

SHANGHAI JIAOTONG UNIVERSITY



计算机系统结构实验报告 – Lab4

姓名：刘一凡

学号：518021910609

完成时间：2020/4/19

目录

[1. 概述 3](#_Toc40977610)

[1.1 实验名称 3](#_Toc40977611)

[1.2 实验目的 3](#_Toc40977612)

[2. 实验描述 3](#_Toc40977613)

[2.1 寄存器模块 3](#_Toc40977614)

[2.1.1 模块描述 3](#_Toc40977615)

[2.1.2 寄存器模块代码 4](#_Toc40977616)

[2.1.3 寄存器模块仿真测试代码 4](#_Toc40977617)

[2.1.4 仿真波形展示 6](#_Toc40977618)

[2.1.5 实验结论 6](#_Toc40977619)

[2.2 内存单元模块 7](#_Toc40977620)

[2.2.1 模块描述 7](#_Toc40977621)

[2.2.2 内存单元模块代码 7](#_Toc40977622)

[2.2.3 内存模块仿真测试代码 8](#_Toc40977623)

[2.2.4 仿真波形展示 9](#_Toc40977624)

[2.2.5 实验结论 10](#_Toc40977625)

[2.3 带符号扩展 10](#_Toc40977626)

[2.3.1 模块描述 10](#_Toc40977627)

[2.3.2 带符号扩展模块代码 10](#_Toc40977628)

[2.3.3 带符号扩展模块仿真测试代码 11](#_Toc40977629)

[2.3.4 仿真波形展示 11](#_Toc40977630)

[2.3.5 实验结论 12](#_Toc40977631)

[3 实验心得 12](#_Toc40977632)

# 1. 概述

## 1.1 实验名称

简单的类MIPS 单周期处理器部件实现– 控制器，ALU

## 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．使用行为仿真

# 2. 实验描述

## 2.1 寄存器模块

### 2.1.1 模块描述

寄存器是指令操作的主要对象，32 位的MIPS 中共有32 个32 位的寄存器。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。

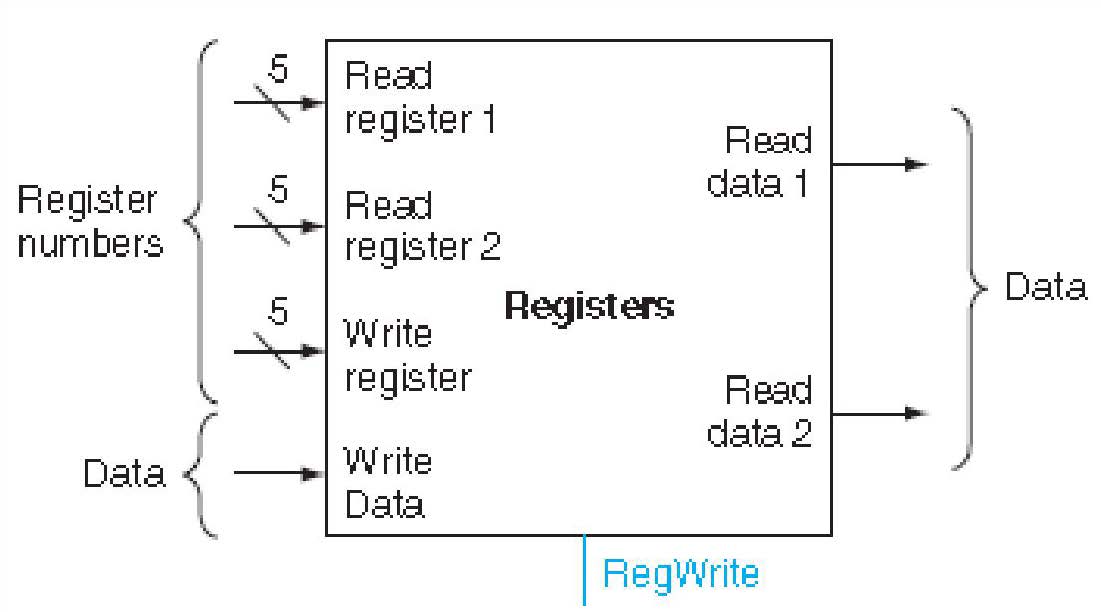


图1：寄存器模块

### 2.1.2 寄存器模块代码

1. `timescale 1ns / 1ps

4. module Registers(
5. input clock\_in,
6. input [25:21] readReg1,
7. input [20:16] readReg2,
8. input [4:0] writeReg,
9. input [31:0] writeData,
10. input regWrite,
11. output [31:0] readData1,
12. output [31:0] readData2
13. );
14. reg [31:0] regFile[31:0];
15. reg [31:0] readData1;
16. reg [31:0] readData2;
18. always @ (readReg1 or readReg2 or regWrite)
19. begin
20. regFile[0]=0;
21. readData1 = regFile[readReg1];
22. readData2 = regFile[readReg2];
23. end
24. always @ (negedge clock\_in)
25. begin
26. **if** (regWrite == 1)
27. regFile[writeReg] = writeData;
28. end
29. endmodule

### 2.1.3 寄存器模块仿真测试代码

1. `timescale 1ns / 1ps
3. module Registers\_tb(
5. );
6. //input
7. reg clock\_in;
8. reg [25:21] readReg1;
9. reg [20:16] readReg2;
10. reg [4:0] writeReg;
11. reg [31:0] writeData;
12. reg regWrite;
14. // output
15. wire [31:0] readData1;
16. wire [31:0] readData2;
18. Registers u0 (
19. .clock\_in(clock\_in),
20. .readReg1(readReg1),
21. .readReg2(readReg2),
22. .writeReg(writeReg),
23. .regWrite(regWrite),
24. .writeData(writeData),
25. .readData1(readData1),
26. .readData2(readData2)
27. );
29. always
30. begin
31. #100 clock\_in = ~clock\_in;
32. end
34. initial begin
35. // Initialize Inputs
36. clock\_in = 0;
37. readReg1 = 0;
38. readReg2 = 0;
39. regWrite = 0;
40. writeReg = 0;
41. writeData = 0;
43. #100;
44. clock\_in = 0;
45. readReg1 = 0;
46. readReg2 = 0;
47. writeReg = 0;
48. regWrite = 0;
49. writeData = 0;
50. //current time 285
51. #285;
52. regWrite = 1'b1;
53. writeReg = 5'b10101;
54. writeData = 32'b11111111111111110000000000000000;
55. //current time 485
56. #200;
57. writeReg = 5'b01010;
58. writeData = 32'b00000000000000001111111111111111;
60. #200;
61. regWrite = 1'b0;
62. writeReg = 5'b00000;
63. writeData = 32'b00000000000000000000000000000000;
64. //735
65. #50;
66. readReg1 = 5'b10101;
67. readReg2 = 5'b01010;
68. end
69. endmodule

### 2.1.4 仿真波形展示

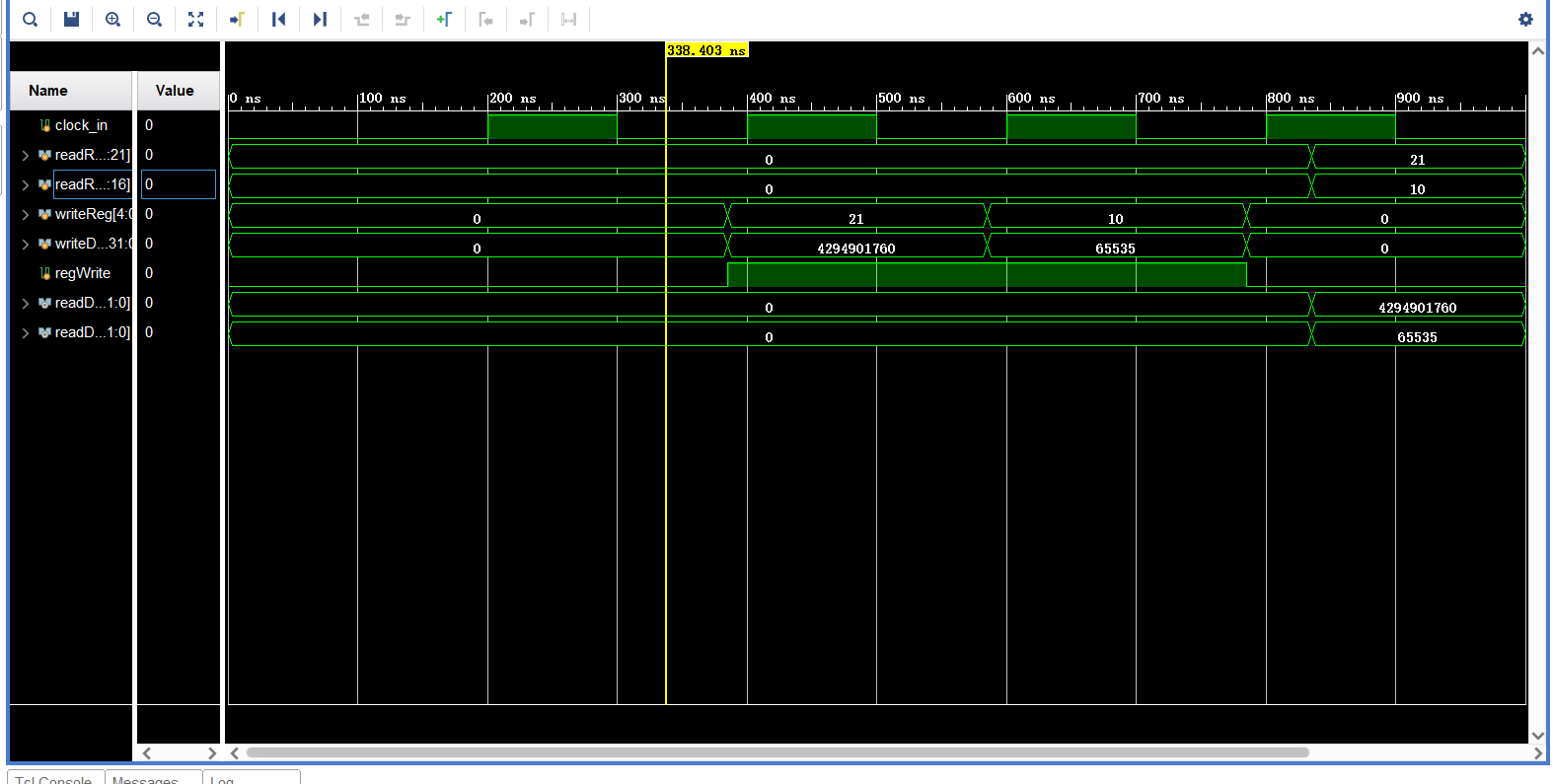


图2：寄存器模块仿真波形图

### 2.1.5 实验结论

仿真波形说明Register模块能够正确地根据readReg信号读取寄存器中的相关内容，并在时钟的下降沿根据writeReg和regWrite将writeData写入寄存器。由此可见，Register单元模块实验成功。

## 2.2 内存单元模块

### 2.2.1 模块描述

内存模块与register模块相似，由于写数据也要考虑信号同步，因此也需要clock\_in。内存单元的实现，也可以用系统带有的Block Memory来生成。实际上，该模块就是随机访问存储器（RAM）。

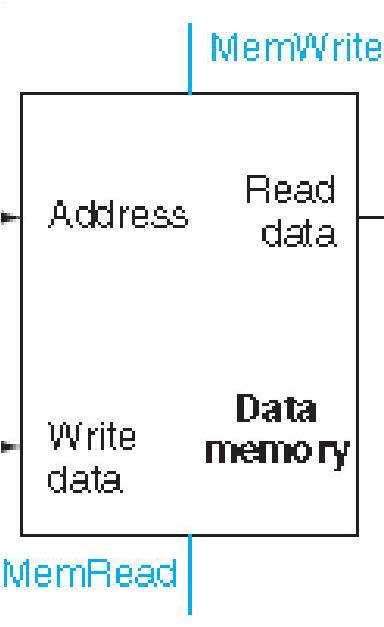


图3：内存单元模块

### 2.2.2 内存单元模块代码

1. `timescale 1ns / 1ps
3. module DataMemory(
4. input Clk,
5. input [31:0] address,
6. input [31:0] writeData,
7. input memWrite,
8. input memRead,
9. output [31:0] readData
10. );
11. reg [31:0] readData;
12. reg [31:0] memFile [0:63];
13. always @ (address)
14. begin
15. **if**(memRead&&!memWrite)
16. readData=memFile[address];
17. **else**
18. readData=0;
19. end
20. always @ ( negedge Clk)
21. begin
22. **if**(memWrite==1)
23. memFile[address]=writeData;
24. end
25. endmodule

### 2.2.3 内存模块仿真测试代码

1. `timescale 1ns / 1ps
3. module dataMemory\_tb(
5. );
6. reg Clk;
7. reg [31:0] address;
8. reg [31:0] writeData;
9. reg memWrite;
10. reg memRead;
12. wire [31:0] readData;
14. DataMemory u0(
15. .Clk(Clk),
16. .address(address),
17. .writeData(writeData),
18. .memWrite(memWrite),
19. .memRead(memRead),
20. .readData(readData)
21. );
22. always #100 Clk = ~Clk;
24. initial begin
25. Clk = 0;
26. address = 0;
27. writeData = 0;
28. memWrite = 0;
29. memRead = 0;
31. #185;
32. memWrite=1'b1;
33. address=32'b000000000000000000000111;
34. writeData=32'he0000000;
36. #100;
37. memWrite=1'b1;
38. writeData=32'hffffffff;
39. address=32'b000000000000000000000110;
41. #185;
42. memRead=1'b1;
43. memWrite=1'b0;
44. address=7;
46. #80;
47. memWrite=1'b1;
48. address=8;
49. writeData=32'haaaaaaaa;
51. #80;
52. address=6;
53. memWrite=1'b0;
54. memRead=1'b1;
56. end
57. endmodule

### 2.2.4 仿真波形展示

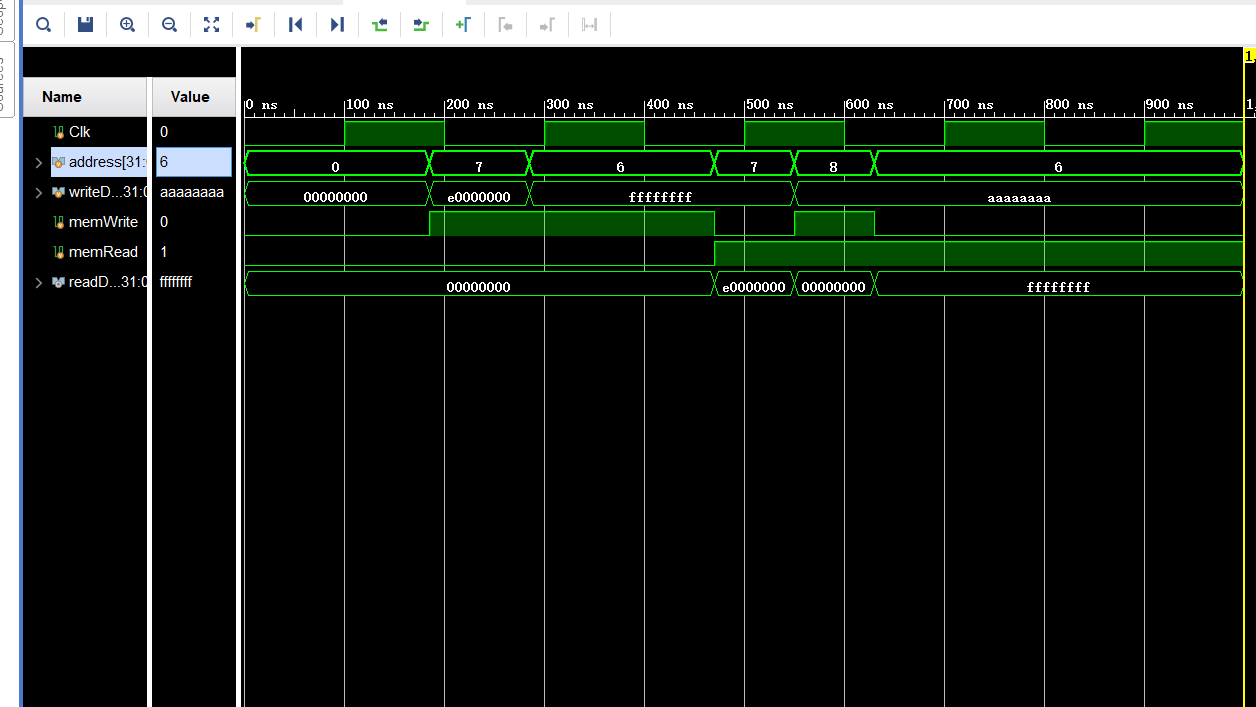


图4：内存单元模块仿真波形图

### 2.2.5 实验结论

仿真波形说明此模块能够正确地根据memRead，address信号读取内存，并在时钟的下降沿根据memWrite，address信号将writeData写入内存，也可以读取内存中的数据到readData中。

当memRead和memWrite均为高电平时，如果写入的地址之前没有数据，那么readData应该为xxxxxxxx。而根据仿真波形的示例，此时应该输出00000000，所以我在实现时进行了初始化，在开始时刻readData的取值为0，从而得到了正确的波形。内存单元模块实验成功。

## 2.3 带符号扩展

### 2.3.1 模块描述

将16位有符号数扩展为32位有符号数。

补码的定义：

（1）正数的补码：与原码相同。

（2）负数的补码：符号位为1，其余位为该数绝对值的原码按位取反，然后整个数加1。

要进行带符号的扩展，我们只需要在前面补足符号即可。

### 2.3.2 带符号扩展模块代码

1. `timescale 1ns / 1ps
3. module signext(
4. input [15:0] inst,
5. output [31:0] data
6. );
8. assign data=(inst[15])? (inst|32'hffff0000):(inst|32'h00000000);
9. endmodule

### 2.3.3 带符号扩展模块仿真测试代码

1. `timescale 1ns / 1ps
3. module signext\_tb(
5. );
7. reg [15:0] inst;
8. wire [31:0] data;
9. signext u0(.inst(inst),
10. .data(data));
12. initial begin
13. inst=0;
14. #100; inst=1;
15. #100; inst=-1;
16. #100; inst=2;
17. #100; inst=-2;
18. #100;
19. end
20. endmodule

### 2.3.4 仿真波形展示

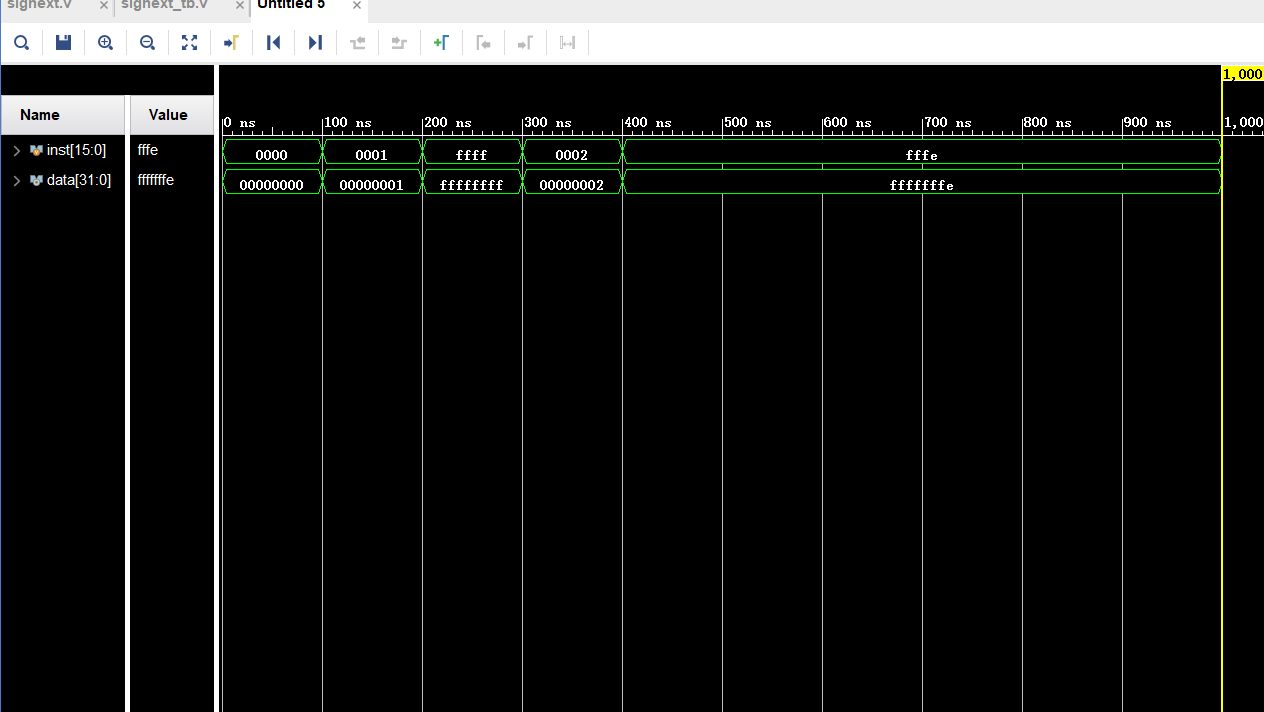


图5：带符号扩展模块仿真波形图

### 2.3.5 实验结论

仿真波形说明带符号扩展模块能够成功将16位补码表示拓展为32位补码，当输入为1，-1，2，-2时能输出正确的扩展后的结果。带符号扩展模块实验成功。

# 3 实验心得

本次实验主要实现的是寄存器，内存的读写操作以及符号位的扩展操作，用到了读写的功能。存储器的读取功能是组合逻辑，但是写入功能则是时序逻辑。因此，在进行内存的写入时需要考虑时钟的同步问题。这次实验中，我初步了解了组合逻辑和时序逻辑的区别。在一开始进行内存读写代码的编写的时候，我没有初始化地址对应的内存的值，导致输出的不是00000000而是xxxxxxxx，在仔细对比了我的仿真波形与示例波形后，我发现了这一问题并赋了初值，从而得到了正确的仿真波形图。

这次实验相对是比较简单的，但是为了得到和示例相仿的波形，还需要在代码的实现逻辑和细节上多加注意，在发现波形不符时通过观察理想输出值和实际输出值的区别，定位出问题的具体位置，再进行修改。

在写有符号扩展时，有同学提到过Verilog的拼接语句，我发现这个语句使有符号扩展部分的代码更加精简。可以这么简单地书写有符号扩展部分的代码：

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

通过查询一些资料，我对Verilog语言有了更多的认识。