式和指令类型设计处理器译码逻辑”这句话的意思，没有去寻找不同指令之间的相同之处，采用了单独译码的方式，将 44 条指令全部通过宏定义写在了一个 macro 文件里面。

解决过程：在把 44 条指令的操作码都列举出来了之后，发现同一类型的操作具有某些共同之处。在思考了是采用卡诺图化简还是采用宏定义的方式来表示这种异同之后，我选择了使用宏定义，因为同一类指令之间的相似度很高，化简结果与宏定义相差不大，而采用宏定义的代码可读性更强。

2、**问题：**不清楚 PC 的值什么时候应该+4，什么时候应该跳转，以及 BEQ、BNE 之类跳转的条件错误。

解决过程：在 debug 的过程中逐步仔细翻阅指令手册，通过比较各个信号的预期值与实际值来定位错误。以及 PC+4 应该是一个确定的操作，最初设计时让 PC 应该+4 时再+4，以为这样可以减少操作，但是这是非常“C 语言”的想法，只要被设计出来，电路就是物理存在的，无论它有没有被使用，增加一个用于判断是否+4 的选择信号非但不能减少操作，反而增加了一个选通信号，造成额外的浪费。

3、**问题：**Address 信号的对齐。

解决过程：Address 的访问应该是按自己对齐的，故低 2 位应该为 0。该问题在查阅手册后得到解决。

4、**问题：**write_strb 信号的赋值。

解决过程：在过 basic 和 medium 组的时候对 strb 信号要求不严格，当时按照自己的猜想随意口胡了一个上去，过了几个测试。后来在 advanced 组的仿真过程中频繁发现 strb 信号的报错，开始思考这个信号的真正含义，于是在一番查阅资料与请教同学之后得到解决，实际上也是对齐的思想。

5、**问题：**访存类指令的数据选择。

解决过程：部分信号如 SWL,SWR,LWL,LWR 等通过研究指令手册上的图示弄明

白了，另一部分信号如 LB,LH 等通过对指令手册上的小字部分进行猜测弄明白了，比如 LB 给的操作是$vAddr_{1\ldots0}\ xor\ BigEndianCPU^2$，我们的实验是小端序的，故$BigEndianCPU^2$应该为 00，即 LB 的 byte 选择应该由地址的低 2 位与 00 的亦或结果来决定。同理可以得到其他类似操作的选通信号。

6、**问题：**移位与各种扩展，开始以为只有有符号扩展与无符号扩展两种，后来发现了很多例外，以及纠结是直接在 CPU 中实现扩展还是通过例化 ALU 来实现扩展。

解决过程：有很多不同的移位与扩展，如果都使用 ALU 来完成的话势必会造成不必要的硬件浪费，因此我对于不涉及其他运算的扩展直接在 CPU 的代码里面实现了，大致上包括零扩展、有符号扩展、无符号扩展、先移位再有符号扩展、特殊的 LUI 扩展和 SLTIU 扩展。

7、**问题：**助教老师在验收时指出，我的 STORE 操作判断是对 6 位操作码进行判断的，会造成硬件的浪费。

解决过程：比对后发现，STORE 指令前三位都是相同的，真正彰显不同的是后三位，因此无须对前三位进行重复比较。在修改的过程中我又产生了一个新的问题：如果不进行全 6 位的比较，而恰好有其他操作码的后三位与 store 的相同时，会不会出现错误呢？分析之后我的答案是对于 write_strb 信号不会，因为该信号仅仅在 STORE 类型的指令时才有效，只需要保证在此时计算正确即可；而对于 write_data 信号而言，其结果不仅依赖于 STORE 类型，也依赖于从寄存器中读取的数，而有的操作前三位与 STORE 类型的共同部分一样，会造成混淆，故对于 write_data 信号我并未做出改变，而仅仅优化了 write_strb 信号。

三、 对讲义中思考题（如有）的理解和回答

四、 在课后，你花费了大约____15____小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

现在后头看，实验本身难度其实并不算大，但是由于之前从未接触过这一类型的设计，再加上另一个班的理论课进度较慢，一开始做实验的时候非常迷茫，不知道要从哪里入手。在自己提前预习了讲义之后，与刘听雨同学进行了一次通话，向他讲述了自己的思路，而他也根据我的初步理解给予了肯定与指正，在此特别感谢刘听雨同学在整个实验二期间对我的莫大帮助。

另外我在 write_strb 信号以及访存类数据的选择这一部分折腾了很久，开始是不知道还需要自己选择数据，所以对于报错很摸不着头脑。在知道了需要自行设计的操作之后，遇到的另一个困难是实现的具体细节，包括但不限于 SB,SH,LB,LH 等指令。已知实验是小端序的前提下还是很容易根据指令手册的图示实现诸如 SWL,SWR,LWL,LWR 等指令，但是其余的访存类指令并没有给出清楚的图示，小字部分看不大懂，网上的博客也大多没有解释我疑惑的点，因此在这一部分只能一边面向 bug 编程，一边向同学请教。**而在我完成实验之后对其他同学的答疑过程中，我发现他们这一部分也和我存在着类似的问题，即不知道指令到底需要我们干什么，希望老师们在后续实验里能够在这部分给予同学们更多**

的帮助和提示。在这里也需要感谢**田郑书媛**与**王嵩岳**两位同学在访存指令部分对我的帮助。