lab04 report

郑航 520021911347

1 实验目的

- 理解寄存器、 数据存储器、 有符号扩展单元的 IO 定义
- Registers 的设计实现
- Data Memory 的设计实现
- 有符号扩展部件的实现
- 对功能模块进行仿真

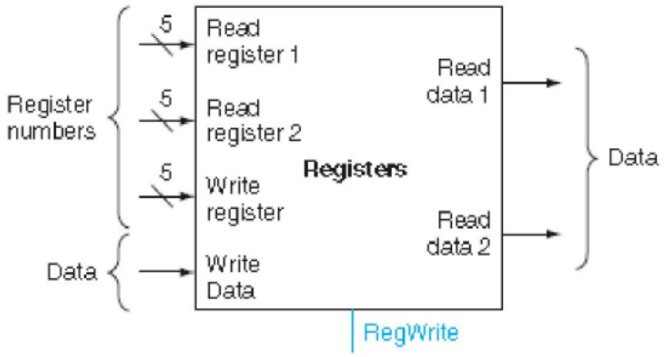
2 实验原理

2.1 Registers原理分析

- 功能：寄存器是指令操作的主要对象， MIPS 中一共有 32 个 32 位的寄存器， 用作数据的缓存，其需要同时支持双通道数据读取、数据写入的功能
- 输入：

```
input [25 : 21] readReg1,
input [20 : 16] readReg2,
input [4 : 0] writeReg,
input [31 : 0] writeData,
input regWrite,
input clk,
```

  - readReg1、readReg2和writeReg是目标读/写寄存器的编号，由于总共有32个寄存器，因此需要5位的信号来标识目标寄存器
  - writeData是32位的输入，表示要写入的数据
  - regWrite是寄存器写的enable信号，高电平表示寄存器可写入
  - clk是时钟信号，在该模块中主要用于同步寄存器写的过程。写数据的操作仅在时钟下降沿进行。因为时钟的下降沿处于一个周期的中部，这时候开始写数据可以保证下一周期开始阶段读取的数据是正确的
- 输出：两个读取的32位数据readData1和readData2，即readReg1和readReg2对应编号的寄存器中的数据
- 模块端口示意图：



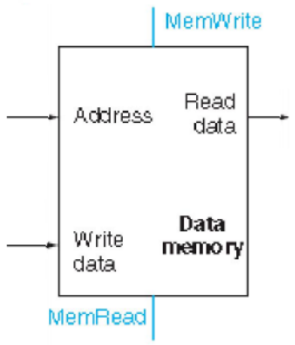
2.2 Data Memory原理分析

- 功能：进行大量数据的存储，包括运行完成的数据或者初始化的数据，其编写与Registers模块类似，也需要考虑clk信号
- 输入：

```
input clk,
input [31 : 0] address,
input [31 : 0] writeData,
input memWrite,
input memRead,
```

  - address是目标读/写位置的编号
  - writeData是32位的输入，表示要写入的数据
  - memWrite和memRead是存储器写/读的enable信号，高电平表示存储器可写入/读出

    - clk是时钟信号，在该模块中主要用于同步memory写的过程。写数据的操作仅在时钟下降沿进行。因为时钟的下降沿处于一个周期的中部，这时候开始写数据可以保证下一周期开始阶段读取的数据是正确的
- 输出：32位的readData，即存储器读时，address对应位置的数据
- 模块端口示意图：



2.3 Sign Extension原理分析

- 功能：将一个输入的16位有符号立即数拓展为 32 位有符号立即数，因此只需要将这个 16 位有符号数的符号位填充在 32 位有符号立即数的高 16 位，再将低16 位复制到 32 位有符号立即数的低 16 位即可
- 输入：16位的inst，即待符号拓展的数据
- 输出：32位的data，即符号拓展后的结果

3 实验过程

3.1 Registers

寄存器一直进行读取操作，而只在时钟下降沿进行写操作，因此我们把读和写分别放在两个always块中

- 在读的always块中，我们关注readReg1或readReg2的信号变化，一旦发生变化就将readReg1和readReg2对应的寄存器中的数据读入到readData1和readData2中
- 在写的always块中，我们关注clk下降沿的到来，每次clk下降沿到达就根据RegWrite 信号，判断是否进行写入操作，是的话就将数据写入到寄存器组里writeReg对应编号的寄存器中

完整代码如下：

```
module Registers(
    input [25 : 21] readReg1,
    input [20 : 16] readReg2,
    input [4 : 0] writeReg,
    input [31 : 0] writeData,
    input regWrite,
    input clk,
    output reg [31 : 0] readData1,
    output reg [31 : 0] readData2
    );

    reg [31 : 0] regFile [31 : 0];

    always @ (readReg1 or readReg2)
```

```
    begin
        readData1 = regFile[readReg1];
        readData2 = regFile[readReg2];
    end

    always @ (negedge clk)
    begin
        if (regWrite)
            regFile[writeReg] = writeData;
    end

    initial begin
        regFile[0] = 0;
    end
endmodule
```

3.2 Data Memory

与3.1Registers功能类似，也是分为读和写两个部分，也分别放在两个always块中

- 在读的always块中，我们关注memRead或address的信号变化，与Registers中不同的是在这里读的部分中也需要增加一个对读数据的使能信号memRead的判断：memRead为1时可读，将address对应的区域中的数据读入到readData中
- 在写的always块中，我们关注clk下降沿的到来，每次clk下降沿到达就根据memWrite信号，判断是否进行写入操作，是的话就将writeData数据写入到存储器里address对应编号的区域中
- 注意到，我们设计的存储器大小只有1024个地址，所以理论上只需要10位的address即可完成对所有数据的寻址，但为了贴合现代计算机设计等实际情况，我们还是设置了32位的address。由此而来的一个问题是，address表示的地址可能大于1023，可能造成错误，因此需要在实现中增加对address大小的判断，若超出1023，则读出时，将读出数据置为0；写入时，不进行写入

完整代码如下：

```
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 : 1023];


    always @ (memRead or address)
    begin
        begin
            // check if the address is valid
            if(address <= 1023)
                if (memRead&&!memWrite)
                    readData = memFile[address];
            else
                readData = 0;
        end
    end

```

```
    always @ (negedge clk)
    begin
        if (memWrite && address <= 1023)
            memFile[address] = writeData;
    end
endmodule
```

3.3 Sign Extension

将指令中的 16 位有符号立即数拓展为 32 位有符号立即数，只需要将这个 16 位有符号数的符号位填充在 32 位有符号立即数的高 16 位，再将低 16 位复制到 32 位有符号立即数的低 16 位即可

我们使用上一个lab中学到的Verilog中的拼接操作，将16位有符号数的符号位复制 16 份，与原数进行拼接即可得到符号扩展后的结果

完整代码如下：

```
module signext(
    input [15 : 0] inst,
    output [31 : 0] data
    );
    assign data = { {16 {inst[15]}}, inst[15 : 0] };
endmodule
```

4 实验结果

4.1 Registers仿真结果

使用 Verilog 编写激励文件，采用软件仿真的形式对主控制器 (Registers) 模块进行测试

4.1.1 激励文件编写

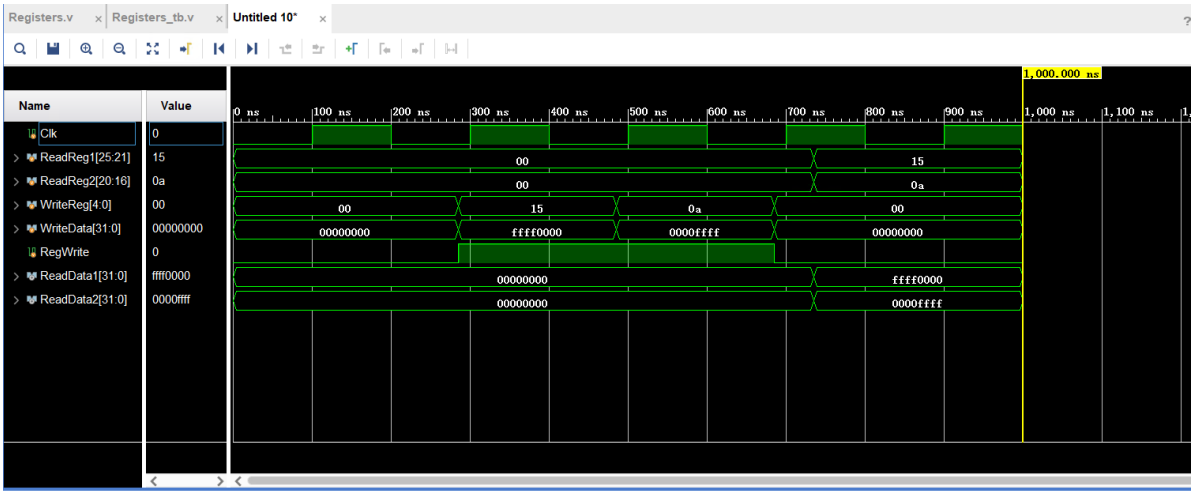
在激励文件中，不同时间间隔内我们不断改变不同的输入信号，测试模块不同功能的反馈

测试代码如下：

```
module Registers_tb(

    );
    reg [25 : 21] ReadReg1;
    reg [20 : 16] ReadReg2;
    reg [4 : 0] WriteReg;
    reg [31 : 0] WriteData;
    reg RegWrite;
    reg Clk;
    wire [31 : 0] ReadData1;
    wire [31 : 0] ReadData2;

    Registers register(
        .readReg1(ReadReg1),
        .readReg2(ReadReg2),
        .writeReg(WriteReg),
        .writeData(WriteData),
```

```
        .regWrite(RegWrite),
        .clk(Clk),
        .readData1(ReadData1),
        .readData2(ReadData2)
    );

    always #100 Clk = ~Clk;

    initial begin
        Clk = 0;
        ReadReg1 = 0;
        ReadReg2 = 0;
        WriteReg = 0;
        WriteData = 0;
        RegWrite = 0;

        #285;   //285
        RegWrite = 1;
        WriteReg = 5'b10101;
        WriteData = 32'b11111111111111110000000000000000;

        #200;   //485
        WriteReg = 5'b01010;
        WriteData = 32'b00000000000000001111111111111111;

        #200;   //685
        RegWrite = 0;
        WriteReg = 5'b00000;
        WriteData = 32'b00000000000000000000000000000000;

        #50;    //735
        ReadReg1 = 5'b10101;
        ReadReg2 = 5'b01010;
    end
endmodule
```

4.1.2 仿真图像



由图像可知，各输出结果都符合我们预期，实验成功

4.2 Data Memory仿真结果

使用 Verilog 编写激励文件，采用软件仿真的形式对存储器 (Data Memory) 模块进行测试

4.2.1 激励文件编写

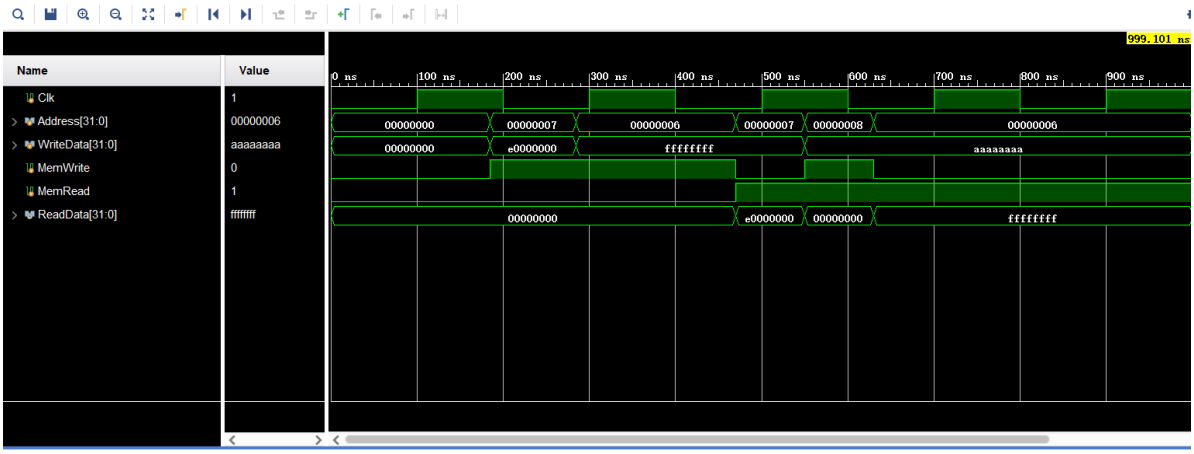
在激励文件中，类似对Registers的激励方式，不同时间间隔内我们不断改变不同的输入信号，测试模块不同功能的反馈

测试代码如下：

```
module dataMemory_tb(
    );

    reg Clk;
    reg [31 : 0] Address;
    reg [31 : 0] WriteData;
    reg MemWrite;
    reg MemRead;
    wire [31 : 0] ReadData;

    dataMemory data_mem(.clk(Clk), .address(Address), .writeData(WriteData),
                .memWrite(MemWrite), .memRead(MemRead),
.readData(ReadData));

    always #100 Clk = ~Clk;

    initial begin
        // Initialize Inputs
        Clk = 0;
        Address = 0;
        WriteData = 0;
        MemWrite = 0;
        MemRead = 0;

        // Current Time: 185 ns
        #185;
        MemWrite = 1;
        Address = 32'b00000000000000000000000000000111;
        WriteData = 32'b11100000000000000000000000000000;

        // Current Time: 285 ns
        #100;
        MemWrite = 1;
        WriteData = 32'hffffffff;
        Address = 32'b00000000000000000000000000000110;

        // Current Time: 470 ns
        #185;
        MemRead = 1;
        MemWrite = 0;
        Address = 32'b00000000000000000000000000000111;

        // Current Time: 550 ns
        #80;
        MemWrite = 1;
        Address = 8;
```

```
46          WriteData = 32'haaaaaaaa;
47
48          // Current Time: 630 ns
49          #80;
50          MemWrite = 0;
51          MemRead = 1;
52          Address = 32'b00000000000000000000000000000110;
53      end
54  endmodule
```

4.2.2 仿真图像



由图像可知，各输出结果都符合我们预期，实验成功

注意到ReadData的第一段，如果此时数据未经初始化的话，读到的应该是xxxxxxxx的未定义数据，所以在data memory中加入了初始化initial块初始化为0；第三段，由于写数据是在clk下降沿，因此在此之前读取数据时也应该是xxxxxxxx的未定义数据，根据实验指导书里的仿真示例，此处应该是输出为全0，因此我在data memory中也加入了一个条件判断逻辑，若读取数据时，本周期同时也会进行写操作（即MemWrite=1），则将输出置为0防止数据冲突，输出了错误的数据，在写入后再重新恢复读取

4.3 Sign Extension仿真结果

使用 Verilog 编写激励文件，采用软件仿真的形式对符号拓展单元 (Sign Extension) 模块进行测试

4.3.1 激励文件编写

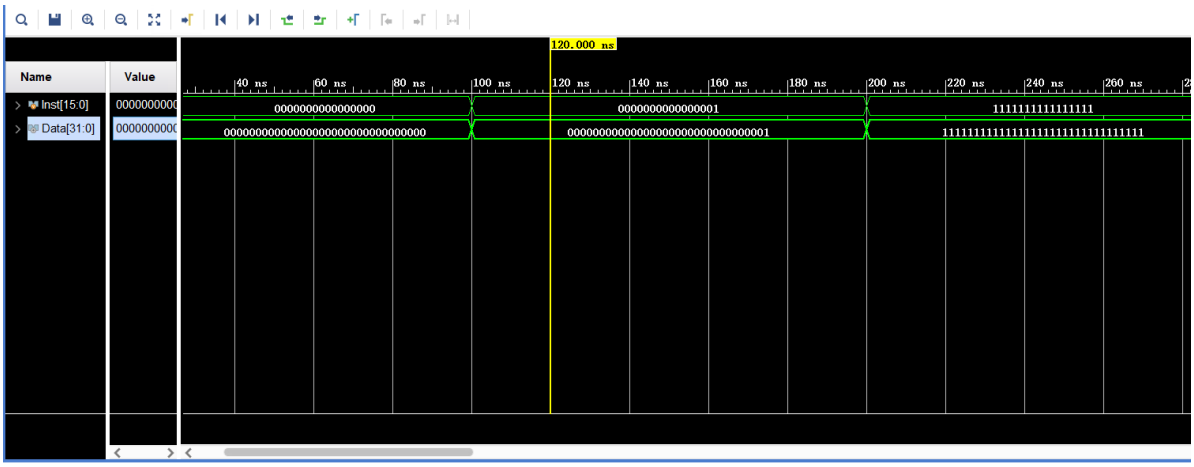
在激励文件中，我们每隔100ns改变一下输入的Inst的值对Sign Extension进行测试，测试数据涵盖了正数和负数

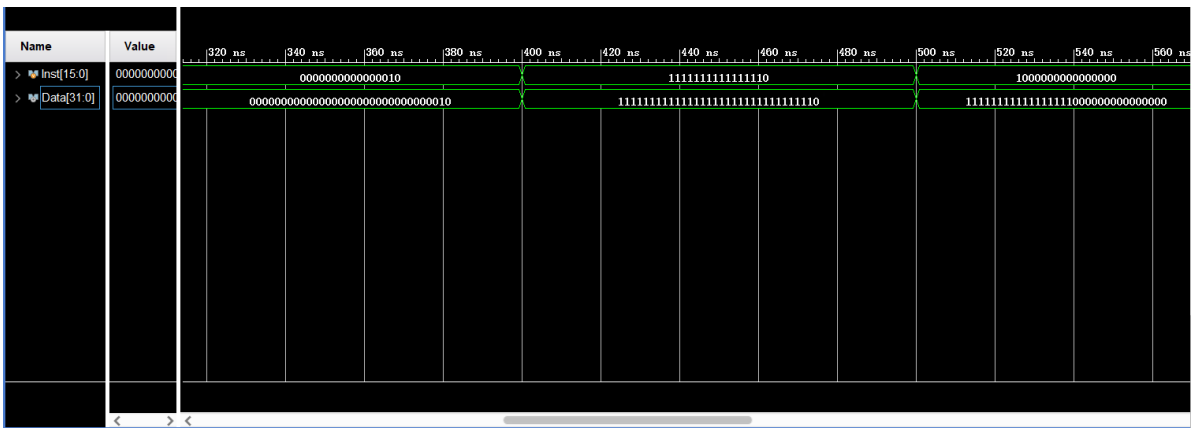
测试代码如下：

```
1   module signext_tb(
2       );
3
4       reg [15 : 0] Inst;
5       wire [31 : 0] Data;
6
7       signext sign_ext(.inst(Inst), .data(Data));
8
9       initial begin
10          Inst = 0;
11
12          #100;
```

```
13          Inst = 1;
14
15          #100;
16          Inst = 16'hffff;
17
18          #100;
19          Inst = 2;
20
21          #100;
22          Inst = 16'hfffe;
23
24          #100;
25          Inst = 16'hf000;
26
27          #100;
28          Inst = 16'h0123;
29      end
30  endmodule
```

4.3.2 仿真结果





由仿真结果可知，不管是正数还是负数，该模块都可以正确完成符号位拓展的功能，实验成功

5 反思总结

本实验设计并实现了类 MIPS 处理器的三个重要组成部件：寄存器 (Register)、存储器 (Data Memory)、有符号扩展单元 (Sign Extension)，并且通过软件仿真模拟的方法验证了它们的正确性，为后面的单周期类 MIPS 处理器以及流水线处理器的实现奠定基础。

之前lab03中完成的三个部件都是组合逻辑电路，而本实验中由于需要同步信号等，引入了clk，Register和Data Memory模块中都增加了时序逻辑的电路，这是本实验比较有挑战的地方。依靠一个统一的clk信号进行信号同步以及在下降沿进行写操作的设计理念，可以在同一个周期内完成读和写操作的同时又保证数据的正确性，也是在系统结构课程内学习数据通路时的一个重点，可以看到这个设计方式在整个类MIPS处理器的设计中都是非常重要的

在data memory的仿真实验中，初始的仿真结果与实验指导书给的仿真结果并不完全相同，主要就是ReadData中存在两段的xxxxxxxx未定义数据，综合分析为什么会有这两段数据及其存在的合理性，以及是否应该、如何才能将其修改为与实验指导书相同的仿真图，花费了我较多的时间，但这个过程也让我对于数据初始化，对于整个读写过程的前后关系有了更深的理解

在进行本实验时，系统结构课程还未完成数据通路部分的学习，因此当时还不太能理解clk下降沿写入这个设计方式的优点，由于是第一次进行时序逻辑电路的设计，因此还是有一点挑战的，完成之后我对于整个类MIPS处理器的结构以及各部分的数据通路联系都有了一个更好的理解，对于Verilog的语法等也愈发熟悉，对于完成接下来的完整的类MIPS处理器的实验非常有帮助