计算机组成原理第 6 次实验报告

实验名称：多周期 CPU 实现

学号：2313546　　姓名：蒋枘言　　班次：1078

一、实验目的

1. 在单周期 CPU 实验完成的提前下，理解多周期的概念。
2. 熟悉并掌握多周期 CPU 的原理和设计。
3. 进一步提升运用 verilog 语言进行电路设计的能力。
4. 为后续实现流水线 cpu 的课程设计打下基础。

二、实验内容说明

1. 多周期 CPU 实验使用同步 IP 核构造 data_ram 和 inst_rom，原始 source_code 中的同名.v 文件和 ngc 文件不要导入到项目。

2. 多周期 CPU 运行的指令在 inst_rom 中，这里面的指令须导入 coe 文件。

3. 请把 ALU 实验中添加的三个运算，自行定义类似 MIPS 指令格式的指令，把对应的指令和功能增加到多周期 CPU 中，并自行在 coe 文件中添加指令，然后进行运行验证（仿真波形验证或实验箱验证即可）。

4. 实验报告中可以不放原理图，关于验证结果的图片（仿真图片或实验箱图片）需要仔细介绍图中的信息和对指令验证的情况。

三、实验步骤

打开 Vivado，导入必要的文件，并创建 IP 核。现在需要添加指令，下面以自定义指令 **ADD_SHL**（移位加法，R 型指令）为例：

指令格式：**ADD_SHL $5, $4, $2**

功能：$5 = $4 + ($2 << 1)，将寄存器$4 的值与$2 左移 1 位后的结果相加，存入$5。

机器码结构：　　opcode: 0x00（R 型指令）

rs: $4（寄存器编号 4 → 5'b00100）

rt: $2（寄存器编号 2 → 5'b00010）

rd: $5（寄存器编号 5 → 5'b00101）

shamt: 0（移位量，此处未使用 → 5'b00000）

funct: 0x28（自定义功能码 6'b101000 → 0x28）

机器码计算：

opcode	rs	rt	rd	shamt	funct
0b000000	0b00100	0b00010	0b00101	0b00000	0b101000

十六进制：0x00822828

二进制：0000 0000 1000 0010 0010 1000 0010 1000

（一）修改译码模块 decode.v

在 decode.v 中识别 ADD_SHL 指令，并生成对应的控制信号。

```
// 在现有的指令译码部分添加以下代码
wire inst_ADD_SHL;
assign inst_ADD_SHL = op_zero                 // R 型指令（opcode=0）
                    & (funct == 6'b101000)    // funct=0x28（自定义功能码）
                    & (rs == 5'd4)            // rs=$4
                    & (rt == 5'd2)            // rt=$2
                    & (sa == 5'd0);           // shamt=0（未使用移位量）
```

将 ADD_SHL 指令映射到 ALU 的加法（inst_add）和左移（inst_sll）操作。

```
// 修改 alu_control 信号生成逻辑
assign alu_control = {
    inst_add | inst_ADD_SHL,    // 加法使能（新增 ADD_SHL）
    inst_sub,
    inst_slt,
    inst_sltu,
    inst_and,
    inst_nor,
    inst_or,
    inst_xor,
    inst_sll | inst_ADD_SHL,   // 左移使能（新增 ADD_SHL）
    inst_srl,
    inst_sra,
    inst_lui};
```

确保 ADD_SHL 的操作数正确传递到 ALU：

```
// 修改 alu_operand1 和 alu_operand2 的选择逻辑
assign alu_operand1 = inst_ADD_SHL ? rs_value :  // 使用 rs 的值（$4）
                      inst_shf_sa ? {27'd0, sa} : rs_value;
assign alu_operand2 = inst_ADD_SHL ? (rt_value << 1) : // rt 左移 1 位（$2 << 1）
                      inst_imm_sign ? {{16{imm[15]}}, imm} : rt_value;
```

（二）修改 ALU 模块 alu.v

在 ALU 中支持加法与左移的组合操作（rs + (rt << 1)）。

```
// 在 alu.v 中添加以下代码
wire [31:0] add_shl_result;
assign add_shl_result = alu_src1 + (alu_src2 << 1);  // rs + (rt << 1)
// 修改 alu_result 的选择逻辑
assign alu_result = (alu_add | alu_ADD_SHL) ? add_shl_result :  // 新增 ADD_SHL 结果
                    alu_sub     ? sub_result :
                    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 :
                    32'd0;
```

在 alu.v 中定义 alu_ADD_SHL 控制信号（如果尚未定义）：

```
// 在 alu_control 解码部分添加
wire alu_ADD_SHL;
assign alu_ADD_SHL = alu_control[11] & alu_control[3];  // 同时触发加法和左移
```

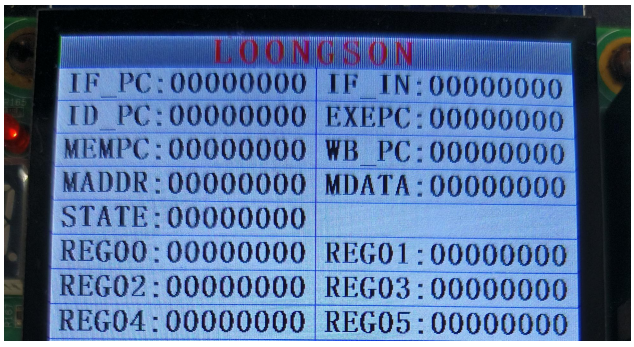
四、实验结果分析

指令执行顺序为：

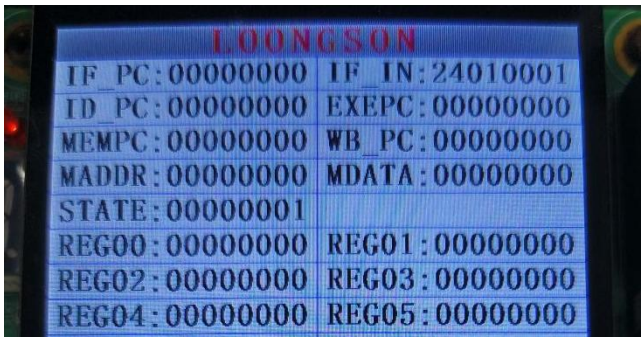
```
addiu $1, $0,#1
sll $2,$1,#4
addu $3,$2,$1
srl $4, $2,#2
add_shl $5, $4, $2
```

上箱验证，得到如下结果：

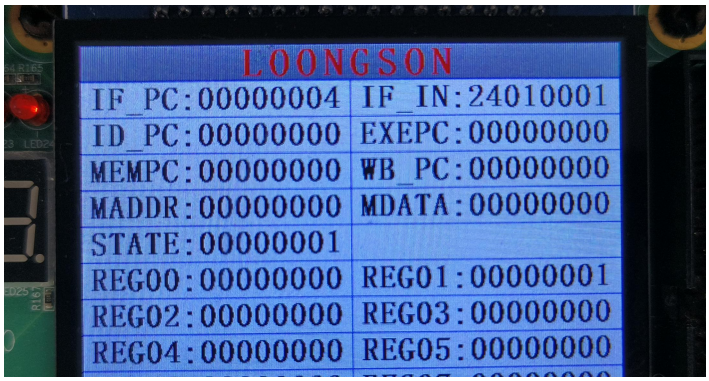
最开始所有值均为 0。



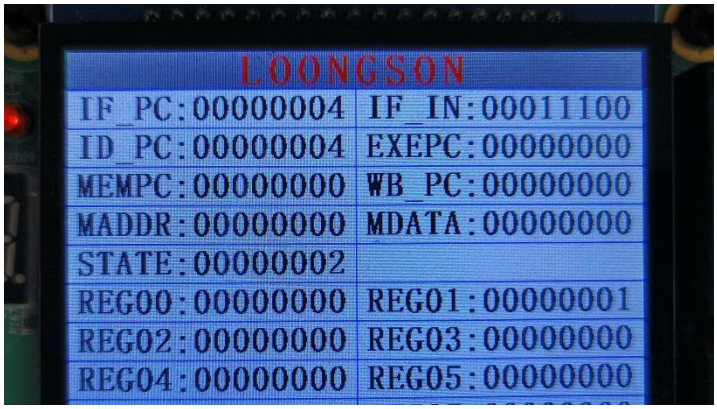
接着 IF 的指令变成了 24010001，即指令 addiu $1, $0,#1。IF 程序计数器保持 00000000。



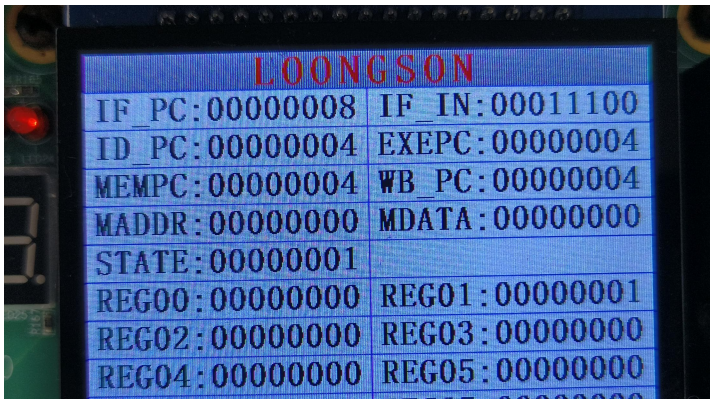
周期继续进行，下一个 STATE=1 时，IF 程序计数器变成了 00000004，说明要开始取 00000004 地址的指令。此时寄存器$1 的已经加 1，变成了 00000001。



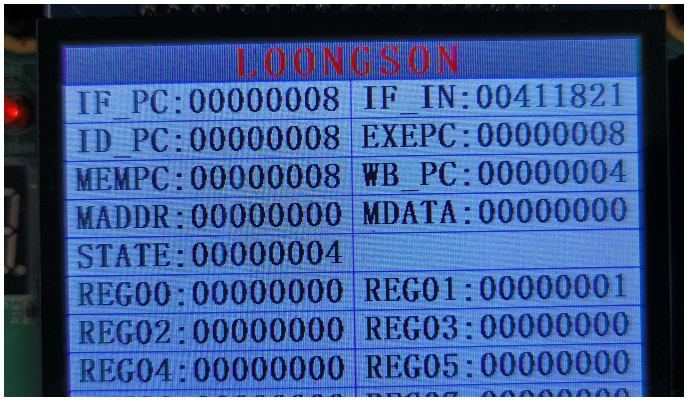
继续往后走一个时钟周期（STATE=2），这时 IF 的指令已经变成了 00011100（即 sll $2,$1,#4）。



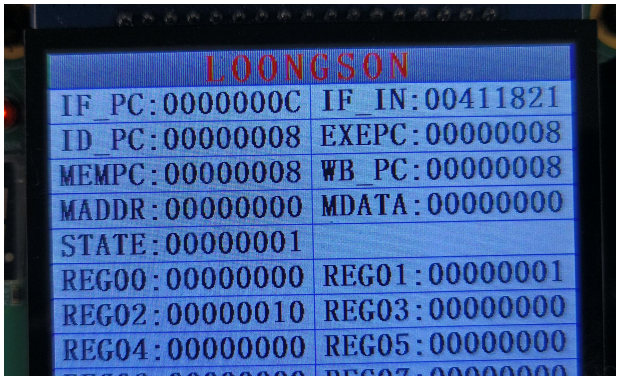
周期继续进行，下一个 STATE=1 时，IF 程序计数器变成了 00000008，说明要开始取 00000008 地址的指令。



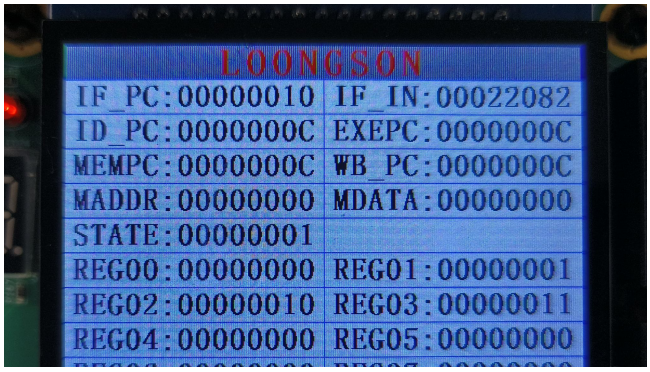
这时取到了 00000008 处的指令 00411821（即 addu $3,$2,$1）。



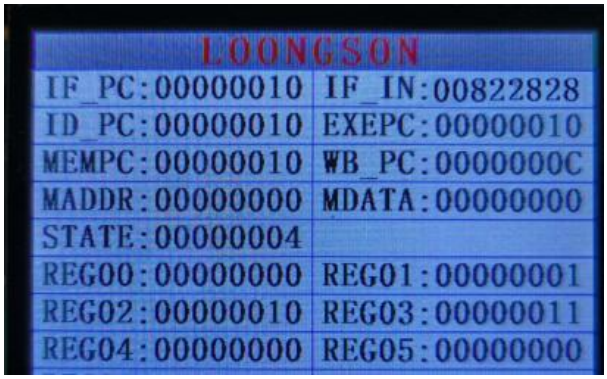
这时指令 sll $2,$1,#4 执行完毕，寄存器$2 的值变为了 00000010。



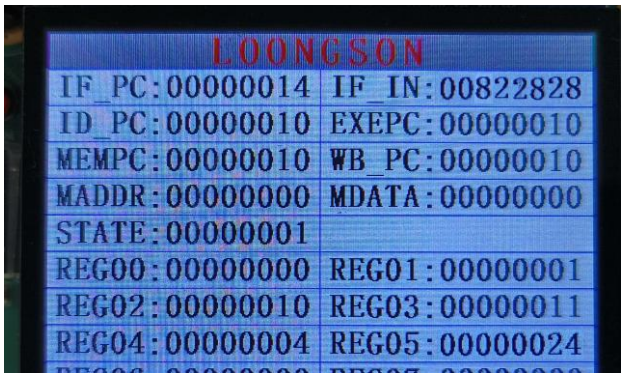
后面指令的执行过程类似，接着读取 0000000C 处的指令 00022082（srl $4, $2,#2）。此时指令 addu $3,$2,$1 执行完毕，寄存器$3 的值变为了 00000011。



现在到了我们添加的指令：IF 的指令已经变成了 00822828（add_shl $5, $4, $2），执行过程和之前完全一样。



$4 的值为 00000004，$2 的值为 00000010，右移一位得到 00000020，与$4 相加应得到 00000024。指令 add_shl $5, $4, $2 执行完毕后，寄存器$5 的值变成了 00000024，完全正确。



五、总结感想

通过本次多周期 CPU 设计实验，更加理解了指令执行流程与硬件控制逻辑的协同运作。从译码到写回，每个过程都直观地呈现了出来。最终还成功运行自定义指令。为后续设计五级流水线 CPU 打下基础。