计算机系统结构试验 Lab04

518030910022 杨涵章

目录

1 实验概述：

1.1 实验名称：

简单的类 MIPS 单周期处理器实现–寄存器、存储器与有符号扩展

1.2 实验目的：

1） 理解 CPU 的寄存器、存储器、有符号扩展
2） Register 的实现
3） Data Memory 的实现
4） 有符号扩展的实现
5） 使用功能仿真

1.3 实验步骤：

1） 按照实验手册指示分别创建寄存器、存储器、符号扩展模块
2） 分模块对三个部分进行代码实现
3） 按照手册上的波形图进行波形测试，以得到和手册相同的波形图

2 实验描述：

2.1 寄存器模块：

2.1.1 模块描述

寄存器模块是有限存储容量的高速存储部件，可以用其暂存指令、数据和地址，寄存器是 MIPS 指令的主要操作对象。从教科书可知，MIPS 处理器一共有 32 个 32 位通用寄存器。在这里仅实现通用寄存器的功能即可。

2.1.2 模块代码

寄存器的读取是组合逻辑，写入是时序逻辑，两者需要进行同步，不然就会发生错误。按照实验手册的指示，由于不确定 WriteReg, WriteData, RegWrite 信号的先后次序，可采用时钟的下降沿作为写操作的同步信号，以防止发生错误。

以下为寄存器模块的具体实现代码：

```
23  module Registers(
24      input clk,
25      input [25:21] readReg1,
26      input [20:16] readReg2,
27      input [4:0] writeReg,
28      input [31:0] writeData,
29      input regwrute,
30      output reg [31:0] readData1,
31      output reg [31:0] readData2
32      );
33
34      reg [31:0] regFile[31:0];
35
36      always @ (readReg1 or readReg2 or writeReg)
37      begin
38          if(readReg1)
39              readData1=regFile[readReg1];
40          else
41              readData1=0;
42          if(readReg2)
43              readData2=regFile[readReg2];
44          else
45              readData2=0;
46      end
47    always @ (negedge clk)
48      begin
49          if (regwrute)
50           regFile[writeReg] = writeData;
51           end
52  endmodule
```

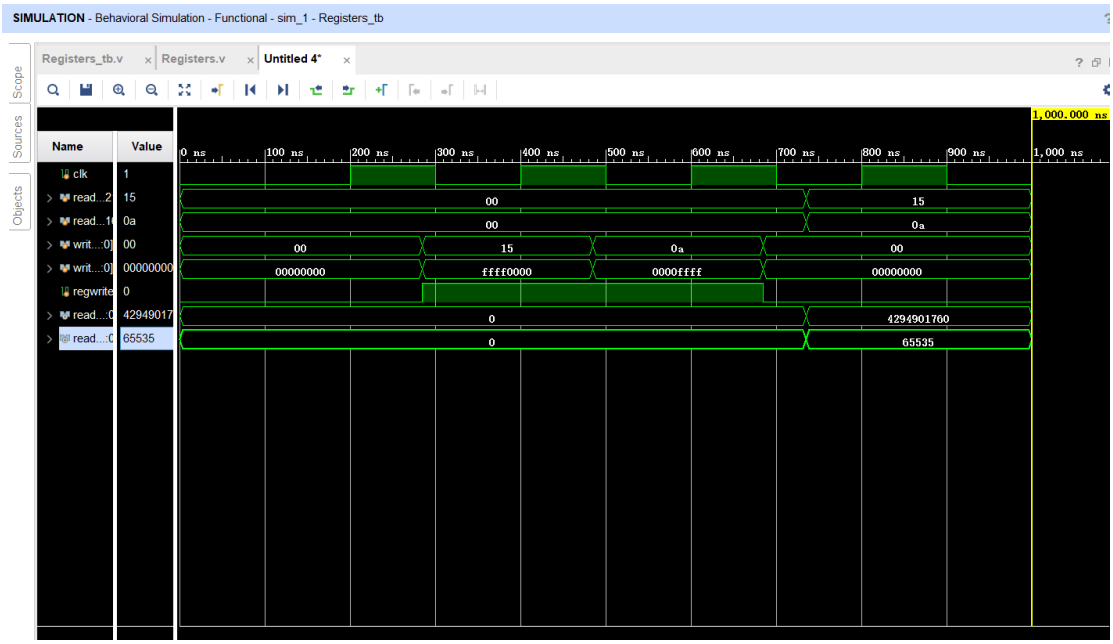
2.1.3 仿真代码

这里直接使用指导书中的激励信号，在不同的时刻对寄存器进行读取或写入操作。

```
47      always #100
48      clk=!clk;
49
50      initial begin
51
52      clk = 0;
53      readReg1 = 0;
54      readReg2 = 0;
55      regwrite = 0;
56      writeReg = 0;
57      writeData = 0;
58      #100
59      clk=0;
60      #185; // 285 ns
61      regwrite = 1'b1;
62      writeReg = 5'b10101;
63      writeData = 32'b11111111111111110000000000000000;
64      #200; // 485 ns
65      writeReg = 5'b01010;
66      writeData = 32'b00000000000000001111111111111111;
67      #200 // 685 ns
68      regwrite = 1'b0;
69      writeReg = 5'b00000;
70      writeData = 32'b00000000000000000000000000000000;
71      #50; // 735 ns
72      readReg1 = 5'b10101;
73      readReg2 = 5'b01010;
74      end
75  endmodule
```

2.1.4 仿真结果

得到的波形和指导书中的一致，对比代码，也符合预期的波形，实验基本成功。



2.2 数据存储器模块

2.2.1 模块描述

该模块即处理器结构中的存储器结构，可以随时任意读写指定有效地址中的数据。

2.2.2 模块代码

实现的方法和寄存器类似，同样需要构造 memFile，这里实现的是 64x32 的数据存储器。在实际构造中，一般是按字节寻址，这里采用和实验手册一样的结构。同样这里需要注意时序逻辑和组合逻辑的信号同步，在时钟的下降沿进行操作。

```
module dataMemory(
    input clk,
    input [31:0] address,
    input [31:0] writedata,
    input memwrite,
    input memread,
    output reg [31:0] readdata
    );

    reg [31:0] memFile[0:63];

    always @ (address)
        begin
            if(memread&&!memwrite)
                readdata=memFile[address];
            else
                readdata=0;
            end

    always @ (negedge clk)
    begin
        if(memwrite)
            memFile[address]=writedata;
    end

endmodule
```

2.2.3 仿真代码

同样这里采用指导书中所给的代码，分别测试了数据存储器在读取、写入内存的激励信号下存储器的行为，得到的波形和实验手册一样，成功完成了实验。

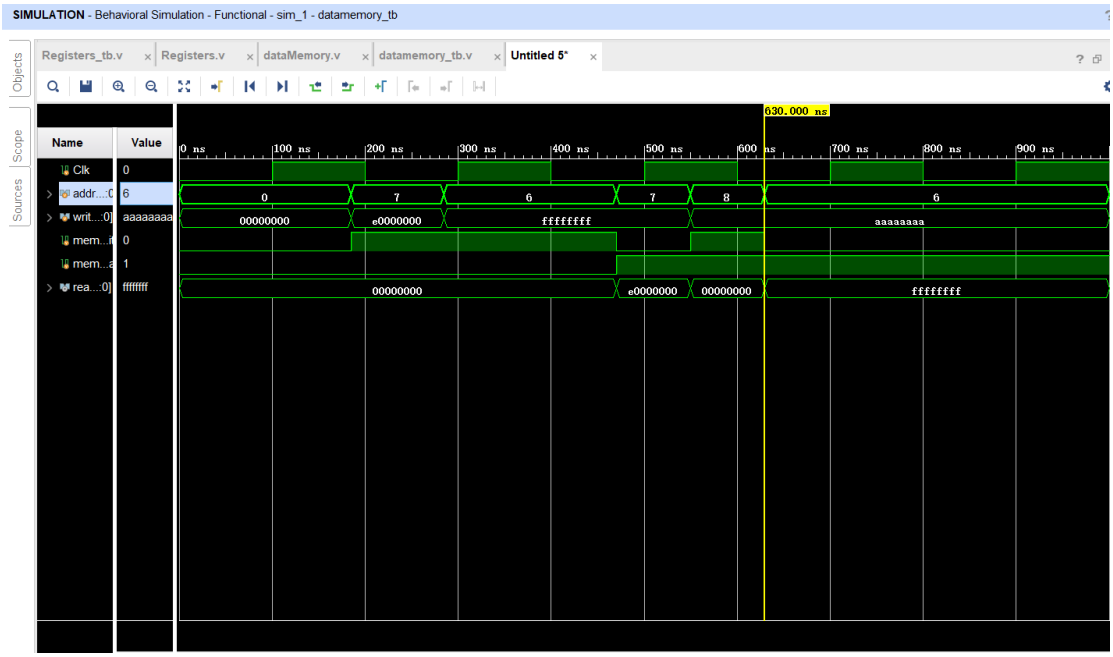
```
module datamemory_tb(

    );
    reg Clk;
    reg [31:0] address;
    reg [31:0] writeData;
    reg memWrite;
    reg memRead;
    wire [31:0] readdata;

    dataMemory mem(
    .clk(Clk),
    .address(address),
    .writedata(writeData),
    .memwrite(memWrite),
    .memread(memRead),
    .readdata(readdata));
    always #100
        Clk = !Clk;
    initial begin // Initialization
    Clk = 0;
    address = 0;
    writeData = 0;
    memWrite = 0;
    memRead = 0;
    #185; // 185 ns
    memWrite = 1'b1;
    address = 7;
    writeData = 32'he0000000;
    #100; // 285 ns
    address = 6;
     writeData = 32'hffffffff;
     #185;
```

```
56          // 470 ns
57          memRead = 1'b1;
58          memWrite = 1'b0;
59          address = 7;
60          #80; // 550 ns
61          memWrite = 1'b1;
62           address = 8;
63           writeData = 32'haaaaaaaa;
64
65           #80;//630ns
66           address=6;
67           memWrite=1'b0;
68           memRead=1'b1;
69           end
70      endmodule
```

2.2.4 仿真结果



2.3 符号扩展模块：

2.3.1 模块描述

该模块能将 16 位有符号数扩展为 32 位有符号数。

2.3.2 模块代码

该实验需要了解计算机中的数字储存方式，但是手册中给出了提示，只需要在前 16 位中补足符号即可，所以我采用或操作，判断数字的 15 符号位，并相应的对数字取或操作，得到对应的 32 位有符号数。

```
module signext(
    input [15:0] inst,
    output [31:0] data
    );

    assign data=(inst[15])?(inst|32'hffff0000):(inst|32'h00000000);

endmodule
```

2.3.3 仿真代码

按照实验手册分别对模块输入整数、负数和 0，这里由于是高级语言的操作，所以不需要转换成补码，直接赋值对应的十进制数即可。

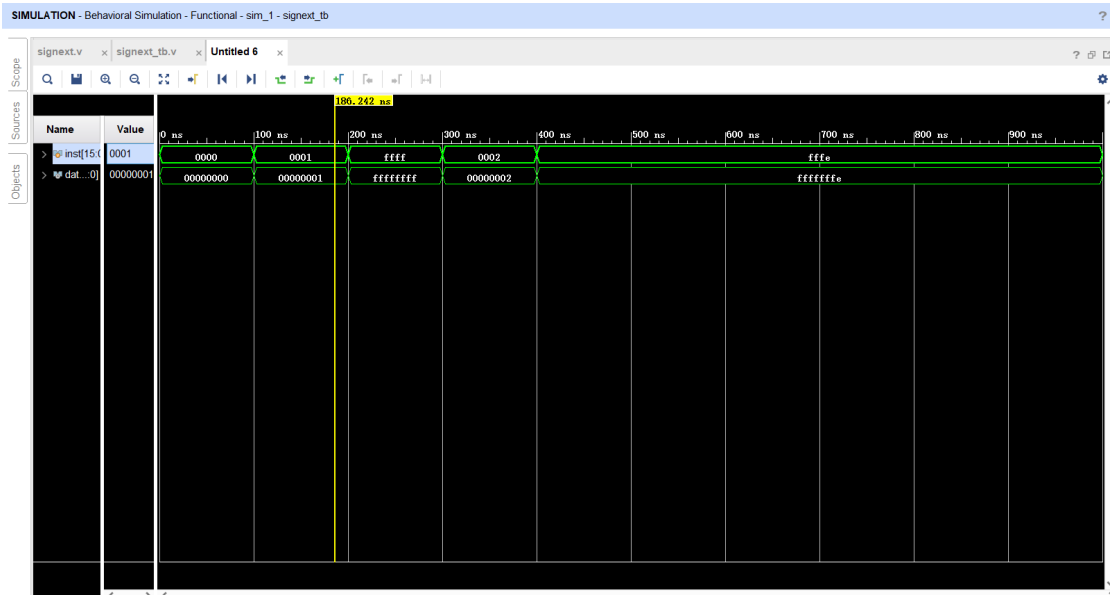
```
module signext_tb(

    );

    reg[15:0] inst;
    wire[31:0] data;
    signext sg(
    .inst(inst),
    .data(data));

    initial begin
        inst=0;
        #100
        inst=1;
        #100
        inst=-1;
        #100
        inst=2;
        #100
        inst=-2;
        #100;
    end
endmodule
```

2.3.4 仿真结果



3 总结与反思：

这次实验涉及组合逻辑和时序逻辑的统一，需要考虑时钟同步的问题。一开始的实现过程中，我没有注意到一点，实现过程中遇到了较大的困难，后来我经过搜索，整体的设计有一定的参考搜集到的资料，实现了两者的统一。但其实实验手册给出的代码架构也有了相应的提示，当初阅读的时候没能发现，浪费了较多时间。

这次实验中，符号扩展较为简单，但实现方式很多。我才用的是进行或运算，同样也可以使用实验 3 运用到的拼接操作（{符号位,Number}），进行符号位的判断之后，拼接对应的扩展位。数据存储器和寄存器类似，都是采用系统的 block memory 实现，同样寄存器的读写和数据的读写类似，在实现一个模块之后，另一个模块也能在较短的时间之内实现。