计算机组成原理第 5 次实验报告

实验名称：存储器和单周期 CPU 实现

学号：2313546　　姓名：蒋枘言　　班次：1078

一、实验目的

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。

二、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，要求：

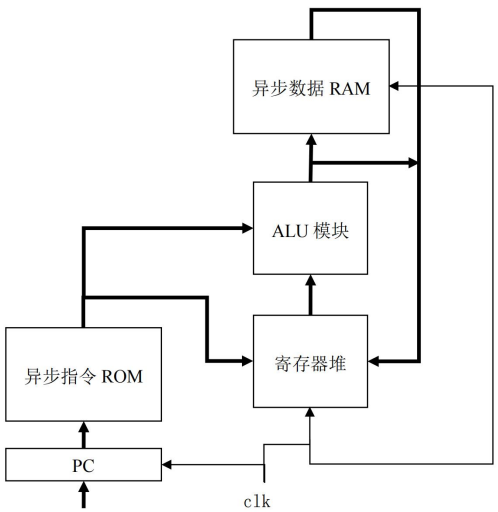
1. ROM 存储器实验请完成验证，总结收获，并对比跟之前的同步异步 RAM 实验有何不同，分析总结原因。

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（单周期 CPU 简单原理图）

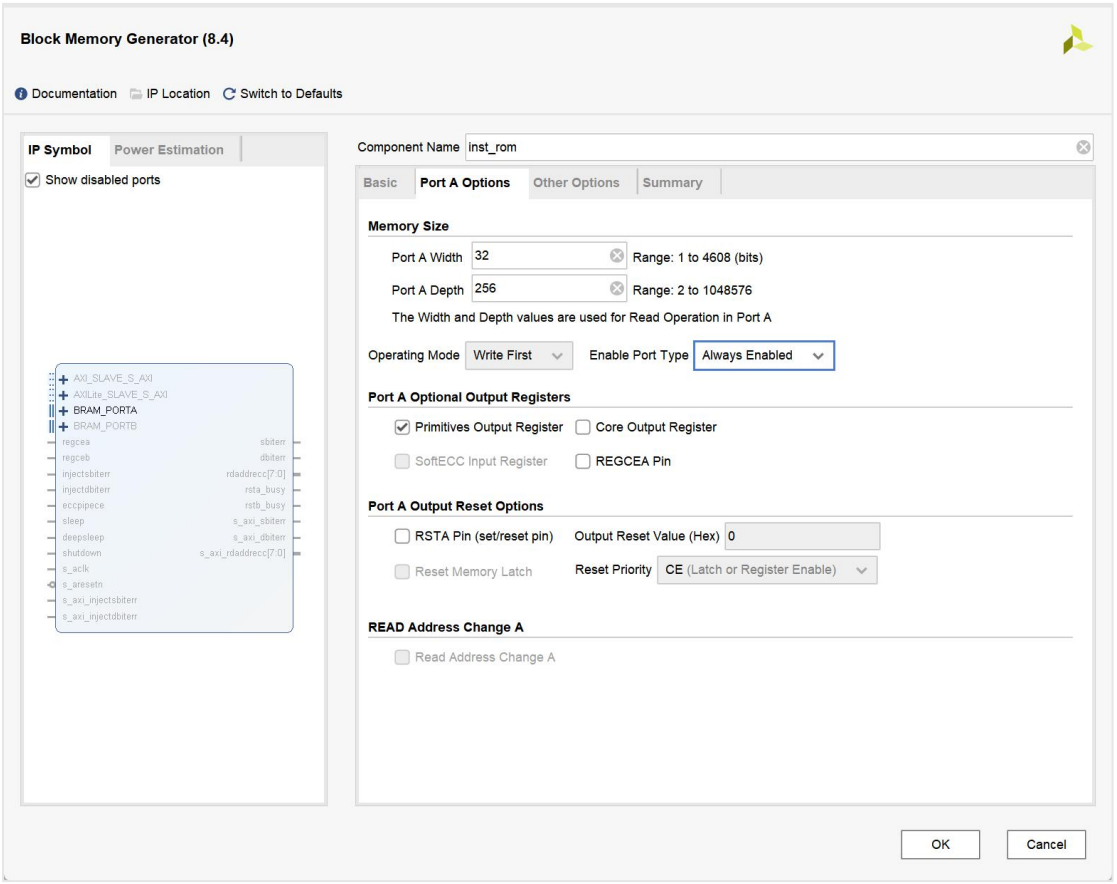


四、实验步骤

（一）ROM 存储器实现

打开 Vivado，对同步 ROM、异步 ROM 分别新建一个工程。按要求添加文件，并且调用 Xilinx 库 IP 生成同步 ROM，对它们进行上箱验证。

对于同步 ROM，需要创建一个 IP 盒，在此之前还需要创建一个.coe 文件，如下图所示：



（二）单周期 CPU 实现

打开 Vivado，新建名称为 single_cycle_cpu 的工程，导入以下文件，进行上箱验证：

名称	修改日期	类型	大小
adder.v	2016/4/29 12:58	V 文件	1 KB
alu.v	2016/4/30 11:03	V 文件	9 KB
data_ram.v	2016/4/30 11:03	V 文件	5 KB
inst_rom.v	2016/5/7 14:14	V 文件	4 KB
regfile.v	2016/4/30 11:02	V 文件	5 KB
single_cycle_cpu.v	2016/5/9 19:53	V 文件	10 KB
single_cycle_cpu.xdc	2017/2/9 11:43	XDC 文件	4 KB
single_cycle_cpu_display.v	2016/11/17 23:11	V 文件	6 KB
tb.v	2016/4/30 11:06	V 文件	2 KB

包含的指令如下：

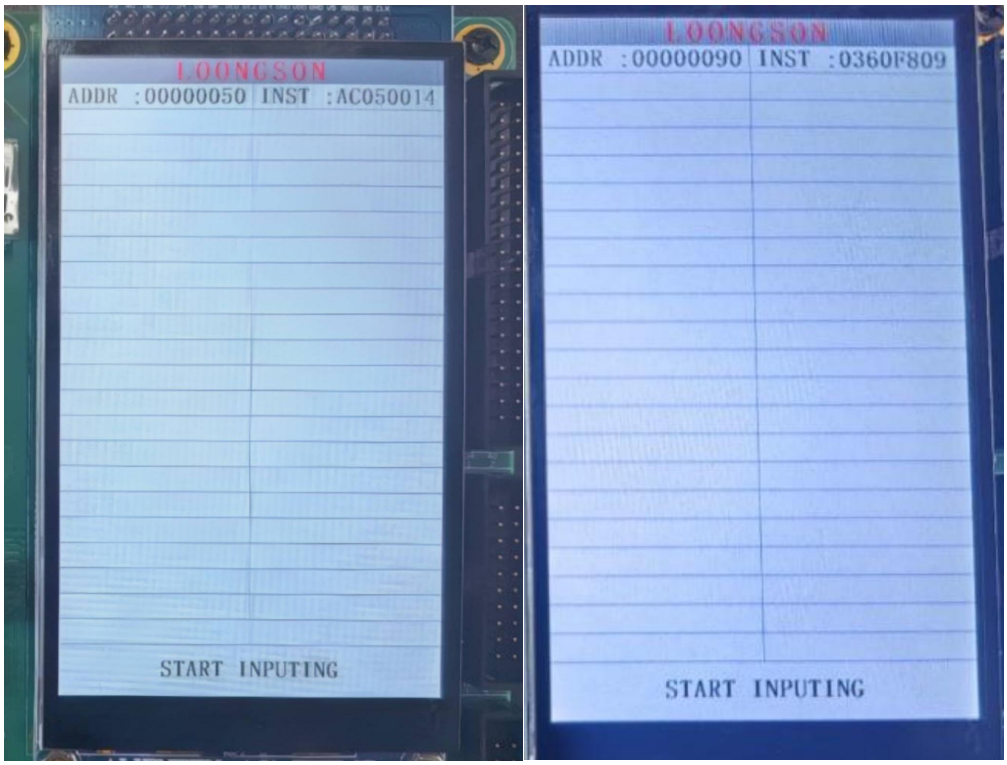
```
//------------ 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果 -----//
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1   | $1 = 0000_0001H
assign inst_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4   | $2 = 0000_0010H
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1   | $3 = 0000_0011H
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
```

五、实验结果分析

（一）ROM 存储器实现

同步 ROM 的结果如下，发现均能正确读取到数据。

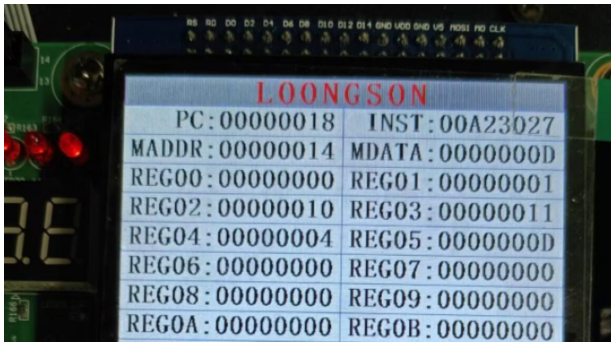
异步 ROM 的结果如下，发现同样能正确读取到数据。



（二）单周期 CPU 实现

每次执行 sw 指令后记录一次当前各个寄存器的值。

sw $5,#19($1)指令结束后，寄存器显示如下：



在这条指令结束后，其前面的代码段完成了对寄存器 $1~$5 的赋值，并将 $5 寄存器的值存在了内存：Mem[0000_0014H]中。

addiu $1,$0,#1 指令执行 $1=$0+1 ，因此寄存器 $1 的值应变为 00000001。

sll $2,$1,#4 指令将 $1 左移四位后再赋值给 $2 ，因此 $2 值应为 00000010。

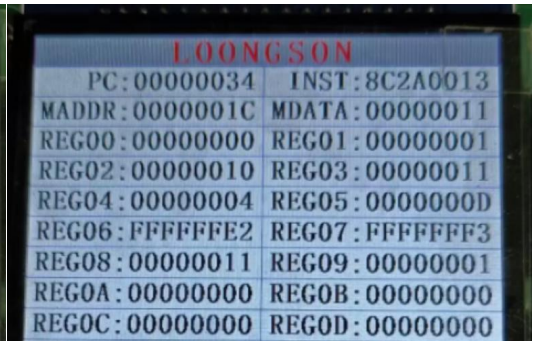
addu $3,$2,$1 指令执行 $3=$2+$1 ,因此 $3 的值应为 00000011。

srl $4,$2,#2 指令把 $2 右移 2 位后传给 $4 ，因此 $4 的值应为 00000004。

subu $5,$3,$4 指令执行 $5=$3-$4 ，因此 $5 的值应为 0000000D。

sw $5,#19($1)指令把 $5 的值存入内存： Mem[0000_0014H] ，为验证该结果，我们把 MADDR 输入 14，读得 MDATA 为 0000000D，写入内存成功。

之后的过程均正确，此处省略一条一条指令的讲解了。



六、总结感想

（一）ROM 与 RAM 对比

1. 数据持久性

ROM：数据永久存储，无需电源维持。

RAM：数据仅在通电时存在。

2. 写入方式

ROM：出厂时写入，写入后不可修改，只可读。

RAM：运行时直接通过 CPU 或外设读写。

3. 访问方式

ROM：按地址顺序读取，随机访问较慢。

RAM：支持高速随机读写。

（二）指令执行过程

1. sll（Shift Left Logical，逻辑左移） 以 sll $2, $1, #4 为例

指令类型：R 型指令

二进制编码

32 位二进制值为：

000000 00000 00001 00010 00100 000000

分解为 R 型指令字段：

opcode: 000000（R 型指令）

rs: 00000（未使用）

rt: 00001（源寄存器$1）

rd: 00010（目标寄存器$2）

shamt: 00100（十进制 4，左移 4 位）

funct: 000000（对应 sll）

执行过程

取指（IF）：从地址 0x04（inst_rom[1]）读取指令。

译码（ID）：识别为 R 型指令（opcode=0），功能码 funct=0 对应 sll。

读取$1 的值（假设$1 = 0x00000002）。

执行（EX）：将$1 的值左移 4 位，0x00000002 << 4 = 0x00000020。

写回（WB）：结果 0x00000020 写入$2 寄存器。

2. lw（Load Word，加载字） 以 lw $10, #19($1)为例

指令类型：I 型指令

二进制编码

32 位二进制值为：

100011 00001 01010 0000000000010011

分解为 I 型指令字段：

opcode: 100011（对应 lw）

rs: 00001（基址寄存器$1）

rt: 01010（目标寄存器$10）

immediate: 0000000000010011（十进制 19，符号扩展为 32 位）

执行过程

取指（IF）：从地址 0x34（inst_rom[13]）读取指令。

译码（ID)：

识别为 lw 指令（opcode=0x23）。

读取基址寄存器$1 的值（假设$1 = 0x00000000）。

地址计算（EX）：

address = $1 + 19 = 0x00000000 + 0x13 = 0x00000013。

访存（MEM）：从内存地址 0x13 读取 32 位数据（假设内存中 0x13 存储值为 0x0000000D）。

写回（WB）：将 0x0000000D 写入$10 寄存器。

3. j（Jump，无条件跳转） 以 j 00H 为例

指令类型：J 型指令

二进制编码

32 位二进制值为：

000010 00000000000000000000000000

分解为 J 型指令字段：

opcode: 000010（对应 j）

target: 00000000000000000000000000（目标地址高 26 位）

执行过程

取指（IF）：从地址 0x4C（inst_rom[19]）读取指令。

译码（ID）：识别为 j 指令（opcode=0x02）。

目标地址计算（EX）：

当前指令地址为 0x4C，PC+4 = 0x50（下一条指令地址）。

取 PC+4 的高 4 位（0x50 的二进制高 4 位为 0000）。

拼接目标地址：0000 || (target << 2) = 0x00000000。

更新 PC：将 PC 设置为 0x00，程序跳转到地址 0x00 执行。