同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	实验 1-89 条流水线 CPU
实验成员	王钧涛 2050254
日　　期	二零二三年　四月　五日

1、实验目的

1. 改造 54 条指令 cpu，支持到 89 条，实现 CP0、异常处理。
2. 下板，通过验证，确认设计正确性。

2、实验内容

主要的实验内容就是改造 CPU，支持到 89 条指令。参考《自己动手写 CPU》，改造 54 条指令 MIPS 流水线 CPU，支持到 89 条指令，实现 CP0、异常处理，需要新增的 35 条指令如下：

·移动操作指令：

movn、movz

·算术操作指令：

clo、madd、maddu、msub、msubu

·转移指令：

b、bal、bgezal、bgtz、blez、bltz、bltzal

·加载存储指令：

ll、lwl、lwr、sc、swl、swr

·异常相关指令：

tge、tgeu、tlt、tltu、tne、teqi、tgei、tgeiu、tlti、tltiu、tnei

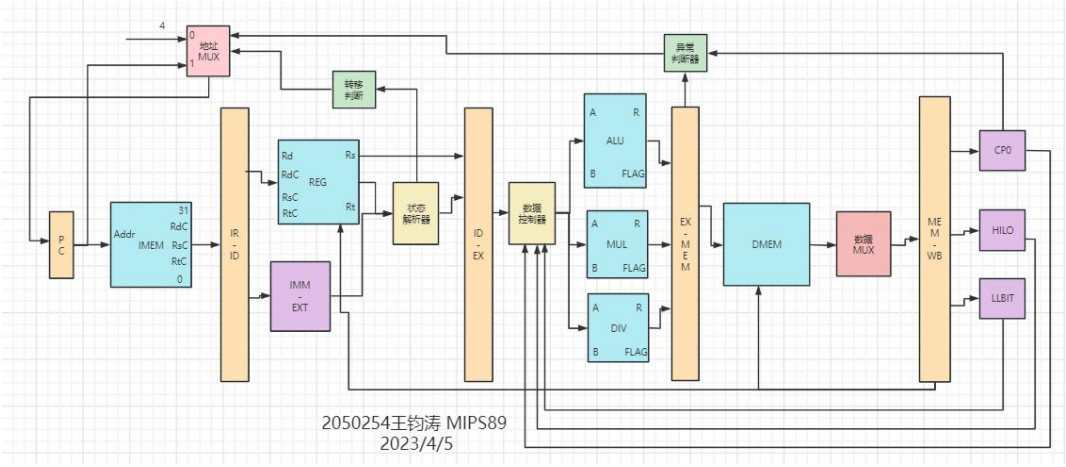
·其他指令：

nop、ssnop、sync、pref

3、实验步骤

1、整体数据通路设计与指令设计：

本次试验所实现的 89 条 MIPS CPU 的整体数据通路如下：



其中，移动操作 MOVN、MOVZ 被当作算数指令一起处理，在 EX 段的基础上修改而成，具体指令格式设计如下：

1. movn

• 格式：movn rd,rs,rt

• 描述：if rt ≠0 then rd rs，判断地址为 rt 的通用寄存器的值，如果不为零，那么将地址为 rs 的通用寄存器的值赋给地址为 rd 的通用寄存器；反之，保持地址为 rd 的通用寄存器不变。movn 即为 Move Conditional on Not Zero 的意思。

2. movz

• 格式：movz rd,rs,rt

• 描述：if rt = 0 then rd rs，与 movn 相反，判断地址为 rt 的通用寄存器的值，如果为零，那么将地址为 rs 的通用寄存器的值赋给地址为 rd 的通用寄存器；反之，保持地址为 rd 的通用寄存器不变。movn 即为 Move Conditional on Zero 的意思。

```
#EX.v
        `EXE_MOVZ_OP:       begin
            moveres <= reg1_i;
        end
        `EXE_MOVN_OP:       begin
            moveres <= reg1_i;
        end
```

II.算术操作指令 clo、madd、maddu、msub、msubu 的引入主要是为了实现位统计和 64 位宽字节的计算，对其的增加同样主要针对 EX 段进行，同时对译码阶段进行修改。具体指令设计如下：

3.clo

·格式：clo rd,rs

·描述：rd coun_leading_ones rs，对地址为 rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“0”的位，将该位之前“1”的个数保存到地址为 rd 的通用寄存器中，如果地址为 rs 的通用寄存器的所有位都为 1(即 0xFFFFFFFF)，那么将保存到地址为 rd 的通用寄存器中。

·部件：PC, NPC, IMEM, RegFile

```
#EX.v
            `EXE_CLZ_OP:        begin
                arithmeticres <= reg1_i[31] ? 0 : reg1_i[30] ? 1 : reg1_i[29] ? 2 :
                                 reg1_i[28] ? 3 : reg1_i[27] ? 4 : reg1_i[26] ? 5 :
                                 reg1_i[25] ? 6 : reg1_i[24] ? 7 : reg1_i[23] ? 8 :
                                 reg1_i[22] ? 9 : reg1_i[21] ? 10 : reg1_i[20] ? 11 :
                                 reg1_i[19] ? 12 : reg1_i[18] ? 13 : reg1_i[17] ? 14 :
                                 reg1_i[16] ? 15 : reg1_i[15] ? 16 : reg1_i[14] ? 17 :
                                 reg1_i[13] ? 18 : reg1_i[12] ? 19 : reg1_i[11] ? 20 :
                                  reg1_i[10] ? 21 : reg1_i[9] ? 22 : reg1_i[8] ? 23 :
                                  reg1_i[7] ? 24 : reg1_i[6] ? 25 : reg1_i[5] ? 26 :
                                  reg1_i[4] ? 27 : reg1_i[3] ? 28 : reg1_i[2] ? 29 :
                                  reg1_i[1] ? 30 : reg1_i[0] ? 31 : 32 ;
```

4.madd

·格式：madd rs,rt

·描述：{HI, LO} {HI, LO}+ rs ×rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数进行乘法运算，运算结果与{HI, LO}相加，相加的结果保存到{HI, LO}中。此处{HI, LO}表示 HI、LO 寄存器连接形成的 64 位数，HI 是高 32 位，LO 是低 32 位。

·部件：PC, NPC, IMEM, RegFile,HI,LO

5.maddu

·格式：maddu rs,rt

·描述：{HI, LO} {HI, LO}+ rs ×rt，将地址为 rs 的通用寄存器的值与

地址为 rt 的通用寄存器的值作为无符号数进行乘法运算，运算结果与{HI, LO}相加，相加的结果保存到{HI, LO}中。此处{HI, LO}表示 HI、LO 寄存器连接形成的 64 位数，HI 是高 32 位，LO 是低 32 位。

• 部件：PC, NPC, IMEM, RegFile,HI,LO

6. msub

• 格式：msub rs,rt

• 描述：{HI, LO} {HI, LO}-rs ×rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数进行乘法运算，运算使用{HI, LO}减去乘法结果，相减的结果保存到{HI, LO}中。

• 部件：PC, NPC, IMEM, RegFile,HI,LO 6

7. msubu

• 格式：msubu rs,rt

• 描述：{HI, LO} {HI, LO}-rs ×rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行乘法运算，运算使用{HI, LO}减去乘法结果，相减的结果保存到{HI, LO}中。

• 部件：PC, NPC, IMEM, RegFile,HI,LO

```
#EX.v
always @ (*) begin
        if(rst == `RstEnable) begin
            hilo_temp_o <= {`ZeroWord,`ZeroWord};
            cnt_o <= 2'b00;
            stallreq_for_madd_msub <= `NoStop;
        end else begin

            case (aluop_i)
                `EXE_MADD_OP, `EXE_MADDU_OP:        begin
                    if(cnt_i == 2'b00) begin
                        hilo_temp_o <= mulres;
                        cnt_o <= 2'b01;
                        stallreq_for_madd_msub <= `Stop;
                        hilo_temp1 <= {`ZeroWord,`ZeroWord};
                    end else if(cnt_i == 2'b01) begin
                        hilo_temp_o <= {`ZeroWord,`ZeroWord};

                        cnt_o <= 2'b10;
                        hilo_temp1 <= hilo_temp_i + {HI,LO};
                        stallreq_for_madd_msub <= `NoStop;
```

```
                    end
                end
                `EXE_MSUB_OP, `EXE_MSUBU_OP:        begin
                    if(cnt_i == 2'b00) begin
                        hilo_temp_o <=  ~mulres + 1 ;
                        cnt_o <= 2'b01;
                        stallreq_for_madd_msub <= `Stop;
                    end else if(cnt_i == 2'b01)begin
                         hilo_temp_o <= {`ZeroWord,`ZeroWord};

                        cnt_o <= 2'b10;
                        hilo_temp1 <= hilo_temp_i + {HI,LO};
                        stallreq_for_madd_msub <= `NoStop;
                    end
                end
                default:    begin
                    hilo_temp_o <= {`ZeroWord,`ZeroWord};
                    cnt_o <= 2'b00;
                     stallreq_for_madd_msub <= `NoStop;

                end
            endcase
        end
    end
```

III.转移指令 b、bal、bgezal、bgtz、blez、bltz、bltzal 的引入在很大程度上丰富了指令集的跳转结构，使得程序控制部分更加精简，效率更高。该部分的设置较为复杂，不仅要向 PC 传输跳转指令和计算出的跳转地址，同时也需要考虑到流水线的暂停、恢复问题和延迟槽的设置问题，主要在 ID 段中进行实现，具体的指令格式设计如下,其中 B 和 BAL 指令直接将特殊情况写入了 BEQ 和 BAL 指令处理段中。

8.b

·格式：b offset

·描述：无条件转移，b 指令可以认为是 beq 指令的特殊情况，当 beq 指令的 rs、rt 都等于 0 时，即为 b 指令。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

9.bal

·格式：baloffset

·描述：无条件转移，并且将转移指令后面第 2 条指令的地址作为返回地址，保存到通用寄存器$31。bal 指令是 bgezal 指令的特殊情况，当 bgezal 指令的 rs 为 0 时，就是 bal 指令。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

10.bgezal

·格式：bgezalrs,offset

·描述：if rs ≥0 then branch，如果地址为 rs 的通用寄存器的值大于等于 0，那么发生转移，并且将转移指令后面第 2 条指令的地址作为返回地址，保存到通用寄存器$31。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

11.bgtz

·格式：bgtzrs,offset 7

·描述：if rs > 0 then branch，如果地址为 rs 的通用寄存器的值大于零，那么发生转移。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

12.blez

·格式：blezrs,offset

·描述：if rs ≤0 then branch，如果地址为 rs 的通用寄存器的值小于等于零，那么发生转移。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

13.bltz

·格式：bltzrs,offset

·描述：if rs < 0 then branch，如果地址为 rs 的通用寄存器的值小于 0，那么发生转移。

·部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

14.bltzal

·格式：bltzal rs,offset

·描述：if rs < 0 then branch，如果地址为 rs 的通用寄存器的值小于 0，

那么发生转移，并且将转移指令后面第 2 条指令的地址作为返回地址，保存到通用寄存器$31。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT18, ADD

```
#EX.v
`EXE_BGEZ:  begin
    wreg_o <= `WriteDisable;     aluop_o <= `EXE_BGEZ_OP;
        alusel_o <= `EXE_RES_JUMP_BRANCH; reg1_read_o <= 1'b1;
 reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
        if(reg1_o[31] == 1'b0) begin
            branch_target_address_o <= pc_plus_4 + imm_sll2_signe
dext;
            branch_flag_o <= `Branch;
            next_inst_in_delayslot_o <= `InDelaySlot;

            end
end
`EXE_BGEZAL:        begin
    wreg_o <= `WriteEnable;      aluop_o <= `EXE_BGEZAL_OP;
        alusel_o <= `EXE_RES_JUMP_BRANCH; reg1_read_o <= 1'b1;
 reg2_read_o <= 1'b0;
        link_addr_o <= pc_plus_8;
        wd_o <= 5'b11111;    instvalid <= `InstValid;
        if(reg1_o[31] == 1'b0) begin
            branch_target_address_o <= pc_plus_4 + imm_sll2_sign
edext;
            branch_flag_o <= `Branch;
            next_inst_in_delayslot_o <= `InDelaySlot;
            end
end
`EXE_BLTZ:      begin
  wreg_o <= `WriteDisable;       aluop_o <= `EXE_BGEZAL_OP;
        alusel_o <= `EXE_RES_JUMP_BRANCH; reg1_read_o <= 1'b1;
 reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
        if(reg1_o[31] == 1'b1) begin
            branch_target_address_o <= pc_plus_4 + imm_sll2_sign
edext;
            branch_flag_o <= `Branch;
            next_inst_in_delayslot_o <= `InDelaySlot;

            end
end
`EXE_BLTZAL:        begin
    wreg_o <= `WriteEnable;      aluop_o <= `EXE_BGEZAL_OP;
```

```
35.          alusel_o <= `EXE_RES_JUMP_BRANCH; reg1_read_o <= 1'b1;
     reg2_read_o <= 1'b0;
36.          link_addr_o <= pc_plus_8;
37.          wd_o <= 5'b11111; instvalid <= `InstValid;
38.          if(reg1_o[31] == 1'b1) begin
39.               branch_target_address_o <= pc_plus_4 + imm_sll2_sign
   edext;
40.              branch_flag_o <= `Branch;
41.              next_inst_in_delayslot_o <= `InDelaySlot;
             end
```

IV. 存储指令 ll、lwl、lwr、sc、swl、swr 为程序设计提供了更为丰富的数据加载和数据存储方式，为精确的字节级存储控制和不同类型数据处理提供了更优化、更精简的编写模式。对该段的设计主要在 MEM 段中进行拓展和修改，具体的指令格式设计如下：

15. ll

• 格式：ll rt,offset(base)

• 描述：从内存中指定的加载地址处，读取一个字节，然后符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM 8

16. lwl

• 格式：lwl rt,offset(base)

• 描述：从内存中指定的加载地址处，加载一个字的最高有效部分。lwl 指令对加载地址没有要求，从而允许地址非对齐加载，这是与 lh、lhu、lw 指令的不同之处。在大端模式、小端模式下，lwl 指令的效果不同。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM

17. lwr

• 格式：lwrrt,offset(base)

• 描述：从内存中指定的加载地址处，加载一个字的最低有效部分。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM

```
1.  #MEM.v
2. `EXE_LWL_OP:         begin
3.     mem_addr_o <= {mem_addr_i[31:2], 2'b00};
4.     mem_we <= `WriteDisable;
```

```
5.      mem_sel_o <= 4'b1111;
6.      mem_ce_o <= `ChipEnable;
7.      case (mem_addr_i[1:0    ])
8.          2'b00:  begin
9.              wdata_o <= mem_data_i[31:0];
10.         end
11.         2'b01:  begin
12.             wdata_o <= {mem_data_i[23:0],reg2_i[7:0]};
13.         end
14.         2'b10:  begin
15.             wdata_o <= {mem_data_i[15:0],reg2_i[15:0]};
16.         end
17.         2'b11:  begin
18.             wdata_o <= {mem_data_i[7:0],reg2_i[23:0]};
19.         end
20.         default:    begin
21.             wdata_o <= `ZeroWord;
22.         end
23.     endcase
24. end
25. `EXE_LWR_OP:        begin
26.     mem_addr_o <= {mem_addr_i[31:2], 2'b00};
27.     mem_we <= `WriteDisable;
28.     mem_sel_o <= 4'b1111;
29.     mem_ce_o <= `ChipEnable;
30.     case (mem_addr_i[1:0])
31.         2'b00:  begin
32.             wdata_o <= {reg2_i[31:8],mem_data_i[31:24]};
33.         end
34.         2'b01:  begin
35.             wdata_o <= {reg2_i[31:16],mem_data_i[31:16]};
36.         end
37.         2'b10:  begin
38.             wdata_o <= {reg2_i[31:24],mem_data_i[31:8]};
39.         end
40.         2'b11:  begin
41.             wdata_o <= mem_data_i;
42.         end
43.         default:    begin
44.             wdata_o <= `ZeroWord;
45.         end
46.     endcase
47. end
48. `EXE_LL_OP:     begin
49.     mem_addr_o <= mem_addr_i;
50.     mem_we <= `WriteDisable;
51.     wdata_o <= mem_data_i;
```

装 订 线

```
52.         LLbit_we_o <= 1'b1;
53.         LLbit_value_o <= 1'b1;
54.         mem_sel_o <= 4'b1111;
55.         mem_ce_o <= `ChipEnable;
56. end
```

18. sc

• 格式：sc rt,offset(base)

• 描述：如果 RMW 序列没有受到干扰，也就是 LLbit 为 1，那么将地址为 rt 的通用寄存器的值保存到内存中指定的存储地址处，同时设置地址为 rt 的通用寄存器的值为 1，设置 LLbit 为 0。如果 RMW 序列受到了干扰，也就是 LLbit 为 0，那么不修改内存，同时设置地址为 rt 的通用寄存器的值为 0。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM

19. swl

• 格式：swl rt,offset(base)

• 描述：将地址为 rt 的通用寄存器的高位部分存储到内存中指定的地址处，存储地址的最后两位确定了要存储 rt 通用寄存器的哪几个字节。swl 指令对存储地址没有对齐要求，这是与 sh、sw 指令的不同之处。在大端模式、小端模式下，swl 指令的效果不同。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM 9

20. swr

• 格式：swrrt,offset(base)

• 描述：将地址为 rt 的通用寄存器的低位部分存储到内存中指定的地址处。

• 部件：PC, NPC, IMEM, RegFile, ALU, EXT16, DMEM

```
1.  #MEM.v
2. `EXE_SWL_OP:         begin
3.     mem_addr_o <= {mem_addr_i[31:2], 2'b00};
4.     mem_we <= `WriteEnable;
5.     mem_ce_o <= `ChipEnable;
6.     case (mem_addr_i[1:0])
7.         2'b00:  begin
8.             mem_sel_o <= 4'b1111;
9.             mem_data_o <= reg2_i;
10.        end
```

```
11.         2'b01:  begin
12.             mem_sel_o <= 4'b0111;
13.             mem_data_o <= {zero32[7:0],reg2_i[31:8]};
14.         end
15.         2'b10:  begin
16.             mem_sel_o <= 4'b0011;
17.             mem_data_o <= {zero32[15:0],reg2_i[31:16]};
18.         end
19.         2'b11:  begin
20.             mem_sel_o <= 4'b0001;
21.             mem_data_o <= {zero32[23:0],reg2_i[31:24]};
22.         end
23.         default:    begin
24.             mem_sel_o <= 4'b0000;
25.         end
26.     endcase
27. end
28. `EXE_SWR_OP:        begin
29.     mem_addr_o <= {mem_addr_i[31:2], 2'b00};
30.     mem_we <= `WriteEnable;
31.     mem_ce_o <= `ChipEnable;
32.     case (mem_addr_i[1:0])
33.         2'b00:  begin
34.             mem_sel_o <= 4'b1000;
35.             mem_data_o <= {reg2_i[7:0],zero32[23:0]};
36.         end
37.         2'b01:  begin
38.             mem_sel_o <= 4'b1100;
39.             mem_data_o <= {reg2_i[15:0],zero32[15:0]};
40.         end
41.         2'b10:  begin
42.             mem_sel_o <= 4'b1110;
43.             mem_data_o <= {reg2_i[23:0],zero32[7:0]};
44.         end
45.         2'b11:  begin
46.             mem_sel_o <= 4'b1111;
47.             mem_data_o <= reg2_i[31:0];
48.         end
49.         default:    begin
50.             mem_sel_o <= 4'b0000;
51.         end
52.     endcase
53. end
54. `EXE_SC_OP:     begin
55.     if(LLbit == 1'b1) begin
56.         LLbit_we_o <= 1'b1;
57.         LLbit_value_o <= 1'b0;
```

```
58.             mem_addr_o <= mem_addr_i;
59.             mem_we <= `WriteEnable;
60.             mem_data_o <= reg2_i;
61.             wdata_o <= 32'b1;
62.             mem_sel_o <= 4'b1111;
63.             mem_ce_o <= `ChipEnable;
64.         end else begin
65.             wdata_o <= 32'b0;
66.         end
67. end
```

V.异常处理指令 tge、tgeu、tlt、tltu、tne、teqi、tgei、tgeiu、tlti、tltiu、tnei 等，实现。此部分代码的数值计算在 EX 段的跳转过程中已经被较好地解决了，因此这里主要修改的是 ID 译码过程，需要考虑到 TRAP 过程对整个 CPU 的数据流的排空问题，具体的指令设计如下：

21.tge

• 格式：tge rs,rt

• 描述：if GPR[rs] ≥GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为有符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

22.tgeu

• 格式：tgeurs,rt

• 描述：if GPR[rs] ≥GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为无符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

23.tlt

• 格式：tlt rs,rt

• 描述：if GPR[rs] < GPR[rt]then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为有符号数进行比较，如果前者小于后者，那么引发自陷异常。

• 部件：PC， NPC,CP0,IMEM,RegFiles,ALU 10

24. tltu

• 格式：tltu rs,rt

• 描述：if GPR[rs] < GPR[rt]then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

• 部件：PC， NPC,CP0,IMEM,RegFiles,ALU

25. tne

• 格式：tne rs,rt

• 描述：if GPR[rs] ≠GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值进行比较，如果不相等，那么引发自陷异常。

• 部件：PC， NPC,CP0,IMEM,RegFiles,ALU

```
#ID.v
`EXE_TEQ: begin
wreg_o <= `WriteDisable;     aluop_o <= `EXE_TEQ_OP;
    alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b0;  reg2_read_o <= 1'b0;
    instvalid <= `InstValid;
end
`EXE_TGE: begin
wreg_o <= `WriteDisable;     aluop_o <= `EXE_TGE_OP;
    alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b1;  reg2_read_o <= 1'b1;
    instvalid <= `InstValid;
end
`EXE_TGEU: begin
wreg_o <= `WriteDisable;     aluop_o <= `EXE_TGEU_OP;
    alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b1;  reg2_read_o <= 1'b1;
    instvalid <= `InstValid;
end
`EXE_TLT: begin
wreg_o <= `WriteDisable;     aluop_o <= `EXE_TLT_OP;
    alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b1;  reg2_read_o <= 1'b1;
    instvalid <= `InstValid;
end
`EXE_TLTU: begin
```

```
23. wreg_o <= `WriteDisable;     aluop_o <= `EXE_TLTU_OP;
24.     alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b1;  reg2_re
    ad_o <= 1'b1;
25.     instvalid <= `InstValid;
26. end
27. `EXE_TNE: begin
28. wreg_o <= `WriteDisable;     aluop_o <= `EXE_TNE_OP;
29.     alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b1;  reg2_re
    ad_o <= 1'b1;
30.     instvalid <= `InstValid;
31. end
32. `EXE_SYSCALL: begin
33. wreg_o <= `WriteDisable;     aluop_o <= `EXE_SYSCALL_OP;
34.     alusel_o <= `EXE_RES_NOP;   reg1_read_o <= 1'b0;  reg2_re
    ad_o <= 1'b0;
35.     instvalid <= `InstValid; excepttype_is_syscall<= `True_v;

36. end
```

26. teqi

• 格式：teqi rs,immediate

• 描述：if GPR[rs] = sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值进行比较，如果两者相等，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

27. tgei

• 格式：tgei rs,immediate

• 描述：if GPR[rs] ≥sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为有符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

• 数据通路：11

28. tgeiu

• 格式：tgeiu rs,immediate

• 描述：if GPR[rs] ≥sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为无符号

数进行比较，如果前者大于等于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

29. tlti

• 格式：tlti rs,immediate

• 描述：if GPR[rs] ＜sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为有符号数进行比较，如果前者小于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

30. tltiu

• 格式：tltiu rs,immediate

• 描述：if GPR[rs] ＜sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

31. tnei

• 格式：tnei rs,immediate

• 描述：if GPR[rs] ≠sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值进行比较，如果两者不相等，那么引发自陷异常。

• 部件：PC, NPC,CP0,IMEM,RegFiles,ALU

```
#ID.v
`EXE_TEQI:          begin
        wreg_o <= `WriteDisable;     aluop_o <= `EXE_TEQI_OP;
        alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg2_read_o <= 1'b0;
    imm <= {{16{inst_i[15]}}, inst_i[15:0]};
    instvalid <= `InstValid;
end
`EXE_TGEI:          begin
        wreg_o <= `WriteDisable;     aluop_o <= `EXE_TGEI_OP;
         alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg2_read_o <= 1'b0;
```

```
11.     imm <= {{16{inst_i[15]}}, inst_i[15:0]};
12.     instvalid <= `InstValid;
13. end
14. `EXE_TGEIU:         begin
15.         wreg_o <= `WriteDisable;     aluop_o <= `EXE_TGEIU_OP;
16.         alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg
    2_read_o <= 1'b0;
17.     imm <= {{16{inst_i[15]}}, inst_i[15:0]};
18.     instvalid <= `InstValid;
19. end
20. `EXE_TLTI:          begin
21.         wreg_o <= `WriteDisable;     aluop_o <= `EXE_TLTI_OP;
22.         alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg
    2_read_o <= 1'b0;
23.     imm <= {{16{inst_i[15]}}, inst_i[15:0]};
24.     instvalid <= `InstValid;
25. end
26. `EXE_TLTIU:         begin
27.         wreg_o <= `WriteDisable;     aluop_o <= `EXE_TLTIU_OP;
28.         alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg
    2_read_o <= 1'b0;
29.     imm <= {{16{inst_i[15]}}, inst_i[15:0]};
30.     instvalid <= `InstValid;
31. end
32. `EXE_TNEI:          begin
33.         wreg_o <= `WriteDisable;     aluop_o <= `EXE_TNEI_OP;
34.         alusel_o <= `EXE_RES_NOP; reg1_read_o <= 1'b1;    reg
    2_read_o <= 1'b0;
35.     imm <= {{16{inst_i[15]}}, inst_i[15:0]};
36.     instvalid <= `InstValid;
37. end
```

VI. 未设计指令

32. nop

• 描述：空指令。

33. ssnop

• 描述：一种特殊类型的空指令，此次实验中实现的 MIPS CPU 中与 nop 的作用相同，所以可以按照 nop 指令的处理方式来处理 ssnop 指令。

34. sync

·描述：用于保证加载、存储操作的顺序。此次实验中实现的 MIPS CPU 中，售加载、存储操作是严格按照指令顺序执行的，所以可将 sync 指令当作 nop 指令处理。

35.pref

·描述：用于缓存预取。此次实验中实现的 MIPS CPU 中没有实现缓存，所以可将 pref 指令当作 nop 指令处理。

这四条指令采用的方法为直接译码成 NOP，未进行额外设计。

后期会考虑在引入三级缓存系统后实现 PREF 指令。

2、架构部件解释说明：

I. IF 段：

地址控制器 PC：负责输出当前 PC 位置以及控制新 PC 的产生。

模块定义如下：

```
module pc_reg(
    input wire clk,
    input wire rst,
    //来自控制模块的信息
    input wire[5:0]                 stall,
    input wire                      flush,
    input wire[`RegBus]             new_pc,
    //来自译码阶段的信息
    input wire                      branch_flag_i,
     input wire[`RegBus]             branch_target_address_i,
     //输出
     output reg[`InstAddrBus]        pc,
     output reg                      ce
);
```

II. ID 段：

指令译码器 ID:负责解析来自 IF 段的指令并将其转化为之后控制的输出。

模块定义如下：

```
module id(
    input wire                    rst,
    input wire[`InstAddrBus]      pc_i,
    input wire[`InstBus]          inst_i,
    //执行命令输入
```

```
    input wire[`AluOpBus]       ex_aluop_i,
    input wire                  ex_wreg_i,
    input wire[`RegBus]         ex_wdata_i,
    input wire[`RegAddrBus]     ex_wd_i,
     //数据相关输入
     input wire                  mem_wreg_i,
     input wire[`RegBus]         mem_wdata_i,
     input wire[`RegAddrBus]     mem_wd_i,
     input wire[`RegBus]         reg1_data_i,
     input wire[`RegBus]         reg2_data_i,
     //延迟槽输入
     input wire                  is_in_delayslot_i,
     output reg                  reg1_read_o,
     output reg                  reg2_read_o,
     output reg[`RegAddrBus]     reg1_addr_o,
     output reg[`RegAddrBus]     reg2_addr_o,
     //功能部件命令输出
     output reg[`AluOpBus]       aluop_o,
     output reg[`AluSelBus]      alusel_o,
     output reg[`RegBus]         reg1_o,
     output reg[`RegBus]         reg2_o,
     output reg[`RegAddrBus]     wd_o,
     output reg                  wreg_o,
     output wire[`RegBus]        inst_o,
     output reg                  next_inst_in_delayslot_o,
     output reg                  branch_flag_o,
     output reg[`RegBus]         branch_target_address_o,
     output reg[`RegBus]         link_addr_o,
     output reg                  is_in_delayslot_o,
     output wire[31:0]           excepttype_o,
     output wire[`RegBus]        current_inst_address_o,
     output wire                 stallreq
);
```

寄存器堆 REGS：负责处理对寄存器的读写

模块定义如下：

```
module regfile(
    input wire clk,
    input wire rst,
    //写端口
    input wire                  we,
    input wire[`RegAddrBus]     waddr,
    input wire[`RegBus]         wdata,
    //读端口 1
    input wire                  re1,
     input wire[`RegAddrBus]     raddr1,
```

```
11.     output reg[`RegBus]         rdata1,
12.     //读端口 2
13.     input wire                  re2,
14.     input wire[`RegAddrBus]     raddr2,
15.     output reg[`RegBus]         rdata2
16. );
```

流水线状态控制器 CTRL:负责控制整体流水线的状态，处理流水线暂停、流水线排空，并在指令地址跳转时产生相应的延迟槽和新 PC 地址。

模块定义如下：

```
1. module ctrl(
2.     input wire                  rst,
3.     input wire[31:0]            excepttype_i,
4.     input wire[`RegBus]         cp0_epc_i,
5.     input wire                  stallreq_from_id,
6.     input wire                  stallreq_from_ex,
7.     output reg[`RegBus]         new_pc,
8.     output reg                  flush,
9.     output reg[5:0]             stall
10.    );
```

III. EX 段：

集成计算/执行器 EX：负责处理 ID 段送到执行段的信息，对计算指令（非乘除指令）充当 ALU 计算单元的作用，同时整体给出后续单元对 HILO 寄存器、CP0 寄存器和转移状态、转移地址的控制。同时，EX 处理器还负责检测后续状态中存在的数据相关问题，并进行相应处理。

模块定义如下：

```
1. module ex(
2.     input wire                   rst,
3.     //送到执行阶段的信息
4.     input wire[`AluOpBus]        aluop_i,
5.     input wire[`AluSelBus]       alusel_i,
6.     input wire[`RegBus]          reg1_i,
7.     input wire[`RegBus]          reg2_i,
8.     input wire[`RegAddrBus]      wd_i,
9.     input wire                   wreg_i,
10.     input wire[`RegBus]          inst_i,
11.     input wire[31:0]             excepttype_i,
12.     input wire[`RegBus]          current_inst_address_i,
13.     //HI、LO 寄存器的值
```

```
14.     input wire[`RegBus]           hi_i,
15.     input wire[`RegBus]           lo_i,
16.     //回写阶段的指令是否要写 HI、LO，用于检测 HI、LO 的数据相关
17.     input wire[`RegBus]           wb_hi_i,
18.     input wire[`RegBus]           wb_lo_i,
19.     input wire                    wb_whilo_i,
20.     //访存阶段的指令是否要写 HI、LO，用于检测 HI、LO 的数据相关
21.     input wire[`RegBus]           mem_hi_i,
22.     input wire[`RegBus]           mem_lo_i,
23.     input wire                    mem_whilo_i,
24.     input wire[`DoubleRegBus]     hilo_temp_i,
25.     input wire[1:0]               cnt_i,
26.     //与除法模块相连
27.     input wire[`DoubleRegBus]     div_ret_i,
28.     input wire                    div_ready_i,
29.     //是否转移、以及 link address
30.     input wire[`RegBus]           link_address_i,
31.     input wire                    is_in_delayslot_i,
32.     //访存阶段的指令是否要写 CP0，用来检测数据相关
33.     input wire                    mem_cp0_reg_we,
34.     input wire[4:0]               mem_cp0_reg_write_addr,
35.     input wire[`RegBus]           mem_cp0_reg_data,
36.     //回写阶段的指令是否要写 CP0，用来检测数据相关
37.     input wire                    wb_cp0_reg_we,
38.     input wire[4:0]               wb_cp0_reg_write_addr,
39.     input wire[`RegBus]           wb_cp0_reg_data,
40.     //与 CP0 相连，读取其中 CP0 寄存器的值
41.     input wire[`RegBus]           cp0_reg_data_i,
42.     output reg[4:0]               cp0_reg_read_addr_o,
43.     //向下一流水级传递，用于写 CP0 中的寄存器
44.     output reg                    cp0_reg_we_o,
45.     output reg[4:0]               cp0_reg_write_addr_o,
46.     output reg[`RegBus]           cp0_reg_data_o,
47.     output reg[`RegAddrBus]       wd_o,
48.     output reg                    wreg_o,
49.     output reg[`RegBus]           wdata_o,
50.     output reg[`RegBus]           hi_o,
51.     output reg[`RegBus]           lo_o,
52.     output reg                    whilo_o,
53.     output reg[`DoubleRegBus]     hilo_temp_o,
54.     output reg[1:0]               cnt_o,
55.     output reg[`RegBus]           div_opdata1_o,
56.     output reg[`RegBus]           div_opdata2_o,
57.     output reg                    div_start_o,
58.     output reg                    signed_div_o,
59.     //加载、存储指令输出
60.     output wire[`AluOpBus]        aluop_o,
61.     output wire[`RegBus]          mem_addr_o,
```

装 订 线

```
62.     output wire[`RegBus]          reg2_o,
63.     output wire[31:0]             excepttype_o,
64.     output wire                   is_in_delayslot_o,
65.     output wire[`RegBus]          current_inst_address_o,
66.     output reg                    stallreq
67. );
```

除法器 DIV：直接使用的之前已经实现的除法器，功能包含实现有符号数和无符号数的除法运算。

模块设计如下：

```
1. module div(
2.     input wire                  clk,
3.     input wire                  rst,
4.     input wire                  signed_div_i,
5.     input wire[31:0]            opdata1_i,
6.     input wire[31:0]            opdata2_i,
7.     input wire                  start_i,
8.     input wire                  annul_i,
9.     output reg[63:0]            ret_o,
10.     output reg                  ready_o
11.    );
```

IV. MEM 段：

访存控制器 MEM：接受并处理 EX 段输入的控制命令，对于只流经 MEM 段且只对 WB 段有影响的数据通路直接原样送出，对其中影响 MEM 操作的数据通路进行拦截并进行处理，实现对各类读写数据存储器操作的指令控制。

```
1. module mem(
2.     input wire                  rst,
3.     //段存储器的输入，只流过
4.     input wire[`RegAddrBus]     wd_i,
5.     input wire                  wreg_i,
6.     input wire[`RegBus]         wdata_i,
7.     input wire[`RegBus]         hi_i,
8.     input wire[`RegBus]         lo_i,
9.     input wire                  whilo_i,
10.     input wire[`AluOpBus]       aluop_i,
11.     input wire[`RegBus]         mem_addr_i,
12.     input wire[`RegBus]         reg2_i,
13.     input wire[`RegBus]         mem_data_i,
14.     //分支转移预测输入
15.     input wire                  LLbit_i,
16.     input wire                  wb_LLbit_we_i,
```

```
    input wire                  wb_LLbit_value_i,
    //异常处理类输入
    input wire                  cp0_reg_we_i,
    input wire[4:0]             cp0_reg_write_addr_i,
    input wire[`RegBus]         cp0_reg_data_i,
    input wire[31:0]            excepttype_i,
    input wire                  is_in_delayslot_i,
    input wire[`RegBus]         current_inst_address_i,
    input wire[`RegBus]         cp0_status_i,
    input wire[`RegBus]         cp0_cause_i,
    input wire[`RegBus]         cp0_epc_i,
    //WB 段操作输入
    input wire                  wb_cp0_reg_we,
    input wire[4:0]             wb_cp0_reg_write_addr,
    input wire[`RegBus]         wb_cp0_reg_data,
    //WB 段操作输出
    output reg[`RegAddrBus]     wd_o,
    output reg                  wreg_o,
    output reg[`RegBus]         wdata_o,
    output reg[`RegBus]         hi_o,
    output reg[`RegBus]         lo_o,
    output reg                  whilo_o,
    output reg                  LLbit_we_o,
    output reg                  LLbit_value_o,
    output reg                  cp0_reg_we_o,
    output reg[4:0]             cp0_reg_write_addr_o,
    output reg[`RegBus]         cp0_reg_data_o,
    //对数据存储器的操作结果输出
    output reg[`RegBus]         mem_addr_o,
    output wire                 mem_we_o,
    output reg[3:0]             mem_sel_o,
    output reg[`RegBus]         mem_data_o,
    output reg                  mem_ce_o,
    output reg[31:0]            excepttype_o,
    output wire[`RegBus]         cp0_epc_o,
    output wire                 is_in_delayslot_o,
    output wire[`RegBus]        current_inst_address_o
);
```

数据存储器 DMEM：负责进行数据存取控制。

具体模块设计如下：

```
module dmem(
    input wire  clk,
    input wire  ce,
    input wire  we,
    input wire[`DataAddrBus] addr_in,
```

```
    input wire[3:0] sel,
    input wire[`DataBus] data_i,
    output reg[`DataBus] data_o,
    output wire[`DataBus] ret
);
```

V. WB 段：

LLBIT 预测器：实现 2bit 分支预测。

具体模块设计如下：

```
module LLbit_reg(
    input wire clk,
    input wire rst,
    input wire flush,
    input wire LLbit_i,
    input wire we,
    output reg LLbit_o
);
```

64 位拓展指令数据存储器 HILO：分为高位 HI 和低位 LO 两个存储器，工作原理与寄存器堆 REGS 相同。

具体模块设计如下：

```
module hilo_reg(
    input wire                      clk,
    input wire                      rst,
    input wire                      we,
    input wire[`RegBus]             hi_i,
    input wire[`RegBus]             lo_i,
    output reg[`RegBus]             hi_o,
    output reg[`RegBus]             lo_o
);
```

协处理器 CP0：具有 32 个协处理寄存器，负责对异常处理、系统调用和中断过程中必须保存的 CPU 现场和一些支持信息进行存储以供之后操作系统开发进行调用。

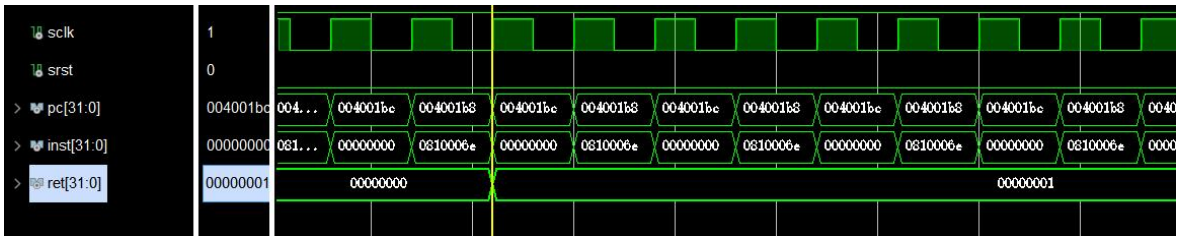
具体模块设计如下：

```
module cp0_reg(
    input wire                  clk,
```

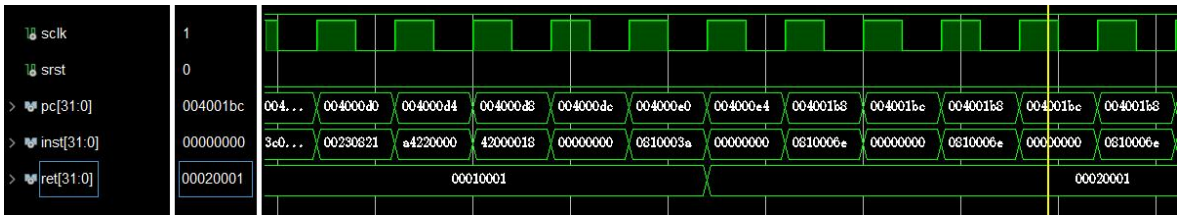
```
3.    input wire                    rst,
4.    input wire                    we_i,
5.    input wire[4:0]               waddr_i,
6.    input wire[4:0]               raddr_i,
7.    input wire[`RegBus]           data_i,
8.    input wire[31:0]              excepttype_i,
9.    input wire[5:0]               int_i,
10.    input wire[`RegBus]           current_inst_addr_i,
11.    input wire                    is_in_delayslot_i,
12.    output reg[`RegBus]           data_o,
13.    output reg[`RegBus]           count_o,
14.    output reg[`RegBus]           compare_o,
15.    output reg[`RegBus]           status_o,
16.    output reg[`RegBus]           cause_o,
17.    output reg[`RegBus]           epc_o,
18.    output reg[`RegBus]           config_o,
19.    output reg[`RegBus]           prid_o,
20.    output reg                    timer_int_o
21.   );
```

3、仿真波形验证：

下板程序直接采用实验下板给出的测试程序。



10 段计算测试点均顺利通过，低地址部分成功显示 1。



高地址部分随着时钟中断不断计数，代表着本次试验设计的 CPU 通过仿真测试，所有功能指令均正常。

4、时序验证

时序验证如下：

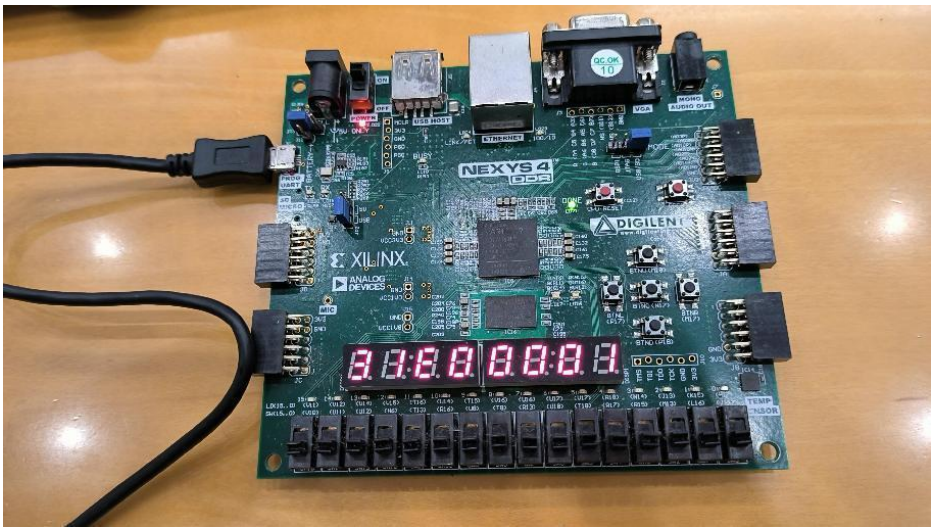
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.172 ns	Worst Hold Slack (WHS):	0.140 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	132	Total Number of Endpoints:	132	Total Number of Endpoints:	67

All user specified timing constraints are met.

在 10ns 的时序约束下，Worst Negative Slack 为 6.172ns，Worst Hold Slack (WHS)为 0.140 ns，Worst Pulse Width Slack (WPWS)为 4.500 ns，都满足了时序要求，这代表本次实验设计的 CPU 在时序约束下不会发生功能性问题。

4、实验结果

下板采用的 CPU 频率为 50Mhz，实验结果如下：



可以看到数码管低半字显示 0x0001，高半字随着时钟中断而计数，验证结果正确，这证明了本次试验设计的 CPU 可以在 50Mhz 下平稳运行，在完成 54 条的前提下很好地完成了 HILO 交互、拓展内存读写、CP0 读写在内的一系列 10 项任务和中断处理任务，为之后的系统移植打下坚实的基础。

5、实验总结

实验的开发过程在《自己动手写 CPU 中》介绍的十分详尽，对移动操作、算术操作、转移、加载存储和异常相关指令的开发过程都进行了详尽的介绍，对于重点、

难点也都给出了详尽的代码，让我有所参考有所学习，因而没有出现过多的问题。在仿真运行时倒是出现了一系列的未通过现象，最初我以为是代码逻辑问题，然而当我实际检查时发现并没有出现逻辑错误，部件内部也能够通过关联测试，最终发现是由于某些寄存器没有初始化导致本身呈现出高阻状态所导致的，进行初始化后这些问题都得到了修复，包括我本来以为会需要耗费大量时间调试的中断处理系统也在下板测试中一边通过。

本次试验是计算机系统实验课程的第一次实验，也是系统结构中整体硬件部分的最终设计。回首往昔，从最初的数字逻辑课程的 FPGA 实验到计算机组成原理的 31 条指令单周期 CPU 设计、54 条指令单周期 CPU 设计，再到系统结构课上的 54 条 CPU 五级分段流水线的设计，最终到本次 89 条五级分段流水线的设计，我对 CPU 的理解在这两年间得到了不断的锤炼和增强，当我在开发板上看到七段数码管随着时钟中断在不断跳动的时候我发自内心感受到一股暖意在汇聚，那是两年来对于硬件的不断学习所带来的热火和勇气。

下一步我将着手进行系统移植的开发，为本门课程也是硬件系列课程的结束交出一份完美的答卷。