计算机组成原理第 7 次实验报告

实验名称：单周期 RISC-V 指令集 CPU

学号：2313546　　姓名：蒋枘言　　班次：1078

一、实验目的

1. 了解 RISC-V 指令集。
2. 实现简单的 RISC-V 指令集 CPU。

二、实验步骤

RISC-V 是一种开源、精简指令集架构，有 6 种常见的指令格式。

格式	用途
R-type	寄存器-寄存器操作（如加法）
I-type	立即数操作 / 加载指令
S-type	存储指令
B-type	分支指令
U-type	高位立即数加载
J-type	跳转指令

例如：

```
add x5, x6, x7          # R-type: x5 = x6 + x7
addi x5, x6, 10          # I-type: x5 = x6 + 10
lw x5, 0(x6)             # I-type:  加载内存[x6 + 0]  到  x5
sw x5, 0(x6)             # S-type:  存储  x5  到内存[x6 + 0]
beq x5, x6, label        # B-type:  如果  x5 == x6，跳转到  label
lui x5, 0x12345          # U-type: x5 = 0x12345000
jal x1, label            # J-type:  跳转并将返回地址存在 x1
```

RISC-V RV32I 基础特性支持：

1. 32 位指令长度
2. 32 个 32 位通用寄存器（x0 固定为 0）
3. 基础整数指令集（算术、逻辑、分支、访存）
4. 简化的指令格式（R/I/S/B/U/J 型）

（一）模块功能详解

1. 程序计数器（PC）

功能：存储当前指令地址

更新逻辑：

```
always @(posedge clk or posedge reset)
   if (reset) pc <= 0;
```

```
    else pc <= pc_next;
```

2. 指令存储器（IMEM）

实现：同步 ROM（实际为寄存器数组）

寻址：pc[7:2]（按字寻址，忽略最低 2 位）

特点：组合逻辑读取

3. 寄存器文件（RF）

结构：32 个 32 位寄存器

特性：①x0 始终为 0；②写操作在时钟上升沿进行；③读操作为组合逻辑。

关键逻辑：

```
// 读取
assign rs1_data = (rs1_addr != 0) ? reg_file[rs1_addr] : 0;

// 写入
always @(posedge clk)
   if (reg_write && rd_addr != 0)
      reg_file[rd_addr] <= wb_data;
```

4. 控制单元（Control Unit）

输入：指令的 opcode 字段（inst[6:0]）

输出：所有控制信号

信号含义：

信号	含义	有效指令
reg_write	寄存器写使能	R/I/L/S/J/U 型
alu_src	ALU 操作数 2 选择(0:reg,1:imm)	I/S 型指令
mem_write	数据存储器写使能	S 型指令(SW/SH/SB)
mem_to_reg	写回数据选择(0:ALU,1:MEM)	Load 指令
branch	分支指令标志	BEQ/BNE 等条件分支
jump	跳转指令标志	JAL/JALR

5. ALU 控制单元

输入：主控制单元的 alu_op + 指令的 funct3/funct7 字段

功能：生成具体的 ALU 操作码

映射关系：

```
case (alu_op)
   3'b000: alu_ctrl = 4'b0010; //  访存指令(ADD)
   3'b001: alu_ctrl = 4'b0110; //  分支指令(SUB)
   3'b010: case ({funct7[5], funct3}) // R 型
                4'b0000: alu_ctrl = 4'b0010; // ADD
                4'b1000: alu_ctrl = 4'b0110; // SUB
            endcase
```

```
    // ...
endcase
```

6. 算术逻辑单元（ALU）

支持操作：

```
case (alu_ctrl)
    4'b0000: result = a & b;      // AND
    4'b0001: result = a | b;      // OR
    4'b0010: result = a + b;      // ADD
    4'b0110: result = a - b;      // SUB
    4'b0111: result = (a < b); // SLT
    4'b1010: result = b;            //  直接输出(用于 LUI)
endcase
```

7. 数据存储器（DMEM）

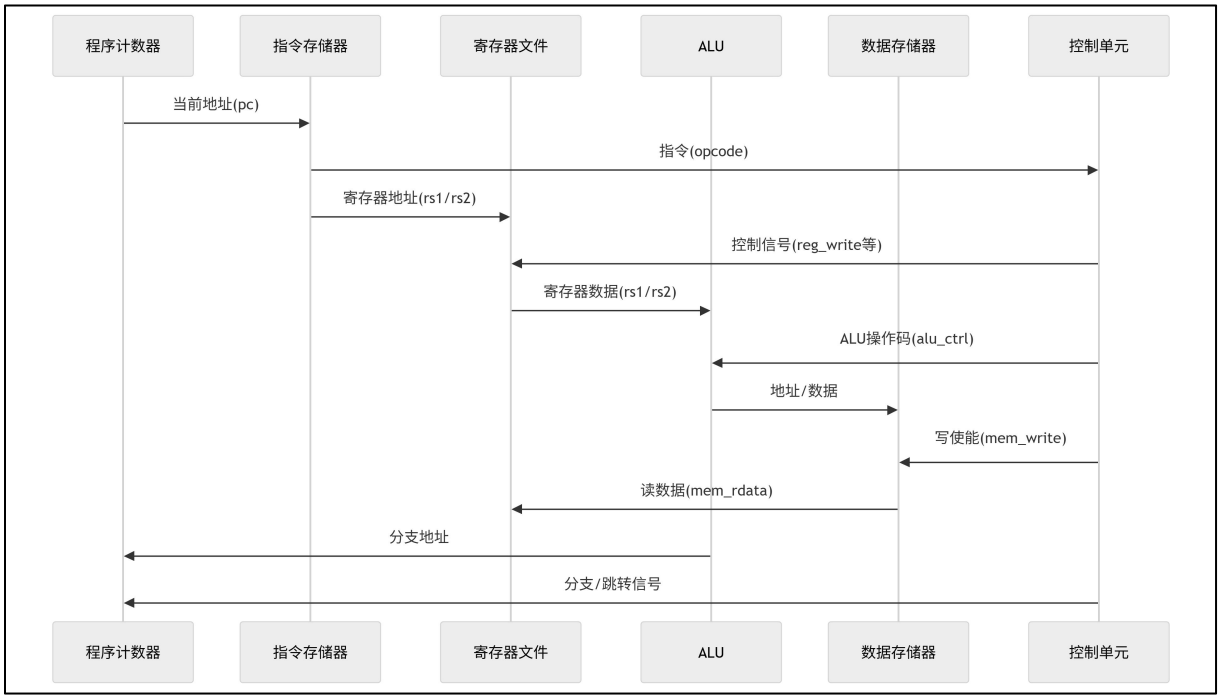
实现：同步 RAM（寄存器数组）

特点：①读操作为组合逻辑；②写操作在时钟上升沿进行

寻址：alu_result[7:2]（按字寻址）

（二）关键数据通路设计

1. 指令执行流程



2. 多路选择器关键路径

ALU 操作数 2 选择：

```
assign alu_src2 = alu_src ? imm_i : rs2_data;
```

写回数据选择：

```
assign wb_data = mem_to_reg ? mem_rdata : alu_result;
```

PC 更新选择：

```
assign pc_src = (branch & alu_zero) | jump;
assign pc_next = pc_src ?
```

```
                (jump ? (rs1_data + imm_i) : (pc + imm_b)) :
                (pc + 4);
```

3. 立即数生成逻辑

RISC-V 的 5 种立即数格式：

```
// I-type: 算术/访存指令
wire [31:0] imm_i = {{20{instr[31]}}, instr[31:20]};

// S-type: 存储指令
wire [31:0] imm_s = {{20{instr[31]}}, instr[31:25], instr[11:7]};

// B-type: 分支指令
wire [31:0] imm_b = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};

// U-type: 高位立即数
wire [31:0] imm_u = {instr[31:12], 12'b0};

// J-type: 跳转指令
wire [31:0] imm_j = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
```

（三）控制信号设计

1. 控制信号生成逻辑

```
always @(*) begin
    // 默认值
    reg_write = 0; alu_src = 0; alu_op = 3'b000;
    mem_write = 0; mem_to_reg = 0; branch = 0; jump = 0;

    case (opcode)
        // R-type (ADD, SUB, etc.)
        7'b0110011: begin
            reg_write = 1;
            alu_op = 3'b010; // 指示 R 型操作
        end

        // I-type (ADDI, LW)
        7'b0010011, 7'b0000011: begin
            reg_write = 1;
            alu_src = 1;       // 使用立即数
            alu_op = (opcode == 7'b0000011) ? 3'b000 : 3'b011;
            mem_to_reg = (opcode == 7'b0000011); // LW 使用内存数据
        end

        // S-type (SW)
        7'b0100011: begin
            alu_src = 1;
```

```
                mem_write = 1;
            end

            // B-type (BEQ, BNE)
            7'b1100011: begin
                branch = 1;
                alu_op = 3'b001; //  减法比较
            end

            // JAL
            7'b1101111: begin
                reg_write = 1;
                jump = 1;
            end

            // JALR
            7'b1100111: begin
                reg_write = 1;
                jump = 1;
                alu_src = 1;
            end
        endcase
    end
```

2. 关键控制场景

(1) 算术指令（ADD）

指令：add x1, x2, x3

数据通路：

1. RF 读取 x2,x3 → ALU
2. ALU 执行加法 → 结果写回 x1

控制信号：

reg_write=1, alu_src=0, mem_write=0, mem_to_reg=0

(2) 访存指令（LW）

指令：lw x1, 4(x2)

数据通路：

1. RF 读取 x2 → ALU+立即数 4 计算地址
2. DMEM 读取该地址数据 → 写回 x1

控制信号：

reg_write=1, alu_src=1, mem_to_reg=1

(3) 分支指令（BEQ）

指令：beq x1, x2, label

数据通路：

1. RF 读取 x1,x2 → ALU 减法
2. 若结果为零，PC = PC + imm_b

控制信号：

branch=1, alu_op=3'b001

（四）完整代码

```
module riscv_cpu_single_cycle (
    input   wire            clk,
    input   wire            reset
);

    // 指令存储器
    reg [31:0] instr_mem [0:63];   // 64x32 位指令存储器
    wire [31:0] instr;             // 当前指令

    // 数据存储器
    reg [31:0] data_mem [0:63];    // 64x32 位数据存储器
    wire [31:0] mem_rdata;         // 存储器读取数据

    // 程序计数器
    reg [31:0] pc;                 // 程序计数器
    wire [31:0] pc_next;           // 下一个 PC 值

    // 寄存器文件
    reg [31:0] reg_file [0:31];    // 32x32 位寄存器文件
    wire [4:0] rs1_addr = instr[19:15];   // 源寄存器 1 地址
    wire [4:0] rs2_addr = instr[24:20];   // 源寄存器 2 地址
    wire [4:0] rd_addr  = instr[11:7];    // 目的寄存器地址

    // 寄存器值
    wire [31:0] rs1_data;          // 源寄存器 1 数据
    wire [31:0] rs2_data;          // 源寄存器 2 数据

    // 立即数生成
    wire [31:0] imm_i = {{20{instr[31]}}, instr[31:20]};                  // I-type
    wire [31:0] imm_s = {{20{instr[31]}}, instr[31:25], instr[11:7]}; // S-type
    wire [31:0] imm_b = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0}; // B-type
    wire [31:0] imm_u = {instr[31:12], 12'b0};                                  // U-type
    wire [31:0] imm_j = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}; // J-type

    // ALU 操作数
    wire [31:0] alu_src1 = rs1_data;
    wire [31:0] alu_src2;

    // ALU
    wire [31:0] alu_result;
    wire        alu_zero;

    // 控制信号
    wire        reg_write;         // 寄存器写使能
    wire        alu_src;           // ALU 操作数 2 选择  (0:寄存器, 1:立即数)
    wire [2:0]  alu_op;            // ALU 操作类型
    wire        mem_write;         // 存储器写使能
    wire        mem_to_reg;        // 写回数据选择  (0:ALU 结果, 1:存储器数据)
    wire        branch;            // 分支指令
    wire        jump;              // 跳转指令

    // 分支/跳转逻辑
    wire        pc_src = (branch & alu_zero) | jump;

    // 指令获取
    assign instr = instr_mem[pc[7:2]];   // 按字寻址(4 字节对齐)

    // 寄存器文件读取
    assign rs1_data = (rs1_addr != 0) ? reg_file[rs1_addr] : 0;
```

```
assign rs2_data = (rs2_addr != 0) ? reg_file[rs2_addr] : 0;

// ALU 操作数 2 选择
assign alu_src2 = alu_src ? imm_i : rs2_data;

// 主控制单元
control_unit ctrl (
    .opcode     (instr[6:0]),
    .reg_write  (reg_write),
    .alu_src    (alu_src),
    .alu_op     (alu_op),
    .mem_write  (mem_write),
    .mem_to_reg (mem_to_reg),
    .branch     (branch),
    .jump       (jump)
);

// ALU 控制单元
wire [3:0] alu_ctrl;
alu_control alu_ctrl_unit (
    .alu_op     (alu_op),
    .funct3     (instr[14:12]),
    .funct7     (instr[31:25]),
    .alu_ctrl   (alu_ctrl)
);

// ALU 实例
alu main_alu (
    .a          (alu_src1),
    .b          (alu_src2),
    .alu_ctrl   (alu_ctrl),
    .result     (alu_result),
    .zero       (alu_zero)
);

// 存储器访问
assign mem_rdata = data_mem[alu_result[7:2]];  // 按字寻址

// 写回数据选择
wire [31:0] wb_data = mem_to_reg ? mem_rdata : alu_result;

// 下一条 PC 计算
assign pc_next = pc_src ?
                 (jump ? (rs1_data + imm_i) : (pc + imm_b)) :  // 跳转或分支
                 (pc + 4);  // 顺序执行

// 程序计数器更新
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc <= 32'h0000_0000;  // 复位时 PC 归零
    end else begin
        pc <= pc_next;
    end
end

// 寄存器文件写入
always @(posedge clk) begin
    if (reg_write && rd_addr != 0) begin
        reg_file[rd_addr] <= wb_data;
    end
end

// 数据存储器写入
always @(posedge clk) begin
```

```
        if (mem_write) begin
            data_mem[alu_result[7:2]] <= rs2_data;
        end
    end

    // 初始化存储器（实际使用时应加载程序）
    initial begin
        // 示例：ADD x1, x0, 5 -> ADDI x1, x0, 5
        instr_mem[0] = 32'h00500093;    // [31:20]=5, rd=1
        // 示例：ADD x2, x1, x1
        instr_mem[1] = 32'h00108133;    // rs1=1, rs2=1, rd=2
        // 示例：SW x2, 0(x0)
        instr_mem[2] = 32'h00202023;    // rs1=0, rs2=2, imm=0
    end

endmodule

// 主控制单元
module control_unit (
    input    wire [6:0] opcode,         // 操作码
    output reg          reg_write,    // 寄存器写使能
    output reg          alu_src,      // ALU 操作数 2 选择
    output reg [2:0]    alu_op,       // ALU 操作类型
    output reg          mem_write,    // 存储器写使能
    output reg          mem_to_reg,   // 写回数据选择
    output reg          branch,       // 分支指令
    output reg          jump          // 跳转指令
);

    always @(*) begin
        // 默认值
        reg_write = 0;
        alu_src   = 0;
        alu_op    = 3'b000;
        mem_write = 0;
        mem_to_reg= 0;
        branch    = 0;
        jump      = 0;

        case (opcode)
            // R-type 指令 (ADD, SUB, etc.)
            7'b0110011: begin
                reg_write = 1;
                alu_op    = 3'b010;
            end

            // I-type 算术指令 (ADDI, etc.)
            7'b0010011: begin
                reg_write = 1;
                alu_src   = 1;
                alu_op    = 3'b011;
            end

            // Load 指令 (LW)
            7'b0000011: begin
                reg_write = 1;
                alu_src   = 1;
                mem_to_reg= 1;
                alu_op    = 3'b000;
            end

            // Store 指令 (SW)
            7'b0100011: begin
```

```
                alu_src     = 1;
                mem_write = 1;
                alu_op      = 3'b000;
            end

            // Branch 指令  (BEQ)
            7'b1100011: begin
                branch      = 1;
                alu_op      = 3'b001;
            end

            // JAL 指令
            7'b1101111: begin
                reg_write = 1;
                jump          = 1;
            end

            // JALR 指令
            7'b1100111: begin
                reg_write = 1;
                jump          = 1;
                alu_src     = 1;
            end

            // LUI 指令
            7'b0110111: begin
                reg_write = 1;
                alu_src     = 1;
                alu_op      = 3'b100;
            end

            // AUIPC 指令
            7'b0010111: begin
                reg_write = 1;
                alu_src     = 1;
                alu_op      = 3'b101;
            end
        endcase
    end
endmodule

// ALU 控制单元
module alu_control (
    input   wire [2:0] alu_op,        //  来自主控制单元
    input   wire [2:0] funct3,        //  指令的 funct3 字段
    input   wire [6:0] funct7,        //  指令的 funct7 字段
    output reg   [3:0] alu_ctrl       // ALU 操作码
);

    always @(*) begin
        case (alu_op)
            //  存储器访问
            3'b000: alu_ctrl = 4'b0010; // ADD (LW/SW)

            //  分支指令
            3'b001: alu_ctrl = 4'b0110; // SUB (BEQ/BNE)

            // R-type 指令
            3'b010: begin
                case ({funct7[5], funct3})
                    // ADD/SUB
                    4'b0000: alu_ctrl = 4'b0010; // ADD
                    4'b1000: alu_ctrl = 4'b0110; // SUB
                    //  其他指令...
```

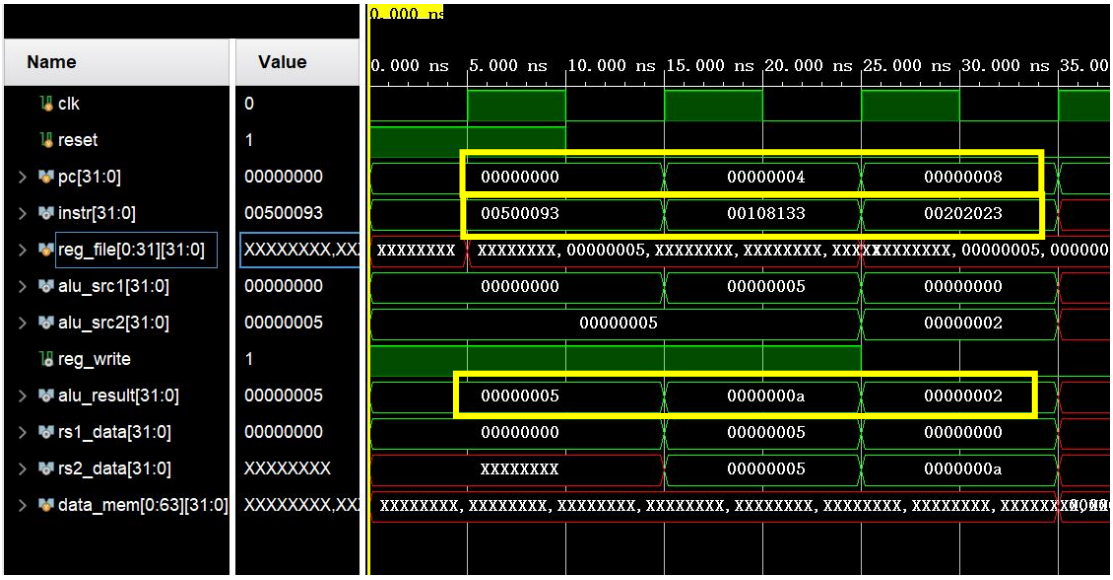
```
                        default: alu_ctrl = 4'b0000;
                    endcase
                end

                // I-type 指令
                3'b011: begin
                    case (funct3)
                        3'b000: alu_ctrl = 4'b0010; // ADDI
                        // 其他立即数指令...
                        default: alu_ctrl = 4'b0000;
                    endcase
                end

                // LUI
                3'b100: alu_ctrl = 4'b1010; // 直接传递立即数

                // AUIPC
                3'b101: alu_ctrl = 4'b0010; // ADD (PC + imm)

                default: alu_ctrl = 4'b0000;
            endcase
        end
endmodule

// ALU 模块
module alu (
        input   wire [31:0] a,              // 操作数 1
        input   wire [31:0] b,              // 操作数 2
        input   wire [3:0]  alu_ctrl, // ALU 控制信号
        output reg  [31:0] result,     // 运算结果
        output wire             zero          // 结果是否为 0
);

        always @(*) begin
            case (alu_ctrl)
                4'b0000: result = a & b;          // AND
                4'b0001: result = a | b;          // OR
                4'b0010: result = a + b;          // ADD
                4'b0110: result = a - b;          // SUB
                4'b0111: result = (a < b) ? 1 : 0; // SLT
                4'b1010: result = b;                // 直接传递 (用于 LUI)
                default: result = 32'h0;
            endcase
        end

        assign zero = (result == 0);
endmodule
```

三、实验结果

目前只验证 3 条指令：

```
ADDI x1, x0, 5      // x1 = 5
ADD x2, x1, x1      // x2 = 10
SW x2, 0(x0)        // 存储到内存 0 地址
```

仿真后的波形图如下：



1. 第一条指令(ADDI x1, x0, 5)

程序计数器 pc = 00000000（初始值）

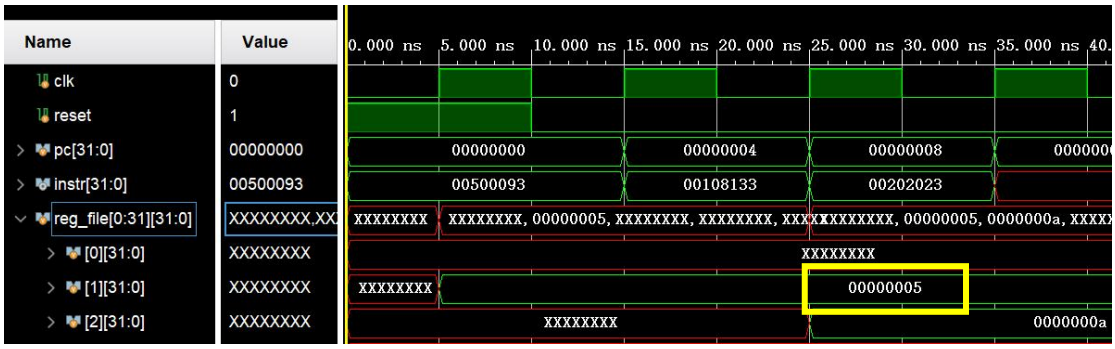
指令操作数 instr = 00500093

alu_src = 1 （使用立即数）

alu_ctrl = 0010（ADD 操作）

reg_write = 1（写寄存器）

下一周期 reg_x1 应变为 5（需展开看）



2. 第二条指令(ADD x2, x1, x1)

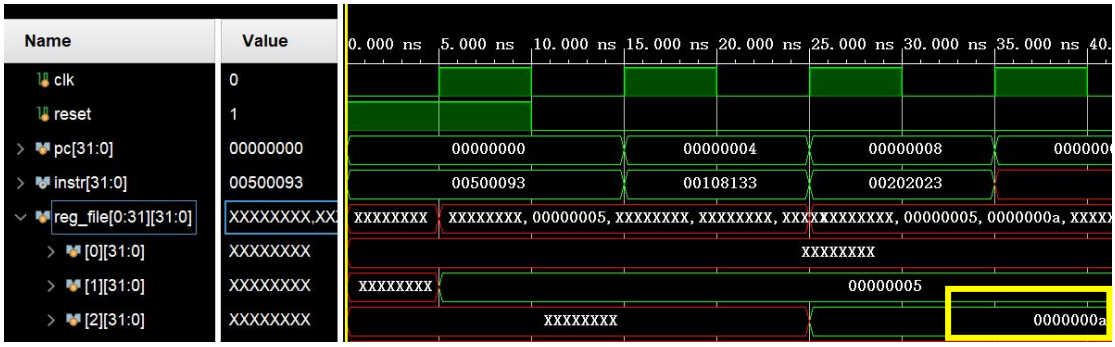
程序计数器 pc = 00000004（加 4）

指令操作数 instr = 00108133

rs1_data 和 rs2_data 都应为 5

alu_result 应变为 10

reg_x2 应更新为 10（十六进制 0000000a）（需展开看）



3. 第三条指令(SW x2, 0(x0))

程序计数器 pc = 00000008（再加 4）

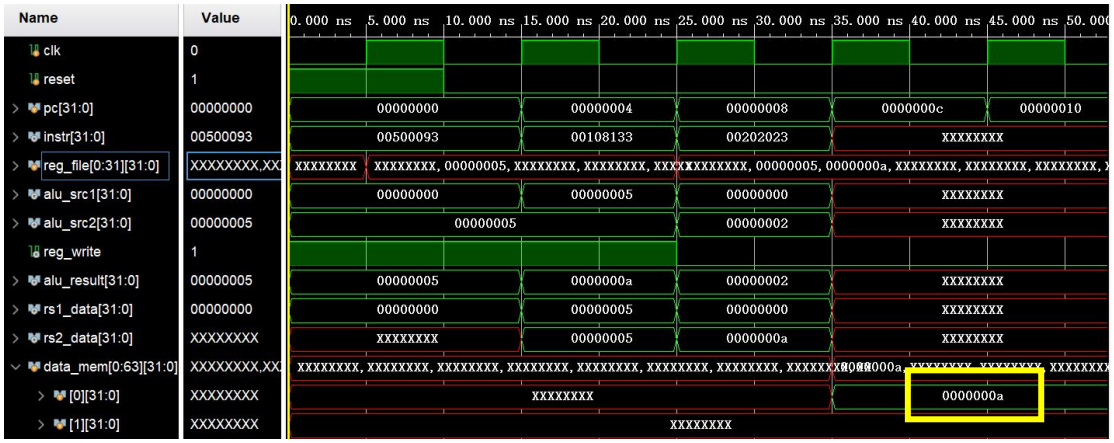
指令操作数 instr = 00202023

mem_write 脉冲为高

alu_result 为 0（存储地址）

rs2_data 为 10（存储值）

下一周期 data_mem[0]应变为 10（十六进制 0000000a）（需展开看）



四、总结感想

通过本次实验，我初步了解了 RISC-V 指令集的知识，发现 RISC-V 指令集与 MIPS 还是有很多区别。RISC-V 指令集控制器逻辑更清晰，指令格式更规整（尤其在立即数处理上），没有$前缀的名字，寄存器按编号命名，RISC-V 指令集还“瘦身”了不少。

本次实验仅完成了**少数基础指令的支持**，无流水线等设计，后续如果有机会再继续深入地学习 RISC-V 指令集。