计算机系统结构实验

lab4: 类 MIPS 单周期处理器存储部件的设计与实现(二)

周旭东 521021910829

2023 April

摘要

学习理解寄存器、数据存储器、有符号扩展单元的 IO 定义，并设计对应三个单元模块，随后对实现的各个模块进行仿真测试。

目录

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 寄存器 Registers 原理

寄存器是用于存储和操作数据的主要组件之一。在 MIPS 架构中，有 32 个 32 位通用寄存器（General Purpose Registers，GPRs），它们用于存储整数数据。

这 32 个寄存器被分配一个编号，从 0 到 31，每个寄存器都有唯一的名称。其中 0 号寄存器（*zero*）总是包含值为 0，不允许对其进行写入操作。1 号寄存器（*at*）被用于汇编器的一些指令，不建议在程序中使用。其他寄存器（2 到 31）则可以在程序中自由使用。

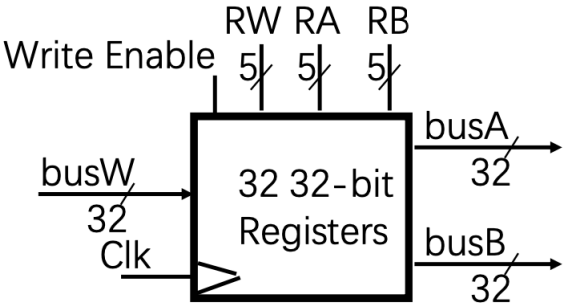

图 1: 寄存器结构

图 1给出了 MIPS 寄存器文件结构，该模块由 32 个 32 位寄存器构成；在 Clk 下降沿进行写操作，设计读操作与时钟无关；另外设计 reset 重制信号下的寄存器清空操作，将其与 clk 下降沿写在同一“always”语块中，避免同时满足时赋值混乱。清空方法为通过循环将寄存器文件中每个寄存器值设为 0。

寄存器的输入有写使能信号、32 位的写入数据、选通的寄存器地址。输出有选通的两个读寄存器的数据。在寄存器中，写使能信号用于控制是否允许写入数据，32 位的写入数

据是将要存储在指定寄存器中的数据，选通的寄存器地址用于指定要进行读写的寄存器。读操作则是通过选通读取的寄存器地址来输出该寄存器中存储的数据。

表 1定义了寄存器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	Clk	改变寄存器状态需要时钟边沿触发	clk
	Write Enable	写寄存器使能信号	regWrite
	busW（32位）	32位输入，指定写入数据寄存器	writeData
	RW（5位）	选通Rw指定的寄存器	writeReg
	RA（5位）	选通RA指定的寄存器	readReg1
	RB（5位）	选通RB指定的寄存器	readReg2
输出	busA（32位）	RA有效时输出读取的数据	readData1
	busB（32位）	RB有效时输出读取的数据	readData2

表 1: 寄存器各信号量及其定义

2.2 数据存储器 Data Memory 原理

数据存储器用于存储程序中使用的数据。在 MIPS 架构中，数据存储器也以字为单位进行寻址。MIPS 架构使用 Load/Store 体系结构，即所有数据的操作都必须通过 Load/Store 指令从数据存储器中读取或写入寄存器。数据存储器可以被访问的方式包括：字节（byte）、半字（half word）和字（word）。

MIPS 中使用的存储器地址是 32 位的，即存储器可以寻址 2^{32} 个字节，也就是 4GB 的存储空间。存储器可以通过它们的地址进行引用。例如，可以使用指令 lw（load word）从数据存储器中读取一个字，使用指令 sw（store word）将一个字写入数据存储器。存储器地址可以使用立即数、寄存器或寄存器相对地址的方式来计算。

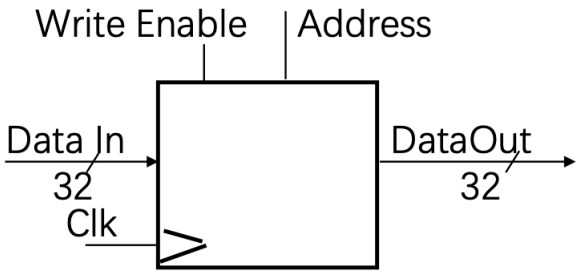

图 2: 存储器结构

图 2展示了存储器的结构，其中自定义内存大小为 256 * 32 位。

存储器的主要功能是存储和读取数据，可以根据地址信号寻址到对应的存储单元。对于 RAM 和 ROM 等可读写存储器，可以使用 Write Enable 信号控制数据的写入；对于只读存储器 ROM，没有写使能信号，只能从指定地址读取数据。它有一个 32 位的地址总线，可以寻址 2^{32} 个不同的地址，也就是 4GB 的空间。存储器的输入信号包括时钟边沿触发信号 Clk、写存储器使能信号 Write Enable、要写入存储器的数据 Data In，以及指定存储器地址的地址信号 Address。存储器的输出信号是读取的数据 DataOut，它的大小也是 32 位。

表 2定义了存储器模块各个输入输出信号量的定义并给出了相应功能的描述。时钟下降沿进行写存储器操作，读操作在读信号和地址的控制下持续进行。

输入/输出量	信号线	描述	定义的信号量
输入	Clk	改变寄存器状态需要时钟边沿触发	clk
	Write Enable	写存储器使能信号	memWrite
	Data In（32位）	写入存储器数据	writeData
	Address（32位）	指定存储器地址	address
输出	DataOut（32位）	输出读取的数据	readData

表 2: 存储器各信号量及其定义

2.3 符号扩展器 Sign Extension 原理

符号扩展器，也称为有符号拓展器部件，是一种用于将有符号数的低位位数拓展到高位的硬件电路。在 MIPS 指令集架构中，符号扩展器通常用于执行算术和逻辑运算时，将 8 位或 16 位的有符号数扩展为 32 位有符号数。

符号扩展器通过将原始数据的最高位（即符号位）复制到高位来进行操作，这样可以保持原始数值的符号不变。例如，对于一个 16 位有符号数，如果其符号位为 1，则在扩展过程中，符号扩展器会将高 16 位填充为 1，以保持符号不变。

在 MIPS 指令集架构中，符号扩展器通常用于执行算术和逻辑指令，如 ADD、SUB、AND、OR、XOR 等。在这些指令中，操作数必须在执行前进行符号扩展，以确保正确的操作结果。

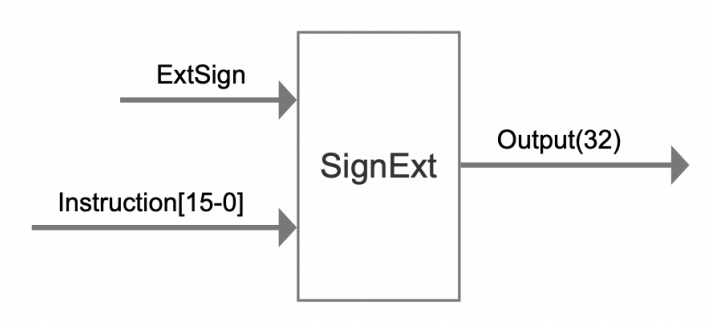

图 3: 符号拓展器结构

表 3展示了符号拓展器 SignExt 的结构。

3 功能实现

3.1 Registers 模块

Registers 模块内部设计了 3 个 reg 类型的变量：RegFile、ReadData1 和 ReadData2。RegFile 是一个 32 个 32 位寄存器的数组，用于存储 32 个寄存器的数据。ReadData1 和 ReadData2 是用于存储从寄存器 1 和寄存器 2 中读取的数据的变量。

模块的 initial 块用于初始化 RegFile 数组，将其中的每个元素赋值为 0。always 块用于读取寄存器数据。当 readReg1、readReg2 或 writeReg 变量发生变化时，always 块将读取 RegFile 数组中对应的寄存器数据，并将其分别存储到 ReadData1、ReadData2 和 WriteData 变量中。

模块的另一个 always 块用于写入数据到寄存器中。当 regWrite 变量为 1 且时钟下降沿到来时，always 块将 WriteData 中的数据写入到 writeReg 指定的寄存器中。最后，模块的 assign 语句用于输出从寄存器 1 和寄存器 2 中读取的数据，将其分别赋值给 readData1 和 readData2 输出端口。

代码如下：

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

```
    );

    reg [31:0] RegFile[31:0];
    reg [31:0] ReadData1;
    reg [31:0] ReadData2;
    reg [31:0] WriteData;
    integer i;

    //init the RegFile
    initial begin
    for(i=0;i<=31;i=i+1)
       RegFile[i]=0;
    end

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

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

    assign readData1 = RegFile[readReg1];
    assign readData2 = RegFile[readReg2];

endmodule
```

3.2 DataMemory 模块

DataMemory 模块实现了一个 64 行 32 位的内存数组，内存地址范围为 0 到 63。在时钟信号（clk）下，如果 memWrite 被置为高电平，则 writeData 被写入到指定的内存地址；如果 memRead 被置为高电平，则从指定的内存地址读取数据，并输出到 readData 端口。在模块初始化时，所有的内存单元都被初始化为 0。如果地址超出了内存范围，将不会发生任何操作。

代码如下：

```
module dataMemory(
    input clk,
    input [31:0] address,
    input [31:0] writeData,
    input memWrite,
    input memRead,
    output [31:0] readData
    );

    reg [31:0] memFile[0:63];
    reg [31:0] ReadData;
    integer i;

    //init the memFile
    initial begin
    for(i=0;i<=31;i=i+1)
       memFile[i]=0;
    end

    always @ (address or memRead)
    begin
        if(address < 63)
        begin
            ReadData = memFile[address];
        end
        else ReadData = 0;
    end

    always @ (negedge clk)
    begin
        if(address < 63)
        begin
            memFile[address] = writeData;
        end
    end

    assign readData = ReadData;

endmodule
```

3.3 SignExt 模块

SignExt 模块将输入的 16 位指令进行符号扩展，即将指令的最高位复制到高 16 位中，使得扩展后的 32 位数据仍然具有相同的符号位。在时钟信号下，当输入端口 inst 发生变化时，Inst 变量被赋值为输入端口的值，接着判断输入的指令最高位是否为 1，如果为 1 则在高 16 位中填充 16 个 1，否则在高 16 位中填充 16 个 0。最后将扩展后的 32 位有符号数据输出到 data 端口。

代码如下：

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

    always @ (inst)
    begin
        Inst = inst;
        if(Inst[15] == 1'b1)
            Data = {16'hffff, Inst};
        else
            Data = {16'h0000, Inst};
    end
    assign data = Data;
endmodule
```

4 实例化和测试

4.1 Registers 测试

根据参考波形编写激励代码如下：

```
module Registers_tb(

    );
    reg clk;
    reg [25:21]ReadReg1;
    reg [20:16]ReadReg2;
    reg [4:0]WriteReg;
    reg [31:0]WriteData;
```

```
    reg RegWrite;
    wire [31:0]ReadData1;
    wire [31:0]ReadData2;

    Register u0(
        .readReg1(ReadReg1),
        .readReg2(ReadReg2),
        .writeReg(WriteReg),
        .writeData(WriteData),
        .regWrite(RegWrite),
        .clk(clk),
        .readData1(ReadData1),
        .readData2(ReadData2)
        );

    //turn over the clock every 100 ns
    always #100 clk=~clk;

    initial begin
        // Initialize Input
        clk = 1;
        ReadReg1 = 0;
        ReadReg2 = 0;
        WriteReg = 0;
        WriteData = 0;
        RegWrite = 0;

        // Current Time: 285 ns
        #285;
        RegWrite = 1;
        WriteReg = 5'b10101;
        WriteData = 32'hffff0000;

        // Current Time: 485 ns
        #200;
        WriteReg = 5'b01010;
        WriteData = 32'h0000ffff;

        #200;
        RegWrite = 1'b0;
        WriteReg = 5'b00000;
        WriteData = 32'h00000000;
```

```
51
52          // Current Time: 735 ns
53          #50;
54          ReadReg1 = 5'b10101;
55          ReadReg2 = 5'b01010;
56      end
57
58  endmodule
```

得到仿真结果

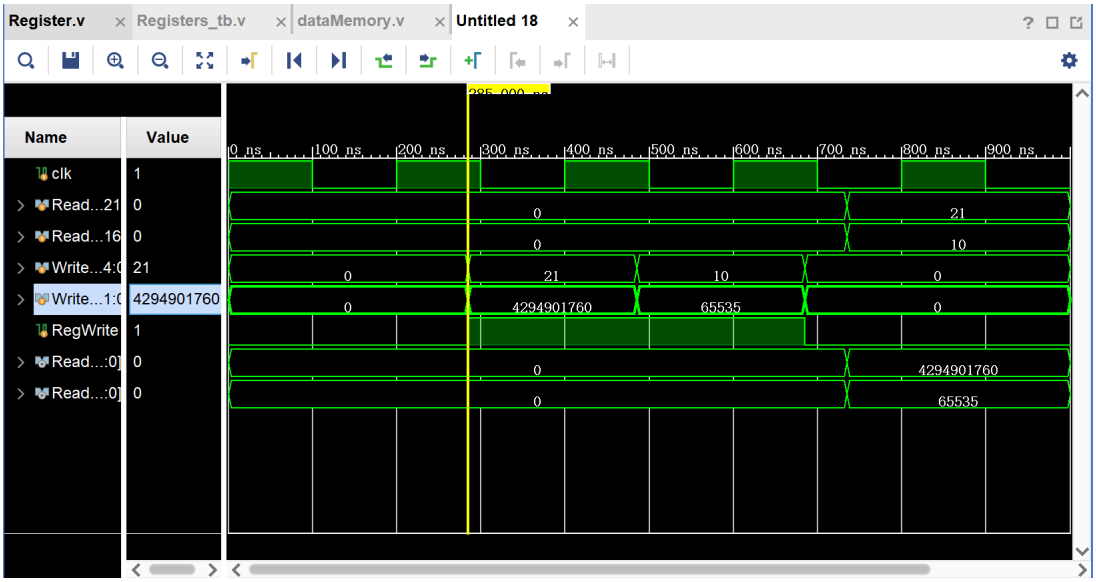

图 4: 寄存器仿真测试结果

与参考波形相同，证明模块运行正常。

4.2 DataMemory 测试

根据参考波形进行代码编写，激励代码如下：

```
1  module dataMemory_tb(
2
3      );
4      reg clk;
5      reg [31 : 0] address;
6      reg [31 : 0] writeData;
7      reg memWrite;
```

```
8       reg memRead;
9       wire [31 : 0] readData;
10
11      dataMemory u0(
12          .clk(clk),
13          .address(address),
14          .writeData(writeData),
15          .memWrite(memWrite),
16          .memRead(memRead),
17          .readData(readData)
18          );
19
20      always #100 clk = ~clk;
21
22      initial begin
23          // Initialize Inputs
24          clk = 0;
25          address = 0;
26          writeData = 0;
27          memWrite = 0;
28          memRead = 0;
29          // Current Time: 185 ns
30          #185;
31          memWrite = 1'b1;
32          address = 32'h00000007;// write data1
33          writeData = 32'h70000000;
34
35          // Current Time: 285 ns
36          #100;
37          memWrite = 1'b1;
38          writeData = 32'hffffffff;
39          address = 32'h00000006;
40
41          // Current Time: 470 ns
42          #185;
43          memRead = 1'b1;
44          memWrite = 0'b0;
45          address = 32'h00000007;// read data1
46
47          // Current Time: 550 ns
48          #80;
49          memWrite = 1;
```

```
50          address = 32'b00000008;// write data 2
51          writeData = 32'haaaaaaaa;
52
53          // Current Time: 630 ns
54          #80;
55          memWrite = 0;
56          memRead = 1;
57          address = 32'h00000006;// read data2
58      end
59  endmodule
```

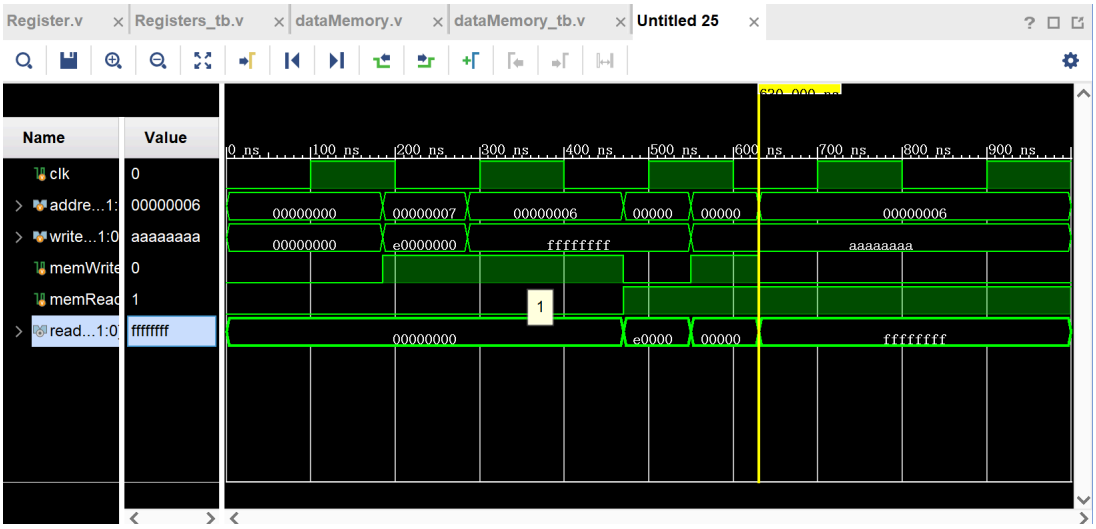

图 5: 存储器仿真测试结果

与参考波形相同，证明模块运行正常。

4.3 SignExt 测试

随机选择一系列数字进行扩展，编写激励代码如下：

```
1  module signext_tb(
2
3      );
4      reg [15 : 0] inst;
5      wire [31 : 0] data;
6
7      signext u0(
```

```
        .inst(inst),
        .data(data)
    );

    initial begin
        inst = 0;

        #100;
        inst = 16'h0001;

        #100;
        inst = 16'hffff;

        #100;
        inst = 16'h0002;

        #100;
        inst = 16'hfffe;

        #100;
        inst = 16'h8120;

        #100;
        inst = 16'h0251;
    end
endmodule
```

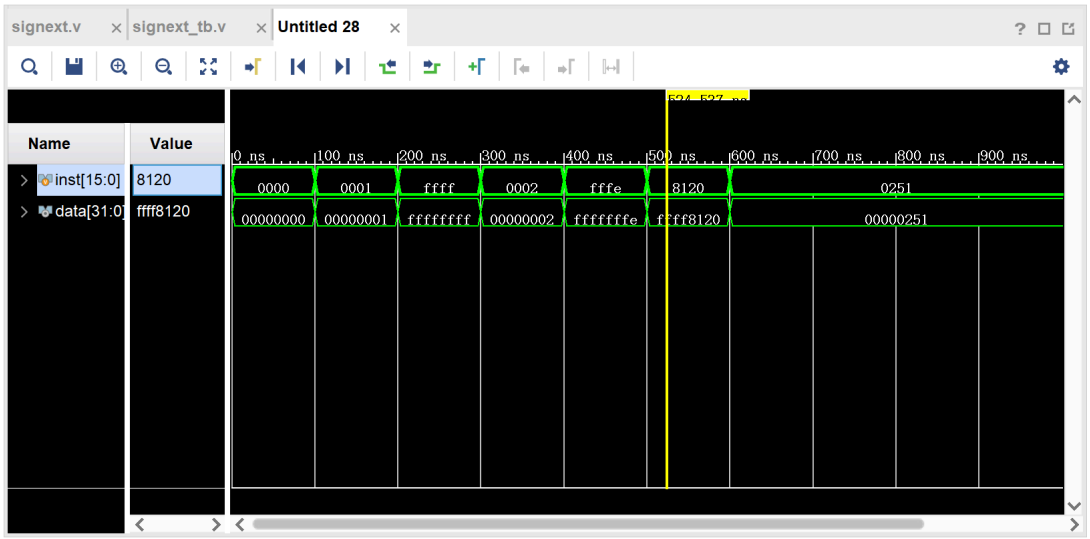

图 6: 符号拓展器仿真测试结果

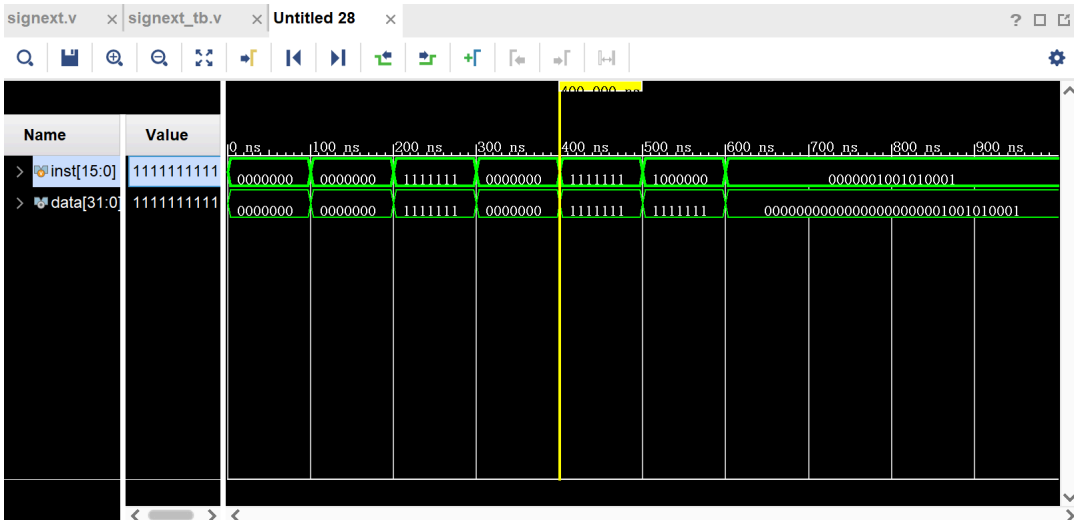

图 7: 符号拓展器仿真测试结果-二进制表示

正负数对应的拓展结果都正常，证明该模块正确。

5 总结与反思

实验 3 在学习理解寄存器、数据存储器、有符号扩展单元的 IO 定义和工作原理:

- 寄存器是用于存储和操作数据的主要组件之一。在 MIPS 架构中，有 32 个 32 位通用寄存器（General Purpose Registers，GPRs），它们用于存储整数数据。
- 数据存储器用于存储程序中使用的数据。在 MIPS 架构中，数据存储器也以字为单位进行寻址。
- 符号拓展器用于将有符号数的低位位数拓展到高位。在 MIPS 指令集架构中，符号扩展器通常用于执行算术和逻辑运算时，将 8 位或 16 位的有符号数扩展为 32 位有符号数。

在此基础上，设计了对应三个单元模块，随后对实现的各个模块进行仿真测试。为实验 5 和实验 6 的处理器及流水线搭建做准备工作，并对 Vertilog 和 Vivado 进行了更加深入的学习。

此外，我在学习 lab5 的设计后，对 signext 模块进行简化修改，并同样通过了仿真测试。

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

    assign data = (extSign ? {{16{inst[15]}}, inst[15:0]} : {16'h0000, inst[15:0]});

endmodule
```

在这里，条件由表达式”extSign” 表示，假设它是一个布尔变量或表达式。如果”extSign” 为真，则赋给”data” 的值是由 16 个”inst[15]” 组成的 16 位值与”inst” 的最低 16 位连接起来的结果。如果”extSign” 为假，则赋给”data” 的值是由 16 个零组成的 16 位值与”inst” 的最低 16 位连接起来的结果。

6 致谢

刘雨桐老师为实验提供了良好的讲解，在课程中时常做出对于任务难度和时间上的提醒，并十分体谅学生，能理解我们的困难并给予帮助，深表感谢。

实验过程中，助教蔡明昕、黄正翔老师多次解决我的困惑和实验中遇到的问题，并且在困难的时候给予我鼓励和帮助，对此深表感激。

感谢黄小平老师及实验室提供的资源和硬件支持。

感谢在实验过程中给予我帮助的同学。

邓倩妮老师为使得教学效果更好，将课程体系由 RISC-V 改为 MIPS，在实验中提供很多帮助，对此表示感谢。