组成原理课程第 五 次实报告

实验名称：单周期 CPU 实验

学号： 2310420 姓名： 王晶 班次： 周二三四节

实验目的

存储器实现

1.了解只读存储器 ROM 和随机存取存储器 RAM 的原理。

2. 理解 ROM 读取数据及 RAM 读取、写入数据的过程。

3. 理解计算机中存储器地址编址和数据索引方法。

4. 理解同步 RAM 和异步 RAM 的区别。

5. 掌握调用 xilinx 库 IP 实例化 RAM 的设计方法。

6. 熟悉并运用 verilog 语言进行电路设计。

7. 为后续设计 cpu 的实验打下基础。

单周期 CPU 实现

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

请根据实验指导手册完成 ROM 存储器实验和单周期 CPU 实验，并撰写实验总结，

1、ROM 存储器实验请完成验证，总结收获，并对比跟之前的同步异步 RAM 实验

有何不同，分析总结原因。

2、单周期 CPU 实验，原理图应基于实验指导手册中的图 7.1，在分析表 7.4 中指令执行过程时，可以在图 7.1 基础上辅助画线表示执行过程。

3、R 型指令和 I 型指令挑两条分析总结执行过程，J 型指令就 1 条，请直接分析总结执行过程。注意，这些指令已经在 inst_rom.v 里面写好，所以请找到对应的指令，逐个分析。从指令的二进制编码开始，分析介绍代码是如何一步一步完成运算并执行的。

4、（提高要求，不强求做出来）把 ALU 实验中添加的三个指令，自行添加到这个单周期 CPU 中，注意指令码只要跟现有的不冲突就行，不必限制在标准的 MIPS 指令格式。

实验原理图

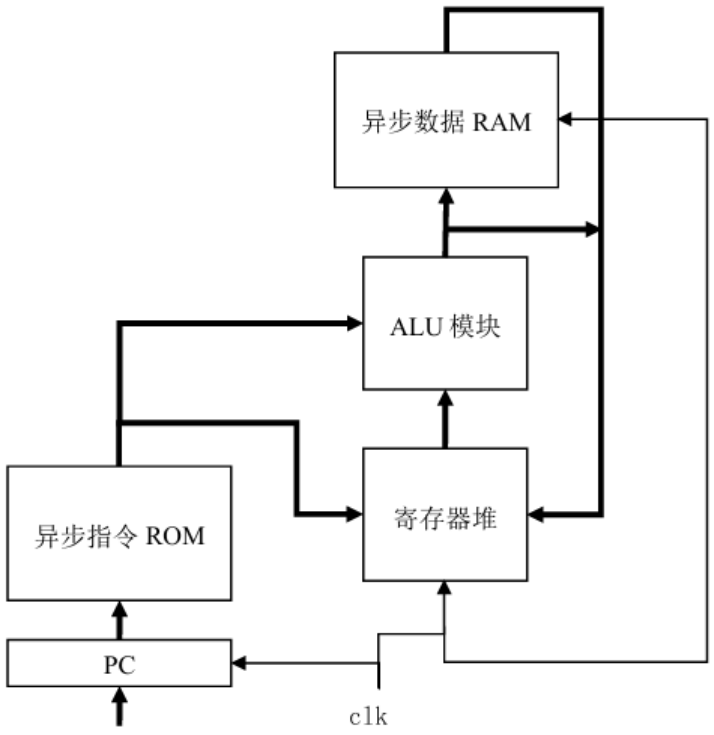
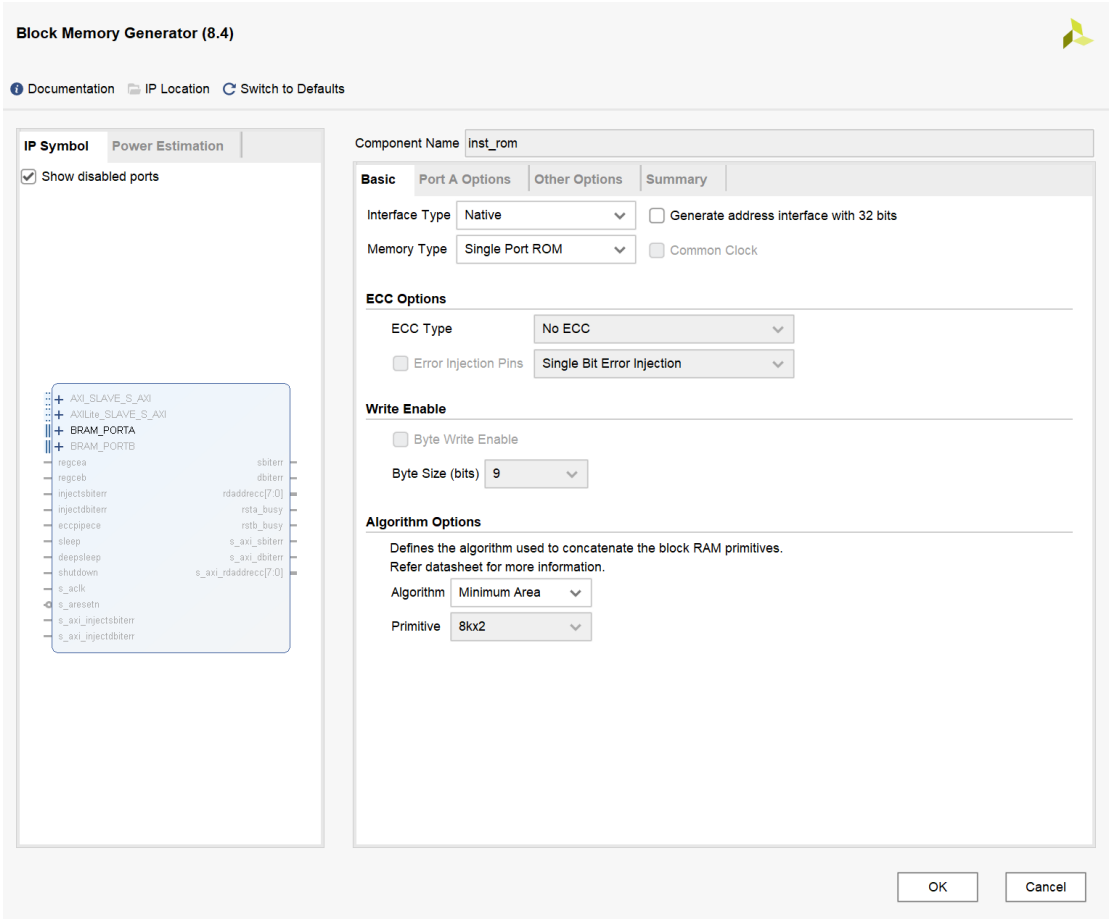


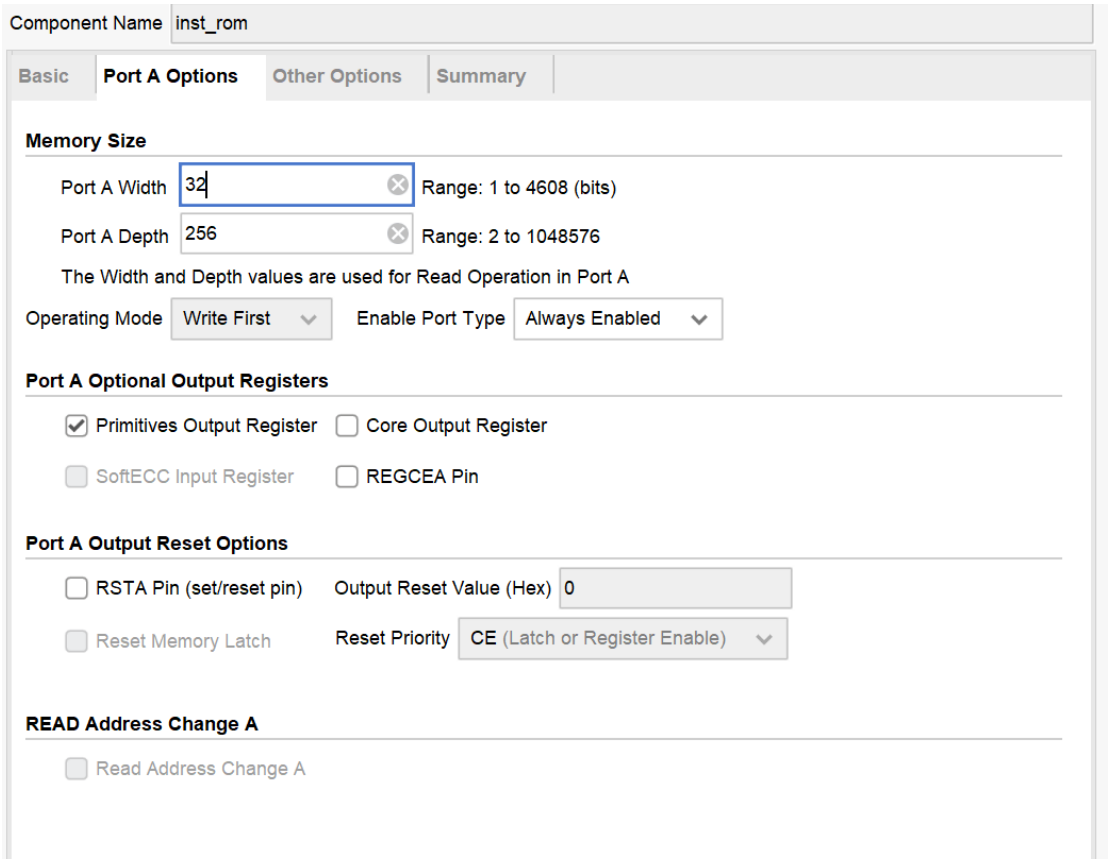
图 7.1 单周期 CPU 的大致框图

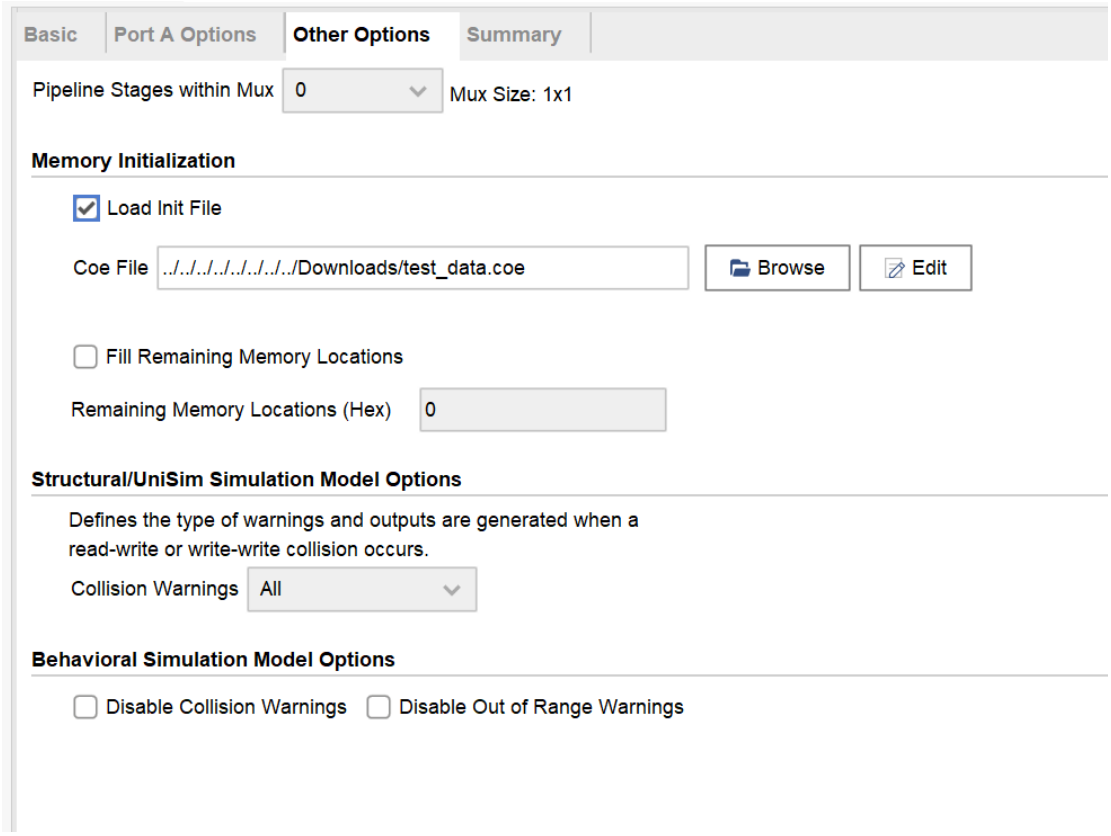
同步 ROM 实验

实验步骤

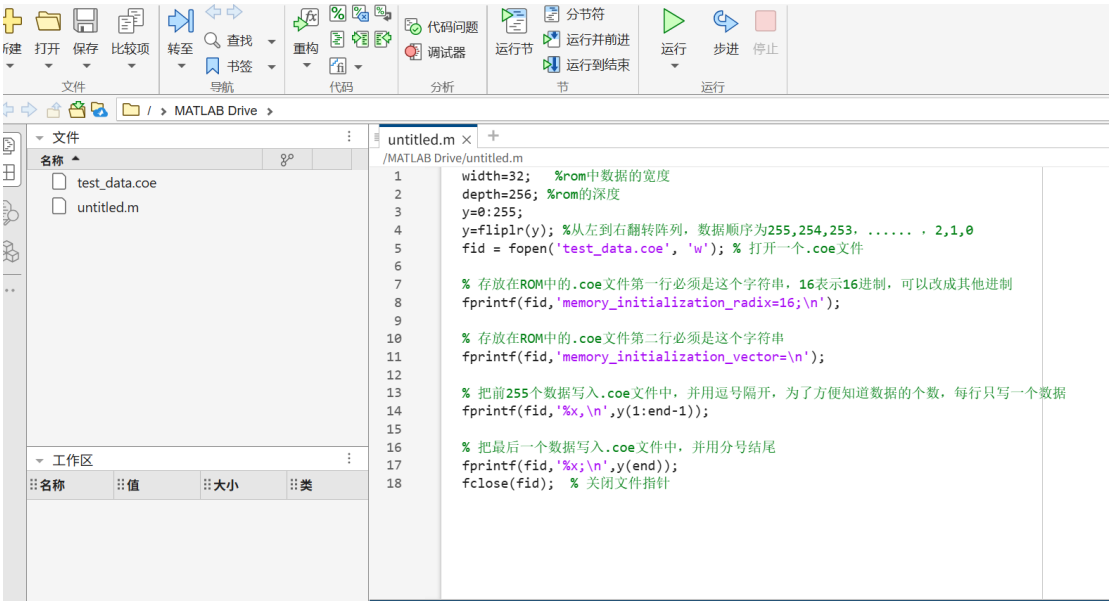
首先要设置 IP 核，按照如下的标准







在 other options 中，我们需要建立一个.coe 文件，我们利用 matlab 生成

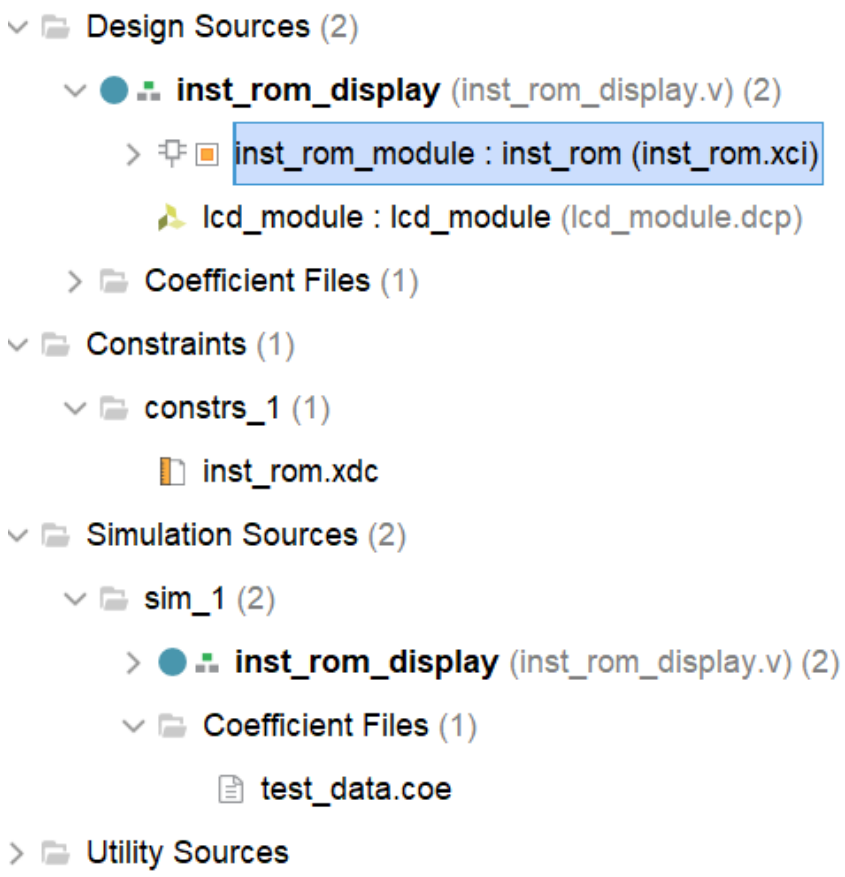


生成的.coe 文件如下

```
memory initialization radix=16;
memory initialization vector=
ff,
fe,
fd,
fc,
fb,
fa,
f9,
f8,
f7,
f6,
f5,
f4,
f3,
f2,
f1,
f0,
ef,
ee,
ed,
ec,
eb,
ea,
e9,
e8,
e7,
e6,
e5,
e4,
e3,
e2,
e1,
e0,
df,
de,
dd,
dc,
db,
da,
d9,
d8,
d7,
```

16 进制数，位数为 8 位

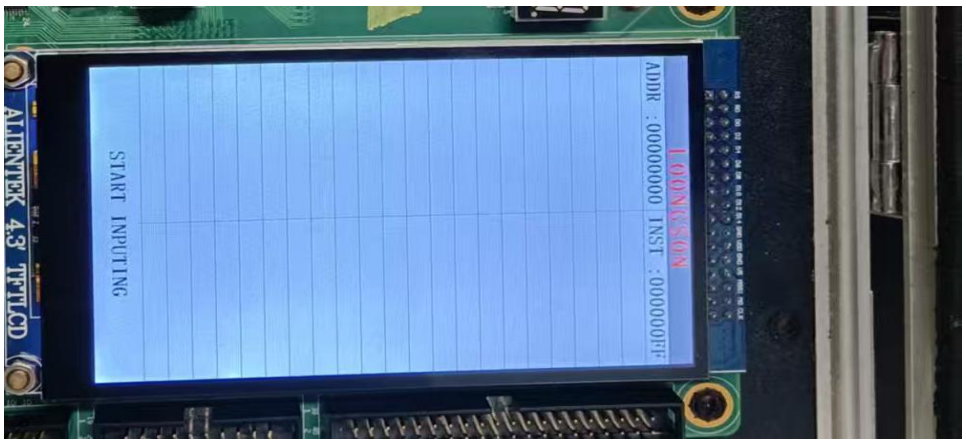
设置 IP 核之后，我们建立 inst_rom_display.v 文件核 inst_rom.xdc 约束文件

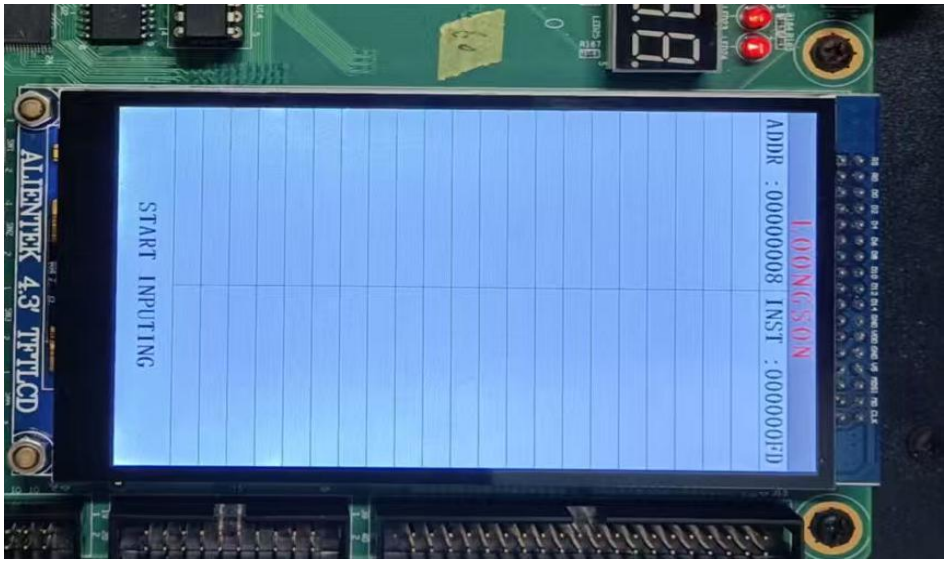


生成 bitstream 文件后，进行上箱验证

实验结果

实验结果如下





地址 00000000 存储的指令为 000000FF

地址 00000008 存储的指令为 000000FD

地址 00000010 存储的指令为 000000FB

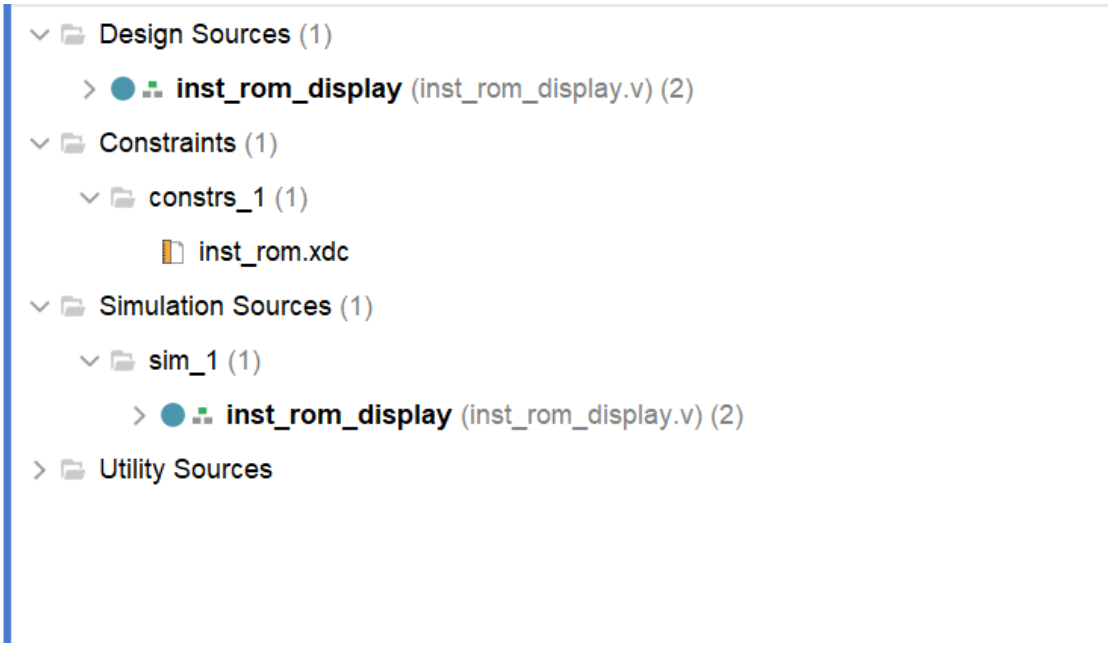
和我们的.coe 文件中设置的数据一致，可知结果正确！

异步 ROM

实验步骤

异步 ROM 不需要借助 IP 核，只需要普通的模块文件即可

我们将源代码导入



设置 inst_rom_display.v 文件，inst_rom.xdc 文件 inst_rom_display.v 文件

生成 bitstream 文件后，进行上箱验证

实验结果

实验结果如下

地址 00000000 存储的指令为 24010001

地址 00000008 存储的指令为 00411821

```
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
assign inst_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4    | $2 = 0000_0010H
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1    | $3 = 0000_0011H
assign inst_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2    | $4 = 0000_0004H
assign inst_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4    | $5 = 0000_000DH
assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
assign inst_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2    | $6 = FFFF_FFE2H
assign inst_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3    | $7 = FFFF_FFF3H
assign inst_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6    | $8 = 0000_0011H
assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0)  | Mem[0000_001CH] = 0000_0011H
assign inst_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7    | $9 = 0000_0001H
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2    | 跳转到指令34H
assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4    | 不执行
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1)  | $10 = 0000_000DH
assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3    | 不跳转
assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1    | $11 = 0000_0000H
assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0)  | Men[0000_001CH] = 0000_0000H
assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0)  | Mem[0000_0010H] = 0000_0004H
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12      | [R12] = 000C_0000H
assign inst_rom[19] = 32'h08000000; // 4CH: j     00H          | 跳转指令00H
```

和我们在 display 文件中硬编码的指令一致，即为 addiu $1 ,$0 和 addu $3 ,$2，结果正确！

总结

同步异步 RAM 和同步异步 ROM 的区别：

1. RAM 是随机访问存储器,可以读写数据,数据可以任意更改,掉电后数据消失
2. ROM 是只读的存储器,只能读取数据,,内容预先写好,运行时不能随便更改
3. RAM 实验可以仿真观察也可以上实验箱验证,但是 ROM 实验只能烧录生成 bitstream 文件后上实验箱验证
4. RAM 实验读和写都要测试,但是 ROM 实验主要测试读操作
5. RAM 实验需要读/写信号选通,但是 ROM 实验只需要读信号
6. RAM 实验主要是测试读写正确性和时序稳定性；ROM 实验主要是测试地址映射正确性和数据一致性。

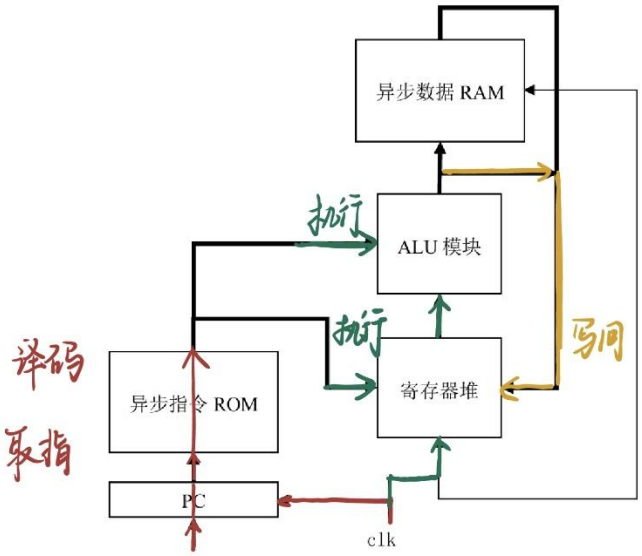
单周期 CPU 实现

实验步骤

将我们的源码文件都导入到 vivado 中

代码解释

（1）SLT 指令



取指:

1. pc 寄存器当前值为 32'h00000028。
2. inst_addr 被赋值为 pc 的值，所以 inst_addr = 32'h00000028。inst_rom_module 使用 inst_addr 的低位 inst_addr[6:2]（即 01010 或十进制 10）作为地址输入。inst_rom_module 从地址 10 中读出指令 32'h00C7482A，并将其输出到 inst 信号线。
3. 计算下一条顺序执行的指令地址：seq_pc = pc + 4 = 32'h0000002C。
4. 计算是否发生跳转（jbr_taken）。inst_J、inst_BEQ、inst_BNE 这些信号都为假。jbr_taken 信号通过逻辑或连接这些指令信号，所以 jbr_taken 在此时为假。
5. next_pc 根据 jbr_taken 的值被赋值：next_pc = jbr_taken ? jbr_target : seq_pc。因为 jbr_taken 为假，next_pc 被赋值为 seq_pc 的值，即 32'h0000002C。这个值会在下一个时钟上升沿加载到 pc 寄存器中。
6. cpu_pc 输出当前 PC 值 32'h00000028，cpu_inst 输出取出的指令 32'h00C7482A

译码:

inst 信号（32'h00C7482A）被分解：

op = inst[31:26] = 6'b000000 (操作码)

rs = inst[25:21] = 5'b00110 (寄存器 $6 的地址)

rt = inst[20:16] = 5'b00111 (寄存器 $7 的地址)

rd = inst[15:11] = 5'b01001 (寄存器 $9 的地址)

sa = inst[10:6] = 5'b00000 (移位量，slt 不用)

funct = inst[5:0] = 6'b101010 (功能码)

控制逻辑根据 op 和 funct 识别指令类型并生成相应的控制信号：

op_zero (~(|op)) 为真，因为 op 全为 0。

sa_zero (~(|sa)) 为真，因为 sa 全为 0。

inst_SLT 信号被置为真 (1'b1)。

其他指令类型信号 (如 inst_ADDU, inst_ADDIU, inst_LW, inst_SW 等) 都为假 (1'b0)。

寄存器堆根据 rs (5'b00110, $6) 和 rt (5'b00111, $7) 地址读出对应的寄存器值： 寄存器 $6 的值被读出到 rs_value 信号。寄存器 $7 的值被读出到 rt_value 信号。

确定 ALU 的输入操作数 (alu_operand1, alu_operand2)： inst_shf_sa (inst_SLL | inst_SRL) 为假。alu_operand1 被赋值为 rs_value。

inst_imm_sign (inst_ADDIU | inst_LUI | inst_LW | inst_SW) 为假。alu_operand2 被赋值为 rt_value。

确定 ALU 的控制信号 (alu_control)：

因为 inst_slt (对应 inst_SLT) 为真，alu_control 中与 inst_slt 对应的位为 1，其他位为 0。根据代码 assign alu_control = {inst_add, inst_sub, inst_slt, ..., inst_lui};，alu_control 的值为 12'b0010_0000_0000 (第三个位为 1)。这个信号告诉 ALU 执行“小于则置位”操作。

执行:

根据 alu_control 信号，ALU 执行带符号的“小于”比较操作 (slt)：比较 alu_operand1 (来自寄存器 $6 的值) 和 alu_operand2 (来自寄存器 $7 的值)。

ALU 的输出 alu_result 被设置为：

如果 (signed)rs_value < (signed)rt_value 为真，则 alu_result = 32'd1。

如果 (signed)rs_value < (signed)rt_value 为假，则 alu_result = 32'd0。

访存:

inst_SW 信号为假 (1'b0)。数据存储器的写使能信号为低电平，数据存储器不会执行写操作。inst_LW 信号为假 (1'b0)。数据存储器也不会执行读操作。

写回:

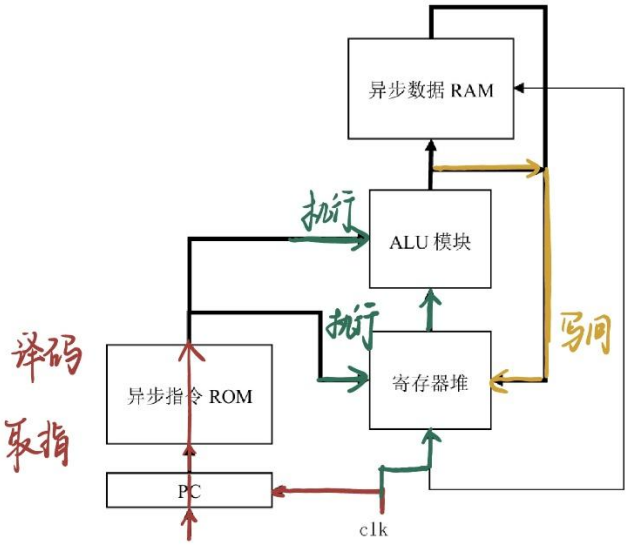
确定写入寄存器的地址 (rf_waddr)： rf_waddr 被赋值为 rd 的值，即 5'b01001 (寄存器 $9 的地址)。

确定写入寄存器的数据 (rf_wdata)： rf_wdata 被赋值为 alu_result 的值。

确定寄存器写使能 (rf_wen)： rf_wen 信号被置为真 (1'b1)。

在下一个时钟上升沿，regfile 模块会将 rf_wdata (即 alu_result 的值) 写入由 rf_waddr 指定的寄存器中，即写入寄存器 $9。

（2）AND 指令



取指：

pc 寄存器当前值为 32'h0000003C。

inst_addr 被赋值为 pc 的值，所以 inst_addr = 32'h0000003C。

inst_rom_module 使用 inst_addr 的低位 inst_addr[6:2]（即 01111 或十进制 15）作为地址输入。

inst_rom_module 从地址 15 中读出指令 32'h00415824，并将其输出到 inst 信号线。

计算下一条顺序执行的指令地址：seq_pc = pc + 4 = 32'h00000040。

计算是否发生跳转（jbr_taken）。inst_J、inst_BEQ、inst_BNE 这些信号都为假，jbr_taken 信号为假。

next_pc 被赋值为 seq_pc 的值，即 32'h00000040。这个值会在下一个时钟上升沿加载到 pc 寄存器中。

cpu_pc 输出当前 PC 值 32'h0000003C，cpu_inst 输出取出的指令 32'h00415824（用于外部显示或调试）。

译码：

inst 信号（32'h00415824）被分解：

op = inst[31:26] = 6'b000000（操作码）

rs = inst[25:21] = 5'b00010（寄存器 $2 的地址）

rt = inst[20:16] = 5'b00001（寄存器 $1 的地址）

rd = inst[15:11] = 5'b01011（寄存器 $11 的地址）

sa = inst[10:6] = 5'b00000（移位量，and 不用）

funct = inst[5:0] = 6'b100100（功能码）

控制逻辑根据 op 和 funct 识别指令类型并生成相应的控制信号：

op_zero（~(|op)）为真，因为 op 全为 0。

sa_zero（~(|sa)）为真，因为 sa 全为 0。

inst_AND 信号被置为真 (1'b1)。

其他指令类型信号 (如 inst_SLT, inst_ADDIU, inst_LW, inst_SW 等) 都为假 (1'b0)。

寄存器堆 (regfile_module) 根据 rs (5'b00010, $2) 和 rt (5'b00001, $1) 地址读出对应的寄存器值： 寄存器 $2 的值被读出到 rs_value 信号。寄存器 $1 的值被读出到 rt_value 信号。

确定 ALU 的输入操作数 (alu_operand1, alu_operand2)：

alu_operand1 被赋值为 rs_value。alu_operand2 被赋值为 rt_value。

确定 ALU 的控制信号 (alu_control)：

因为 inst_and (对应 inst_AND) 为真，alu_control 中与 inst_and 对应的位为 1，其他位为 0。根据代码 assign alu_control = {inst_add, inst_sub, inst_slt, ..., inst_and, ..., inst_lui};，alu_control 的值为 12'b0000_1000_0000 (第五个位为 1)。这个信号告诉 ALU 执行按位逻辑 AND 运算。

执行:

alu_module 接收 alu_operand1 (rs_value)，alu_operand2 (rt_value)，以及 alu_control (12'b0000_1000_0000)。

根据 alu_control 信号，ALU 执行按位逻辑 AND 运算：alu_result = alu_operand1 & alu_operand2 (即寄存器 $2 的值与寄存器 $1 的值进行按位 AND 运算)。

访存:

inst_SW 信号为假 (1'b0)。数据存储器的写使能信号为低电平，数据存储器不会执行写操作。数据存储器也不会执行读操作。

写回:

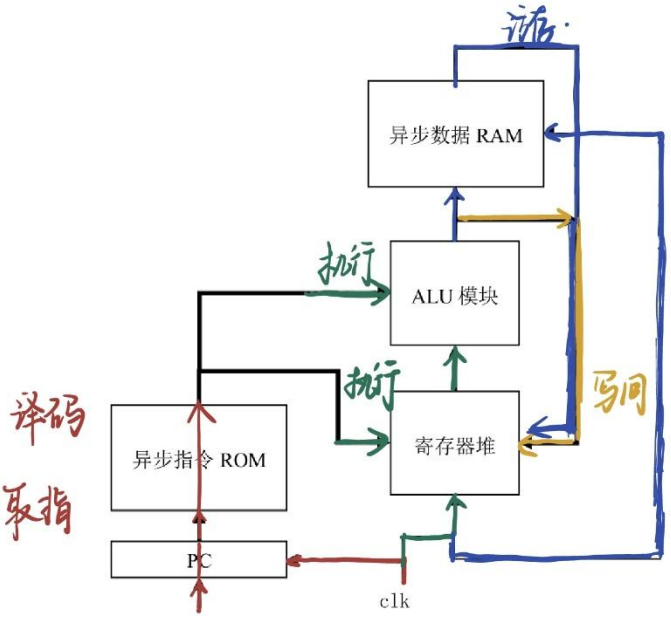
确定写入寄存器的地址 (rf_waddr)： rf_waddr 被赋值为 rd 的值，即 5'b01011 (寄存器 $11 的地址)。

确定写入寄存器的数据 (rf_wdata)： rf_wdata 被赋值为 alu_result 的值 (在执行阶段计算出的按位 AND 结果)。

确定寄存器写使能 (rf_wen)： rf_wen 信号被置为真 (1'b1)。

在下一个时钟上升沿，regfile 模块会将 rf_wdata (即 alu_result 的值) 写入由 rf_waddr 指定的寄存器中，即写入寄存器 $11

（3）LW 指令



取指:

当前 pc 的值为 34H (52)。

inst_addr 被赋值为 pc 的值，即 34H。

指令存储器 inst_rom_module 根据地址 inst_addr[6:2] (即 52[6:2] = 11010_2 = 26_10) 读取指令

从 inst_rom 的地址 13 中读取到的指令 32'h8C2A0013 被输出到 inst 信号。

cpu_pc 输出当前的程序计数器值 34H。

cpu_inst 输出读取到的指令 32'h8C2A0013。

seq_pc 计算下一条指令的地址，seq_pc = pc + 4 = 38H。

由于这是一个 lw 指令，没有发生跳转，因此 jbr_taken 为假，next_pc 被赋值为 seq_pc，即 38H，等待下一个时钟上升沿更新 pc。

译码:

指令 inst (32'h8C2A0013) 的各个字段被解析出来：

op = inst[31:26] = 8C_16 = 100011_2

rs = inst[25:21] = 01_16 = 00001_2 (对应寄存器 $1)

rt = inst[20:16] = 0A_16 = 01010_2 (对应寄存器 $10)

imm = inst[15:0] = 0013_16 = 0000 0000 0001 0011_2 = 19_10

offset = imm = 19_10

根据 op 的值 (100011_2)，inst_LW 信号被置为高电平 (assign inst_LW = (op == 6'b100011);)。

寄存器堆 rf_module 接收到读地址 rs ($1) 和 rt ($10)。

在组合逻辑下，寄存器 $1 的值被读取到 rs_value，寄存器 $10 的值被读取到 rt_value。

执行:

inst_add 信号被置为高电平表明需要进行加法运算来计算内存地址。

inst_imm_sign 信号被置为高电平表示立即数需要进行符号扩展。

sext_imm 将 imm (19) 进行符号扩展。由于 19 是正数，符号位为 0，所以 sext_imm 的值为 32'h00000013。

alu_operand1 被赋值为 rs_value (寄存器 $1 的值)。

alu_operand2 被赋值为 sext_imm (32'h00000013)。

alu_control 信号根据需要执行加法运算的指令 (inst_add) 被设置为对应的控制位。

ALU alu_module 执行加法运算：alu_result = alu_operand1 + alu_operand2 (即 $1 的值 + 19)。

访存:

dm_wen 为低电平，数据存储器不会进行写操作。

dm_addr 被赋值为 alu_result，即计算出的内存地址 ($1 的值 + 19)。

数据存储器 data_ram_module 根据地址 dm_addr[6:2] 读取内存中的数据，并将读取到的数据输出到 dm_rdata。

写回:

inst_wdest_rt 为高电平表示结果需要写回到 rt 寄存器。

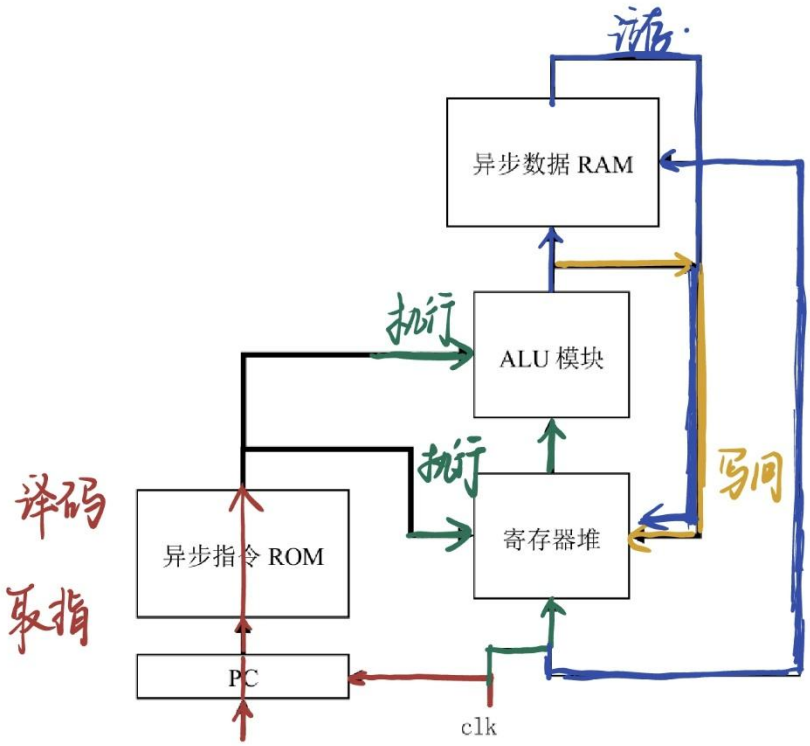
rf_wen 被置为高电平允许写回操作。

rf_waddr 被赋值为 rt，即 $10。

rf_wdata 被赋值为 dm_rdata，即从内存中读取到的值。

在下一个时钟上升沿到来时，dm_rdata 的值将被写入到寄存器堆的 $10 寄存器中。

（4）SW 指令



取指:

当前程序计数器 pc 的值为 40H (64)。

inst_addr 被赋值为 pc 的值，即 40H。

指令存储器 inst_rom_module 根据地址 inst_addr[6:2] (即 64[6:2] = 100000_2 = 32_10) 读取指令。

从 inst_rom 的地址 16 中读取到的指令 32'hAC0B001C 被输出到 inst 信号。

cpu_pc 输出当前的程序计数器值 40H。

cpu_inst 输出读取到的指令 32'hAC0B001C。

seq_pc 计算下一条指令的地址，seq_pc = pc + 4 = 44H。

jbr_taken 为假，next_pc 被赋值为 seq_pc，即 44H，等待下一个时钟上升沿更新 pc。

译码:

指令 inst (32'hAC0B001C) 的各个字段被解析出来：

op = inst[31:26] = AC_16 = 101011_2

rs = inst[25:21] = 00_16 = 00000_2 (对应寄存器 $0)

rt = inst[20:16] = 0B_16 = 01011_2 (对应寄存器 $11)

imm = inst[15:0] = 001C_16 = 0000 0000 0001 1100_2 = 28_10

offset = imm = 28_10

inst_SW 信号被置为高电平

寄存器堆 rf_module 接收到读地址 rs ($0) 和 rt ($11)。

在组合逻辑下，寄存器 $0 的值（通常为 0）被读取到 rs_value，寄存器 $11 的值被读取到 rt_value。

执行:

inst_add 信号被置为高电平表明需要进行加法运算来计算内存地址。

inst_imm_sign 信号被置为高电平表示立即数需要进行符号扩展。

sext_imm 将 imm (28) 进行符号扩展。由于 28 是正数，符号位为 0，所以 sext_imm 的值为 32'h0000001C。

alu_operand1 被赋值为 rs_value (寄存器 $0 的值，通常为 0)。

alu_operand2 被赋值为 sext_imm (32'h0000001C).

alu_control 信号根据需要执行加法运算的指令 (inst_add) 被设置为对应的控制位。

ALU alu_module 执行加法运算：alu_result = alu_operand1 + alu_operand2 (即 0 + 28 = 28，十六进制为 1C)。

访存：

dm_wen 被置为高电平数据存储器允许写操作。

dm_addr 被赋值为 alu_result，即计算出的内存地址 28 (十六进制 1C)。

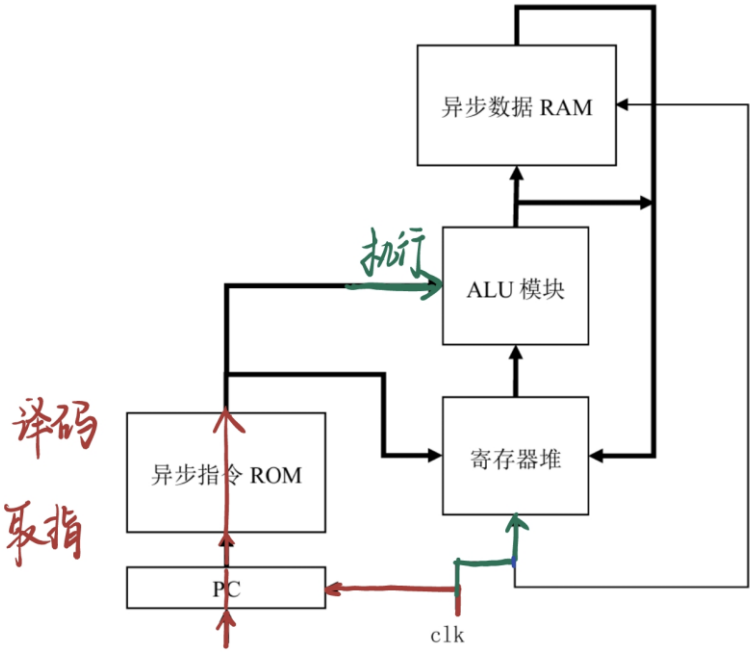
dm_wdata 被赋值为 rt_value，即寄存器 $11 的值。

数据存储器 data_ram_module 在时钟上升沿到来时，将 dm_wdata 的值写入到地址 dm_addr[6:2] (即 28[6:2] = 01110_2 = 14_10) 对应的内存位置。

写回:

rf_wen 为低电平，寄存器堆不会发生写操作

（5）J 指令



取指:

当前程序计数器 pc 的值为 4CH (76)。

inst_addr 被赋值为 pc 的值，即 4CH。

指令存储器 inst_rom_module 根据地址 inst_addr[6:2] (即 76[6:2] = 10011_2 = 19_10) 读取指令。实际寻址 inst_rom 的地址是 76 / 4 = 19。

从 inst_rom 的地址 19 中读取到的指令 32'h08000000 被输出到 inst 信号。

cpu_pc 输出当前的程序计数器值 4CH。

cpu_inst 输出读取到的指令 32'h08000000。

seq_pc 计算下一条顺序执行的指令地址，seq_pc = pc + 4 = 50H。

译码:

指令 inst (32'h08000000) 的各个字段被解析出来:

op = inst[31:26] = 08_16 = 000010_2

target = inst[25:0] = 000000_16 = 0_10

根据 op 的值 (000010_2)，inst_J 信号被置为高电平 (assign inst_J = (op == 6'b000010);)。

无条件跳转信号 j_taken 被置为高电平 (assign j_taken = inst_J;)。

无条件跳转的目标地址 j_target 被计算出来： j_target = {4'b0100, 26'b0, 2'b00} = 32'h40000000。

执行:

jbr_taken 被置为高电平指示需要进行跳转。

next_pc 被赋值为 jbr_target 即 32'h40000000。

访存:

跳转指令不涉及内存访问，因此访存阶段不进行任何操作。

写回:

跳转指令不涉及写回寄存器堆，因此写回阶段不进行任何操作

实验改进:

在单周期 CPU 中添加三个指令,我们选择添加这三条指令:按位同或 nxor；低位加载 hui；算数右移 sra

为此,我们在文件中的修改如下:

（1）在 inst_rom.v 文件中

修改指令存储的总个数为 22

```
wire [31:0] inst_rom[22:0];
```

添加三条指令,将指令存储到 inst_rom 寄存器数组,由于原来的 J 指令会跳转到开头的第 0 条指令,从而实现循环,为了可以顺序执行新增的指令,则将 J 指令的跳转地址给修改为 0x14

```
assign inst_rom[19] = 32'h08000014; // j 0x14
assign inst_rom[20] = 32'hC00E000E; // 30H: hui $14,#14 | $14 =
0000_000EH
assign inst_rom[21] = 32'h00C76831; // 2CH: nxor $13,$6,$7 | $13 =
FFFF_FFEEH
assign inst_rom[22] = 32'h000618C3; // 4CH: sra $3,$6,3 | $3=
FFFFFFFC
```

将新增的三条指令加入其中,以便根据输入的地址 addr 从 inst_rom 指令存储器中选取对应的指令,赋值给 inst 信号

```
always @(*)
    begin
        case (addr)
            5'd0 : inst <= inst_rom[0 ];
            5'd1 : inst <= inst_rom[1 ];
            5'd2 : inst <= inst_rom[2 ];
            5'd3 : inst <= inst_rom[3 ];
            5'd4 : inst <= inst_rom[4 ];
            5'd5 : inst <= inst_rom[5 ];
            5'd6 : inst <= inst_rom[6 ];
            5'd7 : inst <= inst_rom[7 ];
            5'd8 : inst <= inst_rom[8 ];
            5'd9 : inst <= inst_rom[9 ];
```

```
            5'd10: inst <= inst_rom[10];
            5'd11: inst <= inst_rom[11];
            5'd12: inst <= inst_rom[12];
            5'd13: inst <= inst_rom[13];
            5'd14: inst <= inst_rom[14];
            5'd15: inst <= inst_rom[15];
            5'd16: inst <= inst_rom[16];
            5'd17: inst <= inst_rom[17];
            5'd18: inst <= inst_rom[18];
            5'd19: inst <= inst_rom[19];
            5'd20: inst <= inst_rom[20];
            5'd21: inst <= inst_rom[21];
            5'd22: inst <= inst_rom[22];
            default: inst <= 32'd0;
        endcase
    end
```

（2）在 alu.v 文件中

将 alu_control 控制信号改成 13 位

```
input  [13:0] alu_control,  // ALU 控制信号
```

加上下面的代码，实现信号的声明\控制信号的赋值\结果信号的声明\结果信号的赋值

```
wire alu_nxor; //按位同或
wire alu_hui; //低位加载
assign alu_nxor =alu_control[12];
assign alu_hui  =alu_control[13];
wire [31:0]hui_result;
wire [31:0]nxor_result;
assign hui_result = {16'd0, alu_src2[15:0]}; // 立即数装载结果为立即数移位至低半字节
assign nxor_result= ~xor_result; // 同或结果为异或结果按位取反
```

将结果信号加入其中,以便根据不同的控制信号选择不同的运算结果,赋值给 alu_result 信号

```
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 :
                        32'd0;
```

（3）single_cycle.cpu.v 文件

加上下面的代码实现信号声明与三类的指令识别

```
wire inst_NXOR, inst_HUI,inst_SRA;
assign inst_NXOR = (op == 6'b110001); // 逻辑同或运算
assign inst_HUI = (op == 6'b110000); // 立即数装载低半字字
assign inst_SRA = op_zero & (rs==5'd0) & (funct == 6'b000011); //
算数右移
```

加上下面的代码实现信号的声明与赋值

```
wire inst_nxor, inst_hui;
assign inst_hui = inst_HUI; // 立即数装载低位
assign inst_nxor = inst_NXOR;// 逻辑同或
assign inst_sra = inst_SRA;//算术右移
```

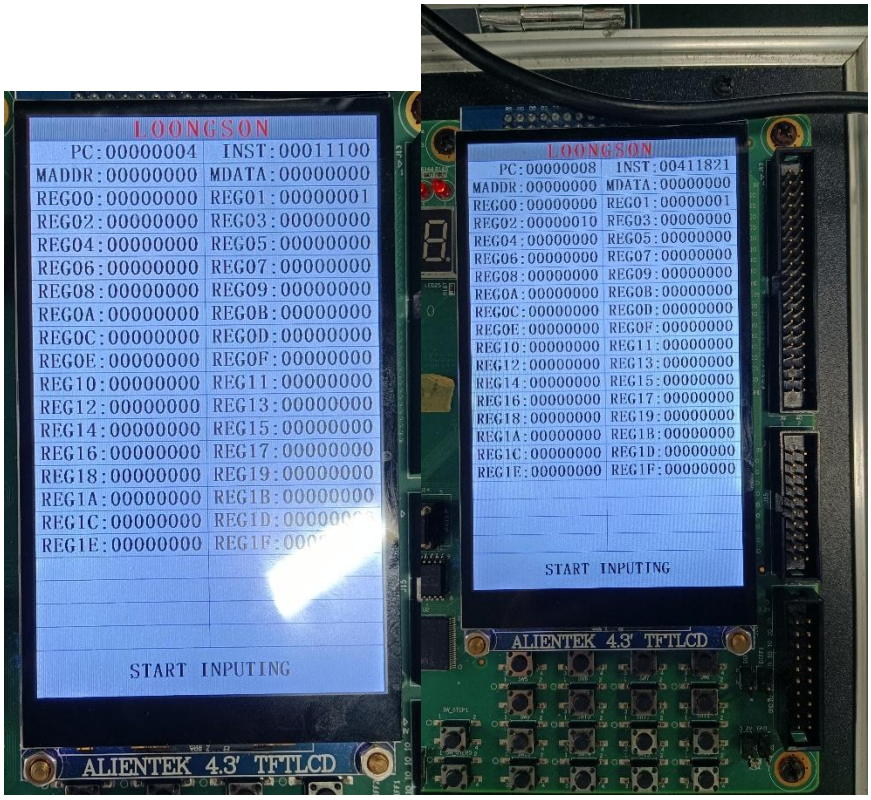
对立即数拓展指令进行修改

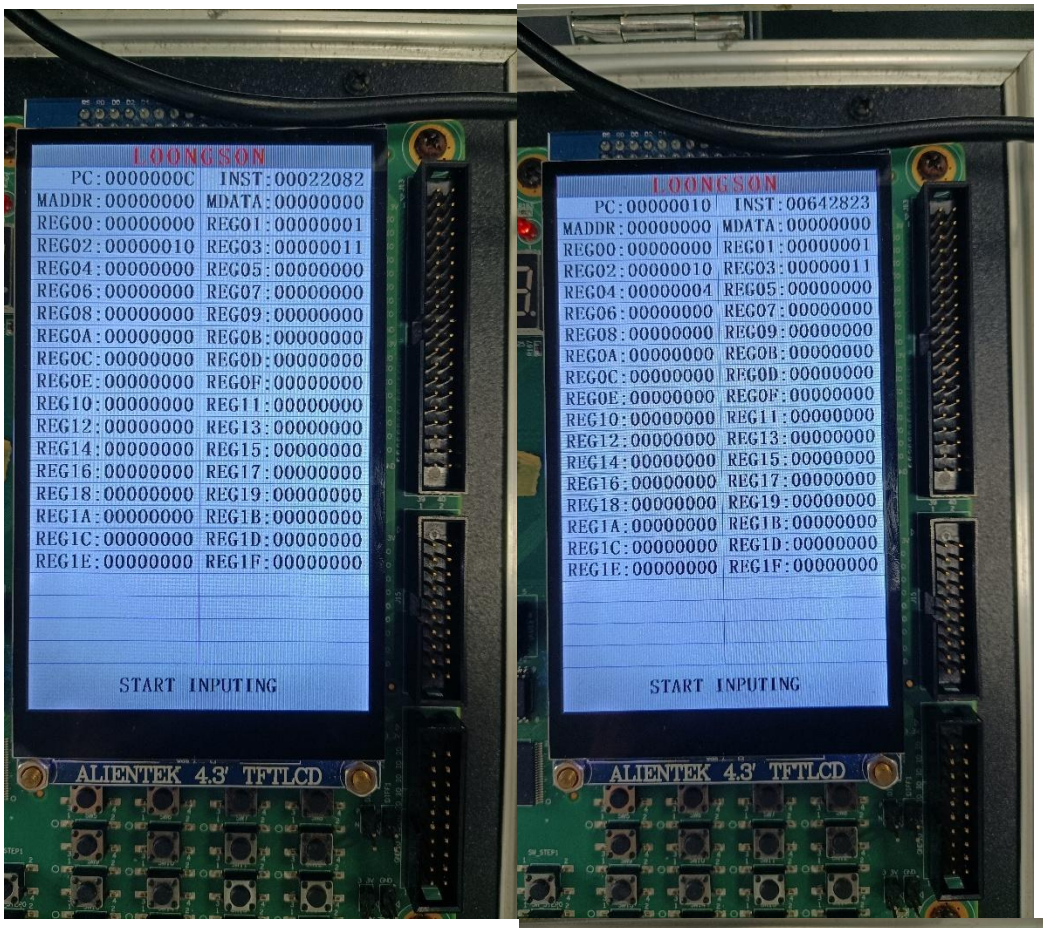
```
wire [31:0] sext_imm;
wire   inst_shf_sa;    //使用 sa 域作为偏移量的指令
wire   inst_imm_sign;  //对立即数作符号扩展的指令
assign sext_imm      = {{16{imm[15]}}, imm};// 立即数符号扩展
assign inst_shf_sa = inst_SLL | inst_SRL | inst_SRA;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW |
inst_HUI;
```

实验结果分析

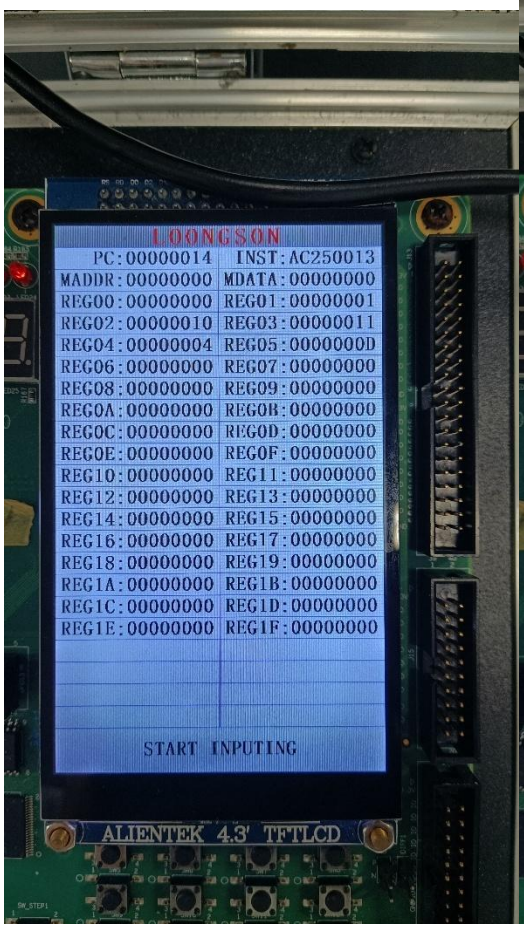
如下图，分别为源代码中 20 条指令所对应的 PC 值和指令地址值，以及执行过程中寄存器的值，其中第 13 条指令不执行

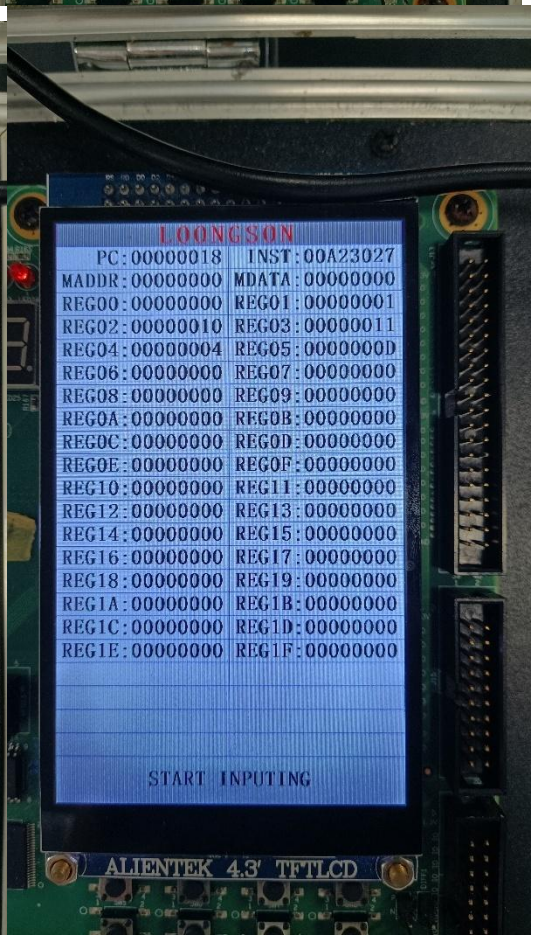
```
assign inst_rom[ 0] = 32'h24010001; // 00H: a
assign inst_rom[ 1] = 32'h00011100; // 04H: s
assign inst_rom[ 2] = 32'h00411821; // 08H: a
assign inst_rom[ 3] = 32'h00022082; // 0CH: s
assign inst_rom[ 4] = 32'h00642823; // 10H: s
assign inst_rom[ 5] = 32'hAC250013; // 14H:
assign inst_rom[ 6] = 32'h00A23027; // 18H:
assign inst_rom[ 7] = 32'h00C33825; // 1CH:
assign inst_rom[ 8] = 32'h00E64026; // 20H:
assign inst_rom[ 9] = 32'hAC08001C; // 24H:
assign inst_rom[10] = 32'h00C7482A; // 28H:
assign inst_rom[11] = 32'h11210002; // 2CH:
assign inst_rom[12] = 32'h24010004; // 30H:
assign inst_rom[13] = 32'h8C2A0013; // 34H:
assign inst_rom[14] = 32'h15450003; // 38H:
assign inst_rom[15] = 32'h00415824; // 3CH:
assign inst_rom[16] = 32'hAC0B001C; // 40H:
assign inst_rom[17] = 32'hAC040010; // 44H:
assign inst_rom[18] = 32'h3C0C000C; // 48H:
assign inst_rom[19] = 32'h08000000; // 4CH:
```



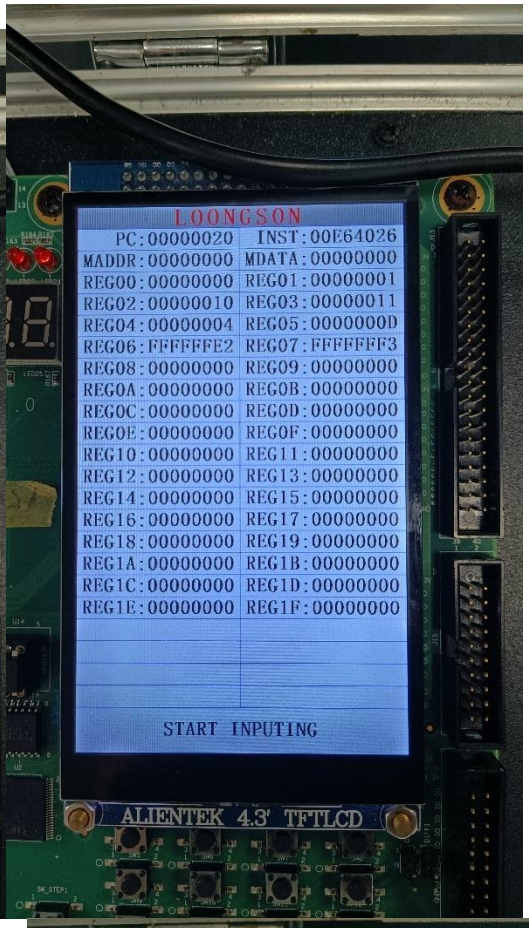

LOONGSON
PC:0000000C INST:00022082
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000000 REG05:00000000
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

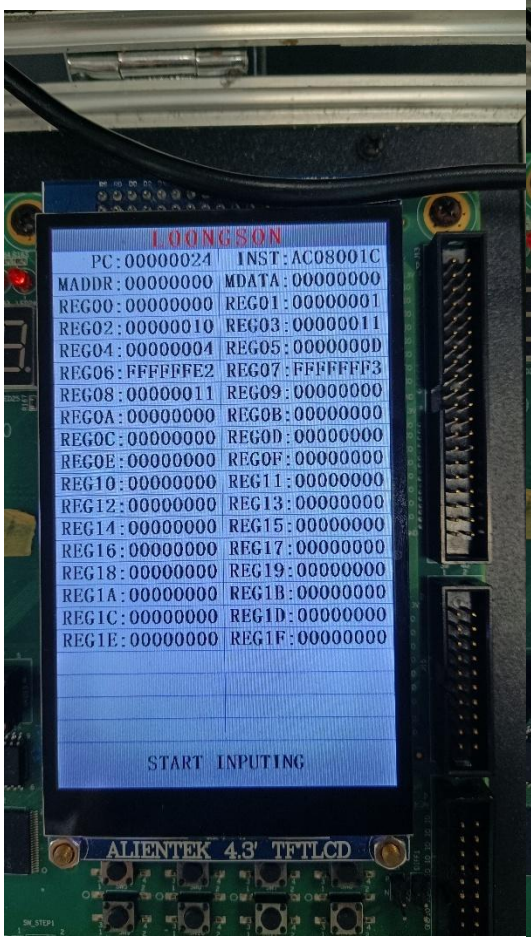
LOONGSON
PC:00000010 INST:00642823
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:00000000
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

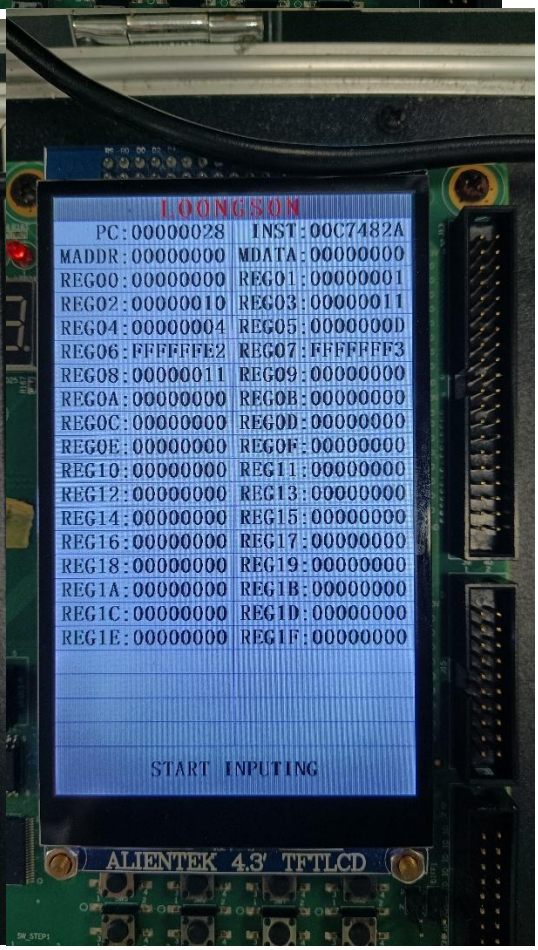

LOONGSON
PC:00000014 INST:AC250013
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

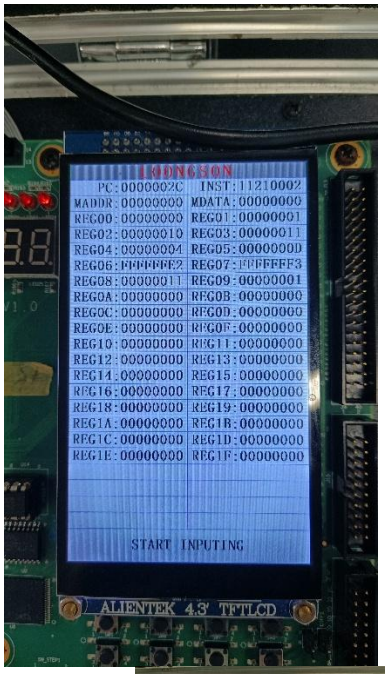

LOONGSON
PC:00000018 INST:00A23027
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

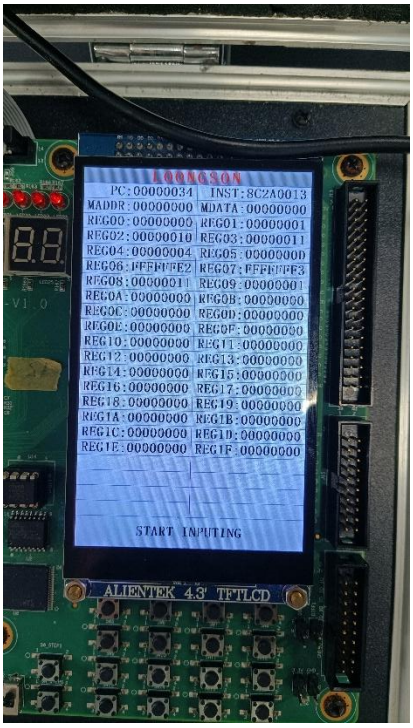

LOONGSON
PC:0000001C INST:00C33825
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

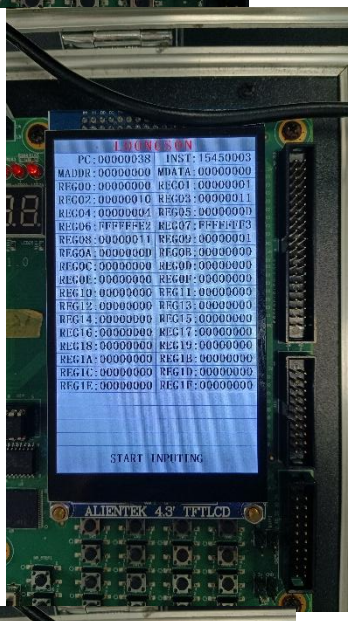

LOONGSON
PC:00000020 INST:00E64026
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000024 INST:AC08001C
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000028 INST:00C7482A
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

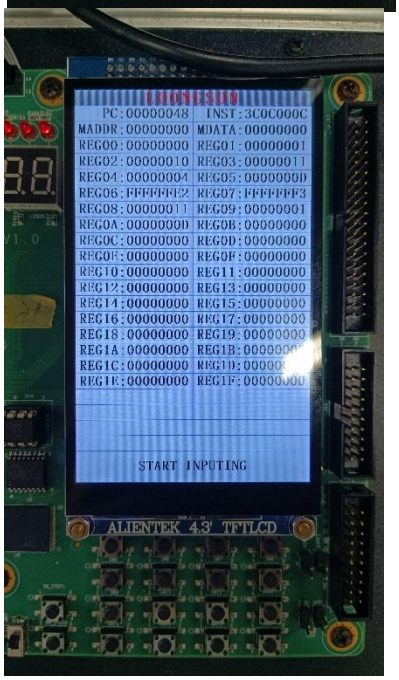

LOONGSON
PC:0000002C INST:11210002
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

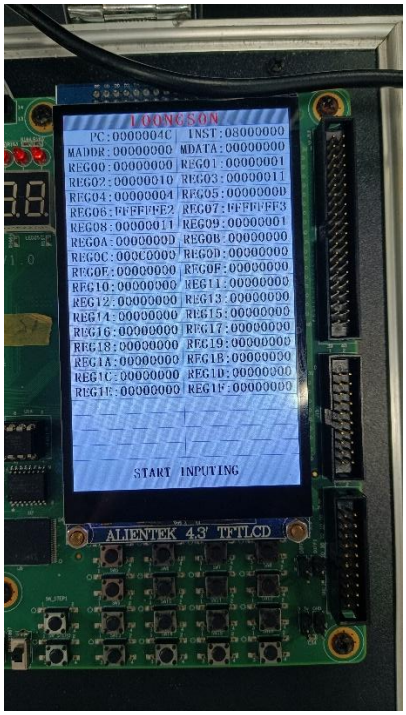

LOONGSON
PC:00000034 INST:8C2A0013
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000038 INST:15450003
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:0000003C INST:00415824
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000040 INST:AC0B001C
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFF2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000048 INST:3C0C000C
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFF2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD



下面三幅图为新增的三条指令验证结果

```
assign inst_rom[20] = 32'hC00E000E; // 30H: hui $14,#14 | $14 =
0000_000EH
assign inst_rom[21] = 32'h00C76831; // 2CH: nxor $13,$6,$7 | $13 =
FFFF_FFEEH
assign inst_rom[22] = 32'h000618C3; // 4CH: sra $3,$6,3 | $3=
FFFFFFFC
```


LOONGSON
PC:00000044 INST:AC040010
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:0000004C INST:08000014
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:000C0000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000050 INST:C00E000E
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:000C0000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

总结感想

1.通过本次单周期 CPU 的设计与实现实验，我对计算机体系结构有了更加深刻和直观的理解。单周期 CPU 是一种结构较为简单的处理器设计方式，每条指令在一个时钟周期内完成所有操作，虽然在实际应用中不常见，但它非常适合作为学习计算机硬件工作原理的入门。

在实验过程中，我主要完成了以下几个方面的内容：

熟悉了指令集体系结构（ISA），理解了指令的编码方式以及各类指令（如 R 型、I 型、J 型）的处理流程。

搭建了数据通路，包括指令存储器、寄存器堆、ALU、数据存储器等关键模块。

理解了控制信号的生成逻辑，包括主控制器和 ALU 控制器的设计。

进行了仿真验证，确保各指令正确执行，调试过程中也加深了对数据流和控制流的理解。

在搭建数据通路时，我深刻体会到各个模块之间的协调配合至关重要。例如，寄存器堆的读写、ALU 操作数的选择、以及如何根据指令类型设置跳转或分支，都需要准确无误地连接和控制。此外，控制单元的设计对于整个 CPU 的正确运行起着决定性作用，它需要根据不同的指令类型生成正确的控制信号。

2.防抖设置

在单周期 CPU 上，用实验箱（开发板）验证时，按一下单步执行按钮，却执行了多条指令， 这是非常典型的机械按键抖动问题：按一下实际上震动了好几次，导致 CPU 以为按了好几下，就连续执行了多条指令。

修改 single_cycle_cpu.v 文件

```
`timescale 1ns / 1ps
//*************************************************************************
*********
//    > 文件名: single_cycle_cpu_display.v
//    > 描述   : 单周期 CPU 显示模块，调用 FPGA 板上的 IO 接口和触摸屏
//    > 作者   : LOONGSON + 修改（防抖版）
```

```
//   > 日期  : 2025-04-29
//*****************************************************************
*********
module single_cycle_cpu_display(
    //时钟与复位信号
    input clk,
    input resetn,    //后缀"n"代表低电平有效

    //脉冲开关，用于产生脉冲 clk，实现单步执行
    input btn_clk,

    //触摸屏相关接口，不需要更改
    output lcd_rst,
    output lcd_cs,
    output lcd_rs,
    output lcd_wr,
    output lcd_rd,
    inout[15:0] lcd_data_io,
    output lcd_bl_ctr,
    inout ct_int,
    inout ct_sda,
    output ct_scl,
    output ct_rstn
    );

//-----{时钟和复位信号}begin
//不需要更改，用于单步调试
    wire cpu_clk;    //单周期 CPU 里使用脉冲开关作为时钟，以实现单步执
行

    // 加入防抖处理
    reg [19:0] btn_cnt;
    reg btn_clean;  // 防抖后的按钮信号

    always @(posedge clk or negedge resetn) begin
        if (!resetn) begin
            btn_cnt <= 20'd0;
            btn_clean <= 1'b1;
        end
        else begin
            if (btn_clk == 1'b0) begin  // 按下（低电平有效）
                if (btn_cnt < 20'd999_999) begin // 20ms 防抖
                    btn_cnt <= btn_cnt + 1'b1;
```

```
                end
                else begin
                    btn_clean <= 1'b0;
                end
            end
            else begin
                btn_cnt <= 20'd0;
                btn_clean <= 1'b1;
            end
        end
    end

    reg btn_clk_r1;
    reg btn_clk_r2;

    always @(posedge clk) begin
        if (!resetn) begin
            btn_clk_r1 <= 1'b0;
            btn_clk_r2 <= 1'b0;
        end
        else begin
            btn_clk_r1 <= ~btn_clean;  // 用防抖后的信号
            btn_clk_r2 <= btn_clk_r1;
        end
    end

    wire clk_en;
    assign clk_en = !resetn || (!btn_clk_r1 && btn_clk_r2);
    BUFGCE cpu_clk_cg(.I(clk),.CE(clk_en),.O(cpu_clk));
//-----{时钟和复位信号}end

//-----{调用单周期 CPU 模块}begin
    //用于在 FPGA 板上显示结果
    wire [31:0] cpu_pc;    //CPU 的 PC
    wire [31:0] cpu_inst;  //该 PC 取出的指令
    wire [ 4:0] rf_addr;   //扫描寄存器堆的地址
    wire [31:0] rf_data;   //寄存器堆从调试端口读出的数据
    reg  [31:0] mem_addr;  //要观察的内存地址
    wire [31:0] mem_data;  //内存地址对应的数据

    single_cycle_cpu cpu(
        .clk     (cpu_clk   ),
        .resetn  (resetn    ),
```

```
        .rf_addr (rf_addr   ),
        .mem_addr(mem_addr  ),
        .rf_data (rf_data   ),
        .mem_data(mem_data  ),
        .cpu_pc  (cpu_pc    ),
        .cpu_inst(cpu_inst  )
    );
//-----{调用单周期 CPU 模块}end

//---------------------{调用触摸屏模块}begin--------------------//
//-----{实例化触摸屏}begin
//此小节不需要更改
    reg         display_valid;
    reg  [39:0] display_name;
    reg  [31:0] display_value;
    wire [5 :0] display_number;
    wire        input_valid;
    wire [31:0] input_value;

    lcd_module lcd_module(
        .clk            (clk           ),   //10Mhz
        .resetn         (resetn        ),

        //调用触摸屏的接口
        .display_valid  (display_valid ),
        .display_name   (display_name  ),
        .display_value  (display_value ),
        .display_number (display_number),
        .input_valid    (input_valid   ),
        .input_value    (input_value   ),

        //lcd 触摸屏相关接口，不需要更改
        .lcd_rst        (lcd_rst       ),
        .lcd_cs         (lcd_cs        ),
        .lcd_rs         (lcd_rs        ),
        .lcd_wr         (lcd_wr        ),
        .lcd_rd         (lcd_rd        ),
        .lcd_data_io    (lcd_data_io   ),
        .lcd_bl_ctr     (lcd_bl_ctr    ),
        .ct_int         (ct_int        ),
        .ct_sda         (ct_sda        ),
        .ct_scl         (ct_scl        ),
        .ct_rstn        (ct_rstn       )
    );
```

```
//-----{实例化触摸屏}end

//-----{从触摸屏获取输入}begin
    always @(posedge clk) begin
        if (!resetn) begin
            mem_addr <= 32'd0;
        end
        else if (input_valid) begin
            mem_addr <= input_value;
        end
    end

    assign rf_addr = display_number-6'd5;
//-----{从触摸屏获取输入}end

//-----{输出到触摸屏显示}begin
    always @(posedge clk) begin
        if (display_number >6'd4 && display_number <6'd37 ) begin
//块号 5~36 显示 32 个通用寄存器的值
            display_valid <= 1'b1;
            display_name[39:16] <= "REG";
            display_name[15: 8] <= {4'b0011,3'b000,rf_addr[4]};
            display_name[7 : 0] <= {4'b0011,rf_addr[3:0]};
            display_value       <= rf_data;
        end
        else begin
            case(display_number)
                6'd1 : begin
                    display_valid <= 1'b1;
                    display_name  <= "   PC";
                    display_value <= cpu_pc;
                end
                6'd2 : begin
                    display_valid <= 1'b1;
                    display_name  <= " INST";
                    display_value <= cpu_inst;
                end
                6'd3 : begin
                    display_valid <= 1'b1;
                    display_name  <= "MADDR";
                    display_value <= mem_addr;
                end
                6'd4 : begin
                    display_valid <= 1'b1;
```

```
                    display_name  <= "MDATA";
                    display_value <= mem_data;
                end
                default : begin
                    display_valid <= 1'b0;
                end
            endcase
        end
    end
//-----{输出到触摸屏显示}end
//----------------------{调用触摸屏模块}end--------------------//
endmodule
```