CSE-COA-LAB-004

计算机组成与系统实验

马叶晟[1] [2]

5140209064

2016 年 4 月 6 日

[1] email: kimi.ysma@gmail.com

[2] 上海交通大学，计算机科学与工程系.

1 实验介绍

1.1 实验名称

简单的类MIPS单周期处理器的实现：寄存器与内存.

1.2 实验目的

1. 理解CPU的寄存器与内存.
2. 使用Verilog进行储存期间的设计
3. 使用ISim进行行为级仿真

1.3 实验范围

本次实验将覆盖以下范围：

1. ISE 13.4的使用
2. 使用VerilogHDL进行逻辑设计
3. register, data memory, sign extension的实现
4. 符号扩展的实现

2 实验内容

2.1 register模块的实现

2.1.1 register模块概述

寄存器模块接受来自top的时钟信号作为写的信号，使得整个寄存器模块是一个同步的整体. 寄存器的读写操作按照相应的规程完成即可. 在MIPS中，寄存器的使用是十分频繁的，因为在计算机系统中寄存器的访问远远快于内存的访问. 在MIPS32中，寄存器分为两类: 通用寄存器(GPR)和特殊寄存器. 通用寄存器共有32个，用$0,$1...$31表示，这些寄存器都32位的，其中$0通常用来表示常数0. 一般而言，各个通用寄存器都遵循一系列约定，而这些约定通常由汇编程序员去遵循. MIPS32架构中的特殊寄存器有三个，对我们实现的精简版的MIPS指令集而言，我们需要关心的是PC(program counter 程序计数器).

对于寄存器的读和写而言，读在任何时间都是可以被执行的，不需要enable信号; 对于写信号，规定在时钟的下降沿边沿触发，同时也需要写enable信号、输入数据和输入地址来完成写的操作.

寄存器的示意图如下所示:

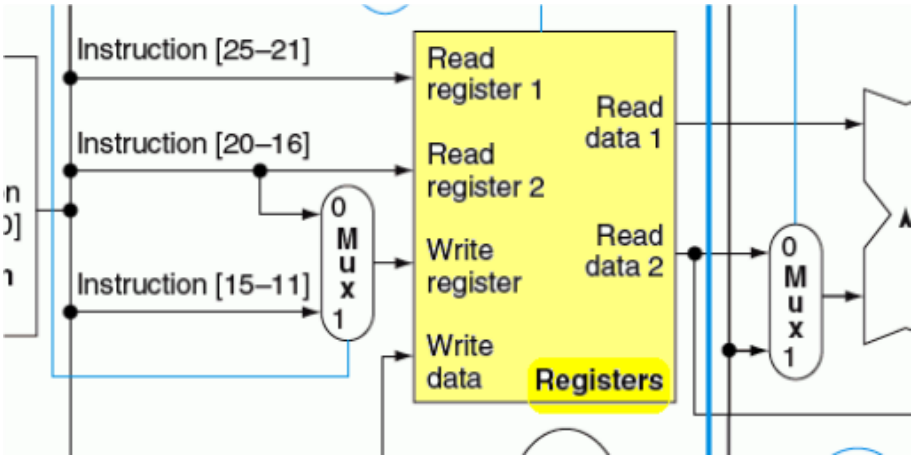

图 1: 寄存器模块的示意图

2.1.2 register模块的端口

寄存器模块的输入端口是: clock_in, regWrite, readReg1, readReg2, writeReg, writeData.

寄存器模块的输出端口是: readData1, readData2.

2.1.3 register模块的Verilog代码实现

在寄存器模块的实现中，有几个需要注意的地方. 首先，在定义变量的时候需要注意变量的位数，有的端口是4位的，有的端口是32位的. 其次，在always块中要注意触发读和写的信号，有时候如果在这些触发信号中漏了一些的话，可能得到的仿真结果对于一些情况是正确的而对另外一些情况是错误

的，而这种类型的bug又是最难以排除的，可能会给lab5和lab6的调试带来许多的困难. 寄存器模块的代码实现见附录1.

2.1.4 register模块的波形仿真

在register模块的仿真中，我使用的测试样例如下所示:

```
    #285;
    regWrite = 1'b1;
    writeReg = 5'b10101;
    writeData = 32'hffff0000;

    #200;
    writeReg = 5'b01010;
    writeData = 32'h0000ffff;

    #200;
    regWrite = 1'b0;
    writeReg = 5'b00000;
    writeData = 32'h00000000;

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

实验仿真的波形图如下所示：

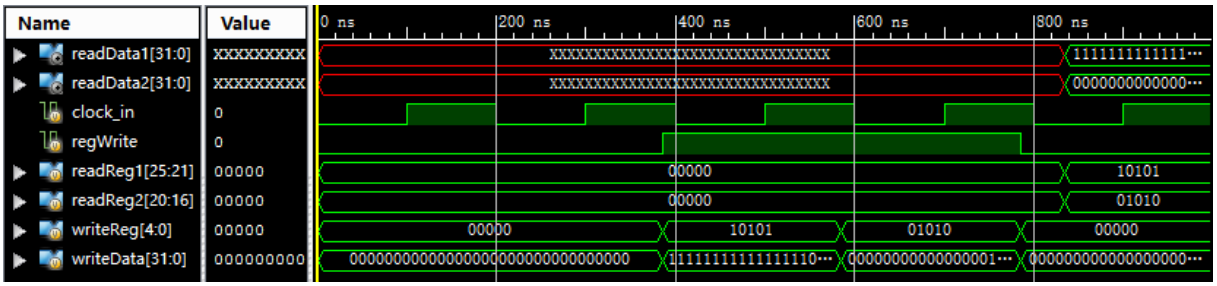

图 2: 寄存器模块的仿真波形图

2.2 内存模块dataMem的实现

2.2.1 dataMem模块概述

dataMem模块的实现与寄存器模块类似，唯一区别是内存模块通过地址来寻址，因为内存相对于寄存器而言需要更大的储存量，因此不可能只通过5位的地址来寻址.

内存模块的示意图如下所示:

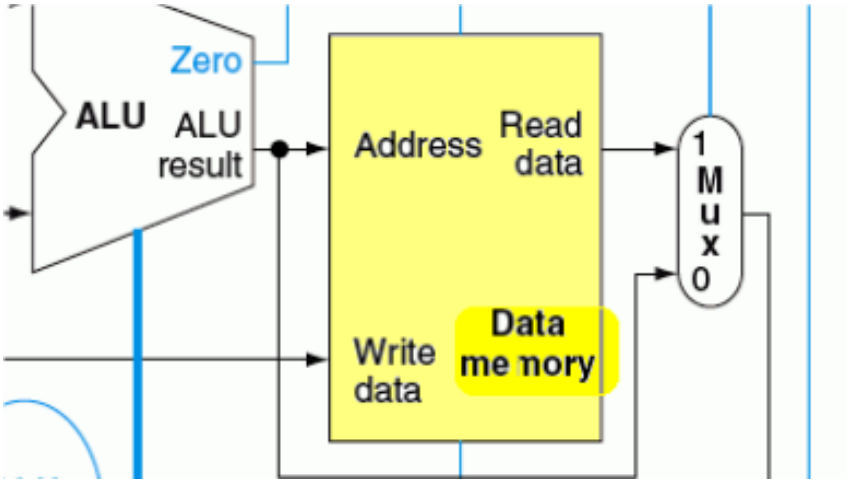

图 3: 内存模块的示意图

2.2.2 dataMem模块的端口

dataMem模块的输入端口有: clock_in, address, writeData, memRead, memWrite.
dataMem模块的输出端口有: readData.

2.2.3 dataMem模块的Verilog代码实现

dataMem模块的Verilog代码实现和寄存器模块是类似的，同样需要注意相关的时序和同步等问题. 内存模块的代码实现见附录2.

2.2.4 dataMem模块的仿真波形

在dataMem模块的仿真中，我使用的测试样例如下所示:

```
1      #185;
2      memWrite = 1'b1;
3      address = 32'h0000000f;
4      writeData = 32'hffff0000;
5
6      #250;
7      memRead = 1'b1;
8      memWrite = 1'b0;
```

实验仿真的波形图如下所示：

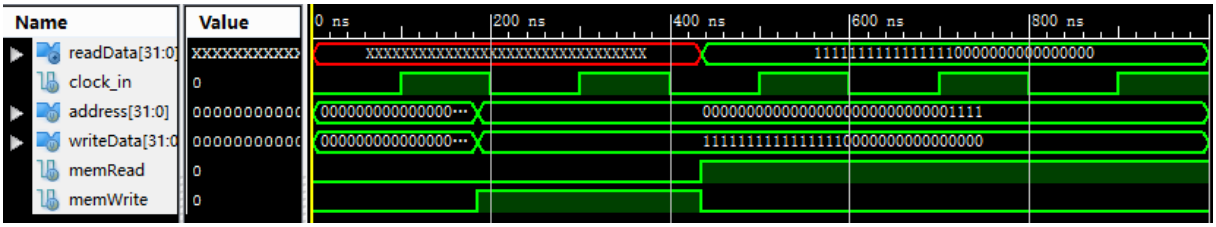

图 4: 内存模块的仿真波形图

2.3 符号扩展模块signExt的实现

2.3.1 signExt模块概述

signExt模块的实现也比较简单，只要知道在Verilog语言中组合数据的规则就可以了.
符号扩展的示意图如下所示：

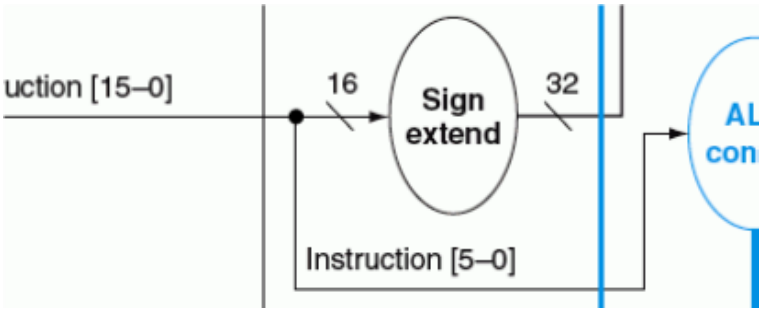

图 5: 符号扩展的示意图

2.3.2 signExt模块的端口

符号扩展模块的输入端口是: inst.
符号扩展模块的输出端口是: data.

2.3.3 signExt模块的Verilog代码实现

signExt模块的代码实现就是把MIPS指令的后16位与第16位的扩展连接起来. 具体的signExt代码实现见附录3.

2.3.4 signExt的仿真波形图

在signExt模块的仿真中，我使用的测试样例如下所示:

```
1     #100;
2     inst = 16'h0000;
3     #100;
```

```
4      inst = 16'h1000;
5      #100;
6      inst = 16'h1111;
```

实验仿真的波形图如下所示：

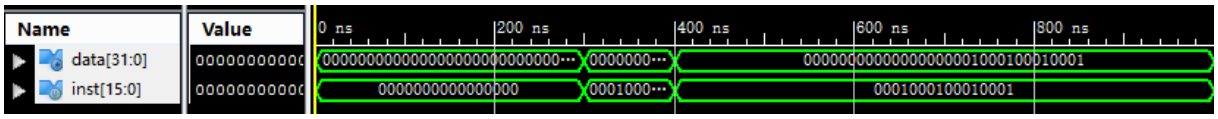

图 6: 符号扩展模块的仿真波形图

3 心得体会

lab4主要完成了3个内容，分别是寄存器、内存和符号扩展单元的实现. 这几个单元是我们在最终实现完整的CPU之前要完成的最后几个模块，而在lab5中我们就将继续完成top模块的实现.

在编写Verilog代码的时候，同一个变量不能同时被几个源驱动，线网类型是这样，寄存器类型也是这样. 如果一个变量同时被若干个值驱动的话，会导致编译器无法确定一个变量究竟是如何被驱动的. 值得注意的是，在寄存器和内存模块中，时序逻辑的正确性非常重要，如果时序逻辑出错的话，在之后的CPU整体调试中可能导致各种各样的问题，而这些由时序逻辑引起的问题在很大程度上是难以排除的，因此我们最好在一开始实现寄存器和内存的时候就把时序逻辑理清楚，避免之后不必要的时间上的浪费. 这里Verilog的编程也凸显出上个学期学习的数字电子技术这门课程的重要性.

在ISE环境下编写Verilog代码，我们就要充分利用好这个工具. 在Verilog编译器报错的时候，我们可以直接点击错误的超链接，可以查询Xilinx的官方文档或者官方论坛，大多数问题在那里都有比较权威的解答. 与之相比，百度得到的结果就显得在质量上有一些参差不齐了.

4 致谢

感谢老师在实验上的仔细指导. 也感谢在网络上帮助别人、热心回答问题的专业人士，他们的回答给我在很多问题的解决上很大的启发.

附录1

register模块的代码实现如下所示：

```
module register(
    clock_in,
    regWrite,
    readReg1,
    readReg2,
    writeReg,
    writeData,
    readData1,
    readData2,
    );

    input clock_in;
    input regWrite;
    input [25:21] readReg1;
    input [20:16] readReg2;
    input [4:0] writeReg;
    input [31:0] writeData;
    output readData1;
    output readData2;

    reg [31:0] readData1;
    reg [31:0] readData2;
    reg [31:0] regFile[31:0];

    always @ (regWrite or readReg1 or readReg2 or writeReg or
       writeData)
    begin
    readData1 <= regFile[readReg1];
    readData2 <= regFile[readReg2];
    end

    always @ (negedge clock_in)
    begin
    if (regWrite)
        regFile[writeReg] <= writeData;
    end
endmodule
```

附录2

dataMem模块的代码实现如下所示：

```
module data_memory(
    clock_in,
    address,
    readData,
    writeData,
    memRead,
    memWrite
    );

    input clock_in;
    input [31:0] address;
    input [31:0] writeData;
```

```
13      input memRead;
14      input memWrite;
15      output reg [31:0] readData;
16
17      reg [31:0] memFile[0:127];
18
19      always @ (address or memRead)
20      begin
21          if (memRead)
22              readData <= memFile[address];
23      end
24
25       always @ (negedge clock_in)
26       begin
27           if (memWrite)
28               memFile[address] = writeData;
29       end
30  endmodule
```

附录3

signExt模块的代码实现如下所示：

```
1  module signext(
2      inst,
3      data
4      );
5      input [15:0] inst;
6      output [31:0] data;
7      assign data = {{16{inst[15]}}, inst};
8  endmodule
```