# 组成原理课程第 六 次实报告

# 实验名称：单周期 CPU 实验改进

学号： 2213041 姓名： 李雅帆 班次： 李涛老师

## 一、实验目的

1. 理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。
2. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
3. 熟悉并掌握单周期 CPU 的原理和设计。
4. 进一步加强运用 verilog 语言进行电路设计的能力。
5. 为后续设计多周期 cpu 的实验打下基础。实验内容说明
6.结合实验指导手册中的实验六（单周期 CPU 实验）完成功能改进，在原有 CPU 基础上，扩充 CPU 可运行的 MIPS 指令，注意：扩充的指令应为一个时钟周期内能够执行完的指令，要求至少一个 R 型，一个 I 型，另外一个自选。

## 二、实验原理图

1.单周期 CPU 的实现框图如下：

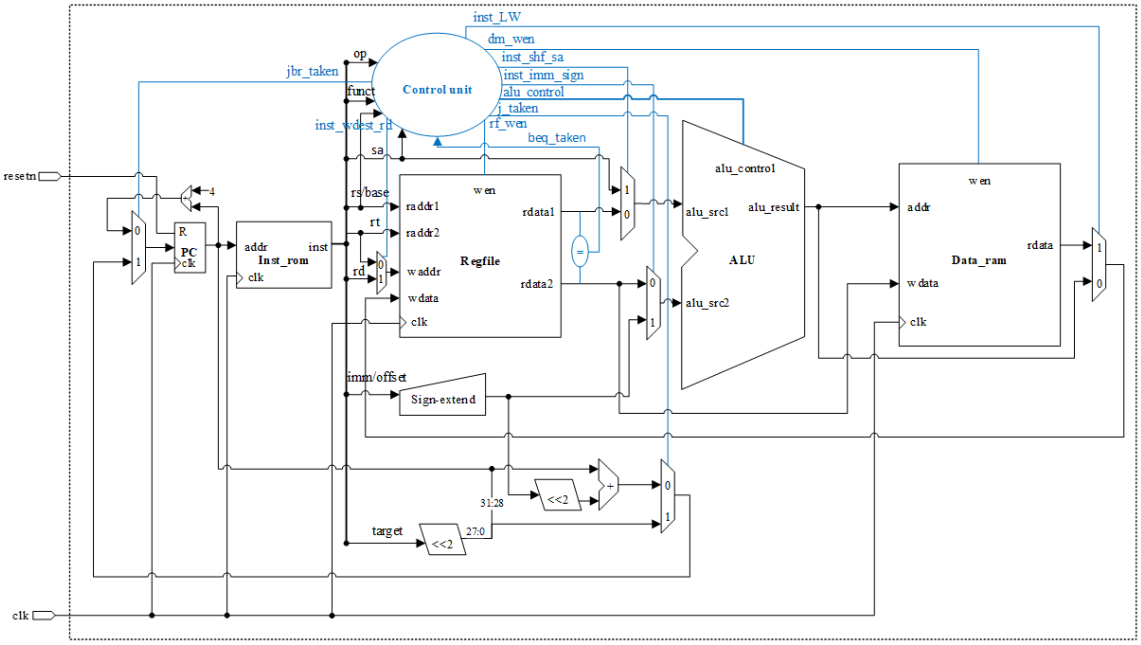



图 7.3 单周期 CPU 的实现框图

2.实验顶层模块框图如下

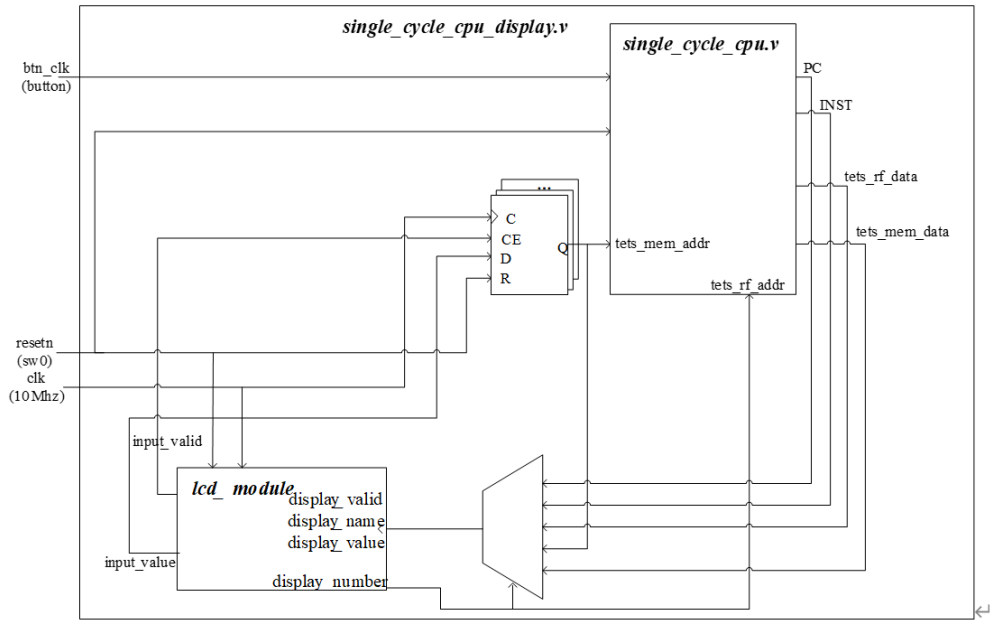



图 7.4 单周期 CPU 参考设计的顶层模块框图

## 三、实验步骤

我通过对原代码进行修改，在单周期 CPU 中实现三条新的指令：无符号小于置位(SLTU), 算数右移 (SRA), 以及装载字节 (LB) 。

### 1.在 single_cycle_cpu.v 中的修改

（1）新增的指令定义”inst_SLTU”,” inst_SRA”, 和 “inst_LB”。将新指令加入到 ALU 控制信号中，以便在执行阶段正确处理这些指令。

```
// 新增指令
assign inst_SLTU  = op_zero & sa_zero    & (funct == 6'b101011);// 小于无符号置位
assign inst_SRA   = op_zero & (rs==5'd0) & (funct == 6'b000011);// 算数右移
assign inst_LB    = (op == 6'b100100);                  // 无符号数装载字节
```

（2） 修改执行和访存模块

```
// 传递到执行模块的ALU源操作数和操作码
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;
wire inst_lb;
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
assign inst_sub = inst_SUBU; // 减法
assign inst_slt = inst_SLT;  // 小于置位
assign inst_sltu= inst_SLTU; // 无符号小于置位，新增
assign inst_and = inst_AND;  // 逻辑与
assign inst_nor = inst_NOR;  // 逻辑或非
assign inst_or  = inst_OR;   // 逻辑或
assign inst_xor = inst_XOR;  // 逻辑异或
assign inst_sll = inst_SLL;  // 逻辑左移
assign inst_srl = inst_SRL;  // 逻辑右移
assign inst_sra = inst_SRA;  // 算数右移，新增
assign inst_lui = inst_LUI;  // 立即数装载高位
assign inst_lb = inst_LB;    // 立即数装载字节，新增
```

①执行部分的修改，确保 ALU 模块能够正确接收和处理新增指令所需的操作数和控制信号。

```
//------------------------------{执行}begin------------------------------//
    wire [31:0] alu_result;

    alu alu_module(
        .alu_control  (alu_control ),  // I, 12, ALU控制信号
        .alu_src1     (alu_operand1),  // I, 32, ALU操作数1
        .alu_src2     (alu_operand2),  // I, 32, ALU操作数2
        .alu_result   (alu_result  )   // O, 32, ALU结果
    );
//------------------------------{执行}end--------------------------------//
```

②访存部分的修改，确保在访存阶段处理新增的访存指令。

```
//------------------------------{访存}begin------------------------------//
    wire [3 :0] dm_wen;
    wire [31:0] dm_addr;
    wire [31:0] dm_wdata;
    wire [31:0] dm_rdata;
    assign dm_wen   = {4{inst_SW}} & resetn;     // 内存写使能，非resetn状态下有效
    assign dm_addr  = alu_result;                // 内存写地址，为ALU结果
    assign dm_wdata = rt_value;                  // 内存写数据，为rt寄存器值
    data_ram data_ram_module(
        .clk   (clk         ),  // I, 1,  时钟
        .wen   (dm_wen      ),  // I, 1,  写使能
        .addr  (dm_addr[6:2]),  // I, 32, 读地址
        .wdata (dm_wdata    ),  // I, 32, 写数据
        .rdata (dm_rdata    ),  // O, 32, 读数据

        //display mem
        .test_addr(mem_addr[6:2]),
        .test_data(mem_data     )
    );
//------------------------------{访存}end--------------------------------//
```

③写回部分的修改，确保新指令能够正确写回结果到寄存器堆。

```
//------------------------------------{写回}begin------------------------------------//
    wire inst_wdest_rt;   // 寄存器堆写入地址为rt的指令
    wire inst_wdest_rd;   // 寄存器堆写入地址为rd的指令
    assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI| inst_LB;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                         | inst_OR   | inst_XOR  | inst_SLL | inst_SRL | inst_SLTU | inst_SRA;
    // 寄存器堆写使能信号，非复位状态下有效
    assign rf_wen   = (inst_wdest_rt | inst_wdest_rd) & resetn;
    assign rf_waddr = inst_wdest_rd ? rd : rt;        // 寄存器堆写地址rd或rt
    assign rf_wdata = inst_LW ? dm_rdata : alu_result;// 写回结果，为load结果或ALU结果
//------------------------------------{写回}end--------------------------------------//
```

**2.在 alu.v 中的修改**

（1）扩展了 alu_control 的位宽：将 alu_control 从 input [11:0] 扩展到 input [12:0]。

```
module alu(
    input  [12:0] alu_control,  // ALU控制信号
    input  [31:0] alu_src1,     // ALU操作数1,为补码
    input  [31:0] alu_src2,     // ALU操作数2，为补码
    output [31:0] alu_result    // ALU结果
    );
```

（2）增加了对 LB 指令的支持：

①在 alu_control 中增加了第 13 位来表示 alu_lb。

```
// ALU控制信号，独热码
wire alu_add;   //加法操作
wire alu_sub;   //减法操作
wire alu_slt;   //有符号比较，小于置位，复用加法器做减法
wire alu_sltu;  //无符号比较，小于置位，复用加法器做减法
wire alu_and;   //按位与
wire alu_nor;   //按位或非
wire alu_or;    //按位或
wire alu_xor;   //按位异或
wire alu_sll;   //逻辑左移
wire alu_srl;   //逻辑右移
wire alu_sra;   //算术右移
wire alu_lui;   //高位加载
wire alu_lb;    //字节加载
```

```
assign alu_lb  = alu_control[12];
assign alu_add  = alu_control[11];
assign alu_sub  = alu_control[10];
assign alu_slt  = alu_control[ 9];
assign alu_sltu = alu_control[ 8];
assign alu_and  = alu_control[ 7];
assign alu_nor  = alu_control[ 6];
assign alu_or   = alu_control[ 5];
assign alu_xor  = alu_control[ 4];
assign alu_sll  = alu_control[ 3];
assign alu_srl  = alu_control[ 2];
assign alu_sra  = alu_control[ 1];
assign alu_lui  = alu_control[ 0];
```

②新增了 lb_result，用于字节加载操作：assign lb_result = {alu_src2[7:0], 24'd0};

```
assign and_result = alu_src1 & alu_src2;      // 与结果为两数按位与
assign or_result  = alu_src1 | alu_src2;      // 或结果为两数按位或
assign nor_result = ~or_result;               // 或非结果为或结果按位取反
assign xor_result = alu_src1 ^ alu_src2;      // 异或结果为两数按位异或
assign lui_result = {alu_src2[15:0], 16'd0}; // 立即数装载结果为立即数移位至高半字节
assign lb_result = {alu_src2[7:0], 24'd0};   // 字节装载
```

③在最终结果选择逻辑中添加了 alu_lb 对应的 lb_result。

```
// 选择相应结果输出
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt           ? slt_result :
                    alu_sltu          ? sltu_result :
                    alu_and           ? and_result :
                    alu_nor           ? nor_result :
                    alu_or            ? or_result  :
                    alu_xor           ? xor_result :
                    alu_sll           ? sll_result :
                    alu_srl           ? srl_result :
                    alu_sra           ? sra_result :
                    alu_lui           ? lui_result :
                    alu_lb           ? lb_result:
                    32'd0;
```

**3.在 inst_rom.v 中的修改**

添加了对三个新 MIPS 指令（SRA, LB, SLTU）的支持。这些新指令被插入到指令存储器中并在指定的地址处进行执行。

（1）新增指令部分

①用 sra 指令替换了 beq 指令，这个指令将寄存器 $2 的值算术右移 1 位，并将结果存储到寄存器 $14 中。

指令地址 0x2C (11 号)

修改前: beq $9 ,$1,#2 (32'h11210002) - 条件跳转指令。

修改后: sra $14, $2, #1 (32'h00027043) - 算术右移指令。

②用 lbu 指令替换了 addiu 指令，这个指令将内存地址为 $0 + 7 的字节装入寄存器 $15 中。

指令地址 0x30 (12 号)

修改前: addiu $1 ,$0,#4 (32'h24010004) - 不执行的指令。

修改后: lbu $15, #7($0) (32'h900F0007) - 装入无符号字节指令。

③用 sltu 指令替换了 lw 指令，这个指令将寄存器 $4 和 $5 的值进行无符号比较，如果 $4 小于 $5，则寄存器 $13 设为 1，否则设为 0。

指令地址 0x34 (13 号)

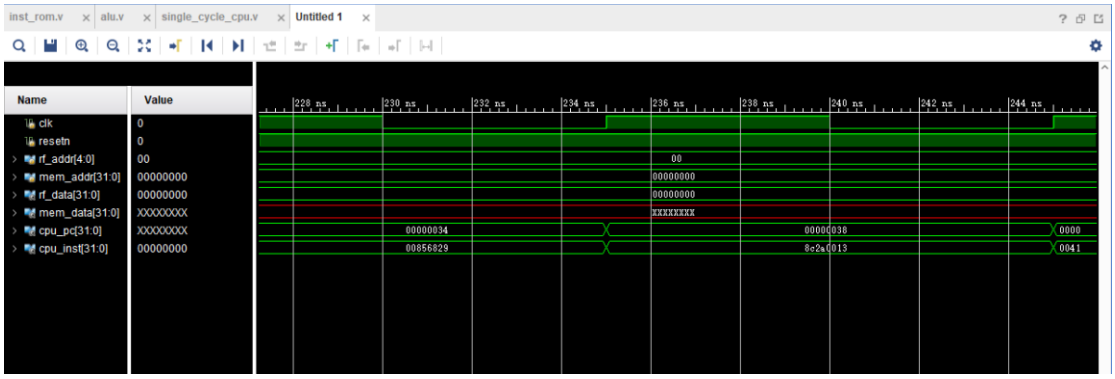
修改前: lw $10,#19($1) (32'h8C2A0013) - 从内存装入字指令。

修改后: sltu $13, $4, $5 (32'h00856829) - 设置无符号小于指令。

```
assign inst_rom[11] = 32'h00027043; // 2CH: sra   $14, $2, #1 | $14 = $2 >> 1
assign inst_rom[12] = 32'h900F0007; // 30H: lbu   $15, #7($0) | $15 = [$7]
assign inst_rom[13] = 32'h00856829; // 34H: sltu  $13, $4, $5 | $13 = $4 < $5 ? 1 : 0
```

## 四、实验结果分析

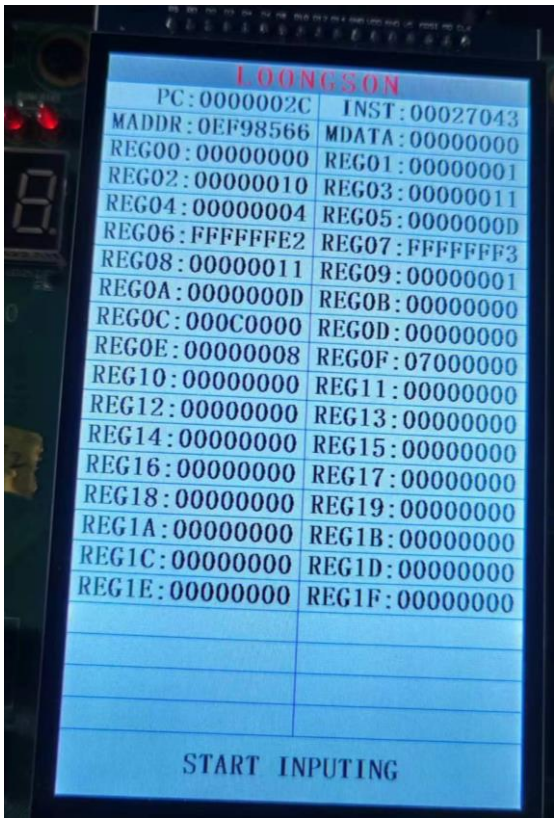
1.仿真结果截图



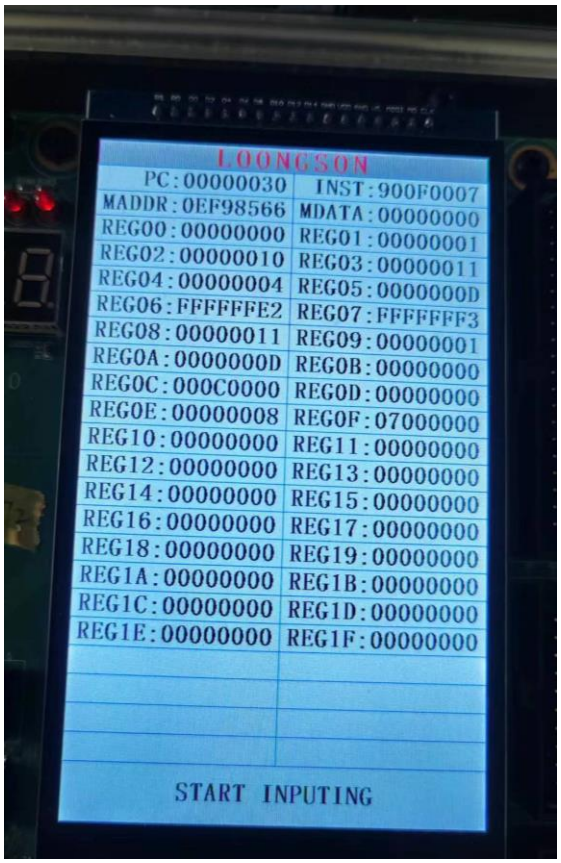

2.试验箱运行结果拍照

由下面的上箱结果可以看出，成功实现了扩充 CPU 可运行的 MIPS 指令，在单周期 CPU 中实现三条新的指令：无符号小于置位 (SLTU), 算数右移 (SRA), 以及装载字节 (LB) 。

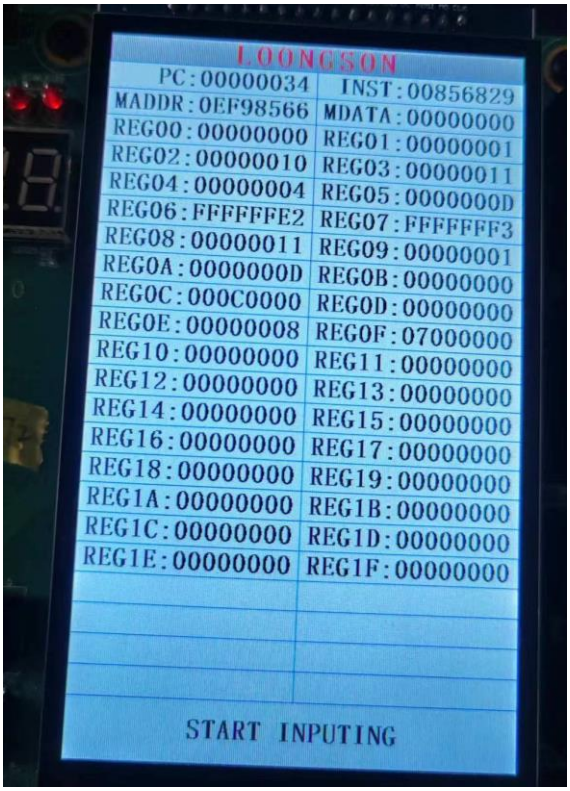
①指令地址 0x2C (11 号)——算术右移指令。




②指令地址 0x30 ——装入无符号字节指令。



③指令地址 0x34 —— 设置无符号小于指令。




## 五、总结感想

这次的实验通过对原有单周期 CPU 进行改进，实现了三条新的 MIPS 指令，这为我对 CPU 设计和 Verilog 语言应用的理解提供了更深入的认识。在扩充指令的过程中，我需要充分理解新指令的功能和编码方式，以确保在一个时钟周期内能够正确执行指令。这个过程不仅考验了我的逻辑思维能力，也提升了我的 Verilog 编程技能。

通过这次实验，我对 MIPS 指令集的常用指令有了更深入的了解，还学会了如何对指令进行归纳分类和扩充。同时，通过实践对单周期 CPU 的原理和设计有了更清晰的认识，对处理器结构中的延迟槽和哈佛结构也有了更深入的认识，我更加熟悉了 CPU 设计的流程和方法。