# 计算机系统结构实验报告
# 实验 5

2021 年 6 月 23 日

**摘要**

基于实验 3、实验 4 的实验结果，本实验对部分已有模块进行修改，并且新实现了指令内存模块、数据选择器模块、PC 寄存器模块。然后将各个模块连接在一起，实现了类 MIPS 单周期处理器。该类 MIPS 单周期处理器支持 16 条 MIPS 指令（包括 R 型指令中的 add、sub、and、or、slt、sll、srl、jr；I 型指令中的 lw、sw、addi、ori、beq；J 型指令中的 j、jal）。本实验将通过软件仿真的形式让处理器运行指令，以此进行实验结果的验证。

## 目录

# 1 实验目的

1. 设计并实现类 MIPS 单周期处理器

2. 功能仿真

# 2 原理分析

## 2.1 主控制器模块

主控制器 (Ctr) 的输入为指令的操作码 (opCode) 字段，主控制器模块对操作码进行译码，向 ALU 控制器、寄存器、数据选择器等部件输出正确的控制信号。

本实验中，主控制器模块可以识别 R 型指令、立即数运算、lw、sw、beq、jump、jr、jal 指令并输出对应的控制信号。

主控制器模块产生的控制信号及说明如表1所示。

| 信号 | 内部寄存器 | 具体说明 |
|---|---|---|
| regDst | RegDst | 目标寄存器的选择信号；低电平：rt 寄存器；高电平：rd 寄存器 |
| aluSrc | ALUSrc | ALU 第二个操作数来源选择信号；低电平：rt 寄存器值，高电平：立即数拓展结果 |
| memToReg | MemToReg | 写寄存器的数据来源选择信号；低电平：ALU 运算结果，高电平：内存读取结果 |
| regWrite | RegWrite | 寄存器写使能信号，高电平说明当前指令需要进行寄存器写入 |
| memRead | MemRead | 内存读使能信号，高电平有效 |
| memWrite | MemWrite | 内存写使能信号，高电平有效 |
| aluOp | ALUOp | 3 位信号，发送给运算单元控制器 (ALUCtr) 用来进一步解析运算类型的控制信号 |
| branch | Branch | 条件跳转信号，高电平说明当前指令是条件跳转指令 |
| jump | Jump | 无条件跳转信号，高电平说明当前指令是无条件跳转指令 |
| jalSign | JalSign | 跳转并链接指令 (JAL) 信号，高电平说明当前指令是 JAL 指令 |
| extSign | ExtSign | 带符号扩展信号，高电平将对立即数进行带符号扩展 |

表 1: 主控制器产生的控制信号

其中 aluOp 信号代表的含义如表2所示。

| aluOp 的信号内容 | 指令 | 具体说明 |
|---|---|---|
| 101 | R | ALUCtr 结合指令 Funct 段决定最终操作 |
| 000 | lw,sw,addi,addiu | ALU 执行加法 |
| 001 | beq | ALU 执行减法 |
| 011 | andi | ALU 执行逻辑与 |
| 100 | ori | ALU 执行逻辑或 |
| 111 | xori | ALU 执行逻辑异或 |
| 010 | slti | ALU 执行带符号数大小比较 |
| 110 | sltiu | ALU 执行无符号数大小比较 |

表 2: aluOp 信号的具体含义以及解析方式

主控制器 (Ctr) 产生的各种控制信号与指令 OpCode 段的对应方式如表 3 所示。

| OpCode | 指令 | aluOp | aluSrc | memRead | memToReg | memWrite | regDst | regWrite | extSign | branch | jump | jalSign |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000000 | R 型指令 | 101 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 100011 | lw | 000 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 101011 | sw | 000 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 000100 | beq | 001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 000010 | j | 101 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 000011 | jal | 101 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 001000 | addi | 000 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 001001 | addiu | 000 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 001100 | andi | 011 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 001010 | xori | 111 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 001101 | ori | 100 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 001010 | slti | 010 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 001011 | sltiu | 110 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

表 3: 各指令对应的主控制器 (Ctr) 控制信号

## 2.2 ALU 控制器模块

ALU 控制器模块接受指令中 Funct 段以及来自 Ctr 模块的 aluOp 信号，输出 aluCtr 信号，aluCtr 信号直接决定 ALU 模块进行的计算操作。

ALUCtr 信号输出与 ALUOp 及 Funct 的对应关系如表4所示。除上表所示之外，本实验中 ALU 控制器模块还负责产生 shamtSign 信号与 jrSign 信号。关于这两个信号的说明如下：

- shamtSign 信号：当指令为 sll、srl、sra 时为高电平，其余情况下为低电平。该信号有效则表示需要使用指令 shamt 段作为 ALU 输入。
- jrSign 信号：当指令为 jr 时为高电平，其余情况下为低电平。

## 2.3 ALU 模块

ALU 模块接受 aluCtr 信号，并根据此信号选择执行对于的 ALU 计算功能。ALU 功能与 ALUCtr 信号的对应关系如表5所示。

ALU 模块产生的输出包括 32 位的运算结果以及 1 位 zero 信号；当运算结果为 0 时，zero 处于高电平，其余时候 zero 处于低电平。

## 2.4 寄存器模块

寄存器（Register）是中央处理器内用来暂存指令、数据和地址的存储器。寄存器的存贮容量有限，读写速度非常快。在计算机体系结构里，寄存器存储在已知时间点所作计算的中间结果，通过快速地访问数据来加速计算机程序的运行。

寄存器模块的输入与输出信号如表6所示。

对于读取操作，寄存器模块会根据寄存器选择信号 readReg1、readReg2 和寄存器内容立即响应并输出结果。对于写入操作，寄存器模块会在时钟下降沿且 regWrite 信号为高电平时，将 writeData 值写入 writeReg 指定的寄存器。

| 指令 | ALUOp | Funct | ALUCtr | 说明 |
|---|---|---|---|---|
| lw,sw,addi,addiu | 000 | xxxxxx | 0010 | ALU 执行加法运算 |
| beq | 001 | xxxxxx | 0110 | ALU 执行减法运算 |
| stli | 010 | xxxxxx | 0111 | ALU 执行带符号数大小比较 |
| stliu | 110 | xxxxxx | 1000 | ALU 执行无符号数大小比较 |
| andi | 011 | xxxxxx | 0000 | ALU 执行逻辑与运算 |
| ori | 100 | xxxxxx | 0001 | ALU 执行逻辑或运算 |
| xori | 111 | xxxxxx | 1011 | ALU 执行逻辑异或运算 |
| add | 101 | 100000 | 0010 | ALU 执行加法运算 |
| addu | 101 | 100001 | 0010 | ALU 执行加法运算 |
| sub | 101 | 100010 | 0110 | ALU 执行减法运算 |
| subu | 101 | 100011 | 0110 | ALU 执行减法运算 |
| and | 101 | 100100 | 0000 | ALU 执行逻辑与运算 |
| or | 101 | 100101 | 0001 | ALU 执行逻辑或运算 |
| xor | 101 | 100110 | 1011 | ALU 执行逻辑异或运算 |
| nor | 101 | 100111 | 1100 | ALU 执行逻辑或非运算 |
| slt | 101 | 101010 | 0111 | ALU 执行带符号数大小比较 |
| sltu | 101 | 101011 | 1000 | ALU 执行无符号数大小比较 |
| sll | 101 | 000000 | 0011 | ALU 执行逻辑左移运算 |
| sllv | 101 | 000100 | 0011 | ALU 执行逻辑左移运算 |
| srl | 101 | 000010 | 0100 | ALU 执行逻辑右移运算 |
| srlv | 101 | 000110 | 0100 | ALU 执行逻辑右移运算 |
| sra | 101 | 000011 | 1110 | ALU 执行算术右移运算 |
| srav | 101 | 000111 | 1110 | ALU 执行算术右移运算 |

表 4: 运算单元控制器 (ALUCtr) 的输入与输出关系

| ALUCtr | ALU 功能 |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0011 | Left Shift (logic) |
| 0100 | Right Shift (logic) |
| 0110 | sub |
| 0111 | set on less than (signed) |
| 1000 | set on less than (unsigned) |
| 1011 | XOR |
| 1100 | NOR |
| 1110 | Right Shift (arithmetic) |

表 5: ALU 执行功能与 ALUCtr 信号的对应方式

| 输入信号 | 长度 | 说明 |
|---|---|---|
| readReg1 | 5 | 读取寄存器 1 的编号 |
| readReg2 | 5 | 读取寄存器 2 的编号 |
| writeReg | 5 | 写入寄存器编号 |
| writeData | 32 | 写入的数据 |
| regWrite | 1 | 写使能信号 |
| clk | 1 | 时钟信号 |
| reset | 1 | 初始化信号 |
| 输出信号 | 长度 | 说明 |
| readData1 | 32 | 读取寄存器 1 的结果 |
| readData2 | 32 | 读取寄存器 2 的结果 |

表 6: 寄存器模块输入、输出信号

## 2.5 内存单元模块

内存 (Memory) 是计算机的重要部件之一，也称内存储器和主存储器，它用于暂时存放 CPU 中的运算数据，与硬盘等外部存储器交换的数据。它是外存与 CPU 进行沟通的桥梁，计算机中所有程序的运行都在内存中进行，内存性能的强弱影响计算机整体发挥的水平。

在本实验中，内存单元模块按字进行寻址，同时，我们不考虑页表、虚拟地址与物理地址的转化，只考虑直接操作物理地址的情况。实验中我们的内存大小设定为 1024 个字，但是为了契合一般的处理器设计，我们的内存模块依然能接受 32 位地址输入，但是仅当内存地址小于 1024 时，内存操作才是有效的。

内存模块输入输出信号如表7所示。

| 输入信号 | 长度 | 说明 |
|---|---|---|
| address | 32 | 内存地址 |
| writeData | 32 | 写入数据 |
| memWrite | 1 | 内存写使能信号 |
| memRead | 1 | 内存读使能信号 |
| clk | 1 | 时钟信号 |
| 输出信号 | 长度 | 说明 |
| readData | 32 | 内存读取结果 |

表 7: 内存模块输入、输出信号

与寄存器模块即时响应读取操作不同，内存模块仅在 memRead 处于高电平时才会进行读取操作。写入操作与寄存器模块相似，内存模块会在时钟下降沿且 memWrite 信号为高电平时，将 writeData 值写入 address 地址。

## 2.6 带符号扩展模块

带符号扩展模块可以根据主控制器模块 (Ctr) 的信号，以带符号扩展或无符号扩展对来自指令的 16 位数进行扩展，扩展结果为一个 32 位的数。

带符号扩展模块的输入输出信号如表8所示。

| 输入信号 | 长度 | 说明 |
|---|---|---|
| inst | 16 | 指令中的立即数 |
| signExt | 1 | 高电平代表进行带括号扩展，否则无符号扩展 |
| 输出信号 | 长度 | 说明 |
| data | 32 | 扩展结果 |

表 8: 带符号扩展模块输入、输出信号

## 2.7 数据选择器模块 (Mux/RegMux)

数据选择器模块接受两个输入信号和一个选择信号，产生一个输出信号。本实验中，我们使用了两种数据选择器，包括 Mux 和 RegMux。Mux 的输入与输出信号均为 32 位，用于对进行数据进行选择。RegMux 的输出和输出信号为 5 位，用于寄存器选取信号的选择。

数据选择器的电路模型如图1所示。

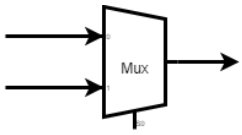


图 1: 数据选择器的电路模型

## 2.8 指令内存模块 (InstMem)

本实验中处理器采用哈佛架构，指令内存与数据内存分离。指令内存模块 (InstMem) 接受一个 32 位地址输入，输出一条 32 位指令。

## 2.9 PC 寄存器模块

PC 寄存器模块用于管理 PC 地址。接受输入 pcIn，在时钟上升沿将 pcIn 保存进 PC 寄存器。输出 pcOut 为当前 PC 地址，pcOut 与 PC 寄存器内容即时同步。当 reset 信号处于高电平时，将 PC 值重置为 0。

## 2.10 顶层模块 (top)

顶层模块将以上所有模块连接在一起，完成单周期 CPU 的功能。顶层模块中的连线包括数据通路和控制通路。数据通路用于传输中间数据，一般为 32 位或 16 位；控制通路用于传输控制信号。

顶层模块主要的数据通路与控制通路如图2所示。图2来自于计算机系统结构课程（CS359）课件，特此向授课教师邓倩妮老师表示感谢。

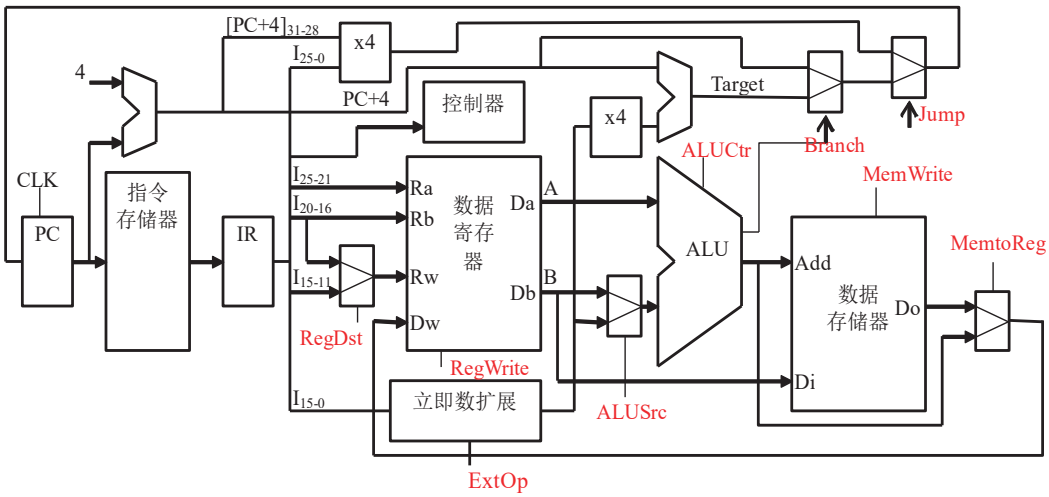


图 2: 顶层模块基础数据通路与控制通路

在图2中，我们使用两个数据选择器用于选择下一个 $PC$ 值，分别对应 beq 和 j 指令。对于非跳转指令，$PC+4$ 将被返回至 PC 寄存器模块。

除图2中所示的基础通路外，我还添加了部分元件用于支持 jal 与 jr 指令。具体改动如下：

- 添加两个数据选择器用于选择写入寄存器编号与寄存器写入数据，以此完成 jal 指令的实现。处理 jal 指令时，jalSign 信号与 regWrite 信号处于高电平，31 号寄存器被选中，同时寄存器写数据端口输入 $PC+4$。
- 在 branch 与 jump 选择器之间插入 jr 选择器，以此完成 jr 指令的实现。处理 jr 指令时，jrSign 信号处于高电平，jr 选择器选择寄存器读取结果，同时由于 jump 信号处于低电平，寄存器读取结果最终被返回至 PC 寄存器模块。

# 3 功能实现

## 3.1 主控制器模块的实现

本实验中，主控制器模块 (Ctr) 的输出结果由指令中 opCode 段决定。通过 case 语句，我们可以将指令与 opCode 对应，并根据每个指令的操作需求输出控制信号。

主控制器模块的完整实现见 Ctr.v，本实验中实现相比实验 3 添加了部分控制信号并将 ALUOp 拓展为 3 位，其余部分与实验 3 中实现类似。部分核心代码如下：

```
always @(opCode)
begin
    case(opCode)
    6'b000000: //R type
    begin
        RegDst = 1;
        ALUSrc = 0;
```

```
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ExtSign = 0;
        JalSign = 0;
        ALUOp = 3'b101;
        Jump = 0;
    end
    6'b100011: //lw
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 1;
        RegWrite = 1;
        MemRead = 1;
        MemWrite = 0;
        Branch = 0;
        ExtSign = 1;
        JalSign = 0;
        ALUOp = 3'b000;
        Jump = 0;
    end
    //后续代码类似，此处省略
    endcase
end
```

受限于篇幅，此处只展示 R 型指令与 lw 指令对应的实现。

代码中 RegDst、ALUSrc、MemToReg 等为控制信号寄存器，与对应输出信号线相连。

## 3.2 ALU 控制器模块的实现

ALU 控制器模块 (ALUCtr) 的输出由 aluOp 与 funct 共同决定，其行为与主控制器模块 (Ctr) 相似。不同的是，aluOp 与 funct 有部分位在部分指令中属于无关的位，因此我们使用 casex 替代 case。casex 中，可以用 x 表示我们不关心的位。

ALU 控制器模块的完整实现见 ALUCtr.v，相比实验 3 中实现，本实验的实现拓展了支持的运算类型，同时对 ALUOp 重新进行了编排。核心代码如下：

```
always @ (aluOp or funct)
begin
    ShamtSign = 0;
```

```
4       JrSign = 0;
5       casex({aluOp,funct})
6           9'b000xxxxxx:  // lw,sw,add,addiu
7               ALUCtrOut = 4'b0010;
8           9'b001xxxxxx:  // beq
9               ALUCtrOut = 4'b0110;
10          9'b010xxxxxx:   //stli
11              ALUCtrOut = 4'b0111;
12          9'b110xxxxxx:   //stliu
13              ALUCtrOut = 4'b1000;
14          9'b011xxxxxx:  // andi
15              ALUCtrOut = 4'b0000;
16          9'b100xxxxxx:  // ori
17              ALUCtrOut = 4'b0001;
18          9'b111xxxxxx:  // xori
19              ALUCtrOut = 4'b1011;
20          9'b101001000:  // jr
21          begin
22              ALUCtrOut = 4'b0101;
23              JrSign = 1;
24          end
25          //R type
26          9'b101000000:  // sll
27          begin
28              ALUCtrOut = 4'b0011;
29              ShamtSign = 1;
30          end
31          9'b101000010:  // srl
32          begin
33              ALUCtrOut = 4'b0100;
34              ShamtSign = 1;
35          end
36          9'b101000011:  // sra
37          begin
38              ALUCtrOut = 4'b1110;
39              ShamtSign = 1;
40          end
41          9'b101000100:  // sllv
42              ALUCtrOut = 4'b0011;
43          9'b101000110:  // srlv
44              ALUCtrOut = 4'b0100;
```

```
        9'b101000111:  // srav
            ALUCtrOut = 4'b1110;

        9'b101100000:  // add
            ALUCtrOut = 4'b0010;
        9'b101100001:  // addu
            ALUCtrOut = 4'b0010;
        9'b101100010:  // sub
            ALUCtrOut = 4'b0110;
        9'b101100011:  // subu
            ALUCtrOut = 4'b0110;
        9'b101100100:  // and
            ALUCtrOut = 4'b0000;
        9'b101100101:  // or
            ALUCtrOut = 4'b0001;
        9'b101100110:  // xor
            ALUCtrOut = 4'b1011;
        9'b101100111:  // nor
            ALUCtrOut = 4'b1100;
        9'b101101010:  // slt
            ALUCtrOut = 4'b0111;
        9'b101101011:  // sltu
            ALUCtrOut = 4'b1000;
    endcase
end
```

## 3.3 ALU 模块的实现

ALU 模块根据 aluCtr 信号完成指定的功能，可以使用 case 语句选择操作，利用 verilog 自带的运算符完成运算。

ALU 模块的完整实现见 ALU.v, 核心代码如下：

```
always @ (input1 or input2 or aluCtr)
begin
    case(aluCtr)
    4'b0000:     //AND
        ALURes = input1 & input2;
    4'b0001:     //OR
        ALURes = input1 | input2;
    4'b0010:     //ADD
        ALURes = input1 + input2;
```

```
    4'b0011:     //Left-shift
        ALURes = input2 << input1;
    4'b0100:     //Right-shift
        ALURes = input2 >> input1;
    4'b0101:
        ALURes = input1;
    4'b0110:     //SUB
        ALURes = input1 - input2;
    4'b0111:     //SLT
        ALURes = ($signed(input1) < $signed(input2));
    4'b1000:     //SLTU
        ALURes = (input1 < input2);
    4'b1011:     //xor
        ALURes = input1 ^ input2;
    4'b1100:     //nor
        ALURes = ~(input1 | input2);
    4'b1110:     //Right-shift-arithmetic
        ALURes = ($signed(input2) >> input1);
    endcase
    if(ALURes==0)
        Zero = 1;
    else
        Zero = 0;
end
```

SLT 运算功能的实现中，由于 Verilog 会默认以无符号数解释 wire 类型，需要通过 $signed 关键词将输入解释为带符号数后再进行比较。

在 ALU 模块的结尾，我们判断运算结果是否为 0，并以此设置 Zero 寄存器，最终作为 zero 信号输出。

## 3.4 寄存器模块的实现

寄存器会一直进行读操作，而由 regWrite 控制写操作。为实现信号同步，保证信号的完整性，写操作仅在时钟下降沿进行。由于读操作即时进行，寄存器内容被修改后，对应寄存器的读取结果也会同时更新。

相比与实验 4 中的实现，本实验中寄存器模块可以响应 reset 信号，当 reset 为高电平时，所有寄存器清零。

寄存器模块的完整实现见 Registers.v, 核心部分代码如下：

```
reg [31:0] RegFile[31:0];
integer i;

```

```
initial begin
    RegFile[0] = 0;
end

assign readData1 = RegFile[readReg1];
assign readData2 = RegFile[readReg2];

always @ (negedge clk or reset)
begin
    if(reset)
    begin
        for(i=0;i<32;i=i+1)
            RegFile[i] = 0;
    end
    else begin
        if(regWrite)
            RegFile[writeReg] = writeData;
    end
end
```

### 3.5 内存单元模块的实现

内存模块由 memRead 控制是否进行读取操作，当 memRead 或 address 信号发生变化或时钟处于下降沿时，内存模块会根据 address 指定的内存地址内容更新数据读取数据，并将其作为结果输出。尽管在实际的计算机系统中，内存写操作和内存读操作不会同时进行，但是本模块依然具备同时写与读的能力。

写操作由 memWrite 信号控制。与寄存器模块相似，为实现信号同步，保证信号的完整性，写操作仅在时钟下降沿进行。

内存模块的完整实现见 dataMemory.v, 核心部分代码如下：

```
reg [31:0] memFile [0:1023];
reg [31:0] ReadData;
always @(memRead or address)
begin
    if(memRead)
    begin
        if(address<1023)
            ReadData = memFile[address];
        else
            ReadData = 0;
    end
```

```
end

always @(negedge clk)
begin
    if(memWrite)
        if(address<1023)
            memFile[address] = writeData;
    if(memRead)
        if(address<1023)
            ReadData = writeData;
end

assign readData = ReadData;
```

### 3.6 带符号扩展模块的实现

带符号扩展可以通过在高 16 位填入立即数第 15 位实现，无符号扩展可以通过在高 16 位填入 0 实现。为了在两种工作模式下切换，可以通过一个三目运算符根据 signExt 信号在两种扩展结果中进行选择。

带符号扩展模块的完整实现见 signext.v, 核心部分代码如下：

```
assign data = signExt?{{16{inst[15]}},inst[15:0]}:{{16{0}},inst
    [15:0]};
```

### 3.7 数据选择器模块的实现

使用 Verilog 自带的三目运算符即可实现数据选择器的功能。Mux 与 RegMux 的差异仅在于输入输出数据长度不同。

数据选择器模块的完整实现见 Mux.v 与 RegMux.v，核心代码如下：

```
assign out = select?input1:input0;
```

### 3.8 指令内存模块的实现

指令内存模块只需要根据 PC 地址输出对应的指令，实现相对简单。

指令内存模块的完整实现见 InstMem.v，核心代码如下：

```
reg [31:0] instFile[0:1023];
assign inst = instFile[address/4];
```

### 3.9 PC 寄存器模块的实现

PC 寄存器模块在时钟上升沿将 pcIn 保存进 PC 寄存器，同时保证 pcOut 输出与 PC 寄存器内容一致。

PC 寄存器模块的完整实现见 PC.v，核心代码如下：

```
reg [31:0] PC;

initial PC = 0;

always @ (posedge clk or reset)
begin
    if(reset)
        PC = 0;
    else
        PC = pcIn;
end
assign pcOut = PC;
```

### 3.10 顶层模块的实现

顶层模块的原理如2.10节所述。为实现顶层模块，首先需要声明需要的所有连接线路。连线的声明如下：

```
wire REG_DST;
wire REG_WRITE;
wire EXT_OP;
wire ALU_SRC;
wire[2:0] ALU_OP;
wire[3:0] ALU_CTR;
wire BRANCH;
wire JUMP;
wire JAL_SIGN;
wire MEM_WRITE;
wire MEM_READ;
wire MEM_TO_REG;
wire ALU_ZERO;
wire SHAMT_SIGN;
wire JR_SIGN;
wire[4:0] WRITE_REG_ID;
wire[4:0] WRITE_REG_ID_AFTER_JAL_MUX;

```

```
wire[31:0] INST;
wire[31:0] REG_WRITE_DATA_AFTER_JAL_MUX;
wire[31:0] REG_WRITE_DATA;
wire[31:0] REG_READ_DATA1;
wire[31:0] REG_READ_DATA2;
wire[31:0] EXT_IMM;
wire[31:0] ALU_INPUT1;
wire[31:0] ALU_INPUT2;
wire[31:0] ALU_OUTPUT;
wire[31:0] MEM_OUTPUT_DATA;
wire[31:0] MEM_INPUT_DATA;
wire[31:0] PC_IN;
wire[31:0] PC_OUT;
wire[31:0] PC_AFTER_BRANCH_MUX;
wire[31:0] PC_AFTER_JR_MUX;
wire[31:0] JUMP_ADDR;
```

上述代码中，第一部分为控制信号连线，第二部分为数据通路连线。

除了将上述连线与各个模块的对应端口相连，还需要完成外部信号 reset 和 clk 的连接。reset 与 PC 寄存器模块、寄存器模块相连，用于完成处理器重置工作；clk 与 PC 寄存器模块、寄存器模块、数据内存模块相连，用于同步数据写入的时间。

以主控制器模块为例，其连接如下：

```
Ctr main_ctr(
    .opCode(INST[31:26]),
    .regDst(REG_DST),
    .aluSrc(ALU_SRC),
    .memToReg(MEM_TO_REG),
    .regWrite(REG_WRITE),
    .memRead(MEM_READ),
    .memWrite(MEM_WRITE),
    .branch(BRANCH),
    .aluOp(ALU_OP),
    .jump(JUMP),
    .extSign(EXT_OP),
    .jalSign(JAL_SIGN)
);
```

顶层模块中完成下一个 PC 地址选择部分的实现如下：

```
Mux branch_mux(
    .select(BRANCH & ALU_ZERO),
    .input1(PC_OUT+4+(EXT_IMM<<2)),
```

```
    .input0(PC_OUT+4),
    .out(PC_AFTER_BRANCH_MUX)
);

Mux jr_mux(
    .select(JR_SIGN),
    .input0(PC_AFTER_BRANCH_MUX),
    .input1(REG_READ_DATA1),
    .out(PC_AFTER_JR_MUX)
);

Mux jump_mux(
    .select(JUMP),
    .input0(PC_AFTER_JR_MUX),
    .input1(JUMP_ADDR),
    .out(PC_IN)
);
```

顶层模块中 jal 指令相关数据选择器的连线如下：

```
Mux jal_data_mux(
    .select(JAL_SIGN),
    .input0(REG_WRITE_DATA),
    .input1(PC_OUT+4),
    .out(REG_WRITE_DATA_AFTER_JAL_MUX)
);

Mux jal_reg_id_mux(
    .select(JAL_SIGN),
    .input0(WRITE_REG_ID),
    .input1(5'b11111),
    .out(WRITE_REG_ID_AFTER_JAL_MUX)
);
```

其中 REG_WRITE_DATA_AFTER_JAL_MUX 与寄存器模块的 writeData 端口相连，WRITE_REG_ID_AFTER_JAL_MUX 与寄存器模块的 writeReg 端口相连。

顶层模块的完整实现见 top.v。

# 4 结果验证

编写如表9所示的汇编代码进行测试。

| 地址 | 指令 | 汇编指令 | 执行结果 | | |
|---|---|---|---|---|---|
| | | | $0 | $1 | $2 |
| 0x00 | 00001000000000000000000000000100 | j | 跳转至 0x10 | | |
| 0x04 | 00000000000000000000000000000000 | nop | 不执行 | | |
| 0x08 | 00000000000000000000000000000000 | nop | 不执行 | | |
| 0x0c | 00000000000000000000000000000000 | nop | 不执行 | | |
| 0x10 | 00001100000000000000000000000110 | jal | 跳转至 0x18, $31 = 0x14 | | |
| 0x14 | 00000000000000000000000000000000 | nop | 空指令 | | |
| 0x18 | 10001100000000010000000000000011 | lw 1, 3(0) | 3 | x | x |
| 0x1c | 10001100000000010000000000000100 | lw 2, 4(0) | 3 | 4 | x |
| 0x20 | 00000000001000100001100000100000 | add 3,1,$2 | 3 | 4 | 7 |
| 0x24 | 00000000001000100001100000100100 | and 3,1,$2 | 3 | 4 | 0 |
| 0x28 | 00000000001000100010000000100010 | sub 4,1,$2 | $4 = -1 | | |
| 0x2c | 00000000000000010000100001000000 | sll 1,1,1 | 6 | 4 | 0 |
| 0x30 | 00000000000000011000110001000000 | sll 3,3,3 | 6 | 4 | 0 |
| 0x34 | 00000000001000110000100000100101 | or 1,3,$1 | 6 | 4 | 0 |
| 0x38 | 00100000000000100000000000000110 | addi 2,0,6 | 6 | 6 | 0 |
| 0x3c | 00101000010000010000000000000001 | slti 1,1,1 | 0 | 6 | 0 |
| 0x40 | 00110100010000010000000000000001 | ori 1,2,1 | 7 | 6 | 0 |
| 0x44 | 00000000010000110000100000000100 | sllv 1,3,$2 | 0 | 6 | 0 |
| 0x48 | 00000000001000100001100000100001 | addu 3,1,$2 | 0 | 6 | 6 |
| 0x4c | 00000000001000100001100000100010 | sub 3,1,$2 | 0 | 6 | -6 |
| 0x50 | 00000000001000100001100000100011 | subu 3,1,$2 | 0 | 6 | -6 |
| 0x54 | 00000000001000100001100000100101 | or 3,1,$2 | 0 | 6 | 6 |
| 0x58 | 00000000001000100001100000100110 | xor 3,1,$2 | 0 | 6 | 6 |
| 0x5c | 00000000001000100001100000100111 | nor 3,1,$2 | 0 | 6 | -7 |
| 0x60 | 00000000001000100001100000101010 | slt 3,1,$2 | 0 | 6 | 1 |
| 0x64 | 00000000000000010000100001000011 | sra 1,1,1 | 0 | 6 | 1 |
| 0x68 | 10101100000000010000000000000001 | sw 2, 1(0) | 写入内存 | | |
| 0x6c | 00010000101000000000000000000001 | beq 5,0,1 | 跳转至 0x74 | | |
| 0x70 | 00000000010000000000000000001000 | jr $2 | 不执行 | | |
| 0x74 | 00000011111100000000000000001000 | jr $31 | 跳转至 0x14 | | |

表 9: 仿真使用的指令

数据内存的初始值如表10所示。

在激励文件中载入指令内存与数据内存数据。

```
1 $readmemb("E:/course/arclab/lab05/mem_inst.dat",top.inst_mem.
      instFile);
2 $readmemh("E:/course/arclab/lab05/mem_data.dat",top.memory.memFile);
```

| 地址 | 数据 |
|---|---|
| 0x00 | 0x00000000 |
| 0x01 | 0x00000001 |
| 0x02 | 0x00000002 |
| 0x03 | 0x00000003 |
| 0x04 | 0x00000004 |
| 0x05 | 0x00000005 |

表 10: 数据内存初始值

仿真结果如图3所示。需要特别说明的是，由于显示缩放问题，部分数字可能没有显示完整。

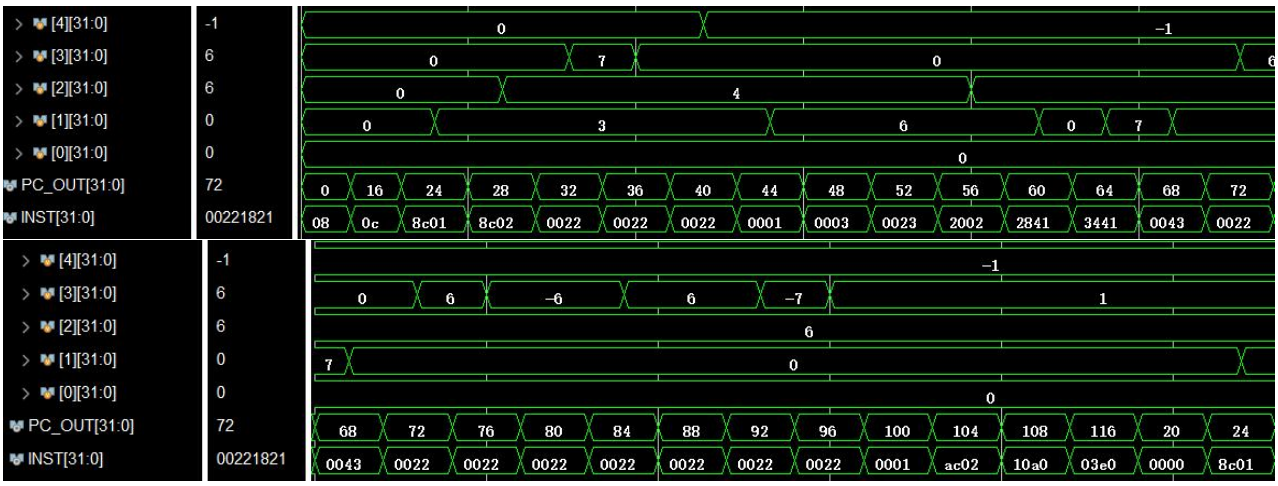


图 3: 仿真截图

可以看出，我们实现的类 MIPS 单周期处理器完成了设计的所有功能。

# 5 总结与反思

本实验结合实验 3 和实验 4 实现的功能模块，实现了一个类 MIPS 单周期处理器。由于我们在计算机系统结构课程已经学习过单周期处理器结构，本实验的实现较为简单。本实验也让我明白，在计算机系统中，想实现一个复杂化的功能，只需要先实现所有简单模块再组合在一起即可。

本实验主要难度在于调试阶段，在初次实现顶层模块时，我没有作出连线图，而是凭借自己的记忆进行连线，出现了较多连线错误，导致调试工作异常复杂。最终，我按照计算机系统结构课程课件上提供的单周期处理器连线图重新检查了所有连线。这样的经历提醒我，在以后的实验中，应当先完成整体设计再开始代码实现。

除此之外，为编写本次实验中用于仿真的代码，我复习了 MIPS 指令的相关知识，重新熟悉了 MIPS 指令的分类、分段结构。

总而言之，本次实验让我受益匪浅。