

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/Logisim 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

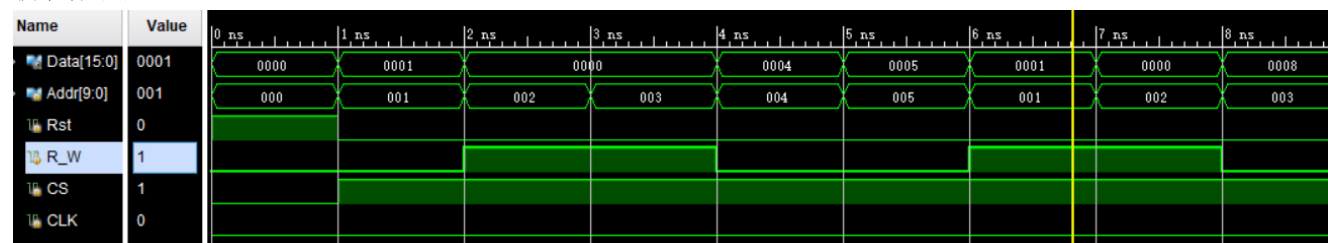
设计文件:

```
module RAM_1Kx16_inout(Data, Addr, Rst, R_W, CS, CLK );
    parameter Addr_Width=10;
    parameter Data_Width=16;
    parameter SIZE=2**Addr_Width;
    inout[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'b1x:for(i=0;i<SIZE;i=i+1) RAM[i]=0;
            4'b101:Data_i<=RAM[Addr];//读数据
            4'b100:RAM[Addr]<=Data;//写数据
            default:Data_i=16'bz;
        endcase
    end
endmodule
```

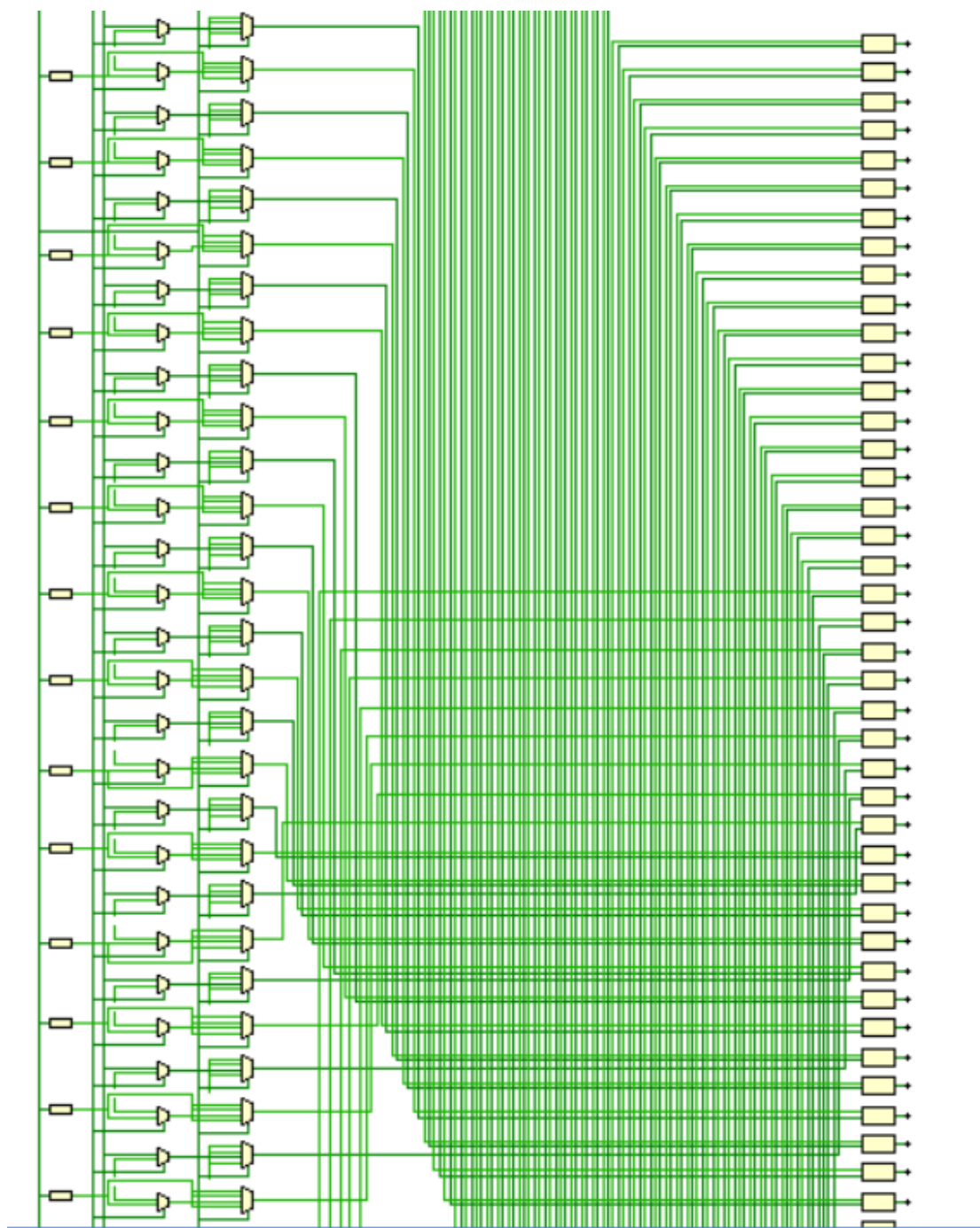
仿真文件：

```
module sim;
    parameter Addr_Width=10;
    parameter Data_Width=16;
    parameter SIZE=2**Addr_Width;
    wire[Data_Width-1:0] Data;
    reg [Addr_Width-1:0] Addr;
    reg Rst, R_W, CS, CLK;
    reg[Data_Width-1:0] Data_i;
    RAM_1Kx16_inout s(Data, Addr, Rst, R_W, CS, CLK );
    initial begin
        {Data_i, Addr, R_W, CS, CLK, Rst}=1;
    end
    assign Data=(R_W==1)?16'bz:Data_i;
    always@(*) fork
        forever #1 Data_i=Data_i+1;
        forever #1 Addr=Addr+1;
        forever #6 Addr=0;
        forever #2 R_W=R_W+1;
        repeat(1) #1 CS=CS+1;
        repeat(1) #1 Rst=Rst+1;
    join
endmodule
```

仿真图形：



电路图：

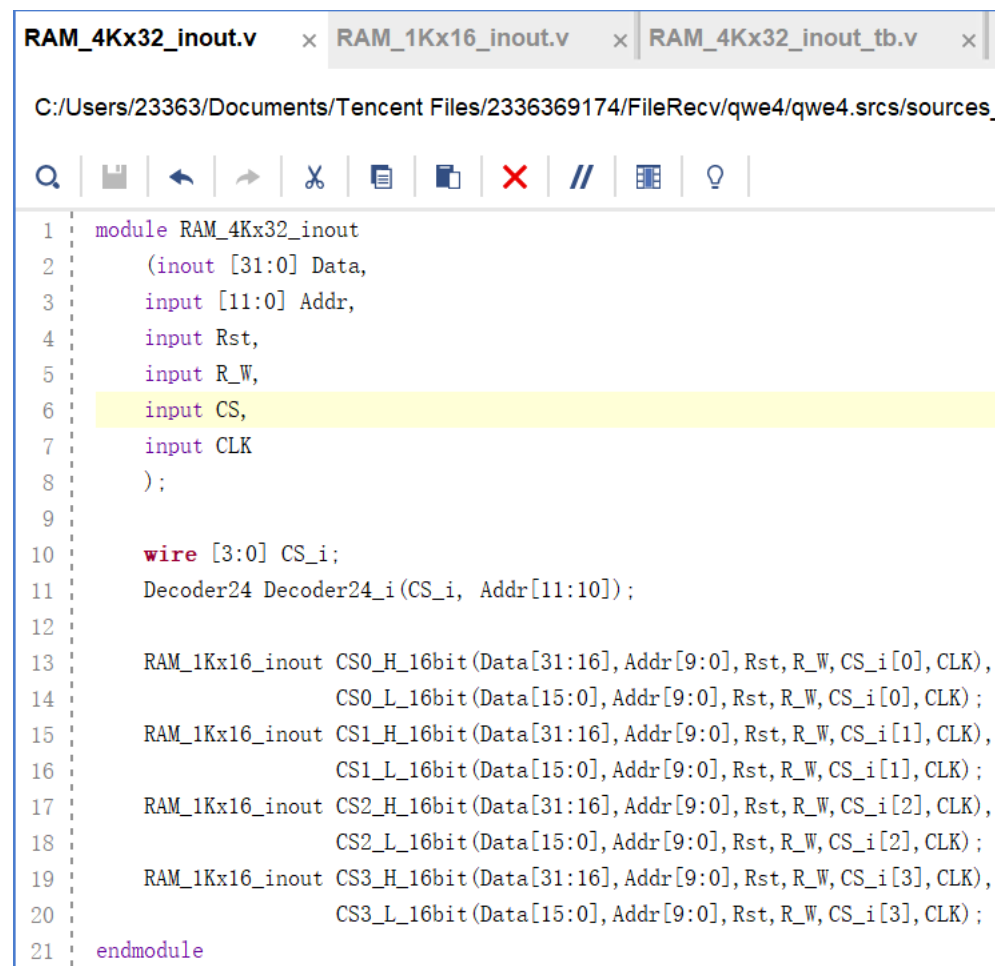


分析：设计存储器时将数据 Data 设计成 inout 类型的变量，并且还设置了一个与 Data 大小相同的 reg 类型变量 Data_i, R_W 信号控制着对存储器的操作，首先是一个 assign 语句，如果 R_W 为 1 表示对存储器进行读操作，则 Data 的值等于 Data_i 的值，如果 R_W 的值为 0 表示对存储器进行写操作，则将 Data 设置为高阻态。然后是一个异步 always 语句，在 always 语句内是一个 case 语句，case 语句内的变量由片选信号 CS，复位信号 Rst，以及控制信号 R_W，当 Rst 信号为 1 的时候，直接让整个 RAM 存储器置 0；若 Rst 信号为 1 且片选信号 CS 有效的时候，此时如果 R_W 信号为 1，表示对存储器 RAM 进行读操作，将 RAM[Addr] 的数据赋值给 Data_i，此时的 Data 的值等于 Data_i 的值，此时等价于将 Data 和 RAM[Addr] 连接了起来，实现了读数据的功能；此时如果 R_W 信号为 0，表示

对存储器 RAM 进行写操作,则将 Data 的值赋给 RAM[Addr], 实现了写数据的功能; 如果 CS, Rst, R_W 不满足上述的情况, 那么就将 Data 置为高阻态。

在仿真图形中, 首先将 Rst 初始化为 1, 将存储器 RAM 全部初始化为 0。然后在 1ns 的时候将 Rst 置 0, CS 片选信号有效, 此时 R_W 信号为 0, Data 的值为 1, Addr 的值为 1, 对 RAM[1] 进行写操作, 将其改为 Data 的值 1。然后在 2ns 到 4ns 的过程中, R_W 信号始终为 1, Addr 的值先是 2 后是 3, 此时是将 RAM[2] 和 RAM[3] 的值先后赋给 Data, 由于 RAM 初始化全为 0, 所以 RAM[2] 和 RAM[3] 都是 0, 在 2ns 到 4ns 的过程中 Data 始终都是 0。在 6ns 的时候, R_W 信号为 1 表示对存储器进行读操作, Addr 为 1, RAM[1] 由于在 1ns 的时候执行了写操作导致其值为 1 而不是初始化的值 0, RAM[1] 的值赋给 Data, 此时 Data 的值为 1, 符合预期结果。

2. 4K*32bit 存储器 设计文件



```
1 module RAM_4Kx32_inout
2     (inout [31:0] Data,
3      input [11:0] Addr,
4      input Rst,
5      input R_W,
6      input CS,
7      input CLK
8  );
9
10     wire [3:0] CS_i;
11     Decoder24 Decoder24_i(CS_i, Addr[11:10]);
12
13     RAM_1Kx16_inout CS0_H_16bit(Data[31:16], Addr[9:0], Rst, R_W, CS_i[0], CLK),
14         CS0_L_16bit(Data[15:0], Addr[9:0], Rst, R_W, CS_i[0], CLK);
15     RAM_1Kx16_inout CS1_H_16bit(Data[31:16], Addr[9:0], Rst, R_W, CS_i[1], CLK),
16         CS1_L_16bit(Data[15:0], Addr[9:0], Rst, R_W, CS_i[1], CLK);
17     RAM_1Kx16_inout CS2_H_16bit(Data[31:16], Addr[9:0], Rst, R_W, CS_i[2], CLK),
18         CS2_L_16bit(Data[15:0], Addr[9:0], Rst, R_W, CS_i[2], CLK);
19     RAM_1Kx16_inout CS3_H_16bit(Data[31:16], Addr[9:0], Rst, R_W, CS_i[3], CLK),
20         CS3_L_16bit(Data[15:0], Addr[9:0], Rst, R_W, CS_i[3], CLK);
21 endmodule
```

设计文件新添了一个 24 译码器, 用来控制使能信号。利用四位使能信号控制 4 个 1K*32bit 存储器。每个 1K*32bit 存储器由 2 个 1K*16bit 存储器分别组成高 16 位和低 16 位。

24 译码器

Decoder24.v x RAM_4Kx32_inout.v x RAM_1Kx16_i

C:/Users/23363/Documents/Tencent Files/2336369174/FileRecv

```
1 module Decoder24(CS_i, Addr);
2     parameter Addr_Width =12;
3     parameter Data_Width=32;
4     input  [11:0] Addr;
5     output [3:0] CS_i;
6
7     assign CS_i = (Addr == 2'b00) ? 4'b0001 :
8                   (Addr == 2'b01) ? 4'b0010 :
9                   (Addr == 2'b10) ? 4'b0100 :
10                  4'b1000 ;
11 endmodule
```

仿真文件 part1:

RAM_4Kx32_inout_tb.v x Decoder24.v x RAM

C:/Users/23363/Documents/Tencent Files/2336369174/F

```
1 module RAM_4Kx32_inout_tb;
2     wire [31:0] Data;
3     reg  [11:0] Addr;
4     reg  Rst, R_W, CS, CLK;
5
6     RAM_4Kx32_inout uut (
7         .Data(Data),
8         .Addr(Addr),
9         .Rst(Rst),
10        .R_W(R_W),
11        .CS(CS),
12        .CLK(CLK)
13    );
14    always begin
15        #5 CLK=~CLK ;
16    end
```

实例化了一些变量并设置了时钟信号

仿真文件 part2:

```

17  ⚙      initial begin
18  :      {CS, Rst, R_W}=4' b111;
19  :      CLK = 0;
20  :      ○    //0
21  :      ○    Addr[11:10]=2' b00;
22  :      {CS, Rst, R_W}=4' b100;
23  :      ○    Addr[9:0]=10;
24  :      ○    force Data=32' h11111111;#10
25  :      ○    release Data;
26  :      ○    //1
27  :      ○    Addr[11:10]=2' b01;
28  :      {CS, Rst, R_W}=4' b100;
29  :      ○    Addr[9:0]=11;
30  :      ○    force Data=32' h11111112;#10
31  :      ○    release Data;
32  :      ○    //2
33  :      ○    Addr[11:10]=2' b10;
34  :      {CS, Rst, R_W}=4' b100;
35  :      ○    Addr[9:0]=12;
36  :      ○    force Data=32' h11111113;#10
37  :      ○    release Data;
38  :      ○    //3
39  :      ○    Addr[11:10]=2' b11;
40  :      {CS, Rst, R_W}=4' b100;
41  :      ○    Addr[9:0]=13;
42  :      ○    force Data=32' h11111114;#10
43  :      ○    release Data;

```

存储器清零并初始化一些变量

依次向地址为

00 0000001010 处写入 32 'h11111111

01 0000001011 处写入 32 'h11111112

10 0000001100 处写入 32 'h11111113

11 0000001101 处写入 32 'h11111114

仿真文件 part3:

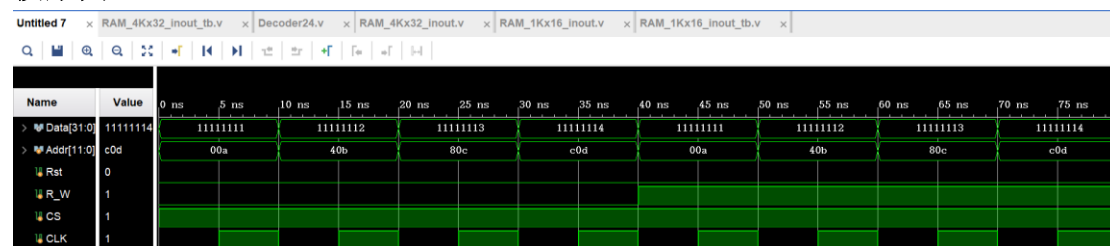
```

44 | ○ | //0
45 | ○ | Addr[11:10]=2' b00;
46 |   | {CS, Rst, R_W}=4' b101;
47 |   | Addr[9:0]=10;
48 |   | #10
49 | ○ | //1
50 | ○ | Addr[11:10]=2' b01;
51 | ○ | {CS, Rst, R_W}=4' b101;
52 | ○ | Addr[9:0]=11;
53 |   | #10
54 | ○ | //2
55 | ○ | Addr[11:10]=2' b10;
56 | ○ | {CS, Rst, R_W}=4' b101;
57 | ○ | Addr[9:0]=12;
58 |   | #10
59 | ○ | //3
60 | ○ | Addr[11:10]=2' b11;
61 | ○ | {CS, Rst, R_W}=4' b101;
62 | ○ | Addr[9:0]=13;
63 |   | #10
64 | ○ | $finish;
65 | ○ | end
66 | ○ | endmodule

```

依次读出地址为
00 0000001010、
01 0000001011、
10 0000001100、
11 0000001101
处的数据

波形图：



如图，一开始 R_W=0 时写数据，R_W=1 时读数据。当地址 ADDR=00 0000001010 时，R_W=1 和 R_W=0 时 Data 的值相同，说明读取的数据和写入的数据一致都为 32' h11111111，表示写入和读取成功。

当地址 ADDR=01 0000001011、10 0000001100、11 0000001101 时同理。

以上说明 4K*32bit 存储器实现了读取和写入的功能，且由设计文件可知其大小确实为 4K*32bit。

3. 指令存储器：

设计文件:

```
1 | define DATA_WIDTH 32
2 | module IMem(
3 |     input [5:0] A,
4 |     output [`DATA_WIDTH-1:0] RD);
5 |     parameter IMEM_SIZE=64;
6 |     reg [`DATA_WIDTH-1:0] RAM[IMEM_SIZE-1:0];
7 |
8 |     initial
9 |         $readmemh("C:\\Users\\23363\\Desktop\\test.dat", RAM);
10 |     assign RD=RAM[A];
11 | endmodule
```

位宽 16 (DATA_WIDTH)

地址位 6 (A)

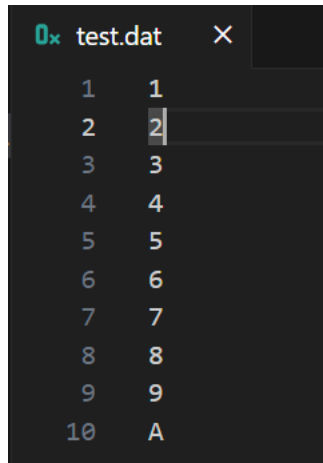
个数 64 (IMEM_SIZE)

仿真文件:

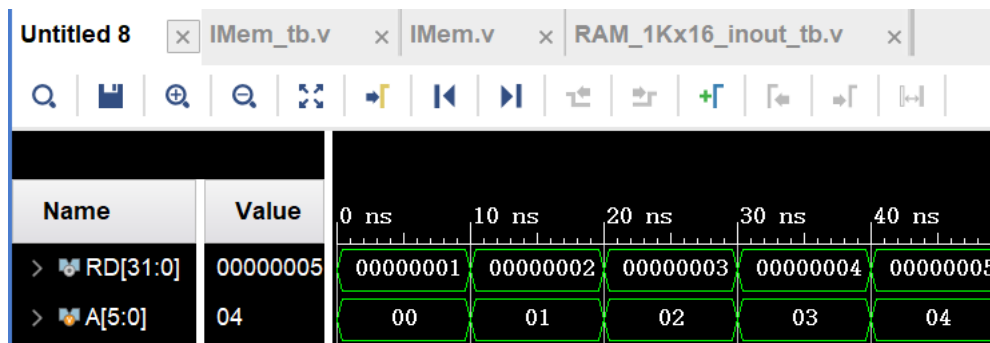
```
1 | module IMem_tb;
2 |     wire [`DATA_WIDTH-1:0] RD;
3 |     reg [5:0] A;
4 |     IMem uut(
5 |         .RD(RD),
6 |         .A(A)
7 |     );
8 |     initial begin
9 |         A=0;#10;
10 |        A=1;#10;
11 |        A=2;#10;
12 |        A=3;#10;
13 |        A=4;#10;
14 |        $finish;
15 |     end
16 | endmodule
```

依次读出地址为 0, 1, 2, 3, 4 处的地址

数据文件:

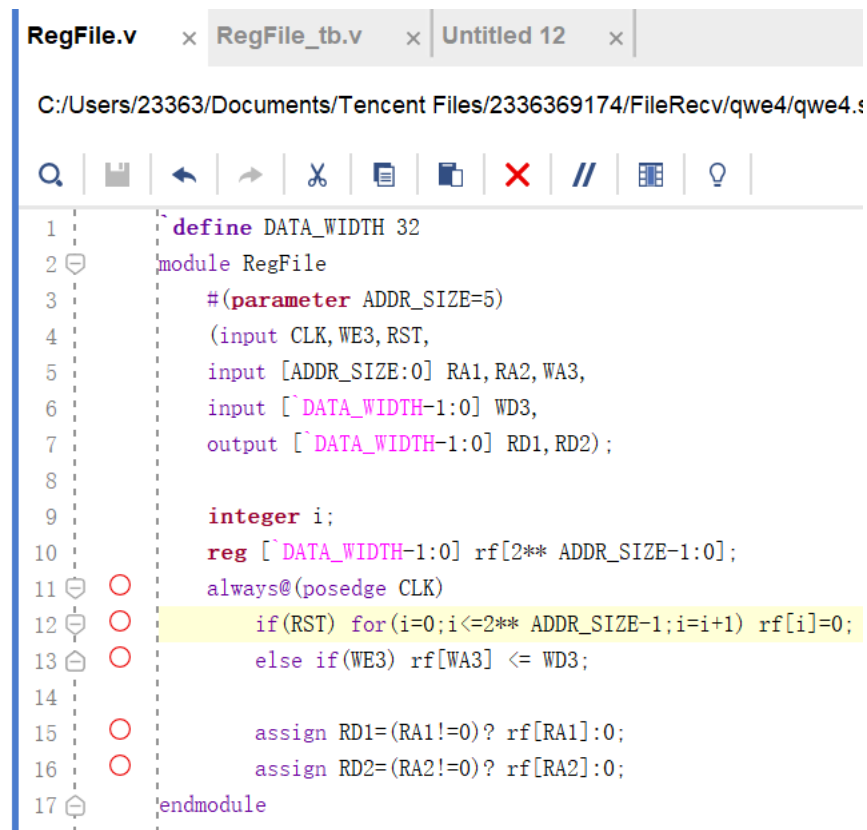


仿真文件：



如图，在地址为 0，1，2，3，4 处 RD 分别读出了数据文件 1~5 行的内容，说明指令寄存器实现了读取文件内容并输出寄存器内容的功能

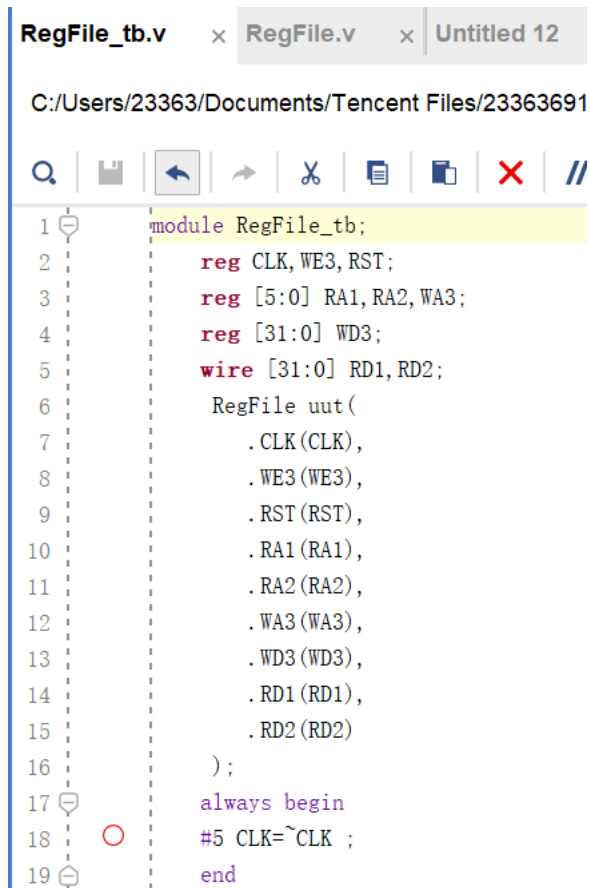
4. 寄存器文件设计：



```
1      define DATA_WIDTH 32
2      module RegFile
3          #(parameter ADDR_SIZE=5)
4          (input CLK, WE3, RST,
5           input [ADDR_SIZE:0] RA1, RA2, WA3,
6           input [`DATA_WIDTH-1:0] WD3,
7           output [`DATA_WIDTH-1:0] RD1, RD2);
8
9          integer i;
10         reg [`DATA_WIDTH-1:0] rf[2** ADDR_SIZE-1:0];
11         always@(posedge CLK)
12             if(RST) for(i=0; i<=2** ADDR_SIZE-1; i=i+1) rf[i]=0;
13             else if(WE3) rf[WA3] <= WD3;
14
15         assign RD1=(RA1!=0)? rf[RA1]:0;
16         assign RD2=(RA2!=0)? rf[RA2]:0;
17     endmodule
```

由一个写端口和两个读端口构成。地址为 6 位（ADDR_SIZE），数据位为 32 位（DATA_WIDTH）。CLK 为时钟信号，WE3 为写使能，RST 为重置寄存器信号（RST=1 时重置）RA1, RA2 为读取地址，WA3 为写入地址，WD3 为写地址，RD1, RD2 为读地址，rf 为一个 64 个寄存器的数组。整体为触发时钟下降信号时触发重置操作和写入读取操作。

仿真文件 part1:



```
1 module RegFile_tb;
2     reg CLK, WE3, RST;
3     reg [5:0] RA1, RA2, WA3;
4     reg [31:0] WD3;
5     wire [31:0] RD1, RD2;
6     RegFile uut(
7         .CLK(CLK),
8         .WE3(WE3),
9         .RST(RST),
10        .RA1(RA1),
11        .RA2(RA2),
12        .WA3(WA3),
13        .WD3(WD3),
14        .RD1(RD1),
15        .RD2(RD2)
16    );
17    always begin
18        #5 CLK=~CLK ;
19    end
```

实例化了一些变量并设置了时钟信号。

仿真文件 part2:

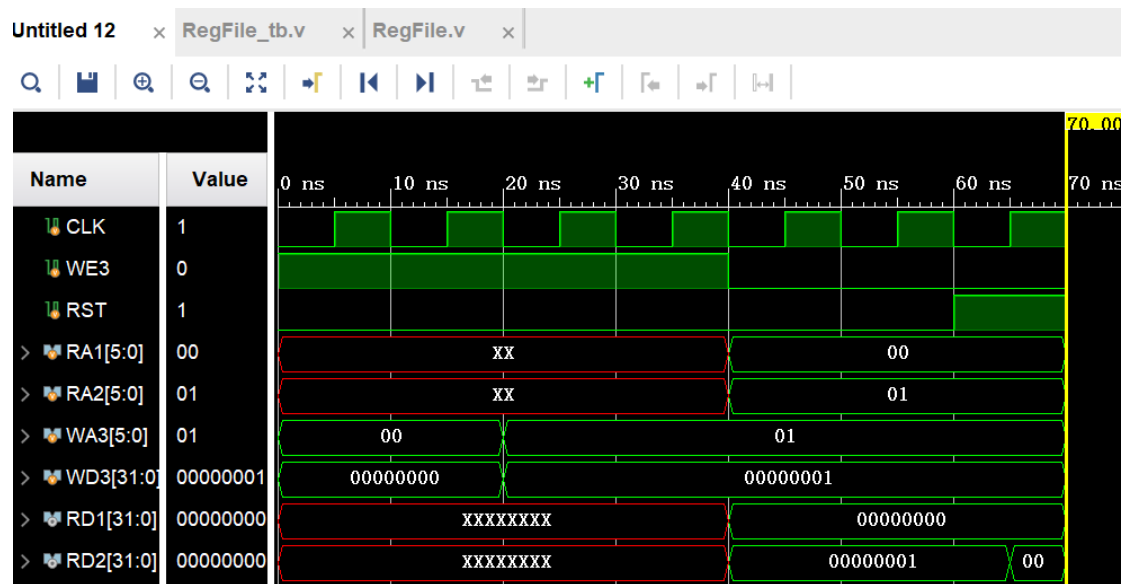
```

20  ⬅ initial begin
21  ○   {CLK, WE3, RST}=3' b001;
22  ○   RST=0;
23
24  ○   WE3=1;
25  ○   WA3=6' b000000;
26  ○   WD3=32' h0;
27  ○   #20;
28  ○   WE3=0;
29
30  ○   WE3=1;
31  ○   WA3=6' b000001;
32  ○   WD3=32' h1;
33  ○   #20;
34  ○   WE3=0;
35
36  ○   WE3=0;
37  ○   RA1=6' b000000;
38  ○   RA2=6' B000001;
39  ○   #20;
40  ○   WE3=0;
41
42  ○   RST=1;
43  ○   #10
44  ○ ➡ $finish;
45  ⬅ end
46  ⬅ endmodule

```

首先重置了寄存器并初始化了一些变量。之后依次
向地址为 6 'b000000 和 6' b000001 处分别写了 32 'h0 和 32' h1
最后重置了寄存器

波形图：



如图 WE3=1 时写入，任何时候都在读取。RST=1 时重置寄存器。一开始未设置读地址，因此读取的数据为 32 'hxxxxxxxxx。

WE3=1 时，0~20ns 向地址为 6' b0 处写入 32 'h0，20~40ns 向地址为 6 'b1 处写入 32 'h1

WE3=0 时，40ns~60ns 读取地址为 RA1 和 RA2 的数据，可以看出 RD1=32' h0，RD2=32' h1，与之前写入的数据一致。

60~70ns，重置了寄存器，因此 RD1, RD2 的数据变为初始值 32 'h0. 由于 RD1 的数据本来就是 32' h0，因此未发生改变。

五、调试和心得体会

本次写了两个存储器和两个寄存器，加深了我对存储器和寄存器的认识提高了我的代码编写能力和对存储器和寄存器的理解。