# 组成原理课程第五次实验报告

# 实验名称：ROM 存储器和单周期 CPU 实验

学号：2310422　姓名：谢小珂　　班次：　1078

## 一、实验目的

1. 了解只读存储器 ROM 的原理。
2. 理解 ROM 读取数据的过程。
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 ROM 和异步 ROM 的区别。
5. 掌握调用 xilinx 库 IP 实例化 ROM 的设计方法。
6. 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。
7. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
8. 熟悉并掌握单周期 CPU 的原理和设计。
9. 进一步加强运用 verilog 语言进行电路设计的能力。
10. 为后续设计多周期 cpu 的实验打下基础。

## 二、实验内容说明

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。

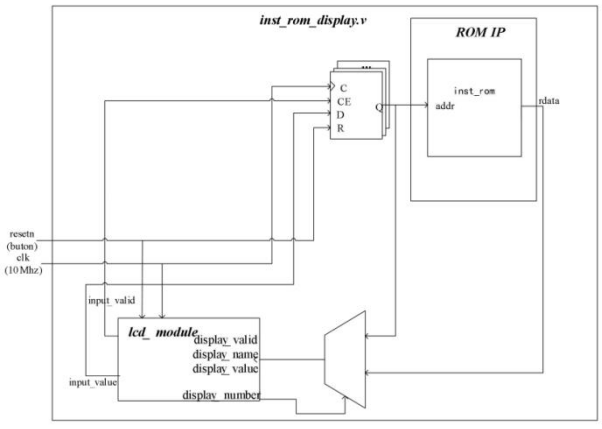
2、单周期 CPU 实验，原理图应基于实验指导手册中的图 7.1，在分析表 7.4 中指令执行过程时，可以在图 7.1 基础上辅助画线表示执行过程。

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。

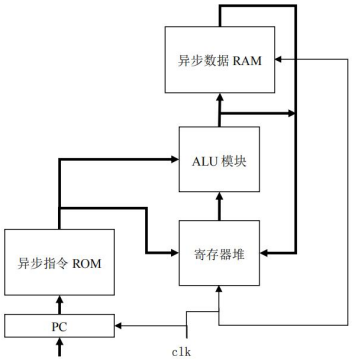
4、（提高要求，不强求做出来）把 ALU 实验中添加的三个指令，自行添加到这个单周期 CPU 中，注意指令码只要跟现有的不冲突就行，不必限制在标准的 MIPS 指令格式。

## 三、实验原理图

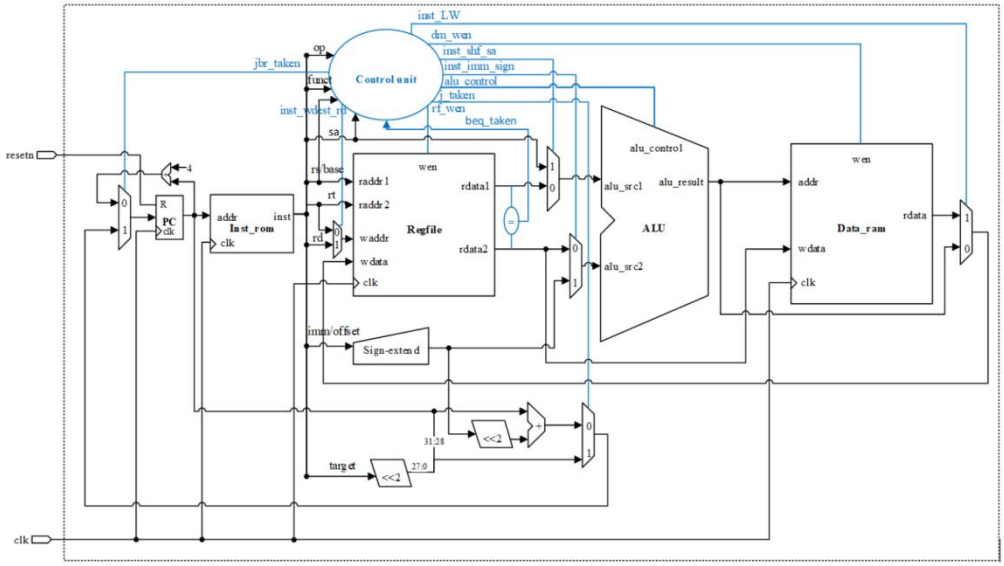
（一）同步 ROM 的顶层模块如下图所示：



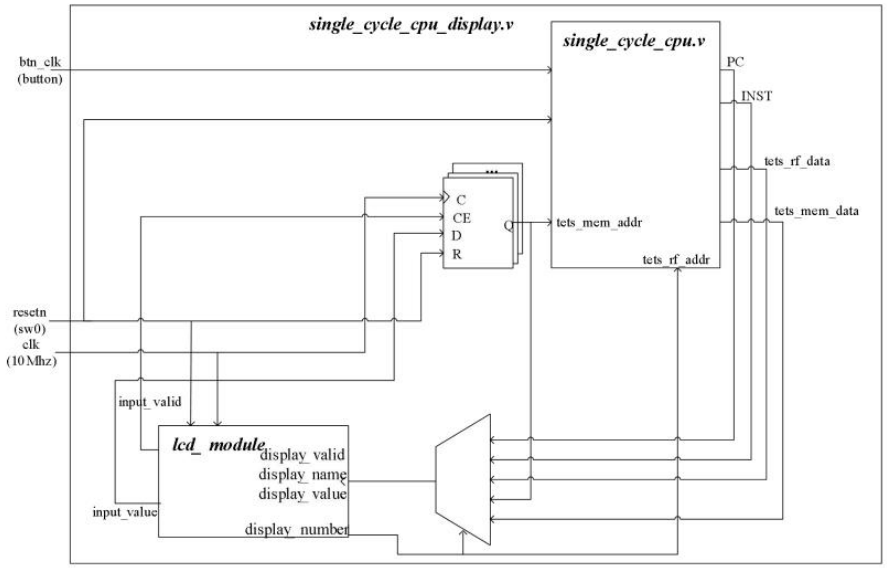
（二）单周期 CPU 实验的原理图如下图所示：




单周期 cpu 的大致框图




单周期 cpu 的实现框图




单周期 cpu 的顶层模块框图

# 四、实验步骤

## （一）ROM 存储器实验

1. 同步 ROM

・生成 IP 核

具体步骤与实验指导手册大致相同，故不再赘述。

2. 异步 ROM

具体步骤与实验指导手册大致相同，故不再赘述。

## （二）单周期 CPU 实验

1. 分析 MIPS 架构指令在指令存储器中的编码和执行过程，理解每条指令的功能及其对寄存器和内存的影响，代码内容如下：

```
//------------- 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果 -----//
   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
```

指令 1：addiu $1, $0, #1

・**编码：**32'h24010001

・**功能：**将立即数 1 无符号加到$0（恒为 0），结果存入$1。

・**执行过程：**

**取指：**PC=00H，读取 24010001。

**解码：**识别为 I 型指令（addiu），立即数=1。

**执行：**$1 = 0 + 1 = 1。

写回：$1 = 0000_0001H。

指令 2：sll $2, $1, #4
- 编码：32'h00011100
- 功能：将$1 逻辑左移 4 位，结果存入$2。
- 执行过程：
  取指：PC=04H，读取 00011100。
  解码：识别为 R 型指令（sll），移位位数=4。
  执行：$2 = 1 << 4 = 0000_0010H（16 进制显示为 0x10，注释有误）。
  写回：$2 = 0000_0010H。

指令 3：addu $3, $2, $1
- 编码：32'h00411821
- 功能：无符号加法，$3 = $2 + $1。
- 执行过程：
  取指：PC=08H，读取 00411821。
  解码：识别为 R 型指令（addu）。
  执行：$3 = 0x10 + 0x1 = 0000_0011H（应为 0x11，注释正确）。
  写回：$3 = 0000_0011H。

指令 4：srl $4, $2, #2
- 编码：32'h00022082
- 功能：将$2 逻辑右移 2 位，结果存入$4。
- 执行过程：
  执行：$4 = 0x10 >> 2 = 0000_0004H。
  写回：$4 = 0000_0004H。

指令 5：subu $5, $3, $4
- 编码：32'h00642823
- 功能：无符号减法，$5 = $3 - $4。
- 执行过程：
  执行：$5 = 0x11 - 0x4 = 0000_000DH。
  写回：$5 = 0000_000DH。

指令 6：sw $5, #19($1)
- 编码：32'hAC250013
- 功能：将$5 的值存储到内存地址$1 + 19 处。
- 执行过程：
  计算地址：$1 + 19 = 1 + 19 = 20（0x14H）。
  内存写入：Mem[0x14] = 0000_000DH。

指令 7：nor $6, $5, $2
- 编码：32'h00A23027
- 功能：按位或非，$6 = ~($5 | $2)。

・执行过程：
    执行：$6 = ~(0xD | 0x10) = ~0x1D = FFFF_FFE2H。
    写回：$6 = FFFF_FFE2H。

指令 8：or $7, $6, $3
・编码：32'h00C33825
・功能：按位或，$7 = $6 | $3。
・执行过程：
    执行：$7 = 0xFFFF_FFE2 | 0x11 = FFFF_FFF3H。
    写回：$7 = FFFF_FFF3H。

指令 9：xor $8, $7, $6
・编码：32'h00E64026
・功能：按位异或，$8 = $7 ^ $6。
・执行过程：
    执行：$8 = 0xFFFF_FFF3 ^ 0xFFFF_FFE2 = 0000_0011H。
    写回：$8 = 0000_0011H。

指令 10：sw $8, #28($0)
・编码：32'hAC08001C
・功能：将$8 存储到内存地址 28（0x1CH）。
・执行过程：
    内存写入：Mem[0x1C] = 0000_0011H。

指令 11：slt $9, $6, $7
・编码：32'h00C7482A
・功能：若$6 < $7，则$9 = 1，否则$9 = 0。
・执行过程：
    执行：0xFFFF_FFE2 < 0xFFFF_FFF3 为真，$9 = 1。
    写回：$9 = 0000_0001H。

指令 12：beq $9, $1, #2
・编码：32'h11210002
・功能：若$9 == $1，则跳转到 PC + 4 + 2*4 = 34H。
・执行过程：
    比较：$9=1，$1=1，条件成立。
    跳转：PC 更新为 34H（跳过下一条指令）。

指令 13：lw $10, #19($1)
・编码：32'h8C2A0013
・功能：从内存地址$1 + 19 = 20（0x14H）加载数据到$10。
・执行过程：
    内存读取：$10 = Mem[0x14] = 0000_000DH。
    写回：$10 = 0000_000DH。

**指令 14**：bne $10, $5, #3
- **编码**：32'h15450003
- **功能**：若$10 != $5，则跳转。此处$10 = $5，不跳转。
- **执行过程**：
  **比较**：$10=0xD，$5=0xD，条件不成立。
  PC 顺序递增。

**指令 15**：and $11, $2, $1
- **编码**：32'h00415824
- **功能**：按位与，$11 = $2 & $1。
- **执行过程**：
  **执行**：$11 = 0x10 & 0x1 = 0000_0000H。
  **写回**：$11 = 0000_0000H。

**指令 16**：sw $11, #28($0)
- **编码**：32'hAC0B001C
- **功能**：将$11 存储到内存地址 28（0x1CH）。
- **执行过程**：
  **内存写入**：Mem[0x1C] = 0000_0000H（覆盖之前的值）。

**指令 17**：sw $4, #16($0)
- **编码**：32'hAC040010
- **功能**：将$4 存储到内存地址 16（0x10H）。
- **执行过程**：
  **内存写入**：Mem[0x10] = 0000_0004H。

**指令 18**：lui $12, #12
- **编码**：32'h3C0C000C
- **功能**：将立即数 12 加载到$12 的高 16 位。
- **执行过程**：
  **执行**：$12 = 0x000C << 16 = 000C_0000H。
  **写回**：$12 = 000C_0000H。

**指令 19**：j 00H
- **编码**：32'h08000000
- **功能**：无条件跳转到地址 00H。
- **执行过程**：
  PC 更新为 00H，程序循环执行。

**2.针对 R 型、J 型、I 型指令进行具体分析，如下：**

**<1>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，决定移位指令的移位位数；
func（6b）：功能位，决定 R 型指令的具体功能。

**例子：**

**（1）addu $3, $2, $1**

- 指令编码：32'h00411821
- 字段分解：

op=000000（R 型） rs=00001（$1） rt=00010（$2）
rd=00011（$3） shamt=00000（无移位） funct=100001（addu 操作码）

- 执行过程：

取指（IF）：PC=08H，从 inst_rom 读取 00411821。
解码（ID）：识别 op=000000 为 R 型指令，funct=100001 为无符号加法。
从寄存器堆读取$1=1、$2=0x10。
执行（EX）：ALU 计算$3 = $2 + $1 = 0x10 + 0x1 = 0x11。
写回（WB）：将结果 0x11 写入寄存器$3。

**（2）and $11, $2, $1**

- 指令编码：32'h00415824
- 字段分解：

op=000000（R 型） rs=00010 （$2） rt=00001（$1）
rd=01011（$11） shamt=00000（无移位操作） funct=100100（and 操作码）

- 执行过程

取指（IF）：PC=3CH，从 inst_rom 读取 32'h00415824。
解码（ID）：识别 op=000000 为 R 型指令，解析 funct=100100 为 and 操作，从寄存器堆读取：$2 = 0000_0010H（0x10）、$1 = 0000_0001H（0x1）
执行（EX）：ALU 执行按位与运算 11=11=2 & $1 = 0x10 & 0x1 = 0000_0000H
写回（WB）：将运算结果 0000_0000H 写入寄存器$11

**<2>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

**例子：**

**（1）sw $5, #19($1)**

- 指令编码：32'hAC250013
- 字段分解：

op=101011（sw 操作码） rs=00001（$1） rt=00101（$5）
imm=0000000000010011（19 的二进制）

·执行过程：

取指（IF）：PC=14H，读取 AC250013。

解码（ID）：识别为存储指令，符号扩展立即数 imm=19，读取$1=1、$5=0xD。

执行（EX）：计算地址$1 + imm = 1 + 19 = 20（0x14H）。

访存（MEM）：将$5=0xD 写入内存地址 0x14。

（2）addiu $1, $0, #1

·指令编码：32'h24010001

·字段分解：

op=001001（addiu 操作码） rs=00000（$0） rt=00001（$1）

imm=0000000000000001（1 的二进制）

·执行过程：

取指（IF）：PC=00H，读取 24010001。

解码（ID）：识别为立即数加法，符号扩展 imm=1。

执行（EX）：ALU 计算$1 = $0 + 1 = 1。

写回（WB）：将结果 1 写入$1

<3>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

例子：

（1）j 00H

·指令编码：32'h08000000

·字段分解：

op=000010（j 操作码） address=00000000000000000000000000（目标地址 00H）

·执行过程：

取指（IF）：PC=4CH，读取 08000000。

解码（ID）：识别为无条件跳转指令。

计算目标地址：拼接高位 PC 和 address：(PC+4)[31:28] | (address<<2) = 0x00000000。

更新 PC：PC 直接跳转到 00H，程序循环执行。

## （三）自行添加三个指令到周期 CPU

·添加的三个指令（无符号大于置位、有符号大于置位、按位取反）

（1）在 alu.v 中修改代码：

```
input [14:0] alu_control, // ALU 控制信号扩展 3 位
//添加控制信号
wire alu_sgt;//有符号大于置位
wire alu_sgtu;//无符号大于置位
wire alu_not;//按位取反
assign alu_sgtu = alu_control[2];
assign alu_sgt = alu_control[1];
```

```
assign alu_not = alu_control[0];
wire [31:0] not_result;
assign not_result = ~alu_src1; // 按位取反
wire [31:0] sgtu_result;
assign sgtu_result = {31'd0, adder_cout & (|adder_result)}; //无符号大于置位
wire [31:0] sgt_result; // 有符号大于置位
assign sgt_result = {31'd0, ~slt_result[0] & (|adder_result)};
// 修改 ALU 结果选择逻辑
assign alu_result = alu_sgt ? sgt_result :
alu_sgtu ? sgtu_result :
alu_not ? not_result : ……
32'd0;
```

（2）在 single_cycle_cpu.v 中修改代码：

```
wire inst_SGTU, inst_SGT , inst_NOT; // 添加指令列表
// 自定义指令编码
assign inst_SGT=op_zero & sa_zero & (funct == 6'b101011); // 有符号大于置位
assign inst_NOT=op_zero & sa_zero & (funct == 6'b101101); // 按位取反
assign inst_SGTU=op_zero & sa_zero & (funct == 6'b101100); // 无符号大于置位
// 修改 ALU 控制信号生成
assign inst_sgt = inst_SGT; // 有符号大于置位
assign inst_not = inst_NOT; // 按位取反
assign inst_sgtu = inst_SGTU; // 无符号大于置位
assign alu_control = { …… ,
inst_sgtu,
inst_sgt,
inst_not};
// 修改写回目标寄存器选择
assign inst_wdest_rd = …… | inst_SGTU | inst_SGT | inst_NOT;
```

（3）在 inst_rom.v 中修改代码：

```
wire [31:0] inst_rom[22:0]; // 添加指令存储器大小
// 添加指令
assign inst_rom[19] = 32'h0065682B; // SGT $13, $3, $5 有符号大于置位
assign inst_rom[20] = 32'h0065702C; // SGTU $14, $3, $5 无符号大于置位
assign inst_rom[21] = 32'h00A0782D; // NOT $15, $5 按位取反
// 添加取指令
case (addr) ……
5'd20: inst <= inst_rom[20];
5'd21: inst <= inst_rom[21];
5'd22: inst <= inst_rom[22];
```

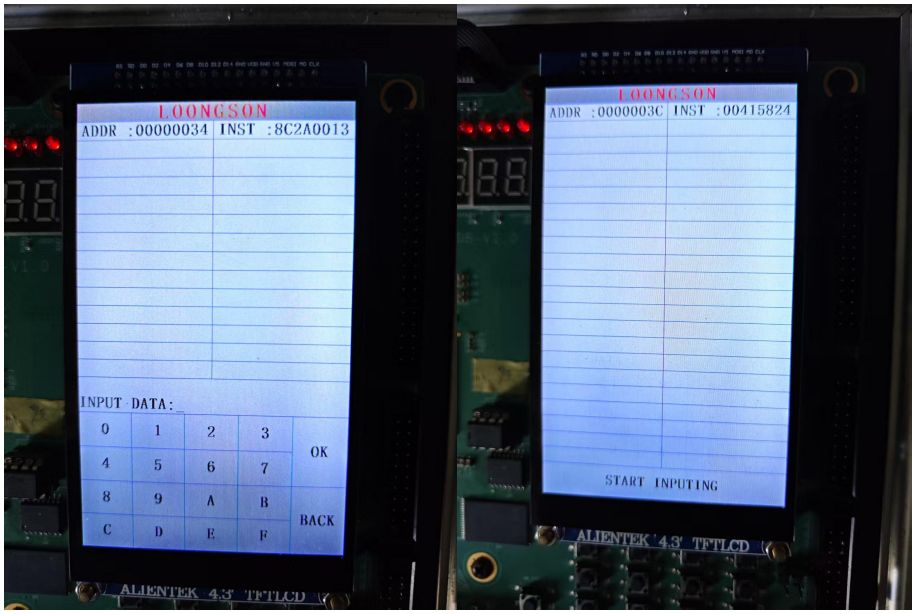
```
default: inst <= 32'd0;
endcase
```

# 五、实验结果分析

## （一）ROM 存储器实验

### （1）同步 ROM

• 实验箱验证




**左图分析（地址：0x34，指令：8C2A0013）**

• 指令类型：I 型指令（存储访问类）

• 预期指令：lw $10, 19($1)（加载字指令）

• 实际读取值：32'h8C2A0013

• 二进制分解：

100011 00001 01010 0000000000010011

op=100011（lw 操作码） ✔

rs=00001（基址寄存器$1） ✔

rt=01010（目标寄存器$10） ✔

offset=0000000000010011（19 的符号扩展） ✔

• 地址计算：$1 + 19 = 1 + 19 = 20（0x14）

•内存取值：确认 Mem[0x14]存储了有效数据（根据前序指令 sw $5,19($1)，为 0x0000000D）。

结论：ROM 在地址 0x34 正确存储了 lw $10,19($1)指令。

**右图分析（地址：0x3C，指令：00415824）**

• 指令类型：R 型指令

• 预期指令：and $11, $2, $1（按位与）

• 实际读取值：32'h00415824

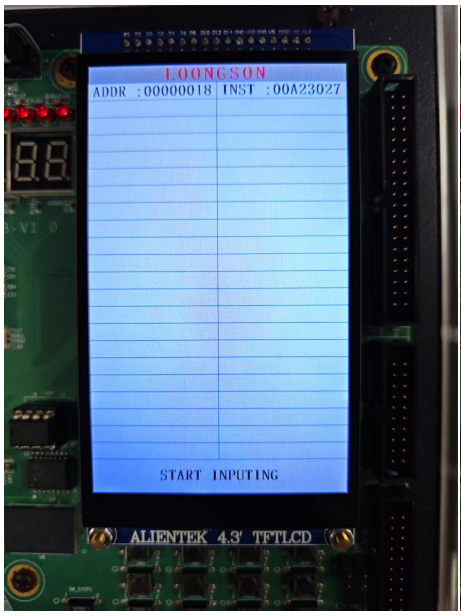
• 二进制分解：

000000 00010 00001 01011 00000 100100

op=000000（R 型） ✔

funct=100100（and 操作码） ✔
rs=00010（2），rt=00001（2），rt=00001（1），rd=01011（$11） ✔
•寄存器输入：$2 = 0x00000010（来自 sll $2,$1,4） $1 = 0x00000001（来自 addiu $1,$0,1）
• ALU 运算：$11 = $2 & $1 = 0x10 & 0x1 = 0x0
结论：ROM 在地址 0x3C 正确存储了 and $11,$2,$1 指令。
**经验证，结果均正确**

**（1）异步 ROM**
**• 实验箱验证**




**左图分析（地址：0x18，指令：00A23027）**
• 预期指令：nor $6, $5, $2（R 型指令）
• 实际读取值：32'h00A23027
• 二进制匹配：
000000 00101 00010 00110 00000 100111
op=000000（R 型） ✔
funct=100111（nor） ✔
rs=5,rt=5,rt=2, rd=$6 ✔
结论：异步 ROM 在地址 0x18 正确存储了指令 nor $6, $5, $2。

**右图分析（地址：0x0C，指令：00022082）**
• 预期指令：srl $4, $2, 2（R 型指令）
• 实际读取值：32'h00022082
• 二进制匹配：
000000 00000 00010 00100 00010 000010
op=000000（R 型） ✔
funct=000010（srl） ✔
rt=2,rd=2,rd=4, shamt=2 ✔
结论：异步 ROM 在地址 0x0C 正确存储了指令 srl $4, $2, 2。
**经验证，结果均正确**

**ROM 与 RAM 的特性对比分析：**

1.ROM（只读存储器）特性详解

- 基本特性：

  数据在制造或烧录时预先写入，工作状态下仅支持读取操作，无法进行动态修改。

  主要用于存储固定程序代码或常量数据。

- 同步 ROM 实现：

  通常通过 FPGA 的 IP 核生成。

  读取操作需要 2 个时钟周期的延迟： 第一个周期用于地址译码、第二个周期完成数据输出。

  典型应用场景：需要与系统时钟同步的处理器指令存储

- 异步 ROM 实现：

  地址信号变化后立即输出对应数据

  典型特征：零延迟响应（如 MIPS 处理器连续指令流场景）

  优势：适用于需要实时响应的控制场合

2.RAM（随机存取存储器）特性详解

- 基本特性：

  支持随时读写操作、具有数据易失性（断电后存储内容丢失）

  主要用途：存储程序运行时的临时变量、堆栈数据等

- 同步 RAM 实现：

  写入操作在时钟上升沿生效、读取操作需要 1 个时钟周期的延迟

  典型特征：读写操作与系统时钟严格同步

- 异步 RAM 实现：

  写入操作完成后数据立即可读、不需要等待时钟信号

  典型应用：需要快速响应的外设接口

3.ROM 与 RAM 的差异对比

- ROM 专注于静态数据的可靠存储和快速读取，优化重点在于地址译码效率和存储密度
  RAM 侧重动态数据管理，优化重点在于读写冲突解决和时序控制
- 同步存储器（ROM/RAM）：

  适合高性能计算场景（如 CPU 缓存）

  优势：时序可控，便于系统集成

  异步存储器（ROM/RAM）：

  适合实时控制场景（如外设接口）

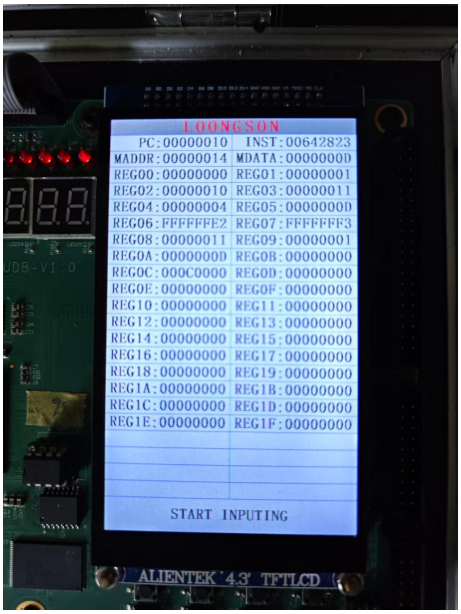
  优势：响应即时，无需等待时钟
- ROM 结构相对简单，单元密度高； RAM 需要额外的读写控制电路，实现更复杂

4.差异原因分析：

- ROM 用于静态程序存储（如 MIPS 指令），无需修改，故牺牲灵活性换取可靠性和密度。
  RAM 需支持动态数据交互（如数组操作），强调读写速度和冲突解决。
- 同步设计（ROM/RAM）通过时钟同步简化时序控制，适合高频系统（如 CPU 缓存）。
  异步设计依赖信号直接触发，实时性强但时序难调试（适合低速外设）。
- ROM 单元结构简单，可高密度集成；RAM 需读写端口和刷新电路，面积开销大。

## （二）单周期 CPU 实验

• 实验箱验证




图为执行完一轮各个寄存器和内存地址为 14h 的值，各个寄存器的值均符合预期。

## （三）自行添加三个指令到周期 CPU

• 仿真测试

```
=== CPU Simulation Test ===
Time     PC       Instruction
10000    00000000    24010001
20000    00000004    00011100
30000    00000008    00411821
40000    0000000c    00022082
50000    00000010    00642823
60000    00000014    ac250013
70000    00000018    00a23027
80000    0000001c    00c33825
90000    00000020    00e64026
100000   00000024    ac08001c
110000   00000028    00c7482a
120000   0000002c    11210002
130000   00000034    8c2a0013
140000   00000038    15450003
150000   0000003c    00415824
160000   00000040    ac0b001c
170000   00000044    ac040010
180000   00000048    3c0c000c
190000   0000004c    0065682b
200000   00000050    0065702c
210000   00000054    00a0782d
220000   00000058    08000000
230000   00000000    24010001

=== Register File Contents ===
R1  = 00000001
R2  = 00000010
R3  = 00000011
R4  = 00000004
R5  = 0000000d
R6  = fffffffe2
R7  = fffffff3
R8  = 00000011
R9  = 00000001
R10 = 0000000d
R11 = 00000000
R12 = 000c0000
R13 = 00000001
R14 = 00000001
R15 = fffffff2
```

寄存器 R13 存储的是 R3 和 R5 的有符号比较结果，当 R3 的值大于 R5 时置为 1，经检验该结果正确；寄存器 R14 存储的是 R3 和 R5 的无符号比较结果，同样在 R3 大于 R5 时置为 1，验证结果正确；寄存器 R15 保存的是 R5 的按位取反值，R5 原值为 0x0000000D，取反后得到 0xFFFFFFF2，计算结果准确无误。

# 六、总结感想

### （1）ROM 存储器实验

通过本次实验，我深入理解了 ROM 的工作原理及其在计算机系统中的关键作用。在同步 ROM 实验中，我掌握了 IP 核的调用方法，并验证了其 2 周期延迟的读取特性；异步 ROM 的零延

迟响应特性让我认识到实时性要求高的场景下存储器的选型要点。与之前 RAM 实验的对比让我更清晰地认识到：

- ROM 的只读特性使其更适合存储固定程序，而 RAM 的读写灵活性适合处理运行时数据；
- 同步设计的时序可控性优于异步设计，但异步设计在实时响应方面更具优势。

实验中遇到的注释错误（如 sll 指令结果应为 0x10 但标注为 0x02）让我意识到二进制验证的重要性。

**（2）单周期 CPU 实验**

通过实现 MIPS 单周期 CPU，我对计算机指令执行的全流程有了系统性认识：

- 通过绘制原理图（图 7.1），我理解了 PC 更新、寄存器读写、ALU 运算等模块的协同工作方式。例如，分析 addu 指令时，需要同步控制寄存器堆、ALU 和写回通路。
- R 型指令的寄存器操作、I 型指令的立即数处理和 J 型指令的跳转控制，让我体会到指令集设计的巧妙之处。
- 在验证 beq 指令时，曾因忽略符号扩展导致跳转地址错误，最终通过波形仿真定位问题。这让我认识到细节处理在硬件设计中的关键性。

**（3）自行添加三个指令到周期 CPU**

为 CPU 添加 SGT（有符号大于置位）、SGTU（无符号大于置位）和 NOT（按位取反）指令的过程充满挑战：

- 需修改 alu.v 增加三个运算单元，并通过 alu_control 信号选择结果。例如，NOT 操作通过~alu_src1 实现，而 SGTU 需结合加法器进位标志。
- 自定义 funct 码时需避开 MIPS 标准编码，最终选择 6'b1010XX 系列。
- 通过仿真确认 R13/R14/R15 的值符合预期（如 R15=0xFFFFFFF2），证明扩展成功。