东南大学

《数字逻辑与计算机体系结构(含实验)II》

课程设计报告

MIPS原型处理器体系结构框架设计

姓　　名：李睿刚　　学　　号：61821326

同　　组：常兆骏　　学　　号：61821328

同　　组：刘嘉乐　　学　　号：61821304

专　　业：吴健雄学院　　实 验 室：计算机硬件技术

实验时间：**2023** 年 **5** 月 **16** 日　　报告时间：**2022** 年 **6** 月 **2** 日

评定成绩：　　审阅教师：

一. 简介

目的

根据课上所学，实现 MIPS 原型处理器体系结构框架设计。

基本要求

1. 根据实际需要和设计实验及编程条件，选择 8 位或 32 位 MIPS 原型机及基本指令集；
2. 选择熟悉的 FPGA 设计开发工具（如 Altera 的 Quartus II 或 Xilinx Vivado）及开源资源；
3. 参考 MIPS 体系结构及 HDL 资料资料，设计 MIPS 处理器原型机的微体系结构并搭建、编程、调试基于 FPGA 的原型 CPU 系统；
4. 根据系统结构，完成 1-3 种功能指令的扩展实现；
5. 针对简单应用，完成程序指令序列机器码汇编生成，并在设计的原型 CPU 上调试（演示）；
6. 演示并介绍开发设计及调试实现方法，并在书面报告上总结；

具体工作

1. 实现或分析其中两条指令的扩展设计方法；
2. 基于机器指令编写程序；
3. 通过仿真时序图对 CPU 的执行进行跟踪验证；
4. 下载后可实际运行（以仿真为主，目标代码可留用运行）；
5. 【提高要求】提出运行陷阱（Trap）机制的实现原理方案，分析不同的设计方法（LPM、IP 核和自编模块）的可扩展性。

完成情况

选择在 vivado 平台上使用 system Verilog 完成 32 位 MIPS 原型机，支持如下 19 条指令：

R 类型：add、sub、and、or、nor、xor、sll、srl、sra、jr

I 类型：addi、andi、ori、xori、lw、sw、beq

J 类型：j、jal

理论上可以支持更多的运算方式（ALU 支持，且指令格式相同），比如 slti、sltu、addiu、multu、multui 等，但设计控制模块时的参照仅有最基础的指令，后来才发现还有这些指令。没有设计 lui 的通路。没有设计自己的指令。没有利用给出的资源，所有代码**都是**自己**从零编写**。

二. 基本实验原理

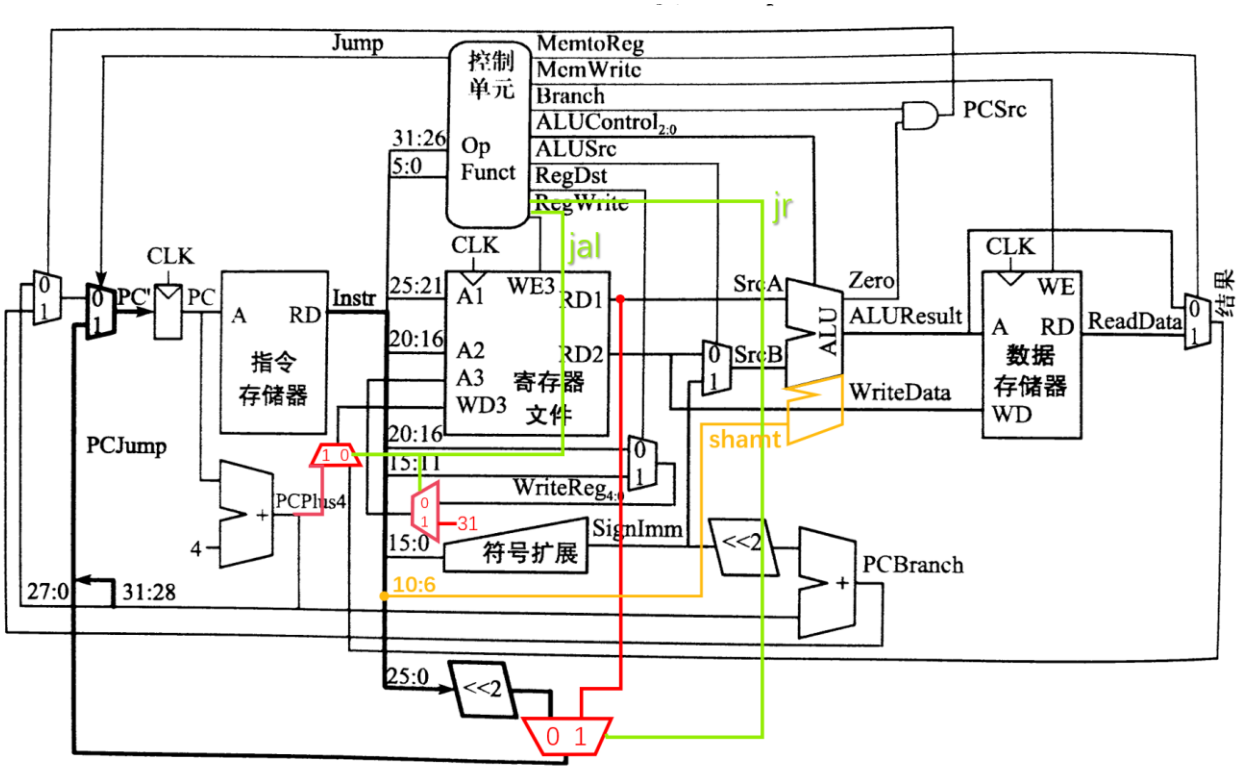
有 7 个子模块：Rom（指令）、Ram（数据存储）、RegisterFile（32 个 32 位寄存器）、PC（更新 PC）、Control（控制信号）、ALU（运算）、符号拓展模块（26 位有符号数拓展到 32 位有符号数）。

有一个整合模块 MIPS（完成数据通路的连接）。

基本框架基于书本，为主要为“移位”、“跳转”功能增加了通路，如下图所示。

移位操作与一般的 R 指令不同，其操作数直接存放于 shamt 字段，且与 rs 字段位置不重合。因此，给 ALU 增加了一个输入，专门用于 shamt 的移位操作，避免了在 SrcA 或 SrcB 上使用复用器选择数据，避免了新增一个控制信号。在 ALU 内部，相关移位操作都不涉及 SrcA,都是 SrcB 和 shamt 的运算。【值得注意的是，这样的设计下，对 sltv 等移位位数来自寄存器的指令的通路设计不友好。如果要实现这些指令，还是得复用 SrcA，或者给 ALU 增加新的运算方式。】

跳转操作主要集中在 PC 的更新上。Jr 的 PC 来自寄存器，需要复用器选择。Jal 比 j 多了存储当前地址，需要在寄存器存储通路上新增寄存器。因此增加了两个控制信号。



数据通路

三. 程序流程和编程实现

如下是所有控制信号的“真值表”：

指令	Branch	Jr	Jump	Jal	RegWE	RegDst	ALUsrcB	ALUOp	MemWE	MemReg
描述	Beq 用	Jr 用	J 和 jal 用	Jal 用	寄存器写使能	寄存器写地址	ALU 操作数	ALU 运算类型	存储器写使能	存储器到寄存器
Beq	1	X	0	X	0	X	0	-	0	X
J	X	0	1	X	0	X	X	X	0	X
Jal	X	0	1	1	1	X	X	X	0	X
Jr	X	1	1	X	0	X	X	X	0	X
[R]运算	0	X	0	0	1	1	0	?	0	0
[i]运算	0	X	0	0	1	0	1	?	0	0
[R]移位	0	X	0	0	1	1	0	?	0	0
Lw	0	X	0	0	1	0	1	+	0	1
Sw	0	X	0	0	0	X	1	+	1	X

其中，“X”表示任意值，“？”表示根据具体指令定。

使用 verilog 不需要设计传统的有限状态机。使用 case 语法，按照 rom 的实现思路即可完成 op、funct 到控制信号的映射。实际综合出来的电路内部就是 rom 和复用器。

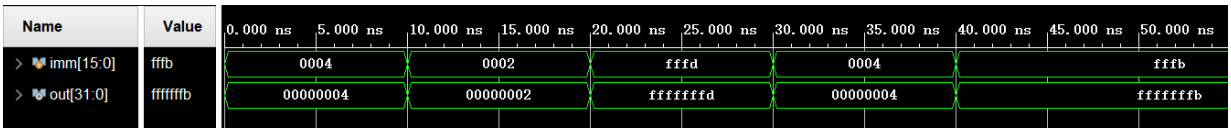
四. 测试方案和测试结果

(一) 符号拓展测试

Testbench.sv

```
module TEST;
    reg [15:0] imm = 4;
    wire [31:0] out;
    SignExtend se(imm, out);
    initial begin
        #10 imm = 2;
        #10 imm = -3;
        #10 imm = 4;
        #10 imm = -5;
    end
endmodule
```

仿真结果：



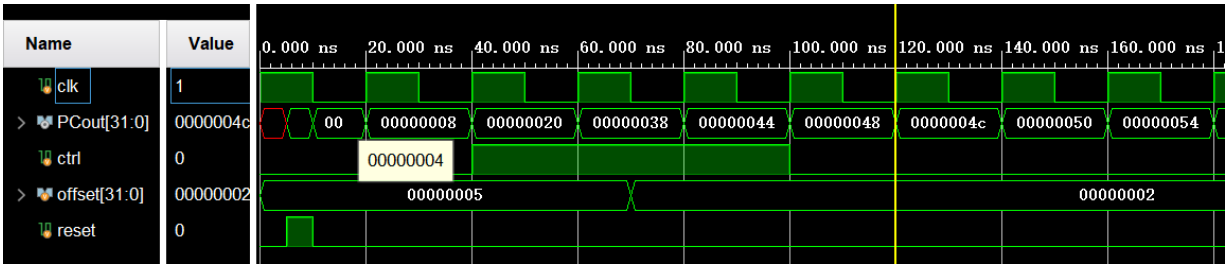
对于正数 4、2，符号拓展后依旧是整数（拓展了 0）；对于负数-3、-5，符号拓展后依旧是负数（拓展了 1），模块功能正常。

(二) PC 测试（在整机测试前的测试。后来因为时序问题，对模块做了修改。）

Testbench.sv

```
module TEST;
    reg clk = 1;
    always begin
        #10 clk = ~clk;
    end
    wire[31:0] PCout;
    reg ctrl = 0;
    reg [31:0] offset = 5;
    reg reset = 0;
    PC pc(clk,offset,ctrl,reset,PCout);
    initial begin
        #5 reset = 1;
        #5 reset = 0;
        #30 ctrl = 1;
        #30 offset = 2;
        #30 ctrl = 0;
    end
endmodule
```

仿真结果：



在 ctrl=0 的时候 PC 每次+4，而在 ctrl=1 时，PC=PC+4+offset*4（模拟 beq），模块功能正常。

(三) 整机（指令）测试

写了个 python 脚本将汇编转为机器码。每次需要点入 rom Ip 重新选择 coe 文件才能完成 rom 的更新，而直接修改 rom.coe 文件不行。以下测试了几乎所有指令。

1. ADDI、LW、SW、ADD

通过 addi 写寄存器，保存内容，再读取到另一个寄存器，最后进行求和。

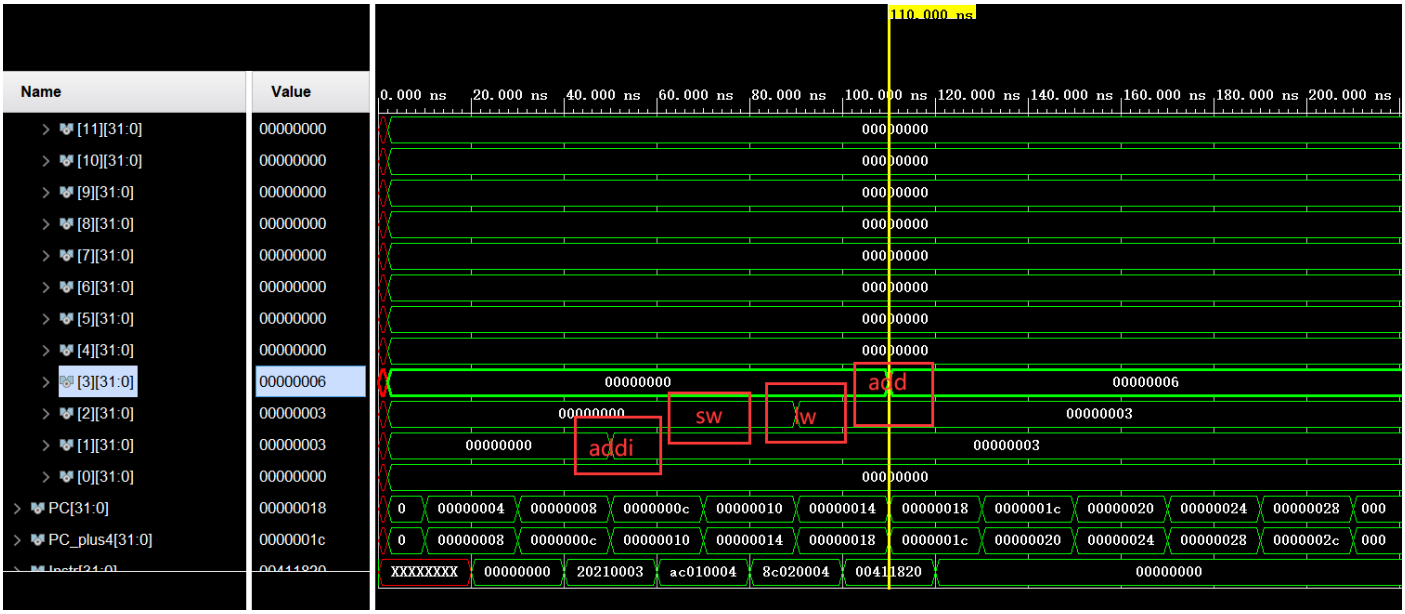
汇编：

```
addi $1, $1, 3
sw $1, 4($0)
lw $2, 4($0)
add $3, $2, $1
```

机器码（rom.coe 内容）：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000000,
00000000000000000000000000000000,
00100000001000010000000000000011,
10101100000000010000000000000100,
10001100000000010000000000000100,
00000000010000010001100000100000
```

仿真结果：



数据从$1 到数据存储器到$2，成功。

2. BEQ、J

编写了一个 for 循环。测试时发现指令总是取到 j 指令后一条，而 PC 已经跳转回去。排查原因后发现 PC 和 Instr 错位，导致 J 使 PC 更新晚 PC 自己更新一步。所以把 PC 的 D 触发器改为时钟下降沿触发，解决了问题。

汇编：

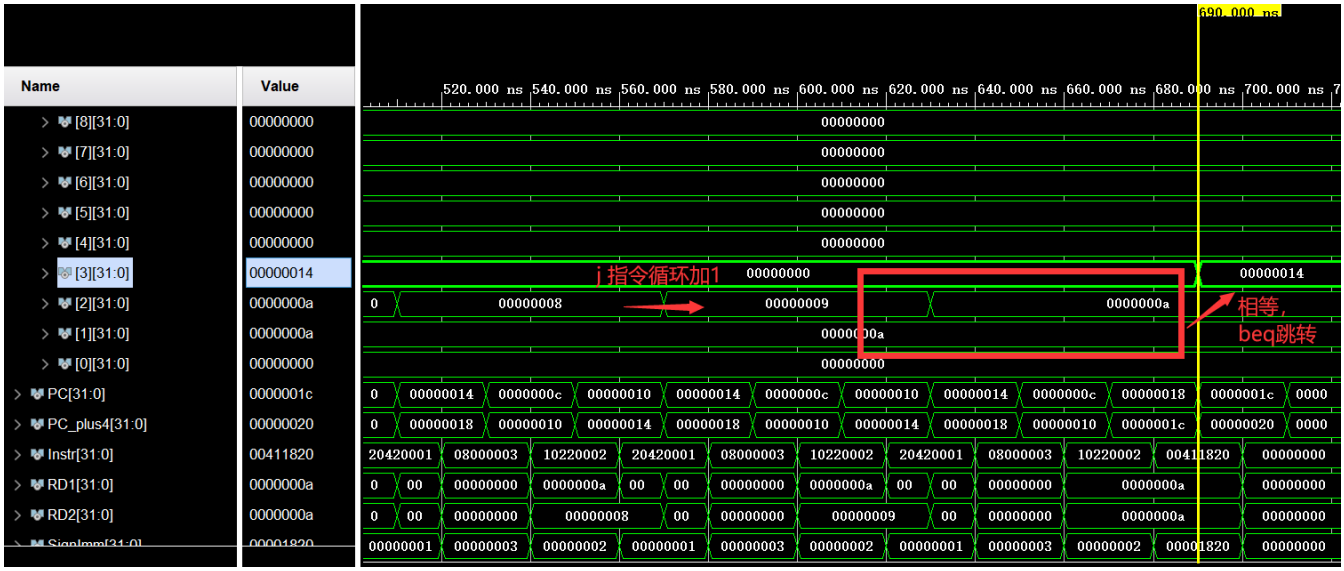
```
addi $1,$0,10
```

```
beq $1,$2,2
addi $2,$2,1
j 3（到 3 是因为前面留了两行 nop）
add $3,$2,$1
```

机器码（rom.coe 内容）：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000000,
00000000000000000000000000000000,
00100000000000010000000000001010,
00010000001000100000000000000010,
00100000010000100000000000000001,
00001000000000000000000000000011,
00000000010000010001100000100000
```

仿真结果：



3. SUB（正负数）、AND、OR、NOR、XOR

汇编：

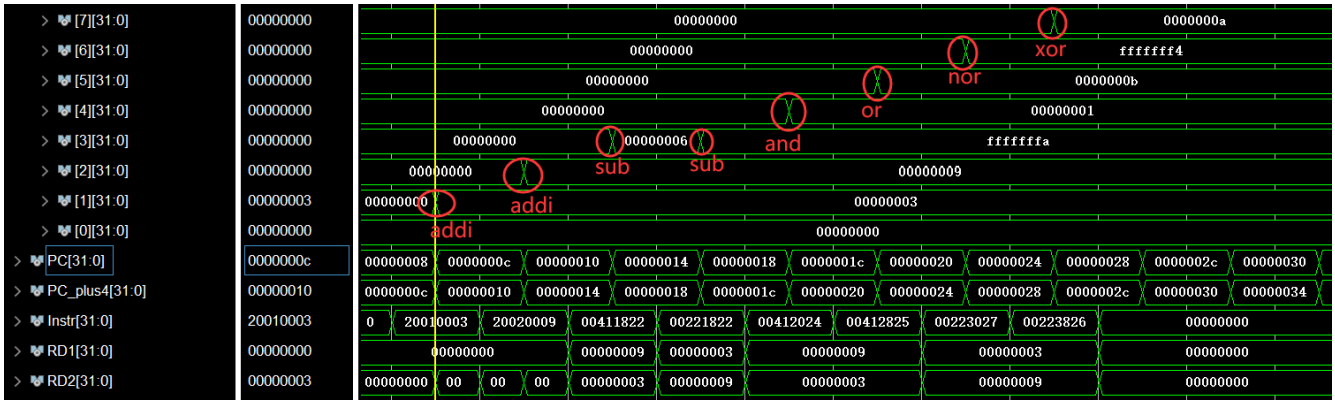
```
addi $1,$0,3
addi $2,$0,9
sub $3,$2,$1
sub $3,$1,$2
and $4,$2,$1
or $5,$2,$1
nor $6,$1,$2
xor $7,$1,$2
```

机器码（rom.coe 内容）：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000000,
00000000000000000000000000000000,
```

```
00100000000000010000000000000011,
00100000000000010000000000001001,
00000000010000010001100000100010,
00000000001000010000110000100010,
00000000010000010010000000100100,
00000000010000010010100000100101,
00000000001000100011000000100111,
00000000001000100011100000100110
```

仿真结果：



4. SLL、SRL、SRA、ORI、ANDI、XORI、JAL

汇编：

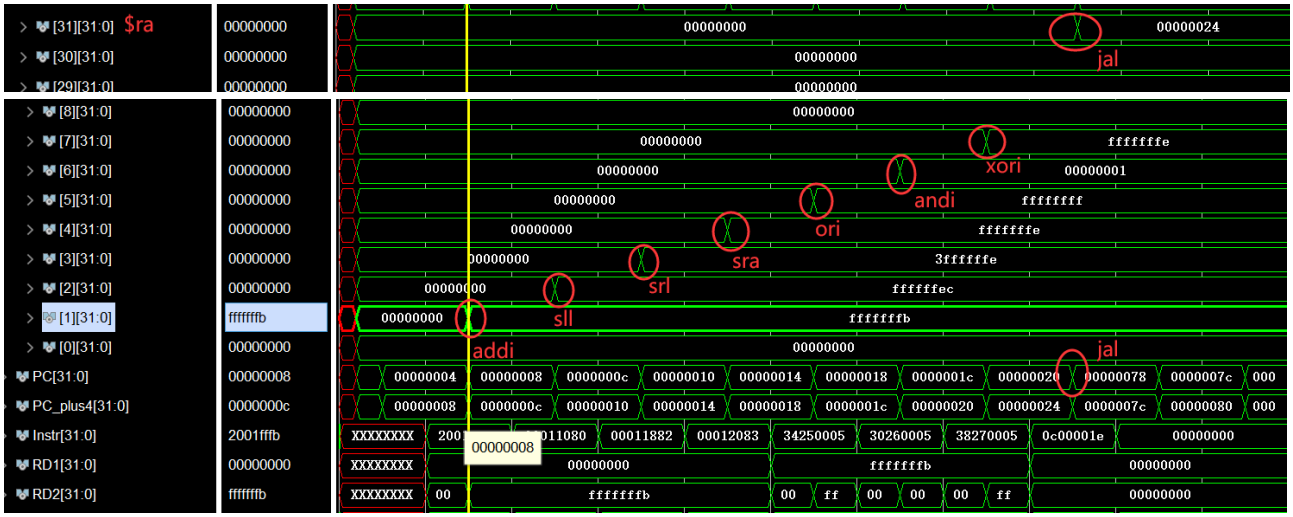
```
addi $1,$0,-5
sll $2,$1,2
srl $3,$1,2
sra $4,$1,2
ori $5,$1,5
andi $6,$1,5
xori $7,$1,5
jal 30
```

机器码（rom.coe 内容）：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000000,
00100000000000011111111111111011,
00000000000000001000100001000000,
00000000000000001000110001000010,
00000000000000001001000001000011,
00110100001001010000000000000101,
00110000001001100000000000000101,
00111000001001110000000000000101,
00001100000000000000000000011110,
```

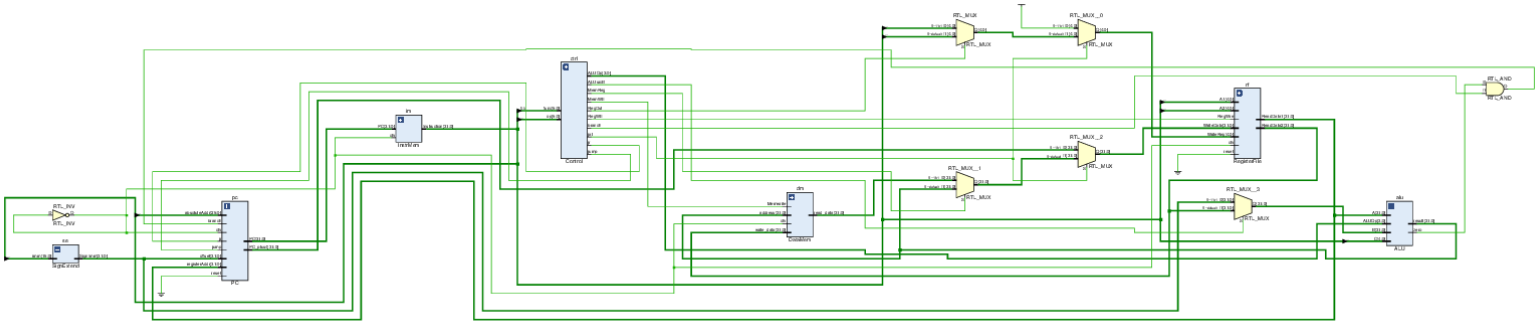
仿真结果：

特意使用了负数，以观察逻辑右移和算数右移的区别。经过验算，结果无误。Jal 指令成功改变 PC，并且保存了当前的位置。



五．系统运行结果与存在问题分析

下图是综合出来的电路，报告中看不清楚。说明所写 cpu 是可以物理实现的。展开每个模块，基本都是一大堆连线。用 verilog 避免了处理这些细枝末节的电路，极大方便了设计。这也是我们不选择用 Quartus 搭积木一样制作 cpu 的主要理由。



存在的问题：首先是如上文所说 ALU 的设计。其次，关于时序还是有些担忧。从 DataMemory 读取数据是下降沿完成，而寄存器保存也是下降沿完成，两个中间又有复用器，实际实现的时候可能会有竞争问题。单周期解决这个或许比较困难，因为只有上升沿和下降沿可以用。如果采用多周期流水线实现就可以避免这个潜在的问题。

六．分析与总结认识

刘嘉乐同学提供了宝贵的资料，常兆骏同学帮我完成了控制模块的编写，其余是我用了一整天完成的。俩组员说没空，我试着写了写，就全写出来了。测试的机器码、改 bug 也是我完成的。时间仓促，仅仅对书上的结构做出了些许修改，增加了我认为重要的指令。难点在于理清书上每一条线构建的目的，但只要过一遍，整个架构都会清晰。在当前指令集下，ALU 实现移位部分可以说是个创新点（上文已论述）。设计上几乎没有难点，实操难点集中在语法和 IP 核的使用。编写过程中，由于 verilog 语法限制，而升级为 system Verilog；不知道 IP 核接口以及作用，用了好些时间找资料。与书上不同，rom 的 IP 读取指令也需要时钟，导致时序的设计出现了 bug（第四部分提到的那个）。

值得一提的是，为了模拟真实环境，我没有用 init 初始化模块（比如 PC 和寄存器）的数据，而是设

置了 reset 端口，模拟开机清零。

原计划封装整个 cpu，并设计专门的显示指令，起到类似“显示器”的作用。但因为时间因素而放弃了。目前的 cpu 仅仅实现了运算，但离使用还差很远，如果继续深入，向外输出（如用 led 或数码管）必不可少（还有接收来自外部的数据）。要实现这点，可以借用 R 指令格式，引出一条类似总线的线，接在寄存器输出 1 后。总线可以读取到寄存器的结果，也可以向 ALU 的 srcA 传输信号。

七．系统应用技术与加强基础展望

没用 Quartus 是因为那样的设计太底层。Vivado 这样的模式很符合我对现代大系统设计的想象，top-down，设计者只需要把握整体的架构，而无需在意具体的硬件连接。以代码形式展现，也能适应以后越写越大的规模。迭代升级也很方便。

本次实验还写了汇编转机器码的脚本，虽然是简单的映射，但是有了写编译器的感觉。

这学期在学习 STM32 编程，每一个配置都需要关注底层，尤其是数据通路。感觉和本次实验有互补之妙，一个强调顶层指导底层，一个强调底层实现顶层，都体现了软硬结合的重要性。

八．代码

IP（rom 和 ram）的调用参考了 https://comp2008.gitee.io/lab2/mem/

项目托管于 https://github.com/madderscientist/codeRoad/tree/main/MIPS_cpu

ALU.sv

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps

module ALU(
    input [3:0] ALUOp,          // ALU 操作控制
    input [31:0] A,             // 输入 1
    input [31:0] B,             // 输入 2
    input [4:0]  C,             // 输入 3（移位）
    output reg zero,            // 运算结果 result 的标志，result 为 0 输出 1，否则输出 0
    output reg [31:0] result     // ALU 运算结果
    );
    always@(*)
     begin
        case (ALUOp)
            4'b0000: result = B << C;   // 逻辑左移
            4'b0001: result = $signed(B) >>> C;   // 算数右移
            4'b0010: result = B >> C;   // 逻辑右移
            4'b0011: result = A * B;    // 无符号乘法
            4'b0100: result = A / B;    // 无符号除法
            4'b0101: result = A + B;    // 无符号加法
            4'b0110: result = A - B;    // 无符号减法
            4'b0111: result = A & B;    // 按位与
```

```
            4'b1000: result = A | B;    // 按位或
            4'b1001: result = A ^ B;    // 按位异或
            4'b1010: result = ~(A | B); // 按位或非
            4'b1011: result = ($signed(A) < $signed(B)) ? 1 : 0; // 有符号比较
            4'b1100: result = (A < B) ? 1 : 0; // 无符号比较
            default : result = 0;
        endcase
        // 设置 zero
        if (result)  zero = 0;
        else  zero = 1;
    end
endmodule
```

Control.sv

```
// 编写：61821328 常兆骏
`timescale 1ns / 1ps
// 有限状态机输出控制信号
/*
instr   |branch |jr     |jump   |jal    |RegWE  |RegDst
|ALUsrcB|ALUOp  |MemWE  |MemReg
beq     |1      |x      |0      |x      |0      |x      |0      |-      |0      |x
j       |x      |0      |1      |x      |0      |x      |x      |x      |0      |x
jal     |x      |0      |1      |1      |1      |x      |x      |x      |0      |x
jr      |x      |1      |1      |x      |0      |x      |x      |x      |0      |x
[R]+-*..|0      |x      |0      |0      |1      |1      |0      |?      |0      |0
[I]+ &  |0      |x      |0      |0      |1      |0      |1      |?      |0      |0
[R]<<>> |0      |x      |0      |0      |1      |1      |0      |?      |0      |0
lw      |0      |x      |0      |0      |1      |0      |1      |+      |0      |1
sw      |0      |x      |0      |0      |0      |x      |1      |+      |1      |x
*/
module Control(
    input [5:0] op,          // op 操作符
    input [5:0] func,        // func 操作符
    // PC 控制
    output reg branch,       // 还需经过与 ALU 的 zero 再作用于 PC
    output reg jr,
    output reg jump,

    output reg jal,          // 写$rs。和 RegDst 共同决定写的地址

    output reg RegWE,        // (RF)写使能信号
    output reg RegDst,       // 选择写的地址 1:[15:11],为 R 指令; 0:[20:16],为 I 指令。
和 jal 共同决定写地址
```

```
    output reg ALUsrcB,     // 0:读取寄存器 2; 1:符号拓展后的立即数
    output reg [3:0] ALUOp, // ALU 操作控制

    output reg MemWE,       // 写数据存储器
    output reg MemReg       // 从内存到寄存器 0:计算结果, 1:读取结果
    );
    initial begin
        branch = 0; jr = 0; jump = 0; jal = 0;
        RegWE = 0; RegDst = 0; ALUsrcB = 0;
        ALUOp = 0; MemWE = 0; MemReg = 0;
    end
    always@(op or func)
    begin
        case (op)
            6'b000000:
            begin
                case (func[5:3])
                    3'b100:
                    begin
                        branch = 0; jr = 0; jump = 0;
                        jal = 0; RegWE = 1; RegDst = 1;
                        ALUsrcB = 0; MemWE = 0; MemReg = 0;
                        case (func[2:0])
                            3'b000: ALUOp =
4'b0101;//add
                            3'b010: ALUOp = 4'b0110;//sub
                            3'b100: ALUOp = 4'b0111;//and
                            3'b101: ALUOp = 4'b1000;//or
                            3'b111: ALUOp = 4'b1010;//nor
                            3'b110: ALUOp = 4'b1001;//xor
                            default:ALUOp =0;
                        endcase
                    end
                    3'b000:
                    begin
                        branch = 0; jr = 0; jump = 0;
                        jal = 0; RegWE = 1; RegDst = 1;
                        ALUsrcB = 0; MemWE = 0; MemReg = 0;
                        case (func[2:0])
                            3'b000: ALUOp = 4'b0000;//逻辑左移
                            3'b010: ALUOp = 4'b0010;//逻辑右移
                            3'b011: ALUOp = 4'b0001;//算数右移
                            default:ALUOp =0;
                        endcase
```

```
                end
                //jr
                3'b001:
                begin
                    branch = 0; jr = 1; jump = 1;
                    jal = 0; RegWE = 0; RegDst = 0;
                    ALUsrcB = 0;
                    ALUOp = 0; MemWE = 0; MemReg = 0;
                end
            endcase
        end
        default:
        begin
            case (op[5:3])
                //i
                3'b001:
                begin
                    branch = 0; jr = 0; jump = 0;
                    jal = 0; RegWE = 1; RegDst = 0;
                    ALUsrcB = 1; MemWE = 0; MemReg = 0;
                    case (op[2:0])
                        3'b000: ALUOp = 4'b0101;//addi
                        3'b100: ALUOp = 4'b0111;//andi
                        3'b101: ALUOp = 4'b1000;//ori
                        3'b110: ALUOp = 4'b1001;//xori
                        default:ALUOp =0;
                    endcase
                end
                //lw
                3'b100:
                begin
                    branch = 0; jr = 0; jump = 0;
                    jal = 0; RegWE = 1; RegDst = 0;
                    ALUsrcB = 1; ALUOp = 4'b0101;
                    MemWE = 0; MemReg = 1;
                end
                //sw
                3'b101:
                begin
                    branch = 0; jr = 0; jump = 0;
                    jal = 0; RegWE = 0; RegDst = 0;
                    ALUsrcB = 1; ALUOp = 4'b0101;
                    MemWE = 1; MemReg = 0;
                end
                //跳转
```

```
                    3'b000:
                    begin
                        jr = 0; RegDst = 0;
                        ALUsrcB = 0; ALUOp = 0;
                        MemWE = 0; MemReg = 0;
                        case (op[2:0])
                        //beq
                            3'b100:
                            begin
                                branch = 1; jump = 0;
                                jal = 0; RegWE = 0;
                                ALUsrcB = 0; ALUOp = 4'b0110;
                            end
                        //j
                            3'b010:
                            begin
                                branch = 0; jump = 1;
                                jal = 0; RegWE = 0;
                            end
                        //jal
                            3'b011:
                            begin
                                branch = 0; jump = 1;
                                jal = 1; RegWE = 1;
                            end
                        endcase
                    end
                endcase
            end
        endcase
    end
endmodule
```

DataMem.sv

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps

// 读取地址只有 14 位，道理同 rom
module DataMem(
    input           clk,        // 时钟
    input [31:0]    address,    // 来自 memorio 模块，来自是 ALU
    input [31:0]    write_data, // 来自译码单元的 read_data2
    input           Memwrite,   // 来自控制单元
    output[31:0]    read_data   // 从存储器中获得的数据
);
```

```
    wire clock = !clk; // 因为使用芯片的固有延迟，RAM 的地址线来不及在时钟上升沿准备好，使得时钟上升沿数据读出有误，所以采用反相时钟，使得读出数据比地址准备好要晚大约半个时钟，从而得到正确地址。
    // 分配 64KB RAM
    ram ram (
        .clka(clock),             // input wire clka
        .wea(Memwrite),          // input wire [0 : 0] wea
        .addra(address[15:2]),   // input wire [13 : 0] addra
        .dina(write_data),       // input wire [31 : 0] dina
        .douta(read_data)        // output wire [31 : 0] douta
    );
endmodule
```

InstMem.sv

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps

// PC 是 32 位，但是 16 字节的寻址空间，每 4 字节一单位，所以只要 14 根地址线
module InstrMem (
    // Program ROM Pinouts
    input           clk,          // ROM clock
    input   [31:0]  PC,           // 来源于取指单元的取指地址（PC）
    output  [31:0]  Instruction   // 给取指单元的读出的数据（指令）
);
    // 分配 64KB ROM，编译器实际只用 64KB ROM
    prgrom instmem(
        .clka(clk), // input wire clka
        .addra(PC[15:2]), // input wire [13 : 0] addra
        .douta(Instruction) // output wire [31 : 0] douta
    );
endmodule
```

PC.sv

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps

// 正常情况：PC+1*4，选下一条
// beq: op rs rt offset[15:0] 给出相对偏移量
// j:   op addr[25:0]         给出绝对位置
// jr:  op rs 0 0 0 funct     读寄存器得到绝对位置
// jal: op addr[25:0]         给出绝对位置 并保存当前地址+1*4

module PC(
    input clk,              // 时钟
```

```
    input [31:0] offset,    // beq 的偏置
    input [25:0] absoluteAddr,  // j(或 jal)的绝对地址
    input [31:0] registerAddr,  // jr 的来自寄存器的地址

    input branch,           // 控制 1 则选择 beq 分支(+4+offset*4) 0 则下一条(+4)
    input jr,               // 控制 1 则选择 jr, 0 则选择 j(jal)
    input jump,             // 控制 1 则选择 j(jal)或 jr, 0 则选择 beq 或+4

    input reset,            // 初值 1 则 PC 清零

    output reg [31:0] PC,   // 输出
    output [31:0] PC_plus4  // PC+1*4 给 jal 存储当前位置用
    );

    assign PC_plus4 = PC + 4;    // 正常情况
    wire [31:0] PC_beq = (offset << 2) + PC_plus4;   // beq
    wire [31:0] PC_j = {PC_plus4[31:28], absoluteAddr<<2}; // j(jal)

    wire [31:0] PC_1 = branch ? PC_beq : PC_plus4;
    wire [31:0] PC_2 = jr ? registerAddr : PC_j;
    wire [31:0] PC_final = jump ? PC_2 : PC_1;

    always @(negedge clk or posedge reset)
        PC <= reset ? 0 : PC_final;
endmodule
```

Registers.sv

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps
// 读写 32 个寄存器
module RegisterFile(
    input           clk,        // 时钟
    input           RegWre,     // 写使能信号，为 1 时，在时钟上升沿写入
    input [4:0]     A1,         // rs 寄存器地址输入端口
    input [4:0]     A2,         // rt 寄存器地址输入端口
    input [4:0]     WriteReg,   // 将数据写入的寄存器端口，其地址来源 rt 或 rd 字段
    input [31:0]    WriteData,  // 写入寄存器的数据输入端口
    input           reset,      // 1 则归零
    output[31:0]    ReadData1,  // rs 寄存器数据输出端口
    output[31:0]    ReadData2,  // rt 寄存器数据输出端口
    output reg [31:0] register[31:0]    // 可以输出
    );
    // 清零
```

```
    integer i;

    // 读寄存器 组合逻辑
    assign ReadData1 = register[A1];
    assign ReadData2 = register[A2];

    // 下降沿写寄存器
    always@(negedge clk or posedge reset) begin
        if (reset) begin
            for (i = 0; i < 32; i = i + 1)
                register[i] <= 0;
        end
        else begin
            // 如果寄存器不为 0，并且 RegWre 为真，写入数据
            if (RegWre && WriteReg != 5'b0)
                register[WriteReg] <= WriteData;
        end
    end

endmodule
```

SignExtend.sv

```
// 编写 61821326 李睿刚
`timescale 1ns / 1ps

module SignExtend(
    input[15:0] imm,
    output [31:0] SignImm
);
    assign SignImm[15:0] = imm[15:0];
    assign SignImm[31:16] = {16{imm[15]}};
endmodule
```

MIPS.sv(整合)

```
// 编写：61821326 李睿刚
`timescale 1ns / 1ps

module MIPS();
    reg clk = 1;
    reg PC_reset = 0;
    reg RF_reset = 0;
    wire [31:0]r[31:0];


    wire[31:0] PC;
```

```
    wire[31:0] PC_plus4;
    wire[31:0] Instr;
    wire[31:0] RD1;
    wire[31:0] RD2;
    wire[31:0] SignImm;
    wire branch, jr, jump, jal, RegWE, RegDst, ALUsrcB, MemWE, MemReg;
    wire[3:0] ALUOp;
    wire ALUzero;
    wire[31:0] ALUResult;
    wire[31:0] DMresult;

    Control ctrl(
        .op(Instr[31:26]),
        .func(Instr[5:0]),
        .branch(branch),
        .jr(jr),
        .jump(jump),
        .jal(jal),
        .RegWE(RegWE),
        .RegDst(RegDst),
        .ALUsrcB(ALUsrcB),
        .ALUOp(ALUOp),
        .MemWE(MemWE),
        .MemReg(MemReg)
    );
    InstrMem im(
        .clk(clk),
        .PC(PC),
        .Instruction(Instr)
    );
    SignExtend se(
        .imm(Instr[15:0]),
        .SignImm(SignImm)
    );
    PC pc(
        .clk(clk),
        .offset(SignImm),
        .absoluteAddr(Instr[25:0]),
        .registerAddr(RD1),
        .branch(ALUzero & branch),
        .jr(jr),
        .jump(jump),
        .reset(PC_reset),
        .PC(PC),
        .PC_plus4(PC_plus4)
```

```
    );
    RegisterFile rf(
        .clk(clk),
        .RegWre(RegWE),
        .A1(Instr[25:21]),
        .A2(Instr[20:16]),
        .WriteReg(jal?5'd31:(RegDst?Instr[15:11]:Instr[20:16])),
        .WriteData(jal?PC_plus4:(MemReg?DMresult:ALUResult)),
        .reset(RF_reset),
        .ReadData1(RD1),
        .ReadData2(RD2),
        .register(r)
    );
    ALU alu(
        .ALUOp(ALUOp),
        .A(RD1),
        .B(ALUsrcB?SignImm:RD2),
        .C(Instr[10:6]),
        .zero(ALUzero),
        .result(ALUResult)
    );
    DataMem dm(
        .clk(clk),
        .address(ALUResult),
        .write_data(RD2),
        .Memwrite(MemWE),
        .read_data(DMresult)
    );

    always #5 clk = ~clk;
    initial begin
        #2 PC_reset = 1; RF_reset = 1;
        #2 PC_reset = 0; RF_reset = 0;
    end
endmodule
```