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

1.1K×16bit 存储器

设计代码

```
module RAM_1Kx16_inout(Data,Addr,Rst,R_W,CS,CLK);
    parameter Addr_Width =10;
    parameter Data_Width =16;
    parameter SIZE=2**Addr_Width;
    input [Data_Width-1:0] Data;
    input [Addr_Width-1:0] Addr;
    input Rst;
    input R_W;
    input CS;
    input CLK;
    integer i;
    reg [Data_Width-1:0] Data_i;
    reg [Data_Width-1:0] RAM [SIZE-1:0];

    assign Data = (R_W)?Data_i:16'bz;
    always @(*) begin
        casex({CS,Rst,R_W})
            4'bx1x : for(i=0;i<=SIZE-1;i=i+1) RAM[i]=0;
            4'b101 : Data_i<=RAM[Addr];
            4'b100 : RAM[Addr]<=Data;
            default : Data_i=16'bz;
        endcase
    end
Endmodule
```

代码实现 1K x 16 位的双端口 RAM

模块有六个输入端口：

Data：16 位数据输入/输出。Addr：10 位地址输入。Rst：复位信号。R_W：读写控制信号。CS：片选信号。CLK：时钟信号。

参数说明：

parameter Addr_Width =10：指定地址总线的宽度为 10 位。

parameter Data_Width =16：指定数据总线的宽度为 16 位。

parameter SIZE=2**Addr_Width：计算 RAM 的大小，这里是 2 的 Addr_Width 次方，即 2^10 = 1024，表示 1K 个存储单元。

局部变量声明：

integer i：整数型变量 `i`，用于循环控制。

reg [Data_Width-1:0] Data_i：16 位寄存器 `Data_i`，用于存储 RAM 的输出数据。

reg [Data_Width-1:0] RAM [SIZE-1:0];：16 位寄存器数组 `RAM`，用于存储 1K x 16 RAM 的内容。

赋值语句：

assign Data = (R_W)?Data_i:16'bz：通过三元运算符，根据读写控制信号 `R_W` 来选择输出数据。如果 `R_W` 为真,则输出 `Data_i`;否则输出 16 位的高阻态。

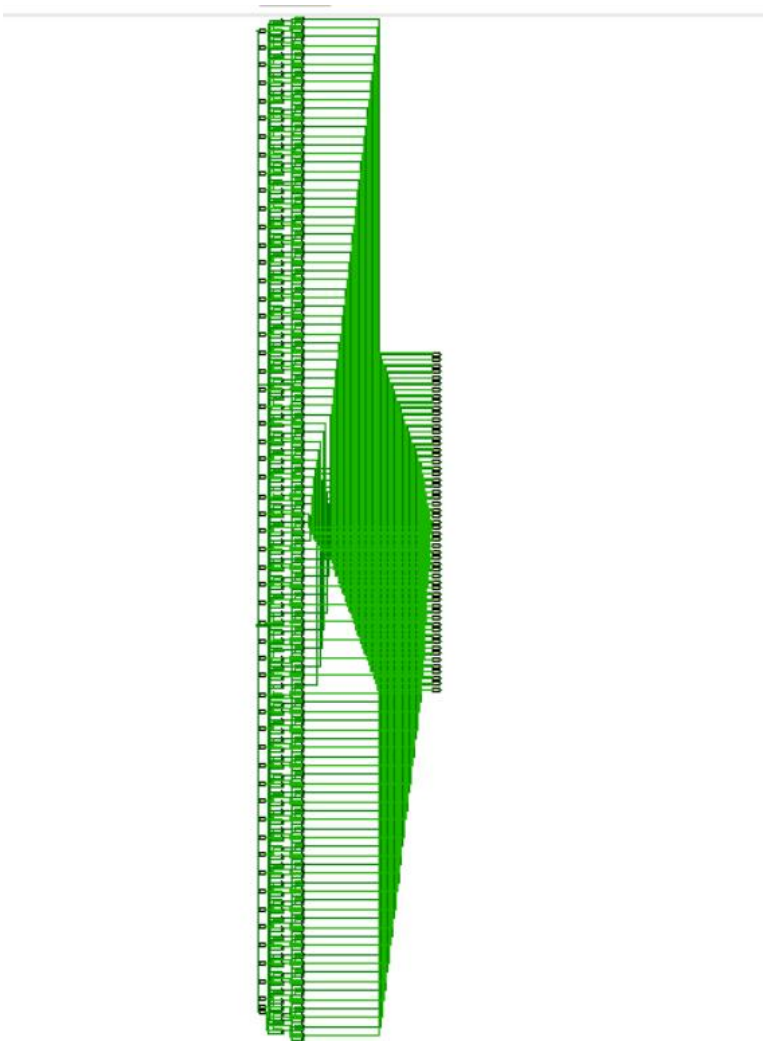
组合逻辑块：

always @(*) begin：这是一个组合逻辑块，表示下面的逻辑与输入端口的变化有关。

casex({CS,Rst,R_W})：这是一个 case 语句，根据 `CS`、`Rst` 和 `R_W` 的值进行不同的操作。

这个模块实现了一个基本的双端口 RAM,可以通过 `Addr` 控制读写的地址，通过 `Data` 进行数据的读写，同时通过其他控制信号进行控制。

RTL 分析



2.4K×32 位的存储器

设计代码

```
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,C
```

```
S_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,C
S_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,C
S_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,C
S_i[3],CLK);
    Endmodule
```

模块名为 RAM_4Kx32_inout。有两个参数：Addr_Width（地址宽度）和 Data_Width（数据宽度）。有五个输入端口：Data（inout）、Addr（地址）、Rst（复位）、R_W（读/写控制）、CS（芯片选择）、和 CLK（时钟）

声明一个 4 位的线 CS_i，用于芯片选择解码。使用一个 2 到 4 线解码器（Decoder24）将地址(Addr)的两个最高位解码为四个芯片选择信号(CS_i)。

实例化了四个 RAM_1Kx16_inout 模块，每个对应一个芯片选择信号：每个实例表示一个 1 千字节（1K）x 16 位的 RAM 块。实例根据芯片选择信号(CS_i)、地址、复位(Rst)、读/写控制(R_W)和时钟(CLK)连接到主输入/输出端口。数据总线(Data)被分为高位和低位，分别连接到每个 RAM 块。

总体而言，实现了一个 4 千字节 x 32 位的 RAM 模块，其中有四个 1 千字节的 RAM 块，通过地址的两个最高位选择。数据总线被分成高位和低位，分别用于每个 RAM 块。

仿真代码

```
module sim_RAM;
    parameter Data_Width = 32;
    parameter Addr_width = 12;
    parameter clk_period = 10;
    reg CLK;
    reg R_W, Rst, CS;
    reg I_O;
    reg[Data_Width-1 : 0] Data_reg;
    wire[Data_Width-1 : 0] Data;
    reg[Addr_width-1 : 0] Addr;
    initial begin
```

```
        CLK = 0;
        forever begin #(clk_period/2) CLK=~CLK; end
    end
    initial begin
        I_O = 1; //默认输入
        R_W=1'b0; //默认写
        Rst=1'b0;
        CS =1'b1; //默认使能
        Addr = 12'b0;
    end
    always begin
        R_W <= 1'b0; // 0#写
        Data_reg <= 32'h0000_0000;
        Addr <= 0;
        #50;
        Data_reg <= 32'h0000_0002;//2#写
        Addr <= 2;
        #50;
        Data_reg <= 32'h0000_0004;//2#写
        Addr <= 4;
        #50;
        Data_reg <= 32'h0000_0006;//2#写
        Addr <= 6;
        #50;
        Data_reg <= 32'h0000_00F6;//246#写
        Addr <= 246;

        #50;
        R_W = 1'b1;
        Addr <= 0; //0#读
        #50; //2#读
        Addr <= 2;
        #50; //4#读
        Addr <= 4;
        #50; //6#读
        Addr <= 6;
        #50; //246#读
        Addr <= 246;
        #50;
        $finish;
     end
     assign Data = (R_W) ? 32'bz : Data_reg;
     RAM_4Kx32_inout ram(Data, Addr,  Rst, R_W, CS, CLK);
Endmodule
```

定义了一个名为 sim_RAM 的模块。包含了三个参数：Data_Width（数据宽度）、Addr_width（地址宽度）、clk_period（时钟周期）。

CLK、R_W、Rst、CS、I_O 分别是时钟、读写控制、复位、芯片选择和输入/输出信号的寄存器。Data_reg 是一个 32 位寄存器，用于存储 RAM 的数据。Data 是一个 32 位线，用于连接 RAM 模块的数据端口。Addr 是一个地址线。使用 forever 循环生成一个半周期为 clk_period/2 的时钟信号。

初始化了一些信号，如默认输入为 1，写操作为默认操作，复位和芯片选择默认为 0，地址默认为 0。

使用 always 块模拟了写操作。每隔 50 个时间单位，进行一次写操作，修改 Data_reg 和 Addr。

接着进行读操作，切换为读模式（R_W = 1'b1），再次修改 Addr 进行读操作。

数据线赋值:

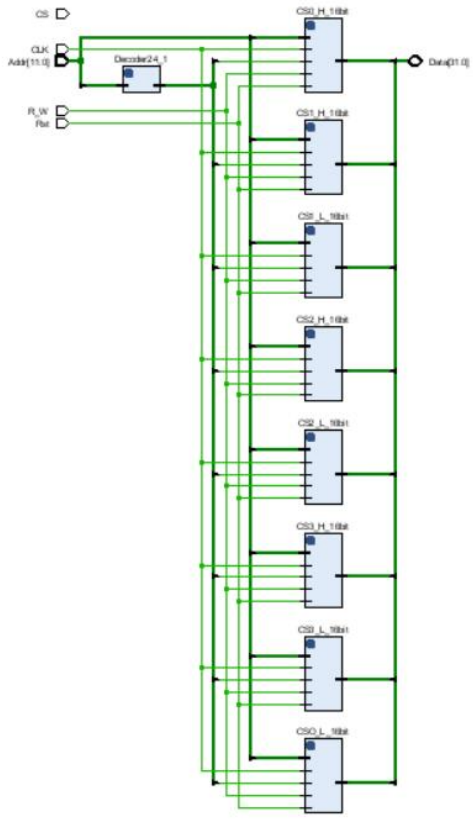
assign Data = (R_W) ? 32'bz : Data_reg; 根据读写控制信号决定将 Data_reg 赋值给 Data 或将 Data 置为高阻态。

RAM 模块实例化:

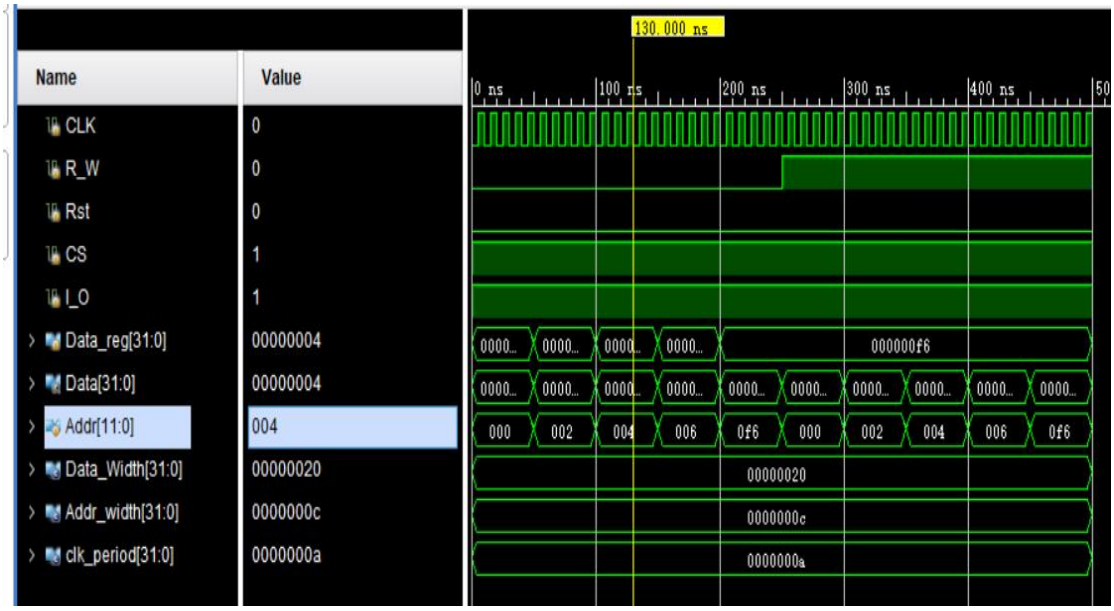
RAM_4Kx32_inout ram(Data, Addr, Rst, R_W, CS, CLK); 实例化了一个名为 ram 的 4K x 32 位的 RAM 模块。

总体来说，仿真代码通过时序模拟测试了 RAM 模块的写入和读取操作。

RTL 分析



波形图分析



分析：首先对 RAM 芯片 0#、 2#、 4#、 6#、 246#（0xf6#） 地址的数据进行写， 分别写(0x0,0x2,0x4,0x6,0xf6)。然后依次读出 0#、 2#、 4#、 6#、 246#(0xf6#)地址的数据，由于 RAM 芯片采用 clk 信号上升沿触发，所以数据线响应相比地址线滞后半个时钟周期。仿真结果显示，依次读出的数据为： 0x0,0x2,0x4,0x6,0xf6,所以说明写入的数据无误，并且能够正确读出

3. 寄存器

设计代码

```
`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
```

实现了一个简单的寄存器文件模块，支持两个读端口和一个写端口。

使用 define 定义了 DATA_WIDTH 宏，表示数据宽度为 32 位。

模块名为 RegFile。

有一个参数 ADDR_SIZE，表示地址的位数，默认值为 5。

输入端口包括时钟信号 CLK，写使能信号 WE3，两个读地址信号 RA1 和 RA2，以及一个写地址信号 WA3 和写数据信号 WD3。

输出端口包括两个读数据信号 RD1 和 RD2。

使用 reg 关键字声明了一个寄存器数组 rf，其大小为 2**ADDR_SIZE 个 32 位寄存器。

使用 always @(posedge CLK) 表示在时钟的上升沿进行操作。

如果写使能信号 WE3 为 1，将写数据 WD3 写入到地址为 WA3 的寄存器中。

使用 assign 分别为读端口 RD1 和 RD2 赋值。

如果读地址 RA1 不为 0，从寄存器数组中读取相应地址的数据；否则，赋值为 0。

同样地，如果读地址 RA2 不为 0，从寄存器数组中读取相应地址的数据；否则，赋值为 0。

仿真代码

```
module RegFile_tb;
  parameter DATA_WIDTH = 32;
  parameter ADDR_SIZE = 5;
  reg CLK;
  reg WE3;
  reg [ADDR_SIZE-1:0] RA1, RA2, WA3;
  reg [DATA_WIDTH-1:0] WD3;
  wire [DATA_WIDTH-1:0] RD1, RD2;
  RegFile #(ADDR_SIZE) uut (
    .CLK(CLK),
    .WE3(WE3),
    .RA1(RA1),
    .RA2(RA2),
    .WA3(WA3),
    .WD3(WD3),
    .RD1(RD1),
    .RD2(RD2)
  );
  initial begin
    CLK = 0;
    forever #5 CLK = ~CLK;
  end
  initial begin
    RA1 = 0; RA2 = 3; WA3 = 3; WD3 = 32'hA5A5A5A5;
    WE3 = 1;
    #10 WE3 = 0;
    RA1 = 0; RA2 = 3; WA3 = 3; WD3 = 0;
    WE3 = 0;
    #10;
    RA1 = 0; RA2 = 0; WA3 = 1; WD3 = 32'h12345678;
    WE3 = 1;
```

```
        #10 WE3 = 0;
        RA1 = 1; RA2 = 0; WA3 = 0; WD3 = 0;
        WE3 = 0;
        #10;
        $stop;
    end
Endmodule
```

分析：

定义了两个参数：DATA_WIDTH 表示数据宽度为 32 位，ADDR_SIZE 表示地址位数为 5 位。

输入和输出信号声明:

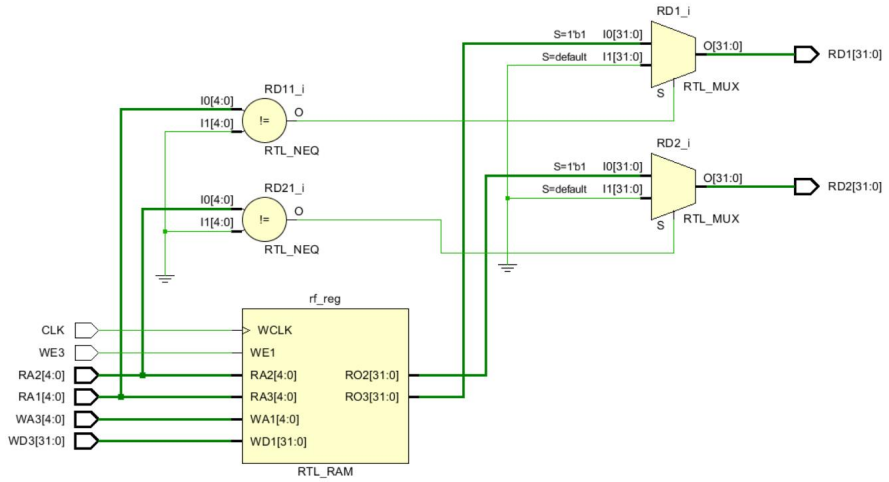
声明了时钟信号 CLK、写使能信号 WE3、读地址信号 RA1 和 RA2、写地址信号 WA3 以及写数据信号 WD3。声明了读数据信号 RD1 和 RD2。

实例化了之前定义的 RegFile 模块，命名为 uut（unit under test）。连接了 RegFile 模块的输入和输出信号。

使用 initial 块生成一个时钟信号 CLK，每 5 个时间单位切换一次。

使用 initial 块设置了一系列的测试序列，包括写入和读取操作。在不同的时间点，改变了读写地址和写入数据，以及写使能信号。使用 # 符号表示延迟时间。

RTL 分析



波形图分析

RA1 在 0~30ns 时读取 0 号地址数据，在 30~40ns 时读取 1 号地址数据。

RA2 在 0~20ns 时读取 3 号地址数据，在 20~40ns 时读取 0 号地址数据。

WA3 在 0~10ns 时写入数据 0xa5a5a5a5 到 3 号地址，在 10~20ns 时写入数据 0x00000000 到 3 号地址，在 20~30ns 时写入数据 0x12345678 到 1 号地址，在 30~40ns 时写入数据 0x00000000 到 0 号地址。

0 号地址初值为 0x00000000，因此 RD1 在前 30ns 内读出的数据是 0x00000000,在 30~40ns 时读出地址 1 中的数据 0x12345678。

3 号地址没有赋初值，所以 RD2 一开始读出错误，后面读出数据 0xa5a5a5a5，在 20~40ns 时，RD2 读出数据 0x00000000。

上述验证了读写的正确性，以及同时读的可行性。

4. 寄存器复位功能的实现

设计代码

```
`define DATA_WIDTH 32

module RegFile
    #(parameter ADDR_SIZE = 5)
    (input CLK,WE3,RESET,
    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];
    integer i;
    always@(posedge CLK)
    begin
```

```
            if(WE3) rf[WA3]<=WD3;
            if(RESET) for (i = 0; i < 2**ADDR_SIZE; i = i + 1)rf[i]
<= 0;
        end
        assign  RD1 = (RA1 != 0) ? rf[RA1] : 0;
        assign  RD2 = (RA2 != 0) ? rf[RA2] : 0;
    endmodule
```

在原本代码的基础上增加了一个 RESET 信号，当 RESET 信号有效时，对输出 rf 的值进行一个 for 循环的重新赋值，从而实现寄存器置零的效果。

仿真代码

```
    module RegFile_tb;
      `define DATA_WIDTH 32
      `define ADDR_SIZE 5
      reg CLK;
      reg WE3;
      reg RESET;
      reg [`ADDR_SIZE-1:0] RA1, RA2, WA3;
      reg [`DATA_WIDTH-1:0] WD3;
      wire [`DATA_WIDTH-1:0] RD1, RD2;
      RegFile #(5) uut (
        .CLK(CLK),
        .WE3(WE3),
        .RESET(RESET),
        .RA1(RA1),
        .RA2(RA2),
        .WA3(WA3),
        .WD3(WD3),
        .RD1(RD1),
        .RD2(RD2)
      );
      initial begin
        CLK = 0;
        forever #5 CLK = ~CLK;
      end
      initial begin
        WE3 = 1;
        RESET = 0;
        RA1 = 1;
        RA2 = 1;
        WA3 = 1;
        WD3 = 8'hFF;
        #100 RESET =1;
        WE3=0;
        #100 $finish;
```

```
  end
endmodule
```

分析：

使用 define 定义了两个宏：DATA_WIDTH 表示数据宽度为 32 位，ADDR_SIZE 表示地址位数为 5 位。

输入和输出信号声明:

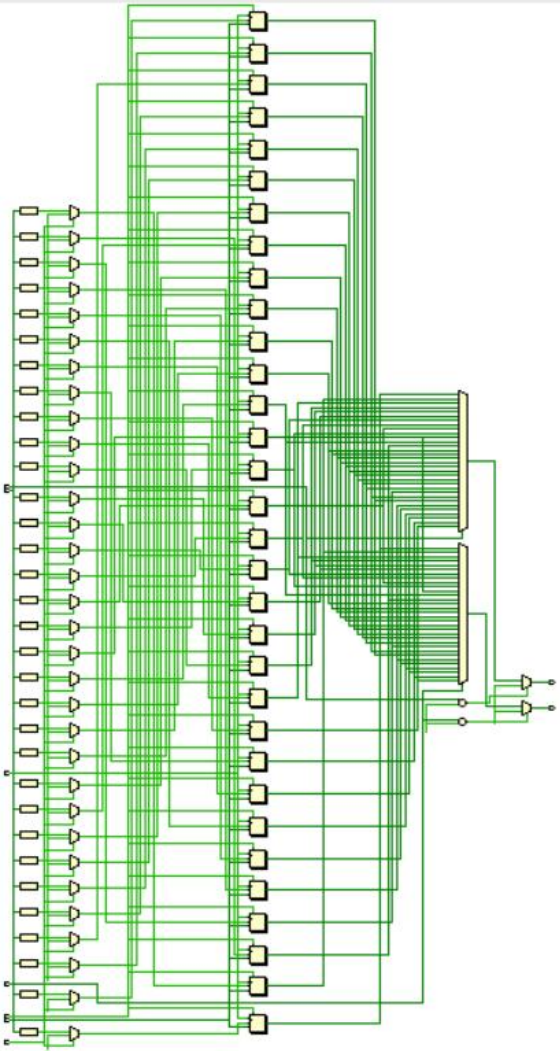
声明了时钟信号 CLK、写使能信号 WE3、复位信号 RESET、读地址信号 RA1 和 RA2、写地址信号 WA3 以及写数据信号 WD3。声明了读数据信号 RD1 和 RD2。

实例化了之前定义的 RegFile 模块，命名为 uut（unit under test）。连接了 RegFile 模块的输入和输出信号。

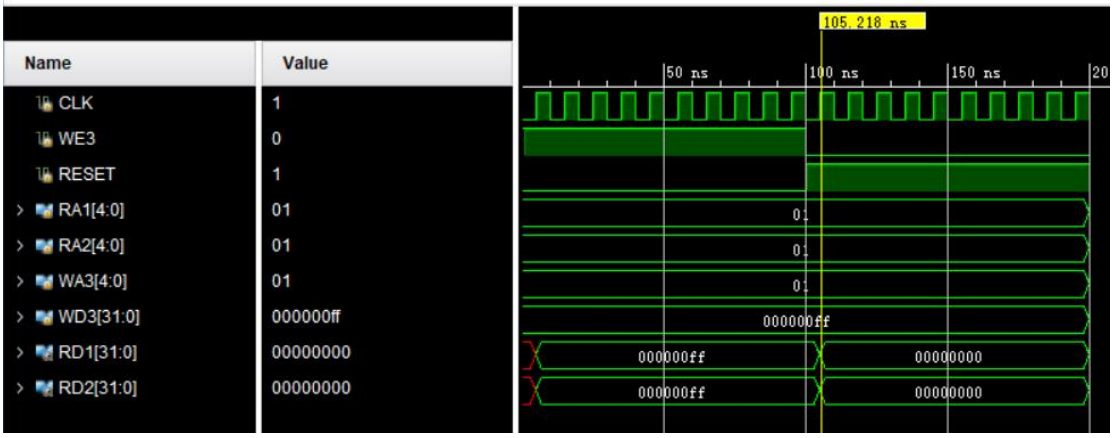
使用 initial 块生成一个时钟信号 CLK，每 5 个时间单位切换一次。

使用 initial 块设置了一系列的测试序列。在初始状态下，写使能信号 WE3 置为 1，复位信号 RESET 置为 0。在 100 个时间单位后，将复位信号 RESET 置为 1，模拟复位操作。在 100 个时间单位后，使用 $finish 终止仿真。

RTL 分析



波形图分析



分析：初始时 RESET 信号为 0，，没有给 1 号地址数据赋值，因此一开始 RA1、RA2 读出数据 RD1、RD2 无效，之后 WA3 向 WD3 写入数据 0x000000ff，RD1、RD2 读出数据也为 0x000000ff，之后设置 RESET 信号为 1，因此 RD1 和 RD2 均为 0x00000000。

五、调试和心得体会

在完成存储器设计与测试以及寄存器组设计与测试的实验过程中，我遇到了一些挑战，但同时也获得了宝贵的经验和收获。以下是我的实验调试与心得体会：

1、在实验中，模块之间的连接和信号传递是关键。我确保正确连接了所有信号，并仔细检查了每个模块的输入输出端口的匹配情况。

2、时钟的正确使用对于存储器和寄存器的设计非常重要。我通过在代码中正确使用 posedge 触发器和时钟延迟，确保了时序逻辑的正确性。

3、 编写有效的测试序列对于验证存储器和寄存器的功能至关重要。我设计了多个测试用例，覆盖了不同的读写操作，以确保模块的全面测试。

将存储器和寄存器组设计为模块，有助于提高代码的可读性和维护性。通过参数化设计，我可以更容易地修改存储器和寄存器的大小，使得代码更具灵活性。在调试过程中，我学到了很多关于 Verilog 语言、硬件描述语言和数字电路设计的知识。通过不断解决问题，我提高了自己的调试技能和理解硬件设计的能力。存储器和寄存器的设计需要考虑时序和同步问题。正确使用时钟信号、触发器和时钟延迟是确保模块正确性的关键。

通过本次实验，我深刻理解了存储器和寄存器的设计原理，并通过实际的调试过程提高了自己的硬件描述语言编程和调试能力。这将对我未来的数字电路设计和硬件工程方向的学习和研究产生积极的影响。