实验六 CPU 综合设计

一、实验目的

1 掌握复杂系统设计方法。
2 深刻理解计算机系统硬件原理。

二、实验内容

1）设计一个基于 MIPS 指令集的 CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；
2）CPU 需要包含寄存器组、RAM 模块、ALU 模块、指令译码模块；
3）该 CPU 能运行基本的汇编指令；（D~C+）
以下为可选内容：
4）实现多周期 CPU（B-~B+）；
5）实现以下高级功能之一（A-~A+）：
(1)实现 5 级流水线 CPU；
(2)实现超标量；
(3)实现 4 路组相联缓存；
可基于 RISC V 、ARM 指令集实现。

三、实验要求

编写相应测试程序，完成所有指令测试。

四、实验代码及结果

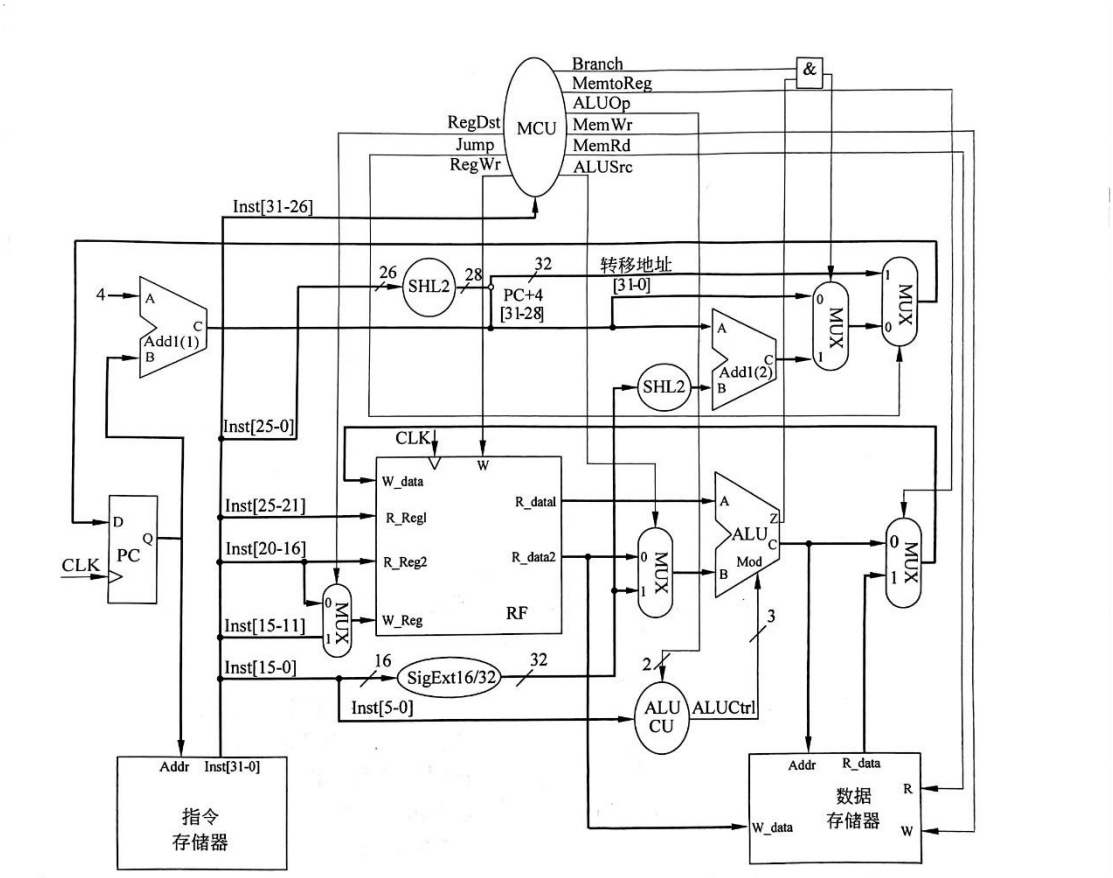
（1）指令说明

本次实验用到的指令解释及其操作码
R-Type 指令：
1 add rd, rs, rt：将 rs 和 rt 寄存器中的值相加，并将结果存储在 rd 寄存器中。
操作码：000000
功能码：20
2 sub rd, rs, rt：将 rs 寄存器中的值减去 rt 寄存器中的值，并将结果存储在 rd 寄存器中。
操作码：000000
功能码：22
I-Type 指令：
1 addi rt, rs, immediate：将 rs 寄存器中的值与立即数相加，并将结果存储在 rt 寄存器中。
操作码：001000
2 lw rt, offset(rs)：将地址为 rs + offset 的内存中的值加载到 rt 寄存器中。
操作码：100011
3 sw rt, offset(rs)：将 rt 寄存器中的值存储到地址为 rs + offset 的内存中。

操作码：101011
4 beq rs, rt, offset：如果 rs 和 rt 寄存器中的值相等，则跳转到当前指令地址加上 offset 的地址。
操作码：000100
J-Type 指令：
j target：无条件跳转到目标地址。
操作码：000010
NOP 指令：
nop：空指令，不执行任何操作。
操作码：000000

（2）CPU 说明

该多周期 CPU 相对比较简单，其数据通路参见下图：



CPU 控制信号表见下：

信号名称	含 义	信号名称	含 义
CLK	时钟信号	MemtoReg	存储器至寄存器选择
RegWr	寄存器写	RegDst	目的寄存器选择
MemRd	存储器读	ALUOp	ALU 操作类型
MemWr	存储器写	ALUctrl	ALU操作控制
ALUSrc	ALU数据源选择	Jump	无条件转移
PCSrc	PC数据源选择	Branch	条件转移

（3）CPU 实现

CPU 模块（主模块）代码如下

```
module CPU(
    input wire clk,    // 时钟
    input wire rst,    // 复位
    input  [31:0] instruction1,  // 输入指令
    output reg [31:0] result  // 输出结果
);

reg [31:0] registers [0:31];
integer i;
initial begin
    for(i=0;i<32;i = i + 1)begin
        registers[i] = 32'b0;
    end
    result = 32'b0;
 end

wire MemToReg, MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump,Zero;
wire [4:0] ALUControl;
reg [5:0] opcode;
reg [4:0] rs, rt, rd;
wire [31:0] immediate;
wire [2:0] state;
reg [31:0] pc;
wire CF;
wire [31:0] imm_ext;
wire [31:0] Alu_in;
wire [31:0] new_result;
wire [31:0] Dm_out;
    always @(posedge clk or posedge rst) begin
        //pc_b <= pc;
        if(rst) begin
            pc <= 32'b0;
            for ( i = 0; i < 32; i = i + 1) begin
                registers[i] <= 32'b0;
            end
         end
        else if (state == 3'b110)begin
            if(Jump == 1) pc <={ pc[31:28],instruction1[26:0]};
            else if(Branch == 1 && new_result == 0) pc <= pc + imm_ext;
            else  pc <= pc +4;
        end
```

```
    end
    // 立即数生成
assign immediate = instruction1[15:0];
// 控制信号生成
always @* begin
    // 从指令中提取字段
    opcode = instruction1[31:26];
    rs = instruction1[25:21];
    rt = instruction1[20:16];
    rd = instruction1[15:11];
end

StateMachine sm(clk,rst,state);
IM Im(.clk(clk),.rst(rst),.state(state),.Addr(pc),.instruction(instruction1));
Controller cu(clk,state,instruction1,MemToReg, MemWrite,
Branch,ALUSrc,RegDst,RegWrite,Jump,ALUControl);
SigExt sigext(immediate,ALUSrc,imm_ext);
MUX2 mux2_alu(ALUSrc,registers[rt],imm_ext,Alu_in);
ALU alu(.F(new_result),.CF(CF),.A(registers[rs]),.B(Alu_in),.OP(ALUControl));
DataMemory dm(clk,state,new_result,registers[rt],MemToReg,MemWrite,Dm_out);

always @(posedge clk) begin
    if(state == 3'b110)begin
    if (RegWrite && new_result)  begin
        // 写入寄存器
        if(opcode != 6'b000000)
        registers[rt] <= MemToReg ? Dm_out :new_result;
        else
        registers[rd] <= MemToReg ? Dm_out : new_result;
    end
    end
end
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // 复位时初始化
        result <= 32'b0;
    end else begin
        // 输出结果
        result <= MemToReg ? Dm_out :new_result;
    end
end
endmodule
```

该 CPU 利用 StateMachine 模块修改状态机，IM 模块取指，Controller 模块译码，

SigExt 模块作符号拓展，MUX2 模块作二路选择，ALU 模计算，DataMemory 模块作内存读取存储。
此外，其中 always 块完成了 PC 更新（含跳转、复位）、指令提取、写回寄存器的工作。
由于 Controller 和 ALU 已在以前的实验中给出，这里不再给出其代码。

StateMachine 模块代码如下：

```
module StateMachine (
    input wire clk,
    input wire rst,
    output reg [2:0] state
);

// 定义状态
parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b100;
parameter S4 = 3'b110;

// 状态变量
reg [2:0] next_state;

// 启动状态
initial begin
    state <= S0;
end

// 状态变化逻辑
always @(posedge clk) begin
    case (state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S4;
        S4: next_state = S0;
        default: next_state = S0;
    endcase
end
// 更新状态
always @(posedge clk) begin
    state <= next_state;
end
```

```
always @(negedge rst) begin
    state <= S0;
end
endmodule
```

StateMachine 根据时钟信号更新状态，指导 CPU 工作。

IM 模块代码如下：

```
module IM (
    input clk,
    input rst,
    input [2:0] state,
    input wire [31:0] Addr,
    output reg [31:0] instruction
);

reg [31:0] RAM[0:1023];

initial begin
    $readmemh("C:/Users/Jiefucious/project_4/instructions.txt", RAM);
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        instruction <= 32'h0; // Reset value
    end else if(state == 3'b000) begin
        instruction <= RAM[Addr/4];
    end
end
endmodule
```

IM 模块在状态 0 时取指令。

SigExt 模块代码如下

```
module SigExt(input [15:0] original,
    input Alusrc,    // 0: Zero-extend; 1: Sign-extend.
    output reg [31:0] extended
    );

    always @(*) begin
        extended[15:0] <= original;
        if(Alusrc == 0) begin
            extended[31:16] <= 0;
        end
        else begin
```

```
            if(original[15] == 0) extended[31:16] <= 0;
            else extended[31:16] <= 16'hFFFF;
        end
    end

endmodule
```

SigExt 模块对立即数作字符拓展。

MUX2 模块代码如下

```
module MUX2(
    input choice,
    input [31:0] in0,
    input [31:0] in1,
    output reg[31:0] out
    );
    always @* begin
     out = (choice == 0) ? in0 : in1;
    end
endmodule
```

这是一个二路选择器，在 CPU 中根据控制信号选择 ALU 输入。

DataMemory 模块如下

```
module DataMemory(
    input clk,
    input [2:0]state,
    input [31:0] DAddr,
    input [31:0] DataIn,
    input RD,    // 读控制，1 有效
    input WR,    // 写控制，1 有效
    output [31:0] DataOut    // 读出 32 位数据
    );
    reg [31:0] RAM[0:1023];
    assign DataOut[31:0]   = (RD==1) ? RAM[DAddr] : 32'bz;
    always @(posedge clk) begin
        if(WR == 1 && state == 3'b100) begin
            RAM[DAddr] <= DataIn[31:0];
        end
    end
endmodule
```

其在存数/取数阶段工作，读出内存或通过 ALU 结果将数据存储到内存中。

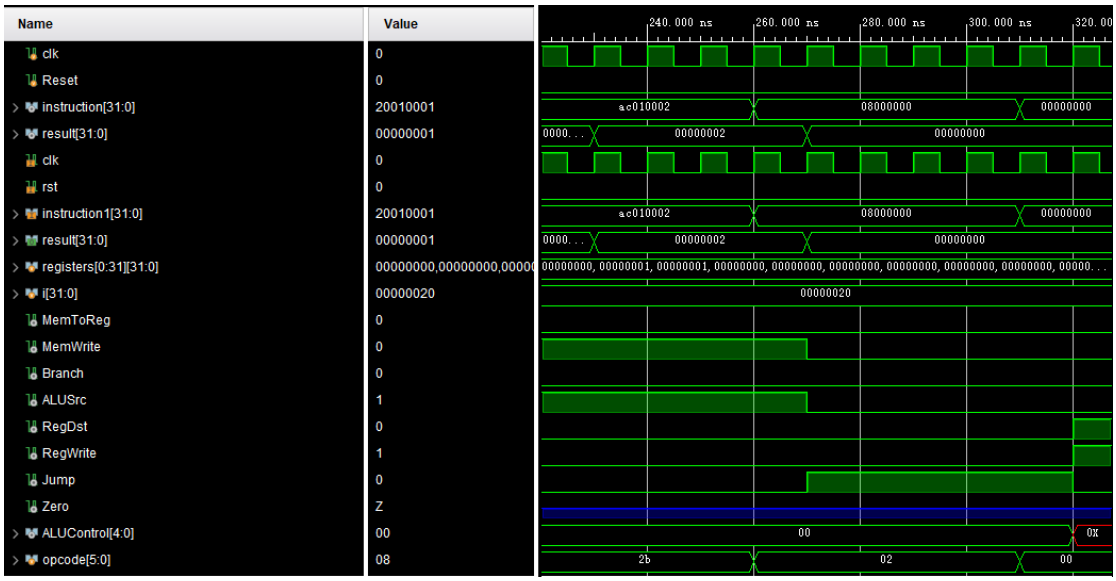
（4）仿真结果

对所要求的每条指令单独分析，发现无误。下面选择有代表性的几条指令一起进

行仿真，其涵盖了对写寄存器、写内存以及跳转。

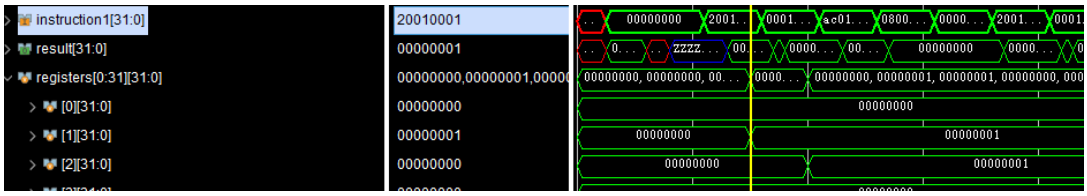
```
addi rt, rs, 1
add rd, rs, rt
sw rt, offset(rs)
j 0
```

结果如下图

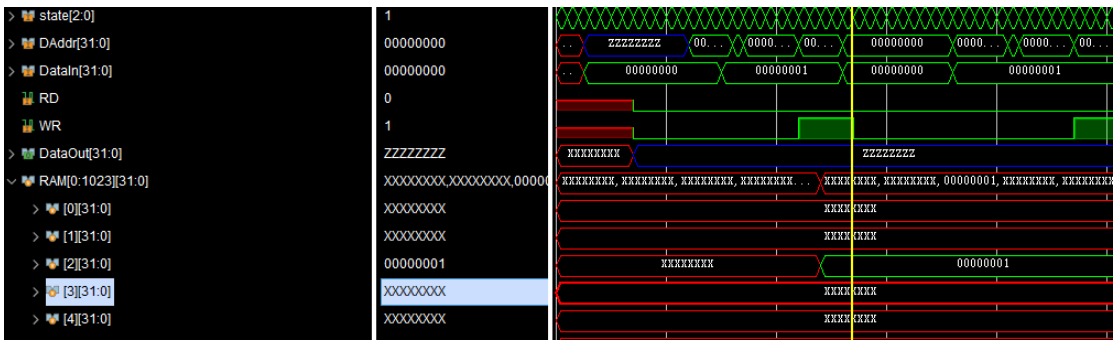


下面作进一步分析：

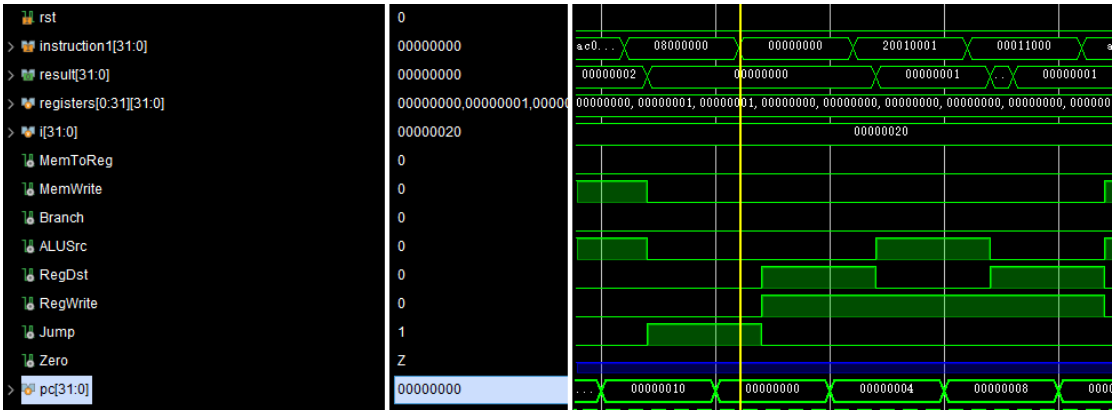
第一条指令会将寄存器 1 赋值为 1，第二条指令时二号寄存器也被赋值为 1。



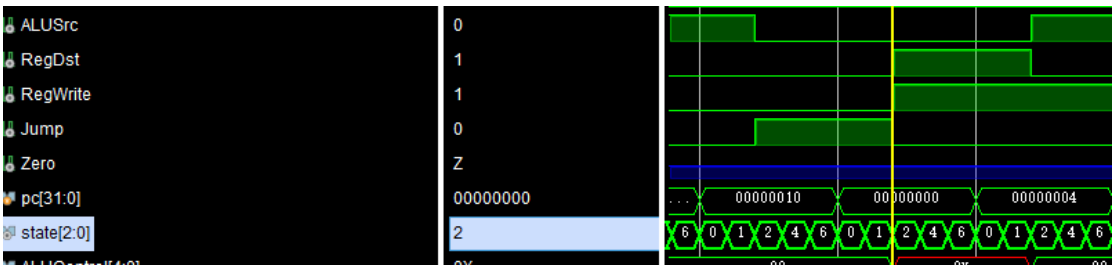
第三条指令会把 1 写入内存为 2 处。



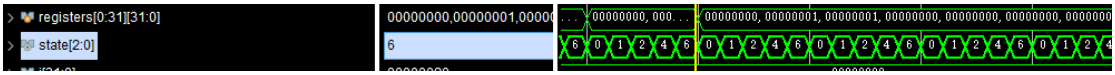
Jump 指令将 pc 跳转回 0。



多周期体现见下图：



时钟更新，状态更新，随之指令译码。类似地，内存读写会在状态值为 4 时进行，寄存器写回和 pc 更新在状态值为 6 时进行。



五、调试和心得体会

在调试中出现了以下问题：

（1）传参出错。需要注意 reg 类型数据应该在 always 块中赋值。

（2）时钟控制错误。在多周期 CPU 设计中，明确模块在时钟下如何工作是非常重要的，模块应该在时钟的控制下有序地运转。

（3）数据选择问题。需要注意，有时候向 ALU 中传入的是寄存器值，有时候是立即数，这时需要用数据选择器。另外，写回寄存器也需注意是写回 rd 寄存器还是 rt 寄存器。

心得体会：

（1）有助于帮助我理解计算机组成原理。在开始实现多周期 CPU 之前，深入理解计算机组成的基本概念是至关重要的。这包括指令集体系结构和基本的运算逻辑。

（2）提高我对 CPU 的理解。多周期 CPU 通常将指令执行划分为若干个阶段，每个阶段执行特定的操作。这些阶段通常包括取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）等。每个阶段的逻辑可以独立设计和优化。

（3）提高硬件描述语言熟练程度。