计算机组成原理（物联网）
实验报告

目录

一、实验目的

本设计旨在深入理解并掌握处理器体系结构的基本原理与实现方法，基于 RISC-V 指令集架构设计并实现一个多周期处理器。RISC-V 是一种开放、精简且模块化的指令集架构，具有良好的教学与工业实践价值。与单周期或流水线结构不同，多周期处理器将指令执行过程划分为多个阶段，并通过共享功能部件和控制信号进行调度，在保证功能正确性的同时，提升硬件资源利用率并降低系统功耗。通过本设计，不仅可以加深对 CPU 微结构的理解，还能掌握 Verilog HDL 编程、RTL 级建模、时序控制和硬件调试等关键技能。

二、实验任务

本实验的任务是设计并实现一个基于 RISC-V 32I 指令集的多周期处理器，支持整数运算、访存操作、分支跳转等基本指令功能。处理器应具备取指、译码、执行、访存和写回五个基本阶段，并通过状态机或控制信号实现每条指令的多周期执行过程。具体任务包括：

1）实现 PC 和指令存储器模块，实现顺序与跳转指令的正确取指；

2）设计寄存器堆、ALU 运算单元和数据存储器模块，实现整数运算与读写功能；

3）编写控制单元，实现对不同类型指令的周期调度和控制信号生成；

4）通过 Testbench 进行功能仿真与波形分析，验证处理器的正确性

表 2.1　本设计所支持的 RISC-V 32I 指令集功能描述

序号	指令	功能描述	指令格式	指令分类
1	ADD	整数加法（有符号）	R	算术运算
2	SUB	整数减法	R	算术运算
3	SLL	左移（逻辑）	R	位移
4	SLT	小于置 1（有符号比较）	R	比较
5	SLTU	小于置 1（无符号比较）	R	比较
6	XOR	异或	R	逻辑运算
7	SRL	右移（逻辑）	R	位移
8	SRA	右移（算术）	R	位移
9	OR	或运算	R	逻辑运算
10	AND	与运算	R	逻辑运算
11	ADDI	加立即数	I	算术运算
12	SLTI	小于立即数置 1（有符号）	I	比较
13	SLTIU	小于立即数置 1（无符号）	I	比较
14	XORI	异或立即数	I	逻辑运算
15	ORI	或立即数	I	逻辑运算
16	ANDI	与立即数	I	逻辑运算
17	SLLI	左移立即数	I	位移
18	SRLI	逻辑右移立即数	I	位移
19	SRAI	算术右移立即数	I	位移
20	LUI	加载高 20 位立即数	U	常量生成
21	AUIPC	加载 PC 相对高位立即数	U	常量生成
22	JAL	跳转并链接	J	跳转控制
23	JALR	寄存器跳转并链接	I	跳转控制

序号	指令	功能描述	指令格式	指令分类
24	BEQ	相等跳转	B	分支控制
25	BNE	不等跳转	B	分支控制
26	BLT	小于跳转（有符号）	B	分支控制
27	BGE	大于等于跳转（有符号）	B	分支控制
28	BLTU	小于跳转（无符号）	B	分支控制
29	BGEU	大于等于跳转（无符号）	B	分支控制
30	LB	读取字节（有符号扩展）	I	访存
31	LH	读取半字（有符号扩展）	I	访存
32	LW	读取字	I	访存
33	LBU	读取字节（零扩展）	I	访存
34	LHU	读取半字（零扩展）	I	访存
35	SB	写入字节	S	访存
36	SH	写入半字	S	访存
37	SW	写入字	S	访存

三、设计与实现

3.1 多周期 CPU 设计与概述

本设计基于 RISC-V 32I 指令集，完成了一个多周期处理器的设计与实现。RISC-V 作为开源、模块化的精简指令集架构，具备结构简单、扩展性强、易于硬件实现等特点，适合作为教学与科研平台。处理器采用多周期结构，将指令执行过程划分为多个阶段（包括取指、译码、执行、访存、写回），通过控制单元按步骤有序驱动各功能模块，有效提升了硬件资源的复用率，设计过程中采用 Verilog HDL 进行模块建模与仿真验证，并通过 Testbench 完成功能测试与指令运行

验证。该多周期处理器具备良好的模块化结构，通过本项目，不仅实现了处理器从指令解码到控制执行的完整流程，也为理解处理器内部微结构和控制机制提供了实践平台。本设计的多周期 CPU 的状态转移图如下图所示：

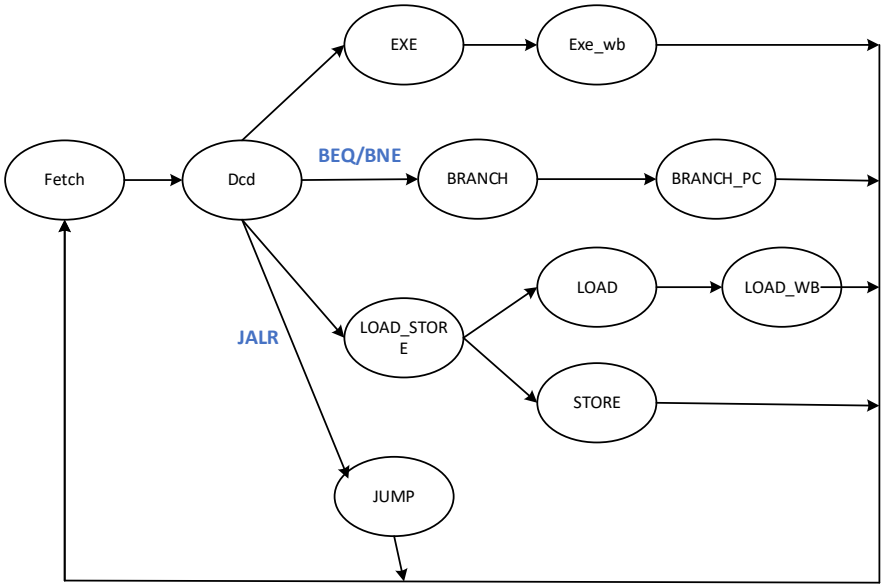

图 3.1　多周期 CPU 状态转换图 (FSM)

3.2 CPU 数据通路图

下面两张图为 Vivado 生成的结构图，清楚了每条指令的功能后，我们就知道了执行这条指令需要的元件。首先必 须要有的是指令寄存器。既然有指令寄存器，就要有从寄存器里取出具体指令的器件，即取指令部件。 指令取出来之后我们要怎样执行它，这还需要一个翻译指令的过程，即一条 指令他的每一小段对应的具体意义是什么。于是就需要一个指令译码器来执行这个任务。我们都知道整个数据通路错综复杂，每一条指令怎么样走出属于他们自己的数据通路呢？没对每个多路选怎的的时候需要有哪些信号来控制呢？

控制单元部件就解决了这一难题。他根据最初输入的指令选取 opcode, func3, func7 指令段。最终会得到每条指令对应的一串各个控制信号的值。在执行的过程中，我么们还需要对一些运算结果进行存储。这个功能就交给寄存器和存储器。还有一些指令是和立即数进行运算。而我们进行的运算都是 32 位的形式。这就需要有一个立即数扩展部件。

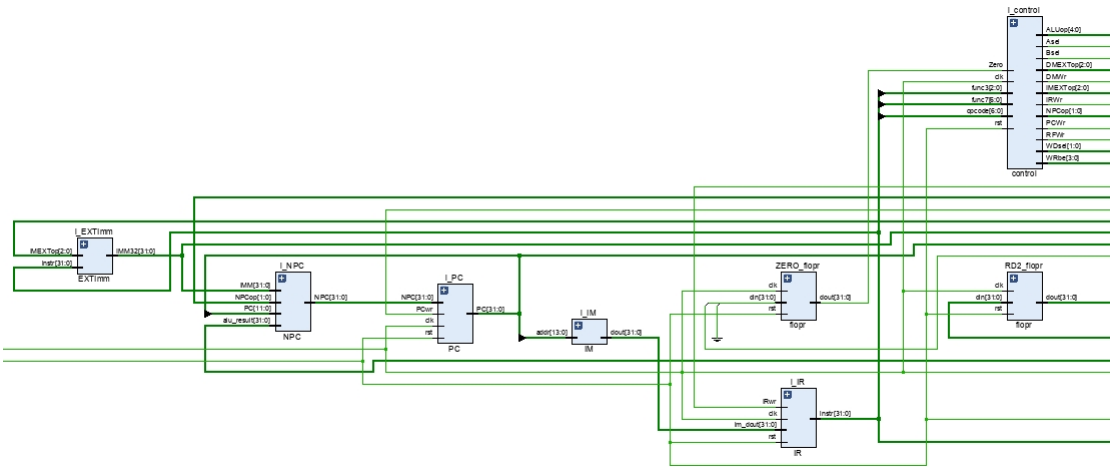

图 3.2　处理器整体数据通路 Vivado 结构图

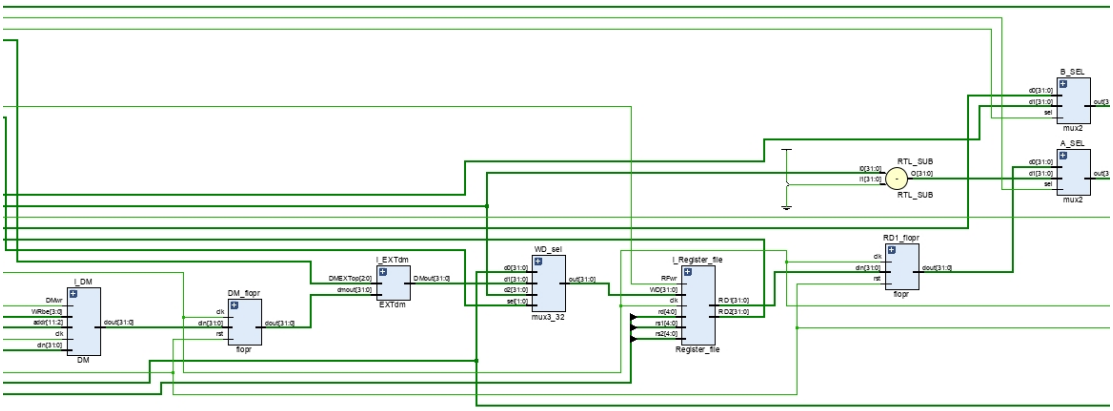

图 3.3　ALU 控制信号功能描述

在执行指令时会遇到不同的运算，这些运算数据大部分是在算术

逻辑部件(ALU)中进行的。在一个模块中，会遇到很多种多个输入单个输出的情况。这需要多路选择器部件来完成这一任务。我们是从简单的 R 型指令的数据通路开始构建其数据通路，在此基础上再构建 I 型指令的数据通路。因为 I 型指令和 R 型指令和相近，就只许添加一个扩展部件。接着我们添加 U 型指令的数据通路，这个指令执行的时候再 R,I 指令的基础上就可以运行。只需改变一些信号的值。接着我们添加了 S 型指令，分为 load 和 store 两类。再接着添加 B 型指令。最后添加 J 型指令。

3.4 ALU 设计

表 3.1 ALU 控制信号功能描述

ALU 控制码（ALU_Control）	操作名称	对应 RISC-V 指令	功能描述
0000	ADD	ADD, ADDI	有符号加法运算
0001	SUB	SUB	有符号减法运算
0010	AND	AND, ANDI	按位与
0011	OR	OR, ORI	按位或
0100	XOR	XOR, XORI	按位异或
0101	SLL	SLL, SLLI	逻辑左移
0110	SRL	SRL, SRLI	逻辑右移
0111	SRA	SRA, SRAI	算术右移
1000	SLT	SLT, SLTI	小于置 1（有符号比较）
1001	SLTU	SLTU, SLTIU	小于置 1（无符号比较）
1010	PASS_B	LUI	直接输出输入操作数 B
1011	BEQ_CMP	BEQ	判断是否相等（用于分支）
1100	BNE_CMP	BNE	判断是否不等

ALU 控制码（ALU_Control）	操作名称	对应 RISC-V 指令	功能描述
1101	BLT_CMP	BLT, BLTU	判断 A 是否小于 B
1110	BGE_CMP	BGE, BGEU	判断 A 是否大于等于 B
1111	NOP	NOP	空操作，输出保持不变

关键代码设计：

```
module ALU(
    input [31:0] A,
    input [4:0]    ALUop,
    output[31:0] result,
    output          Zero

    );

    reg [31:0] result_reg;
    integer    i;

    always @ (A or B or ALUop)
    begin
        case(ALUop)
  `ALUOP_ADD:result_reg = A + B;
  `ALUOP_SLT : result_reg = (A < B) ? 32'd1 : 32'd0;
`ALUOP_SLTU: result_reg = ({1'b0,A}<{1'b0,B}) ? 32'd1 : 32'd0;    SLTU SLTIU
`ALUOP_AND:result_reg = A & B;
`ALUOP_OR    : result_reg = A | B;
`ALUOP_XOR : result_reg = A ^ B;
`ALUOP_SLL : result_reg = (A << B[4:0]);
`ALUOP_SRL : result_reg = (A >> B[4:0]);
`ALUOP_SUB : result_reg = A - B;
```

```
`ALUOP_SRA :    result_reg = A >> B;
`ALUOP_LUI : result_reg = B; //lui
 `ALUOP_AUIPC:result_reg = A + B; //auipc
 `ALUOP_JALR: result_reg = A + B; //jalr
 `ALUOP_BEQ : result_reg = (A == B) ? 32'd1 : 32'd0;//BEQ
 `ALUOP_BNE : result_reg = (A != B) ? 32'd1 : 32'd0;//BNE
 `ALUOP_BLT : result_reg = (A < B)    ? 32'd1 : 32'd0;//BLT
 `ALUOP_BLTU: result_reg = ({1'b0,A}<{1'b0,B}) ? 32'd1 : 32'd0;
 `ALUOP_BGE : result_reg = (A > B)    ? 32'd1 : 32'd0;//BGE
 `ALUOP_BGEU: result_reg = ({1'b0,A}>{1'b0,B}) ? 32'd1 : 32'd0;
  default:         result_reg = 32'd0;
          endcase
    end
 assign result =    result_reg;
 assign Zero     = (result_reg == 32'd1) ? 1'b1 : 1'b0;
```

3.5 控制器设计

表 3.2　控制单元主要信号及其功能说明

信号名	位宽	说明
clk	1	时钟信号，控制状态跳转
rst	1	复位信号，高电平复位
Zero	1	比较结果（如用于分支指令的结果）
opcode	7	指令低 7 位，用于识别指令类型
func3	3	指令 [14:12] 位，进一步区分指令功能
func7	7	指令 [31:25] 位，进一步区分部分 R 型指令
RFWr	1	通用寄存器写使能，高电平表示写入
DMWr	1	数据存储器写使能，高电平表示写入
PCWr	1	PC 写使能，高电平允许写入下一指令地址

信号名	位宽	说明
IRWr	1	指令寄存器写使能，高电平表示从 IM 读取并写入 IR
ALUop	5	ALU 运算操作控制信号，选择不同的运算类型（如加减乘除等）
IMEXTop	3	指令立即数扩展方式（如 I 型、S 型、B 型、U 型、J 型）
DMEXTop	3	数据存储器读出数据的扩展方式（符号扩展、零扩展等）
NPCop	2	NPC 控制方式（00: PC+4, 01: 分支，10: 跳转，11: jalr 等）
WDsel	2	写回寄存器的数据选择（如来自 ALU, DM, 或 PC+4）
WRbe	4	数据存储器写字节使能信号（如 byte, half-word, word）
Asel	1	ALU 输入 A 选择信号（0: 寄存器，1: PC）
Bsel	1	ALU 输入 B 选择信号（0: 寄存器，1: 立即数）

逻辑代码如下：

```
module control(
    input           clk,
    input           rst,
    input           Zero,
    input   [6:0] opcode, //    instr[6:0]
    input   [2:0] func3,
    input   [6:0] func7,

    output reg          RFWr,
    output reg          DMWr,
    output reg          PCWr,

    output reg          IRWr,
    output reg [4:0] ALUop,
    output reg [2:0] IMEXTop,
```

```
  output reg [2:0] DMEXTop，
output reg [1:0] NPCop,//npc
  output reg [1:0] WDsel,//
output reg [3:0] WRbe,         output reg              Asel,
  output reg             Bsel


  );

  localparam    Fetch          =    4'b0000,
                DCD            =    4'b0001,
                EXE            =    4'b0010,
                Branch         =    4'b0011,
                Load_store =    4'b0100,
                JMP            =    4'b0101,
                Load           =    4'b0110,
                Store          =    4'b0111,
                Exe_WB         =    4'b1000,
                Load_WB        =    4'b1001,
                Branch_pc      =    4'b1010;
        wire RType;      // Type of R-Type Instruction
  wire IType;      // Tyoe of Imm        Instruction
 wire BrType;    // Type of Branch Instruction
wire JType;      // Type of Jump      Instruction
  wire LdType;    // Type of Load      Instruction           lb lh lw lbu lhu
  wire StType;    // Type of Store     Instruction      sb sh sw
  wire MemType; // Type pf Memory Instruction(Load/Store)
  wire LUI_AUIPC;//lui    auip
  assign RType      = (opcode == `INSTR_Rtype );
  assign IType      = (opcode == `INSTR_Itype_imm );
  assign BrType     = (opcode == `INSTR_Btype_branch );
```

```
assign JType      = (opcode == `INSTR_Jtype_jal    );
assign LdType     = (opcode == `INSTR_Itype_load );
assign StType     = (opcode == `INSTR_Stype_store );
assign MemType = LdType || StType;
assign LUI_AUIPC=(opcode == `INSTR_Utype_lui || opcode == `INSTR_Utype_auipc ||
opcode == `INSTR_Itype_jalr);
        reg [3:0] state;
reg [3:0] nextstate;

always @(posedge clk or posedge rst) begin
    if(rst)
        state <= Fetch;
    else
        state <= nextstate;
end

always @(*) begin
    case(state)
        Fetch    : nextstate = DCD;
        DCD      : begin
            if(RType || IType || LUI_AUIPC) nextstate = EXE;
            else if(BrType)       nextstate = Branch;
            else if(MemType)      nextstate = Load_store;
            else if(JType)        nextstate = JMP;
            else                  nextstate = Fetch;                    end
        EXE              : nextstate = Exe_WB;
        Branch           : nextstate = Branch_pc;
        Load_store : begin
            if(LdType)            nextstate = Load;
            else if(StType)       nextstate = Store;
```

```
                end
            JMP             : nextstate = Fetch;
            Load            : nextstate = Load_WB;
            Store           : nextstate = Fetch;
            Exe_WB          : nextstate = Fetch;
            Load_WB         : nextstate = Fetch;
            Branch_pc       : nextstate = Fetch;
            default         : ;
        endcase
    end

    always @(*) begin
        case (state)
            Fetch : begin
                        RFWr    =   1'b0;                   DMWr    =
1'b0;               PCWr    = 1'b1;
                IRWr    = 1'b1;             ALUop = 5'b0;
                IMEXTop=3'b0;
            NPCop = `NPC_PLUS4;pc = pc + 4   01             WDsel =
2'b0;               Asel    = 1'b0;
                Bsel    = 1'b0;
                WRbe    = 4'b0;
                DMEXTop=3'b0;
                end
            DCD    : begin
                RFWr    = 1'b0;
                DMWr    = 1'b0;
                PCWr    = 1'b0;
                IRWr    = 1'b0;
                ALUop = 5'b0;
```

```
                IMEXTop=3'b0;
                NPCop = 2'b0;
                WDsel = 2'b0;
                Asel    = 1'b0;
                Bsel    = 1'b0;
                WRbe    = 4'b0;
                DMEXTop=3'b0;
                end
            EXE    : begin
                RFWr    = 1'b0;
                DMWr    = 1'b0;
                PCWr    = 1'b0;
                IRWr    = 1'b0;
                NPCop = 2'b0;
                WDsel = 2'b0;
                WRbe    = 4'b0;
                DMEXTop=3'b0;
                if(LUI_AUIPC)begin
                    Bsel       = 1'b1;
                            if(opcode == `INSTR_Utype_lui)    begin IMEXTop =
`UTYPE_IMM;ALUop = `ALUOP_LUI; Asel = 1'b0;end
                        else if(opcode == `INSTR_Utype_auipc)begin IMEXTop =
`UTYPE_IMM; ALUop = `ALUOP_AUIPC; Asel = 1'b1;end
                    else begin    IMEXTop = `ITYPE_IMM; ALUop = `ALUOP_JALR;
Asel = 1'b0; end
                end
                if(RType ) begin
                    Asel       = 1'b0;
                    Bsel       = 1'b0;
                    IMEXTop = 3'b0;
```

```
        case(func3)
            `FUNCT_ADDSUB : begin
                if(func7 == 7'b0000000) ALUop = `ALUOP_ADD;
                else                    ALUop = `ALUOP_SUB;
                end
            `FUNCT_SLL    : ALUop = `ALUOP_SLL;
            `FUNCT_SLT    : ALUop = `ALUOP_SLT;
            `FUNCT_SLTU   : ALUop = `ALUOP_SLTU;
            `FUNCT_XOR    : ALUop = `ALUOP_XOR;
            `FUNCT_SRLSRA : begin
                if(func7 == 7'b0000000) ALUop = `ALUOP_SRL;
                else                    ALUop = `ALUOP_SRA;
                end
            `FUNCT_OR     : ALUop = `ALUOP_OR;
            `FUNCT_AND    : ALUop = `ALUOP_AND;
        endcase
    end

    if(IType ) begin    andi slli srli/srai
        Asel    = 1'b0;
        Bsel    = 1'b1;
        IMEXTop = `ITYPE_IMM;
        case(func3)
            `FUNCT_ADDI : ALUop = `ALUOP_ADD;
            `FUNCT_SLTI : ALUop = `ALUOP_SLT;
            `FUNCT_SLTIU: ALUop = `ALUOP_SLTU;
            `FUNCT_XORI : ALUop = `ALUOP_XOR;
            `FUNCT_ORI    : ALUop = `ALUOP_OR;
            `FUNCT_ANDI : ALUop = `ALUOP_AND;
            `FUNCT_SLLI : ALUop = `ALUOP_SLL;
```

```
                `FUNCT_SRLISRAI : begin
                    if(func7 == 0000000) ALUop = `ALUOP_SRL;
                    else                 ALUop = `ALUOP_SRA;
                end

            endcase
        end
        end

    Branch : begin
        RFWr    = 1'b0;
        DMWr    = 1'b0;
        IRWr    = 1'b0;
        WDsel = 2'b0;
        Asel    = 1'b0;
        Bsel    = 1'b0;
        WRbe    = 4'b0;
        IMEXTop= 3'b0;
        DMEXTop=3'b0;
        NPCop = 2'b0;
        PCWr    = 1'b0;
        case(func3)
            `FUNCT_BEQ : ALUop = `ALUOP_BEQ;
            `FUNCT_BNE : ALUop = `ALUOP_BNE;
            `FUNCT_BLT : ALUop = `ALUOP_BLT;
            `FUNCT_BGE : ALUop = `ALUOP_BGE;
            `FUNCT_BLTU: ALUop = `ALUOP_BLTU;
            `FUNCT_BGEU: ALUop = `ALUOP_BGEU;
        endcase
        end
```

```
Branch_pc : begin
        RFWr    = 1'b0;
        DMWr    = 1'b0;
        IRWr    = 1'b0;
        WDsel = 2'b0;
        Asel    = 1'b0;
        Bsel    = 1'b0;
        WRbe    = 4'b0;
        ALUop = 5'b0;
        IMEXTop=`BTYPE_IMM;
        DMEXTop=3'b0;
        if(Zero) begin PCWr    = 1'b1; NPCop = `NPC_BRANCH; end
        else          begin PCWr    = 1'b0; NPCop = 2'b0; end
        end
Load_store : begin        //绔嬪嵆鏁版墠灞□ 杩愮畻寰楀埌鏃板潈
        RFWr    = 1'b0;
        DMWr    = 1'b0;
        PCWr    = 1'b0;
        IRWr    = 1'b0;
        ALUop = `ALUOP_ADD;
        NPCop = 2'b0;
        WDsel = 2'b0;
        Asel    = 1'b0;
        Bsel    = 1'b1;
        WRbe    = 4'b0;
        DMEXTop=3'b0;
        if(LdType) IMEXTop = `ITYPE_IMM;
        else IMEXTop = `STYPE_IMM;
        end
Load    : begin           //浠巏 mem 璇诲嚭鏁版嵁
```

```
        RFWr    = 1'b0;
        DMWr    = 1'b0;
        PCWr    = 1'b0;
        IRWr    = 1'b0;
        ALUop = 5'b0;
        IMEXTop=3'b0;
        NPCop = 2'b0;
        WDsel = 2'b0;
        Asel    = 1'b0;
        Bsel    = 1'b0;
        WRbe    = 4'b0;
        end
    Store    : begin          //鍐欏垎 mem
        RFWr    = 1'b0;
        DMWr    = 1'b1;
        PCWr    = 1'b0;
        IRWr    = 1'b0;
        ALUop = 5'b0;
        IMEXTop=3'b0;
        NPCop = 2'b0;
        WDsel = 2'b0;
        Asel    = 1'b0;
        Bsel    = 1'b0;
        //WRbe    = 4'b0;
        DMEXTop=3'b0;
        case(func3)
            `FUNCT_SB : WRbe = 4'b0001;    //store low 8bit
            `FUNCT_SH : WRbe = 4'b0011;    //store low 16bit
            `FUNCT_SW : WRbe = 4'b1111;    //store low 32bit
        endcase
```

```
        end
    JMP     : begin         //鏃犳潯浠惰烦杞□    jal
        RFWr    = 1'b1; //闆€鍐欏櫒 rd
        DMWr    = 1'b0;
        PCWr    = 1'b1;
        IRWr    = 1'b0;
        ALUop = 5'b0;
        IMEXTop = `JTYPE_IMM;
        NPCop = `NPC_JUMP;
        WDsel = `WDSEL_JMP;     //pc+4 to rd
        Asel    = 1'b0;
        Bsel    = 1'b0;
        WRbe    = 4'b0;
        DMEXTop=3'b0;
        end
                        Exe_WB : begin
        RFWr    = 1'b1;
        DMWr    = 1'b0;
        IRWr    = 1'b0;
        ALUop = 5'b0;
        IMEXTop=3'b0;
        Asel    = 1'b0;
        Bsel    = 1'b0;
        WRbe    = 4'b0;
        DMEXTop=3'b0;
        if(opcode == `INSTR_Itype_jalr)begin
            WDsel = `WDSEL_JMP;     //pc+4 to rd
            PCWr    = 1'b1;
            NPCop = `NPC_AUIPC;
        end
```

```
else if(opcode == `INSTR_Utype_auipc)begin
        WDsel = `WDSEL_ALU;
        PCWr    = 1'b1;
        NPCop = `NPC_AUIPC;
end
else begin
        WDsel = `WDSEL_ALU;
        PCWr    = 1'b0;
        NPCop = 2'b0;
end
end
     Load_WB : begin
RFWr    = 1'b1; //闇€鍐欑埞 rd
DMWr    = 1'b0;
PCWr    = 1'b0;
IRWr    = 1'b0;
ALUop = 5'b0;
IMEXTop=3'b0;
NPCop = 2'b0;
WDsel = `WDSEL_DM;
Asel    = 1'b0;
Bsel    = 1'b0;
WRbe    = 4'b0;
case(func3)
        `FUNCT_LB : DMEXTop = `DMEXT_LB;
        `FUNCT_LH : DMEXTop = `DMEXT_LH;
        `FUNCT_LW : DMEXTop = `DMEXT_LW;
        `FUNCT_LBU: DMEXTop = `DMEXT_LBU;
        `FUNCT_LHU: DMEXTop = `DMEXT_LHU;
endcase
```

```
                end
            endcase
        end
          endmodule
```

设计原理：

这个控制单元模块是一个 基于状态机（FSM）实现的多周期控制器，它配合其他部件（如寄存器堆、ALU、指令寄存器 IR、数据存储器 DM 等）共同完成 RISC-V 指令的译码、执行和控制过程。下面从几个核心角度说明其实现原理：

本模块采用 有限状态机（FSM）控制流程，通过当前状态和输入（如 opcode, func3, func7）来决定下一状态以及各个控制信号的输出，实现对每条 RISC-V 指令的多周期控制。

处理器每条指令会经历不同的状态（如：Fetch、DCD、EXE、Load、Store 等），控制单元根据当前状态设置对应控制信号，从而驱动数据通路完成指令执行。

状态机（FSM）原理：

控制器使用两个寄存器。

状态跳转逻辑如下（简化）：

```
if (rst)
    state <= Fetch;
else
    state <= nextstate;
```

各主要状态解释：

表 3.3　状态机各主要状态解释

状态名	功能
Fetch	取指：从指令存储器读取指令到 IR 寄存器
DCD	译码：识别指令类型并准备操作数
EXE	执行：ALU 执行运算、生成地址等
Load_store	地址计算：为 Load/Store 计算内存地址
Load	从数据存储器读数据
Store	向数据存储器写数据
Exe_WB	运算结果写回寄存器
Load_WB	Load 数据写回寄存器
JMP	跳转指令处理
Branch	分支条件计算
Branch_pc	若分支成立，写入新 PC 地址

指令类型识别与控制信号生成：

在 DCD 阶段，通过 opcode 来判断是哪类指令：

```
assign RType  = (opcode == INSTR_Rtype );
assign IType  = (opcode == INSTR_Itype_imm );
assign BrType  = (opcode == INSTR_Btype_branch );
assign JType  = (opcode == INSTR_Jtype_jal  );
assign LdType  = (opcode == INSTR_Itype_load );
assign StType  = (opcode == INSTR_Stype_store );
```

根据类型不同，状态机会进入不同路径，最终在每个状态中生成控制信号：

EXE 状态中，根据 func3、func7 决定 ALU 操作，如 add、sll 等；

Store 状态中， WRbe 控制写入 byte/half-word/word；

Load_WB 中， DMEXTop 控制符号扩展或零扩展；

JMP 中， WDsel 控制将 PC+4 写入 rd。

控制信号分时复用与输出：

每个状态仅激活与该状态功能对应的控制信号，其余保持默认值。这样可以：

降低控制冲突

避免错误写入

精简数据通路控制

例如：

取指时：开启 IRWr 和 PCWr

执行阶段：激活 ALU 控制，选择操作数

写回阶段：开启 RFWr、选择写回源

四、指令测试

4.1 I 型指令测试

测试指令：

```
addi x5，x0, 0
addi x6，x0，5
addi t2, t1, -1 # t2 = t1 - 1
```

Vivado 测试波形图如下：

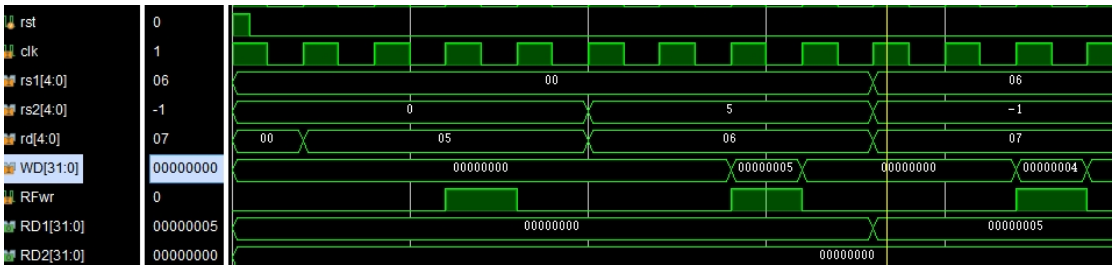

图 4.1 I 型指令 (addi) 测试波形图，寄存器 x5 和 x6 的写回结果

此处 WD 为写回寄存器的数据，rs1 和 rs2 是读源寄存器的地址，RD1 以及 RD2 是读出的数据，由上图可知指令运行正确，

测试指令：

```
lw t5, 0(t3)      # 读取当前元素到 t5
lw t6, 4(t3)      # 读取下一个元素到 t6
```

Vivado 测试波形图如下：

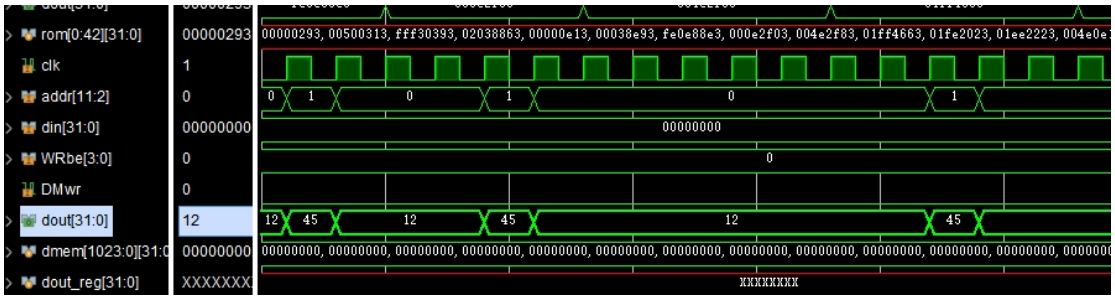

图 4.2 I 型指令 (lw) 测试波形图（1） 从数据存储器读取数据(dout)

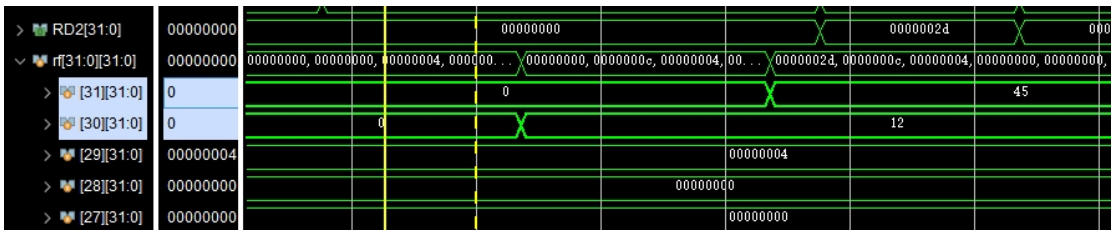

图 4.3 I 型指令 (lw) 测试波形图（2） 将数据写回寄存器

Dout 信号表示从数据存储器取出来的数据，addr 为读写的地址，在 mem 中读出的数据正确，第二张图为寄存器写入波形图，数据存储器取出的数据被正常写入寄存器。

4.2 B 型指令测试

测试指令：

beq t2,x0 done

t2 寄存器的值与寄存器 0 进行比较，为 0 则跳转。

Vivado 测试波形图如下：

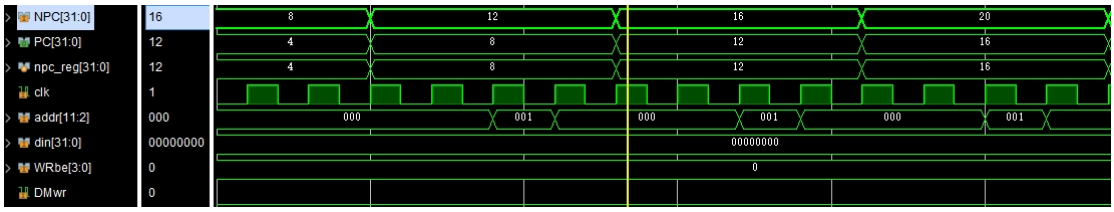

图 4. 4　B 型指令（beq）测试波形图，条件不满足时 PC 顺序加 4 的行为

由上图可知，PC 的地址一直为加 4，没有进行跳转，指令运行正确。

4.3 R 型指令验证

测试指令：

addi x5,x0,0x1

addi x6,x0,0x2

add x7,x6,x5

slt x8,x7,x5

sltu x9 ,x8,x5

Vivado 测试波形图如下：

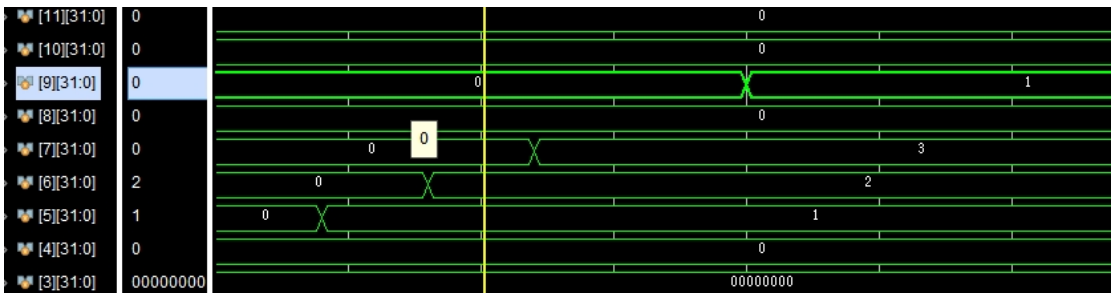

图 4. 5 R 型指令（add, slt）测试波形图，x7, x8, x9 寄存器的运算结果

在对 x5,6 进行寄存器的初始化以后 x7,8,9 寄存器的变化符合指令的

运行。

4.4 J 型指令验证

测试指令：

addi x5,x0,0x1

addi x6,x0,0x2

jal x0, 8

addi x6,x0,0x3

波形图如下：

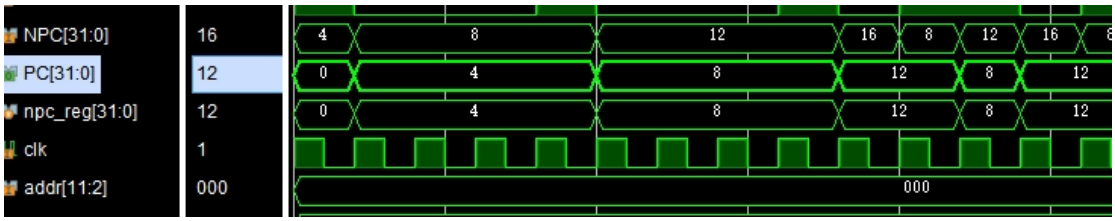

图 4. 6 J 型指令（jal）测试波形图，PC 执行无条件跳转的过程

在 PC 的执行过程中，PC 在执行到 12 以后会回到 8，正好符合 jal 的跳转地址，运行符合预期。

4.5 U 型指令

测试指令：

addi x5,x0,0x1

lui x5,0x3

波形图如下：

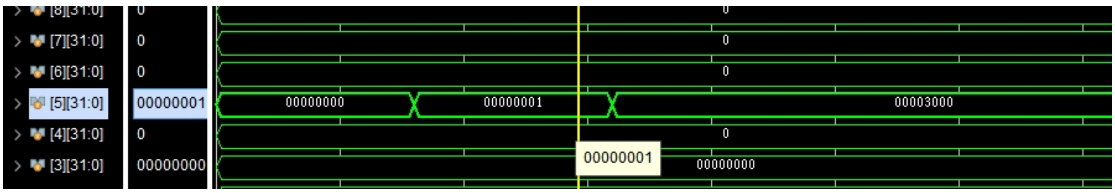

图 4. 7　U 型指令（lui）测试波形图，立即数高位加载到 x5 寄存器的结果

在对 x5 寄存器初始化以后执行 lui 指令，执行结果符合预期。

4.6 S 型指令验证

测试指令：

```
sw t6, 0(t3)
sw t5, 4(t3)
```

波形图如下：

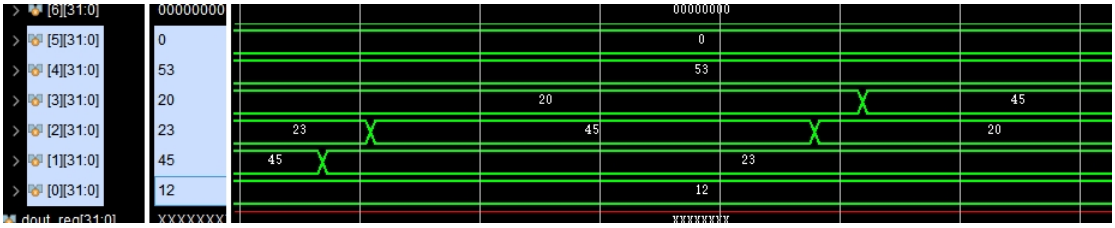

图 4.8　S 型指令（sw）测试波形图，待写入存储器的源寄存器 t5、t6 的值

可以看见寄存器中的数字被正确写入数据存储器。

4.7 总体验证

验证指令：

```
_start:
    addi x5，x0, 0          #   x5 存放数组的起始地址（RAM 起始地址）
                                    # x6 存放数组长度的地址
   addi x6，x0，5
outer_loop:
    addi t2, t1, -1 # t2 = t1 - 1 (内层循环次数)
    beqz t2, done    # 如果 t2 == 0，结束排序
    li t3, 0x0          # t3 用于存储当前元素地址（重置为数组开始）
inner_loop:
    addi t4, t2, 0   # t4 = t2 (循环计数器)
    beqz t4, outer_loop # 如果 t4 == 0，结束内层循环，进入下一轮外层循环
```

```
        lw t5, 0(t3)       # 读取当前元素到 t5
        lw t6, 4(t3)       # 读取下一个元素到 t6
        blt t5, t6, no_swap # 如果 t5 < t6，不交换
        # 交换元素
        sw t6, 0(t3)       # 将 t6 存入当前位置
        sw t5, 4(t3)       # 将 t5 存入下一位置
no_swap:
        addi t3, t3, 4    # t3 指向下一对元素
        addi t2, t2, -1 # 内层循环计数器 - 1
        j inner_loop       # 跳转到内层循环
done:
        # 排序完成后进入死循环
        j done
```

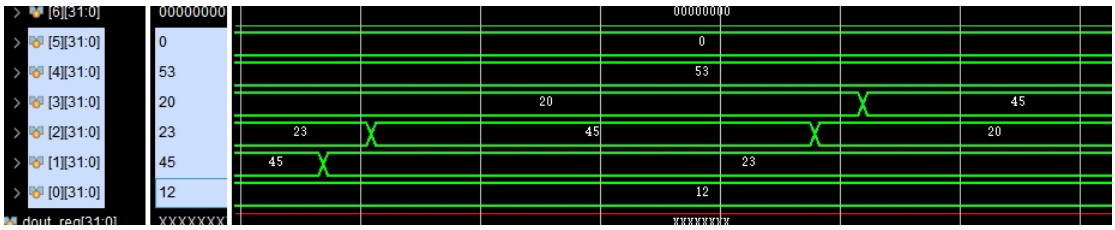

图 4.9　排序算法验证波形图（排序前）

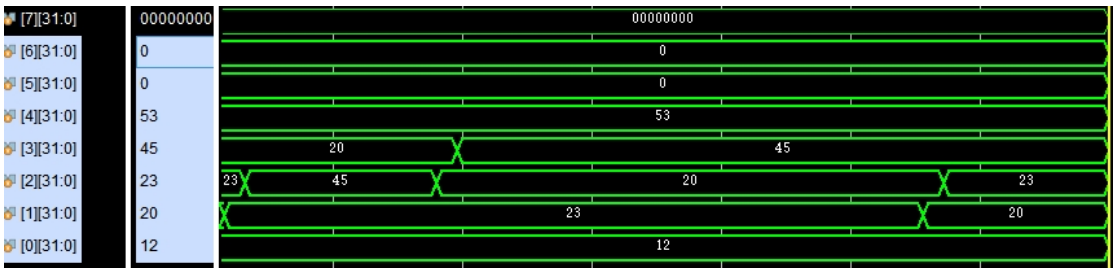

图 4. 10 排序算法验证波形图（排序后）

由波形图可见，对于五个乱序的数字，在经过排序程序以后，数字变为有序数字，符合预期，验证结果正确。

五、结论

在本次计算机组成原理实验中，我们小组通过紧密协作与不懈努力，成功设计并实现了一个能够正确执行 RISC-V 32I 指令集的多周期 CPU。从最初的理论学习，到数据通路的设计搭建，再到控制单元的精细调试，我们完整地走过了将一个处理器从概念蓝图变为可执行硬件逻辑的全过程。

整个项目的核心挑战在于设计一个精确、稳健的有限状态机（FSM）控制单元。如何为 R、I、S、B、U、J 等不同类型的指令，在取指、译码、执行、访存、写回等不同阶段，生成准确无误的控制信号，是我们投入最多心血的地方。我们通过反复的逻辑推演、代码实现以及细致的波形仿真分析，逐一攻克了分支预测、访存时序以及数据依赖等难题，最终确保了数据通路的稳定运行。

当看到我们编写的排序算法在模拟器中成功运行，将一组乱序的数字正确排列时，我们深刻体会到了理论与实践相结合带来的巨大成就感。我们的处理器最终成功通过了所有预设指令的仿真测试，验证了设计的正确性与可靠性。

通过这次实践，我们对 CPU 内部的微观世界有了前所未有的直观认识。我们不再仅仅从课本上理解“多周期”的概念，而是亲手体会了它如何通过复用 ALU 等核心部件来优化硬件资源，以及它相比单周期设计在时钟频率上的优势。同时，这次设计也让我们清晰地看到了多周期 CPU 与更高效的流水线 CPU 在性能上的差异，激发了我们

对更高级处理器架构深入探索的浓厚兴趣。

总而言之，这次实验不仅是一次课程任务的完成，更是一次宝贵的工程实践。它不仅锻炼了我们的 Verilog HDL 编程能力和硬件调试技巧，更重要的是，它将抽象的计算机体系结构知识内化为了我们解决问题的能力与信心。这段从零到一构建 CPU 的经历，无疑为我们未来的学习和探索之路奠定了坚实的基础。