计算机系统结构实验

实验 4 报告

简单的类 MIPS 单周期处理器
部件实现–寄存器、存储器与有符号扩展

Log Creative

2021 年 6 月 27 日

目录

1 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. DataMemory的实现
4. 有符号扩展的实现
5. 使用行为仿真

2 原理分析

2.1 寄存器堆

寄存器堆是指令操作的主要对象。5 位的 `readReg1`，`readReg2`，`writeReg` 将用于指定需要的寄存器地址，即访问 32（$= 2^5$）个寄存器的其中一个。在时钟下沿，并且 `regWrite` 变为可用时，才会写入数据。（在这里规定均在时钟下沿写入数据，这样才不会读取错误的数据）读取得到的数据将会通过 `readData1` 和 `readData2` 输出。

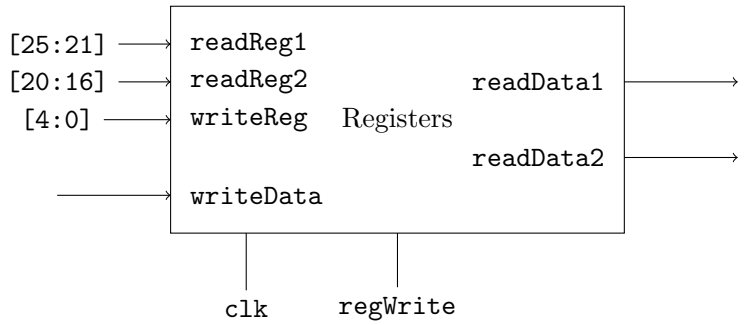

图 1: 寄存器堆模块

2.2 内存单元模块

内存模块获取 32 位地址 `address` 的输入。读写控制信号是独立的，一个时钟周期最多激活存储器读 `memRead` 和存储器写 `memWrite` 中的其中一个。需要写入的数据通过 `writeData` 获取，输出的数据从 `readData` 输出。

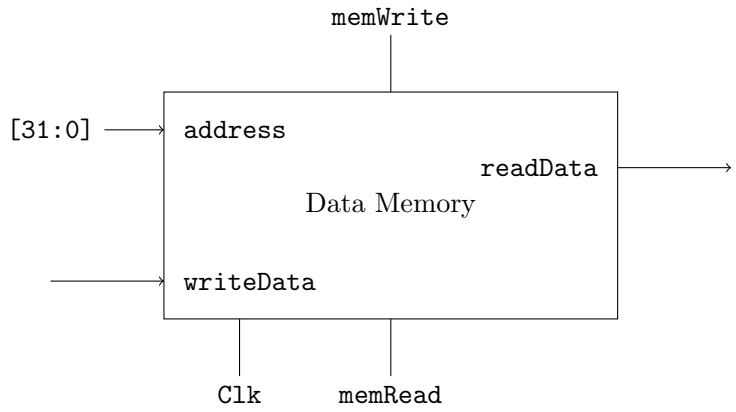

图 2: 内存模块

2.3 带符号扩展

将 16 位数符号扩展为 32 位的符号数。

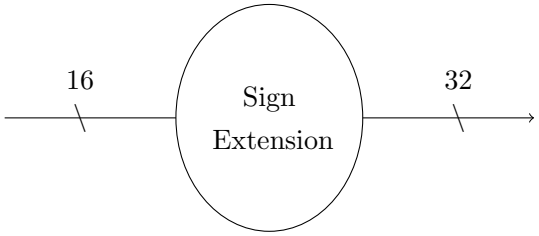

图 3: 带符号扩展

3 代码实现

3.1 寄存器堆

32 位的 MIPS 中共有 32 个 32 位的寄存器，在寄存器内部用下面的语句表示。

```
reg [31:0] RegFile [31:0];
```

按照原理，安排读取语句和写入语句。

Listing 1: Registers.v

```
reg [31:0] RegFile [31:0];
reg [31:0] ReadData1;
reg [31:0] ReadData2;

```

```
    always @(readReg1 or readReg2) begin
        ReadData1 = RegFile[readReg1];
        ReadData2 = RegFile[readReg2];
    end

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

    assign readData1 = ReadData1;
    assign readData2 = ReadData2;
```

3.2 内存单元模块

如果是读数据，如果 memRead 被激活或者地址发生了改变，首先会判断地址是否在物理地址空间内（这里存储器只有 64 个，地址空间却为 32 位），如果是就会从 readData 输出数据；否则是无效地址将清空读到的数据；如果是写数据，如果 memWrite 被激活，而且在物理地址空间内，则会在时钟下沿从 writeData 读取数据，并写入寄存器。

Listing 2: dataMemory.v

```
    reg [31:0] MemFile[0:63];

    always @(memRead or address) begin
        if(address<31'h00000020)
            readData = MemFile[address];
        else
            readData = 31'h00000000;
    end

    always @(negedge Clk) begin
        if(memWrite && address<31'h00000020)
            MemFile[address] = writeData;
    end
```

3.3 带符号扩展

这里采用了扩展复制第一位至前16位的方法进行符号扩展。

Listing 3: signext.v

```
assign data = {{16 {inst[15]}},inst[15:0]};
```

4 仿真结果

4.1 寄存器堆

寄存器堆的激励文件如下。时钟周期被设定为 100ns。

Listing 4: Registers_tb.v

```
initial begin
    readReg1 = 0;
    readReg2 = 0;
    writeReg = 0;
    writeData = 0;
    regWrite = 0;
    clk = 0;

    #285;
    regWrite = 1'b1;
    writeReg = 21;
    writeData = 32'b11111111111111110000000000000000;

    #200;
    writeReg = 10;
    writeData = 32'b00000000000000001111111111111111;

    #200;
    regWrite = 1'b0;
    writeReg = 0;
    writeData = 32'b00000000000000000000000000000000;

    #50;
    readReg1 = 5'b10101;
    readReg2 = 5'b01010;
end
```

图 4 显示了仿真结果。在时钟下沿稳定写入信号，向地址 21 写入 FFFF0000 ，向地址 10 写入 0000FFFF，然后读取地址 21 和 10 的数据，分别为 FFFF0000 和 0000FFFF 输出到 readData1 和 readData2。

4.2 内存单元模块

内存单元模块的激励文件如下。

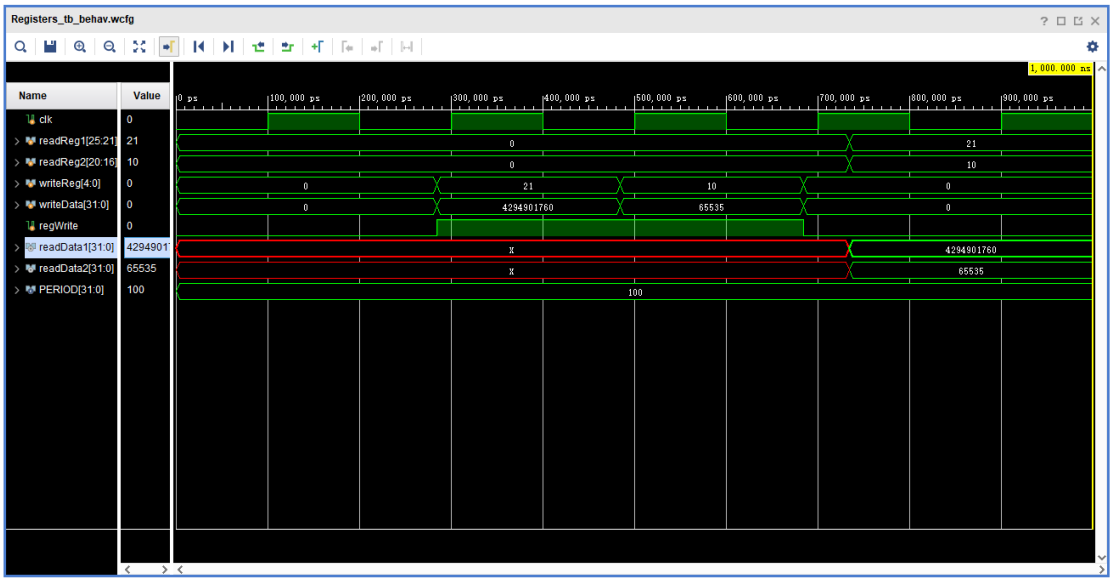

图 4: 寄存器堆的仿真结果

Listing 5: dataMemory_tb.v

```
initial begin
    address = 0;
    writeData = 0;
    Clk = 0;
    memWrite = 0;
    memRead = 0;

    #185;
    memWrite = 1'b1;
    address = 32'b00000000000000000000000000000111;
    writeData = 32'b11100000000000000000000000000000;

    #100;
    memWrite = 1'b1;
    writeData = 32'hffffffff;
    address = 32'b00000000000000000000000000000110;

    #185;
    memRead = 1'b1;
    memWrite = 1'b0;
    address = 32'b00000000000000000000000000000111;

    #80;
    memWrite = 1;
    address = 8;
    writeData = 32'haaaaaaaa;

    #80;
    memWrite = 0;
    memRead = 1;
```

```
        address = 6;
    end
```

内存模块的仿真结果如图 5 所示。在后面的读取数据时，地址 7 读到了之前写入的 e0000000，地址 6 读到了之前写入的 aaaaaaaa。仿真结果是正确的。

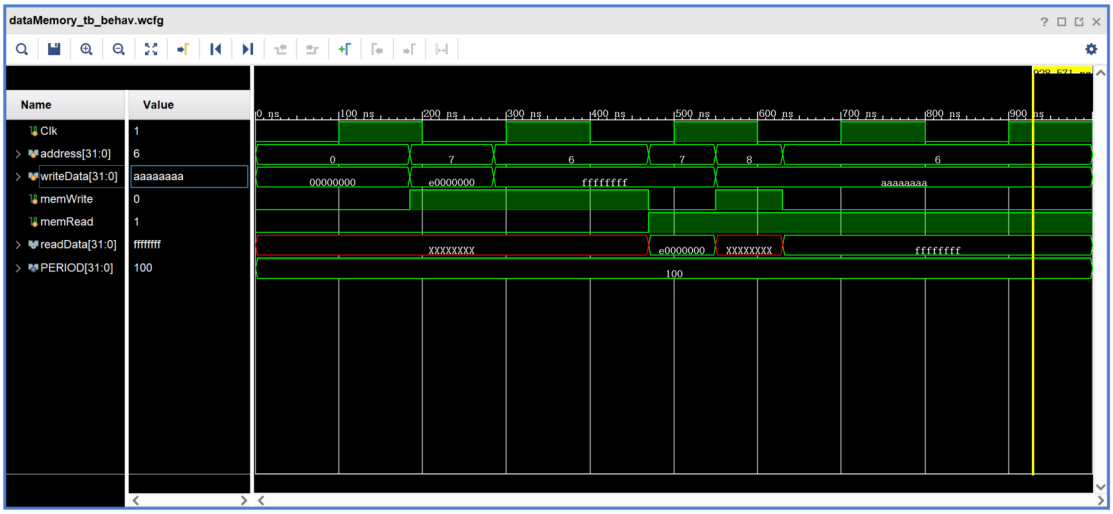

图 5: 存储模块仿真结果

4.3 带符号扩展

带符号扩展的激励文件如下。

Listing 6: signext_tb.v

```
initial begin
    inst = 0;

    #100 inst = 1;
    #100 inst = -1;
    #100 inst = 2;
    #100 inst = -2;
end
```

带符号扩展的仿真结果如图 6 所示。对于所有的输入，符号位扩展是正确的。

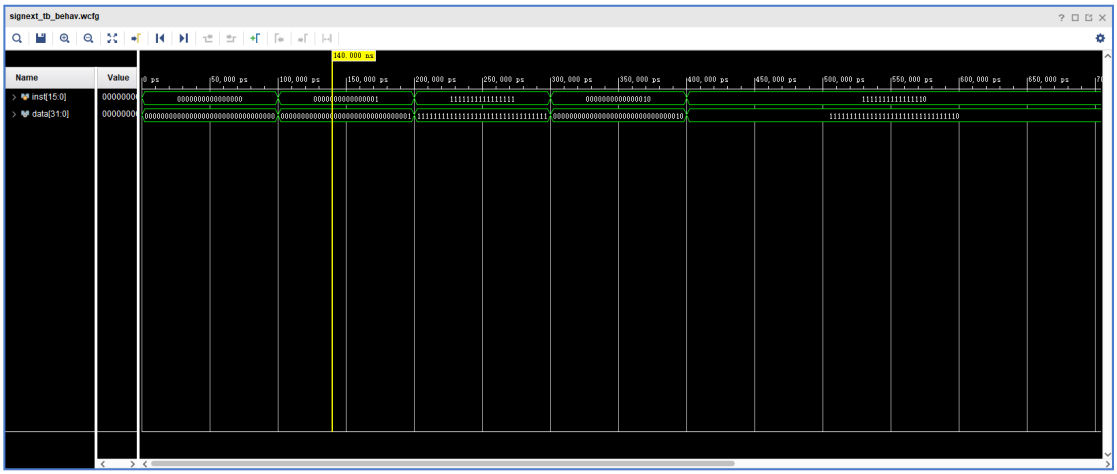

图 6: 带符号扩展仿真结果

5 实验心得

通过本次实验，更加熟悉了三个部件的构造方法。特别是时钟下沿的写入，是值得注意的。对于内存地址变化时就应当重新读取这一点也是需要特别注意的，否则会导致地址变化时无响应情形。需要注意是有效地址的内存才可以读取和写入，这一点在之后的设计中也是非常重要的。符号位扩展再次熟悉了拼接操作。