第一章 实验目的

- 了解只读存储器 ROM 和随机存取存储器 RAM 的原理。
- 理解 ROM 读取数据及 RAM 读取、写入数据的过程。
- 理解计算机中存储器地址编址和数据索引方法。
- 理解同步 RAM 和异步 RAM 的区别。
- 掌握调用 Xilinx 库 IP 实例化 RAM 的设计方法。
- 熟悉并运用 verilog 语言进行电路设计。
- 为后续设计 cpu 的实验打下基础。

第二章 实验任务与要求

2.1 试验任务

- 学习存储器的设计及原理，如：ROM 读地址索引读取数据过程及时序，RAM 读写时序，同步和异步的区别等。
- 学习计算机中内存地址编址和数据索引方法。
- 自行设计本次实验的方案，画出结构框图，详细标出输入输出端口，确定存储器宽度、深度和写使能位数。
- 学习 Vivado 工具中调用库 IP 的方法。
- 本次实验要求调用 Xilinx 库 IP 实例化一块 RAM。实例化的 RAM 选择为同步 RAM。本次实验的 RAM 建议设置为两个端口，一个端口用来正常的读写，另一个端口作为调试端口只使用读功能用于观察存储器内部数据。
- 调用 Xilinx 库 IP 实例化一块 RAM，并进行仿真，得到正确的波形图。
- 将以上设计作为一个单独的模块，设计一个外围模块去调用该模块。外围模块中需调用封装好的 LCD 触摸屏模块，显示 RAM 的正常端口的地址、待写入的数据和读出的数据，显示调试端口的地址和读出的数据。并且需要利用触摸功能输入正常端口的地址和写数据，以及调试端口的地址
- 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。

注意：存储器深度不要过大，避免耗费过多的 FPGA 上的资源。本次实验要求实现同步的存储器。而异步存储器的搭建方法同寄存器堆的搭建，但不同的是，寄存器堆中读写端口是分开的，但对于异步 RAM 要求读写共用一个端口，只是会增加一个写使能信号。可以自行尝试搭建异步的 ROM 和 RAM，在单周期 CPU 实验中会用到异步的 ROM 作为指令存储器，而异步 RAM 作为数据存储器。

2.2 实验要求

1. 实验箱结果截图
2. 核心代码注释
3. 解释同步方式和异步方式访问存储时的差异（控制信号以及访问流程）

第三章 实验结果

3.1 实验代码与注释

3.1.1 data ram

```
`timescale 1ns / 1ps

module data_ram(
    input          clk,          // 时钟
    input  [3:0]   wen,          // 字节写使能
    input  [4:0] addr,          // 地址
    input  [31:0] wdata,         // 写数据
    output reg [31:0] rdata,         // 读数据

    //调试端口，用于读出数据显示
    input  [4 :0] test_addr,
    output reg [31:0] test_data
);
    reg [31:0] DM[31:0];  //数据存储器，字节地址7'b000_0000~7'b111_1111

    //写数据
    always @(posedge clk)     // 当写控制信号为1，数据写入内存
    begin
        if (wen[3])
        begin
            DM[addr][31:24] <= wdata[31:24];
        end
    end
    always @(posedge clk)
    begin
        if (wen[2])
        begin
            DM[addr][23:16] <= wdata[23:16];
        end
```

```
30      end
31      always @(posedge clk)
32      begin
33          if (wen[1])
34          begin
35              DM[addr][15: 8] <= wdata[15: 8];
36          end
37      end
38      always @(posedge clk)
39      begin
40          if (wen[0])
41          begin
42              DM[addr][7 : 0] <= wdata[7 : 0];
43          end
44      end
45
46      //读数据,取4字节
47      always @(*)
48      begin
49          case (addr)
50              5'd0 : rdata <= DM[0 ];
51              5'd1 : rdata <= DM[1 ];
52              5'd2 : rdata <= DM[2 ];
53              5'd3 : rdata <= DM[3 ];
54              5'd4 : rdata <= DM[4 ];
55              5'd5 : rdata <= DM[5 ];
56              5'd6 : rdata <= DM[6 ];
57              5'd7 : rdata <= DM[7 ];
58              5'd8 : rdata <= DM[8 ];
59              5'd9 : rdata <= DM[9 ];
60              5'd10: rdata <= DM[10];
61              5'd11: rdata <= DM[11];
62              5'd12: rdata <= DM[12];
63              5'd13: rdata <= DM[13];
64              5'd14: rdata <= DM[14];
65              5'd15: rdata <= DM[15];
66              5'd16: rdata <= DM[16];
```

```
            5'd17: rdata <= DM[17];
            5'd18: rdata <= DM[18];
            5'd19: rdata <= DM[19];
            5'd20: rdata <= DM[20];
            5'd21: rdata <= DM[21];
            5'd22: rdata <= DM[22];
            5'd23: rdata <= DM[23];
            5'd24: rdata <= DM[24];
            5'd25: rdata <= DM[25];
            5'd26: rdata <= DM[26];
            5'd27: rdata <= DM[27];
            5'd28: rdata <= DM[28];
            5'd29: rdata <= DM[29];
            5'd30: rdata <= DM[30];
            5'd31: rdata <= DM[31];
        endcase
    end
    //调试端口，读出特定内存的数据
    always @(*)
    begin
        case (test_addr)
            5'd0 : test_data <= DM[0 ];
            5'd1 : test_data <= DM[1 ];
            5'd2 : test_data <= DM[2 ];
            5'd3 : test_data <= DM[3 ];
            5'd4 : test_data <= DM[4 ];
            5'd5 : test_data <= DM[5 ];
            5'd6 : test_data <= DM[6 ];
            5'd7 : test_data <= DM[7 ];
            5'd8 : test_data <= DM[8 ];
            5'd9 : test_data <= DM[9 ];
            5'd10: test_data <= DM[10];
            5'd11: test_data <= DM[11];
            5'd12: test_data <= DM[12];
            5'd13: test_data <= DM[13];
            5'd14: test_data <= DM[14];
            5'd15: test_data <= DM[15];
```

```
            5'd16: test_data <= DM[16];
            5'd17: test_data <= DM[17];
            5'd18: test_data <= DM[18];
            5'd19: test_data <= DM[19];
            5'd20: test_data <= DM[20];
            5'd21: test_data <= DM[21];
            5'd22: test_data <= DM[22];
            5'd23: test_data <= DM[23];
            5'd24: test_data <= DM[24];
            5'd25: test_data <= DM[25];
            5'd26: test_data <= DM[26];
            5'd27: test_data <= DM[27];
            5'd28: test_data <= DM[28];
            5'd29: test_data <= DM[29];
            5'd30: test_data <= DM[30];
            5'd31: test_data <= DM[31];
        endcase
    end
endmodule
```

3.1.2 inst rom

```
`timescale 1ns / 1ps

module inst_rom(
    input       [4 :0] addr, // 指令地址
    output reg [31:0] inst       // 指令
    );

    wire [31:0] inst_rom[19:0];  // 指令存储器, 字节地址7'b000_0000~7'b111_1111
    //------------- 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
```

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

```
31      //读指令,取4字节
32      always @(*)
33      begin
34          case (addr)
35              5'd0 : inst <= inst_rom[0 ];
36              5'd1 : inst <= inst_rom[1 ];
37              5'd2 : inst <= inst_rom[2 ];
38              5'd3 : inst <= inst_rom[3 ];
39              5'd4 : inst <= inst_rom[4 ];
40              5'd5 : inst <= inst_rom[5 ];
41              5'd6 : inst <= inst_rom[6 ];
42              5'd7 : inst <= inst_rom[7 ];
43              5'd8 : inst <= inst_rom[8 ];
44              5'd9 : inst <= inst_rom[9 ];
45              5'd10: inst <= inst_rom[10];
46              5'd11: inst <= inst_rom[11];
47              5'd12: inst <= inst_rom[12];
48              5'd13: inst <= inst_rom[13];
49              5'd14: inst <= inst_rom[14];
50              5'd15: inst <= inst_rom[15];
51              5'd16: inst <= inst_rom[16];
52              5'd17: inst <= inst_rom[17];
53              5'd18: inst <= inst_rom[18];
54              5'd19: inst <= inst_rom[19];
55              default: inst <= 32'd0;
56          endcase
57      end
58  endmodule
```

3.2 上机结果

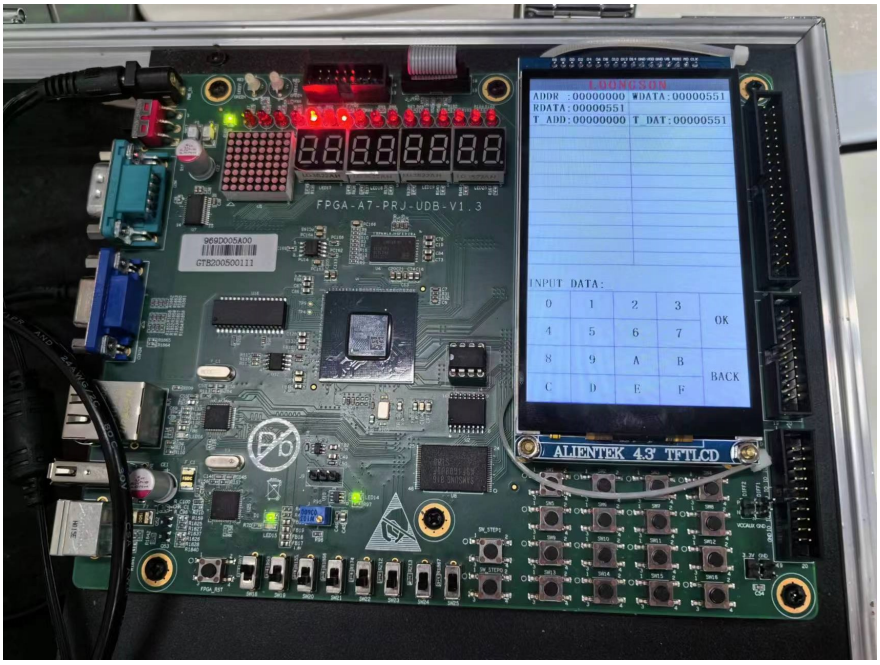
LOONGSON
ADDR :00000000 WDATA:00000551
RDATA:00000551
T_ADD:00000000 T_DAT:00000551
INPUT DATA:
0 1 2 3
4 5 6 7
8 9 A B
C D E F
OK
BACK
ALIENTEK 4.3' TFTLCD
FPGA-A7-PRJ-UDB-V1.3

图 3.1 上机结果

第四章 思考与讨论

同步访问

同步访问是指数据传输或访问过程与系统时钟同步进行。在同步系统中，所有操作（如读取和写入）都在时钟信号的特定边缘触发，例如在时钟的上升沿或下降沿。

控制信号：

- **时钟信号（Clock）**：这是同步访问的核心，所有操作都根据这个时钟信号的节拍进行。
- **使能信号（Enable）**：决定何时可以对存储设备进行访问。
- **读/写信号（Read/Write）**：指示进行的是读操作还是写操作。

访问流程：

1. 系统时钟产生节拍：确定操作的具体时机。
2. 设置地址和数据：在时钟信号到来之前，地址和数据线被设置。
3. 使能信号激活：允许数据在指定的时钟边缘被读取或写入。
4. 读/写操作执行：根据时钟信号的触发，执行读或写操作。
5. 数据稳定：数据在下一个时钟信号到来前保持稳定，以确保数据正确读取或写入。

异步访问

异步访问不依赖于系统的主时钟信号，而是使用控制信号来直接管理读写操作的时序。

控制信号：

- **请求信号（Request）**：用来从主设备向存储设备发起访问请求。
- **就绪/应答信号（Ready/Acknowledge）**：存储设备用它来响应主设备的请求，表示数据已经准备好或已成功接收。
- **读/写信号**：与同步方式相同，指示进行的是读操作还是写操作。

访问流程：

1. 请求信号发起：主设备发出请求信号，要求进行数据读取或写入。
2. 设置地址和数据：如果是写操作，数据会与请求一同发送。
3. 等待就绪/应答信号：存储设备在数据准备就绪或接收完毕后，发回应答信号。
4. 执行读/写操作：在接收到应答信号后，完成数据传输。
5. 完成操作：操作完成后，请求和应答信号被撤销，系统进入下一状态。

总结

同步访问与系统时钟紧密相关，需要所有操作严格按照时钟信号进行，适合于速度要求严格且时序要求高的环境。异步访问则更灵活，不依赖时钟信号，适合于时钟不方便传

递或者设备之间时钟不同步的情况。两者的选择取决于具体的系统设计和

附　　录

参考链接：https://github.com/zehua0417/ComputerOrganizationAndArchitecture_exp