计组实验第六次实验报告

实验名称：alu模块的复现与改进

姓名：田晋宇　　学号：2212039　　班次：李涛班

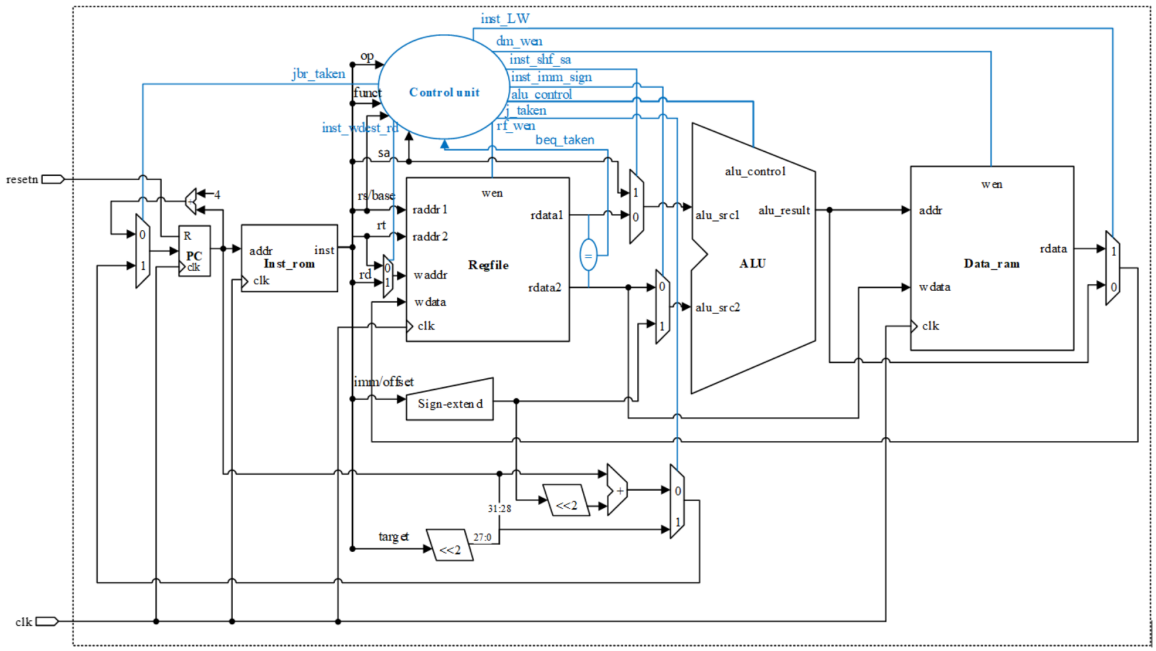
实验目的

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

1. 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。建议在 ALU 实验改进基础上补充。
2. 实验报告中原理图为指导手册中的 display 模块图，不用修改，报告中的内容和展示的结 果应扩充指令的步骤和实验结果。
3. 对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。

实验原理图



1. 指令

单周期CPU能够执行MIPS指令集系统的一个子集，共16条指令，包括存储访问指令、运算指令、跳转指令。根据拥有的字段类型不同，我们将指令分为 **R 型指令**、**I 型指令** 和 **J 型指令**。

- **R 型指令**

op	rs	rt	rd	shamd	funct

op 段（6b）： 恒为0b000000；

rs（5b）、rt（5b） ： 两个源操作数所在的寄存器号；

rd（5b）： 目的操作数所在的寄存器号；

shamt（5位）： 位移量，移位指令的移位位数；

func（6b）： 决定 R 型指令的具体功能。

- **I 型指令**

op	rs	rt	constant or address

op段（6b）： 决定 I 型指令类型；

rs（5b）： 是第一个源操作数所在的寄存器号；

rt （5b）： 是第二个源操作数所在的寄存器号 或 目的操作数所在的寄存器编号。

constant or address（16b）： 立即数或地址

- **J 型指令**

op	address

op段（6b）： 决定 J 型指令类型；

constant or address（26b）： 转移地址

2. 不同指令的执行过程

- **R 型指令**

1. 从指令存储器中取指令，更新 PC 。
2. ALU 根据 funct 字段确定 ALU 的功能。
3. 从寄存器堆中读出寄存器 rs 和 rt。
4. ALU 根据 2 中确定的功能，对从寄存器堆读出的数据进行操作。
5. 将运算结果写入到 rd 字段对应的目标寄存器。

I 型指令

- 存取指令：
    1. 从指令存储器中取指令，更新 PC 。
    2. ALU 根据 op 字段确定 ALU 的功能。

3. 从寄存器堆中读出寄存器 rs 的值，并将其与符号扩展后的指令低16位立即数的值相加。
4. 若为存储指令，则将 rt 寄存器中的值存到上步相加得到的存储器地址；

   若为取数指令，则将 上步所得存储器地址里所存的数据放到 rt 目标寄存器中。

- 分支指令：
  1. 从指令存储器中取指令，更新 PC 。
  2. 从寄存器堆中读出寄存器 rs 和 rt 的值。
  3. 将所读寄存器的两值相减。
  4. 根据上步的结果是否为0，将 PC+4 的值或 address 字段所对应地址存入PC中。

J 型指令

1. 从指令存储器中取指令，更新 PC 。
2. 取出 address 字段，作为目标跳转地址。
3. 将目标跳转地址存入PC中。

实验步骤

在本次实验中，我一共增加了三个指令，选择之前写过的按位同或操作作为添加的R型指令，选择低位加载作为添加的I型指令，再增加一条I型立即数与。

1. **对single_cycle_cpu.v 文件的修改**

将添加的两个操作添加到指令列表中

- inst_NXOR 表示按位同或操作
- inst_HUI 表示低位加载操作

```
wire inst_NXOR,
wire inst_HUI;
wire inst_ANDI;
```

接着为其添加对应的操作数：

```
assign inst_NXOR  = (op == 6'b110001);                  // 逻辑同或运算
assign inst_HUI   = (op == 6'b110000);                  // 立即数装载低半字节
assign inst_ANDI  = (op == 6'b111111);                  //立即数与
```

传递到执行模块的ALU模块的操作码,并对一些操作进行扩展：

```
wire inst_nxor, inst_hui, inst_andi;
assign inst_hui = inst_HUI;  // 立即数装载低位
assign inst_nxor = inst_NXOR;// 逻辑同或
assign inst_andi = inst_ANDI;  // 立即数装载低位
```

```
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW | inst_HUI |
inst_ANDI;
```

```
assign alu_control = {inst_nxor,
                      inst_hui,
                      inst_andi,
                      inst_add,        // ALU操作码，独热编码
```

```
                    inst_sub,
                    inst_slt,
                    inst_sltu,
                    inst_and
                    inst_nor,
                    inst_or,
                    inst_xor,
                    inst_sll,
                    inst_srl,
                    inst_sra,
                    inst_lui};
```

```
assign inst_wdest_rt = inst_ADDIU | inst_LW   | inst_LUI | inst_HUI|
inst_ANDI;
assign inst_wdest_rd = inst_ADDU  | inst_SUBU | inst_SLT | inst_AND |
 inst_NOR  | inst_OR   | inst_XOR  | inst_SLL | inst_SRL | inst_NXOR ;
```

2. **对alu.v的修改**

   首先对ALU的控制信号位数进行拓展

   ```
   input  [14:0] alu_control,  // ALU控制信号
   ```

   添加新添加的指令的控制信号,结果信号:

   ```
   wire alu_nxor;  //按位同或
   wire alu_hui;   //低位加载
   wire alu_andi;  //按位与

   assign alu_andi = alu_control[14];
   assign alu_hui  = alu_control[13];
   assign alu_nxor = alu_control[12];

   wire [31:0] hui_result;
   wire [31:0] nxor_result;
   wire [31:0] andi_result;
   ```

   添加新的指令的运算实现方式:

   ```
   assign hui_result = {16'd0, alu_src2[15:0]};
   assign nxor_result= ~xor_result;
   assign andi_result= alu_src1 & alu_src2;
   ```

   结果选择输出

   ```
   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_hui          ? hui_result :
                    alu_nxor         ? nxor_result:
                    alu_andi         ? andi_result:
                    32'd0;
```

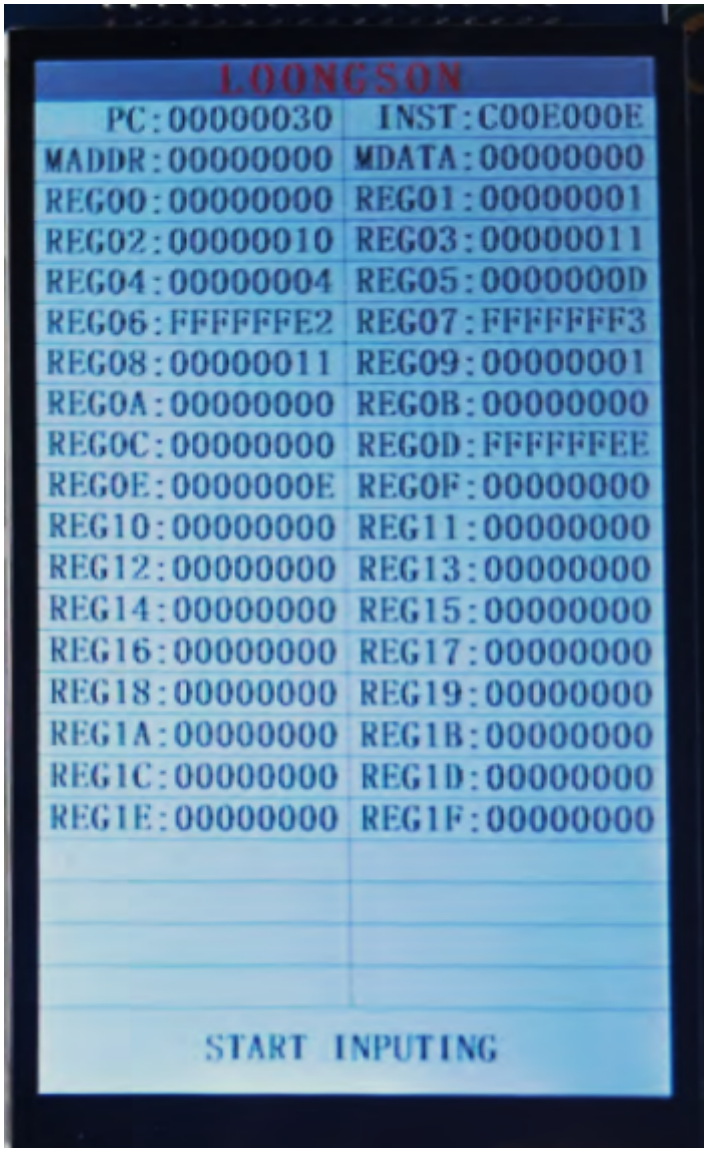
3. **对inst_rom.v文件的修改**

   该模块用来存储和提供CPU的指令集，我们修改指令，使新添加的指令能够运行并显示结果

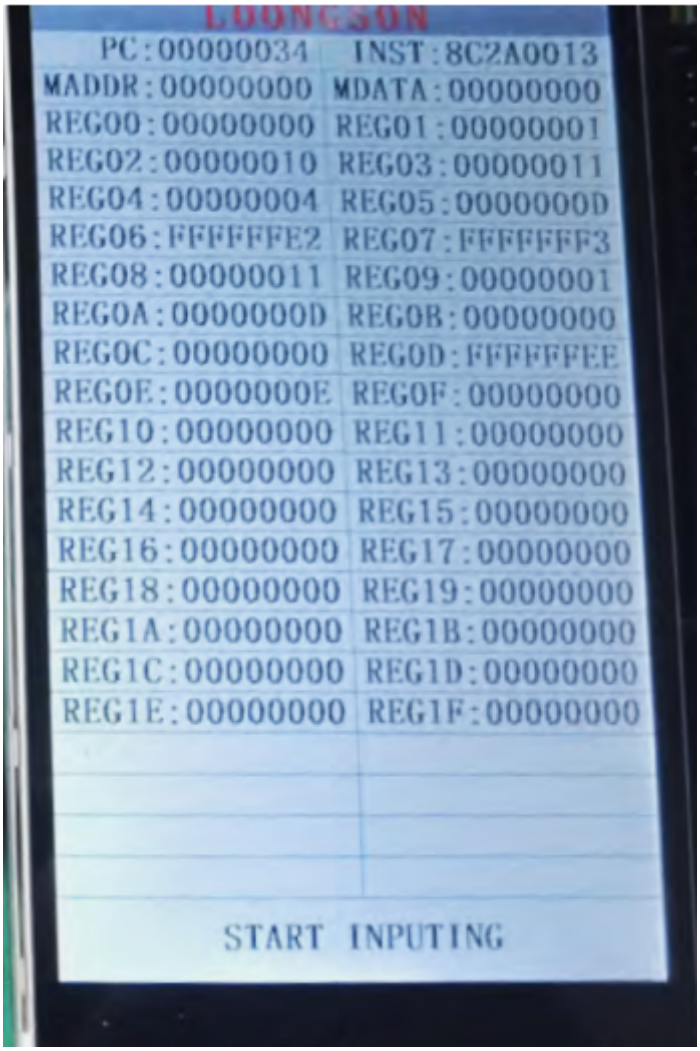
```
assign inst_rom[0] = 32'hC0080380; //addi
assign inst_rom[1] = 32'h00C76831; //xnor
assign inst_rom[2] = 32'hC00E000E; //hui
```

实验结果

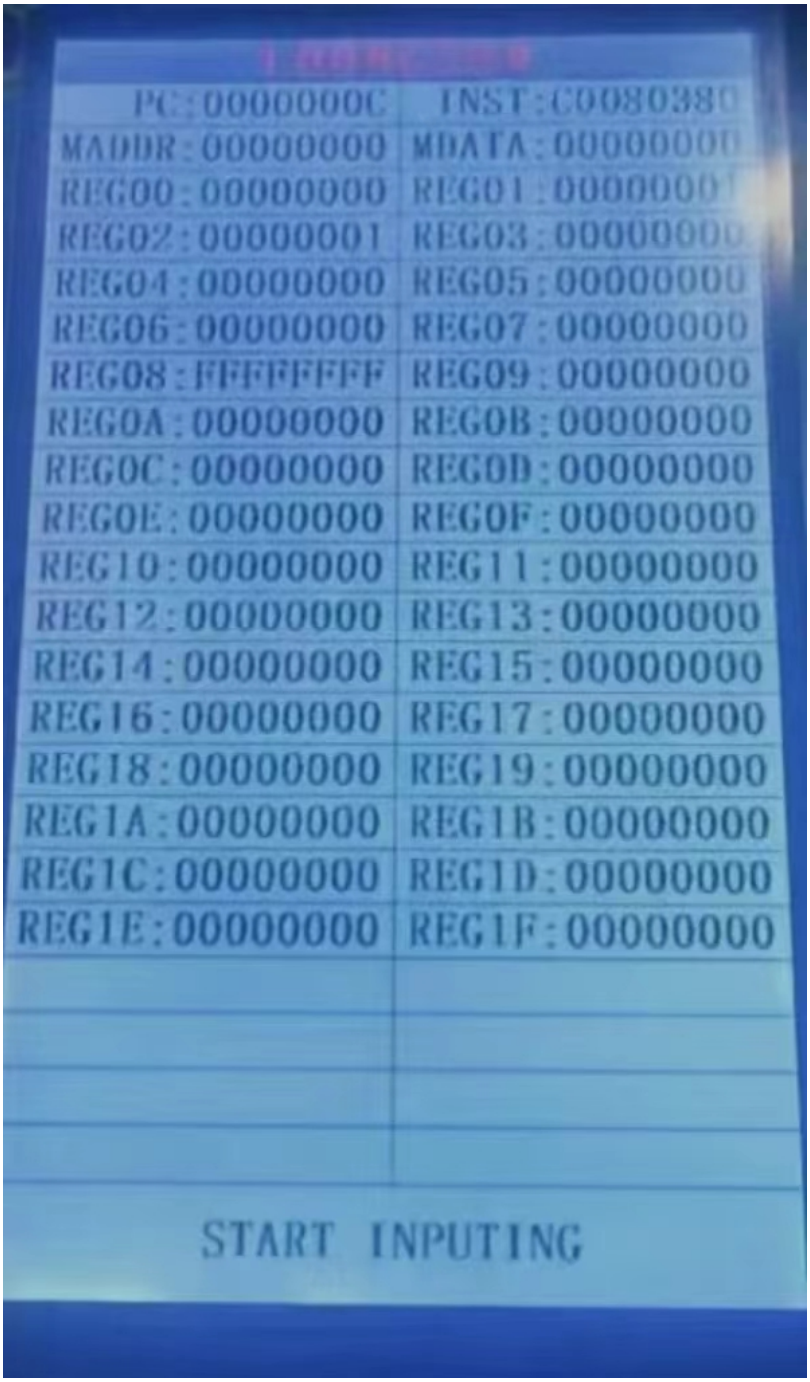
1. 同或（XONR）操作运行的指令编码为`0x00C76831`对应的汇编语句为`nxor $13, $6, $7`。此处$6的值为0xFFFFFFE2，$7的值为0xFFFFFFF3，运算结果存入$13为0xFFFFFFEE。



2. 低位装载（HUI） 操作运行的指令编码为`0xC00E000E`对应的汇编语句为`hui $14`表示将立即数14加载到寄存器$14，结果为0x0000000E。



3. 立即数与（ANDI） 操作的指令是`hc0080380`，对应的汇编语句为`andi $2,$1,1`，将寄存器$1中的值与立即数1相与，寄存器$2得到最终结果为0x00000001。



实验感想

在本次实验中，我深入理解了MIPS指令结构以及常用指令的功能和编码，通过归纳分类，进一步加深了对这些指令的理解。此外，我学习并熟悉了MIPS体系的处理器结构，特别是延迟槽和哈佛结构的概念。通过设计和实现单周期CPU，我掌握了其原理和设计方法，并在实际操作中运用了Verilog语言进行电路设计，增强了编程能力和硬件设计技能。

在实验过程中，我成功复现了单周期CPU的设计，并在实验箱上验证了其功能的正确性。此外，我分别添加了一种R型指令和两种I型指令，进一步扩展了CPU的指令集，并通过实验箱验证了这些新增指令的正确性。这不仅加深了我对MIPS指令集和CPU结构的理解，也提升了我解决实际问题的能力。

总体而言，本次实验让我在理论知识和实践操作之间建立了更加紧密的联系，为后续多周期CPU的设计打下了坚实的基础。同时，通过不断的调试和优化，我的Verilog编程能力和硬件调试技能得到了显著提高，为未来的学习和研究积累了宝贵的经验。