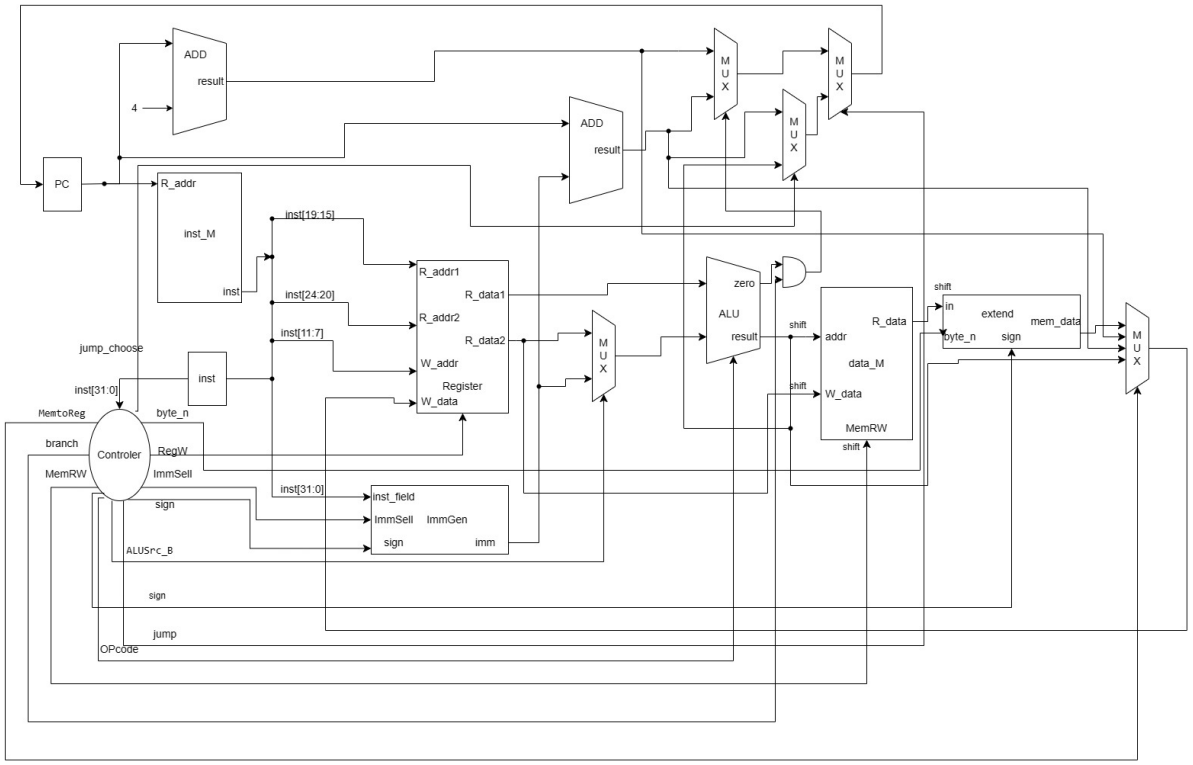
浙江大学实验报告 Lab4-3

专业：计算机科学与技术　姓名：仇国智　学号：3220102181　日期：2024/4/10
课程名称：计算机组成与设计　实验名称:实现单周期 CPU-指令拓展　指导老师：刘海风　成绩：

操作方法与实验步骤

绘制Datapath图



源文件编写

将原来lab2的工程复制一份,删除 `SCPU` 模块,依据原理图,自行设计 `SCPU` 模块(下辖 `ALU` , `Controler` , `Datapath` , `extend` , `ImmGEN` , `Regs` 模块, `ALU` , `Regs` 已经在前面实验实现,可直接复制)

`SCPU` 模块

`SCPU` 模块主要连接 `Controler` 和 `Datapath` 模块,并且包含了 `Inst` 指令的寄存器(实际上应该把 `Regs` 和 `PC` 的寄存器都拉到 `Datapath` ,这样结构会更加清晰),还计算了 `MemRW` (实际上这个变量没有必要存在),,同时对 `Data_out` 和 `wea` 进行了左位移偏移处理(因为 `RAM` 的地址是四字节对齐的,所以会对 `Addr_out` 造成截断, `Data_out` 和 `wea` 需要与截断后的 `Addr_out` 对齐,即代码 30 和 31 行).

```
1  `timescale 1ns / 1ps
2  `include "Lab4.vh"
3  module SCPU (
4    `RegFile_Regs_Outputs
5    input clk,
6    input rst,
7    input MIO_ready,
8    input [31:0] inst_in,
9    input [31:0] Data_in,
```

```
    output CPU_MIO,
    output MemRW,
    output wire [31:0] PC_out,
    output [31:0] Data_out,
    output [31:0] Addr_out,
    output wire [3:0] wea
  );
    wire [`IMM_SEL_WIDTH-1:0] ImmSell;
    wire ALUSrc_B;
    wire [`MEM2REG_WIDTH-1:0] MemtoReg;
    wire Jump;
    wire Branch;
    wire RegWrite;
    wire sign;
    wire jump_choose;
    // 0 1byte,1 2byte,2 4byte
    wire [1:0] byte_n;
    assign MemRW = |wea;
    wire [ 3:0] wea_temp;
    wire [31:0] Data_out_temp;
    assign wea = wea_temp << (Addr_out % 4);
    assign Data_out = MemRW ? (Data_out_temp << ((Addr_out % 4) << 3)) :
Data_out_temp;
    wire [`ALU_OP_WIDTH-1:0] ALU_Control;
    reg [31:0] inst;
    always @(posedge clk or posedge rst) begin
      if (rst) begin
        inst <= 32'h0;
      end else begin
        inst <= inst_in;
      end
    end
    Controler v1 (
        .OPcode(inst[6:2]),
        .MIO_ready(MIO_ready),
        .Fun7(inst[31:25]),
        .Fun3(inst[14:12]),

        .wea(wea_temp),
        .CPU_MIO(CPU_MIO),
        .ImmSell(ImmSell),
        .ALUSrc_B(ALUSrc_B),
        .MemtoReg(MemtoReg),
        .Jump(Jump),
        .Branch(Branch),
        .RegWrite(RegWrite),
        .ALU_Control(ALU_Control),
        .sign(sign),
        .byte_n(byte_n),
        .jump_choose(jump_choose)
    );
    DataPath v2 (
        `RegFile_Regs_Arguments
        .clk(clk),
        .rst(rst),
        .inst_field(inst),
```

```
        .Data_in(Data_in),
        .ImmSell(ImmSell),
        .ALUSrc_B(ALUSrc_B),
        .MemtoReg(MemtoReg),
        .Jump(Jump),
        .Branch(Branch),
        .RegWrite(RegWrite),
        .ALU_Control(ALU_Control),
        .sign(sign),
        .byte_n(byte_n),
        .jump_choose(jump_choose),

        .PC_out  (PC_out),
        .Data_out(Data_out_temp),
        .Addr_out(Addr_out)
    );
endmodule
```

`Controler` 模块

`Controler` :根据指令不同要求解码即可,下面是各个输出属性的解释.代码很繁杂,或许可以通过将指令的不同输入组合然后存入一个 `ROM` 中,然后通过 `OPcode` 和 `Fun7` 和 `Fun3` 确定指令索引以读入 `ROM` 中的数据,但这样会降低代码可读性.

- `wea` : 写入地址选择
- `ImmSell` : 立即数模式选择
- `MemtoReg` : 写入寄存器数据来源选择
- `ALU_Control` : ALU控制信号
- `ALUSrc_B` : ALU第二个操作数来源选择
- `Jump` : 是否跳转
- `Branch` : 是否分支
- `RegWrite` : 是否写入寄存器
- `sign` : 字节/半字读入时是否进行有符号扩展
- `byte_n` : 读入字节数
- `jump_choose` : 跳转地址选择(PC+立即数/ALU输出)

```
    `include "Lab4.vh"
    module Controler (
        input wire [4:0] OPcode,
        input wire MIO_ready,
        input wire [6:0] Fun7,
        input wire [2:0] Fun3,
        output reg CPU_MIO,
        output reg [3:0] wea,
        output reg [`IMM_SEL_WIDTH-1:0] ImmSell,
        output reg [`MEM2REG_WIDTH-1:0] MemtoReg,
        output reg [`ALU_OP_WIDTH-1:0] ALU_Control,
        output reg ALUSrc_B,
        output reg Jump,
        output reg Branch,
```

```
15          output reg RegWrite,
16          output reg sign,
17          output reg [1:0] byte_n,
18          output reg jump_choose
19      );
20        always @(*) begin
21          case (OPcode)
22            `OPCODE_ALU: begin
23              CPU_MIO <= 1'b0;
24              wea <= `WEA_READ;
25              ImmSell <= 3'b0;
26              MemtoReg <= `MEM2REG_ALU;
27              case (Fun3)
28                `FUNC_ADD: ALU_Control <= Fun7[5] ? `ALU_OP_SUB :
  `ALU_OP_ADD;
29                `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
30                `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
31                `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
32                `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
33                `FUNC_OR: ALU_Control <= `ALU_OP_OR;
34                `FUNC_AND: ALU_Control <= `ALU_OP_AND;
35                `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
36                default: ALU_Control <= 4'b0;
37              endcase
38              ALUSrc_B <= 1'b0;
39              Jump <= 1'b0;
40              Branch <= 1'b0;
41              RegWrite <= 1'b1;
42              sign <= 1'b0;
43              byte_n <= `WORD;
44              jump_choose <= `JUMP_PC_IMM;
45            end
46            `OPCODE_ALU_IMM: begin
47              CPU_MIO <= 1'b0;
48              wea <= `WEA_READ;
49              ImmSell <= `IMM_SEL_I;
50              MemtoReg <= `MEM2REG_ALU;
51              case (Fun3)
52                `FUNC_ADD: ALU_Control <= `ALU_OP_ADD;
53                `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
54                `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
55                `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
56                `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
57                `FUNC_OR: ALU_Control <= `ALU_OP_OR;
58                `FUNC_AND: ALU_Control <= `ALU_OP_AND;
59                `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
60                default: ALU_Control <= 4'b0;
61              endcase
62              ALUSrc_B <= 1'b1;
63              Jump <= 1'b0;
64              Branch <= 1'b0;
65              RegWrite <= 1'b1;
66              sign <= 1'b1;
67              byte_n <= `WORD;
68              jump_choose <= `JUMP_PC_IMM;
69            end
```

```
70            `OPCODE_LOAD: begin
71              CPU_MIO <= 1'b1;
72              wea <= `WEA_READ;
73              ImmSell <= `IMM_SEL_I;
74              MemtoReg <= `MEM2REG_MEM;
75              ALU_Control <= 4'b0;
76              ALUSrc_B <= 1'b1;
77              Jump <= 1'b0;
78              Branch <= 1'b0;
79              RegWrite <= 1'b1;
80              sign <= ~(Fun3 == `FUNC_BYTE_UNSIGNED || Fun3 ==
    `FUNC_HALF_UNSIGNED);
81              case (Fun3)
82                `FUNC_BYTE, `FUNC_BYTE_UNSIGNED: byte_n <= `BYTE;
83                `FUNC_HALF, `FUNC_HALF_UNSIGNED: byte_n <= `HALF;
84                `FUNC_WORD: byte_n <= `WORD;
85                default: byte_n <= `WORD;
86              endcase
87              jump_choose <= `JUMP_PC_IMM;
88            end
89            `OPCODE_STORE: begin
90              CPU_MIO <= 1'b1;
91              case (Fun3)
92                `FUNC_BYTE: wea <= `WEA_BYTE;
93                `FUNC_HALF: wea <= `WEA_HALF;
94                `FUNC_WORD: wea <= `WEA_WORD;
95                default: wea <= `WEA_READ;
96              endcase
97              ImmSell <= `IMM_SEL_S;
98              MemtoReg <= `MEM2REG_MEM;
99              ALU_Control <= 4'b0;
100             ALUSrc_B <= 1'b1;
101             Jump <= 1'b0;
102             Branch <= 1'b0;
103             RegWrite <= 1'b0;
104             sign <= 1'b1;
105             byte_n <= `WORD;
106             jump_choose <= `JUMP_PC_IMM;
107           end
108           `OPCODE_BRANCH: begin
109             CPU_MIO <= 1'b0;
110             wea <= `WEA_READ;
111             ImmSell <= `IMM_SEL_B;
112             MemtoReg <= `MEM2REG_ALU;
113             case (Fun3)
114               `FUNC_EQ:  ALU_Control <= `ALU_OP_SUB;
115               `FUNC_NE:  ALU_Control <= `ALU_OP_EQ;
116               `FUNC_LT:  ALU_Control <= `ALU_OP_SGE;
117               `FUNC_GE:  ALU_Control <= `ALU_OP_SLT;
118               `FUNC_LTU: ALU_Control <= `ALU_OP_SGEU;
119               `FUNC_GEU: ALU_Control <= `ALU_OP_SLTU;
120               default:   ALU_Control <= 4'b0;
121             endcase
122             ALUSrc_B <= 1'b0;
123             Jump <= 1'b0;
124             Branch <= 1'b1;
```

```
            RegWrite <= 1'b0;
            sign <= 1'b1;
            byte_n <= `WORD;
          end
          `OPCODE_JAL: begin
            CPU_MIO <= 1'b0;
            wea <= `WEA_READ;
            ImmSell <= `IMM_SEL_J;
            jump_choose <= `JUMP_PC_IMM;
            MemtoReg <= `MEM2REG_PC_PLUS;
            ALU_Control <= `ALU_OP_ADD;
            ALUSrc_B <= 1'b1;
            Jump <= 1'b1;
            Branch <= 1'b0;
            RegWrite <= 1'b1;
            sign <= 1'b1;
            byte_n <= `WORD;
          end
          `OPCODE_JALR: begin
            CPU_MIO <= 1'b0;
            wea <= `WEA_READ;
            ImmSell <= `IMM_SEL_I;
            MemtoReg <= `MEM2REG_PC_PLUS;
            ALU_Control <= `ALU_OP_ADD;
            ALUSrc_B <= 1'b1;
            Jump <= 1'b1;
            Branch <= 1'b0;
            RegWrite <= 1'b1;
            sign <= 1'b1;
            byte_n <= `WORD;
            jump_choose <= `JUMP_ALU;
          end
          `OPCODE_LUI: begin
            CPU_MIO <= 1'b0;
            wea <= `WEA_READ;
            ImmSell <= `IMM_SEL_U;
            MemtoReg <= `MEM2REG_ALU;
            ALU_Control <= `ALU_OP_R2;
            ALUSrc_B <= 1'b1;
            Jump <= 1'b0;
            Branch <= 1'b0;
            RegWrite <= 1'b1;
            sign <= 1'b1;
            byte_n <= `WORD;
            jump_choose <= `JUMP_PC_IMM;
          end
          `OPCODE_AUIPC: begin
            CPU_MIO <= 1'b0;
            wea <= `WEA_READ;
            ImmSell <= `IMM_SEL_U;
            MemtoReg <= `MEM2REG_IMM_PC;
            ALU_Control <= `ALU_OP_ADD;
            ALUSrc_B <= 1'b1;
            Jump <= 1'b0;
            Branch <= 1'b0;
            RegWrite <= 1'b1;
```

```
            sign <= 1'b1;
            byte_n <= `WORD;
            jump_choose <= `JUMP_PC_IMM;
          end
          `OPCODE_PASS: begin
            CPU_MIO <= 1'b0;
            wea <= `WEA_READ;
            ImmSell <= `IMM_SEL_I;
            MemtoReg <= `MEM2REG_ALU;
            ALU_Control <= 4'b0;
            ALUSrc_B <= 1'b0;
            Jump <= 1'b0;
            Branch <= 1'b0;
            RegWrite <= 1'b0;
            sign <= 1'b1;
            byte_n <= `WORD;
            jump_choose <= `JUMP_PC_IMM;
          end
        endcase
      end
    endmodule
```

`extend` 模块

`extend` :根据是否有符号 `sign` 和读取字节数 `byte_n` 来生成读取数据的32位扩展形式

```
`include "Lab4.vh"
module extend (
    input wire [1:0] byte_n,
    input wire [31:0] in,
    input wire sign,
    output [31:0] mem_data
);
  assign mem_data=byte_n==`WORD?in:
                byte_n==`HALF?(sign?{{16{in[15]}},in[15:0]}:
{16'b0,in[15:0]}):
                byte_n==`BYTE?(sign?{{24{in[7]}},in[7:0]}:{24'b0,in[7:0]})
                :32'b0;
endmodule
```

1. `ImmGEN`:根据指令类型 `ImmSell` 进行立即数扩展.

```
`include "Lab4.vh"
module ImmGen (
    input wire [`IMM_SEL_WIDTH-1:0] ImmSell,
    input wire [31:0] inst_field,
    input wire sign,
    output wire [31:0] Imm
);
  wire [31:0] I_Imm, S_Imm, B_Imm, J_Imm, U_Imm;
  assign I_Imm = sign ? {{20{inst_field[31]}}, inst_field[31:20]} : {20'b0,
inst_field[31:20]};
  assign S_Imm = sign?{{20{inst_field[31]}}, inst_field[31:25],
inst_field[11:7]}:{20'b0, inst_field[31:25], inst_field[11:7]};
```

```
  assign B_Imm = sign?{{19{inst_field[31]}}, inst_field[31], inst_field[7], inst_field[30:25], inst_field[11:8], 1'b0}:{19'b0, inst_field[31], inst_field[7], inst_field[30:25], inst_field[11:8], 1'b0};

  assign J_Imm = sign?{{11{inst_field[31]}}, inst_field[19:12], inst_field[20], inst_field[30:21], 1'b0}:{10'b0,inst_field[31],inst_field[19:12], inst_field[20], inst_field[30:21], 1'b0};
  assign U_Imm = {inst_field[31:12], 12'b0};
  assign Imm = (ImmSell == `IMM_SEL_I) ? I_Imm :
               (ImmSell == `IMM_SEL_S) ? S_Imm :
               (ImmSell == `IMM_SEL_B) ? B_Imm :
               (ImmSell == `IMM_SEL_J) ? J_Imm :
               (ImmSell == `IMM_SEL_U) ? U_Imm : 32'b0;
endmodule
```

ALU 模块

ALU:修改原 ALU 模块增添了若干运算.

```
`timescale 1ns / 1ps
module ALU (
    input  [31:0] A,
    input  [31:0] B,
    input  [ 3:0] ALU_operation,
    output [31:0] res,
    output        zero
);
  wire signed [31:0] A_s = $signed(A);
  wire signed [31:0] B_s = $signed(B);
  wire [31:0] A_u = $unsigned(A);
  wire [31:0] B_u = $unsigned(B);
  wire [31:0] result0 = A_s + B_s;
  wire [31:0] result1 = A_s - B_s;
  wire [31:0] result2 = A << B[4:0];
  wire [31:0] result3 = (A_s < B_s) ? 32'b1 : 32'b0;
  wire [31:0] result4 = (A_u < B_u) ? 32'b1 : 32'b0;
  wire [31:0] result5 = A ^ B;
  wire [31:0] result6 = A >> B[4:0];
  wire [31:0] result7 = A_s >>> B_s[4:0];
  wire [31:0] result8 = A | B;
  wire [31:0] result9 = A & B;
  wire [31:0] result10 = ~|result1;
  wire [31:0] result11 = ~|result3;
  wire [31:0] result12 = ~|result4;
  wire [31:0] result13 = B;
  assign res = (ALU_operation==4'b0000)?result0:
               (ALU_operation==4'b0001)?result1:
               (ALU_operation==4'b0010)?result2:
               (ALU_operation==4'b0011)?result3:
               (ALU_operation==4'b0100)?result4:
               (ALU_operation==4'b0101)?result5:
               (ALU_operation==4'b0110)?result6:
               (ALU_operation==4'b0111)?result7:
               (ALU_operation==4'b1000)?result8:
```

```
                (ALU_operation==4'b1001)?result9:
                (ALU_operation==4'b1010)?result10:
                (ALU_operation==4'b1011)?result11:
                (ALU_operation==4'b1100)?result12:
                (ALU_operation==4'b1101)?result13:
                32'b0;
  assign zero = ~(|res) ? 1'b1 : 1'b0;
endmodule
```

`Datapath` 模块

`Datapath`:将各个部件连接起来,实际上CPU的核心部件,决定了CPU运作的逻辑,代码 1 到 68 行主要是部件之间的简单连接,下面分析一下复杂逻辑部分(其中要注意 `extend` 模块的 `in` 即内存读入需要进行便宜操作以便和地址对齐,原因同上).

- 通过 `ALUSrc_B` 选择 `ALU` 的第二个操作数

```
assign adder_2=ALUSrc_B?imm:Rs2_data;
```

- 进行下一条指令地址 `PC_out` 的选择
  - `PC_temp`:用于存储临时的程序计数器值.
  - `PC_add_4`:等于 `PC_temp` 加 4,即不进行跳转,正常下一条指令
  - `PC_imm`:通过 `PC` 相对偏移量进行跳转
  - `Branch_final`:等于 `Branch` 信号和 `ALU` 模块计算为零 `zero` 信号的逻辑与,这用于判断是否需要进行条件分支跳转.
  - `PC_branch`:根据 `Branch_final` 的值选择 `PC_imm` 或 `PC_add_4`
  - `PC_jump`:根据 `Jump` 信号的值选择 `PC_branch` 或者根据 `jump_choose` 的值选择 `ALU_out` 或 `PC_imm`.这用于处理无条件跳转的情况.

```
wire[31:0] PC_temp;
wire[31:0] PC_add_4=PC_temp+4;
wire[31:0] PC_imm=imm+PC_temp;
wire Branch_final=Branch&zero;
wire[31:0] PC_branch=Branch_final?PC_imm:PC_add_4;
wire[31:0] PC_jump=Jump?((jump_choose==`JUMP_ALU)?
ALU_out:PC_imm):PC_branch;
```

- 根据 `MemtoReg` 选择写入寄存器的源数据

```
    assign W_data=MemtoReg==`MEM2REG_MEM?mem_out:
          MemtoReg==`MEM2REG_ALU?ALU_out:
          MemtoReg==`MEM2REG_PC_PLUS?PC_add_4:
          MemtoReg==`MEM2REG_IMM_PC?PC_imm:32'b0;
```

下面是完整代码

```
`include "Lab4.vh"
module DataPath (
    `RegFile_Regs_Outputs
    input  wire clk,
```

```
    input  wire rst,
    input  wire [31:0] inst_field,
    input  wire [31:0] Data_in,
    input  wire [`IMM_SEL_WIDTH-1:0] ImmSell,
    input  wire ALUSrc_B,
    input  wire [`MEM2REG_WIDTH-1:0] MemtoReg,
    input  wire Jump,
    input  wire Branch,
    input  wire RegWrite,
    input  wire [`ALU_OP_WIDTH-1:0] ALU_Control,
    input wire sign,
    input wire[1:0] byte_n,
    input wire jump_choose,
    output wire [31:0] PC_out,
    output wire [31:0] Data_out,
    output wire [31:0] Addr_out
);
reg[31:0] PC=32'h4;
wire[31:0] imm;
ImmGen U1 (
    .ImmSell(ImmSell),
    .inst_field(inst_field),
    .sign(1'b1),
    .Imm(imm)
);
wire [31:0] Rs1_data, Rs2_data;
wire [4:0] Rs1_addr, Rs2_addr, W_addr;
wire[31:0]W_data;
assign Rs1_addr = inst_field[19:15];
assign Rs2_addr = inst_field[24:20];
assign W_addr = inst_field[11:7];
Regs U2 (
    `RegFile_Regs_Arguments
    .clk(clk),
    .rst(rst),
    .Rs1_addr(Rs1_addr),
    .Rs2_addr(Rs2_addr),
    .Wt_addr(W_addr),
    .Wt_data(W_data),
    .RegWrite(RegWrite),
    .Rs1_data(Rs1_data),
    .Rs2_data(Rs2_data)
);
wire[31:0] adder_2;
wire [31:0] ALU_out;
wire zero;
ALU U3 (
    .A(Rs1_data),
    .B(adder_2),
    .ALU_operation(ALU_Control),
    .res(ALU_out),
    .zero(zero)
);
assign Addr_out=ALU_out;
assign Data_out=Rs2_data;
wire [31:0] mem_out;
```

```
61
62  extend U4(
63      .byte_n(byte_n),
64      .in(Data_in>>((Addr_out%4)<<3)),
65      .sign(sign),
66      .mem_data(mem_out)
67  );
68
69  assign adder_2=ALUSrc_B?imm:Rs2_data;
70
71  wire[31:0] PC_temp;
72  wire[31:0] PC_add_4=PC_temp+4;
73  wire[31:0] PC_imm=imm+PC_temp;
74  wire Branch_final=Branch&zero;
75  wire[31:0] PC_branch=Branch_final?PC_imm:PC_add_4;
76  wire[31:0] PC_jump=Jump?((jump_choose==`JUMP_ALU)?ALU_out:PC_imm):PC_branch;
77  always@(posedge clk or posedge rst)
78  begin
79      if(rst)
80      begin
81          PC<=32'hFFFFFFFC;
82      end
83      else
84      begin
85          PC<=PC_jump;
86      end
87  end
88  assign PC_temp=PC;
89  assign PC_out=PC_jump;
90  assign W_data=MemtoReg==`MEM2REG_MEM?mem_out:
91              MemtoReg==`MEM2REG_ALU?ALU_out:
92              MemtoReg==`MEM2REG_PC_PLUS?PC_add_4:
93              MemtoReg==`MEM2REG_IMM_PC?PC_imm:32'b0;
94
95  endmodule
```

头文件宏定义

下面是一些我修改过的头文件宏定义

```
1   /* WHAT'S THIS HEADERFILE FOR? */
2   /*
3     * Reffered to code written by PanZiyue, TA of 2020_CO
4     * Macro for opcode/func3 for RV32I
5     * declaration, inputs/outputs, assignment for debug signals(RegFile)
6   */
7   /* wea */
8   `define WEA_READ 4'b0000
9   `define WEA_BYTE 4'b0001
10  `define WEA_HALF 4'b0011
11  `define WEA_WORD 4'b1111
12  /*JUMP CHOOSE*/
13  `define JUMP_PC_IMM 1'b0
14  `define JUMP_ALU 1'b1
15  /* Byte/Half/Word */
```

```
`define BYTE 2'b00
`define HALF 2'b01
`define WORD 2'b10
/* ALU Operation(Using in Lab1) */
`define ALU_OP_WIDTH  4

`define ALU_OP_ADD      `ALU_OP_WIDTH'd0
`define ALU_OP_SUB      `ALU_OP_WIDTH'd1
`define ALU_OP_SLL      `ALU_OP_WIDTH'd2
`define ALU_OP_SLT      `ALU_OP_WIDTH'd3
`define ALU_OP_SLTU     `ALU_OP_WIDTH'd4
`define ALU_OP_XOR      `ALU_OP_WIDTH'd5
`define ALU_OP_SRL      `ALU_OP_WIDTH'd6
`define ALU_OP_SRA      `ALU_OP_WIDTH'd7
`define ALU_OP_OR       `ALU_OP_WIDTH'd8
`define ALU_OP_AND      `ALU_OP_WIDTH'd9
`define ALU_OP_EQ       `ALU_OP_WIDTH'd10
`define ALU_OP_SGE      `ALU_OP_WIDTH'd11
`define ALU_OP_SGEU     `ALU_OP_WIDTH'd12
`define ALU_OP_R2       `ALU_OP_WIDTH'd13

/*-----------------------------------*/

/* Inst decoding(Using in Lab4/5) */
/* Opcode(5-bits) */
// R-Type
`define OPCODE_ALU      5'b01100
// I-Type
`define OPCODE_ALU_IMM  5'b00100
`define OPCODE_LOAD     5'b00000
`define OPCODE_JALR     5'b11001
`define OPCODE_ENV      5'b11100
`define OPCODE_JALR     5'b11001
// S-Type
`define OPCODE_STORE    5'b01000
// B-Type
`define OPCODE_BRANCH   5'b11000
// J-Type
`define OPCODE_JAL      5'b11011
// U-Type
`define OPCODE_LUI      5'b01101
`define OPCODE_AUIPC    5'b00101

// not use
`define OPCODE_PASS     5'b00000

/* Func3(3-bits) */
// R-Type & I-Type(ALU)
// For R-Type, SUB if inst[30] else ADD
`define FUNC_ADD        3'd0
// Shift Left (Logical)
`define FUNC_SL         3'd1
`define FUNC_SLT        3'd2
`define FUNC_SLTU       3'd3
`define FUNC_XOR        3'd4
// Shift Right Arith if inst[30] else Logical
```

```
`define FUNC_SR          3'd5
`define FUNC_OR          3'd6
`define FUNC_AND         3'd7

// I-Type(Load) & S-Type
`define FUNC_BYTE        3'd0
`define FUNC_HALF        3'd1
`define FUNC_WORD        3'd2
`define FUNC_BYTE_UNSIGNED 3'd4
`define FUNC_HALF_UNSIGNED 3'd5

// B-Type
`define FUNC_EQ          3'd0
`define FUNC_NE          3'd1
`define FUNC_LT          3'd4
`define FUNC_GE          3'd5
`define FUNC_LTU         3'd6
`define FUNC_GEU         3'd7
/*-------------------------------------*/

// JAR
`define FUNC_JALR        3'd0
/* ImmSel signals */
// NOTE: You may add terms in Lab4-3 to implement more inst.
`define IMM_SEL_WIDTH 3

`define IMM_SEL_I    `IMM_SEL_WIDTH'd0
`define IMM_SEL_S    `IMM_SEL_WIDTH'd1
`define IMM_SEL_B    `IMM_SEL_WIDTH'd2
`define IMM_SEL_J    `IMM_SEL_WIDTH'd3
`define IMM_SEL_U    `IMM_SEL_WIDTH'd4
/*-------------------------------------*/

/* Mem2Reg signals */
// NOTE: You may add terms in Lab4-3 to implement more inst.
`define MEM2REG_WIDTH 2

`define MEM2REG_ALU          `MEM2REG_WIDTH'd0
`define MEM2REG_MEM          `MEM2REG_WIDTH'd1
`define MEM2REG_PC_PLUS      `MEM2REG_WIDTH'd2
`define MEM2REG_IMM_PC       `MEM2REG_WIDTH'd3
// `define MEM2REG_IMM          `MEM2REG_WIDTH'd4
`define YOUR_REGS MY_REGS
```

仿真

SCPU 模块仿真

仿真激励代码 SCPU_tb 如下:

```
module SCPU_tb ();
  reg clk;
  reg rst;
  reg [31:0] inst_in;
  reg [31:0] Data_in;
```

```
  wire MemRW;
  wire [31:0] PC_out;  // Next PC
  wire [31:0] Data_out;  //Rs2 or Imm
  wire [31:0] Addr_out;  //ALU_out
  wire [3:0] wea;
  SCPU SCPU (
      .clk(clk),
      .rst(rst),
      .inst_in(inst_in),
      .Data_in(Data_in),
      .MemRW(MemRW),
      .PC_out(PC_out),
      .Data_out(Data_out),
      .Addr_out(Addr_out),
      .wea(wea)
  );
  always begin
    #5 clk = ~clk;
  end
  //     # U型指令
  //     lui x1, 0x625          # 加载上位立即数到0x00625000
  //     auipc x2, 0xA38        # 加载到程序计数器上位立即数到0x00A38000

  //     # 基础算术逻辑指令
  //     add x3, x1, x2       # x3 = x1 + x2
  //     sub x4, x3, x1       # x4 = x3 - x1
  //     xor x5, x4, x3       # x5 = x4 ^ x3
  //     or  x6, x5, x4       # x6 = x5 | x4
  //     and x7, x6, x5       # x7 = x6 & x5
  //     sll x8, x7, x1       # x8 = x7 << x1
  //     srl x9, x8, x2       # x9 = x8 >> x2
  //     sra x10, x9, x1      # x10 = x9 >> x1 (arithmetic)
  //     slt x11,x10,x9
  //     sltu x12,x11,x10

  //     # 立即操作指令
  //     addi x11, x10, 0x10 # x11 = x10 + 0x10
  //     xori x12, x11, 0xFF # x12 = x11 ^ 0xFF
  //     ori  x13, x12, 0x1F # x13 = x12 | 0x1F
  //     andi x14, x13, 0x3F # x14 = x13 & 0x3F
  //     slli x15, x14, 0x2  # x15 = x14 << 0x2
  //     srli x16, x15, 0x2  # x16 = x15 >> 0x2
  //     srai x17, x16, 0x2  # x17 = x16 >> 0x2 (arithmetic)
  //     slti x18,x17,-1
  //     sltiu x18,x17,-1

  //     # 分支跳转与链接指令
  //     jal x18, 76         # 跳转到标签end，并将返回地址保存到x18
  //     jalr x19, x18, 2    # 通过x18跳到返回地址，保存下一条指令地址到x19

  //     # 内存加载与存储指令
  //     lb x20,5(x1)          # 从x1 + arr加载一个字节到x20
  //     lh x21, 5(x1)          # 加载半字
  //     lw x22, 4(x1)          # 加载字
  //     lbu x23, 5(x1)         # 无符号加载字节
  //     lhu x24, 5(x1)         # 无符号加载半字
```

```
62    //     sb x20, 5(x1)           # 将x20的最低字节存回内存
63    //     sh x21, 5(x1)           # 存储半字
64    //     sw x22, 4(x1)           # 存储字
65
66    //     # 分支指令
67    //     beq x21, x22, 28
68    //     bne x21, x22, 28
69    //     blt x21, x22, 24
70    //     bge x21, x22, 16
71    //     bltu x21, x22, 16
72    //     bgeu x21, x22, 8
73    // 006250B7
74    // 00A38117
75    // 002081B3
76    // 40118233
77    // 003242B3
78    // 0042E333
79    // 005373B3
80    // 00139433
81    // 002454B3
82    // 4014D533
83    // 009525B3
84    // 00A5B633
85    // 01050593
86    // 0FF5C613
87    // 01F66693
88    // 03F6F713
89    // 00271793
90    // 0027D813
91    // 40285893
92    // FFF8A913
93    // FFF8B913
94    // 04C0096F
95    // 002909E7
96    // 00508A03
97    // 00509A83
98    // 0040AB03
99    // 0050CB83
100   // 0050DC03
101   // 014082A3
102   // 015092A3
103   // 0160A223
104   // 016A8E63
105   // 016A9E63
106   // 016ACC63
107   // 016AD863
108   // 016AE863
109   // 016AF463
110   initial begin
111     clk = 0;
112     rst = 0;
113     inst_in = 0;
114     Data_in = 0;
115     #10 rst = 1;
116     #10 rst = 0;
117     // # U型指令
```

```
    // lui x1 1573          # 加载上位立即数到0x00625000
    inst_in = 32'h006250B7;
    // MemRW = 0;
    // PC_out = h00000004;
    // Data_out = h00000000;
    // Addr_out = h00625000;
    // wea = 4'b0000;

    // auipc x2, 0xA38       # 加载到程序计数器上位立即数到0x00A38000
    #10 inst_in = 32'h00A38117;
    // MemRW = 0;
    // PC_out = h00000008;
    // Data_out = h00000000;
    // Addr_out = h00A38000;
    // wea = 4'b0000;

    // # 基础算术逻辑指令
    // add x3, x1, x2       # x3 = x1 + x2
    #10 inst_in = 32'h002081B3;
    // MemRW = 0;
    // PC_out = h0000000C;
    // Data_out = h00a38004;
    // Addr_out = h0105d004;
    // wea = 4'b0000;

    // sub x4, x3, x1       # x4 = x3 - x1
    #10 inst_in = 32'h40118233;
    // MemRW = 0;
    // PC_out = h00000010;
    // Data_out = h00625000;
    // Addr_out = h00a38004;
    // wea = 4'b0000;

    // xor x5, x4, x3       # x5 = x4 ^ x3
    #10 inst_in = 32'h003242B3;
    // MemRW = 0;
    // PC_out = h00000014;
    // Data_out = h0105d004;
    // Addr_out = h01a65000;
    // wea = 4'b0000;

    // or  x6, x5, x4       # x6 = x5 | x4
    #10 inst_in = 32'h0042E333;
    // MemRW = 0;
    // PC_out = h00000018;
    // Data_out = h00a38004;
    // Addr_out = h01a7d004;
    // wea = 4'b0000;

    // and x7, x6, x5       # x7 = x6 & x5
    #10 inst_in = 32'h005373B3;
    // MemRW = 0;
    // PC_out = h0000001C;
    // Data_out = h01a65000;
    // Addr_out = h01a65000;
    // wea = 4'b0000;
```

```
174
175     // sll x8, x7, x1      # x8 = x7 << x1
176     #10 inst_in = 32'h00139433;
177     // MemRW = 0;
178     // PC_out = h00000020;
179     // Data_out = h00625000;
180     // Addr_out = h01a65000;
181     // wea = 4'b0000;
182
183     // srl x9, x8, x2      # x9 = x8 >> x2
184     #10 inst_in = 32'h002454B3;
185     // MemRW = 0;
186     // PC_out = h00000024;
187     // Data_out = h00a38004;
188     // Addr_out = h001a6500;
189     // wea = 4'b0000;
190
191     // sra x10, x9, x1     # x10 = x9 >> x1 (arithmetic)
192     #10 inst_in = 32'h4014D533;
193     // MemRW = 0;
194     // PC_out = h00000028;
195     // Data_out = h00625000;
196     // Addr_out = h001a6500;
197     // wea = 4'b0000;
198
199     // slt x11,x10,x9      # x11 = x10 < x9
200     #10 inst_in = 32'h009525B3;
201     // MemRW = 0;
202     // PC_out = h0000002C;
203     // Data_out = h001a6500;
204     // Addr_out = h00000000;
205     // wea = 4'b0000;
206
207     // sltu x12,x11,x10    # x12 = x11 < x10
208     #10 inst_in = 32'h00A5B633;
209     // MemRW = 0;
210     // PC_out = h00000030;
211     // Data_out = h001a6500;
212     // Addr_out = h00000001;
213     // wea = 4'b0000;
214
215     // # 立即操作指令
216     // addi x11, x10, 0x10 # x11 = x10 + 0x10
217     #10 inst_in = 32'h01050593;
218     // MemRW = 0;
219     // PC_out = h00000034;
220     // Data_out = h00000000;
221     // Addr_out = h001a6510;
222     // wea = 4'b0000;
223
224     // xori x12, x11, 0xFF # x12 = x11 ^ 0xFF
225     #10 inst_in = 32'h0FF5C613;
226     // MemRW = 0;
227     // PC_out = h00000038;
228     // Data_out = h00000000;
229     // Addr_out = h001a65ef;
```

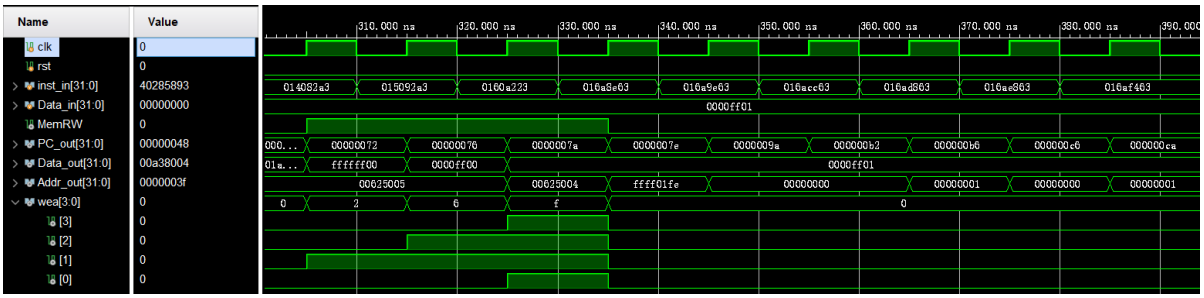
```
    // wea = 4'b0000;

    // ori  x13, x12, 0x1F # x13 = x12 | 0x1F
    #10 inst_in = 32'h01F66693;
    // MemRW = 0;
    // PC_out = h0000003C;
    // Data_out = h00000000;
    // Addr_out = h001a65ff;
    // wea = 4'b0000;

    // andi x14, x13, 0x3F # x14 = x13 & 0x3F
    #10 inst_in = 32'h03F6F713;
    // MemRW = 0;
    // PC_out = h00000040;
    // Data_out = h00000000;
    // Addr_out = h0000003f;
    // wea = 4'b0000;

    // slli x15, x14, 0x2  # x15 = x14 << 0x2
    #10 inst_in = 32'h00271793;
    // MemRW = 0;
    // PC_out = h00000044;
    // Data_out = h00a38004;
    // Addr_out = h000000fc;
    // wea = 4'b0000;

    // srli x16, x15, 0x2  # x16 = x15 >> 0x2
    #10 inst_in = 32'h0027D813;
    // MemRW = 0;
    // PC_out = h00000048;
    // Data_out = h00a38004;
    // Addr_out = h0000003f;
    // wea = 4'b0000;

    // srai x17, x16, 0x2  # x17 = x16 >> 0x2 (arithmetic)
    #10 inst_in = 32'h40285893;
    // MemRW = 0;
    // PC_out = h0000004C;
    // Data_out = h00a38004;
    // Addr_out = h0000000f;
    // wea = 4'b0000;

    // slti x18,x17,-1     # x18 = x17 < -1
    #10 inst_in = 32'hFFF8A913;
    // MemRW = 0;
    // PC_out = h00000050;
    // Data_out = h00000000;
    // Addr_out = h00000000;
    // wea = 4'b0000;

    // sltiu x18,x17,-1    # x18 = x17 < -1
    #10 inst_in = 32'hFFF8B913;
    // MemRW = 0;
    // PC_out = h00000054;
    // Data_out = h00000000;
    // Addr_out = h00000001;
```

```
    // wea = 4'b0000;

    // # 分支跳转与链接指令
    // jal x18, 76        # 跳转到标签end，并将返回地址保存到x18
    #10 inst_in = 32'h04C0096F;
    // MemRW = 0;
    // PC_out = h000000a0;
    // Data_out = h001a6500;
    // Addr_out = h0000004c;
    // wea = 4'b0000;

    // jalr x19, x18, 2    # 通过x18跳到返回地址，保存下一条指令地址到x19
    #10 inst_in = 32'h002909E7;
    // MemRW = 0;
    // PC_out = h0000005a;
    // Data_out = h00a38004;
    // Addr_out = h0000005a;
    // wea = 4'b0000;

    // # 内存加载与存储指令
    // lb x20,5(x1)          # 从x1 + arr加载一个字节到x20
    #10 inst_in = 32'h00508A03;
    Data_in = 32'h0000FF01;
    // MemRW = 1;
    // PC_out = h0000005e;
    // Data_out = h01a65000;
    // Addr_out = h00625005;
    // wea = 4'b0000;

    // lh x21, 5(x1)          # 加载半字
    #10 inst_in = 32'h00509A83;
    // MemRW = 1;
    // PC_out = h00000062;
    // Data_out = h01a65000;
    // Addr_out = h00625005;
    // wea = 4'b0000;

    // lw x22, 4(x1)          # 加载字
    #10 inst_in = 32'h0040AB03;
    // MemRW = 1;
    // PC_out = h00000066;
    // Data_out = h00a38004;
    // Addr_out = h00625004;
    // wea = 4'b0000;

    // lbu x23, 5(x1)         # 无符号加载字节
    #10 inst_in = 32'h0050CB83;
    // MemRW = 1;
    // PC_out = h0000006a;
    // Data_out = h01a65000;
    // Addr_out = h00625005;
    // wea = 4'b0000;

    // lhu x24, 5(x1)         # 无符号加载半字
    #10 inst_in = 32'h0050DC03;
    // MemRW = 1;
```

```
    // PC_out = h0000006e;
    // Data_out = h01a65000;
    // Addr_out = h00625005;
    // wea = 4'b0000;

    // sb x20, 5(x1)          # 将x20的最低字节存回内存
    #10 inst_in = 32'h014082A3;
    // MemRW = 1;
    // PC_out = h00000072;
    // Data_out = hffffff00;
    // Addr_out = h00625005;
    // wea = 4'b0010;

    // sh x21, 5(x1)          # 存储半字
    #10 inst_in = 32'h015092A3;
    // MemRW = 1;
    // PC_out = h00000076;
    // Data_out = 0000ff00;
    // Addr_out = h00625005;
    // wea = 4'b0110;

    // sw x22, 4(x1)          # 存储字
    #10 inst_in = 32'h0160A223;
    // MemRW = 1;
    // PC_out = h0000007a;
    // Data_out = h0000ff01;
    // Addr_out = h00625004;
    // wea = 4'b1111;

    // # 分支指令
    // beq x21, x22, 28
    #10 inst_in = 32'h016A8E63;
    // MemRW = 0;
    // PC_out = h0000007e;
    // Data_out = h0000ff01;
    // Addr_out = hffff01fe;
    // wea = 4'b0000;

    // bne x21, x22, 28
    #10 inst_in = 32'h016A9E63;
    // MemRW = 0;
    // PC_out = h0000009a;
    // Data_out = h0000ff01;
    // Addr_out = h00000000;
    // wea = 4'b0000;

    // blt x21, x22, 24
    #10 inst_in = 32'h016ACC63;
    // MemRW = 0;
    // PC_out = h000000b2;
    // Data_out = h0000ff01;
    // Addr_out = h00000000;
    // wea = 4'b0000;

    // bge x21, x22, 16
    #10 inst_in = 32'h016AD863;
```

```
398     // MemRW = 0;
399     // PC_out = h000000b6;
400     // Data_out = h0000ff01;
401     // Addr_out = h00000001;
402     // wea = 4'b0000;
403 
404     // bltu x21, x22, 16
405     #10 inst_in = 32'h016AE863;
406     // MemRW = 0;
407     // PC_out = h000000c6;
408     // Data_out = h0000ff01;
409     // Addr_out = h00000000;
410     // wea = 4'b0000;
411 
412     // bgeu x21, x22, 8
413     #10 inst_in = 32'h016AF463;
414     // MemRW = 0;
415     // PC_out = h000000ca;
416     // Data_out = h0000ff01;
417     // Addr_out = h00000001;
418     #10;
419     $finish();
420 
421   end
422 endmodule
```

仿真结果如下



仿真结果符合根据指令集编写的正确输出表

相对时间	指令 (inst_in)	MemRW	PC_out	Data_out	Addr_out	wea
0	32'h006250B7	0	h00000004	h00000000	h00625000	0000
10	32'h00A38117	0	h00000008	h00000000	h00A38000	0000
20	32'h002081B3	0	h0000000C	h00a38004	h0105d004	0000
30	32'h40118233	0	h00000010	h00625000	h00a38004	0000
40	32'h003242B3	0	h00000014	h0105d004	h01a65000	0000
50	32'h0042E333	0	h00000018	h00a38004	h01a7d004	0000
60	32'h005373B3	0	h0000001C	h01a65000	h01a65000	0000
70	32'h00139433	0	h00000020	h00625000	h01a65000	0000
80	32'h002454B3	0	h00000024	h00a38004	h001a6500	0000
90	32'h4014D533	0	h00000028	h00625000	h001a6500	0000
100	32'h009525B3	0	h0000002C	h001a6500	h00000000	0000
110	32'h00A5B633	0	h00000030	h001a6500	h00000001	0000
120	32'h01050593	0	h00000034	h00000000	h001a6510	0000
130	32'h0FF5C613	0	h00000038	h00000000	h001a65ef	0000
140	32'h01F66693	0	h0000003C	h00000000	h001a65ff	0000
150	32'h03F6F713	0	h00000040	h00000000	h0000003f	0000
160	32'h00271793	0	h00000044	h00a38004	h000000fc	0000
170	32'h0027D813	0	h00000048	h00a38004	h0000003f	0000
180	32'h40285893	0	h0000004C	h00a38004	h0000000f	0000
190	32'hFFF8A913	0	h00000050	h00000000	h00000000	0000
200	32'hFFF8B913	0	h00000054	h00000000	h00000001	0000
210	32'h04C0096F	0	h000000a0	h001a6500	h0000004c	0000
220	32'h002909E7	0	h0000005a	h00a38004	h0000005a	0000
230	32'h00508A03	1	h0000005e	h01a65000	h00625005	0000
240	32'h00509A83	1	h00000062	h01a65000	h00625005	0000

相对时间	指令 (inst_in)	MemRW	PC_out	Data_out	Addr_out	wea
250	32'h0040AB03	1	h00000066	h00a38004	h00625004	0000
260	32'h0050CB83	1	h0000006a	h01a65000	h00625005	0000
270	32'h0050DC03	1	h0000006e	h01a65000	h00625005	0000
280	32'h014082A3	1	h00000072	hffffff00	h00625005	0010
290	32'h015092A3	1	h00000076	0000ff00	h00625005	0110
300	32'h0160A223	1	h0000007a	h0000ff01	h00625004	1111
310	32'h016A8E63	0	h0000007e	h0000ff01	hffff01fe	0000
320	32'h016A9E63	0	h0000009a	h0000ff01	h00000000	0000
330	32'h016ACC63	0	h000000b2	h0000ff01	h00000000	0000
340	32'h016AD863	0	h000000b6	h0000ff01	h00000001	0000
350	32'h016AE863	0	h000000c6	h0000ff01	h00000000	0000
360	32'h016AF463	0	h000000ca	h0000ff01	h00000001	0000

`datapath` 模块仿真

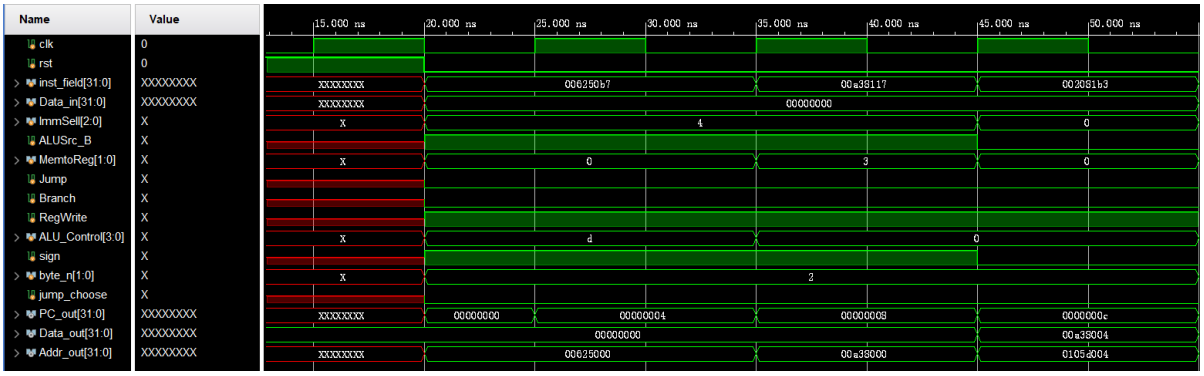
如下是仿真代码

```
// Testbench for DataPath module
`timescale 1ns / 1ps
`include "../sources_1/new/Lab4.vh"
module DataPath_tb;

  // Inputs
  reg clk = 1'b0;
  reg rst;
  reg [31:0] inst_field;
  reg [31:0] Data_in;
  reg [`IMM_SEL_WIDTH-1:0] ImmSell;
  reg ALUSrc_B;
  reg [`MEM2REG_WIDTH-1:0] MemtoReg;
  reg Jump;
  reg Branch;
  reg RegWrite;
  reg [`ALU_OP_WIDTH-1:0] ALU_Control;
  reg sign;
  reg [1:0] byte_n;
  reg jump_choose;

  // Outputs
  wire [31:0] PC_out;
  wire [31:0] Data_out;
  wire [31:0] Addr_out;

  // Instantiate the Unit Under Test (UUT)
```

```
  DataPath uut (
      .clk(clk),
      .rst(rst),
      .inst_field(inst_field),
      .Data_in(Data_in),
      .ImmSell(ImmSell),
      .ALUSrc_B(ALUSrc_B),
      .MemtoReg(MemtoReg),
      .Jump(Jump),
      .Branch(Branch),
      .RegWrite(RegWrite),
      .ALU_Control(ALU_Control),
      .sign(sign),
      .byte_n(byte_n),
      .jump_choose(jump_choose),
      .PC_out(PC_out),
      .Data_out(Data_out),
      .Addr_out(Addr_out)
  );

  // Initialize all inputs
  initial begin
    rst = 0;
    #10 rst = 1;
    #10 rst = 0;
    // lui x1 1573
    inst_field <= 32'h006250B7;
    Data_in <= 32'h00000000;
    ImmSell <= `IMM_SEL_U;
    MemtoReg <= `MEM2REG_ALU;
    ALU_Control <= `ALU_OP_R2;
    ALUSrc_B <= 1'b1;
    Jump <= 1'b0;
    Branch <= 1'b0;
    RegWrite <= 1'b1;
    sign <= 1'b1;
    byte_n <= `WORD;
    jump_choose <= `JUMP_PC_IMM;
    #15;
    // auipc x2, 0xA38
    inst_field <= 32'h00A38117;
    Data_in <= 32'h00000000;
    ImmSell <= `IMM_SEL_U;
    MemtoReg <= `MEM2REG_IMM_PC;
    ALU_Control <= `ALU_OP_ADD;
    ALUSrc_B <= 1'b1;
    Jump <= 1'b0;
    Branch <= 1'b0;
    RegWrite <= 1'b1;
    sign <= 1'b1;
    byte_n <= `WORD;
    jump_choose <= `JUMP_PC_IMM;
    #10;

    // add x3, x1, x2
    inst_field <= 32'h002081B3;
```

```
    Data_in <= 32'h00000000;
    ImmSell <= 3'b0;
    MemtoReg <= `MEM2REG_ALU;
    ALUSrc_B <= 1'b0;
    Jump <= 1'b0;
    Branch <= 1'b0;
    RegWrite <= 1'b1;
    sign <= 1'b0;
    byte_n <= `WORD;
    jump_choose <= `JUMP_PC_IMM;
    ALU_Control <= `ALU_OP_ADD;
    #10;

    $finish;

  end
  // Clock generation
  always #5 clk = ~clk;  // Generate a clock with a period of 10ns

endmodule
```

以下为仿真图像:



该输入为 `SCPU` 仿真的前三条代码,仿真图像符合前表

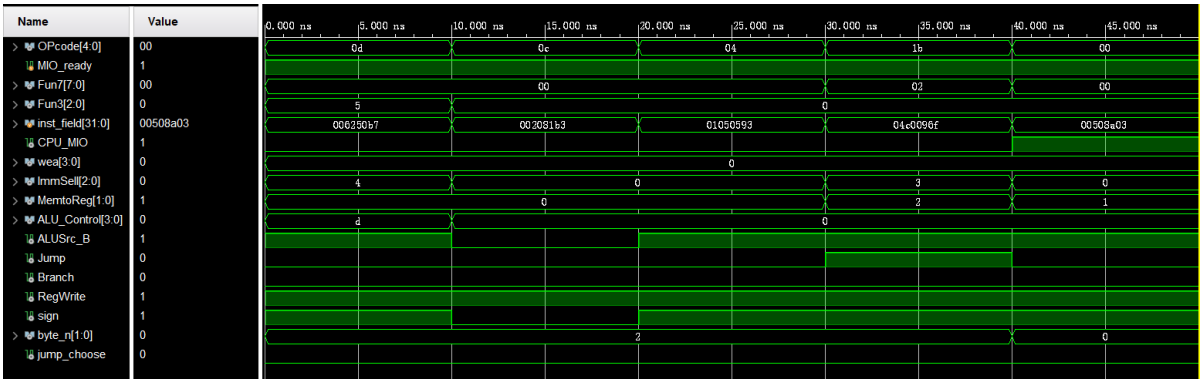
`controler` 模块仿真

以下为仿真代码

```
`timescale 1ns / 1ps
module Controler_tb ();
  wire [4:0] OPcode;
  reg MIO_ready;
  wire [7:0] Fun7;
  wire [2:0] Fun3;
  reg [31:0] inst_field;
  wire CPU_MIO;
  wire [3:0] wea;
  wire [2:0] ImmSell;
  wire [1:0] MemtoReg;
  wire [3:0] ALU_Control;
  wire ALUSrc_B;
  wire Jump;
  wire Branch;
  wire RegWrite;
  wire sign;
```

```
  wire [1:0] byte_n;
  wire jump_choose;
  Controler uut (
      .OPcode(OPcode),
      .MIO_ready(MIO_ready),
      .Fun7(Fun7),
      .Fun3(Fun3),
      .CPU_MIO(CPU_MIO),
      .wea(wea),
      .ImmSell(ImmSell),
      .MemtoReg(MemtoReg),
      .ALU_Control(ALU_Control),
      .ALUSrc_B(ALUSrc_B),
      .Jump(Jump),
      .Branch(Branch),
      .RegWrite(RegWrite),
      .sign(sign),
      .byte_n(byte_n),
      .jump_choose(jump_choose)
  );
  assign OPcode = inst_field[6:2];
  assign Fun7   = inst_field[31:25];
  assign Fun3   = inst_field[14:12];
  initial begin
    // lui x1 1573
    MIO_ready  = 1'b1;
    inst_field = 32'h006250B7;
    #10;
    // add x3, x1, x2
    inst_field = 32'h002081B3;
    #10;
    // addi x11, x10, 0x10
    inst_field = 32'h01050593;
    #10;
    // jal x18, 76
    inst_field = 32'h04C0096F;
    #10;
    // lb x20,5(x1)
    inst_field = 32'h00508A03;
    #10;
    $finish;
  end
endmodule
```

以下为仿真图像:



仿真第一条指令为 `lui` ,可以看到输出 `wea` 为0不写入, `ImmSell` 为4是U型指令, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为1选取立即数作为第二个操作数, `ALU_Control` 为d表示 `ALU` 的操作为保留第二操作数输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;仿真第二条指令为 `add` ,可以看到输出 `wea` 为0不写入, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为0选取寄存器作为第二个操作数, `ALU_Control` 为0表示 `ALU` 的操作为将第一第二操作数相加输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;仿真第三条指令为 `add` ,可以看到输出 `wea` 为0不写入, `ImmSell` 为1是I型指令, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为1选取立即数作为第二个操作数, `ALU_Control` 为0表示 `ALU` 的操作为将第一第二操作数相加输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;其余指令照前分析均符合预期.

实验结果与分析

可以看到x31为666说明通过验收代码.

浙江大学实验报告 Lab4-4

专业：计算机科学与技术　姓名：仇国智　学号：3220102181　日期：2024/4/10
课程名称：计算机组成与设计　实验名称:实现单周期 CPU-异常与中断　指导老师：刘海风　成绩：

操作方法与实验步骤

编写源文件

在4-3的基础上增加了 `RV_INT` , `CSRRegs` 模块.
`CSRRegs` 模块:

```
1  `include "Lab4.vh"
2  module CSRRegs (
3      input                                clk,
4      input                                rst,
5      input       [                  11:0] raddr,
6      input       [                  11:0] waddr,
7      input       [                  31:0] wdata,
8      input                                csr_w,
9      input       [`CSR_CHANGE_MODE_WIDTH-1:0] csr_wsc_mode,
10     input       [                  31:0] mepc_bypasss_in,
11     input       [                  31:0] mcause_bypass_in,
12     input       [                  31:0] mtval_bypass_in,
13     input       [                  31:0] mstatus_bypass_in,
14     output      [                  31:0] rdata,
15     output reg  [                  31:0] mepc,
16     output reg  [                  31:0] mcause,
```

```
    output reg [                    31:0] mtval,
    output reg [                    31:0] mtvec,
    output reg [                    31:0] mstatus
);
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      mepc <= 32'h0;
      mcause <= 32'h0;
      mtval <= 32'h0;
      mtvec <= 32'h44;
      mstatus <= 32'h8;
    end else if (csr_w) begin
      case (csr_wsc_mode)
        `CSR_WRITE:
        case (waddr)
          `MEPC: mepc <= wdata;
          `MCAUSE: mcause <= wdata;
          `MTVAL: mtval <= wdata;
          `MTVEC: mtvec <= wdata;
          `MSTATUS: mstatus <= wdata;
        endcase
        `CSR_OR:
        case (waddr)
          `MEPC: mepc <= mepc | wdata;
          `MCAUSE: mcause <= mcause | wdata;
          `MTVAL: mtval <= mtval | wdata;
          `MTVEC: mtvec <= mtvec | wdata;
          `MSTATUS: mstatus <= mstatus | wdata;
        endcase
        `CSR_CLEAR:
        case (waddr)
          `MEPC: mepc <= mepc & !wdata;
          `MCAUSE: mcause <= mcause & !wdata;
          `MTVAL: mtval <= mtval & !wdata;
          `MTVEC: mtvec <= mtvec & !wdata;
          `MSTATUS: mstatus <= mstatus & !wdata;
        endcase
        `CSR_BYPASS:
        begin
            mepc <= mepc_bypasss_in;
            mcause <= mcause_bypass_in;
            mtval <= mtval_bypass_in;
            mstatus <= mstatus_bypass_in;
        end
      endcase
    end
  end
assign rdata = (raddr == `MEPC) ? mepc :
               (raddr == `MCAUSE) ? mcause :
               (raddr == `MTVAL) ? mtval :
               (raddr == `MTVEC) ? mtvec :
               (raddr == `MSTATUS) ? mstatus :
               32'h0;
endmodule
```

该模块用于储存CSR寄存器的值,并且可以通过`csr_wsc_mode`来选择写入模式. `csr_wsc_mode`的值有`CSR_WRITE`,`CSR_OR`,`CSR_CLEAR`,`CSR_BYPASS`四种,分别对应写入,设定,清除,和批量写入模式.批量写入时,会将`mepc_bypasss_in`,`mcause_bypass_in`,`mtval_bypass_in`,`mstatus_bypass_in`的值写入到对应的寄存器中.
`RV_INT`模块:

```
`include "Lab4.vh"
module RV_INT (
    input clk,
    input rst,
    input INT,  // 外部中断信号
    input ecall,  // ECALL 指令
    input mret,  // MRET 指令
    input illegal_inst,  // 非法指令信号
    // input l_access_fault,  // 数据访存不对齐
    // input j_access_fault,  // 跳转地址不对齐
    input [31:0] PC,  // PC
    input [11:0] raddr,
    input [11:0] waddr,
    input [31:0] wdata,
    input csr_w,
    input wire [31:0] inst_field,
    input [`CSR_CHANGE_MODE_WIDTH-1:0] csr_wsc_mode,
    output trap_normal_change,  // 用于指示trap和正常流程的转换
    output [31:0] PC_change,  // 用于指示PC流程的切换的PC
    output [31:0] rdata,  //CSR的值
    `CSR_OUTPUTS
);
  wire [31:0] mepc_bypasss_in;
  wire [31:0] mcause_bypass_in;
  wire [31:0] mtval_bypass_in;
  wire [31:0] mstatus_bypass_in;
  wire [11:0] csr_wsc_mode_real;
  wire csr_w_real;
  CSRRegs U1 (
      .clk(clk),
      .rst(rst),
      .raddr(raddr),
      .waddr(waddr),
      .wdata(wdata),
      .csr_w(csr_w_real),
      .csr_wsc_mode(csr_wsc_mode_real),
      .mepc_bypasss_in(mepc_bypasss_in),
      .mcause_bypass_in(mcause_bypass_in),
      .mtval_bypass_in(mtval_bypass_in),
      .mstatus_bypass_in(mstatus_bypass_in),
      .rdata(rdata),
      .mepc(mepc),
      .mcause(mcause),
      .mtval(mtval),
      .mtvec(mtvec),
      .mstatus(mstatus)
  );
  wire trap_start=mstatus[`MSTATUS_MIE]&&
(INT||ecall||illegal_inst);//||l_access_fault||j_access_fault);
```

```
  wire trap_exit = mret;
  assign trap_normal_change = trap_exit || trap_start;
  assign csr_wsc_mode_real = trap_start || trap_exit ? `CSR_BYPASS :
csr_wsc_mode;
  assign csr_w_real = trap_start || trap_exit || csr_w;
  assign PC_change = trap_start ? {mtvec[31:2], 2'b00} : trap_exit ? mepc :
PC;
  assign mepc_bypasss_in = trap_start ? PC : mepc;
  assign mcause_bypass_in = trap_start ? (INT ? `MCAUSE_INT : ecall ?
`MCAUSE_ECALL :illegal_inst? `MCAUSE_ILL_INST:32'h0) : mcause;
  assign mtval_bypass_in = trap_start ? (illegal_inst ? inst_field : 32'h0)
: mtval;
  assign mstatus_bypass_in = trap_start ?
  {mstatus[31:`MSTATUS_MPIE+1], mstatus[`MSTATUS_MIE],
mstatus[`MSTATUS_MPIE-1:`MSTATUS_MIE+1], 1'b0, mstatus[`MSTATUS_MIE-1:0]}:
  trap_exit?
  {mstatus[31:`MSTATUS_MPIE+1], mstatus[`MSTATUS_MPIE],
mstatus[`MSTATUS_MPIE-1:`MSTATUS_MIE+1], 1'b1, mstatus[`MSTATUS_MIE-1:0]}:
  mstatus;

endmodule
```

该模块用于处理异常和中断,当`mstatus`的`MIE`位为1时,并且有中断信号,或者是`ecall`指令,或者是非法指令时,会触发异常,并且批量写入CSR寄存器的值(此时写入`mstatus`的的`MIE`位为`0`,阻止`trap`处理过程中再次触发异常),同时跳转`PC`的值到`mtvec`中的值.当`mret`指令到来时,会从`mepc`中恢复PC的值,同时批量恢复CSR寄存器的值.注意到只有当中断发生和结束时,才会批量写入CSR寄存器的值同时跳转PC的值,而在其他情况下,只会按地址读取修改CSR寄存器的值.故需要`trap_normal_change`来指示非平凡状态,其余`trap`指令的执行与正常流程一致.

编写异常处理软件

```
00100093 addi x1,x0,1
00100113 addi x2,x0,1
001101B3 add x3,x2,x1
003100B3 add x1,x2,x3
00308133 add x2,x1,x3
001101B3 add x3,x2,x1
00000073 ecall
003100B3 add x1,x2,x3
00308133 add x2,x1,x3
001101B3 add x3,x2,x1
#INT
003100B3 add x1,x2,x3
00308133 add x2,x1,x3
001101B3 add x3,x2,x1
FFFFFFFF add x0,x0,x0 #ill inst
003100B3 add x1,x2,x3
00308133 add x2,x1,x3
001101B3 add x3,x2,x1
30002FF3 csrrs x31,0x300,x0 # MSTATUS
34102F73 csrrs x30,0x341,x0 # MEPC
34202EF3 csrrs x29,0x342,x0 # MCAUSE
34302E73 csrrs x28,0x343,x0 # MTVAL
30502DF3 csrrs x27,0x305,x0 # MTVEC
```

```
80000D37 li x26,0x8000000b
00BD0D13 li x26,0x8000000b
01DD0463 beq x26,x29,else
004F0F13 addi x30,x30,4 #MEPC=MEPC+4
341F1073 else: csrrw x0,0x341,x30 # MEPC=MEPC
30505073 cssrwi 0x,0x305,0
305D9073 csrrw x0,0x305,x27
342FD073 csrrsi x0,0x342,-1
342FF073 csrrci x0,0x342,-1
343F3D73 csrrc x26,0x343,x30
30200073 mret
```

这里要注意通过读出的`mcause`的值来判断异常的类型,以确定`mepc`是否需要加4.

编写仿真激励文件

此时已经将软件代码写入到了`ROM`中,下面为仿真激励模块:

```
`include "../sources_1/new/Lab4.vh"
module exception_tb ();
    `CSR_DECLARATION;
    reg clk = 1'b0;
    reg rst;
    wire [31:0] inst_in;
    reg [31:0] Data_in;
    reg INT;
    wire MemRW;
    wire [31:0] PC_out;
    wire [31:0] Data_out;
    wire [31:0] Addr_out;
    wire [3:0] wea;
    wire [`CSR_CHANGE_MODE_WIDTH-1:0] csr_wsc_mode;
    wire CPU_MIO;
    wire trap_normal_change;
    wire mret;
    wire ecall;
    wire [31:0]`YOUR_REGS[31:0];
    `RegFile_Regs_Declaration
    SCPU uut (
         .clk(clk),
         .rst(rst),
         .MIO_ready(MIO_ready),
         .inst_in(inst_in),
         .Data_in(Data_in),
         .INT(INT),
         .CPU_MIO(CPU_MIO),
         .MemRW(MemRW),
         .PC_out(PC_out),
         .Data_out(Data_out),
         .Addr_out(Addr_out),
         .wea(wea),
         .csr_wsc_mode(csr_wsc_mode),
         .trap_normal_change(trap_normal_change),
            .mret(mret),
            .ecall(ecall),
```

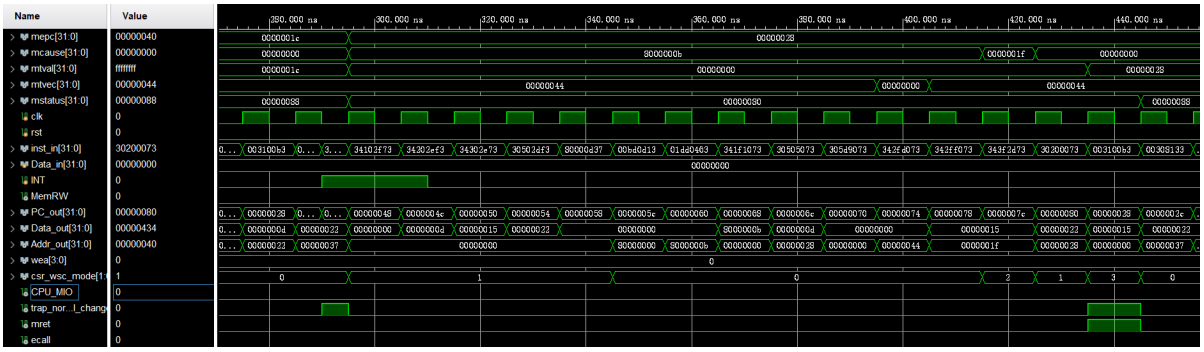
```
        `RegFile_Regs_Arguments
        `CSR_ARGUMENTS
    );
    `RegFile_Regs_Assignments
    U2 U2 (
      .a  (PC_out[11:2]),
      .spo(inst_in)
  );
always begin
    #5 clk = ~clk;
end
initial begin
    rst = 0;
    INT=0;
    Data_in=32'h00000000;
    #10;
    rst = 1;
    #10;
    rst = 0;
    #270;
    INT=1;
    #20;
    INT=0;
end
endmodule
```

下面为仿真激励图像及其分析:

`ecall` 指令触发异常:

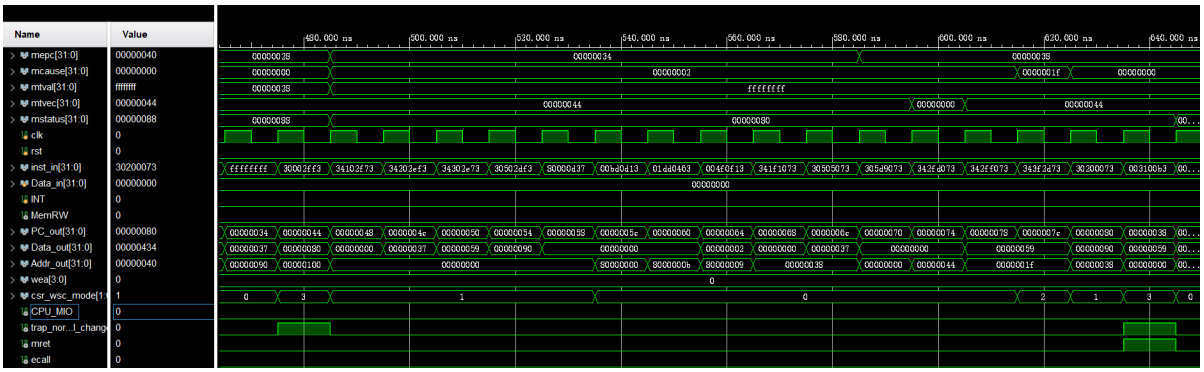
`trap` 的进入和退出实现了了 `PC` +4.在开始时, `ecall` 信号处于高位,对应的异常代码 8000000b 存入 `mcause` 中,并且 `mstatus` 的 `MIE` 位被清除,阻止再次触发异常.同时 `mepc` 被写入 `PC` 的值,跳转到 `mtvec` 中指向的值.在期间实现了 `PC` +4.在结束时, `mepc` 的值被写入 `PC` 的值,并且 `mstatus` 的 `MIE` 位被恢复,允许再次触发异常.

`INT` 信号触发异常:



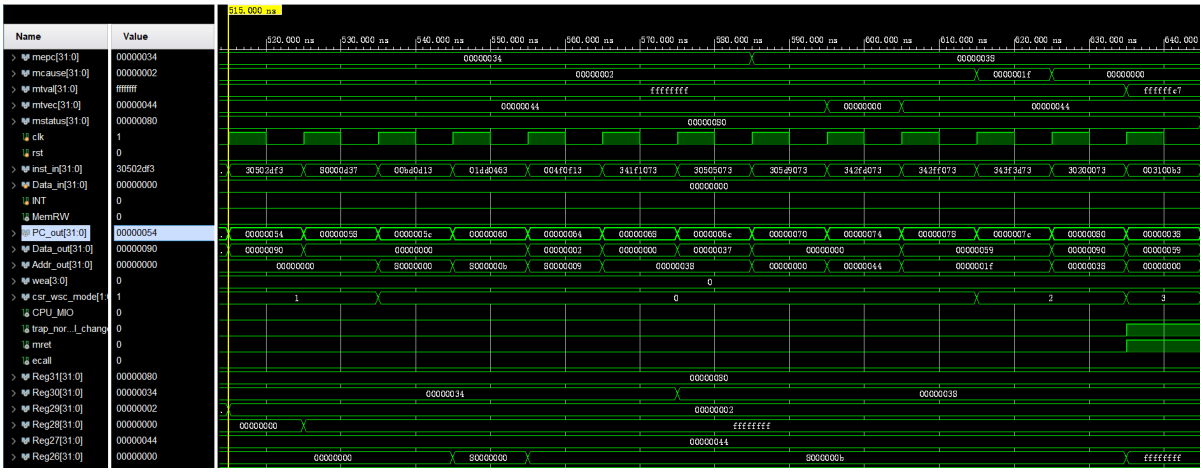
分析同上,但是注意此时 `PC` 未进行递增,同时 `INT` 亮起超过一个周期,期间未发生二次 `trap` .

非法指令触发异常:



分析同上,但注意此时 `mtval` 中存入了非法指令的值 `FFFFFFFF`.

CSR寄存器的读写:



`csrrs` 执行使得 `x27` 的值为 `0x00000044` ,同时 `mtval` 与0相与不发生改变, `csrrw` 执行使得 `x30` 的值为 `x30` 寄存器的值 `0x00000038` ,

`cssrwi` 执行使得 `mtvec` 的值为0, `csrrw` 执行使得 `mtvec` 的值为 `x27` 的值 `0x00000044` , `csrrsi` 执行使得 `mcause` 的值为 `0x0000001F` (即 `0x00000002` 与 `0x0000001F` 取或), `csrrci` 执行使得 `mcause` 的值为 `0x00000000` (即 `0x0000001F` 与 `0xFFFFFFD0` 取与), `csrrc` 执行使得 `mtval` 的值为 `0xFFFFFFFF` 与 `x30` 的 `0x00000038` 取反后的值取与得 `0xFFFFFFC7` .

其余修改模块

下面是其它一些修改过的模块

`Lab4.vh`

```
/* WHAT'S THIS HEADERFILE FOR? */
/*
 * Reffered to code written by PanZiyue, TA of 2020_CO
 * Macro for opcode/func3 for RV32I
 * declaration, inputs/outputs, assignment for debug signals(RegFile)
*/
// CSR ouput
`define CSR_OUTPUTS \
      output wire [31:0] mepc,\
      output wire [31:0] mcause,\
      output wire [31:0] mtval,\
      output wire [31:0] mtvec,\
      output wire [31:0] mstatus
// CSR ARGUMENTS
`define CSR_ARGUMENTS \
      .mepc(mepc),\
      .mcause(mcause),\
      .mtval(mtval),\
```

```
      .mtvec(mtvec),\
      .mstatus(mstatus)
// CSR DECLARATION
`define CSR_DECLARATION \
      wire [31:0] mepc;\
      wire [31:0] mcause;\
      wire [31:0] mtval;\
      wire [31:0] mtvec;\
      wire [31:0] mstatus
//direct inst
`define ECALL 15'b0000000_000_11100
`define MRET 15'b0011000_000_11100
/*MCAUSE*/
`define MCAUSE_ECALL 32'h8000_0003
`define MCAUSE_INT 32'h8000_000b
`define MCAUSE_ILL_INST 32'h0000_0002
/*MASTATUS*/
`define MSTATUS_MIE 5'd3
`define MSTATUS_MPIE 5'd7
/*CSR*/
`define MSTATUS 12'h300
`define MEPC 12'h341
`define MCAUSE 12'h342
`define MTVAL 12'h343
`define MTVEC 12'h305

/*CSR change mode*/
`define CSR_CHANGE_MODE_WIDTH 2
`define CSR_WRITE 2'b00
`define CSR_OR 2'b01
`define CSR_CLEAR 2'b10
`define CSR_BYPASS 2'b11
/* wea */
`define WEA_READ 4'b0000
`define WEA_BYTE 4'b0001
`define WEA_HALF 4'b0011
`define WEA_WORD 4'b1111
/*JUMP CHOOSE*/
`define JUMP_PC_IMM 1'b0
`define JUMP_ALU 1'b1
/* Byte/Half/Word */
`define BYTE 2'b00
`define HALF 2'b01
`define WORD 2'b10
/* ALU Operation(Using in Lab1) */
`define ALU_OP_WIDTH 4

`define ALU_OP_ADD `ALU_OP_WIDTH'd0
`define ALU_OP_SUB `ALU_OP_WIDTH'd1
`define ALU_OP_SLL `ALU_OP_WIDTH'd2
`define ALU_OP_SLT `ALU_OP_WIDTH'd3
`define ALU_OP_SLTU `ALU_OP_WIDTH'd4
`define ALU_OP_XOR `ALU_OP_WIDTH'd5
`define ALU_OP_SRL `ALU_OP_WIDTH'd6
`define ALU_OP_SRA `ALU_OP_WIDTH'd7
`define ALU_OP_OR `ALU_OP_WIDTH'd8
```

```
`define ALU_OP_AND `ALU_OP_WIDTH'd9
`define ALU_OP_EQ `ALU_OP_WIDTH'd10
`define ALU_OP_SGE `ALU_OP_WIDTH'd11
`define ALU_OP_SGEU `ALU_OP_WIDTH'd12
`define ALU_OP_R2 `ALU_OP_WIDTH'd13
`define ALU_OP_R1 `ALU_OP_WIDTH'd14

/*-----------------------------------*/

/* Inst decoding(Using in Lab4/5) */
/* Opcode(5-bits) */
// R-Type
`define OPCODE_ALU 5'b01100
// I-Type
`define OPCODE_ALU_IMM 5'b00100
`define OPCODE_LOAD 5'b00000
`define OPCODE_JALR 5'b11001
`define OPCODE_ENV 5'b11100
`define OPCODE_JALR 5'b11001
// S-Type
`define OPCODE_STORE 5'b01000
// B-Type
`define OPCODE_BRANCH 5'b11000
// J-Type
`define OPCODE_JAL 5'b11011
// U-Type
`define OPCODE_LUI 5'b01101
`define OPCODE_AUIPC 5'b00101

`define OPCODE_CSR 5'b11100
// not use
`define OPCODE_PASS 5'b00000

/* Func3(3-bits) */
// R-Type & I-Type(ALU)
// For R-Type, SUB if inst[30] else ADD
`define FUNC_ADD 3'd0
// Shift Left (Logical)
`define FUNC_SL 3'd1
`define FUNC_SLT 3'd2
`define FUNC_SLTU 3'd3
`define FUNC_XOR 3'd4
// Shift Right Arith if inst[30] else Logical
`define FUNC_SR 3'd5
`define FUNC_OR 3'd6
`define FUNC_AND 3'd7

// I-Type(Load) & S-Type
`define FUNC_BYTE 3'd0
`define FUNC_HALF 3'd1
`define FUNC_WORD 3'd2
`define FUNC_BYTE_UNSIGNED 3'd4
`define FUNC_HALF_UNSIGNED 3'd5

// B-Type
`define FUNC_EQ 3'd0
```

```
`define FUNC_NE 3'd1
`define FUNC_LT 3'd4
`define FUNC_GE 3'd5
`define FUNC_LTU 3'd6
`define FUNC_GEU 3'd7
/*-----------------------------------*/

// CSR FUNC
`define FUNC_CSRRW 3'd1
`define FUNC_CSRRS 3'd2
`define FUNC_CSRRC 3'd3
`define FUNC_CSRRWI 3'd5
`define FUNC_CSRRSI 3'd6
`define FUNC_CSRRCI 3'd7

// JAR
`define FUNC_JALR 3'd0
/* ImmSel signals */
// NOTE: You may add terms in Lab4-3 to implement more inst.
`define IMM_SEL_WIDTH 3

`define IMM_SEL_I `IMM_SEL_WIDTH'd0
`define IMM_SEL_S `IMM_SEL_WIDTH'd1
`define IMM_SEL_B `IMM_SEL_WIDTH'd2
`define IMM_SEL_J `IMM_SEL_WIDTH'd3
`define IMM_SEL_U `IMM_SEL_WIDTH'd4
`define IMM_SEL_CSR `IMM_SEL_WIDTH'd6
/*-----------------------------------*/

/* Mem2Reg signals */
// NOTE: You may add terms in Lab4-3 to implement more inst.
`define MEM2REG_WIDTH 3

`define MEM2REG_ALU `MEM2REG_WIDTH'd0
`define MEM2REG_MEM `MEM2REG_WIDTH'd1
`define MEM2REG_PC_PLUS `MEM2REG_WIDTH'd2
`define MEM2REG_IMM_PC `MEM2REG_WIDTH'd3
`define MEM2REG_CSR `MEM2REG_WIDTH'd4
// `define MEM2REG_IMM         `MEM2REG_WIDTH'd4
/*-----------------------------------*/

/*-----------------------------*/
/******* generated code *******/
/*-----------------------------*/

/* RegFiles Ports & debug signals */
/* NOTE:
 * AFTER you change "..." in macro YOUR_REGS to the name of your reg-array
in module Regs, such as regs,
 * you need to *uncomment* the line "`define YOUR_REGS regs" below for
using this set of macros
*/
// `define YOUR_REGS ...
`define YOUR_REGS MY_REGS
`ifdef YOUR_REGS

```

```
`define RegFile_Regs_Outputs \
    output [31:0] Reg00, \
    output [31:0] Reg01, \
    output [31:0] Reg02, \
    output [31:0] Reg03, \
    output [31:0] Reg04, \
    output [31:0] Reg05, \
    output [31:0] Reg06, \
    output [31:0] Reg07, \
    output [31:0] Reg08, \
    output [31:0] Reg09, \
    output [31:0] Reg10, \
    output [31:0] Reg11, \
    output [31:0] Reg12, \
    output [31:0] Reg13, \
    output [31:0] Reg14, \
    output [31:0] Reg15, \
    output [31:0] Reg16, \
    output [31:0] Reg17, \
    output [31:0] Reg18, \
    output [31:0] Reg19, \
    output [31:0] Reg20, \
    output [31:0] Reg21, \
    output [31:0] Reg22, \
    output [31:0] Reg23, \
    output [31:0] Reg24, \
    output [31:0] Reg25, \
    output [31:0] Reg26, \
    output [31:0] Reg27, \
    output [31:0] Reg28, \
    output [31:0] Reg29, \
    output [31:0] Reg30, \
    output [31:0] Reg31,

`define RegFile_Regs_Assignments \
    assign Reg00 = `YOUR_REGS[0]; \
    assign Reg01 = `YOUR_REGS[1]; \
    assign Reg02 = `YOUR_REGS[2]; \
    assign Reg03 = `YOUR_REGS[3]; \
    assign Reg04 = `YOUR_REGS[4]; \
    assign Reg05 = `YOUR_REGS[5]; \
    assign Reg06 = `YOUR_REGS[6]; \
    assign Reg07 = `YOUR_REGS[7]; \
    assign Reg08 = `YOUR_REGS[8]; \
    assign Reg09 = `YOUR_REGS[9]; \
    assign Reg10 = `YOUR_REGS[10]; \
    assign Reg11 = `YOUR_REGS[11]; \
    assign Reg12 = `YOUR_REGS[12]; \
    assign Reg13 = `YOUR_REGS[13]; \
    assign Reg14 = `YOUR_REGS[14]; \
    assign Reg15 = `YOUR_REGS[15]; \
    assign Reg16 = `YOUR_REGS[16]; \
    assign Reg17 = `YOUR_REGS[17]; \
    assign Reg18 = `YOUR_REGS[18]; \
    assign Reg19 = `YOUR_REGS[19]; \
    assign Reg20 = `YOUR_REGS[20]; \
```

```
    assign Reg21 = `YOUR_REGS[21]; \
    assign Reg22 = `YOUR_REGS[22]; \
    assign Reg23 = `YOUR_REGS[23]; \
    assign Reg24 = `YOUR_REGS[24]; \
    assign Reg25 = `YOUR_REGS[25]; \
    assign Reg26 = `YOUR_REGS[26]; \
    assign Reg27 = `YOUR_REGS[27]; \
    assign Reg28 = `YOUR_REGS[28]; \
    assign Reg29 = `YOUR_REGS[29]; \
    assign Reg30 = `YOUR_REGS[30]; \
    assign Reg31 = `YOUR_REGS[31];

`define RegFile_Regs_Arguments \
    .Reg00(Reg00), \
    .Reg01(Reg01), \
    .Reg02(Reg02), \
    .Reg03(Reg03), \
    .Reg04(Reg04), \
    .Reg05(Reg05), \
    .Reg06(Reg06), \
    .Reg07(Reg07), \
    .Reg08(Reg08), \
    .Reg09(Reg09), \
    .Reg10(Reg10), \
    .Reg11(Reg11), \
    .Reg12(Reg12), \
    .Reg13(Reg13), \
    .Reg14(Reg14), \
    .Reg15(Reg15), \
    .Reg16(Reg16), \
    .Reg17(Reg17), \
    .Reg18(Reg18), \
    .Reg19(Reg19), \
    .Reg20(Reg20), \
    .Reg21(Reg21), \
    .Reg22(Reg22), \
    .Reg23(Reg23), \
    .Reg24(Reg24), \
    .Reg25(Reg25), \
    .Reg26(Reg26), \
    .Reg27(Reg27), \
    .Reg28(Reg28), \
    .Reg29(Reg29), \
    .Reg30(Reg30), \
    .Reg31(Reg31),

`define RegFile_Regs_Declaration \
    wire [31:0] Reg00; \
    wire [31:0] Reg01; \
    wire [31:0] Reg02; \
    wire [31:0] Reg03; \
    wire [31:0] Reg04; \
    wire [31:0] Reg05; \
    wire [31:0] Reg06; \
    wire [31:0] Reg07; \
    wire [31:0] Reg08; \
```

```
    wire [31:0] Reg09; \
    wire [31:0] Reg10; \
    wire [31:0] Reg11; \
    wire [31:0] Reg12; \
    wire [31:0] Reg13; \
    wire [31:0] Reg14; \
    wire [31:0] Reg15; \
    wire [31:0] Reg16; \
    wire [31:0] Reg17; \
    wire [31:0] Reg18; \
    wire [31:0] Reg19; \
    wire [31:0] Reg20; \
    wire [31:0] Reg21; \
    wire [31:0] Reg22; \
    wire [31:0] Reg23; \
    wire [31:0] Reg24; \
    wire [31:0] Reg25; \
    wire [31:0] Reg26; \
    wire [31:0] Reg27; \
    wire [31:0] Reg28; \
    wire [31:0] Reg29; \
    wire [31:0] Reg30; \
    wire [31:0] Reg31;

`define VGA_RegFile_Inputs \
    input [31:0] Reg00, \
    input [31:0] Reg01, \
    input [31:0] Reg02, \
    input [31:0] Reg03, \
    input [31:0] Reg04, \
    input [31:0] Reg05, \
    input [31:0] Reg06, \
    input [31:0] Reg07, \
    input [31:0] Reg08, \
    input [31:0] Reg09, \
    input [31:0] Reg10, \
    input [31:0] Reg11, \
    input [31:0] Reg12, \
    input [31:0] Reg13, \
    input [31:0] Reg14, \
    input [31:0] Reg15, \
    input [31:0] Reg16, \
    input [31:0] Reg17, \
    input [31:0] Reg18, \
    input [31:0] Reg19, \
    input [31:0] Reg20, \
    input [31:0] Reg21, \
    input [31:0] Reg22, \
    input [31:0] Reg23, \
    input [31:0] Reg24, \
    input [31:0] Reg25, \
    input [31:0] Reg26, \
    input [31:0] Reg27, \
    input [31:0] Reg28, \
    input [31:0] Reg29, \
    input [31:0] Reg30, \
```

```
353     input [31:0] Reg31,
354
355 `define VGA_RegFile_Arguments \
356     .x0 (Reg00), \
357     .ra (Reg01), \
358     .sp (Reg02), \
359     .gp (Reg03), \
360     .tp (Reg04), \
361     .t0 (Reg05), \
362     .t1 (Reg06), \
363     .t2 (Reg07), \
364     .s0 (Reg08), \
365     .s1 (Reg09), \
366     .a0 (Reg10), \
367     .a1 (Reg11), \
368     .a2 (Reg12), \
369     .a3 (Reg13), \
370     .a4 (Reg14), \
371     .a5 (Reg15), \
372     .a6 (Reg16), \
373     .a7 (Reg17), \
374     .s2 (Reg18), \
375     .s3 (Reg19), \
376     .s4 (Reg20), \
377     .s5 (Reg21), \
378     .s6 (Reg22), \
379     .s7 (Reg23), \
380     .s8 (Reg24), \
381     .s9 (Reg25), \
382     .s10(Reg26), \
383     .s11(Reg27), \
384     .t3 (Reg28), \
385     .t4 (Reg29), \
386     .t5 (Reg30), \
387     .t6 (Reg31),
388
389 `endif  // YOUR_REGS
390
391 `define VGA_Debug_Signals_Inputs \
392     input [31:0] pc, \
393     input [31:0] inst, \
394     input [4:0] rs1, \
395     input [31:0] rs1_val, \
396     input [4:0] rs2, \
397     input [31:0] rs2_val, \
398     input [4:0] rd, \
399     input [31:0] reg_i_data, \
400     input reg_wen, \
401     input is_imm, \
402     input is_auipc, \
403     input is_lui, \
404     input [31:0] imm, \
405     input [31:0] a_val, \
406     input [31:0] b_val, \
407     input [3:0] alu_ctrl, \
408     input [2:0] cmp_ctrl, \
```

```
409     input [31:0] alu_res, \
410     input cmp_res, \
411     input is_branch, \
412     input is_jal, \
413     input is_jalr, \
414     input do_branch, \
415     input [31:0] pc_branch, \
416     input mem_wen, \
417     input mem_ren, \
418     input [31:0] dmem_o_data, \
419     input [31:0] dmem_i_data, \
420     input [31:0] dmem_addr,
421
422 `define VGA_Debug_Signals_Arguments \
423     .pc(pc), \
424     .inst(inst), \
425     .rs1(rs1), \
426     .rs1_val(rs1_val), \
427     .rs2(rs2), \
428     .rs2_val(rs2_val), \
429     .rd(rd), \
430     .reg_i_data(reg_i_data), \
431     .reg_wen(reg_wen), \
432     .is_imm(is_imm), \
433     .is_auipc(is_auipc), \
434     .is_lui(is_lui), \
435     .imm(imm), \
436     .a_val(a_val), \
437     .b_val(b_val), \
438     .alu_ctrl(alu_ctrl), \
439     .cmp_ctrl(cmp_ctrl), \
440     .alu_res(alu_res), \
441     .cmp_res(cmp_res), \
442     .is_branch(is_branch), \
443     .is_jal(is_jal), \
444     .is_jalr(is_jalr), \
445     .do_branch(do_branch), \
446     .pc_branch(pc_branch), \
447     .mem_wen(mem_wen), \
448     .mem_ren(mem_ren), \
449     .dmem_o_data(dmem_o_data), \
450     .dmem_i_data(dmem_i_data), \
451     .dmem_addr(dmem_addr),
452
453 `define VGA_Debug_Signals_Outputs \
454     output [31:0] pc, \
455     output [31:0] inst, \
456     output [4:0] rs1, \
457     output [31:0] rs1_val, \
458     output [4:0] rs2, \
459     output [31:0] rs2_val, \
460     output [4:0] rd, \
461     output [31:0] reg_i_data, \
462     output reg_wen, \
463     output is_imm, \
464     output is_auipc, \
```

```
    output is_lui, \
    output [31:0] imm, \
    output [31:0] a_val, \
    output [31:0] b_val, \
    output [3:0] alu_ctrl, \
    output [2:0] cmp_ctrl, \
    output [31:0] alu_res, \
    output cmp_res, \
    output is_branch, \
    output is_jal, \
    output is_jalr, \
    output do_branch, \
    output [31:0] pc_branch, \
    output mem_wen, \
    output mem_ren, \
    output [31:0] dmem_o_data, \
    output [31:0] dmem_i_data, \
    output [31:0] dmem_addr,
```

ALU模块:

```
`timescale 1ns / 1ps
module ALU (
    input  [31:0] A,
    input  [31:0] B,
    input  [ 3:0] ALU_operation,
    output [31:0] res,
    output        zero
);
  wire signed [31:0] A_s = $signed(A);
  wire signed [31:0] B_s = $signed(B);
  wire [31:0] A_u = $unsigned(A);
  wire [31:0] B_u = $unsigned(B);
  wire [31:0] result0 = A_s + B_s;
  wire [31:0] result1 = A_s - B_s;
  wire [31:0] result2 = A << B[4:0];
  wire [31:0] result3 = (A_s < B_s) ? 32'b1 : 32'b0;
  wire [31:0] result4 = (A_u < B_u) ? 32'b1 : 32'b0;
  wire [31:0] result5 = A ^ B;
  wire [31:0] result6 = A >> B[4:0];
  wire [31:0] result7 = A_s >>> B_s[4:0];
  wire [31:0] result8 = A | B;
  wire [31:0] result9 = A & B;
  wire [31:0] result10 = ~|result1;
  wire [31:0] result11 = ~|result3;
  wire [31:0] result12 = ~|result4;
  wire [31:0] result13 = B;
  wire [31:0] result14 = A;
  assign res = (ALU_operation==4'b0000)?result0:
               (ALU_operation==4'b0001)?result1:
               (ALU_operation==4'b0010)?result2:
               (ALU_operation==4'b0011)?result3:
               (ALU_operation==4'b0100)?result4:
               (ALU_operation==4'b0101)?result5:
               (ALU_operation==4'b0110)?result6:
```

```
                (ALU_operation==4'b0111)?result7:
                (ALU_operation==4'b1000)?result8:
                (ALU_operation==4'b1001)?result9:
                (ALU_operation==4'b1010)?result10:
                (ALU_operation==4'b1011)?result11:
                (ALU_operation==4'b1100)?result12:
                (ALU_operation==4'b1101)?result13:
                (ALU_operation==4'b1110)?result14:
                32'b0;
  assign zero = ~(|res) ? 1'b1 : 1'b0;
endmodule
```

`Controler`模块

```
// RISC-V Controler
`include "Lab4.vh"
module Controler (
    input wire [4:0] OPcode,
    input wire MIO_ready,
    input wire [6:0] Fun7,
    input wire [2:0] Fun3,
    output reg CPU_MIO,
    output reg [3:0] wea,
    output reg [`IMM_SEL_WIDTH-1:0] ImmSell,
    output reg [`MEM2REG_WIDTH-1:0] MemtoReg,
    output reg [`ALU_OP_WIDTH-1:0] ALU_Control,
    output reg ALUSrc_B,
    output reg Jump,
    output reg Branch,
    output reg RegWrite,
    output reg sign,
    output reg [1:0] byte_n,
    output reg jump_choose,
    output reg illegal_inst,
    output reg csr_w,
    output reg [`CSR_CHANGE_MODE_WIDTH-1:0] csr_wsc_mode,
    output ecall,
    output mret
);
  assign ecall = {Fun7,Fun3,OPcode} == `ECALL;
  assign mret = {Fun7,Fun3,OPcode} == `MRET;
  always @(*) begin
    case (OPcode)
      `OPCODE_ALU: begin
        CPU_MIO <= 1'b0;
        wea <= `WEA_READ;
        ImmSell <= 3'b0;
        MemtoReg <= `MEM2REG_ALU;
        case (Fun3)
          `FUNC_ADD: ALU_Control <= Fun7[5] ? `ALU_OP_SUB : `ALU_OP_ADD;
          `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
          `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
          `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
          `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
          `FUNC_OR: ALU_Control <= `ALU_OP_OR;
```

```
          `FUNC_AND: ALU_Control <= `ALU_OP_AND;
          `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
          default: ALU_Control <= 4'b0;
        endcase
        ALUSrc_B <= 1'b0;
        Jump <= 1'b0;
        Branch <= 1'b0;
        RegWrite <= 1'b1;
        sign <= 1'b0;
        byte_n <= `WORD;
        jump_choose <= `JUMP_PC_IMM;
        illegal_inst <= 1'b0;
        csr_w <= 1'b0;
        csr_wsc_mode <= `CSR_WRITE;
      end
      `OPCODE_ALU_IMM: begin
        CPU_MIO <= 1'b0;
        wea <= `WEA_READ;
        ImmSell <= `IMM_SEL_I;
        MemtoReg <= `MEM2REG_ALU;
        case (Fun3)
          `FUNC_ADD: ALU_Control <= `ALU_OP_ADD;
          `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
          `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
          `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
          `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
          `FUNC_OR: ALU_Control <= `ALU_OP_OR;
          `FUNC_AND: ALU_Control <= `ALU_OP_AND;
          `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
          default: ALU_Control <= 4'b0;
        endcase
        ALUSrc_B <= 1'b1;
        Jump <= 1'b0;
        Branch <= 1'b0;
        RegWrite <= 1'b1;
        sign <= 1'b1;
        byte_n <= `WORD;
        jump_choose <= `JUMP_PC_IMM;
        illegal_inst <= 1'b0;
        csr_w <= 1'b0;
        csr_wsc_mode <= `CSR_WRITE;
      end
      `OPCODE_LOAD: begin
        CPU_MIO <= 1'b1;
        wea <= `WEA_READ;
        ImmSell <= `IMM_SEL_I;
        MemtoReg <= `MEM2REG_MEM;
        ALU_Control <= 4'b0;
        ALUSrc_B <= 1'b1;
        Jump <= 1'b0;
        Branch <= 1'b0;
        RegWrite <= 1'b1;
        sign <= ~(Fun3 == `FUNC_BYTE_UNSIGNED || Fun3 ==
`FUNC_HALF_UNSIGNED);
        case (Fun3)
          `FUNC_BYTE, `FUNC_BYTE_UNSIGNED: byte_n <= `BYTE;
```

```
97            `FUNC_HALF, `FUNC_HALF_UNSIGNED: byte_n <= `HALF;
98            `FUNC_WORD: byte_n <= `WORD;
99            default: byte_n <= `WORD;
100         endcase
101         jump_choose <= `JUMP_PC_IMM;
102         illegal_inst <= 1'b0;
103         csr_w <= 1'b0;
104         csr_wsc_mode <= `CSR_WRITE;
105       end
106       `OPCODE_STORE: begin
107         CPU_MIO <= 1'b1;
108         case (Fun3)
109           `FUNC_BYTE: wea <= `WEA_BYTE;
110           `FUNC_HALF: wea <= `WEA_HALF;
111           `FUNC_WORD: wea <= `WEA_WORD;
112           default: wea <= `WEA_READ;
113         endcase
114         ImmSell <= `IMM_SEL_S;
115         MemtoReg <= `MEM2REG_MEM;
116         ALU_Control <= 4'b0;
117         ALUSrc_B <= 1'b1;
118         Jump <= 1'b0;
119         Branch <= 1'b0;
120         RegWrite <= 1'b0;
121         sign <= 1'b1;
122         byte_n <= `WORD;
123         jump_choose <= `JUMP_PC_IMM;
124         illegal_inst <= 1'b0;
125         csr_w <= 1'b0;
126         csr_wsc_mode <= `CSR_WRITE;
127       end
128       `OPCODE_BRANCH: begin
129         CPU_MIO <= 1'b0;
130         wea <= `WEA_READ;
131         ImmSell <= `IMM_SEL_B;
132         MemtoReg <= `MEM2REG_ALU;
133         case (Fun3)
134           `FUNC_EQ:  ALU_Control <= `ALU_OP_SUB;
135           `FUNC_NE:  ALU_Control <= `ALU_OP_EQ;
136           `FUNC_LT:  ALU_Control <= `ALU_OP_SGE;
137           `FUNC_GE:  ALU_Control <= `ALU_OP_SLT;
138           `FUNC_LTU: ALU_Control <= `ALU_OP_SGEU;
139           `FUNC_GEU: ALU_Control <= `ALU_OP_SLTU;
140           default:   ALU_Control <= 4'b0;
141         endcase
142         ALUSrc_B <= 1'b0;
143         Jump <= 1'b0;
144         Branch <= 1'b1;
145         RegWrite <= 1'b0;
146         sign <= 1'b1;
147         byte_n <= `WORD;
148         jump_choose <= `JUMP_PC_IMM;
149         illegal_inst <= 1'b0;
150         csr_w <= 1'b0;
151         csr_wsc_mode <= `CSR_WRITE;
152       end
```

```
        `OPCODE_JAL: begin
          CPU_MIO <= 1'b0;
          wea <= `WEA_READ;
          ImmSell <= `IMM_SEL_J;
          jump_choose <= `JUMP_PC_IMM;
          MemtoReg <= `MEM2REG_PC_PLUS;
          ALU_Control <= `ALU_OP_ADD;
          ALUSrc_B <= 1'b1;
          Jump <= 1'b1;
          Branch <= 1'b0;
          RegWrite <= 1'b1;
          sign <= 1'b1;
          byte_n <= `WORD;
          illegal_inst <= 1'b0;
          csr_w <= 1'b0;
          csr_wsc_mode <= `CSR_WRITE;
        end
        `OPCODE_JALR: begin
          CPU_MIO <= 1'b0;
          wea <= `WEA_READ;
          ImmSell <= `IMM_SEL_I;
          MemtoReg <= `MEM2REG_PC_PLUS;
          ALU_Control <= `ALU_OP_ADD;
          ALUSrc_B <= 1'b1;
          Jump <= 1'b1;
          Branch <= 1'b0;
          RegWrite <= 1'b1;
          sign <= 1'b1;
          byte_n <= `WORD;
          jump_choose <= `JUMP_ALU;
          illegal_inst <= 1'b0;
          csr_w <= 1'b0;
          csr_wsc_mode <= `CSR_WRITE;
        end
        `OPCODE_LUI: begin
          CPU_MIO <= 1'b0;
          wea <= `WEA_READ;
          ImmSell <= `IMM_SEL_U;
          MemtoReg <= `MEM2REG_ALU;
          ALU_Control <= `ALU_OP_R2;
          ALUSrc_B <= 1'b1;
          Jump <= 1'b0;
          Branch <= 1'b0;
          RegWrite <= 1'b1;
          sign <= 1'b1;
          byte_n <= `WORD;
          jump_choose <= `JUMP_PC_IMM;
          illegal_inst <= 1'b0;
          csr_w <= 1'b0;
          csr_wsc_mode <= `CSR_WRITE;
        end
        `OPCODE_AUIPC: begin
          CPU_MIO <= 1'b0;
          wea <= `WEA_READ;
          ImmSell <= `IMM_SEL_U;
          MemtoReg <= `MEM2REG_IMM_PC;
```

```
209           ALU_Control <= `ALU_OP_ADD;
210           ALUSrc_B <= 1'b1;
211           Jump <= 1'b0;
212           Branch <= 1'b0;
213           RegWrite <= 1'b1;
214           sign <= 1'b1;
215           byte_n <= `WORD;
216           jump_choose <= `JUMP_PC_IMM;
217           illegal_inst <= 1'b0;
218           csr_w <= 1'b0;
219           csr_wsc_mode <= `CSR_WRITE;
220         end
221         `OPCODE_CSR: begin
222           CPU_MIO <= 1'b0;
223           wea <= `WEA_READ;
224           ImmSell <= `IMM_SEL_CSR;
225           MemtoReg <= `MEM2REG_CSR;
226           ALUSrc_B <= 1'b1;
227           RegWrite <= 1'b1;
228           case(Fun3)
229             `FUNC_CSRRW:
230             begin
231               ALU_Control <= `ALU_OP_R1;
232               csr_wsc_mode <= `CSR_WRITE;
233             end
234             `FUNC_CSRRS:
235             begin
236               ALU_Control <= `ALU_OP_R1;
237               csr_wsc_mode <= `CSR_OR;
238             end
239             `FUNC_CSRRC:
240             begin
241               ALU_Control <= `ALU_OP_R1;
242               csr_wsc_mode <= `CSR_CLEAR;
243             end
244             `FUNC_CSRRWI:
245             begin
246               ALU_Control <= `ALU_OP_R2;
247               csr_wsc_mode <= `CSR_WRITE;
248             end
249             `FUNC_CSRRSI:
250             begin
251               ALU_Control <= `ALU_OP_R2;
252               csr_wsc_mode <= `CSR_OR;
253             end
254             `FUNC_CSRRCI:
255             begin
256               ALU_Control <= `ALU_OP_R2;
257               csr_wsc_mode <= `CSR_CLEAR;
258             end
259             default:
260             begin
261               ALU_Control <= `ALU_OP_R1;
262               csr_wsc_mode <= `CSR_BYPASS;
263             end
264           endcase
```

```
        Jump <= 1'b0;
        Branch <= 1'b0;
        sign <= 1'b0;
        byte_n <= `WORD;
        jump_choose <= `JUMP_PC_IMM;
        illegal_inst <= 1'b0;
        csr_w <= 1'b1;
      end
      default:
      begin
        CPU_MIO <= 1'b0;
        wea <= `WEA_READ;
        ImmSell <= 3'b0;
        MemtoReg <= `MEM2REG_CSR;
        ALU_Control <= 4'b0;
        ALUSrc_B <= 1'b0;
        Jump <= 1'b0;
        Branch <= 1'b0;
        RegWrite <= 1'b0;
        sign <= 1'b0;
        byte_n <= `WORD;
        jump_choose <= `JUMP_PC_IMM;
        illegal_inst <= 1;
        csr_w <= 1'b1;
        csr_wsc_mode <= `CSR_BYPASS;
      end

    endcase
  end
endmodule
```

`DataPath`模块

```

`include "Lab4.vh"
module DataPath (
    `RegFile_Regs_Outputs
    input wire clk,
    input wire rst,
    input wire [31:0] inst_field,
    input wire [31:0] Data_in,
    input wire [`IMM_SEL_WIDTH-1:0] ImmSell,
    input wire ALUSrc_B,
    input wire [`MEM2REG_WIDTH-1:0] MemtoReg,
    input wire Jump,
    input wire Branch,
    input wire RegWrite,
    input wire [`ALU_OP_WIDTH-1:0] ALU_Control,
    input wire sign,
    input wire [1:0] byte_n,
    input wire jump_choose,
    input wire illegal_inst,
    input wire csr_w,
    input wire [`CSR_CHANGE_MODE_WIDTH-1:0] csr_wsc_mode,
    input wire ecall,
```

```
    input wire mret,
    input wire INT,
    output wire [31:0] PC_out,
    output wire [31:0] Data_out,
    output wire [31:0] Addr_out,
    output wire trap_normal_change,
    `CSR_OUTPUTS
);
  reg  [31:0] PC = 32'h4;
  wire [31:0] imm;
  ImmGen U1 (
      .ImmSell(ImmSell),
      .inst_field(inst_field),
      .sign(1'b1),
      .Imm(imm)
  );
  wire [31:0] Rs1_data, Rs2_data;
  wire [4:0] Rs1_addr, Rs2_addr, W_addr;
  wire [31:0] W_data;

  wire [31:0] ALU_out;
  wire [11:0] CSR_raddr = inst_field[31:20];
  wire [31:0] CSR_wdata = ALU_out;
  wire [31:0] CSR_waddr = inst_field[31:20];
  wire [31:0] PC_change;
  wire [31:0] CSR_rdata;

  RV_INT U5 (
      .clk(clk),
      .rst(rst),
      .INT(INT),
      .ecall(ecall),
      .mret(mret),
      .illegal_inst(illegal_inst),
      .PC(PC),
      .raddr(CSR_raddr),
      .waddr(CSR_waddr),
      .wdata(CSR_wdata),
      .csr_w(csr_w),
      .csr_wsc_mode(csr_wsc_mode),
      .trap_normal_change(trap_normal_change),
      .PC_change(PC_change),
      .rdata(CSR_rdata),
      .inst_field(inst_field),
      `CSR_ARGUMENTS
  );
  assign Rs1_addr = inst_field[19:15];
  assign Rs2_addr = inst_field[24:20];
  assign W_addr   = inst_field[11:7];
  Regs U2 (
      `RegFile_Regs_Arguments
      .clk(clk),
      .rst(rst),
      .Rs1_addr(Rs1_addr),
      .Rs2_addr(Rs2_addr),
      .Wt_addr(W_addr),
```

```
        .Wt_data(W_data),
        .RegWrite(RegWrite & !trap_normal_change),
        .Rs1_data(Rs1_data),
        .Rs2_data(Rs2_data)
    );
    wire [31:0] adder_2;
    wire zero;
    ALU U3 (
        .A(Rs1_data),
        .B(adder_2),
        .ALU_operation(ALU_Control),
        .res(ALU_out),
        .zero(zero)
    );
    assign Addr_out = ALU_out;
    assign Data_out = Rs2_data;
    wire [31:0] mem_out;
    extend U4 (
        .byte_n(byte_n),
        .in(Data_in >> ((Addr_out % 4) << 3)),
        .sign(sign),
        .mem_data(mem_out)
    );

    assign adder_2 = ALUSrc_B ? imm : Rs2_data;

    wire [31:0] PC_add_4 = PC + 4;
    wire [31:0] PC_imm = imm + PC;
    wire Branch_final = Branch & zero;
    wire [31:0] PC_branch = Branch_final ? PC_imm : PC_add_4;
    wire [31:0] PC_jump = Jump ? ((jump_choose == `JUMP_ALU) ? ALU_out :
PC_imm) : PC_branch;
    assign PC_out = trap_normal_change ? PC_change : PC_jump;

    always @(posedge clk or posedge rst) begin
      if (rst) begin
        PC <= 32'hFFFFFFFC;
      end else begin
        PC <= PC_out;
      end
    end

    assign W_data=MemtoReg==`MEM2REG_MEM?mem_out:
                MemtoReg==`MEM2REG_ALU?ALU_out:
                MemtoReg==`MEM2REG_PC_PLUS?PC_add_4:
                MemtoReg==`MEM2REG_IMM_PC?PC_imm:
                MemtoReg==`MEM2REG_CSR?CSR_rdata:32'b0;

endmodule
```

下板验证

注意这里的 PC 为正在执行指令的下一条指令

`ecall` 指令的退出与进入:

```
RV32I Single Cycle CPU

pc: 00000044    inst: 30002FF3

x0: 00000000    ra: 00000003    sp: 00000005    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000008       mcause: 00000000        mepc: 00000000  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 0000005C    inst: 00BD0D13

x0: 00000000    ra: 00000003    sp: 00000005    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 80000003
t5: 00000018    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000080       mcause: 80000003        mepc: 00000018  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 00000020    inst: 00308133

x0: 00000000    ra: 00000003    sp: 00000005    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 80000003
t5: 0000001C    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000088       mcause: 00000000        mepc: 0000001C  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

硬件中断的进入与退出:

```
RV32I Single Cycle CPU

pc: 00000044    inst: 30002FF3

x0: 00000000    ra: 0000000D    sp: 00000015    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 80000003
t5: 0000001C    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000088       mcause: 00000000        mepc: 0000001C  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 0000005C    inst: 00BD0D13

x0: 00000000    ra: 0000000D    sp: 00000015    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 800000B
t5: 00000024    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000080       mcause: 8000000B        mepc: 00000024  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 00000028    inst: 003100B3

x0: 00000000    ra: 0000000D    sp: 00000015    gp: 00000008    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 8000000B
t5: 00000024    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000088       mcause: 00000000        mepc: 00000024  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

非法指令中断的进入与退出,注意`mtval`正确保存了非法指令的值:

```
RV32I Single Cycle CPU

pc: 00000044    inst: 30002FF3

x0: 00000000    ra: 00000037    sp: 00000059    gp: 00000090    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: 00000000    t4: 8000000B
t5: 00000024    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000088       mcause: 00000000        mepc: 00000024  mtval: 00000000
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 0000005C    inst: 00BD0D13

x0: 00000000    ra: 00000037    sp: 00000059    gp: 00000090    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000044    t3: FFFFFFFF    t4: 00000002
t5: 00000034    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000080       mcause: 00000002        mepc: 00000034  mtval: FFFFFFFF
mtvec:   00000044       mie: 00000000   mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 0000003C    inst: 00308133

x0: 00000000    ra: 00000037    sp: 00000059    gp: 00000090    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:FFFFFFFF    s11:00000044    t3: FFFFFFFF    t4: 00000002
t5: 00000038    t6: 00000080

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd:  00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0       is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0     cmp_ctrl: 0
alu_res: 00000000       cmp_res: 0

is_branch: 0    is_jal: 0       is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0     csr_r_data:00000000
mstatus: 00000088       mcause: 00000000        mepc: 00000038  mtval: FFFFFFC7
mtvec:   00000044       mie: 00000000   mip: 00000000
```

思考题

在涉及到一个大立即数的读入时，我们经常能想到使用 lui & addi 来实现，比如下面这段代码就将 0x22223333 赋给了 t0:

lui t0, 0x22223
addi t0, t0, 0x333
你是否能通过以下代码得到 0xDEADBEEF？如果你觉得不能的话，先解释为什么不能，再修改代码中的一个字符，使得以下代码有效地得到 0xDEADBEEF。（如果你觉得可以的话，请重新学习 RISC-V ISA）

lui t1, 0xDEADB
addi t1, t1, 0xEEF
btw, 如果你把上边代码放到 Venus 上，会发现它给了你一个报错，不要理会它，它理解错了。
回答:
不行,因为立即数是有符号扩展的,`EEF` 会扩展为 `0xFFFFFEEF`,加上 `0xDEADB000` 得到 `0xDEADAEEF`,当立即数实际解释为负数时,会扩展为负数造成前五位产生一个退位,故无法得到 `0xDEADBEEF`.修改 `0xDEADB` 为 `0xDEADC` 即可得到 `0xDEADBEEF`.