中国科学院大学

University of Chinese Academy of Sciences

计算机组成原理实验报告

MIPS 单周期 CPU 设计

姓名 徐昕妍

学号 2023K8009970008

班级 2311 班

2025 年 5 月 25 日

目录

1 实验目的

本实验旨在设计一个基于 MIPS 指令集架构的单周期 CPU，具体包括 CPU 的指令集设计、控制信号设计等。通过本实验，可以深入理解计算机组成原理中的 CPU 设计原理，加深对计算机体系结构的理解，加深对 CPU 各模块单元和数据通路的理解。

2 实验过程

核心代码主要分为四个部分，ALU 算术单元模块、regfile 寄存器堆模块、shifter 位移模块以及 top 顶层模块。以下是各核心模块的代码及解释。

2.1 ALU 算术单元模块

核心代码如下：

```
module alu(
    input [`DATA_WIDTH - 1:0] A,
    input [`DATA_WIDTH - 1:0] B,
    input [2:0] ALUop,
    output Overflow,
    output CarryOut,
    output Zero,
    output [`DATA_WIDTH - 1:0] Result
);
    // 控制信号
    wire subtract = (ALUop == 3'b110 || ALUop == 3'b111 || ALUop ==
       3'b011);

    // 统一加法器逻辑
    wire [31:0] B_sel = subtract ? ~B : B;
    wire [32:0] sum_ext = {1'b0, A} + {1'b0, B_sel} + subtract;

    // 符号位处理
    wire sign_A = A[31];
    wire sign_B = B[31];

    // 进位标志优化
    wire B_is_zero = ~(|B);
    assign CarryOut = (ALUop == 3'b010) ? sum_ext[32] :
```

```
                    (ALUop == 3'b110 || ALUop== 3'b011) ? (B_is_zero
                        ? 1'b0 : ~sum_ext[32]) : 1'b0;

    // 溢出判断改进
    wire op_add = (ALUop == 3'b010);
    wire op_sub = (ALUop == 3'b110);
    wire same_sign_add = (sign_A == (subtract ? ~sign_B : sign_B));
    wire overflow = (same_sign_add && (sum_ext[31] != sign_A));
     // 当操作数符号相同但结果符号不同时触发
    assign Overflow = (op_add | op_sub) ? overflow : 1'b0;
    //仅在加减法操作有效

    // 有符号比较
    wire slt_sign = (sign_A ^ sign_B) ? sign_A :
                    (sum_ext[31] ^ (overflow & op_sub));
                    // 符号不同时直接比较原始符号，符号相同时结合结果
                       符号和溢出修正
    wire [31:0] result_slt = {31'b0, slt_sign};

     // 无符号比较 (SLTU)
    wire [31:0] result_sltu = {31'b0, CarryOut};

    // 结果选择
    assign Result = (ALUop == 3'b000) ? (A & B) :
                    (ALUop == 3'b001) ? (A | B) :
                    (ALUop == 3'b100) ? (A ^ B) :        // XOR
                    (ALUop == 3'b101) ? ~(A | B) :      // NOR
                    (op_add | op_sub) ? sum_ext[31:0] :
                    (ALUop == 3'b011) ? result_sltu :   // SLTU
                    (ALUop == 3'b111) ? result_slt : 32'b0;

    // 零标志
    assign Zero = ~(|Result);

endmodule
```

该 ALU 模块主要支持 8 种算术逻辑运算：AND、OR、ADD、SLTU、XOR、NOR、SUB、SLT。

2.1.1 算术运算（ADD/SUB）

将减法转换为补码加法 (取反加一)，采用统一加法架构：

```
sum_ext = A + (subtract ? ~B : B) + subtract
```

标志位生成：

```
// 进位标志（减法时取反）
wire B_is_zero = ~(|B);
assign CarryOut = (ALUop == 3'b010) ? sum_ext[32] :
            (ALUop == 3'b110 || ALUop== 3'b011) ? (B_is_zero ? 1'b0 :
                ~sum_ext[32]) : 1'b0

// 溢出标志（符号位不一致时触发）
overflow = (same_sign_add && (sum_ext[31] != sign_A));
```

对于进位标志：

- 加法时进位直接用最高位。
- 减法和无符号比较时，借位信号用最高位的反码表示。
- 其他指令不关心进位，置 0。

2.1.2 比较运算（SLT/SLTU）

有符号位比较：

- 符号不同时：直接比较操作数符号（负数更小）
- 符号相同时：结合减法结果和溢出标志判断

```
wire slt_sign = (sign_A ^ sign_B) ? sign_A : (sum_ext[31] ^ (overflow
    & op_sub));
```

无符号位比较：

- 直接使用 CarryOut 来判断无符号大小。

```
result_sltu = {31'b0, CarryOut};
```

2.1.3 结果选择逻辑

ALUop (3-bit)	操作说明
000	按位与 (AND)
001	按位或 (OR)
100	异或 (XOR)
101	按位 NOR (NOR)
010	加法 (ADD)
110	减法 (SUB)
011	无符号比较 (SLTU)
111	有符号比较 (SLT)

表 1: ALU 操作码对应的功能说明

2.2 regfile 寄存器堆模块

核心代码如下:

```
module reg_file(
  input clk,
  input rst,
  input [`ADDR_WIDTH - 1:0] waddr,
  input [`ADDR_WIDTH - 1:0] raddr1,
  input [`ADDR_WIDTH - 1:0] raddr2,
  input wen,
  input [`DATA_WIDTH - 1:0] wdata,
  output [`DATA_WIDTH - 1:0] rdata1,
  output [`DATA_WIDTH - 1:0] rdata2
);

  // TODO: Please add your logic code here
  reg [`DATA_WIDTH-1:0] regfile[0:(1<<`ADDR_WIDTH)-1];
  assign rdata1=(raddr1==0)?32'b0:regfile[raddr1];
  assign rdata2=(raddr2==0)?32'b0:regfile[raddr2];

  always @(posedge clk) begin
    if (wen==1&&(waddr!=0)) begin
      regfile[waddr]<=wdata;
    end
  end
```

```

endmodule
```

输入输出信号如下表所示：

端口名	功能说明
clk	时钟信号，写操作在上升沿同步进行
rst	复位信号（本设计未使用）
waddr	写地址，指明哪个寄存器要写数据
raddr1	第一个读地址，支持读一个寄存器
raddr2	第二个读地址，支持读另一个寄存器
wen	写使能，高电平时允许写操作
wdata	写入的数据
rdata1	第一个读端口的输出数据
rdata2	第二个读端口的输出数据

表 2: 寄存器文件模块端口信号说明

2.2.1 寄存器数组定义

```
reg [`DATA_WIDTH-1:0] regfile[0:(1<<`ADDR_WIDTH)-1];
```

定义一个寄存器数组，数组大小是 2^{ADDR_WIDTH}，每个元素宽度是 DATA_WIDTH

2.2.2 读操作

```
assign rdata1 = (raddr1 == 0) ? 32'b0 : regfile[raddr1];
assign rdata2 = (raddr2 == 0) ? 32'b0 : regfile[raddr2];
```

- 读操作是组合逻辑，读地址为 0 时，强制输出全 0
- 读地址非 0 时，直接输出对应寄存器的内容。

2.2.3 写操作

```
always @(posedge clk) begin
    if (wen == 1 && (waddr != 0)) begin
        regfile[waddr] <= wdata;
    end
end
```

- 写操作在时钟上升沿进行，保证同步。
- 只有当写使能为 1 且写地址不为 0 时，写入数据。
- 保持寄存器 0 固定为 0，不允许写入。

2.3 shifter 位移模块

核心代码如下：

```
module shifter (
  input  [`DATA_WIDTH - 1:0] A,
  input  [              4:0] B,
  input  [              1:0] Shiftop,
  output [`DATA_WIDTH - 1:0] Result
);
  // TODO: Please add your logic code here

  // 算术右移符号位处理
        wire [`DATA_WIDTH - 1:0] arith_shift_mask =
        (B == 0) ? 0 :  // B=0 时无需填充
        ({32{A[31]}} & ~((32'b1 << (32 - B)) - 1)); // 生成高 B 位为
           符号位的掩码


// 使用条件运算符实现多路选择器
        assign Result = (Shiftop == 2'b00) ? (A << B) :       // 左移
               (Shiftop == 2'b10) ? (A >> B) :        // 逻辑右移
               (Shiftop == 2'b11) ? ((A >> B) | arith_shift_mask) :
                  // 算术右移
               32'b0;                                 // 默认值


endmodule
```

位移类型如下表所示：

Shiftop (2-bit)	移位操作说明
00	逻辑左移
10	逻辑右移
11	算术右移
其他	输出全 0（默认）

表 3: 移位操作码与功能对应表

2.3.1 算术右移的特殊处理

算术右移要求用符号位填充左边空出的高位，即保持符号不变:
arith_shift_mask 是用来生成高位符号填充的掩码:

```
    wire [`DATA_WIDTH - 1:0] arith_shift_mask = (B == 0) ? 0 :
  ({32{A[31]}} & ~((32'b1 << (32 - B)) - 1));
```

- 如果移位量 B == 0，掩码全为 0，不用填充。
- 32A[31] 是将符号位（最高位）扩展成 32 位全 1 或全 0（取决于符号）。
- ((32'b1 « (32 - B)) - 1) 生成一个低位全 1、高位全 0 的掩码。
- 取反后变成高位全 1、低位全 0 的掩码。
- 两者与运算后，就得到要填充的符号位掩码，高位填充符号位。

最终算术右移结果为

```
(A >> B) | arith_shift_mask
```

2.4 top 顶层模块

顶层模块主要分为 8 个部分：指令解码、指令类型判断与控制信号生成、寄存器堆实例化、ALU 操作、写使能控制、跳转和分支指令、内存读写操作以及程序计数器更新。

2.4.1 指令解码

```
wire [5:0] opcode = Instruction[31:26];  // 操作码（6位）
wire [4:0] rs = Instruction[25:21];      // 源寄存器1地址
wire [4:0] rt = Instruction[20:16];      // 源寄存器2地址
wire [4:0] rd = Instruction[15:11];      // 目标寄存器地址
wire [5:0] func = Instruction[5:0];      // 功能码
```

```
wire [15:0] imm = Instruction[15:0];      // 立即数 (I-type)
wire [25:0] tar = Instruction[25:0];      // 跳转目标地址 (J-type)
```

该模块主要实现从 32 位指令中提取操作码（opcode）、源寄存器（rs 和 rt）、目标寄存器（rd）以及其他字段。

字段名	功能说明
‘opcode‘	操作码，决定指令类型，如 ‘R-type‘、‘I-type‘、‘J-type‘ 等
‘rs‘	源寄存器 1 地址，用于从寄存器文件中读取数据
‘rt‘	源寄存器 2 地址，用于从寄存器文件中读取数据
‘rd‘	目标寄存器地址，指示运算结果存储的目标寄存器
‘func‘	功能码，特定于 ‘R-type‘ 指令，用于区分不同的算术逻辑操作
‘imm‘	立即数，‘I-type‘ 指令中的常量，参与运算
‘tar‘	目标地址，‘J-type‘ 指令中的跳转地址

表 4: 指令字段功能说明

2.4.2 指令类型判断与控制信号生成

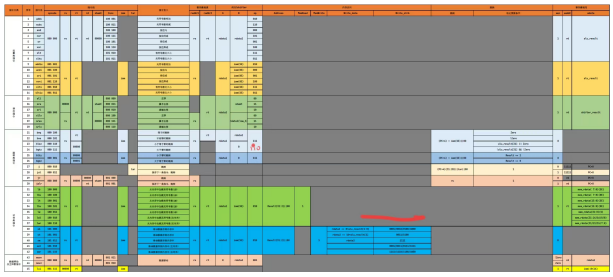

图 1: 指令译码表

根据指令的 opcode 和 func 字段生成不同的控制信号，用于决定执行哪种操作。指令类型包括：

```
    assign R_TYPE = (opcode == 6'b000000)? 1:0;
    assign REGIMM = (opcode == 6'b000001) ?1:0;
    assign J_TYPE = (opcode == 6'b000010 || opcode == 6'b000011)?1:0;
    assign JR_TYPE = (R_TYPE && (func == 6'b001000 || func == 6'
       b001001))?1:0;
    assign Branch = (opcode[5:2] == 4'b0001) ?1:0;     //I-Type 分支指
       令
    assign OP_I = (opcode[5:3] == 3'b001)   ?1:0;    //I-Type 计算指令
```

```
    assign LD    = (opcode[5:3] == 3'b100) ?1:0;      //I-Type 内存读指令
    assign ST    = (opcode[5:3] == 3'b101) ?1:0;      //I-Type 内存写指令
```

信号名	说明
R_TYPE	指令是寄存器类型指令（R-type），即操作码为 6'b000000。
REGIMM	指令是特殊寄存器立即数分支指令，操作码为 6'b000001。
J_TYPE	指令是跳转类型指令，操作码为 6'b000010（j）或 6'b000011（jal）。
JR_TYPE	指令是寄存器跳转指令（jr 或 jalr），为 R_TYPE 且功能码为 6'b001000 或 6'b001001。
Branch	指令是分支类 I 型指令，操作码的高四位为 4'b0001。
OP_I	指令是 I 型算术逻辑指令，操作码的高三位为 3'b001。
LD	指令是 I 型加载指令，操作码的高三位为 3'b100。
ST	指令是 I 型存储指令，操作码的高三位为 3'b101。

表 5: CPU 指令译码控制信号说明

控制信号由 MemRead、MemWriteh 和 RF_waddr 生成。

```
//MemRead
    assign MemRead =opcode[5:3]== 3'b100 ?1 :0 ;
//Memwrite
    assign MemWrite =opcode[5:3]== 3'b101 ?1 : 0;

    assign RF_waddr = R_TYPE  ? rd  :
      (opcode == 6'b000011)? 5'b11111 :  // jal指令目标为31号寄存器
         rt;
```

1.MemRead 信号

控制是否从内存读取数据。

在 MIPS 指令集中，操作码前三位为 100 的通常是内存加载类指令，当检测到这样的 opcode 时，将 MemRead 信号置为 1，表示启动内存读取操作。

2.MemWrite 信号

控制是否向内存写入数据。

操作码前三位为 101 的通常是内存存储类指令，当检测到这样的 opcode 时，将 MemWrite 信号置为 1，表示启动内存写操作。

3.RF_waddr 寄存器写地址选择

确定寄存器文件中写入数据的目标寄存器地址。

- 如果是 R 型指令 (R_TYPE == 1)，写入目标寄存器是 rd;
- 如果是 jal 指令 (opcode == 000011)，写入寄存器 31 (5'b11111)，也就是 $ra 寄存器，用于存储返回地址;
- 如果是 I 型指令，写入目标寄存器是 rt。

2.4.3 寄存器堆实例化

```
//reg_file模块
  reg_file instance_reg_file(
    .clk(clk),
    .waddr(RF_waddr),
    .raddr1(rs),
    .raddr2(rt),
    .wen(RF_wen),
    .wdata(RF_wdata),
    .rdata1(rsdata),
    .rdata2(rtdata)
  );
```

2.4.4 ALU 操作

```
//ALUSrc[2]为0时，第一个操作数取0，其余取rsdata;ALUSrc[1:0]为00时，第二个操作数B取0，为10时取imm（SE），为01时取imm（0E），为11时取rtdata
  assign ALUSrc [2]= (R_TYPE && (func[5:0]==6'b001011 ||func[5:0]==6'b001010)) ? 0: 1;

  assign ALUSrc [1:0]= ((R_TYPE &&func[5]==1)||opcode==6'b000100||opcode==6'b000101||(R_TYPE && (func[5:0]==6'b001011 || func[5:0]==6'b001010)))   ? 2'b11:
                       (opcode==6'b000110)||(opcode==6'b000111)||(opcode==6'b000001)   ? 2'b00:
            (opcode==6'b001100)||(opcode==6'b001101)||(opcode==6'b001110)   ? 2'b01:2'b10;

  wire [31:0] A;
```

```
  wire [31:0] B;
  wire [2:0] ALUop;
  wire [31:0]alu_result;
  assign A= ALUSrc[2]==1'b0 ? 32'b0: rsdata ;


  assign B= ALUSrc[1:0]==2'b00 ? 32'b0:
            ALUSrc[1:0]==2'b01 ?imm_0E:
            ALUSrc[1:0]==2'b10 ?imm_SE:
                    rtdata;

        ALUop instance_ALUop(
    .opcode(opcode),
    .func(func),
        .ALUop(ALUop)
  );

  alu instance_alu(
    .A(A),
    .B(B),
    .ALUop(ALUop),
    .Zero(Zero),
    .Result(alu_result)
  );
```

ALUSrc 分别控制 ALU 的两个操作数 A 和 B 如何选择：
ALUSrc[2]：控制 ALU 第一个操作数 A 是取 0 还是 rsdata（寄存器 rs 的数据）。
ALUSrc[1:0]：控制 ALU 第二个操作数 B 的来源：

- 00：第二操作数为 0。
- 01：第二操作数为立即数的零扩展版本（imm_0E）。
- 10：第二操作数为立即数的符号扩展版本（imm_SE）。
- 11：第二操作数为寄存器 rt 的数据（rtdata）。

ALUSrc 具体赋值如下：

```
assign ALUSrc[2] = (R_TYPE && (func == 6'b001011 || func == 6'b001010
   )) ? 0 : 1;
```

如果是 R 型指令，且功能码是：

001011 （sltiu），或者 001010（slti）那么 ALUSrc[2] = 0，第一个操作数取 0。（仅需要对 rtdata 中的立即数进行运算，A 置为 0）

否则 ALUSrc[2] = 1，第一个操作数取 rsdata。

```
assign ALUSrc[1:0] = ((R_TYPE && func[5] == 1) || opcode == 6'b000100
    || opcode == 6'b000101 ||
(R_TYPE && (func == 6'b001011 || func == 6'b001010))) ? 2'b11 :
                     (opcode == 6'b000110) || (opcode == 6'b000111) ||
                         (opcode == 6'b000001) ? 2'b00 :
                      (opcode == 6'b001100) || (opcode == 6'b001101)
                         || (opcode == 6'b001110) ? 2'b01 :
                      2'b10;
```

- 如果是 R 型指令且 func 的最高位(func[5])是 1(对应运算指令 addu,subu,and,nor,or,xor,slt,sltu),

  或者是分支指令 beq (000100)、bne (000101),

  或者是 sltiu（001011）或 slti（001010），则 ALUSrc[1:0] = 2'b11，第二操作数取 rtdata。

- 如果是分支指令 blez (000110)、bgtz (000111)，或 REGIMM (000001) 指令，则 ALUSrc[1:0] = 2'b00，第二操作数取 0

- 如果是逻辑立即数指令 andi (001100)、ori (001101)、xori (001110)，则 ALUSrc[1:0] = 2'b01，第二操作数取零扩展立即数 imm_0E

2.4.5 写使能（RF_wen）控制

```
assign RF_wen = Branch ||
                REGIMM ||
                ST ||
                (opcode == 6'b000010) ||
                (opcode == 6'b000000 && func == 6'b001000) ||
                (opcode == 6'b000000 && func == 6'b001011 && Zero) ||
                (opcode == 6'b000000 && func == 6'b001010 && ~Zero) ?
                    0 : 1;
```

当指令属于 Branch、REGIMM、ST 或者指令是 j (000010)、jr (opcode == 6'b000000 且 func == 6'b001000)、movn(opcode == 6'b000000 且 func == 6'b001011 且 Zero)、movz(opcode == 6'b000000 且 func == 6'b001010 且 Zero)，禁用写使能。

以上条件指令都不需要写寄存器结果，禁止写使能。

其他指令默认允许写寄存器，写使能为 1。

2.4.6 跳转和分支指令

```
    //跳转指令
  wire is_jorb;  //记录是否跳转
  wire [31:0] next_pc;// 跳转地址
  assign is_jorb =
    // 跳转指令（J/JR类型）
                (J_TYPE || JR_TYPE) ? 1'b1 :
    // 分支指令（Branch类型）
                (Branch) ? (
                        (opcode[2:1] == 2'b10) ?
                        (opcode[0] ^ Zero) :
                        (opcode[0] ^ (alu_result[31] || Zero))
                ) :
    // 寄存器条件跳转（REGIMM类型）
    (REGIMM && rt[4:0]==5'b00000 && alu_result==1) ? 1'b1 :
    (REGIMM && rt[4:0]==5'b00001 && alu_result==0) ? 1'b1 :
    // 默认不跳转
                                                    1'b0;
```

- J_TYPE/JR_TYPE（跳转指令）如 j, jal, jr, jalr 等，固定跳转，is_jorb=1
- Branch（条件分支）

  分支指令根据条件跳转：

  beq、bne（opcode[2:1] == 2'b10），与 Zero 标志结合判断是否跳转；

  blez（小于等于 0 跳转，000110 ）、bgtz（大于 0 跳转，000111 ）跳转条件：opcode[0] 异或 (alu_result[31] || Zero)
- REGIMM 类型：opcode == 6'b000001:

  bltz（小于 0 跳转，rt == 5'b00000），当 $rs < 0$,alu_result == 1 , 跳转；

  bgez（大于等于 0 跳转，rt == 5'b00001），当 $rs \geq 0$, alu_result == 0 , 跳转。

```
assign pc_add4 = PC + 4;  // 顺序执行地址（PC + 4）

assign next_pc =
    (Branch || REGIMM) ? (pc_add4 + imm_SE00) :    // 分支和REGIMM跳
       转，跳转到PC+4加偏移
    (J_TYPE) ? {pc_add4[31:28], tar, 2'b00} :      // J型跳转，拼接跳
       转地址（高4位取PC+4高4位）
```

```
    rsdata;                                          // JR类型跳转，目标地址取寄存器rsdata
```

计算下一条 PC 地址。

- 分支/REGIMM 跳转

  计算跳转目标地址为当前 PC+4 加上立即数偏移（符号扩展且左移两位）。

- J_TYPE 跳转

  拼接跳转目标地址（使用跳转指令中的目标字段 tar，高 4 位取当前 PC+4 高 4 位，低 2 位补 0）。

- JR_TYPE 跳转

  目标地址取寄存器 rsdata，用于寄存器跳转指令。

2.4.7 内存读写操作

```
assign lwl_data =
                (alu_result[1:0] == 2'b00) ? {Read_data[7:0],
                   rtdata[23:0]} :
                (alu_result[1:0] == 2'b01) ? {Read_data[15:0],
                   rtdata[15:0]} :
                (alu_result[1:0] == 2'b10) ? {Read_data[23:0],
                   rtdata[7:0]}  :
                                Read_data;


assign lwr_data =
    (alu_result[1:0] == 2'b00) ? Read_data :
    (alu_result[1:0] == 2'b01) ? {rtdata[31:24], Read_data[31:8]} :
    (alu_result[1:0] == 2'b10) ? {rtdata[31:16], Read_data[31:16]} :
                                 {rtdata[31:8],  Read_data[31:24]};


assign lb_mem_data =
    (alu_result[1:0] == 2'b00) ? {{24{Read_data[7]}},  Read_data
       [7:0]}    :
    (alu_result[1:0] == 2'b01) ? {{24{Read_data[15]}}, Read_data
       [15:8]} :
    (alu_result[1:0] == 2'b10) ? {{24{Read_data[23]}}, Read_data
       [23:16]} :
```

```
                                    {{24{Read_data[31]}}, Read_data
                                       [31:24]};

assign lbu_mem_data =
    (alu_result[1:0] == 2'b00) ? {24'h0, Read_data[7:0]}    :
    (alu_result[1:0] == 2'b01) ? {24'h0, Read_data[15:8]}   :
    (alu_result[1:0] == 2'b10) ? {24'h0, Read_data[23:16]}   :
                                    {24'h0, Read_data[31:24]};


assign lh_mem_data =
    (alu_result[1:0] == 2'b00) ? {{16{Read_data[15]}}, Read_data
       [15:0]} :
                                    {{16{Read_data[31]}}, Read_data
                                       [31:16]};

assign lhu_mem_data =
    (alu_result[1:0] == 2'b00) ? {16'h0, Read_data[15:0]}  :
                                    {16'h0, Read_data[31:16]};
```

这部分代码是对访存类指令的数据处理:

- lwl 指令，部分字节加载，利用访存地址低两位 alu_result[1:0] 来决定数据拼接方式，部分字节用内存读数据，剩余部分用原寄存器数据 rtdata 补齐。
- lwr 指令，和 lwl 类似，但拼接方向相反，也是根据访存地址低两位进行拼接。
- lb 指令，带符号字节加载。提取对应字节后，按符号位进行扩展到 32 位。
- lbu 指令，无符号字节加载，直接零扩展对应字节。
- lh 和 lhu 指令，分别是带符号和无符号半字加载，按照地址低位判断加载低半字还是高半字，并做符号/零扩展。

```
    assign mem_data =
    (opcode[2:0] == 3'b000) ? lb_mem_data   :
    (opcode[2:0] == 3'b100) ? lbu_mem_data  :
    (opcode[2:0] == 3'b001) ? lh_mem_data   :
    (opcode[2:0] == 3'b101) ? lhu_mem_data  :
    (opcode[2:0] == 3'b011) ? Read_data     :
```

```
    (opcode[2:0] == 3'b010) ? lwl_data      :
                              lwr_data;    // 默认分支
```

根据指令 opcode[2:0] 的低 3 位判断是哪种加载指令，选择对应的处理数据：

表 6: 加载指令类型与对应的加载数据选择

opcode[2:0]	加载指令	对应加载数据
000	lb	带符号字节加载 (lb_mem_data)
100	lbu	无符号字节加载 (lbu_mem_data)
001	lh	带符号半字加载 (lh_mem_data)
101	lhu	无符号半字加载 (lhu_mem_data)
011	lw	32 位字加载 (Read_data)
010	lwl	非对齐字左加载 (lwl_data)
其它	lwr	非对齐字右加载 (lwr_data)

```
// I_type内存写
assign Write_data =
    opcode == 6'b101000 ? rtdata << (8 * alu_result[1:0]) :
    opcode == 6'b101001 ? rtdata << (16 * alu_result[1]) :
    opcode == 6'b101011 ? rtdata :
    (opcode == 6'b101010 && alu_result[1:0] == 2'b00) ? {24'b0,
       rtdata[31:24]} :
    (opcode == 6'b101010 && alu_result[1:0] == 2'b01) ? {16'b0,
       rtdata[31:16]} :
    (opcode == 6'b101010 && alu_result[1:0] == 2'b10) ? {8'b0, rtdata
       [31:8]} :
    (opcode == 6'b101010 && alu_result[1:0] == 2'b11) ? rtdata :
    (opcode == 6'b101110 && alu_result[1:0] == 2'b00) ? rtdata :
    (opcode == 6'b101110 && alu_result[1:0] == 2'b01) ? {rtdata
       [23:0], 8'b0} :
    (opcode == 6'b101110 && alu_result[1:0] == 2'b11) ? {rtdata[7:0],
        24'b0} :
    {rtdata[15:0], 16'b0};
```

这部分代码是对存储指令的数据处理，根据访存地址低位偏移对寄存器数据进行相应位移或字节拼接，确保数据正确对齐写入。

```
// 生成字节写使能信号（4bit）
assign strb_1[0] = (alu_result[1:0] == 2'b00);
assign strb_1[1] = (alu_result[1:0] == 2'b01);
```

```
assign strb_1[2] = (alu_result[1:0] == 2'b10);
assign strb_1[3] = (alu_result[1:0] == 2'b11);

assign strb_2[1:0] = {2{(alu_result[1] == 1'b0)}};
assign strb_2[3:2] = {2{(alu_result[1] == 1'b1)}};

assign strb_3 = 4'b1111;

assign strb_4 = 4'b1111 >> (3 - alu_result[1:0]);

assign strb_5 = 4'b1111 << alu_result[1:0];

assign Write_strb =
    opcode == 6'b101000 ? strb_1 :
    opcode == 6'b101001 ? strb_2 :
    opcode == 6'b101011 ? strb_3 :
    opcode == 6'b101010 ? strb_4 :
                          strb_5;
```

根据访存指令和地址低两位生成正确的字节写使能（Write_strb），控制写内存时哪些字节被写入，保证对齐和部分写入的正确性。

2.4.8 程序计数器（PC）更新

```
reg[31:0]reg_pc;
  always @(posedge clk or posedge rst) begin
                if (rst) begin
                        reg_pc <= 32'b0;          // 复位时 PC 设为初始地址
                end else begin
                        if (is_jorb) begin
                                reg_pc <= next_pc;        // 发生跳转时更新为 next_pc
                        end else begin
                        reg_pc <= pc_add4;         // 顺序执行时 PC += 4
                        end
                end
        end

```

```
14  assign PC = reg_pc;
```

当复位信号 rst 为高时，PC 被初始化为 0，通常代表程序起始位置。
is_jorb 是跳转信号，如果跳转，则 PC 更新为跳转地址 next_pc。
否则，PC 加 4，顺序执行下一条指令。

2.5 最终执行结果

测试通过，波形图如下：

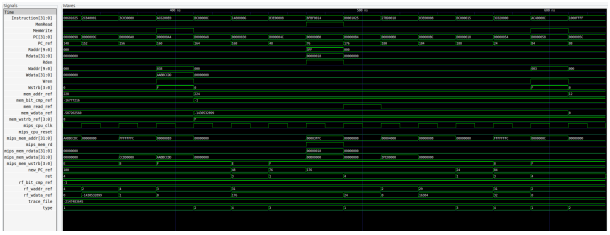

图 2: 最终波形图

3 实验中遇到的问题

3.1 gtkwave 波形显示问题

最开始 gtkwave 的波形只能显示到 90ns，无法通过波形来 debug，多次尝试无果后选择卸载重装 gtkwave，最终波形可以显示到 630ns。

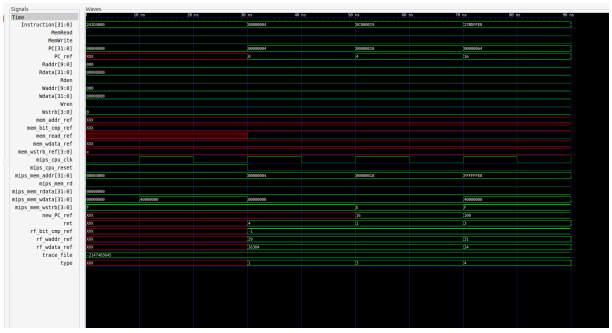

图 3: 问题波形

3.2 ALU 模块的设计漏洞

在 basic、medium、advanced 三个阶段，ALU 模块的设计也经历了三次改变。
basic 阶段的 ALU 模块如下：

```
1 module alu(
2   input [`DATA_WIDTH - 1:0] A,
```

```
    input [`DATA_WIDTH - 1:0] B,
    input [2:0] ALUop,
    output Overflow,
    output CarryOut,
    output Zero,
    output [`DATA_WIDTH - 1:0] Result
);

    // TODO: Please add your logic code here
    wire [`DATA_WIDTH - 1:0]and_result,or_result,add_result,sub_result,
        slt_result,xor_result, nor_result, sltu_result;
    wire add_carryout,sub_carryout;
    wire add_overflow,sub_overflow;

    assign and_result=A&B;

    assign or_result=A|B;

    assign xor_result=A^B;

    assign nor_result=~(A|B);

    assign {add_carryout,add_result}=A+B;

    assign {sub_carryout,sub_result}=A+(~B+1'b1);

    assign {compare_carryout,slt_result}=A+(~B+1'b1);

    wire [31:0] dummy;
    wire sltu_carryout;
    assign {sltu_carryout, dummy} = A + (~B + 1'b1);
    assign sltu_result = (sltu_carryout == 1'b0) ? 32'd1 : 32'd0;

    assign add_overflow=(A[`DATA_WIDTH-1]&&B[`DATA_WIDTH-1]&&~
        add_result[`DATA_WIDTH-1])|(~A[`DATA_WIDTH-1]&&~B[`DATA_WIDTH
        -1]&&add_result[`DATA_WIDTH-1]);
    assign sub_overflow=(A[`DATA_WIDTH-1]&&~B[`DATA_WIDTH-1]&&~
        sub_result[`DATA_WIDTH-1])|(~A[`DATA_WIDTH-1]&&B[`DATA_WIDTH
        -1]&&sub_result[`DATA_WIDTH-1]);

```

```
  wire slt_overflow = (~A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1]) |
                      (A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
                         sub_result[`DATA_WIDTH-1]) |
                      (~A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1] & ~
                         sub_result[`DATA_WIDTH-1]);
    assign slt_result = (sub_result[`DATA_WIDTH-1] ^ slt_overflow) ?
       32'b1 : 32'b0;

  assign Result = (ALUop == 3'b000) ? and_result :
                  (ALUop == 3'b001) ? or_result  :
                  (ALUop == 3'b010) ? add_result :
                  (ALUop == 3'b110) ? sub_result :
                  (ALUop == 3'b111) ? slt_result :
                  (ALUop == 3'b011) ? xor_result :
                  (ALUop == 3'b100) ? nor_result :
                  (ALUop == 3'b101) ? sltu_result :
                  32'b0;

assign Overflow = (ALUop == 3'b010) ? add_overflow :
                    (ALUop == 3'b110) ? sub_overflow :
                    1'b0;

assign CarryOut = (ALUop == 3'b010) ? add_carryout :
                    (ALUop == 3'b110) ? sub_carryout :
                    1'b0;

assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;



endmodule
```

可以看到第一版 ALU 模块计算溢出较为复杂，需要针对加法和减法分别判断；对 SLT 有符号比较的溢出判断比较冗长且手动写死。

溢出和比较处理繁琐，不够统一，可读性和维护性一般。

basic 阶段指令较简单，使用的 ALU 模块即使稍有漏洞也能顺利完成指令。

medium 阶段的 ALU 设计如下：

```
module alu(
    input [DATA_WIDTH - 1:0] A,
    input [DATA_WIDTH - 1:0] B,
```

```
    input [2:0] ALUop,
    output Overflow,
    output CarryOut,
    output Zero,
    output [DATA_WIDTH - 1:0] Result
);
    wire [DATA_WIDTH - 1:0] and_result, or_result, add_result;
    wire [DATA_WIDTH - 1:0] xor_result, nor_result, slt_result,
       sltu_result;
    wire add_carryout, add_overflow;

    // 基本逻辑运算
    assign and_result = A & B;
    assign or_result  = A | B;
    assign xor_result = A ^ B;
    assign nor_result = ~(A | B);

    // 加法运算（用于 add/addu/addiu/lw/sw）
    assign {add_carryout, add_result} = A + B;
    assign add_overflow = (~A[31] & ~B[31] & add_result[31]) | (A[31]
        & B[31] & ~add_result[31]);

    // 比较运算（基于加法实现）
    wire [31:0] neg_B = ~B + 1;  // B 的补码（相当于 -B）
    wire [31:0] diff;            // A - B（通过 A + (-B) 实现）
    wire diff_carryout;
    assign {diff_carryout, diff} = A + neg_B;  // diff = A - B（基于
       加法）

    // SLT（有符号比较）：A < B ?
    wire slt_sign = diff[31];  // 若 A - B 为负，则 A < B
    wire slt_overflow = (A[31] != B[31]) & (diff[31] != A[31]);  //
       溢出检测
    assign slt_result = (slt_sign ^ slt_overflow) ? 32'b1 : 32'b0;

    // SLTU（无符号比较）：A < B ?
    assign sltu_result = (diff_carryout == 0) ? 32'b1 : 32'b0;  // 若
       无进位，则 A < B

    // 结果选择
```

```
    assign Result = (ALUop == 3'b000) ? and_result :     // AND
                    (ALUop == 3'b001) ? or_result  :     // OR
                    (ALUop == 3'b010) ? add_result :     // ADD
                    (ALUop == 3'b110) ? diff :           // SUB (A - B
                        )
                    (ALUop == 3'b111) ? slt_result :     // SLT (有符
                        号比较)
                    (ALUop == 3'b101) ? sltu_result :    // SLTU (无符
                        号比较)
                    (ALUop == 3'b011) ? xor_result :     // XOR
                    (ALUop == 3'b100) ? nor_result :     // NOR
                    32'b0;

    // 标志位
    assign Overflow = (ALUop == 3'b010) ? add_overflow :  // ADD 溢出
                      (ALUop == 3'b110) ? slt_overflow :  // SUB 溢出
                      1'b0;

    assign CarryOut = (ALUop == 3'b010) ? add_carryout :  // ADD 进位
                      (ALUop == 3'b110) ? diff_carryout : // SUB 进位
                      1'b0;

    assign Zero = (Result == 32'b0);  // 结果是否为 0
endmodule
```

这一版设计相较于第一版，统一使用 diff = A + (B + 1) 来表示减法 (SUB)，溢出判断更简洁，但同时也存在很多问题。

```
wire slt_overflow = (A[31] != B[31]) & (diff[31] != A[31]);
```

在处理减法溢出时仍存在一定局限，有些边界情况可能没有正确捕获。SLT 和 SLTU 的处理也非常混乱，容易出错。

这些问题虽然在 medium 阶段没有暴露出来，但在 advanced 阶段却成为了频繁报错的重要原因。

而在最终版着重对溢出和比较进行修改，统一了加减法逻辑；溢出判断简化为判断“操作数符号相同但结果符号不同”；SLT 的符号位判断更加合理，考虑了溢出对比较的影响。

由此可见，ALU 模块的设计是一个逐渐完善的过程，在设计过程中不仅需要考虑操作的正确性，还需要考虑代码的可维护性和简洁性。

3.3 Top 模块的设计思路优化

在 basic 和 medium 阶段，我的 top 模块主要设计思路还是先枚举所有指令的操作码，再通过或运算设计使能信号控制、ALU 操作码等。部分代码如下：

```
    wire is_addiu = (opcode == 6'b001001);
    wire is_bne   = (opcode == 6'b000101);
    wire is_lw    = (opcode == 6'b100011);
    wire is_sw    = (opcode == 6'b101011);
    wire is_sll   = (opcode == 6'b000000 && funct == 6'b000000);
    ...
    // ALU操作码选择
    reg [2:0] alu_op;
    always @(*) begin
        if (is_addiu || is_addu || is_lw || is_sw || is_jal)
            alu_op = 3'b010; // 加法
        else if (is_beq || is_bne)
            alu_op = 3'b110; // 减法
        else if (is_or || is_ori)
            alu_op = 3'b001; // 或
        else if (is_slt || is_slti)
            alu_op = 3'b111; // 有符号比较
        else if (is_sltiu)
            alu_op = 3'b101; // 无符号比较
        else
            alu_op = 3'b000; // 默认与操作
    end

    // ALU输入选择
    wire [31:0] alu_A = rs_val;
    wire [31:0] alu_B = (is_addiu || is_lw || is_sw || is_slti ||
       is_sltiu) ? imm_ext :
                        (is_ori) ? imm_zero :
                        (is_lui) ? {imm, 16'b0} : // lui特殊处理
                        rt_val;
        ...
assign RF_wen = is_addiu || is_lw || is_sll || is_addu || is_or ||
   is_ori ||
                    is_slt || is_slti || is_sltiu || is_lui || is_jal
                       ;

```

```
34      assign RF_waddr = is_jal ? 5'd31 : // jal写$ra
35                        (is_sll || is_addu || is_or || is_slt) ? rd :
                             // R-type写rd
36                        rt; // 其他写rt
37
38      assign RF_wdata = is_jal ? PC + 8 : // jal保存PC+8
39                        is_lui ? {imm, 16'b0} : // lui处理
40                        is_sll ? shifter_result : // sll写移位结果
41                        is_lw ? Read_data : // lw写内存数据
42                        alu_result; // 其他写ALU结果
43          ...
```

但事实证明这样的设计在指令较少时还勉强可以胜任，当进入 advanced 阶段，想要通过这种方式实现 45 种指令无疑是非常困难的。同时这种设计非常繁琐，逻辑混乱，缺少对指令分类信号（如 R_TYPE、I_TYPE 等）的统一管理，扩展难度较大。

因此在 advanced 阶段，我选择将指令模块化细分，采用更加细致的写使能控制处理，考虑更多特殊情况，使得各个模块更加完善，不会出现遗漏。

同时也感谢计算机科学与技术专业的官祥磊同学在这部分提供的帮助，为我如何优化 top 模块提供了思路和指导。

4 实验思考与心得

本次实验对我来说确实是一次挑战，但也是一次收获。课本上的概念在编写代码的过程中逐渐清晰，每一条指令都有完整的生命周期，从取指、译码到执行、访存和写回，每一个环节都需要精心设计和调试。

调试阶段给了我最为宝贵的实践经验。通过反复查看仿真波形，我学会了如何系统地分析问题，从指令译码到控制信号生成，再到执行单元的输出，每一个环节都需要严格验证。这种排错过程虽然痛苦，但却极大地提升了我的硬件调试能力。

最后再次感谢官祥磊同学提供的帮助与指导，为我优化设计思路指明方向。

计算机组成原理不是抽象的理论，而是可以亲手实现的具体系统。每一个看似简单的功能背后，都需要周密的思考和反复的验证。