实验四 存储器阵列设计

一、实验目的

1 掌握 Verilog 语言和 Vivado、Logisim 开发平台的使用；

2 掌握存储器和寄存器组的设计和测试方法。

二、实验内容

1 存储器设计与测试

2 寄存器组设计与测试

三、实验要求

1 掌握 Vivado 或 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

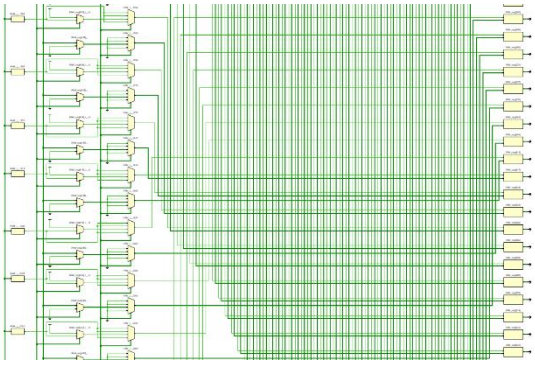
2 记录设计和调试过程（Verilog 代码/电路图/表达式/真值表，Vivado 仿真结果，Logisim 验证结果等）；

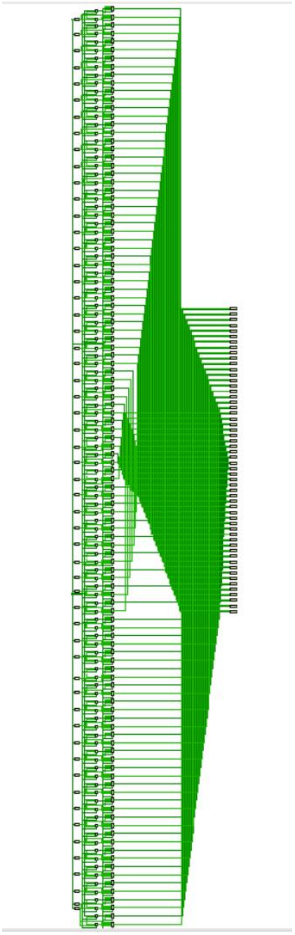
3 分析 Vivado 仿真波形/Logism 验证结果，注重输入输出之间的对应关系。

四、实验过程及分析

存储器设计：

```
22
23  module RAM_1Kx16_inout(Data, Addr, Rst, R_W, CS, CLK);
24      parameter Addr_Width=10;
25      parameter Data_Width=16;
26      parameter SIZE=2 ** Addr_Width;
27      inout [Data_Width-1:0] Data;
28      input [Addr_Width-1:0] Addr;
29      input Rst;
30      input R_W;
31      input CS;
32      input CLK;
33      integer i;
34      reg [Data_Width-1:0] Data_i;
35      reg [Data_Width-1:0] RAM [SIZE-1:0];
36      assign Data = (R_W) ? Data_i: 16'bz;
37      always @ (*) begin
38          casex({CS,Rst,R_W})
39              4'bx1x : for(i = 0; i <= SIZE-1;i = i+1) RAM[i] = 0;
40              4'b101: Data_i<=RAM [Addr];
41              4'b100:RAM[Addr]<=Data;
42              default : Data_i = 16'bz;
43          endcase
44      end
45  endmodule
46
```





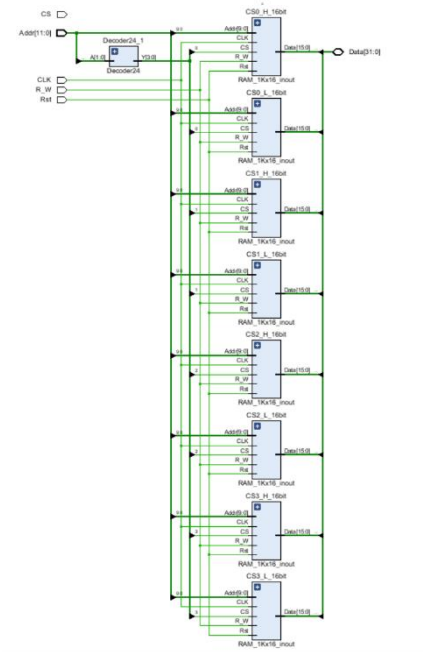
这 Verilog 模块实现了一个 1K x 16 位的双向 RAM（Random Access Memory）。
Data_i 寄存器用于存储从 RAM 中读取的数据。
Data：双向端口，用于连接 RAM 的数据总线。
Addr：输入端口，用于指定 RAM 的读写地址。
Rst：输入端口，用于复位整个 RAM，将存储单元清零。
R_W：输入端口，用于控制 RAM 的读写操作。
CS：输入端口，用于片选，通过此信号控制 RAM 的操作是否生效。
CLK：输入端口，时钟信号。
使用状态机实现不同的操作：
当 CS，Rst，R_W 组合为 4'bx1x 时，执行循环将整个 RAM 清零。
当 CS，Rst，R_W 组合为 4'b101 时，将 RAM 中地址为 Addr 的数据赋给 Data_i。
当 CS，Rst，R_W 组合为 4'b100 时，将 Data 的值写入 RAM 中地址为 Addr 的位置。
对于其他组合，将 Data_i 置为高阻态。
使用 assign Data = (R_W) ? Data_i : 16'bz; 根据读写控制信号 R_W 的状态，将 Data_i 的值赋给 Data（读操作），或者将 Data 置为高阻态（写操作）。
这个模块允许在指定的地址执行读写操作，可以通过 Rst 信号将整个 RAM 清零，同时支持片选信号 CS 控制是否执行 RAM 操作。

字扩展与位扩展：

```
module RAM_4Kx32_inout
    #(parameter Addr_Width=12,
    Data_Width=32)
    (inout [Data_Width-1: 0] Data,
    input [Addr_Width-1: 0] Addr,
    input Rst,
    input R_W,
    input CS,
    input CLK);
    wire [3:0] CS_i;
    Decoder24 Decoder24_1(CS_i, Addr[Addr_Width-1: Addr_Width-2]);

    RAM_1Kx16_inout CS0_H_16bit(Data[Data_Width-1: Data_Width/2], Addr[Addr_Width-3:0], Rst, R_W, CS_i[0], CLK),
                    CS0_L_16bit(Data[Data_Width/2-1: 0], Addr[Addr_Width-3:0], Rst, R_W, CS_i[0], CLK);
    RAM_1Kx16_inout CS1_H_16bit(Data[Data_Width-1: Data_Width/2], Addr[Addr_Width-3:0], Rst, R_W, CS_i[1], CLK),
                    CS1_L_16bit(Data[Data_Width/2-1: 0], Addr[Addr_Width-3:0], Rst, R_W, CS_i[1], CLK);
    RAM_1Kx16_inout CS2_H_16bit(Data[Data_Width-1: Data_Width/2], Addr[Addr_Width-3:0], Rst, R_W, CS_i[2], CLK),
                    CS2_L_16bit(Data[Data_Width/2-1: 0], Addr[Addr_Width-3:0], Rst, R_W, CS_i[2], CLK);
    RAM_1Kx16_inout CS3_H_16bit(Data[Data_Width-1: Data_Width/2], Addr[Addr_Width-3:0], Rst, R_W, CS_i[3], CLK),
                    CS3_L_16bit(Data[Data_Width/2-1: 0], Addr[Addr_Width-3:0], Rst, R_W, CS_i[3], CLK);

endmodule
```

```
module Decoder24(
    output reg [3:0] Y,
    input [1:0] A
);

always @*
    case (A)
        2'b00: Y = 4'b0001;
        2'b01: Y = 4'b0010;
        2'b10: Y = 4'b0100;
        2'b11: Y = 4'b1000;
        default: Y = 4'b0000;
    endcase

endmodule
```

```
module sim_4K;
    parameter Addr_Width=12,Data_Width=32;
    reg Rst,R_W,CS,CLK;
    reg [Addr_Width-1:0]Addr;
    wire [Data_Width-1:0]Data;
    reg [Data_Width-1:0]Data_i;

    RAM_4Kx32_inout RAM1(Data,Addr,Rst,R_W,CS,CLK);

    assign Data = (R_W) ? 32'bz: Data_i;

    initial begin
        Rst=1;
        CS=1;
        R_W=0;
        Data_i=0;
        CLK=1;
        Addr=0;
        #10 Rst=0;
        for (Addr=0;Addr<=5;Addr=Addr+1) #10 Data_i=Data_i+1;
        #10 R_W=1;
        for (Addr=0;Addr<=10;Addr=Addr+1) #10 ;
    end

endmodule
```



Decoder24:

声明了一个 Verilog 模块，模块名为 Decoder24，有两个端口：输出端口 Y（4 位寄存器）和输入端口 A（2 位寄存器）。
使用 always @*表示组合逻辑，根据输入 A 的值进行不同的解码操作：

当 A 为 2'b00 时，输出 Y 为 4'b0001。
当 A 为 2'b01 时，输出 Y 为 4'b0010。
当 A 为 2'b10 时，输出 Y 为 4'b0100。
当 A 为 2'b11 时，输出 Y 为 4'b1000。
对于其他情况，使用 default 语句将输出 Y 置为 4'b0000。
总结：
这个模块实现了一个 2-4 译码器，将 2 位的输入编码为 4 位的输出。根据不同的输入值，输出的 4 位二进制数 Y 有不同的值。此模块适用于数字电路设计中的译码器应用，可以将输入的 2 位二进制数映射到 4 位输出。

RAM_4K32:

声明了一个 Verilog 模块，模块名为 RAM_4Kx32_inout，有六个端口，包括数据端口 Data、地址端口 Addr、复位端口 Rst、读写控制端口 R_W、片选端口 CS 和时钟端口 CLK。此外，通过参数 Addr_Width 和 Data_Width 指定了地址宽度和数据宽度。

通过实例化 Decoder24 模块，将 Addr 的高两位用于片选信号 CS_i 的解码。
实例化了八个 RAM_1Kx16_inout 模块，用于处理每个片选信号对应的数据。每个模块接收相应的高位和低位的数据，并使用相应的 CS_i 信号，通过相同的时钟信号 CLK 进行操作。

这个模块实现了一个 4K x 32 位的双向 RAM，通过实例化八个 1K x 16 位的 RAM 子模块，通过 Decoder24 将地址的高两位解码成四个片选信号。每个子模块对应一个片选信号，用于处理相应的数据。这样的设计使得 4K x 32 位的 RAM 可以分为八个 1K x 16 位的 RAM，并且每个部分可以独立地进行读写操作。

Sim_4K:

声明了一个 Verilog 模块，模块名为 sim_4K，并定义了两个参数：Addr_Width 为 12，Data_Width 为 32。
声明了一些寄存器和线网，其中 Rst、R_W、CS、CLK、Addr 为输入端口，Data 为输出端口，Data_i 为内部寄存器。
实例化了一个 RAM_4Kx32_inout 模块，通过端口连接与 sim_4K 模块的相应信号和数据。
根据读写控制信号 R_W 的状态，将 Data 连接到 Data_i 或置为高阻态。

在仿真开始时，进行了一系列的初始化操作：

Rst 被置为 1，表示复位。
CS 被置为 1，表示不选中 RAM。
R_W 被置为 0，表示写入操作。
Data_i 被置为 0，作为写入数据的起始值。
CLK 被置为 1，表示时钟信号。
Addr 被置为 0，作为起始地址。
然后，通过#10 的延时，进行了一些状态的切换：

Rst 被置为 0，表示结束复位。
使用 for 循环，从地址 0 开始，每隔 10 个时间单位，Data_i 增加 1，总共执行 6 次。
R_W 被置为 1，表示读取操作。
使用 for 循环，从地址 0 开始，每隔 10 个时间单位，执行了 11 次。
总结：
这个模块实现了对 RAM_4Kx32_inout 模块的简单仿真。在初始化过程中，进行了一系列的状态设置和数据写入操作，以模拟 RAM 的读写过程。

时序输出关系：
Rst 在初始时为 1，表示 RAM 处于复位状态，此时不进行读写操作。
CS 在初始时为 1，表示不选中 RAM，也就是 RAM 不响应读写操作。
R_W 在初始时为 0，表示写入操作。在 for 循环中，每隔 10 个时间单位，Data_i 递增，相应地写入到 RAM 中。这模拟了对 RAM 的写入过程。
在 R_W 被置为 1 后，表示读取操作。在 for 循环中，每隔 10 个时间单位，没有进行实际的写入或读取操作，因为 for 循环中没有相关的操作。这阶段用于观察读取操作的影响。

综上所述，仿真的时序输出关系为：

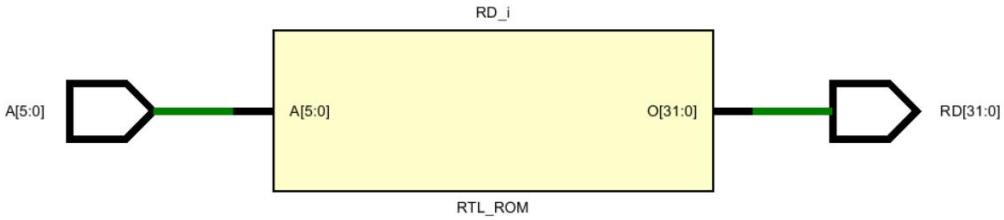
初始时，RAM 被置为复位状态，不进行读写操作。
随后，进行了一系列的写入操作，模拟了写入过程。
最后，进行了一系列的读取操作，模拟了读取过程。

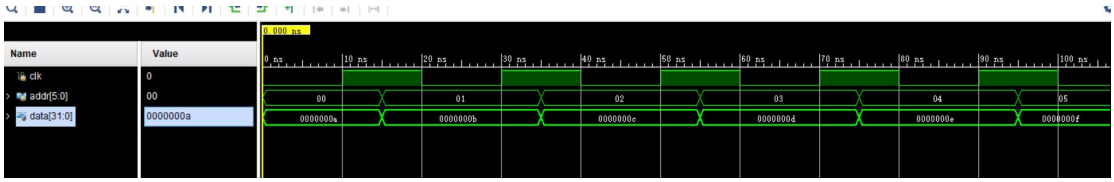
可以看到时序图所示：
地址 addr 在循环递增，一开始 R_W 置于写入状态，向 data 中写入数据，后来 R_W 置于读取状态，data 读出前面写入的 addr 处的数值，而 data_i 保持不变。

指令存储器设计：

```
`define DATA_WIDTH 32
module IMem(
    input [5: 0] A,
    output [`DATA_WIDTH -1:0] RD);
    parameter IMEM_SIZE = 64;

    reg [`DATA_WIDTH-1: 0] RAM[IMEM_SIZE-1:0];
    integer i;
    initial begin
        for(i=0; i<`DATA_WIDTH; i=i+1) RAM[i] = i+10;
    end
    assign RD=RAM [A];

endmodule
```

```
`define DATA_WIDTH 32
module sim_imem;
    reg clk;
    reg [5:0] addr;
    wire [`DATA_WIDTH-1:0] data;

    IMem IMem_inst (
        .A(addr),
        .RD(data)
    );

    initial begin
        clk = 0;
        addr = 0;
        repeat (10) begin
            #10 clk = ~clk;
        end
    end

    always @(posedge clk) begin
        #5 addr <= addr + 1;
    end
endmodule
```





这段 Verilog 代码定义了一个指令存储器模块 IMem

模块声明：模块名为 IMem，有一个 6 位输入端口 A 用于指定读取的存储器地址，和一个 32 位输出端口 RD 用于输出从存储器读取的数据。

参数声明：定义了参数 IMEM_SIZE，表示存储器的大小为 64 个 32 位的存储单元。

内部存储器声明：声明了一个名为 RAM 的内部存储器，是一个大小为 IMEM_SIZE 的数组，每个元素是 32 位宽度。

初始化块：在仿真开始时，使用 for 循环，将 RAM 的前 DATA_WIDTH 个元素初始化为递增的值，即 RAM[0] 初始化为 10，RAM[1] 初始化为 11，以此类推。

数据输出：通过 assign 语句，将存储器中地址为 A 的数据赋给输出端口 RD。这表示在每个时钟周期，从指定地址读取存储器中的数据。

这个模块的作用是模拟一个简单的指令存储器，可以根据输入的地址 A 从存储器中读取相应的 32 位指令，并将其输出到 RD 端口。在初始化时，存储器的前 DATA_WIDTH 个位置被初始化为递增的值。

定义了一个名为 sim_imem 的仿真模块，包含了一个时钟信号 clk、一个 6 位宽的地址信号 addr 以及一个 32 位宽的数据线 data。实例化了 IMem 模块，并将其输入输出端口与仿真模块的信号相连接。

在仿真开始时，初始化了时钟信号 clk 为 0，地址信号 addr 为 0。然后通过 repeat 循环，每次循环都在时钟上升沿（每隔 10 个时间单位）切换 clk 的状态，即从 0 切换到 1 或从 1 切换到 0。这样，通过 clk 的变化来模拟时钟信号。

在时钟上升沿发生时，addr 的值递增 1。通过 #5 的延时，表示在下一个时钟上升沿前进行这个操作。这段代码模拟了在时钟上升沿时不断递增地址的行为。

总结时序输出关系：

初始时，clk 为 0，addr 为 0。

通过 repeat 循环，clk 在每个循环中发生状态切换（0->1 或 1->0）。

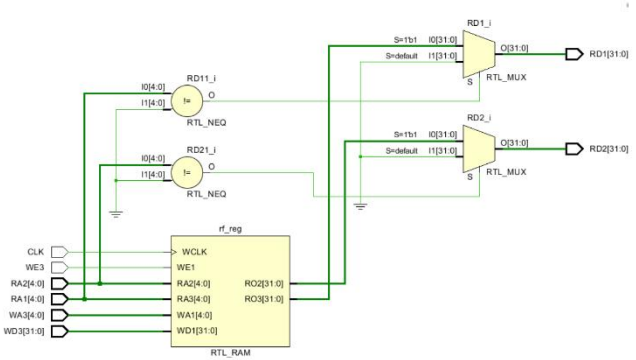
在每个时钟上升沿，addr 递增 1。

通过 IMem 模块实例，data 会根据 addr 的值从指令存储器中读取相应的 32 位数据。

由时序图可以看出每过 15ns，会从 ram 中读出一个 addr 地址处的数据，在 design 文件中可以看出 ram 初始化的赋值，所以从 a 开始输出，一直递增。

寄存器文件设计：

```
`define DATA_WIDTH 32
module RegFile
  #(parameter ADDR_SIZE=5)
  (input CLK,WE3,
  input [ADDR_SIZE-1:0]RA1,RA2,WA3,
  input [`DATA_WIDTH-1:0]WD3,
  output [`DATA_WIDTH-1:0]RD1,RD2);

  reg[`DATA_WIDTH-1:0]rf[2**ADDR_SIZE-1:0];

  always@(posedge CLK)
       if(WE3) rf[WA3] <=WD3;
  assign RD1=(RA1 !=0)? rf[RA1]:0;
  assign RD2=(RA2 !=0)? rf[RA2]:0;
endmodule
```




```
module sim_reg;
    reg CLK;
    reg WE3;
    reg [4:0] RA1, RA2,WA3;
    reg [31:0] WD3;
    wire [31:0] RD1, RD2;

    RegFile U1 (
      .CLK(CLK),
      .WE3(WE3),
      .RA1(RA1),
      .RA2(RA2),
      .WA3(WA3),
      .WD3(WD3),
      .RD1(RD1),
      .RD2(RD2)
    );

    initial begin
        CLK = 0;
        WE3 = 0;
        RA1=4'b0000;
        RA2=4'b0000;
        WA3=4'b0000;
        WD3=32'h12345678;
        #9 WA3=4'b0011;
        #1 RA1=4'b0011;
        #60 RA2=4'b0001;
        WD3=32'h88888888;
        WA3=4'b0001;

    end
```

声明了一个名为 RegFile 的 Verilog 模块，其中包含了一系列输入和输出端口。CLK 是时钟信号，WE3 是写使能信号，RA1 和 RA2 是两个读地址输入，WA3 是写地址输入，WD3 是写数据输入，RD1 和 RD2 是两个读数据输出。
使用 parameter 关键字声明了一个参数 ADDR_SIZE，默认值为 5。这个参数指定了地址的位宽。
声明了一个名为 rf 的内部寄存器数组，每个寄存器是 DATA_WIDTH 位宽，数组的大小为 2^ADDR_SIZE。这实现了一个简单的寄存器文件。
在时钟的上升沿，如果写使能信号 WE3 为 1，将写数据 WD3 写入到寄存器文件的相应写地址 WA3 处。
通过 assign 语句，实现了读数据的输出逻辑。如果读地址 RA1 不为 0，则将寄存器文件中的相应位置的数据赋给 RD1，否则 RD1 被赋值为 0。同样的逻辑应用在 RD2 上。

总结：
这个模块实现了一个带有写使能的寄存器文件，可以进行读和写操作。在每个时钟的上升沿，如果写使能信号为 1，将写数据写入到指定的写地址。读操作通过比较读地址是否为 0 来确定是否进行读取。

仿真：

实例化了 RegFile 模块，并将其输入输出端口连接到仿真模块的信号。CLK 是时钟信号，WE3 是写使能信号，RA1 和 RA2 是两个读地址输入，WA3 是写地址输入，WD3 是写数据输入，RD1 和 RD2 是两个读数据输出。
在仿真开始时，初始化了时钟信号 CLK 为 0，写使能信号 WE3 为 0，读写地址和写数据信号也被初始化。然后在仿真过程中，通过 # 符号的延时，改变了各个信号的值，模拟了时钟周期内的变化。
在仿真的每个 10 个时间单位的时钟周期内，时钟信号 CLK 发生状态切换（0->1 或 1->0）。这个语句模拟了时钟的边沿生成。

时序输出关系：
时钟信号 CLK：在每个 10 个时间单位的时钟周期内，时钟信号 CLK 发生状态切换。
写使能信号 WE3：在每个 10 个时间单位的时钟周期内，写使能信号 WE3 发生状态切换。
读写地址和写数据信号：在仿真开始时，读写地址和写数据信号被初始化。在仿真过程中，通过延时和指定的时间序列，这些信号的值被改变，模拟了读写地址和写数据的变化。
读数据信号 RD1 和 RD2：在仿真过程中，通过 RegFile 模块实例进行读操作，读数据信号 RD1 和 RD2 的值会随着读地址的变化而改变。在 #1 RA1 = 4'b0011; 处，RA1 的值变

为 4'b0011,因此 RD1 的值将对应于寄存器文件中相应位置的数据。在 #60 RA2 = 4'b0001; 处，RA2 的值变为 4'b0001，因此 RD2 的值将对应于寄存器文件中相应位置的数据。

写数据信号 WD3：在仿真过程中，通过 WD3 = 32'h88888888; 和 WA3 = 4'b0001; 的操作，将写数据信号 WD3 写入到寄存器文件的相应写地址 WA3 处。

五、调试和心得体会

在本次计算机组成实验中，我深入学习了硬件描述语言（Verilog）的基本概念和用法，并通过设计和仿真实际的硬件电路，深入理解了计算机组成原理的一些关键概念。

首先，我了解了 Verilog 语言的基础知识，包括模块化设计、数据类型、参数化、内部存储器的使用等。Verilog 的模块化设计允许我将电路划分为不同的模块，提高了代码的可读性和可维护性。

其次，通过实现 RAM 模块和寄存器文件模块，我加深了对存储器和寄存器的理解。在 RAM 模块中，我学会了如何使用 Verilog 描述存储器的读写过程，以及如何使用 casex 语句处理不同的控制信号。在寄存器文件模块中，我了解了时序逻辑的实现，以及如何通过时钟信号和写使能信号进行同步写入操作。

实验中，我还学到了如何编写测试文件进行仿真。通过对各个模块的功能进行测试，我能够验证设计的正确性，发现潜在的问题并进行调试。

最后，在整个实验的过程中，我深刻体会到硬件设计的精确性和时序的重要性。每一个模块都需要按照时序要求正确地执行，否则可能导致电路不稳定或不正确的结果。同时，我对计算机组成原理的知识有了更深层次的理解，通过实际设计和仿真，我更好地掌握了理论知识的应用。

总体而言，这次计算机组成实验为我提供了一个很好的机会，让我在实际操作中学到了很多关于硬件设计和 Verilog 语言的知识。这不仅增强了我对计算机组成原理的理解，也提升了我的实际问题解决能力。通过这次实验，我对硬件描述语言和计算机硬件设计有了更深刻的认识，这对我的学术和职业发展都具有重要的意义。