同济大学
计算机科学与技术系

计算机组成原理课程实验报告

学　　号 **2151769**

姓　　名 吕博文

专　　业 计算机科学与技术

授课老师 陈永生

日　　期 **2023.7.6**

一、实验目标

本次实验要求通过 Verilog 语言设计实现 MIPS54 条 cpu 的功能，这里我选择了单周期 54 条 cpu 的实现与仿真。

二、总体设计

1、 设计思路：

CPU 主要由控制器和运算器组成，其中运算器 ALU 实现较为简单，在之前的实验中也多次涉及到了，主要在于控制器的编写，控制器需要实现的功能是根据 PC 取指令，对指令进行译码分析后得到相应的指令信号，控制运算器的运算，寄存器堆的读写，内存的读写等等。

编写过程主要分为以下几大模块：

（1） Sccomp_dataflow:顶层模块，负责调用 cpu 模块产生控制信号，调用内存模块控制读写，调用 PC 模块负责更新 PC 寄存器的值。

（2） PC 模块：负责更新 pc 的值，单独将 PC 模块分离出来是为了整体结构设计更加优化。

（3） IMEM 模块：实例化 IP 核，产生指令存储器。

（4） DMEM 模块：自定义内存模块，负责内存读写。

（5） CPU 模块：核心模块，负责调用控制器模块产生控制信号，调用寄存器堆模块控制寄存器读写，调用 ALU 模块控制运算，调用 MDU 模块控制乘除法运算，调用 CP0 模块控制中断。

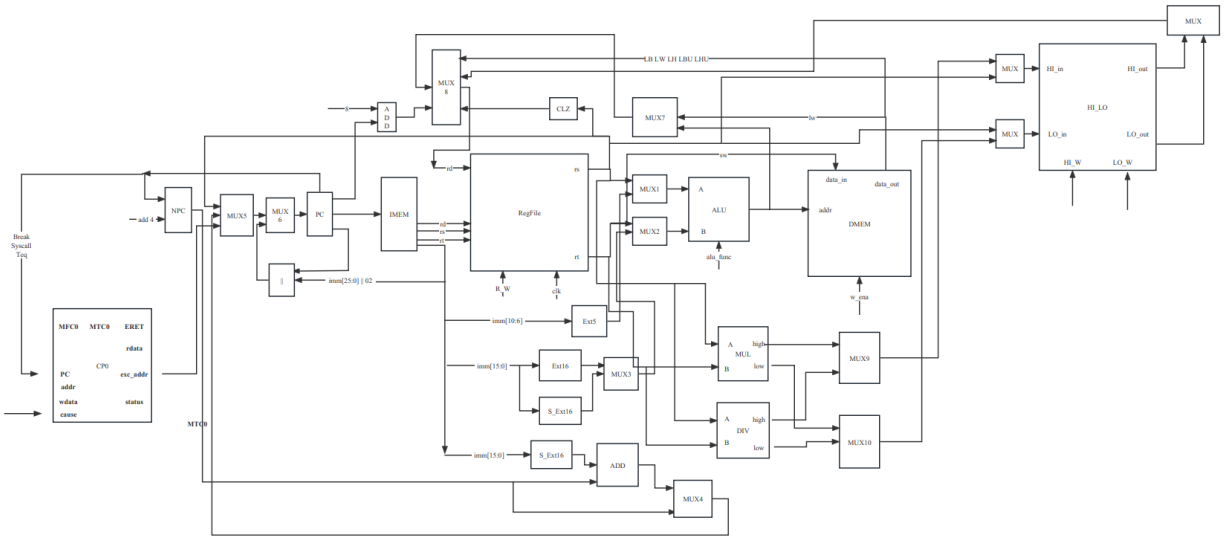
（6） ALU 模块：完成运算

（7） MDU 模块：自定义的模块，将乘除法封装为一个模块，便于一起处理。

（8） RegFile 模块：寄存器堆模块，控制寄存器读写

（9） CP0 模块：控制特殊指令，中断。

2、 总数据数据通路图：



3、54 条指令数据通路设计

R 型指令

1、ADD

指令流程:

PC →IMEM

PC+4→NPC

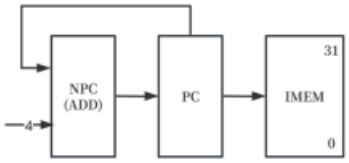
NPC→PC

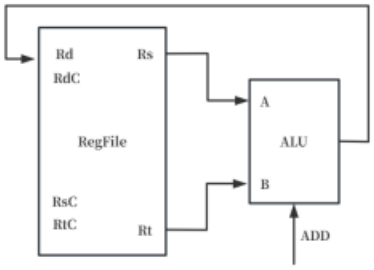
Rs→A , Rt →B

(A + B → ANS)

ANS→Rd

指令流程图：





2、ADDU

指令流程：

PC→IMEM

PC + 4→NPC

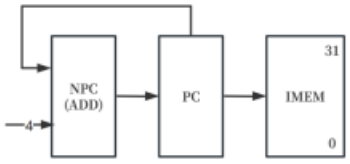
NPC→PC

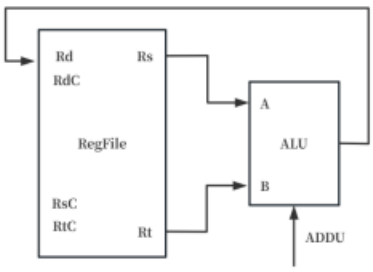
Rs→A, Rt→B

(A + B →ANS)

ANS→Rd

指令流程图：





3、SUB

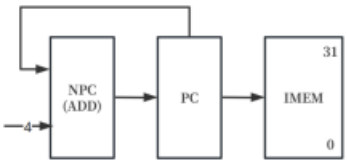
指令流程：

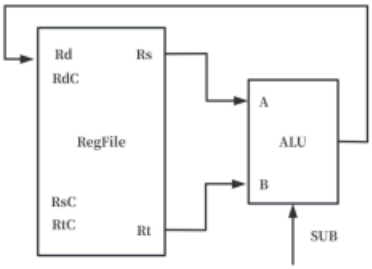
PC→IMEM

PC+4→NPC
NPC→PC

Rs→A, Rt→B
(A-B→ANS)
ANS→Rd

指令流程图：





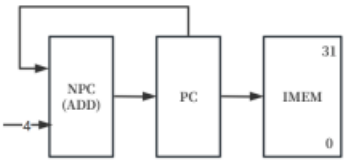
4、SUBU

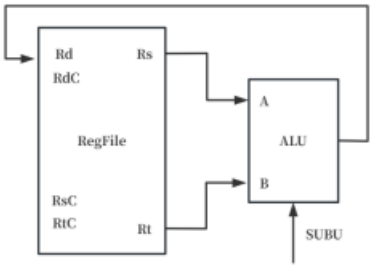
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

Rs→A, Rt→B
A- B→ANS
ANS→Rd

指令流程图：





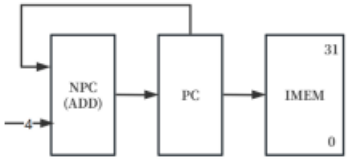
5、AND

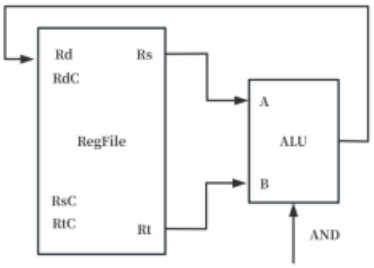
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

Rs→A, Rt→B
A&B→ANS
ANS→Rd

指令流程图：





6、OR

指令流程：

PC→IMEM

PC+4→NPC

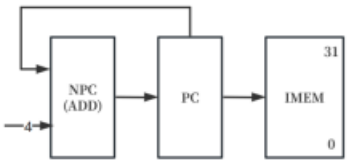
NPC→PC

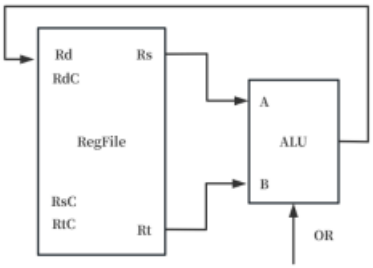
Rs→A, Rt→B

A|B→ANS

ANS→Rd

指令流程图：





7、XOR

指令流程：

PC→IMEM

PC+4→NPC

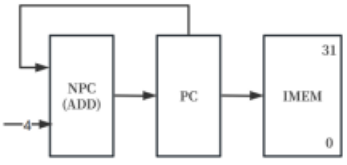
NPC→PC

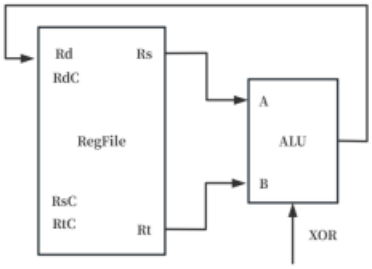
Rs→A , Rt→B

A xor B→ANS

ANS→Rd

指令流程图：





8、NOR

指令流程：

PC→IMEM

PC+4→NPC

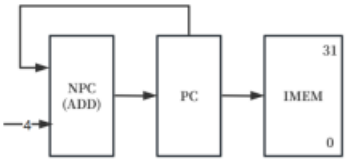
NPC→PC

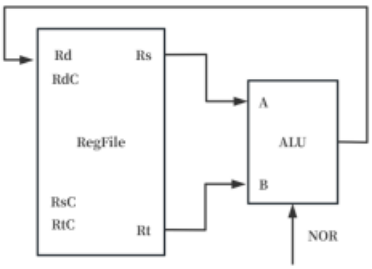
Rs→A, Rt→B

A nor B →ANS

ANS→Rd

指令流程图：





9、SLT

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

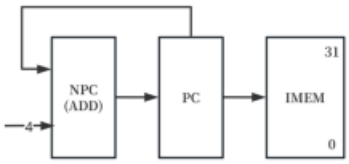
Rs→A,　Rt→B

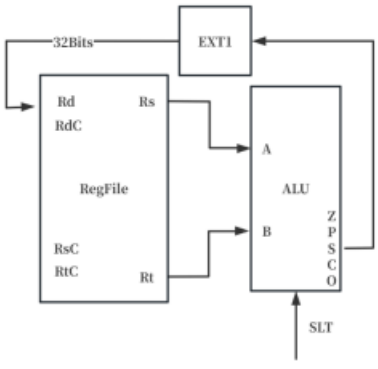
A – B →ANS

SF→EXT1

EXT1_OUT →Rd

指令流程图：





10、　SLTU

指令流程：

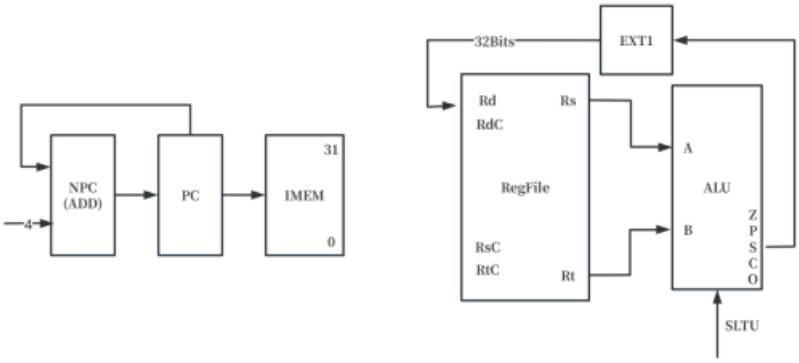
PC→IMEM

PC+4→NPC

NPC→PC

Rs→A, Rt→B
A – B →ANS
SF→EXT1
EXT1_OUT→Rd

指令流程图：



11、 SLL

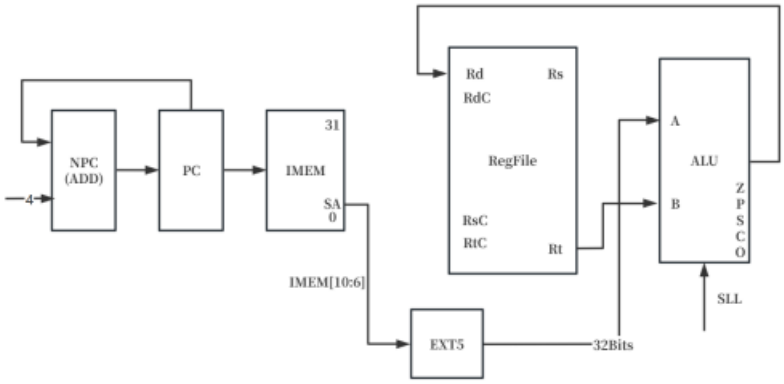
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[10:6] →EXT5
EXT5_OUT→A

Rt→B
B<<A →ANS
ANS→Rd

指令流程图：



12、 SRL

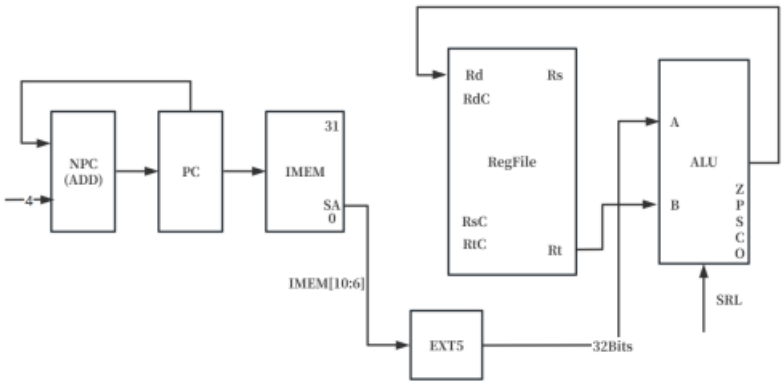
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[10:6] →EXT5
EXT5_OUT→A

Rt→B
B>>A →ANS
ANS→Rd

指令流程图：



13、 SRA

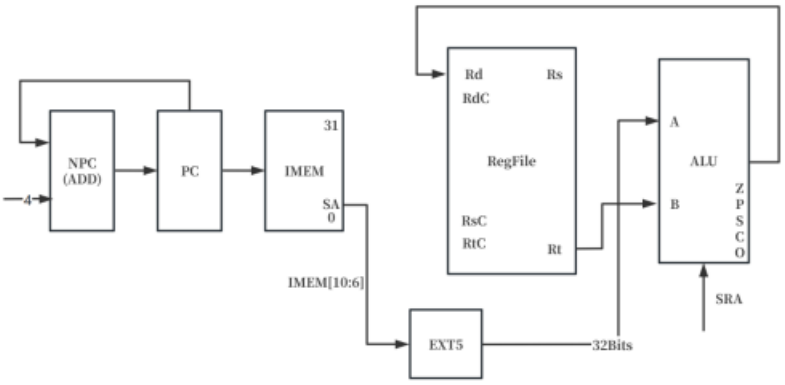
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[10:6] →EXT5
EXT5_OUT→A

Rt→B
B>>A →ANS
ANS→Rd

指令流程图：



14、 SLLV

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

Rs[4:0]→EXT5

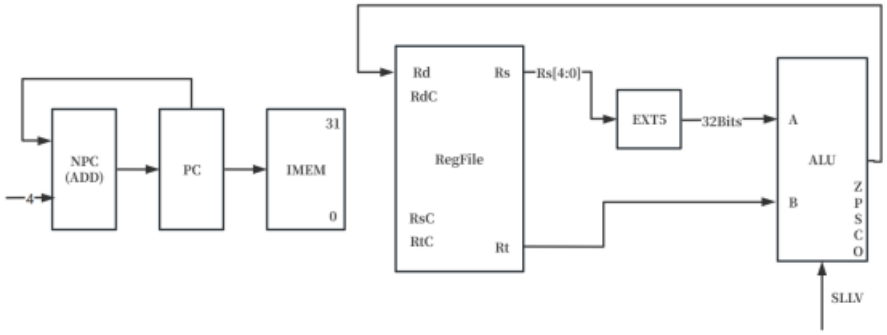
EXT5_OUT→A

Rt→B

B<<A →ANS

ANS→Rd

指令流程图：



15、 SRLV

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

Rs[4:0]→EXT5

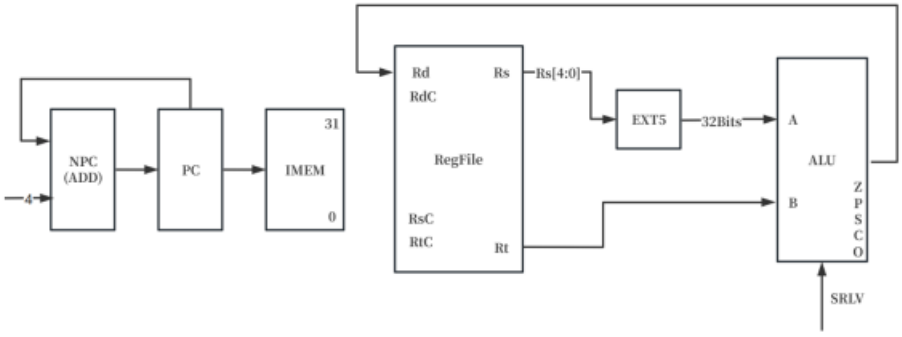
EXT5_OUT→A

Rt→B

B>>A →ANS

ANS→Rd

指令流程图：



16、 SRAV

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

Rs[4:0]→EXT5

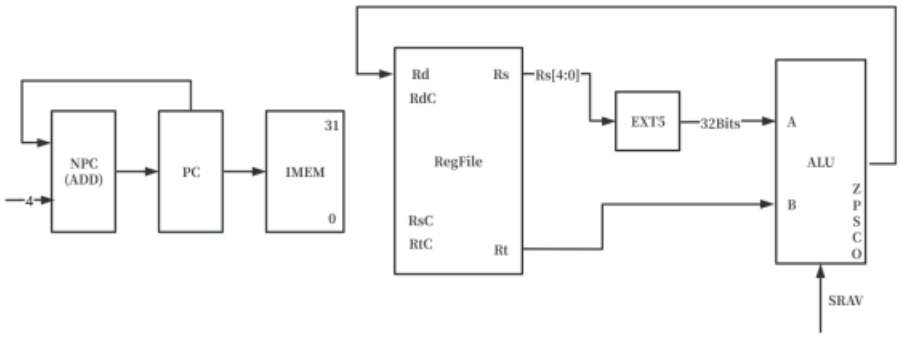
EXT5_OUT→A

Rt→B

B>>A →ANS

ANS→Rd

指令流程图：



17、 JR

指令流程：

PC→IMEM

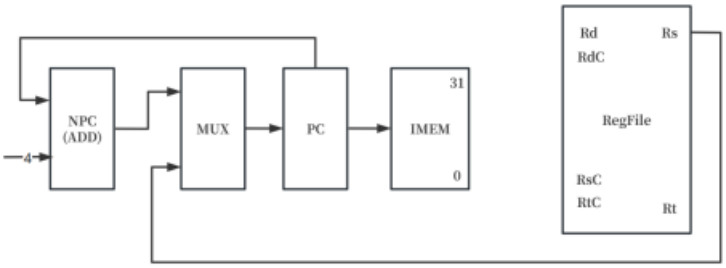
PC+4→NPC

Rs→MUX

MUX_OUT→PC

NPC→MUX

指令流程图：



I 型指令

18、 ADDI

指令流程：

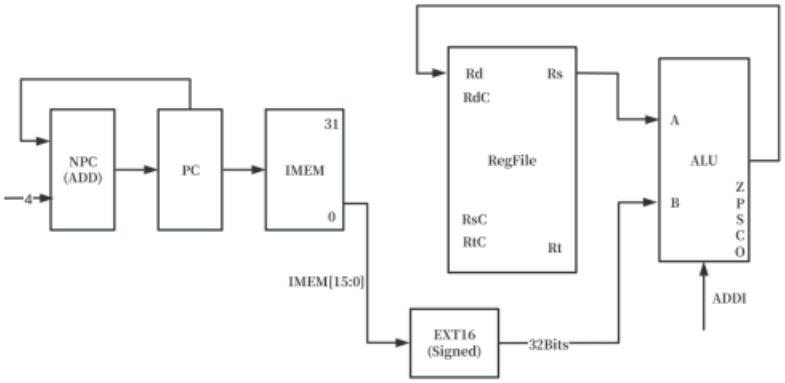
PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16
EXT16_OUT→B
Rs→A
A+B→ANS
ANS→Rd

指令流程图：



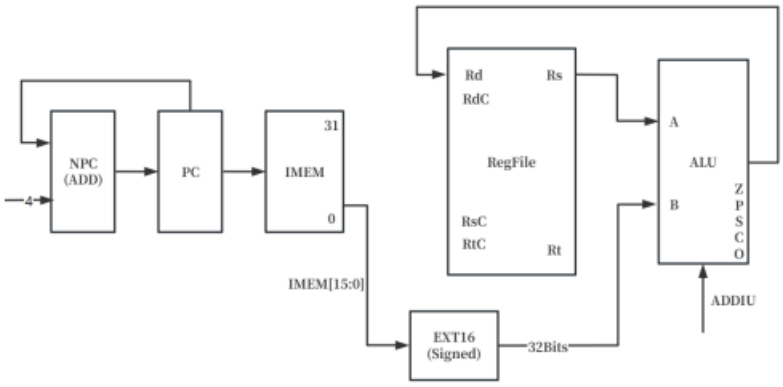
19、 ADDIU

指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[15:0]→EXT16
EXT16_OUT→B
Rs→A
A+B→ANS
ANS→Rd

指令流程图：



20、 ANDI

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

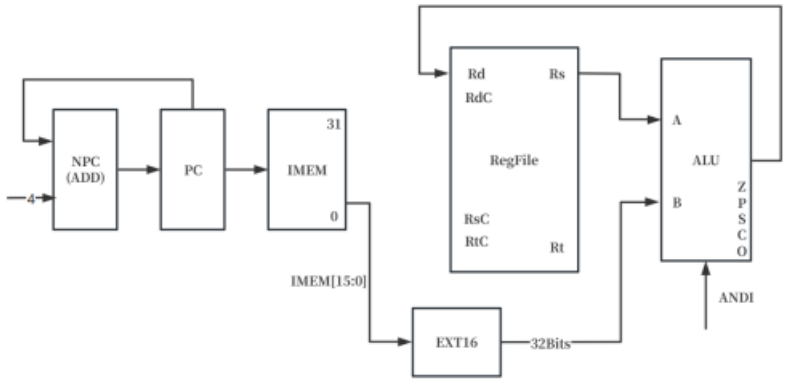
EXT16_OUT→B

Rs→A

A&B→ANS

ANS→Rd

指令流程图：



21、 ORI

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

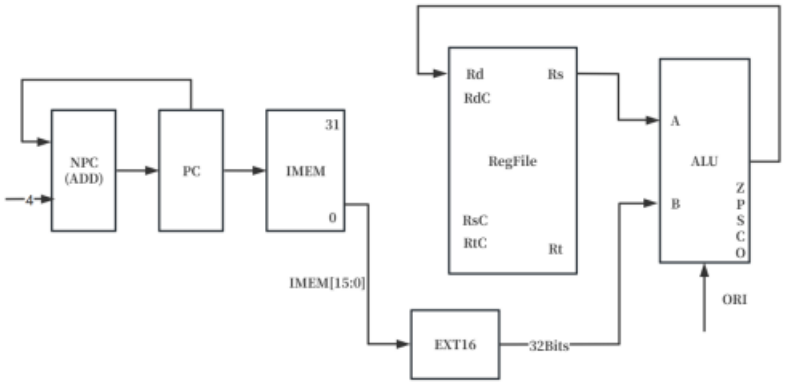
EXT16_OUT→B

Rs→A

A or B→ANS

ANS→Rd

指令流程图：



22、 XORI

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

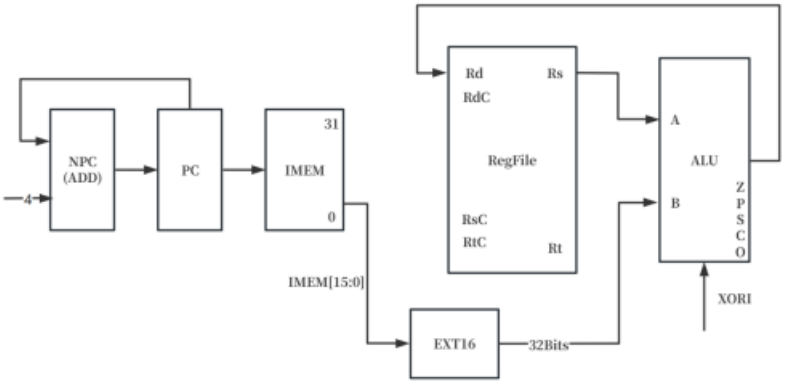
EXT16_OUT→B

Rs→A

A xor B→ANS

ANS→Rd

指令流程图：



23、 LW

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

EXT16_OUT→B

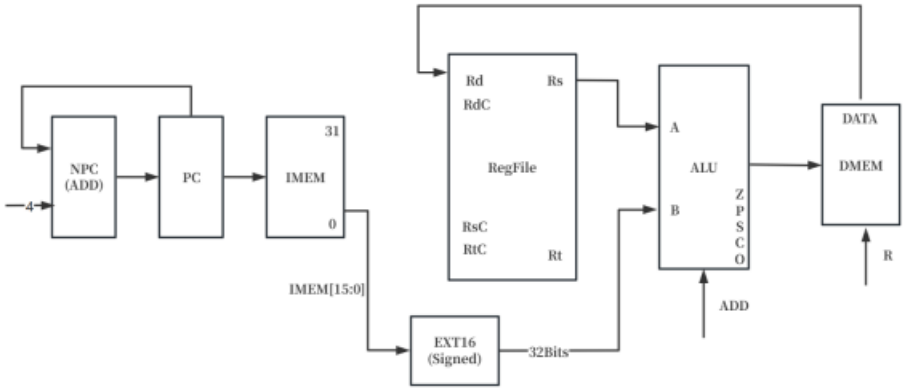
Rs→A

A + B→ANS

ANS→DMEM_ADDR

DMEM_OUT→Rd

指令流程图：



24、 SW

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

EXT16_OUT→B

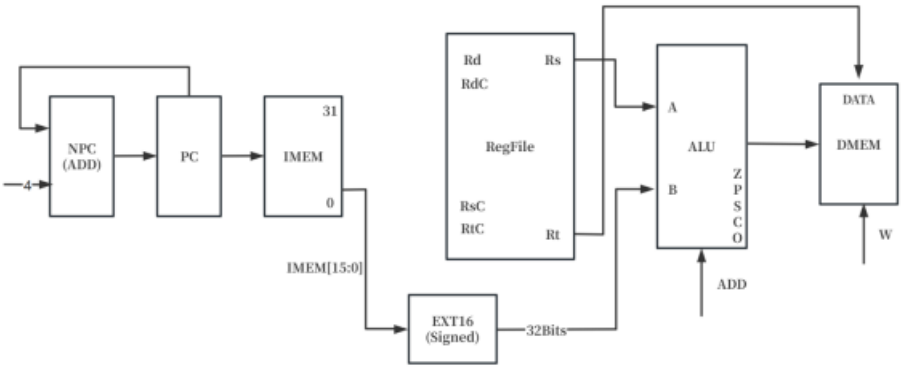
Rs→A

A + B→ANS

Rt→DMEM

ANS→DMEM_ADDR

指令流程图：



25、 BEQ

指令流程：

PC→IMEM

PC+4→NPC

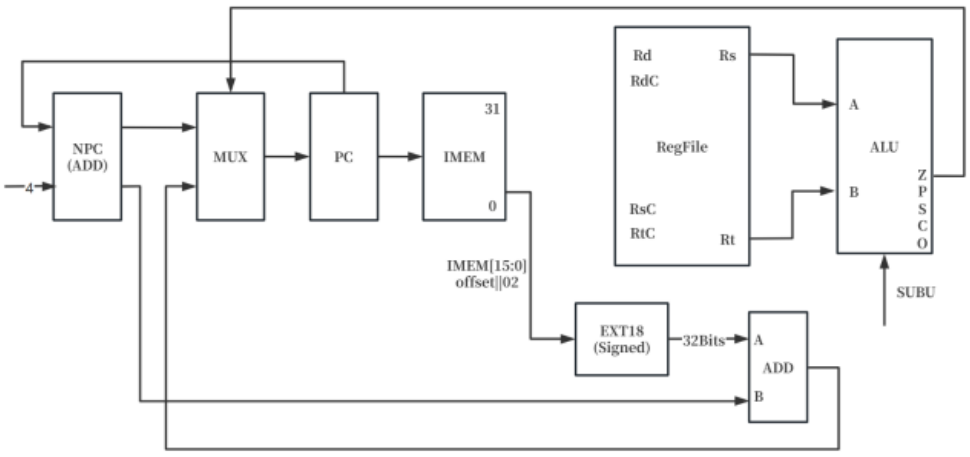
NPC→PC

IMEM[15:0] || 02→EXT18

EXT18_OUT→ADD_A
NPC→ADD_B
ADD_A + ADD_B →ADD_OUT
ADD_OUT→MUX

Rs→A
Rt→B
A+B→ANS

指令流程图：



26、 BNE

指令流程：

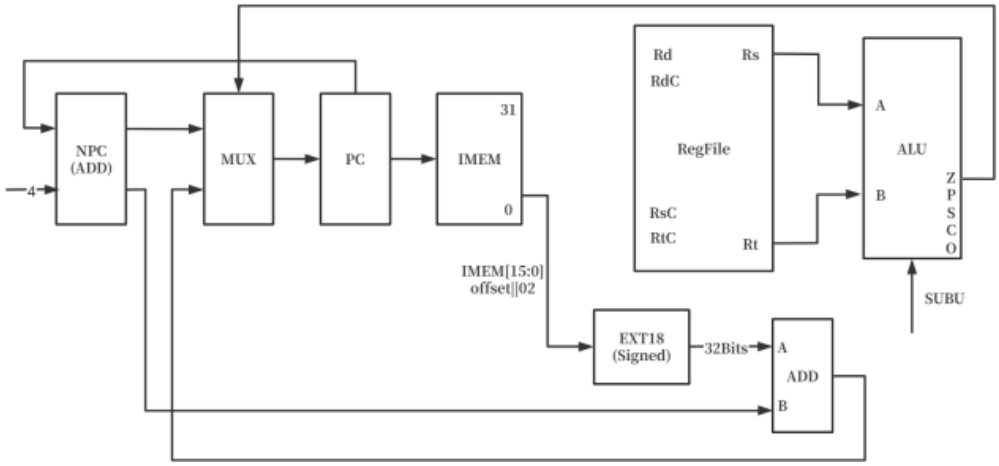
PC→IMEM
PC+4→NPC
NPC→PC

IMEM[15:0] || 02→EXT18
EXT18_OUT→ADD_A
NPC→ADD_B
ADD_A + ADD_B →ADD_OUT
ADD_OUT→MUX

Rs→A
Rt→B
A+B→ANS
Z→MUX

MUX→PC

指令流程图：



27、 SLTI

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16

EXT16_OUT→B

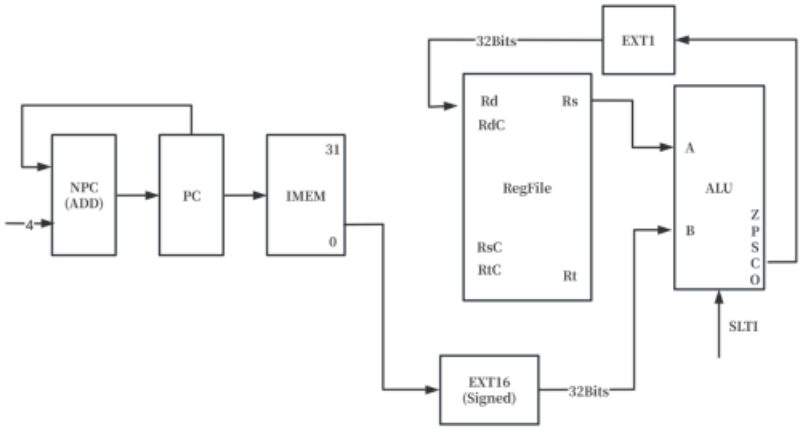
Rs→A

A – B →ANS

CF →EXT1

EXT1_OUT→Rd

指令流程图：



28、 SLTIU

指令流程：

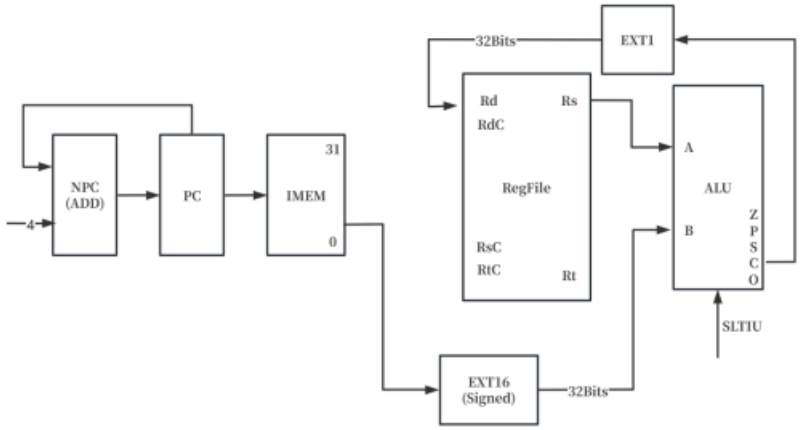
PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16
EXT16_OUT→B
Rs→A
A – B →ANS
CF →EXT1
EXT1_OUT→Rd

指令流程图：



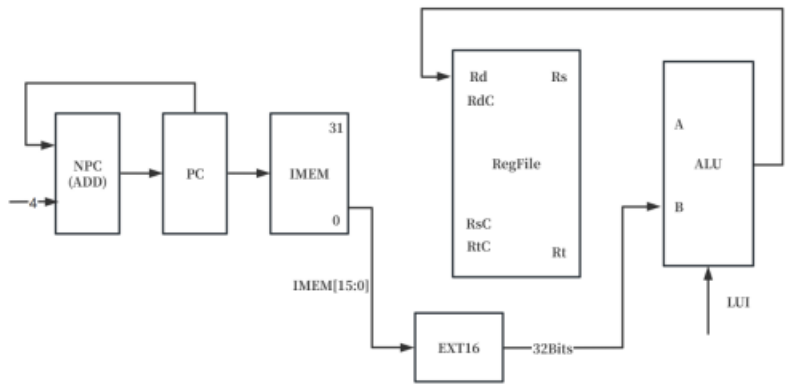
29、 LUI

指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[15:0]→EXT16
EXT16_OUT→B
ANS→Rd

指令流程图：



J 型指令

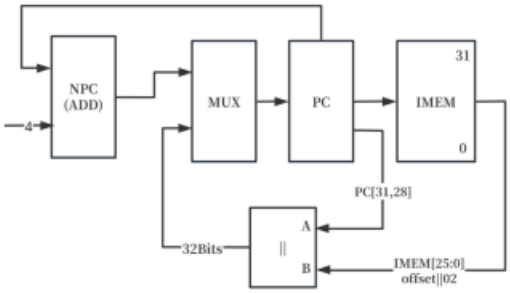
30、 J

指令流程：

PC→IMEM
PC+4→NPC
NPC→MUX

PC[31:28]→ || A
IMEM[25:0] ||02 →|| B
|| OUT →MUX
MUX_OUT→PC

指令流程图：



31、 JAL

指令流程：

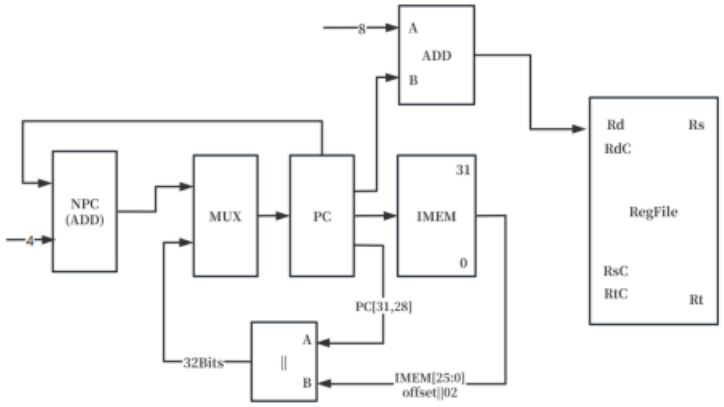
PC→IMEM
PC+4→NPC
NPC→MUX

8→ADD_A
PC→ADD_B
ADD_A + ADD_B → ADD_OUT
ADD_OUT →Rd

PC[31:28] → || A
IMEM[25:0] || 02 →|| B
|| OUT→MUX

MUX_OUT →PC

指令流程图：



32、 CLZ

指令流程：

PC→IMEM

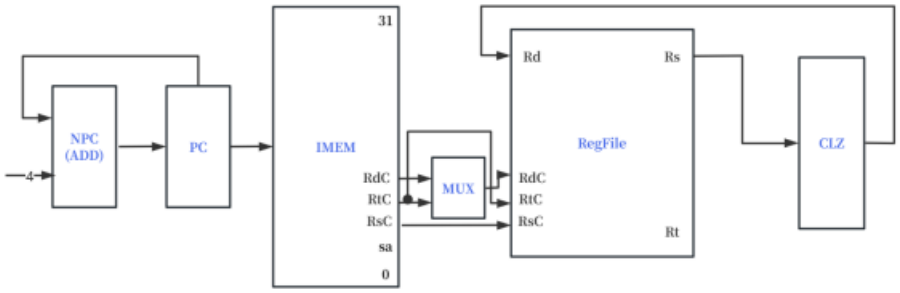
PC+4→NPC

NPC→MUX

Rs→CLZ_IN

CLZ_OUT→Rd

指令流程图：



33、 JALR

指令流程：

PC→IMEM

PC→ADD_A

ADD_A + ADD_B →ADD_OUT

PC+4→NPC

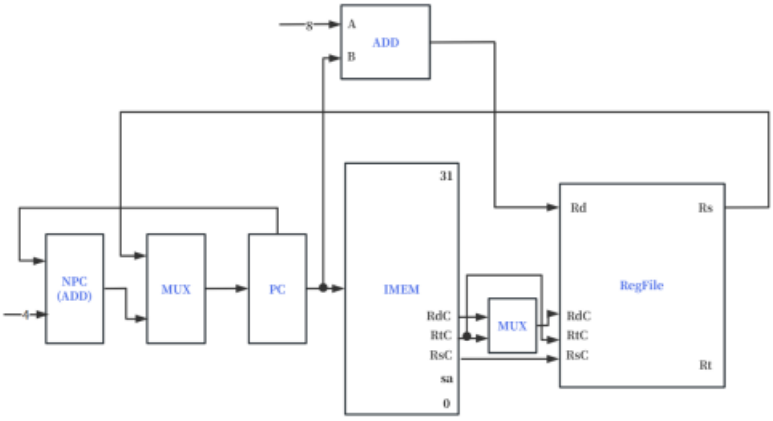
NPC →MUX

Rs→MUX

MUX(Rs) →PC

ADD_OUT→Rd

指令流程图：



34、 MTHI

指令流程：

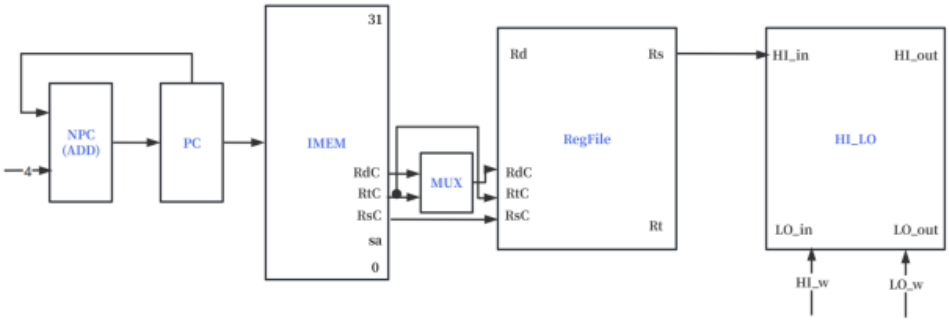
PC→IMEM

PC+4→NPC

NPC→PC

Rs→HI_IN(HI_W)

指令流程图：



35、 MTLO

指令流程：

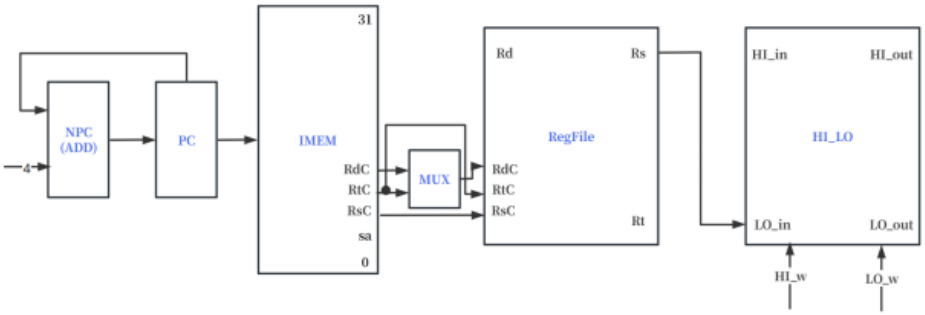
PC→IMEM

PC+4→NPC

NPC→PC

Rs→HI_IN(LO_W)

指令流程图：



36、 MFHI

指令流程：

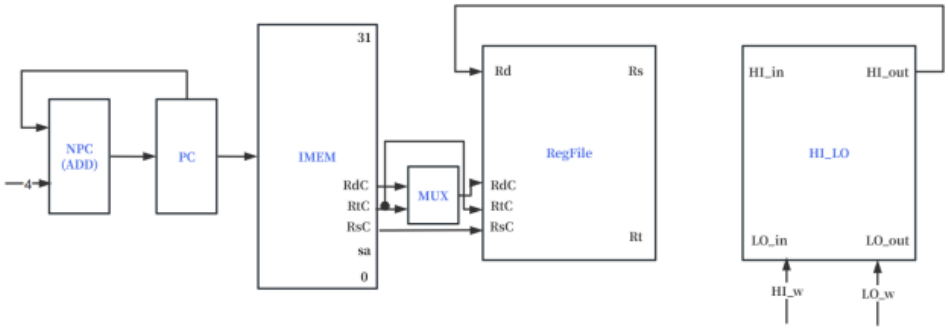
PC→IMEM

PC+4→NPC

NPC→PC

HI_OUT→Rd

指令流程图：



37、 MFLO

指令流程：

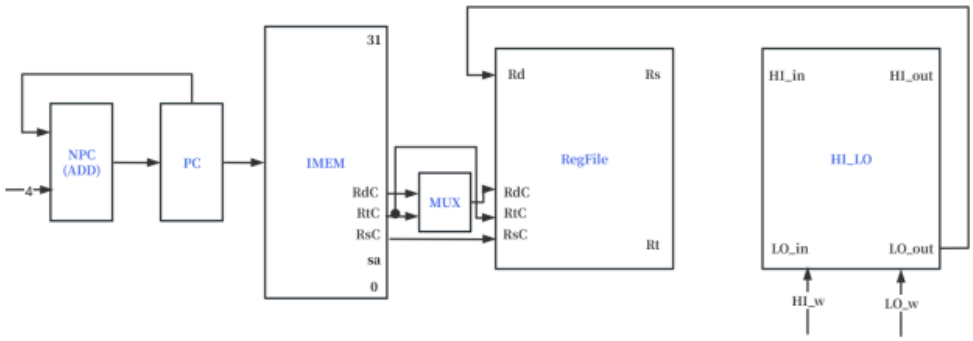
PC→IMEM

PC+4→NPC

NPC→PC

LO_OUT→Rd

指令流程图：



38、 SB

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0] →EXT16_IN

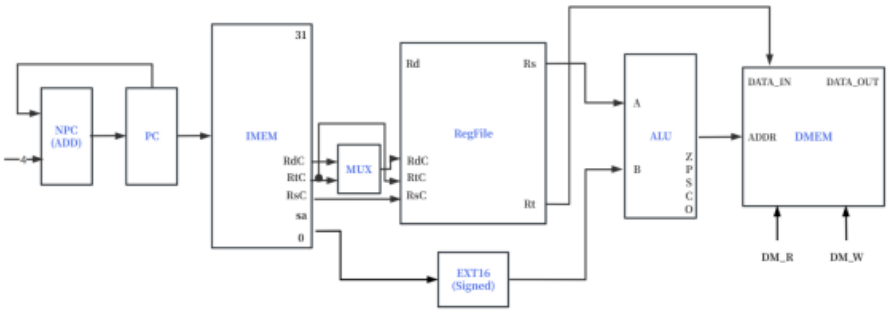
Rs →ALU_A

EXT16_OUT→ALU_B

ALU_OUT →DM_ADDR

Rt[7:0] →DATA_IN(DM_W)

指令流程图：



39、 SH

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0] →EXT16_IN

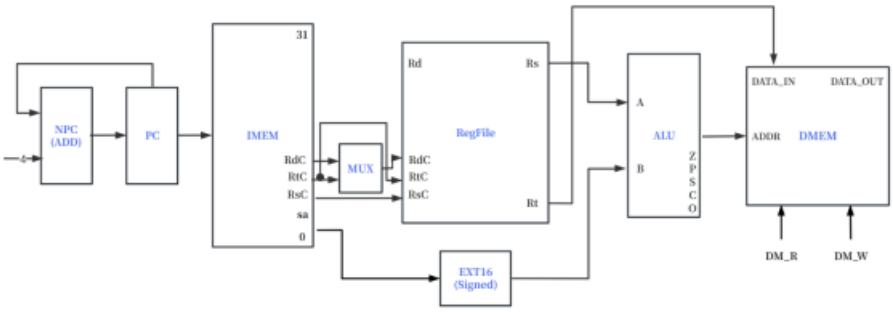
Rs →ALU_A

EXT16_OUT→ALU_B

ALU_OUT →DM_ADDR

Rt[15:0] →DATA_IN(DM_W)

指令流程图：



40、 LB

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

IMEM[15:0]→EXT16_IN

Rs→ALU_A

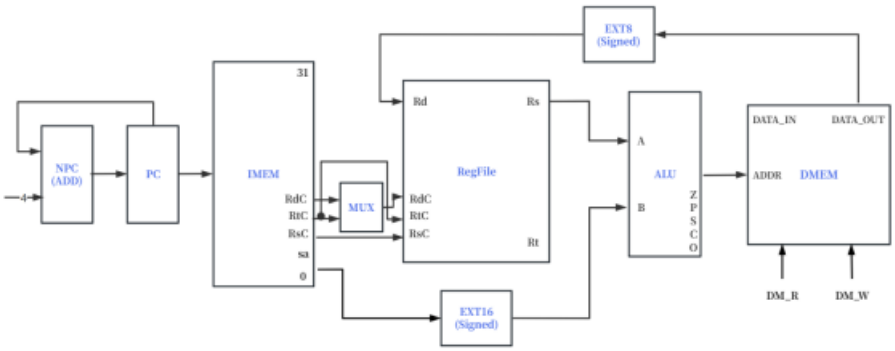
EXT16_OUT→ALU_B

ALU_OUT→DM_ADDR

DATA_OUT→EXT8_IN(DM_R)

EXT8_OUT →Rd

指令流程图：



41、 LH

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

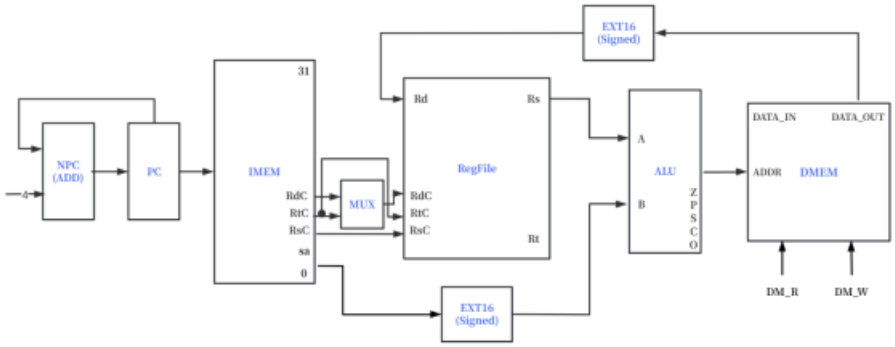
IMEM[15:0]→EXT16_IN

Rs→ALU_A

EXT16_OUT→ALU_B

ALU_OUT→DM_ADDR
DATA_OUT→EXT16_IN(DM_R)
EXT16_OUT →Rd

指令流程图：



42、 LBU

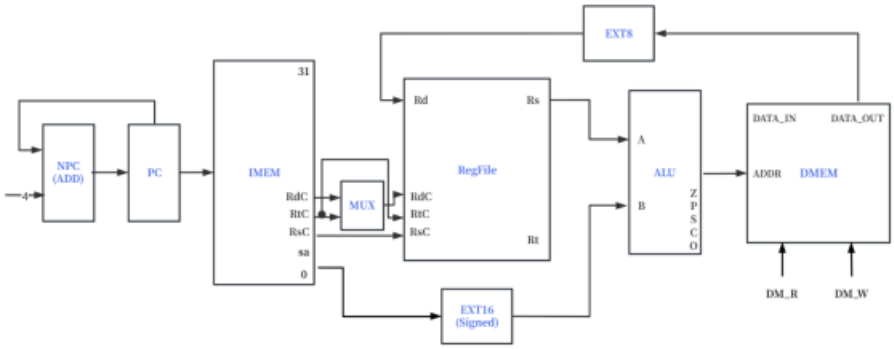
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[15:0]→EXT16_IN
Rs→ALU_A
EXT16_OUT→ALU_B

ALU_OUT→DM_ADDR
DATA_OUT→EXT8_IN(DM_R)
EXT8_OUT →Rd

指令流程图：



43、 LHU

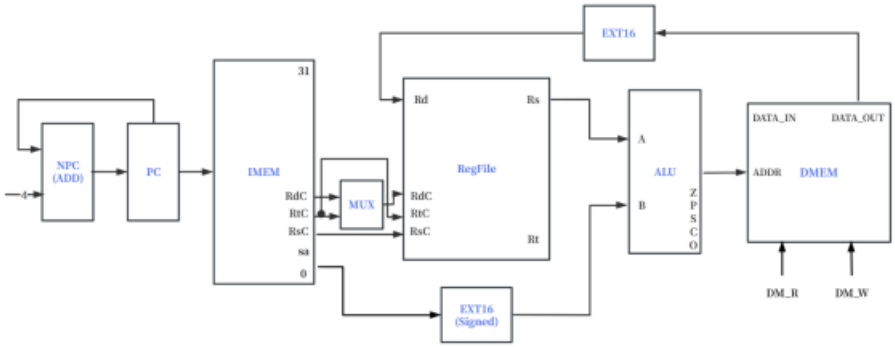
指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

IMEM[15:0]→EXT16_IN
Rs→ALU_A
EXT16_OUT→ALU_B

ALU_OUT→DM_ADDR
DATA_OUT→EXT16_IN(DM_R)
EXT16_OUT →Rd

指令流程图：



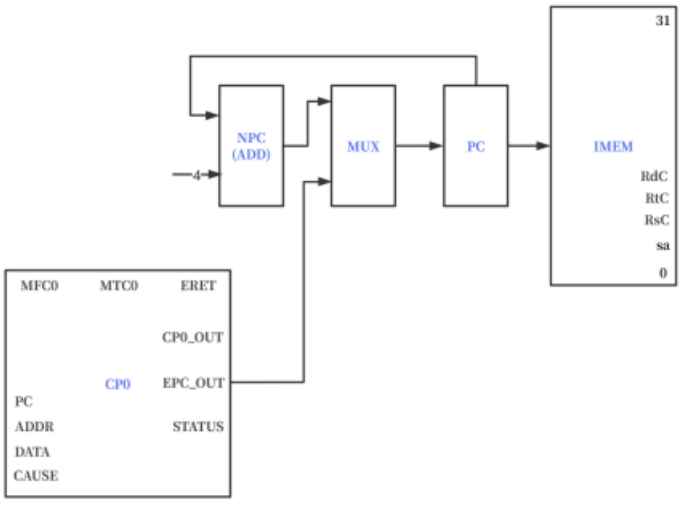
44、 ERET

指令流程：

PC→IMEM
PC+4→NPC
NPC→MUX

STATUS>>5 →STATUS
EPC_OUT→MUX
MUX(EPC_OUT)→PC

指令流程图：



45、 BREAK

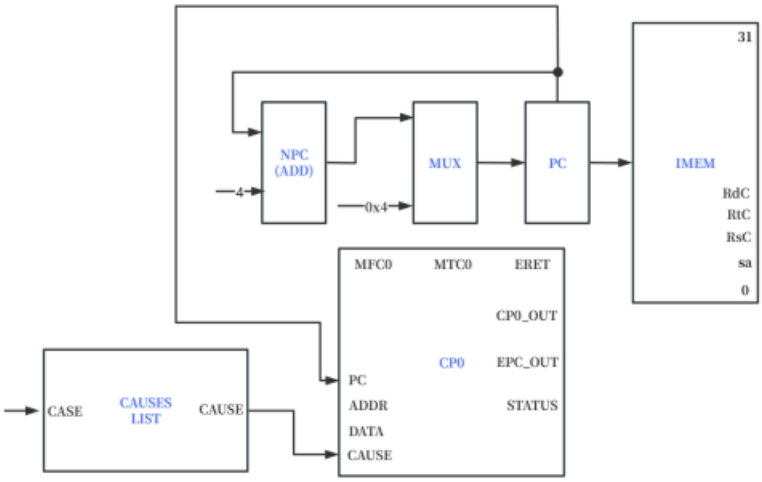
指令流程：

PC→IMEM
PC+4→NPC
NPC→MUX

PC→CP0
CAUSE-LIST_CAUSE →CP0_CAUSE
STATUS<<5 →STATUS

0X4→MUX
MUX(0X4)→PC

指令流程图：



46、 SYSCALL

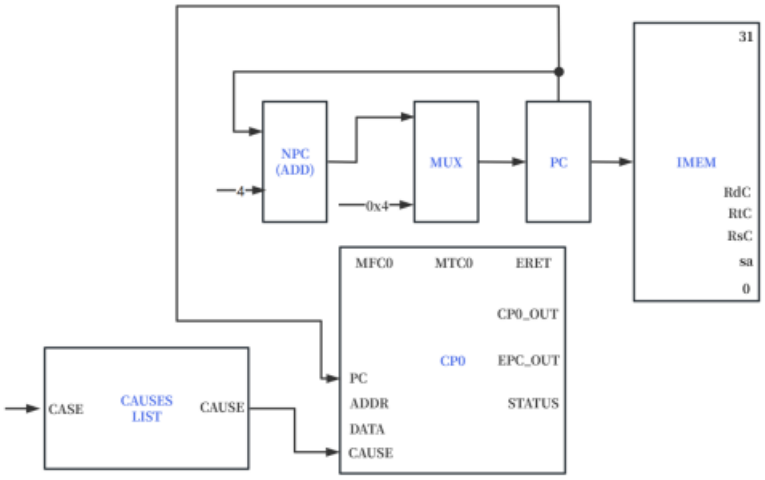
指令流程：

PC→IMEM
PC+4→NPC
NPC→MUX

PC→CP0
CAUSE-LIST_CAUSE →CP0_CAUSE
STATUS<<5 →STATUS

0X4→MUX
MUX(0X4)→PC

指令流程图：



47、 TEQ

指令流程：

PC→IMEM

PC+4→NPC

NPC→MUX

0X4→MUX

Rs→ALU_A

Rt→ALU_B

ZERO →Rs -Rt == 0

If ZERO:

PC→CP0

CAUSE-LIST_CAUSE →CP0_CAUSE

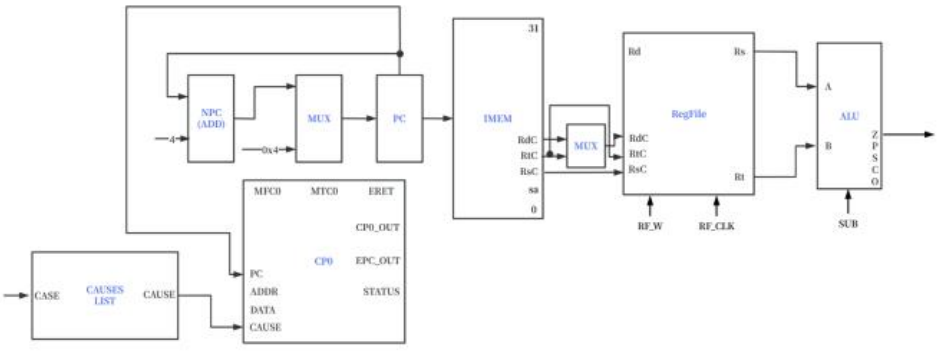
STATUS<<5 →STATUS

MUX(0X4) →PC

Else:

MUX(NPC) →PC

指令流程图：



48、 MFC0

指令流程：

PC→IMEM

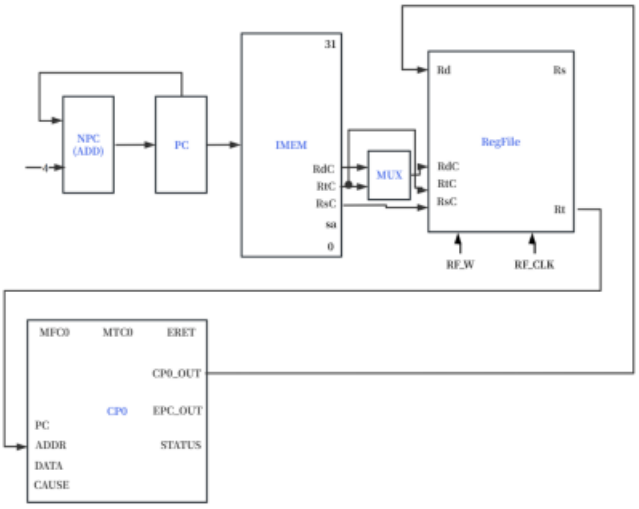
PC+4→NPC

NPC→PC

Rt→ADDR

CP0_OUT →Rd

指令流程图：



49、 MTC0

指令流程：

PC→IMEM

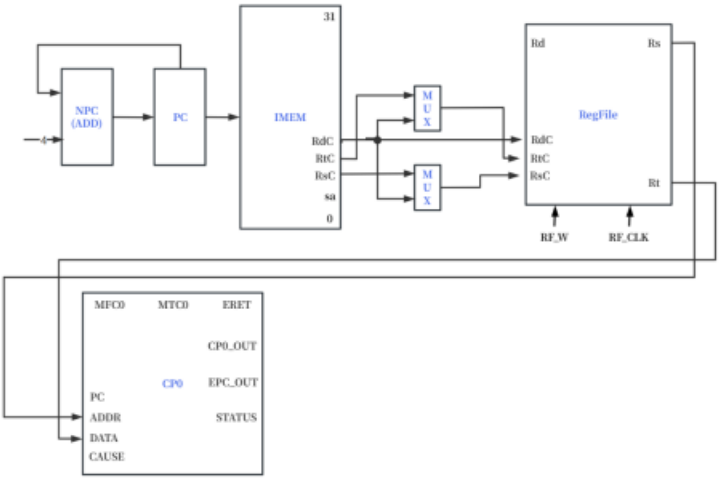
PC+4→NPC

NPC→PC

Rt→ADDR

Rs→CP0_DATA

指令流程图：



50、 MUL

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

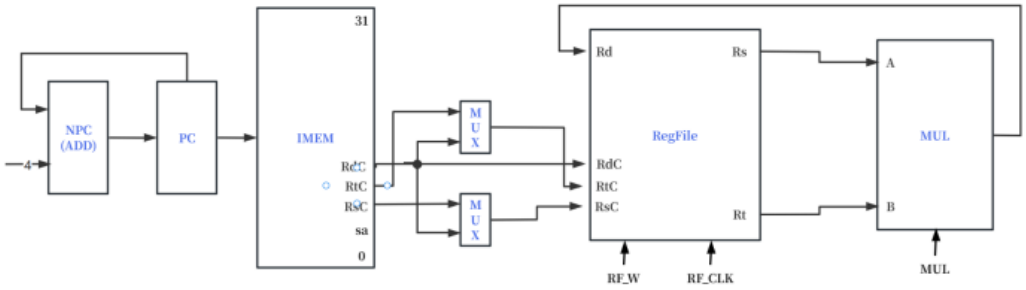
Rs →MUL_A

Rt→MUL_B

MUL_A*MUL_B →ANS

ANS[31:0] →Rd

指令流程图：



51、 MULTU

指令流程：

PC→IMEM

PC+4→NPC

NPC→PC

Rs →MUL_A

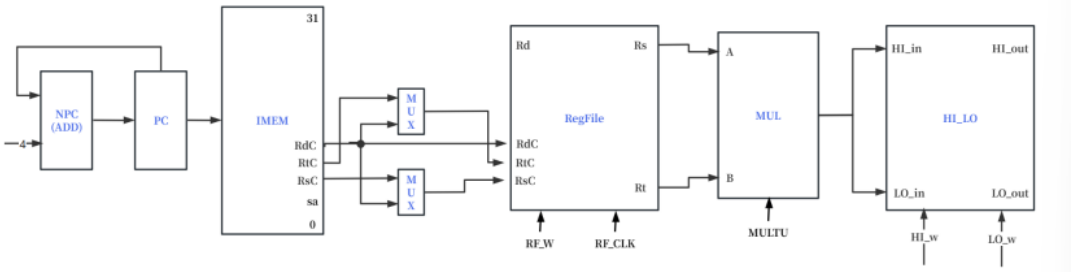
Rt→MUL_B

MUL_A*MUL_B →ANS

ANS[31:0]→LO_IN

ANS[63:32]→HI_IN

指令流程图：



52、 DIV

指令流程：

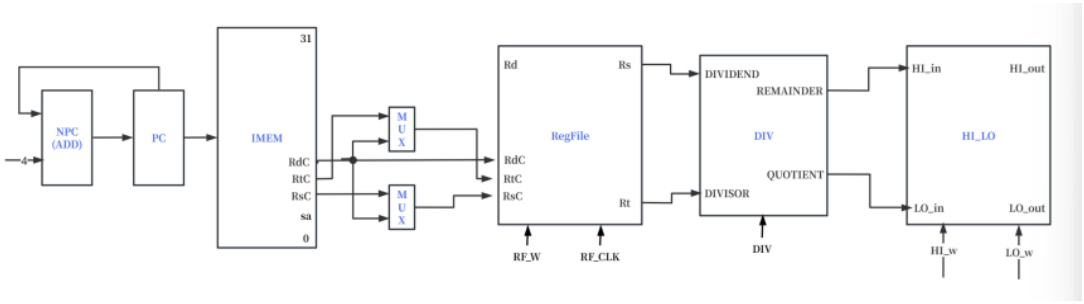
PC→IMEM

PC+4→NPC

NPC→PC

Rs→DIVDEND
Rt→DIVSOR
Rs/Rt →QUOTENT
Rs%Rt →REMAINDER
QUOTENT →HI_IN
REMAINDER →LO_IN

指令流程图：



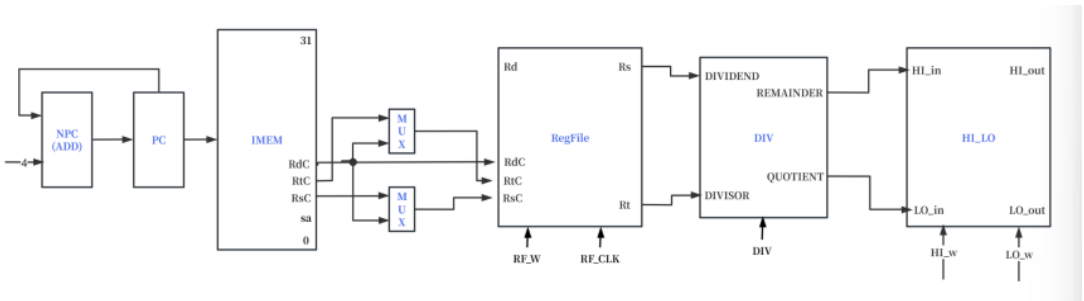
53、 DIVU

指令流程：

PC→IMEM
PC+4→NPC
NPC→PC

Rs→DIVDEND
Rt→DIVSOR
Rs/Rt →QUOTENT
Rs%Rt →REMAINDER
QUOTENT →HI_IN
REMAINDER →LO_IN

指令流程图：



54、 BGEZ

指令流程：

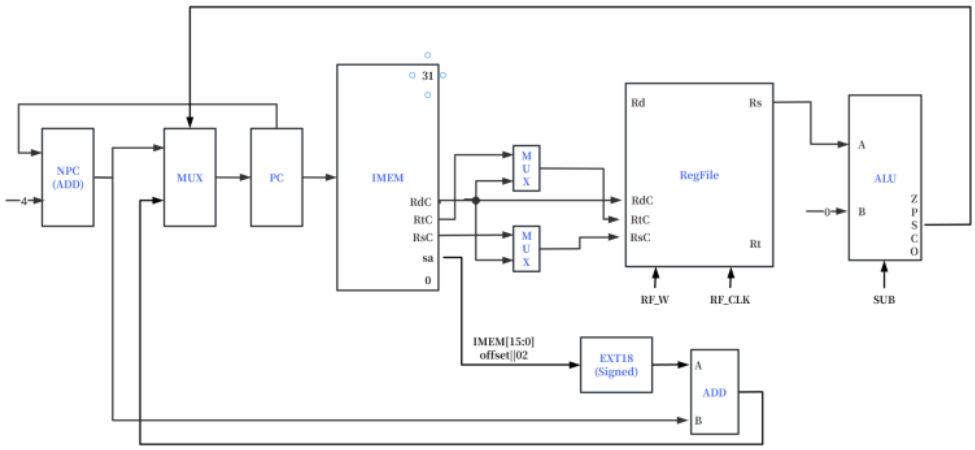
PC→IMEM
PC+4→NPC
NPC→PC

Rs →ALU_A
0→ALU_B
Rs – 0 →ANS

IMEM[15:0] || 02 →EXT18_IN
EXT18_OUT →ADD_A
NPC→ADD_B
ADD_A+ADD_B →ADD_OUT

If:
    MUX(NPC) →PC
Else:
    MUX(ADD_OUT)→PC

指令流程图：



三、主要模块设计

（1） sccomp_dataflow.v:顶层模块，负责调用其他各个模块构成整体；

1、 网站提交版本：

```
`timescale 1ns / 1ps

module sccomp_dataflow(
    input clk_in,
    input reset,
    output[31:0]inst,
    output[31:0]pc
    );
      wire[31:0] _pc;//符合 MARS 格式的 pc
      assign _pc = pc-32'h00400000;
      wire [31:0] _addr,addr;//符合 MARS 格式的地址
```

```
13、      assign _addr = (addr -32'h10010000);//这里不同于 cpu31，我们选择 DMEM 存储数据采用标准的 8 位是为了方便 LB 等取字节指令的执行
14、
15、      wire[31:0]ram_wdata,ram_rdata;
16、      wire ram_ena;
17、
18、      //CPU
19、      cpu sccpu(
20、          .clk(clk_in),
21、          .rst(reset),
22、          .inst(inst),
23、          .pc(pc),
24、          .ram_ena(ram_ena),
25、          .ram_addr(addr),
26、          .ram_rdata(ram_rdata),
27、          .ram_wdata(ram_wdata)
28、      );
29、
30、      //ROM
31、      IMEM rom(
32、          .addr(_pc[12:2]),
33、          .inst(inst)
34、      );
35、
36、      //RAM
37、      DMEM ram(
38、          .clk(clk_in),
39、          .w_ena(ram_ena),
40、          .addr(_addr),
41、          .wdata(ram_wdata),
42、          .rdata(ram_rdata)
43、      );
44、  endmodule
```

（2）下板测试版本：

```
1.`timescale 1ns / 1ps
2.
3.module sccomp_dataflow(
4.    input clk_in,
5.    input reset,
6.    output [7:0] o_seg,
7.    output [7:0] o_sel
8.    );
9.    wire [31:0] inst,pc;
10.     wire[31:0] _pc;//符合 MARS 格式的 pc
```

```
    assign _pc = pc-32'h00400000;
    wire [31:0] _addr,addr;//符合 MARS 格式的地址
    assign _addr = (addr -32'h10010000);//这里不同于 cpu31，我们选择
DMEM 存储数据采用标准的 8 位是为了方便 LB 等取字节指令的执行

    wire[31:0]ram_wdata,ram_rdata;
    wire ram_ena;
    wire clk_cpu;

    Divider u_divider(
        .clk(clk_in),
        .rst_n(~reset),
        .clk_out(clk_cpu)
    );

    //CPU
    cpu sccpu(
        .clk(clk_cpu),
        .rst(reset),
        .inst(inst),
        .pc(pc),
        .ram_ena(ram_ena),
        .ram_addr(addr),
        .ram_rdata(ram_rdata),
        .ram_wdata(ram_wdata)
    );

    //ROM
    IMEM rom(
        .addr(_pc[12:2]),
        .inst(inst)
    );

    //RAM
    DMEM ram(
        .clk(clk_cpu),
        .w_ena(ram_ena),
        .addr(_addr),
        .wdata(ram_wdata),
        .rdata(ram_rdata)
    );
     seg7x16 seg7x16_inst(
        .clk(clk_in),
        .reset(reset),
```

```
        .cs(1'b1),
        .i_data(inst),
        .o_seg(o_seg),
        .o_sel(o_sel)
    );
endmodule
```

（2） IMEM.v:实例化 IP 核，产生指令内存：

```
`timescale 1ns / 1ps

module IMEM(
    input [10:0] addr,
    output [31:0] inst
    );
    //IMEM 模块就是调用了我们事先实现好的 IP 核，给定一个地址，返回对应的指令，在 cpu 指令流程图中与 pc 相连接，起到读取指令的作用
    dist_mem_gen_0 inst_get(.a(addr),.spo(inst));
endmodule
```

（3） DMEM.v:内存模块，控制读写内存，注意不同于 CPU31，我们这里的 DMEM 模块编写时采用了更加规范化的 7 为一个地址的形式，是为了方便之后的 LB，SB 等对内存单个字节进行的操作。

```
`timescale 1ns / 1ps

module DMEM(
    input clk,
    input w_ena,//控制在存储器中的读写信号
    input [31:0]addr,
    input [31:0] wdata,
    output [31:0] rdata
    );
     reg[7:0] DM_data[0:1023];
     //这里我们向寄存器写采用同步逻辑，而读寄存器内容采用异步逻辑
     always@(negedge clk)
     begin
         if(w_ena)begin
             DM_data[addr + 3] <= wdata[31:24];
             DM_data[addr + 2] <= wdata[23:16];
             DM_data[addr + 1] <= wdata[15:8];
             DM_data[addr]     <= wdata[7:0];
         end
     end
     assign rdata[31:24] = (w_ena == 0) ? DM_data[addr + 3] : 8'b0;
```

```
    assign rdata[23:16] = (w_ena == 0) ? DM_data[addr + 2] : 8'b0;
    assign rdata[15:8]  = (w_ena == 0) ? DM_data[addr + 1] : 8'b0;
    assign rdata[7:0]   = (w_ena == 0) ? DM_data[addr]     : 8'b0;

endmodule
```

（4） cpu 模块：调用 ALU，MDU，CP0，RegFile 以及控制器模块，实现 cpu 的核心功能。

```
`timescale 1ns / 1ps

module cpu(
    input clk,
    input rst,
    input[31:0]inst,
    output [31:0]pc,
    output ram_ena,
    output[31:0]ram_addr,
     input [31:0]ram_rdata,
     output[31:0]ram_wdata
     );
     //定义需要在模块中调用的变量
     wire[31:0]pc_next;
     wire pc_ena;

     wire[4:0]rs,rt,rd;

     wire[31:0]rdata1,rdata2,wdata;
     wire[31:0]alu_a,alu_b,alu_res;

     wire w_ena;
     wire[5:0]alu_func;

     wire[4:0]waddr;
     wire zero,carry,negative,overflow;

     //cp0_port
     wire [31:0] cp0_rdata;
     wire[31:0] exc_addr;
     wire mtc0,eret,teq_exc;
     wire [3:0] cause;

     //MDU port
     wire [2:0]mduc;
```

```
36.     wire [63:0]mul_res;
37.     wire[31:0]high,low;
38.
39.     //PC port，完成 PC_next-->PC 的工作
40.     PCReg pc_reg(
41.         .clk(clk),
42.         .rst(rst),
43.         .ena(1),
44.         .pc_in(pc_next),
45.         .pc_out(pc)
46.     );
47.
48.     //Control_Unit 模块，完成指令的译码并产生各总线信号的模块
49.     Control_Unit my_control(
50.         .inst(inst),.pc(pc),
51.         .pc_out(pc_next),
52.         .rs(rs),.rt(rt),
53.         .rdata1(rdata1),.rdata2(rdata2),
54.         .alu_func(alu_func),.alu_a(alu_a), .alu_b(alu_b),
55.         .ram_rdata(ram_rdata),.alu_res(alu_res),
56.         .w_ena(w_ena), .waddr(waddr),.wdata(wdata),
57.         .ram_ena(ram_ena), .ram_addr(ram_addr),.ram_wdata(ram_wdata),

58.
59.         .cp0_rdata(cp0_rdata),.exc_addr(exc_addr),
60.         .mtc0(mtc0),.eret(eret),.teq_exc(teq_exc),
61.         .cause(cause),
62.         .rd(rd),
63.
64.         .mul_res(mul_res[31:0]),
65.         .high(high),.low(low),
66.         .mdu_op(mduc)
67.     );
68.
69.     //CP0
70.     CP0 cp0_unit(
71.         .clk(clk),.rst(rst),
72.         .mtc0(mtc0),
73.         .pc(pc),
74.         .addr(rd),.wdata(rdata2),
75.         .eret(eret),.teq_exc(teq_exc),.cause(cause),
76.         .rdata(cp0_rdata),.exc_addr(exc_addr)
77.     );
78.     //Register
```

```
    RegFile cpu_ref(
        .clk(clk),
        .rst(rst),
        .w_ena(w_ena),
        .waddr(waddr),
        .wdata(wdata),
        .raddr1(rs),
        .rdata1(rdata1),
        .raddr2(rt),
        .rdata2(rdata2)
    );

    //ALU
    ALU alu_part(
        .a(alu_a),
        .b(alu_b),
        .res(alu_res),
        .alu_func(alu_func),
        .zero(zero),
        .carry(carry),
        .negative(negative),
           .overflow(overflow)
      );

      //MDU
      MDU mdu_unit(
          .clk(clk),.rst(rst),
          .mduc(mduc),
          .a(rdata1),.b(rdata2),
          .mul_res(mul_res),
          .high(high),.low(low),
          .pc_ena(pc_ena)
      );
  endmodule
```

（5） ALU 模块：根据传入的操作类型进行相关计算并将结果返回。

```
`timescale 1ns / 1ps

module ALU(
    input[31:0] a,
    input[31:0] b,
    output[31:0] res,
    input[5:0] alu_func,
```

```
    output zero,
    output carry,
     output negative,
     output overflow
     );
     //参数中的 alu_func 代表执行运算的类型，由指令的 opt 和 func 决定，在传入 ALU 模块之前就计算好了
     //下面定义几个运算类型的表示,按照文档顺序定义
     parameter ADD = 0 ;
     parameter ADDU =1;
     parameter SUB = 2;
     parameter SUBU = 3;
     parameter AND = 4;
     parameter OR = 5;
     parameter XOR = 6;
     parameter NOR = 7;
     parameter SLT = 8;
     parameter SLTU = 9;
     parameter SLL = 10;
     parameter SRL = 11;
     parameter SRA = 12;
     parameter SLLV = 13;
     parameter SRLV = 14;
     parameter SRAV = 15 ;
     parameter LUI = 16;
     wire signed [31:0]sign_a,sign_b;//将给定数据转化为有符号数方便接下来进行计算
     assign sign_a = a;
     assign sign_b = b;
     reg[32:0]tmp_res;//暂时存储结果
     always@(*)
     begin
         case(alu_func)
         ADD : tmp_res<=sign_a+sign_b;
         ADDU : tmp_res<=a+b;
         SUB : tmp_res<=sign_a-sign_b;
         SUBU : tmp_res<=a-b;
         AND : tmp_res<=a & b;
         OR : tmp_res<=a | b;
         XOR : tmp_res<= a ^ b;
         NOR : tmp_res<= ~(a|b);
         SLT : tmp_res<=(sign_a<sign_b);
         SLTU : tmp_res<=(a<b);
         SLL : tmp_res<=(b<<a);
```

```
        SRL : tmp_res<=(b>>a);
        SRA : tmp_res<=(sign_b>>>sign_a);
        SLLV : tmp_res<= (b<<a[4:0]);
        SRLV : tmp_res<=(b>>a[4:0]);
        SRAV : tmp_res<=(sign_b>>>sign_a[4:0]);
        LUI : tmp_res<={b[15:0],16'b0};
        endcase
    end
    assign res = tmp_res[31:0];
    assign zero=(tmp_res==32'b0)?1:0;
    assign carry = tmp_res[32];
    assign overflow = tmp_res[32];
    assign negative = tmp_res[31];
endmodule
```

（6） MDU 模块：乘法器和除法器的结合，调用 MUL 和 DIV 模块返回相关结果：

```
`timescale 1ns / 1ps

//Multiply and Division Unit  乘除法器
module MDU(
    input clk,
    input rst,
    input [2:0]mduc,//控制乘除法的类型
    input [31:0] a,
    input [31:0] b,
     output [63:0] mul_res,
     output reg [31:0] high,
     output reg [31:0] low,
     output reg pc_ena
     );
     //定义常量
     parameter MDU_multu = 3'h2;
     parameter MDU_div = 3'h3;
     parameter MDU_divu = 3'h4;
     parameter MDU_mthi = 3'h5;
     parameter MDU_mtlo = 3'h6;

     //定义乘法的相关接口
     wire [63:0] mult_res,multu_res;
     assign mul_res = mult_res;

     //除法相关接口
```

```
27.     wire [63:0]div_res,divu_res;
28.     wire div_start,divu_start;
29.     wire div_busy,divu_busy;
30.     wire div_over,divu_over;
31.     assign div_over = ~ div_busy;
32.     assign divu_over = ~ divu_busy;
33.
34.     assign div_start = (mduc == MDU_div&&div_busy == 0)?1:0;
35.     assign divu_start = (mduc == MDU_divu&&divu_busy == 0)?1:0;
36.
37.     always @(*)
38.     begin
39.         case(mduc)
40.             MDU_div:pc_ena = (div_over == 1||mduc!=MDU_div)?1:0;
41.             MDU_divu:pc_ena = (divu_over ==1||mduc!=MDU_divu)?1:0;
42.             default:pc_ena = 1;
43.         endcase
44.     end
45.
46.     always @(negedge clk or posedge rst)
47.     begin
48.         if(rst == 1)begin
49.             high<=32'b0;
50.             low<=32'b0;
51.         end
52.         else
53.         begin
54.             case (mduc)
55.                 MDU_multu:{high,low}<=multu_res;
56.                 MDU_div:{low,high}<=div_res;
57.                 MDU_divu:{low,high}<=divu_res;
58.                 MDU_mthi:high<=a;
59.                 MDU_mtlo:low<=a;
60.             endcase
61.         end
62.     end
63.
64.     //乘法器
65.     MUL mul_unit(
66.         .a(a),
67.         .b(b),
68.         .mult_res(mult_res),
69.         .multu_res(multu_res)
70.     );
```

```
71.
72.     //除法器
73.     DIV div_unit(
74.         .a(a),
75.         .b(b),
76.         .div_res(div_res),
77.         .divu_res(divu_res),
78.         .div_busy(div_busy),
79.         .divu_busy(divu_busy)
80.     );
81.
82. endmodule
```

（7） MUL 模块：实现有符号乘法和无符号乘法：

```
1.`timescale 1ns / 1ps
2.
3.
4.module MUL(
5.    input [31:0]a,//乘数 a
6.    input [31:0]b,//乘数 b
7.    output [63:0]mult_res,//有符号乘
8.    output [63:0]multu_res//无符号乘
9.    );
10.     wire [63:0] unsigned_a,unsigned_b;
11.     wire signed [63:0] signed_a,signed_b;
12.     assign unsigned_a = {32'b0,a};
13.     assign unsigned_b = {32'b0,b};
14.
15.     assign signed_a = {{32{a[31]}},a};
16.     assign signed_b = {{32{b[31]}},b};
17.
18.     assign mult_res = signed_a*signed_b;
19.     assign multu_res = unsigned_a*unsigned_b;
20. endmodule
```

（8） DIV 模块：实现有符号除法和无符号除法

```
1.`timescale 1ns / 1ps
2.
3.module DIV(
4.    input [31:0]a,
5.    input [31:0]b,
6.    output[63:0]div_res,
```

```
    output[63:0]divu_res,
    output div_busy,
    output divu_busy
     );
     wire signed[31:0]signed_a,signed_b;
     assign signed_a = a;
     assign signed_b = b;
     wire[31:0] signed_q,signed_r,unsigned_q,unsigned_r;

     assign signed_r = (b == 32'b0)?32'b0:(signed_a%signed_b);
     assign signed_q = (b == 32'b0)?32'b0:(signed_a/signed_b);

     assign unsigned_r = (b == 32'b0)?32'b0:(a%b);
     assign unsigned_q = (b == 32'b0)?32'b0:(a/b);

     assign div_res = {signed_q,signed_r};
     assign divu_res = {unsigned_q,unsigned_r};
     assign div_busy = 0;
     assign divu_busy = 0;
endmodule
```

（9） RegFile 模块：实现寄存器堆的读写控制：

```
`timescale 1ns / 1ps



module RegFile(
    input clk,
    input rst,
    input w_ena,
    input [4:0]waddr,
     input [31:0] wdata,
     input [4:0]raddr1,
     output[31:0]rdata1,
     input [4:0]raddr2,
     output[31:0]rdata2
     );
     reg[31:0] array_reg[0:31];//定义 RegFile 中的寄存器
     integer i;
     //write 信号
     always@(negedge clk or posedge rst)
     begin
         if(rst)begin
```

```
            for(i=0;i<32;i=i+1)array_reg[i]<=32'b0;//rst 信号发挥作用，全部寄存器内容清空
        end
        else if(rst==0)begin
            if(w_ena&&waddr!=5'b0)begin
                //写信号且不能改变寄存器 0 中的内容
                array_reg[waddr]<=wdata;
            end
        end
    end

    //read 信号
    assign rdata1 = array_reg[raddr1];
    assign rdata2 = array_reg[raddr2];
endmodule
```

（10） PCReg 模块：产生下一条指令地址并输出

```
`timescale 1ns / 1ps


module PCReg(
    input clk,
    input rst,
    input ena,
    input [31:0]pc_in,
    output  reg[31:0] pc_out
    );
    always@(negedge clk or posedge rst)
    begin
        if(rst)pc_out<=32'h00400000;//这是 MARS 中取指令的开始地址
        else if(rst==0&&ena)pc_out<=pc_in;
    end
endmodule
```

（11） CP0 模块：控制 CP0 读写信号和错误返回信号：

```
`timescale 1ns / 1ps


module CP0(
    input clk,
    input rst,
    input mtc0,//cp0 write
```

```
8.    input [31:0]pc,//指令
9.    input [4:0] addr,//address
10.    input [31:0] wdata,//向 cp0 写的数据
11.    input eret,//是否为错误返回信号
12.    input teq_exc,
13.    input[3:0]cause,//异常原因
14.    output [31:0]rdata,//cp0 读出的数据
15.    output[31:0]exc_addr //返回异常保存的地址
16.    );
17.
18.    parameter  STATUS_POS = 12;
19.    parameter  CAUSE_POS = 13;
20.    parameter  EPC_POS = 14;
21.    parameter  SYS_ERR = 4'b1000;
22.    parameter  BREAK_ERR = 4'b1001;
23.    parameter  TEQ_ERR = 4'b1101;
24.
25.    reg[31:0] cp0_reg[0:31];
26.    wire [31:0] status;
27.    assign status = cp0_reg[STATUS_POS];
28.    wire exception;
29.    assign exception = (status[0] == 1)&&
30.    ((status[1] == 1&& cause == SYS_ERR)||
31.    (status[2] == 1&&cause == BREAK_ERR)||
32.    (status == 1&&cause == TEQ_ERR));
33.
34.    assign rdata = cp0_reg[addr];
35.    assign exc_addr = (eret?cp0_reg[EPC_POS]:32'h00400004);
36.
37.    integer i;
38.    always @(negedge clk or posedge rst)
39.    begin
40.        if(rst == 1)begin
41.            for(i = 0;i<32;i=i+1)
42.                cp0_reg[i]<=32'b0;
43.        end
44.        else
45.        begin
46.            if(mtc0 == 1)
47.                cp0_reg[addr]<=wdata;
48.            else if(exception)begin
49.                cp0_reg[STATUS_POS]<=(status<<5);
50.                cp0_reg[CAUSE_POS]<={24'b0,cause,2'b0};
51.                cp0_reg[EPC_POS]<=pc;
```

```
            end
            else if(eret == 1)begin
                cp0_reg[STATUS_POS]<=(status>>5);
            end
        end
    end
endmodule
```

（12） Control_Unit 模块：中心控制器模块，根据指令码产生控制信号并输出

```
`timescale 1ns / 1ps

//本模块作用是分析指令确定所有控制信号
module Control_Unit(
    input [31:0] inst,//得到的 cpu 指令
    input [31:0] pc,//上一条 pc
    output reg [31:0] pc_out,//因为涉及到转移指令，所以需要分情况讨论下一条 pc 的值

    output [4:0] rs,
     output [4:0] rt,//解析指令得到寄存器编码并输出方便 RegFile 模块调用

     input[31:0] rdata1,
     input [31:0] rdata2,//从 RegFile 里读取到的数据

     output reg[5:0] alu_func,//分析指令得到的 alu 执行操作类型信号
     output reg[31:0] alu_a,
     output reg[31:0] alu_b,

     input [31:0] ram_rdata,//从 DMEM 中读取到的数据
     input [31:0] alu_res,//从 ALU 模块中计算得到的数据

     output reg w_ena,//RegFile 写使能信号
     output [4:0] waddr,//写寄存器信号,指出是哪个寄存器
     output reg [31:0] wdata,//具体要写入寄存器中的值

     output ram_ena,//RAM 使能信号，控制读和写
     output [31:0] ram_addr,//RAM 写入的地址
     output reg [31:0] ram_wdata, //RAM 写入的值

     input [31:0] cp0_rdata,//cp0 读入数据
     input [31:0]exc_addr,//cp0 返回保存的地址
     output mtc0,//cp0 写信号
     output eret,//错误信号
```

```
34.     output teq_exc,
35.     output reg [3:0] cause,//错误代码
36.     output [4:0]rd,//cp0 addr
37.
38.     input[31:0] mul_res,
39.     input[31:0]high,
40.     input [31:0]low,
41.     output reg [2:0]mdu_op//乘除法类型
42.     );
43.
44.     //接下来首先定义几个有关指令具体类型的常量，方便接下来继续编写代码
45.     //alu_func 的类型
46.     parameter ADD = 0 ;
47.     parameter ADDU =1;
48.     parameter SUB = 2;
49.     parameter SUBU = 3;
50.     parameter AND = 4;
51.     parameter OR = 5;
52.     parameter XOR = 6;
53.     parameter NOR = 7;
54.     parameter SLT = 8;
55.     parameter SLTU = 9;
56.     parameter SLL = 10;
57.     parameter SRL = 11;
58.     parameter SRA = 12;
59.     parameter SLLV = 13;
60.     parameter SRLV = 14;
61.     parameter SRAV = 15 ;
62.     parameter LUI = 16;
63.     //三种指令类型及其具体分类
64.     parameter No_op = 6'b111111;
65.     parameter No_func = 6'b111111;
66.     parameter R_type_op = 6'b000000;
67.     parameter add_func = 6'b100000;
68.     parameter addu_func = 6'b100001;
69.     parameter sub_func = 6'b100010;
70.     parameter subu_func = 6'b100011;
71.     parameter and_func = 6'b100100;
72.     parameter or_func = 6'b100101;
73.     parameter xor_func = 6'b100110;
74.     parameter nor_func = 6'b100111;
75.     parameter slt_func = 6'b101010;
76.     parameter sltu_func = 6'b101011;
77.     parameter sll_func = 6'b000000;
```

```
78.     parameter srl_func = 6'b000010;
79.     parameter sra_func = 6'b000011;
80.     parameter sllv_func = 6'b000100;
81.     parameter srlv_func = 6'b000110;
82.     parameter srav_func = 6'b000111;
83.     parameter jr_func = 6'b001000;
84.
85.     parameter addi_op = 6'b001000;
86.     parameter addiu_op = 6'b001001;
87.     parameter andi_op = 6'b001100;
88.     parameter ori_op = 6'b001101;
89.     parameter xori_op = 6'b001110;
90.     parameter lw_op = 6'b100011;
91.     parameter sw_op = 6'b101011;
92.     parameter beq_op = 6'b000100;
93.     parameter bne_op = 6'b000101;
94.     parameter slti_op = 6'b001010;
95.     parameter sltiu_op = 6'b001011;
96.     parameter lui_op = 6'b001111;
97.
98.     parameter j_op = 6'b000010;
99.     parameter jal_op = 6'b000011;
100.
101.       parameter lb_op = 6'b100000;
102.       parameter lbu_op = 6'b100100;
103.       parameter lh_op = 6'b100001;
104.       parameter lhu_op =  6'b100101;
105.       parameter sb_op = 6'b101000;
106.       parameter sh_op = 6'b101001;
107.
108.       parameter ex_op1 = 6'b000000;
109.       parameter jalr_func = 6'b001001;
110.       parameter syscall_func = 6'b001100;
111.       parameter teq_func = 6'b110100;
112.       parameter break_func = 6'b001101;
113.       parameter multu_func = 6'b011001;
114.       parameter div_func = 6'b011010;
115.       parameter divu_func = 6'b011011;
116.       parameter mthi_func = 6'b010001;
117.       parameter mtlo_func = 6'b010011;
118.       parameter mfhi_func = 6'b010000;
119.       parameter mflo_func = 6'b010010;
120.
121.       parameter ex_op2 = 6'b011100;
```

```
122.        parameter clz_func = 6'b100000;
123.        parameter mul_func = 6'b000010;
124.
125.        parameter CP0_op = 6'b010000;
126.        parameter eret_func = 6'b011000;
127.        parameter mfc0_rs = 5'b00000;
128.        parameter mtc0_rs = 5'b00100;
129.
130.        parameter bgez_op = 6'b000001;
131.
132.        parameter  SYS_ERR = 4'b1000;
133.        parameter  BREAK_ERR = 4'b1001;
134.        parameter  TEQ_ERR = 4'b1101;
135.
136.        parameter MDU_default = 0;
137.        parameter MDU_multu = 2;
138.        parameter MDU_div = 3;
139.        parameter MDU_divu = 4;
140.        parameter MDU_mthi = 5;
141.        parameter MDU_mtlo = 6;
142.
143.        //首先，我们按照所有可能的指令格式将指令分解为可能有意义的几段
144.        wire[5:0]op;
145.        wire [4:0]shamt;
146.        wire [5:0]func;
147.        wire [15:0] imm;
148.        wire [25:0] addr;
149.
150.        assign op = inst[31:26];
151.        assign shamt = inst[10:6];
152.        assign func = inst[5:0];
153.        assign imm = inst[15:0];
154.        assign addr = inst[25:0];
155.
156.        assign rs = inst[25:21];
157.        assign rt = inst[20:16];
158.        assign rd = inst[15:11];
159.
160.        wire [31:0] shamt_ex;//shamt 的 32 位扩充
161.        wire [31:0] imm_ex;//imm 的 32 位扩充
162.
163.        assign shamt_ex = {27'b0,shamt};
164.        assign imm_ex = (op == andi_op||op == ori_op||op == xori_op)?{
    16'b0,imm}:{{16{imm[15]}},imm};
```

```
165.
166.       //确定指令需要写入哪个寄存器
167.       assign waddr = (op == R_type_op||op == ex_op2)?rd:((op == jal_
   op)?5'b11111:rt);
168.
169.       //确定所有可能的下一个 pc 的值
170.       wire[31:0]npc;
171.       wire [31:0]pc_jump;
172.       wire [31:0] pc_branch;
173.
174.       assign npc = pc+4;
175.       assign pc_jump = {npc[31:28],addr,2'b00};
176.       assign pc_branch = npc +{{14{imm[15]}},imm,2'b00};
177.
178.       //根据指令确定 DMEM 的信号
179.       assign ram_ena = (op == sw_op||op == sb_op||op == sh_op)?1:0;/
   /1 是写信号，0 是读信号
180.       assign ram_addr = rdata1 + imm_ex;//lw 或 sw 指令需要的 DMEM 地址
181.       reg [31:0]load_data;//向 DMEM 中写入的值
182.
183.       //CP0 准备工作
184.       assign eret = (op == CP0_op&&func == eret_func)?1:0;
185.       assign mtc0 = (op == CP0_op&&rs == mtc0_rs)?1:0;
186.       assign teq_exc = (rdata1 == rdata2)?1:0;
187.       wire mfc0;
188.       assign mfc0 = (op == CP0_op&&rs == mfc0_rs)?1:0;
189.
190.       //ALU 部分信号
191.       always@(*)
192.       begin
193.           //首先给 ALU_data 赋值
194.           case(op)
195.           R_type_op:
196.           begin
197.               case(func)
198.               add_func,sub_func,
199.               addu_func,subu_func,
200.               and_func,or_func,xor_func,
201.               nor_func,slt_func,sltu_func,
202.               sllv_func,srlv_func,srav_func:
203.               begin
204.                   alu_a<=rdata1;
205.                   alu_b<=rdata2;
206.               end
```

```
207.                sll_func,srl_func,sra_func:
208.                begin
209.                    alu_a<=shamt_ex;
210.                    alu_b<=rdata2;
211.                end
212.                default:
213.                begin
214.                    alu_a<=rdata1;
215.                    alu_b<=rdata2;
216.                end
217.                endcase
218.            end
219.            addi_op,addiu_op,andi_op,
220.            ori_op,xori_op,slti_op,
221.            sltiu_op,lui_op:
222.            begin
223.                alu_a<=rdata1;
224.                alu_b<=imm_ex;
225.            end
226.            default:
227.            begin
228.                alu_a<=rdata1;
229.                alu_b<=rdata2;
230.            end
231.            endcase
232.
233.            //接下来给 ALU_func 赋值
234.            case(op)
235.                R_type_op:
236.                begin
237.                    case(func)
238.                        add_func : alu_func<=ADD;
239.                        addu_func : alu_func<=ADDU;
240.                        sub_func : alu_func<=SUB;
241.                        subu_func : alu_func<=SUBU;
242.                        and_func : alu_func<=AND;
243.                        or_func : alu_func<=OR;
244.                        xor_func : alu_func<=XOR;
245.                        nor_func : alu_func<=NOR;
246.                        slt_func : alu_func<=SLT;
247.                        sltu_func : alu_func<=SLTU;
248.                        sll_func : alu_func<=SLL;
249.                        sllv_func : alu_func<=SLLV;
250.                        srl_func : alu_func<=SRL;
```

```
                    srlv_func : alu_func<=SRLV;
                    sra_func : alu_func<=SRA;
                    srav_func : alu_func<=SRAV;
                    default :alu_func<=ADDU;
                endcase
            end
            addi_op : alu_func<=ADD;
            addiu_op : alu_func<=ADDU;
            andi_op : alu_func<=AND;
            ori_op : alu_func<=OR;
            xori_op : alu_func<=XOR;
            slti_op : alu_func<=SLT;
            sltiu_op : alu_func<=SLTU;
            lui_op : alu_func<=LUI;
            default : alu_func<=ADDU;
        endcase
    end

    //RAM 部分信号
    always@(*)
    begin
        //load 信号
        case(op)
            lw_op : load_data<=ram_rdata;

            lb_op:     load_data <= {{24{ram_rdata[7]}}, ram_rdata[7:0]};
            lbu_op:    load_data <= {24'b0, ram_rdata[7:0]};
            lh_op:     load_data <= {{16{ram_rdata[15]}}, ram_rdata[15:0]};
            lhu_op:    load_data <= {16'b0, ram_rdata[15:0]};
            default : load_data<=ram_rdata;
        endcase
        //sw 信号
        case(op)
            sw_op : ram_wdata<=rdata2;

            sb_op : ram_wdata<={24'b0,rdata2[7:0]};
            sh_op : ram_wdata<={16'b0,rdata2[15:0]};

            default : ram_wdata<=rdata2;
        endcase
    end

```

```
293.      //RegFile 部分信号
294.      always@(*)
295.      begin
296.         case(op)
297.          R_type_op:
298.          begin
299.              case(func)
300.                  jr_func : w_ena<=0;
301.
302.                  syscall_func,break_func:w_ena<=0;
303.
304.                  multu_func,div_func,divu_func,
305.                  mthi_func,mtlo_func:w_ena<=0;
306.
307.                  default:w_ena<=1;
308.              endcase
309.          end
310.          addi_op,addiu_op,andi_op,
311.          ori_op,xori_op,lw_op,
312.          slti_op,sltiu_op,lui_op,
313.          jal_op:w_ena<=1;
314.
315.          lb_op,lbu_op,
316.          lh_op,lhu_op:
317.                      w_ena<=1;
318.          CP0_op:     w_ena<=(rs == mfc0_rs)?1:0;
319.
320.          ex_op2 :    w_ena<=1;
321.          default:    w_ena<=0;
322.         endcase
323.         case(op)
324.              jal_op : wdata<=npc;
325.              lw_op : wdata<=load_data;
326.
327.              R_type_op:
328.              begin
329.                  case(func)
330.                      jalr_func : wdata<=npc;
331.                      mfhi_func : wdata<=high;
332.                      mflo_func : wdata<=low;
333.
334.                      default: wdata<=alu_res;
335.                  endcase
336.              end
```

```
337.
338.                 ex_op2:
339.                 begin
340.                     case(func)
341.                         clz_func:
342.                         begin
343.                             casex (rdata1)
344.                                 32'b1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h00;
345.                                 32'b01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h01;
346.                                 32'b001xxxxxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h02;
347.                                 32'b0001xxxxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h03;
348.                                 32'b00001xxxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h04;
349.                                 32'b000001xxxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h05;
350.                                 32'b0000001xxxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h06;
351.                                 32'b00000001xxxxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h07;
352.                                 32'b000000001xxxxxxxxxxxxxxxxxxxxxx?:
    wdata <= 32'h08;
353.                                 32'b0000000001xxxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h09;
354.                                 32'b00000000001xxxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h0a;
355.                                 32'b000000000001xxxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h0b;
356.                                 32'b0000000000001xxxxxxxxxxxxxxxxxxx:
    wdata <= 32'h0c;
357.                                 32'b00000000000001xxxxxxxxxxxxxxxxxx:
    wdata <= 32'h0d;
358.                                 32'b000000000000001xxxxxxxxxxxxxxxxx:
    wdata <= 32'h0e;
359.                                 32'b0000000000000001xxxxxxxxxxxxxxxx:
    wdata <= 32'h0f;
360.                                 32'b00000000000000001xxxxxxxxxxxxxxx:
    wdata <= 32'h10;
361.                                 32'b000000000000000001xxxxxxxxxxxxxx:
    wdata <= 32'h11;
```

```
                                32'b00000000000000000001xxxxxxxxxxxx:
    wdata <= 32'h12;
                                32'b00000000000000000001xxxxxxxxxxx:
    wdata <= 32'h13;
                                32'b00000000000000000001xxxxxxxxxx:
    wdata <= 32'h14;
                                32'b00000000000000000001xxxxxxxxx:
    wdata <= 32'h15;
                                32'b00000000000000000001xxxxxxxx:
    wdata <= 32'h16;
                                32'b00000000000000000001xxxxxxx:
    wdata <= 32'h17;
                                32'b00000000000000000001xxxxxx:
    wdata <= 32'h18;
                                32'b00000000000000000001xxxxx:
    wdata <= 32'h19;
                                32'b00000000000000000001xxxx:
    wdata <= 32'h1a;
                                32'b00000000000000000001xxx:
    wdata <= 32'h1b;
                                32'b00000000000000000001xx:
    wdata <= 32'h1c;
                                32'b00000000000000000001x:
    wdata <= 32'h1d;
                                32'b00000000000000000000:
    wdata <= 32'h1e;
                                default: wdata <= 32'h00;
                            endcase
                        end
                        mul_func:wdata<=mul_res;
                        default:wdata<=32'b0;
                    endcase
                end

                lb_op,lbu_op,lh_op,lhu_op: wdata<=load_data;

                CP0_op:wdata<=(rs == mfc0_rs)?cp0_rdata:alu_res;

                default: wdata<=alu_res;
            endcase
```

```
391.        end
392.
393.        //PCRegFile
394.        always@(*)
395.        begin
396.            case(op)
397.                R_type_op :
398.                begin
399.                    case(func)
400.                        jr_func : pc_out<=rdata1;
401.
402.                        jalr_func : pc_out<=rdata1;
403.                        syscall_func,break_func,teq_func:
404.                                        pc_out<=exc_addr;
405.                        default : pc_out<=npc;
406.                    endcase
407.                end
408.                j_op,jal_op : pc_out<=pc_jump;
409.                beq_op : pc_out<=(rdata1 == rdata2)?pc_branch:npc;
410.                bne_op : pc_out<=(rdata1!=rdata2)?pc_branch:npc;
411.
412.                CP0_op:
413.                begin
414.                    case(func)
415.                        eret_func : pc_out<=exc_addr;
416.                        default : pc_out<=npc;
417.                    endcase
418.                end
419.                bgez_op:
420.                    pc_out<=(rdata1[31] == 0)?pc_branch:npc;
421.                default : pc_out<=npc;
422.            endcase
423.        end
424.
425.        //CP0
426.        always@(*)
427.        begin
428.            if(op == ex_op1)
429.            begin
430.                case(func)
431.                    syscall_func : cause<=SYS_ERR;
432.                    teq_func :cause<=TEQ_ERR;
433.                    break_func:cause<=BREAK_ERR;
434.                    default : cause<=4'b0000;
```

```
435.                endcase
436.            end
437.        end
438.
439.        //MDU
440.        always@(*)
441.        begin
442.            if(op == ex_op1)
443.            begin
444.                case(func)
445.                    multu_func : mdu_op<=MDU_multu;
446.                    div_func   : mdu_op<=MDU_div;
447.                    divu_func  : mdu_op<=MDU_divu;
448.                    mthi_func  : mdu_op<=MDU_mthi;
449.                    mtlo_func  : mdu_op<=MDU_mtlo;
450.                    default    : mdu_op<=MDU_default;
451.                endcase
452.            end
453.            else
454.                mdu_op <= MDU_default;
455.        end
456.    endmodule
```

（13） Divider 模块：分频器模块，调试下板使用

```
1.`timescale 1ns / 1ps
2.
3.
4.module Divider(
5.    input clk,
6.    input rst_n,
7.    output reg clk_out
8.    );
9.    reg [31:0] count3=32'd0; //50,000,000 分频
10.     //50,000,000 分频
11.     always @(posedge clk)
12.     begin
13.         if(!rst_n)
14.         begin
15.         count3 <= 1'b0;
16.         clk_out <= 0;
17.         end
18.         else if(count3 == 32'd50000000)
19.         begin
```

```
            count3 <= 32'd0;
            clk_out <= ~clk_out;
        end
        else
            count3 <= count3+1'b1;
    end
endmodule
```

（14） Seg7x16 模块：七段数码管，下板时实时将执行的指令显示在七段数码管上。

```
`timescale 1ns / 1ps


module seg7x16(
    input clk,
    input reset,
    input cs,
    input [31:0] i_data,    //需要数码管输出的内容
    output [7:0] o_seg,     //输出内容
     output [7:0] o_sel      //片选信号
    );

    reg [14:0] cnt;
     always @ (posedge clk, posedge reset)
      if (reset)
        cnt <= 0;
      else
        cnt <= cnt + 1'b1;

    wire seg7_clk = cnt[14];

     reg [2:0] seg7_addr;

     always @ (posedge seg7_clk, posedge reset)
       if(reset)
          seg7_addr <= 0;
        else
          seg7_addr <= seg7_addr + 1'b1;

     reg [7:0] o_sel_r;

     always @ (*)
       case(seg7_addr)
```

```
          7 : o_sel_r = 8'b01111111;
          6 : o_sel_r = 8'b10111111;
          5 : o_sel_r = 8'b11011111;
          4 : o_sel_r = 8'b11101111;
          3 : o_sel_r = 8'b11110111;
          2 : o_sel_r = 8'b11111011;
          1 : o_sel_r = 8'b11111101;
          0 : o_sel_r = 8'b11111110;
       endcase

    reg [31:0] i_data_store;
    always @ (posedge clk, posedge reset)
      if(reset)
         i_data_store <= 0;
       else if(cs)
         i_data_store <= i_data;

    reg [7:0] seg_data_r;
    always @ (*)
      case(seg7_addr)
         0 : seg_data_r = i_data_store[3:0];
         1 : seg_data_r = i_data_store[7:4];
         2 : seg_data_r = i_data_store[11:8];
         3 : seg_data_r = i_data_store[15:12];
         4 : seg_data_r = i_data_store[19:16];
         5 : seg_data_r = i_data_store[23:20];
         6 : seg_data_r = i_data_store[27:24];
         7 : seg_data_r = i_data_store[31:28];
       endcase

    reg [7:0] o_seg_r;
    always @ (posedge clk, posedge reset)
      if(reset)
         o_seg_r <= 8'hff;
       else
         case(seg_data_r)
               4'h0 : o_seg_r <= 8'hC0;
               4'h1 : o_seg_r <= 8'hF9;
               4'h2 : o_seg_r <= 8'hA4;
               4'h3 : o_seg_r <= 8'hB0;
               4'h4 : o_seg_r <= 8'h99;
               4'h5 : o_seg_r <= 8'h92;
               4'h6 : o_seg_r <= 8'h82;
               4'h7 : o_seg_r <= 8'hF8;
```

```
                4'h8 : o_seg_r <= 8'h80;
                4'h9 : o_seg_r <= 8'h90;
                4'hA : o_seg_r <= 8'h88;
                4'hB : o_seg_r <= 8'h83;
                4'hC : o_seg_r <= 8'hC6;
            4'hD : o_seg_r <= 8'hA1;
                4'hE : o_seg_r <= 8'h86;
                4'hF : o_seg_r <= 8'h8E;
          endcase

     assign o_sel = o_sel_r;
     assign o_seg = o_seg_r;

endmodule
```

四、测试/调试过程

（1） 前仿真测试模块：

Cpu_tb.v:前仿真模块通过实时将 pc，inst，和每个寄存器的值打印到一个 txt 文件中通过文件中的内容和 MARS 中逐步执行的内容进行比较来调试。

```
`timescale 1ns / 1ps


module cpu_tb;

reg clk;            //时钟信号
reg rst;            //复位信号
wire [31:0] inst;   //要执行的指令
wire [31:0] pc;     //下一条指令的地址
reg  [31:0] cnt;    //计数器，已经执行了几条指令
integer file_open;

initial
begin
    clk = 1'b0;
    rst = 1'b1;
    #50 rst = 1'b0;
    cnt = 0;
end

always  #50 clk = ~clk;

```

```
always @ (posedge clk) begin
    cnt = cnt + 1'b1;
    if(1)begin
        file_open = $fopen("C:\\Users\\14864\\Desktop\\output_my.txt", "a+");
        $fdisplay(file_open, "OP: %d", cnt);
        $fdisplay(file_open, "Instr_addr = %h", sc_inst.inst);
        $fdisplay(file_open, "$regfile0 = %h", sc_inst.sccpu.cpu_ref.array_reg[0]);
        $fdisplay(file_open, "$regfile1   = %h", sc_inst.sccpu.cpu_ref.array_reg[1]);
        $fdisplay(file_open, "$regfile2   = %h", sc_inst.sccpu.cpu_ref.array_reg[2]);
        $fdisplay(file_open, "$regfile3   = %h", sc_inst.sccpu.cpu_ref.array_reg[3]);
        $fdisplay(file_open, "$regfile4  = %h", sc_inst.sccpu.cpu_ref.array_reg[4]);
        $fdisplay(file_open, "$regfile5  = %h", sc_inst.sccpu.cpu_ref.array_reg[5]);
        $fdisplay(file_open, "$regfile6  = %h", sc_inst.sccpu.cpu_ref.array_reg[6]);
        $fdisplay(file_open, "$regfile7  = %h", sc_inst.sccpu.cpu_ref.array_reg[7]);
        $fdisplay(file_open, "$regfile8  = %h", sc_inst.sccpu.cpu_ref.array_reg[8]);
        $fdisplay(file_open, "$regfile9  = %h", sc_inst.sccpu.cpu_ref.array_reg[9]);
        $fdisplay(file_open, "$regfile10  = %h", sc_inst.sccpu.cpu_ref.array_reg[10]);
        $fdisplay(file_open, "$regfile11   = %h", sc_inst.sccpu.cpu_ref.array_reg[11]);
        $fdisplay(file_open, "$regfile12   = %h", sc_inst.sccpu.cpu_ref.array_reg[12]);
        $fdisplay(file_open, "$regfile13   = %h", sc_inst.sccpu.cpu_ref.array_reg[13]);
        $fdisplay(file_open, "$regfile14   = %h", sc_inst.sccpu.cpu_ref.array_reg[14]);
        $fdisplay(file_open, "$regfile15   = %h", sc_inst.sccpu.cpu_ref.array_reg[15]);
        $fdisplay(file_open, "$regfile16   = %h", sc_inst.sccpu.cpu_ref.array_reg[16]);
        $fdisplay(file_open, "$regfile17   = %h", sc_inst.sccpu.cpu_ref.array_reg[17]);
```

```
47.         $fdisplay(file_open, "$regfile18   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[18]);
48.         $fdisplay(file_open, "$regfile19   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[19]);
49.         $fdisplay(file_open, "$regfile20   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[20]);
50.         $fdisplay(file_open, "$regfile21   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[21]);
51.         $fdisplay(file_open, "$regfile22   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[22]);
52.         $fdisplay(file_open, "$regfile23   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[23]);
53.         $fdisplay(file_open, "$regfile24   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[24]);
54.         $fdisplay(file_open, "$regfile25   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[25]);
55.         $fdisplay(file_open, "$regfile26   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[26]);
56.         $fdisplay(file_open, "$regfile27   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[27]);
57.         $fdisplay(file_open, "$regfile28   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[28]);
58.         $fdisplay(file_open, "$regfile29   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[29]);
59.         $fdisplay(file_open, "$regfile30   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[30]);
60.         $fdisplay(file_open, "$regfile31   = %h", sc_inst.sccpu.cpu_r
    ef.array_reg[31]);
61.         $fdisplay(file_open, "$pc   = %h\n", sc_inst.pc);
62.         $fclose(file_open);
63.     end
64. end
65.
66. sccomp_dataflow sc_inst(
67.     .clk_in(clk),
68.     .reset(rst),
69.     .inst(inst),
70.     .pc(pc)
71. );
72. endmodule
```

（2） 后仿真测试模块

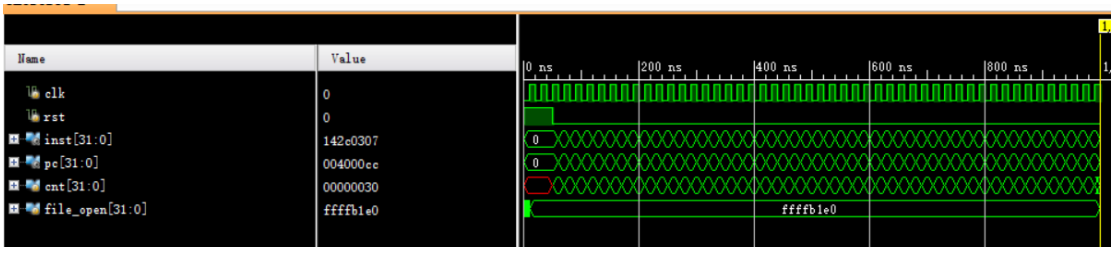
后仿真模块与前仿真模块基本一致，因为 Modelsim 不支持$display 函数，所以需要把所有的打印函数注释掉使用：

```
1.`timescale 1ns / 1ps
2.
3.
4.module cpu_tb;
5.
6.reg clk;            //时钟信号
7.reg rst;            //复位信号
8.wire [31:0] inst;   //要执行的指令
9.wire [31:0] pc;     //下一条指令的地址
10. reg  [31:0] cnt;    //计数器，已经执行了几条指令
11. integer file_open;
12.
13. initial
14. begin
15.     clk = 1'b0;
16.     rst = 1'b1;
17.     #50 rst = 1'b0;
18.     cnt = 0;
19. end
20.
21. always  #50 clk = ~clk;
22.
23.
24. sccomp_dataflow sc_inst(
25.     .clk_in(clk),
26.     .reset(rst),
27.     .inst(inst),
28.     .pc(pc)
29. );
30. endmodule
```

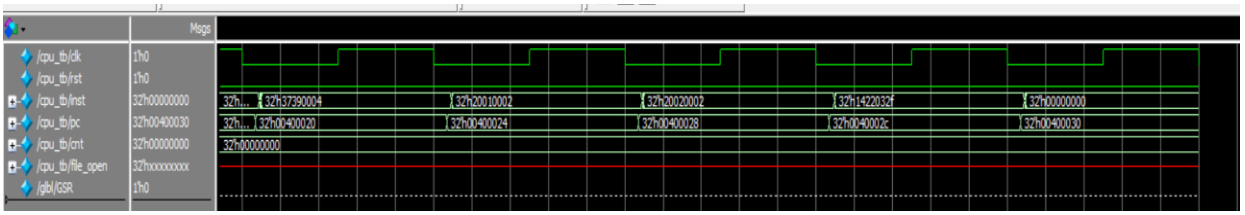
五、实验结果分析

（该部分可截图说明）

（1） 前仿真实验结果：



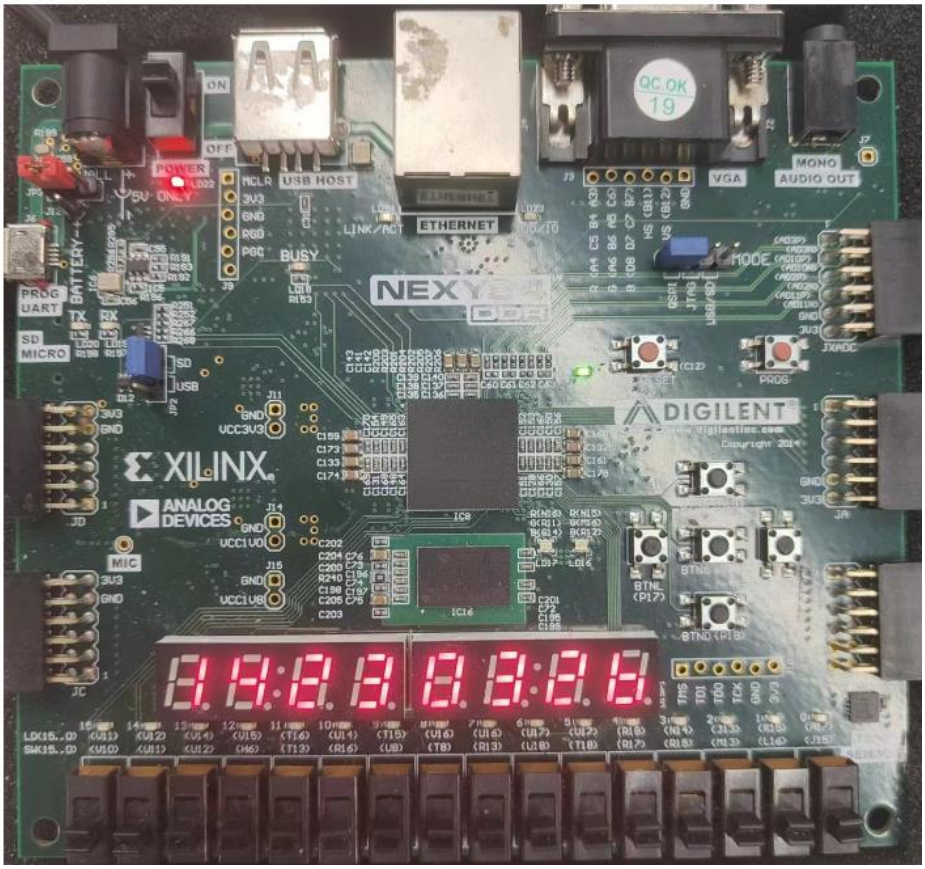
（2） 后仿真实验结果：

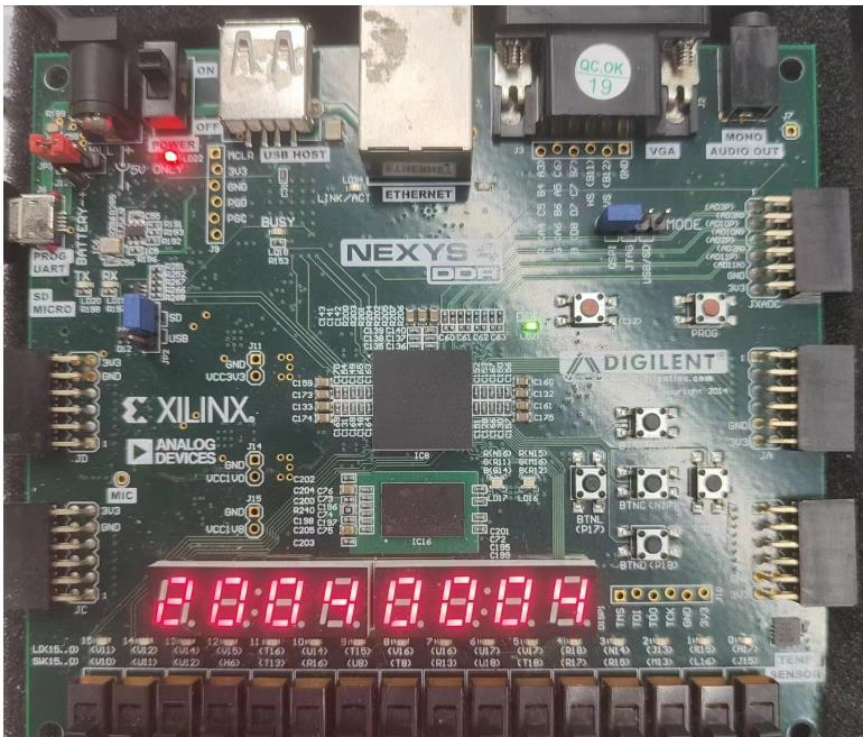


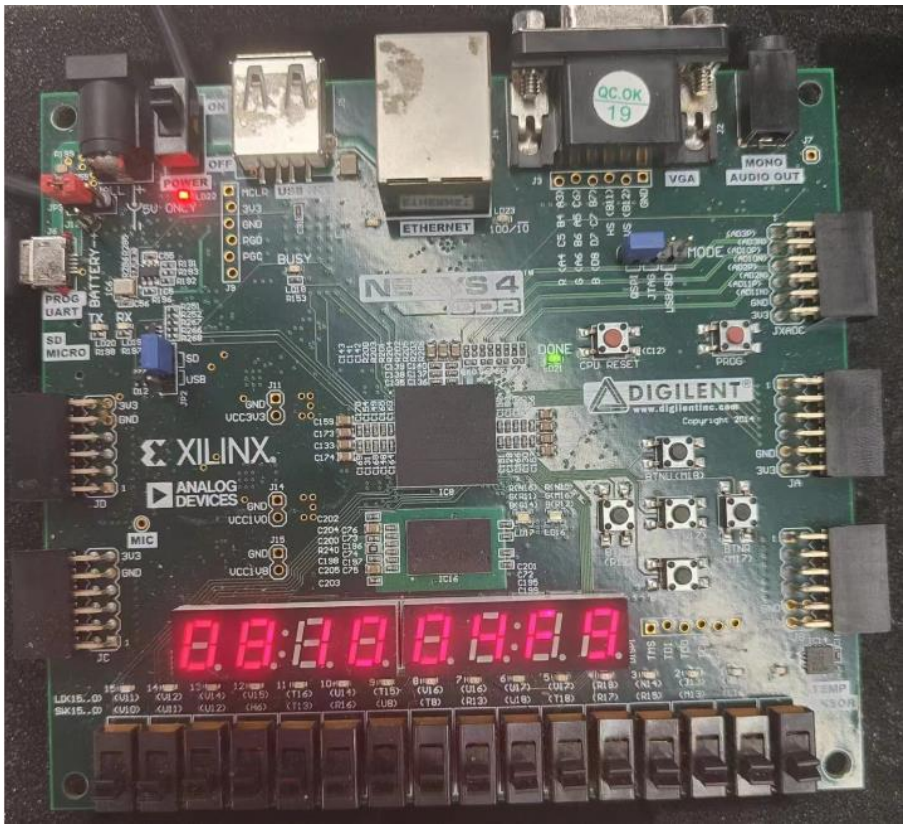
后仿真下可以明显看出波形延迟

（3） 下板实验结果

下板后在控制合适的时钟周期下大约会 1s 左右显示一个指令的内容，下面简单列出几条：







六、结论

本次 54 条单周期 CPU 各部件编写与连接正确，能够在本地顺利通过前仿真，后仿真以及下板实验，同时也能在网站上测试通过。

七、心得体会及建议

1、心得体会

对于这种大型的 CPU 设计实验，前期对程序合理的分块以及良好的注释有助于编写效率和正确性的提高。

本次 54 条 CPU 的编写就很大程度上利用了之前 31 条指令编写的成果，因为在之前 31 条指令 cpu 的编写时就着重注意了程序的分块性，所以在额外拓展 23 条指令时，只需要单独编写其对应的模块，最后统一连接即可，原来的 31 条指令的实现并没有做很大改动。

本次 CPU 设计时也出现了一些错误，比如之前为了简便，我对于数据选择器大部分都是用三目运算符去替代，结果导致在拓展乘法指令时，有一个控制信号忘记加入原来三目运算符的条件中导致乘法运算取的寄存器出错，另外还有，在 31 条 cpu 编写时为了简单起见，DMEM 模块的编写我都直接以 32 位为一个单位进行编址，但是新增的 LB，SB 等指令是对内存取字节，所以在本次的编写中我将 DMEM 模块更改为了更标准的以字节为单位编址的形式，可拓展性也大大提高。

2、建议

建议之后提供的测试 coe 文件可以同时包括对应的 MARS 指令对照文件，方便测试每条指令的正确性。