多周期处理器大作业实验报告

无 91 2019010999 杨坤泽

1. 数据通路设计

1.1

数据通路部分主要代码如下：

```
1.     Controller Controller_1(reset,clk,opcode,func,PCWrite,PCWrite
   Cond,IorD,MemWrite,MemRead,IRWrite,MemtoReg,RegDst,RegWrite,ExtOp
   ,LuiOp,ALUSrcA,ALUSrcB,ALUOp,PCSource);
2.     PC PC_1(reset,clk,signal,PC_in,PC);
3.     InstAndDataMemory I_1(reset,clk,address,B_o,MemRead,MemWrite,
   Mem_data);
4.     InstReg I_2(reset, clk, IRWrite, Mem_data,opcode, rs, rt, rd,
    shamt, func);
5.     RegisterFile R_1(reset, clk, RegWrite, rs, rt, Write_register
   , Write_data, Read_data1, Read_data2);
6.     ImmProcess I_3(ExtOp, LuiOp, {rs,rt,rd,shamt,func}, ImmExtOut
   , ImmExtShift);
7.     ALUControl A_1(ALUOp, func, ALUConf, Sign);
8.     ALU A_2(ALUConf, Sign, In1, In2, Zero, Result);
```

数据通路部分主要是将各个模块进行连接，保证多周期处理器的数据传输准确无误。需要注意的是变量在声明的时候要保证比特数和模块中的一致。

数据通路中主要包含的寄存器代码如下：

```
1.     RegTemp MDR(reset,clk,Mem_data,MDR_o);
2.     RegTemp A(reset,clk,Read_data1,A_o);
3.     RegTemp B(reset,clk,Read_data2,B_o);
4.     RegTemp ALUOut(reset,clk,Result,ALUOut_o);
```

寄存器的作用主要是将存储数据并将数据在下一个周期中使用出来。MDR寄存器主要存储从存储器里读出的数据，可能会在lw指令中用到，将某一地址中取出的数据存入，在后续的$RegisterFile$模块中将数据存入寄存器中；A寄存器主要存储从$RegisterFile$模块中读取出的Rs寄存器中的数据；B寄存器主要存储从$RegisterFile$模块中读取出的Rs寄存器中的数据；$ALUOut$寄存器是存储ALU模块的运算结果。

其实PC模块也是一个寄存器，主要存储可能用于跳转的指令地址与正常的指令地址；$InstReg$为指令寄存器，用于输入指令并进行指令的$decode$。

多路选择器代码如下：

```
1.     assign address=IorD?ALUOut_o:PC;
2.     assign signal=(PCWriteCond&&Zero)||PCWrite;
```

```
    assign Write_register=RegDst?rd:rt;
    assign Write_data=MemtoReg?MDR_o:ALUOut_o;
    assign In1=(ALUSrcA==2'b00)?A_o:(ALUSrcA==2'b01)?PC:(ALUSrcA==2'b11)?shamt:32'b0;
    assign In2=(ALUSrcB==2'b00)?B_o:(ALUSrcB==2'b01)?32'd4:(ALUSrcB==2'b11)?ImmExtShift:(ALUSrcB==2'b10)?ImmExtOut:32'b0;
    assign PC_in=(PCSource==2'b00)?Result:(PCSource==2'b01)?ALUOut_o:(PCSource==2'b10)?{PC[31:28],{rs,rt,rd,shamt,func}<<2}:(PCSource==2'b11)?A_o:32'b0;
```

$address$信号为$InstAndDataMemory$模块接收到的地址信号，由$IorD$控制信号决定多路选择器输出$ALUOut_o$还是PC；

$signal$不是多路选择器，而是判断PC是否写入的组合逻辑，由$PCWrite$、$Zero$和$PCWriteCond$信号共同构成；

$Write_Register$是写入寄存器号，由$RegDst$控制信号决定写入rd中还是写入rt中，因为有一些指令中没有rd寄存器号；

$Write_data$是写入寄存器的数据，由$MemtoReg$控制信号决定输入MDR_o还是$ALUOut_o$中的数据；

$In1$和$In2$是ALU的输入数据，前者由$ALUSrcA$决定输入A_o、PC还是用于偏移指令的偏移量$shamt$，后者由$ALUSrcB$决定输入B_o、4、扩展后的立即数还是扩展偏移后的立即数；

PC_in是PC的输入数据，由$PCSource$控制信号决定输入ALU的计算结果$Result$、$ALUOut\_o$、可能用于jr指令的rs还是j指令跳转的地址。

2. 控制信号分析与有限状态机实现

2.1

$PCWrite$：PC寄存器的写使能信号，1代表写入，0代表不写入；

$PCWriteCond$：分支指令时会用到的防止因ALU计算产生两个相同的数而认为指令是分支指令的辨别信号，1代表是beq指令，0代表不是；

$IorD$：决定存储器的读取地址，1代表读入ALU计算结果，0代表读入PC；

$MemWrite$：存储器的写使能信号，1代表写入，0代表不写入；

$MemRead$：存储器的读使能信号，1代表读取，0代表不读取；

$IRWrite$：指令存储器的写使能信号，1代表写入，0代表不写入；

$MemtoReg$：寄存器堆写入数据多路选择器的决定信号，1代表写入数据寄存器MDR的输出，0代表写入ALU计算寄存器的输出；

$RegDst$：寄存器堆写入寄存器的决定信号，1代表写入rd，0代表写入rt；

$RegWrite$：寄存器堆的写使能信号，1代表写入，0代表不写入；

$ExtOp$：立即数拓展方式的决定信号，1代表有符号拓展，0代表无符号拓展；

$LuiOp$：是否进行立即数左移16位的操作，1代表是，0代表不是；

$ALUSrcA$：ALU输入数据的决定信号，00代表输入rs，01代表输入PC，11代表输入偏移指令的偏移量；

$ALUSrcB$：ALU输入数据的决定信号，00代表输入rt，01代表输入4，11

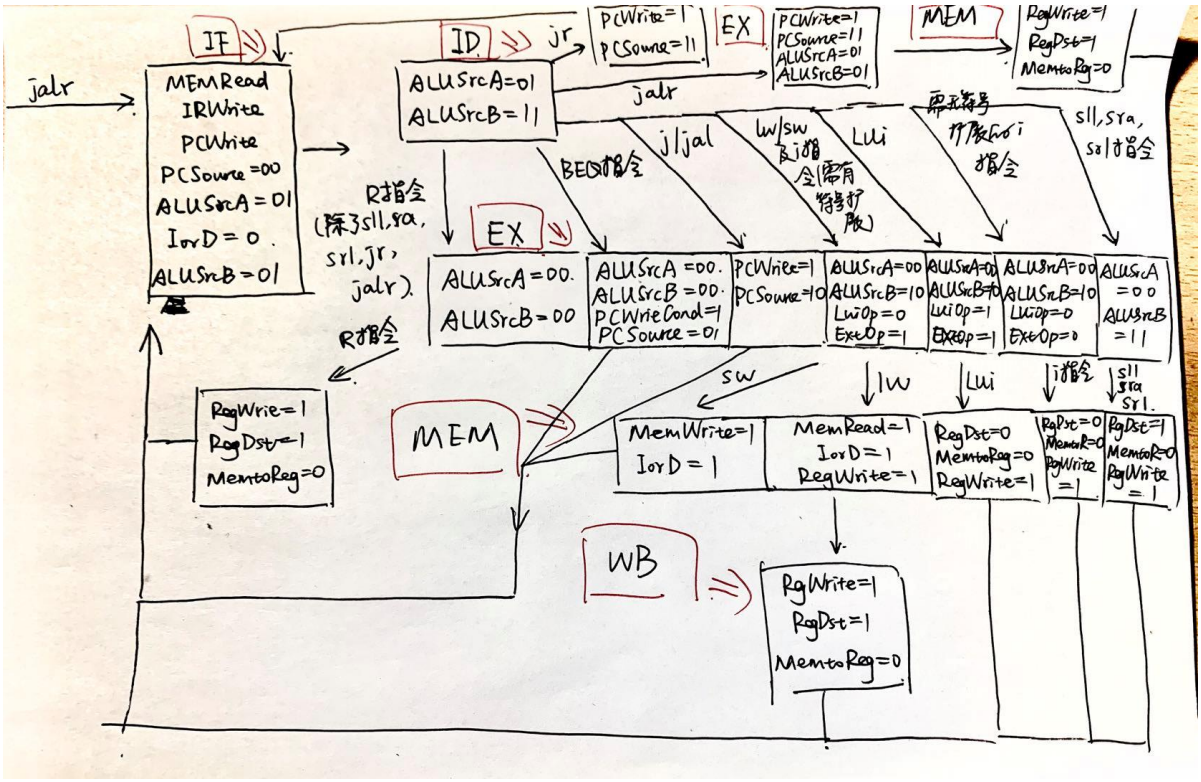
代表输入偏移后的拓展立即数；10代表输入拓展立即数；

$ALUOp$：ALU所进行操作的决定变量，根据不同的指令生成不同的信号，控制ALU执行指令所需要的计算功能，如求和、做差等等；

$PCSource$：PC输入数据的决定变量，00代表直接输入ALU的计算结果，01代表输入$ALUOut$寄存器的输出值，10代表输入j指令的跳转地址，11代表输入rs。

2.2

状态转移图如下，与简单指令集相比加入了i指令，jr、$jalr$等指令的状态，也改变了例如$ALUSrcA$等控制信号的比特数：



代码较长，写在$code$文件中，且加有注释方便阅读。

备注：在做后续题目的时候发现上述状态转移图与数据通路设计存在问题，在此订正。首先是$RegDst$与$MemtoReg$信号应该改为$2bit$，用途是在jal指令后将地址保存在ra寄存器中（此处将ra定义为30号寄存器）。修改后的代码如下：

```
            else if(OpCode==6'h03)begin//jal 指令
                PCWrite<=1'b1;
                PCSource<=2'b10;
                RegDst<=2'b11;
                MemtoReg<=2'b11;
                RegWrite<=1'b1;
                state<=sIF;
                end
```

3. *ALU*功能拓展

3.1

该指令为R型指令，包含3个寄存器的寄存器号与$Funct$码。定义机器码如下：$OpCode$：$6'b0$，rs，rt，rd，$shamt$：$5'b0$，$Funct$：$0x2f$

3.2

认为计算结果$rd = rs - rt$，根据真值表可以得到相当于是按位运算，$rd = rs\&(\sim rt)$，即可得到结果。

代码修改思路为：

先在$Controller$中修改$ALUOp$，定义为111时代表$setsub$运算；在$ALUControl$中定义$setsub$指令的$ALUConf = 11111$；最后在ALU模块中添加计算语句即可。

3.3

首先进行理论计算，$A = 0xabcd1234$，$B = 0xcdef3456$，理论上两者的差值应该为$0x22000220$。先存储数据，思路为先存入$0x0000abcd$，再左移16位后与$0x00001234$相加；另一个数据同样；最后调用$setsub$即可。修改代码如下：

```
            // addi $a0, $zero, 0xabcd
            RAM_data[8'd0] <= {6'h08, 5'd0 , 5'd4 , 16'habcd};
            // sll $a2, $a0, 16,a2=0xabcd0000
            RAM_data[8'd1] <= {6'h00, 5'd0 , 5'd4 , 5'd6 , 5'd16
   , 6'h00};
            // addiu $a1, $a2, 0x1234
            RAM_data[8'd2] <= {6'h09, 5'd6 , 5'd5 , 16'h1234};
            // addi $a3, $zero, 0xcdef
            RAM_data[8'd3] <= {6'h08, 5'd0 , 5'd7 , 16'hcdef};
            // sll $t0, $a3, 16,a2=0xcdef0000
            RAM_data[8'd4] <= {6'h00, 5'd0 , 5'd7 , 5'd8 , 5'd16
   , 6'h00};
            // addiu $t1, $t0, 0x3456
            RAM_data[8'd5] <= {6'h09, 5'd8 , 5'd9 , 16'h3456};
            // setsub $t2 $a1 $t1
```

```
14.             RAM_data[8'd6] <= {6'h00, 5'd5 , 5'd9 , 5'd10, 5'b0,
    6'h2f};
15.             // Loop:
16.             // j Loop
17.             RAM_data[8'd7] <= {6'h02, 26'd7};
```

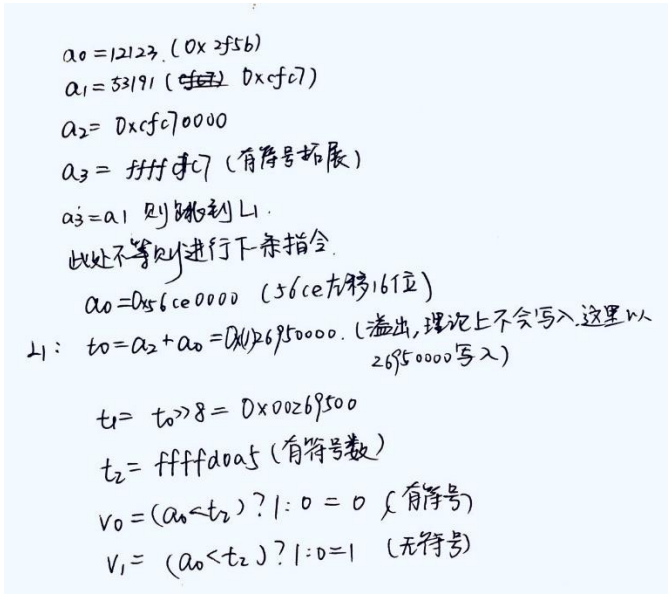
仿真结果如下（仅查看了寄存器中的数值）：

5号寄存器即为a_1，9号寄存器即为t_1；最终结果存在10号寄存器即t_2中。观察到结果与理论计算的正确结果一致，因此ALU功能拓展是正确的。

4. 汇编程序分析-1

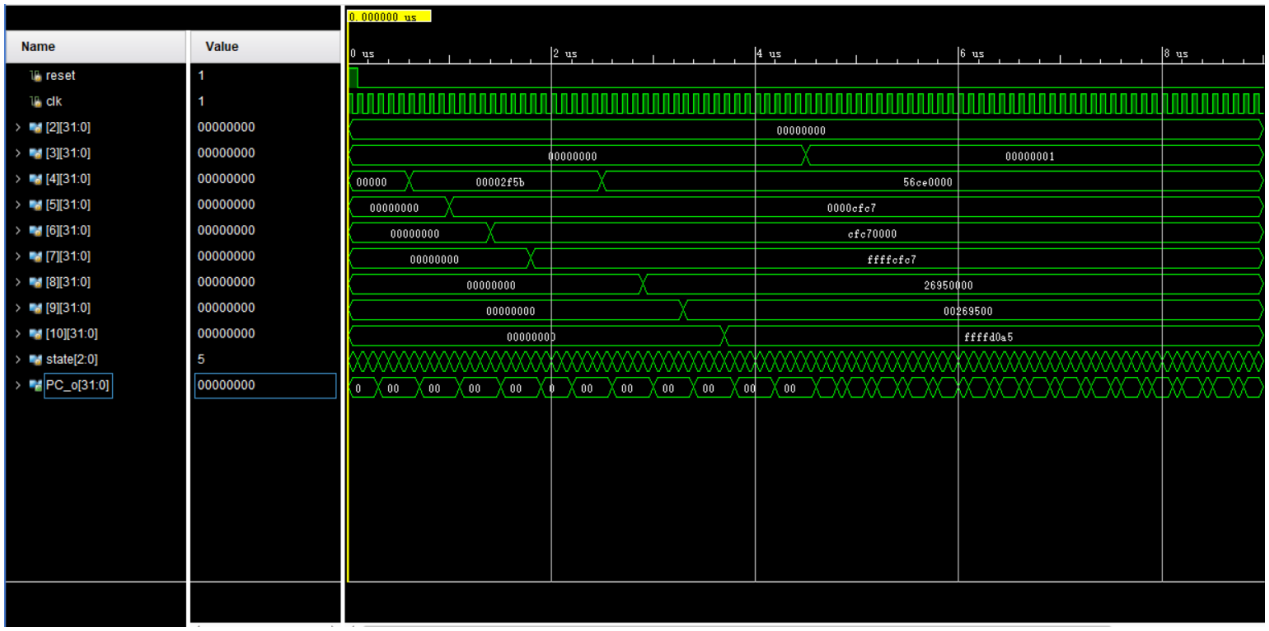
4. 1

具体计算结果及过程如下图所示：



4.2

在*vivado*中进行仿真验证结果，仿真结果如下：



首先观察寄存器中的结果，在程序中寄存器即为连续的数组，其中2号为v_0，3号为v_1，4、5、6、7号为a_0、a_1、a_2、a_3，8、9、10号为t_0、t_1、t_2。将仿真结果与实际计算结果对比，发现最终寄存器中存储的数据正确无误；且观察PC的变化，发现PC每次加4，最后进入30与26的循环，符合程序中最后的循环体特点。因此多周期处理器的设计功能正确，具体代码在*code*文件夹中。

5. 汇编程序分析-2

5.1

为代码加上注释后：

```
0 addi $a0, $zero, 5        #a0=5
1 xor $v0, $zero, $zero     #异或操作，v0=0
2 jal sum                   #跳到sum 部分执行代码
Loop:
3 beq $zero, $zero, Loop    #循环体，作为结束
sum:
4 addi $sp, $sp, -8         #开辟栈空间
5 sw $ra, 4($sp)            #保护现场，将跳回地址存在sp+4 中
6 sw $a0, 0($sp)            #保护现场，将a 存在sp 中
7 slti $t0, $a0, 1          #a0<1 则t0=1，否则t0=0
8 beq $t0, $zero, L1        #t0 是否是0，是的话跳到L1 执行代码
9 addi $sp, $sp, 8
10 jr $ra
L1:
11 add $v0, $a0, $v0        #v0=a0+v0
12 addi $a0, $a0, -1        #a0=a0-1
13 jal sum                  #跳到sum
14 lw $a0, 0($sp)           #恢复现场
15 lw $ra, 4($sp)           #恢复现场
16 addi $sp, $sp, 8         #恢复栈空间
17 add $v0, $a0, $v0        #v0=a0+v0
18 jr $ra                   #跳回
```

这段程序实现的功能是求n到0的整数和的2倍。*Loop*是循环，作为程序的结束；*sum*段代码主要是判断求和是否结束；*L*1段代码主要是2次求和并跳回*sum*判断部分。

5.2

翻译的机器码如下：

```
            // addi $a0, $zero, 5
            RAM_data[8'd0] <= {6'h08, 5'd0 , 5'd4 , 16'd5};
            //addi $sp $zero 0x3fc
            RAM_data[8'd1] <= {6'h08, 5'd0 , 5'd31 , 16'hdf};
            // xor $v0 $zero $zero
            RAM_data[8'd2] <= {6'h00, 5'd0 , 5'd0 , 5'd2 , 5'd0 ,
    6'h26};
            // jal sum
            RAM_data[8'd3] <= {6'h03, 26'd5};

            // Loop:
            //j Loop
```

```
            RAM_data[8'd4] <= {6'h02,26'd4};

            //sum:
            // addi $sp $sp -8
            RAM_data[8'd5] <= {6'h08, 5'd31 , 5'd31 , 16'hfff8};
            // sw $ra, 4($sp)
            RAM_data[8'd6] <= {6'h2b, 5'd31 , 5'd30 , 16'd4};
            // sw $a0, 0($sp)
            RAM_data[8'd7] <= {6'h2b, 5'd31 , 5'd4 , 16'd0};
            //slti $t0, $a0, 1
            RAM_data[8'd8] <= {6'h0a, 5'd4 , 5'd8 , 16'd1};
            //beq $t0, $zero, L1
            RAM_data[8'd9] <= {6'h04, 5'd8 , 5'd0 , 16'hfffe};
            //addi $sp, $sp, 8
            RAM_data[8'd10] <= {6'h08, 5'd31 , 5'd31 , 16'd8};
            //jr $ra
            RAM_data[8'd11] <= {6'h00, 5'd30 , 15'd0 , 6'd08};

            //L1:
            //add $v0, $a0, $v0
            RAM_data[8'd12] <= {6'h00, 5'd4 , 5'd2 , 5'd2 , 5'b0
   , 6'h20};
            //addi $a0, $a0, -1
            RAM_data[8'd13] <= {6'h08, 5'd4 , 5'd4 , 16'hffff};
            //jal sum
            RAM_data[8'd14] <= {6'h03, 26'd5};
            //lw $a0, 0($sp)
            RAM_data[8'd15] <= {6'h23, 5'd31 , 5'd4 , 16'h0};
            //lw $ra, 4($sp)
            RAM_data[8'd16] <= {6'h23, 5'd31 , 5'd30 , 16'd4};
            //addi $sp, $sp, 8
            RAM_data[8'd17] <= {6'h08, 5'd31 , 5'd31 , 16'd8};
            //add $v0, $a0, $v0
            RAM_data[8'd18] <= {6'h00, 5'd4 , 5'd2 , 5'd2 , 5'b0
   , 6'd20};
            //jr $ra
            RAM_data[8'd19] <= {6'h00, 5'd30 , 15'd0 , 6'd08};
```

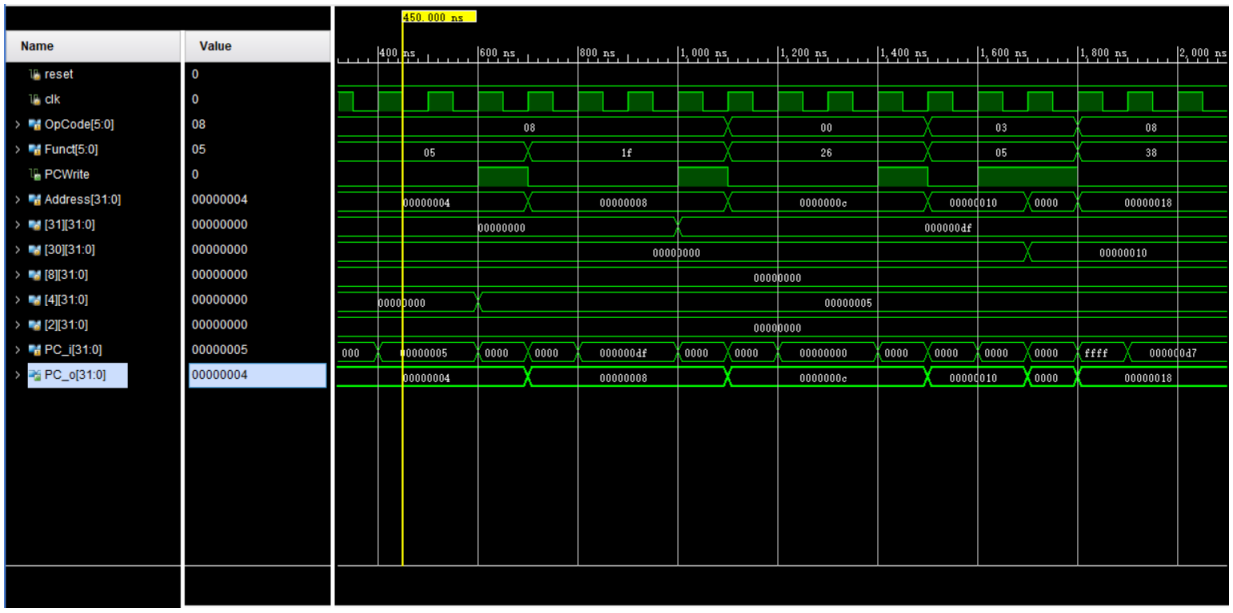
5.3

仿真验证，结果如下：

此处的4号寄存器为a_0，2号寄存器为v_0，a_0中的数值为5，v_0中的数值为30，符合预期结果。在调试过程中遇到了太多太多的问题，主要是没有考虑到jal指令使用后需要将下一条地址写在ra寄存器中，且在考虑sp的初始位置数值与存储器内部的数组设置时出现了问题，耽误了很长时间。

5.4

PC寄存器的输入输出情况如图（仅包含部分，完整情况见代码的仿真结果）



PC寄存器的读取值每次+4，在遇到跳转时会跳到对应的指令位置；PC的数值与存储器中指令编号为4倍的关系；

a_0的变化为5 4 3 2 1 0 1 2 3 4 5，与程序中的先减小后出栈还原寄存器中的数值的预期结果一致；

v_0的变化为5 9 12 14 15 15 16 18 21 25 30，先是正常的求5到1的整数和，之后是循环中求与出栈的数值的和，最后是正确结果30；

sp初始值为$0xdf$，之后在循环过程中每次减8，开辟出的栈空间分别存储a_0与ra中的地址，最后在恢复现场的过程中存回；

ra为30号寄存器，在jal指令后将后一条指令的地址存在其中，在jr指令中使用，以达到跳回的目的；程序中的变化与预期结果一致。

6. 实验感想

在写本次多周期处理器的过程中遇到了很多很多的问题，也感觉到了很多困难；好在最终通过不断地$debug$与研究完成了大作业，并且感受到本次大作业对于理解课上的知识与学会熟练运用处理器的原理有着很大的帮助。实验中的代码在文件夹$code$中一并提交，感谢助教们本学期的辛苦付出！