lab05 report

郑航 520021911347

lab05 report
- 1 实验目的
- 2 实验原理
  - 2.1 Ctr原理分析
    - 2.1.1 模块描述
    - 2.1.2 输入的OpCode
    - 2.1.3 输出的控制信号
    - 2.1.4 OpCode和输出信号的对应关系
  - 2.2 ALUCtr原理分析
    - 2.2.1 模块描述
    - 2.2.2 输入的ALUOp和Funct和输出的ALUCtrOut的对应关系
  - 2.3 ALU原理分析
    - 2.3.1 模块描述
    - 2.3.2 ALUCtrOut与ALU运算方式的对应关系
  - 2.4 Register原理分析
  - 2.5 Data Memory原理分析
  - 2.6 Sign Extension原理分析
  - 2.7 Instruction Memory原理分析
  - 2.8 Mux原理分析
  - 2.9 RegMux原理分析
  - 2.10 PC原理分析
  - 2.11 Top原理分析
- 3 实验过程
  - 3.1 Ctr
  - 3.2 ALUCtr
  - 3.3 ALU
  - 3.4 Register
  - 3.5 Data Memory
  - 3.6 Sign Extension
  - 3.7 Instruction Memory
  - 3.8 Mux
  - 3.9 RegMux
  - 3.10 PC
  - 3.11 Top
- 4 实验结果
  - 4.1 数据和指令的编写
  - 4.2 激励文件编写
    - 4.1.2 仿真图像
- 5 反思总结
  - 5.1 difficulty
  - 5.2 summary
- 6 附录
  - 6.1 Ctr完整代码
  - 6.2 ALUCtr完整代码
  - 6.3 ALU完整代码
  - 6.4 Registers完整代码
  - 6.5 data memory完整代码

6.6 signext完整代码

1 实验目的

- 理解简单的类 MIPS 单周期处理器的工作原理（即几类基本指令执行时所需的数据通路和与之对应的控制线路及其各功能部件间的互联定义、 逻辑选择关系）
- 完成简单的类 MIPS 单周期处理器
  - 9 条 MIPS 指令(lw, sw, beq, add, sub, and, or, slt, j) CPU 的实现与调试
  - 拓展至 16 条指令(增加 addi, andi, ori, sll, srl, jal, jr) CPU 的实设计与实现
- 仿真测试

2 实验原理

2.1 Ctr原理分析

相比lab03里的Ctr，原理类似，只是增加了两个新的输出信号extSign和jalSign，并将ALUOp由两位拓展至三位（为了支持更多指令）

2.1.1 模块描述

- 功能：对指令的高6位OpCode 进行分析，对不同的指令和指令类型发出对应的一系列控制信号
- 输入：6位的OPCode（2.1.2详细说明）
- 输出：一系列控制信号（2.1.3详细说明）

2.1.2 输入的OpCode

不同的OpCode对应不同的指令类型和指令，对应关系如下表：

OpCode	指令
000000	R型
000010	j
000100	beq
100011	lw
101011	sw
001000	addi
001100	andi
001101	ori
000011	jal

2.1.3 输出的控制信号

Ctr是控制中心，需要根据输入指令，输出一系列控制信号指导各部件的功能执行，具体的信号和信号对应含义如下表：

输出信号	含义
ALUSrc	算术逻辑运算单元 (ALU) 的第二个操作数的来源 (0: 使用 rt； 1: 使用立即数)
ALUOp	ALU控制信号，发送给运算单元控制器 (ALUCtr) 用来进一步解析运算类型
Branch	条件跳转信号，高电平说明当前指令是条件跳转指令 (branch)
extSign	符号扩展信号，高电平说明当前指令需要进行符号拓展
JalSign	jal 指令信号，高电平说明当前指令是 jal 指令
Jump	无条件跳转信号，高电平说明当前指令是无条件跳转指令 (jump)
memRead	内存读的enable信号，高电平说明当前指令需要进行内存读取 (load)
memToReg	写寄存器的数据来源 (0: 使用 ALU 运算结果； 1: 使用内存读取数据)
memWrite	内存写的enable信号，高电平说明当前指令需要进行内存写入 (store)
regDst	目标寄存器选择信号 (0: 写入 rt 代表的寄存器； 1: 写入 rd 代表的寄存器)
regWrite	寄存器写的enable信号，高电平说明当前指令需要进行寄存器写入

2.1.4 OpCode和输出信号的对应关系

输出信号	000000	000010	000100	100011	101011	001000	001100	001101	000011
ALUSrc	0	0	0	1	1	1	1	1	0
ALUOp	101	110	001	000	000	010	011	100	110
Branch	0	0	1	0	0	0	0	0	0
extSign	0	0	1	1	1	1	0	0	0
JalSign	0	0	0	0	0	0	0	0	1
Jump	0	1	0	0	0	0	0	0	1
memRead	0	0	0	1	0	0	0	0	0
memToReg	0	0	0	1	0	0	0	0	0
memWrite	0	0	0	0	1	0	0	0	0
regDst	1	0	0	0	0	0	0	0	0
regWrite	1	0	0	1	0	1	1	1	1

2.2 ALUCtr原理分析

相比lab03里的ALUCtr，原理类似，输入的ALUOp由两位拓展至三位（为了支持更多指令），ALUCtrOut依旧是四位但是有了更多的组合，可以支持让ALU完成更多的操作

2.2.1 模块描述

- 结合Ctr传入的ALUOp和指令的后6位Funct，综合分析后通过ALUCtrOut指导ALU执行具体的功能
- 输入：3位的ALUOp和6位的Funct
- 输出：4位的ALUCtrOut

2.2.2 输入的ALUOp和Funct和输出的ALUCtrOut的对应关系

ALUOp	Funct	ALUCtrOut	对应指令	ALU操作
101	100000	0010	add	加法
101	100010	0110	sub	减法
101	100100	0000	and	逻辑与
101	100101	0001	or	逻辑或
101	101010	0111	slt	小于时置位
000	xxxxxx	0010	lw	加法
000	xxxxxx	0010	sw	加法
001	xxxxxx	0110	beq	减法
110	xxxxxx	0010	j	不运算
010	xxxxxx	0010	addi	加法
011	xxxxxx	0000	andi	逻辑与
100	xxxxxx	0001	ori	逻辑或
101	000000	0011	sll	逻辑左移
101	000010	0100	srl	逻辑右移
101	001000	0101	jr	不运算
110	xxxxxx	0101	jal	不运算

2.3 ALU原理分析

相比lab03里的ALU，原理类似，输入的ALUCtrOut依旧是四位但是有了更多的组合，可以支持让ALU完成更多的操作（如逻辑左移和逻辑右移）

2.3.1 模块描述

- 对两个输入的数据，根据ALUCtrOut指定的运算方式对其进行运算，并输出运算结果，并根据结果是否为零将zero进行置位
- 输入：32位的inputA和inputB，4位的ALUCtrOut
- 输出：32位的aluRes，即计算结果；1位的zero，本质上是一个置位行为，若aluRes为0则将其置为1，否则为0

2.3.2 ALUCtrOut与ALU运算方式的对应关系

ALUCtrOut	ALU运算方式
0000	逻辑与
0001	逻辑或
0010	加法
0110	减法
0111	小于时置位
0101	无运算
0011	逻辑左移
0100	逻辑右移

2.4 Register原理分析

与lab04中的Register完全一致，不再赘述

2.5 Data Memory原理分析

与lab04中的Data Memory完全一致，不再赘述

2.6 Sign Extension原理分析

与lab04中的Sign Extension完全一致，不再赘述

2.7 Instruction Memory原理分析

- 功能：与Data Memory的原理基本一致，但不需要支持写操作，只需要支持读操作即可。在Instruction Memory中含一组1024个的32位存储器块，作为指令存储的位置
- 输入：32位的address，表示所要读的指令所在的位置
- 输出：32位的inst，即address对应的位置的32位指令
- 注意，由于PC每次增加4，即address每次增加4个字节，但是inst和instFile都是32位的，因此在取地址时需要将address除以4作为index

2.8 Mux原理分析

- 功能：作为选路器，根据一个输入的控制信号，选择两个输入通路的其中一个作为输出
- 输入：一个1位的控制信号select，两个32位的数据输入input0和input1
- 输出：一个32位的out，其值等于input0和input1中被选中的一项的值

2.9 RegMux原理分析

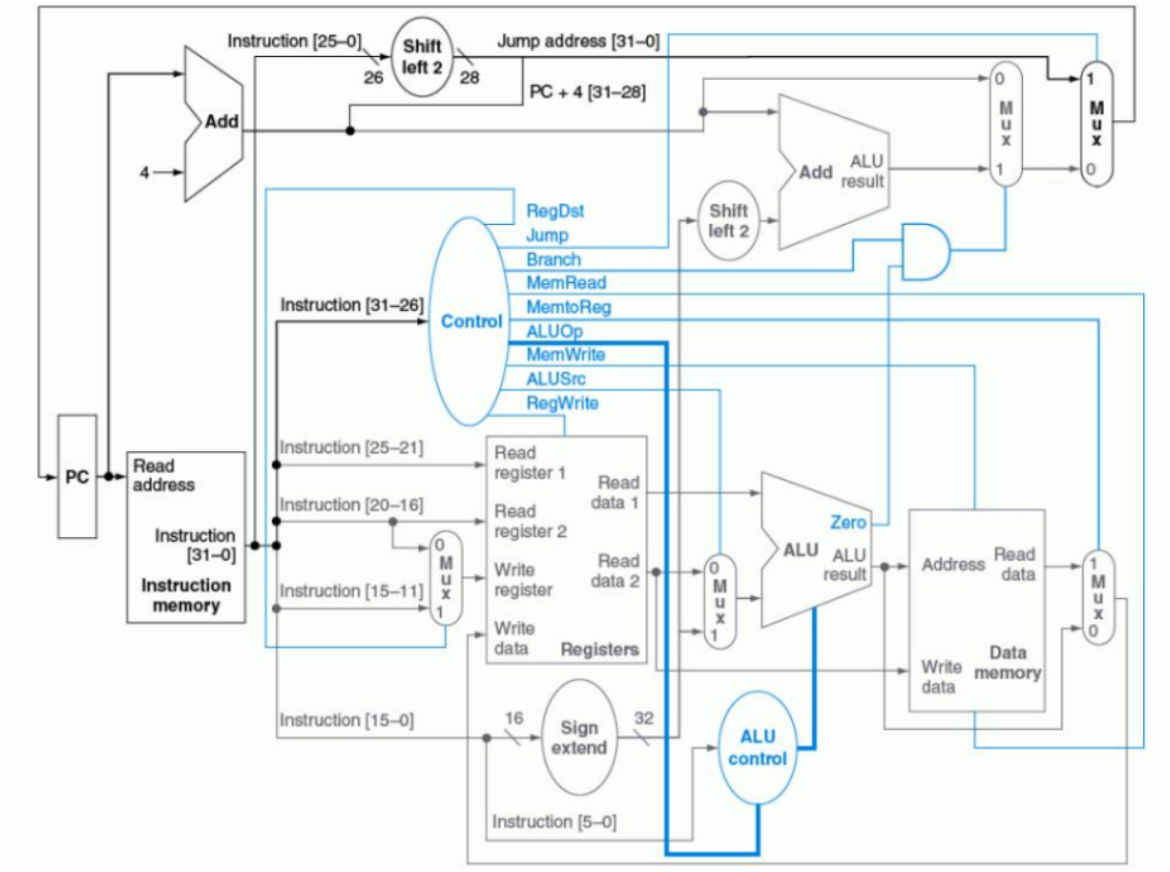
5位选路器，功能与Mux类似，只是输入和输出的数据都是5位的而不是32位的，主要用于在Register模块前进行Register的选择（Register编号为5位）

2.10 PC原理分析

- 功能：根据reset信号和clk信号更新PC，但不进行PC具体值的计算，计算交给外部的其他逻辑模块，PC内只进行PC的置零和维持原值
- input：1位的clk和1位的reset信号，32位的PCIn，其中clk用于同步，在clk的下降沿进行PC的更新操作，reset是重启信号，一旦reset为1就将PC置为0，否则PC维持原值PCIn
- output：32位的PCOut，作为PC的新值

2.11 Top原理分析

顶层模块的实现就是将上述所有模块实例化并有序组合到一起的过程，参考系统结构课内所学的处理器结构，结构如下：



3 实验过程

3.1 Ctr

Ctr的实现与lab03中的实现非常类似，主体也是一个case块，然后针对所有可能case对输出的控制信号进行赋值，具体的赋值对应关系在原理分析中已给出，注意做好default情况的处理

代码较长，此处不加展示，完整代码见附录

3.2 ALUCtr

ALUCtr的实现与lab03中的实现非常类似，主体也是一个casex块，然后针对所有可能case对ALUCtrOut进行赋值，具体的赋值对应关系在原理分析中已给出，注意做好default情况的处理

代码较长，此处不加展示，完整代码见附录

3.3 ALU

ALU的实现与lab03中的实现非常类似，主体也是一个case块，然后根据ALUCtrOut的不同值对inputA和inputB进行不同运算

代码较长，此处不加展示，完整代码见附录

3.4 Register

与lab04中的Register完全一致，不再赘述，完整代码见附录

3.5 Data Memory

与lab04中的Data Memory完全一致，不再赘述，完整代码见附录

3.6 Sign Extension

与lab04中的Sign Extension完全一致，不再赘述，完整代码见附录

3.7 Instruction Memory

由于不需要支持写的功能，因此实现较为简单。用一个数目为1024的32位的存储器组instFile来存储指令，每次根据输入的address右移2位（除以4）作为index，在instFile中取出对应指令输出即可

代码如下：

```
module InstMemory(
        input[31:0] address,
        output[31:0] inst
    );

    reg [31:0] instFile[0:1023];

    assign inst = instFile[address>>2];
endmodule
```

3.8 Mux

如原理分析所述，其功能是根据select信号选择两个输入中的一个进行输出，因此其实现只需要简单使用一个三目运算符即可

代码如下：

```
module Mux(
    input select,
    input [31:0] input0,
    input [31:0] input1,
    output [31:0] out
    );
    assign out = select?input0:input1;
endmodule
```

3.9 RegMux

与3.8的Mux非常类似，只是输入输出位数不同而已

代码如下：

```
module RegMux(
    input select,
    input [4:0] input0,
    input [4:0] input1,
    output [4:0] out
    );
    assign out = select?input0:input1;
endmodule
```

3.10 PC

实现主体是一个always块，一旦clk下降沿到来或reset信号发生改变即进行PC的重新赋值，赋值数值需要根据reset信号而定

代码如下：

```
module PC(
    input [31:0] pcIn,
    input clk,
    input reset,
    output [31:0] pcOut
    );

    reg [31:0] PC;

    initial PC = 0;

    always @ (posedge clk or reset)
    begin
        if(reset)
            PC = 0;
        else
            PC = pcIn;
    end
    assign pcOut = PC;
endmodule
```

3.11 Top

Top模块主要就是对各模块根据需求进行实例化，并用一系列数据线进行连接，我们将Ctr模块的实例化及其对应数据线展示如下：

```
    wire [31 : 0] INST;
    wire REG_DST;
    wire ALU_SRC;
    wire MEM_TO_REG;
    wire REG_WRITE;
    wire MEM_READ;
    wire MEM_WRITE;
    wire BRANCH;
    wire EXT_SIGN;
    wire JAL_SIGN;
    wire [2 : 0] ALU_OP;
    wire JUMP;
    Ctr main_ctr (
        .opCode(INST[31 : 26]),
        .regDst(REG_DST),
        .aluSrc(ALU_SRC),
        .memToReg(MEM_TO_REG),
        .regWrite(REG_WRITE),
        .memRead(MEM_READ),
        .memWrite(MEM_WRITE),
        .branch(BRANCH),
        .aluOp(ALU_OP),
        .jump(JUMP),
        .extSign(EXT_SIGN),
        .jalSign(JAL_SIGN)
    );
```

其他模块的实例化也类似，代码较长此处不进行完全展示，完整代码详见附录1

4 实验结果

4.1 数据和指令的编写

编写初始数据如下：

地址	数据	地址	数据	地址	数据	地址	数据
0	00000001	8	00001111	16	000000FF	24	00000024
1	00000010	9	00000111	17	00000100	25	00000025
2	00000100	10	00000100	18	00001100	26	00000026
3	00001000	11	00001000	19	00000100	27	00000027
4	00010000	12	00000010	20	00000010	28	00000028
5	00100000	13	00000000	21	00000001	29	00000029
6	01000000	14	0000000F	22	0000000F	30	00000030
7	10000000	15	00000FFF	23	000000FF	31	00000031

设计初始指令如下（含指令含义和指令执行结果）：

指令地址	指令（二进制）	指令	指令结果
0	100011 00000 00001 0000000000000000	lw $1, 0($0)	$1 = Mem[0] = 1
1	100011 00001 00010 0000000000000000	lw $2, 0($1)	$2 = Mem[1] = 16
2	100011 00000 00011 0000000000000010	lw $3, 2($0)	$3 = Mem[2] = 256
3	100011 00001 00100 0000000000001111	lw $4, 15($1)	$4 = Mem[16] = 255
4	000000 00001 00011 00101 00000 100000	add $5, $1, $3	$5 = 1 + 256 = 257
5	000000 00011 00100 00110 00000 100010	sub $6, $3, $4	$6 = 256 –255 = 1
6	000000 00101 00110 00111 00000 100100	and $7, $5, $6	$7 = 257 & 1 = 1
7	100011 00000 01001 0000000000001001	lw $9, 9($0)	$9 = Mem[9] = 273
8	100011 00000 01010 0000000000001010	lw $10, 10($0)	$10 = Mem[10] = 256
9	000000 01001 01010 01000 00000 100101	or $8, $9, $10	$8 = 273 |256 = 273
10	001000 01010 01011 0000000100000000	addi $11, $10, 256	$11 = 256 + 256 = 512
11	100011 00000 01100 0000000000001100	lw $12, 12($0)	$12 = Mem[12] = 16
12	001100 01100 01101 0000000011111111	andi $13, $12, 255	$13 = 16 & 255 = 16
13	100011 00000 10000 0000000000010000	lw $16, 16($0)	$16 = Mem[16] = 255
14	001101 10000 01111 0000000100000000	ori $15, $16, 256	$15 = 255 |256 = 511
15	101011 01110 01111 0000000000000111	sw $15, 7($14)	Mem[7] = 511
16	000000 00000 00110 10100 00100 000000	sll $20, $6, 4	$20 = $6 << 4 = 16
17	000000 00000 01010 10110 00100 000010	srl $22, $10, 4	$22 = $10 >> 4 = 16

指令地址	指令（二进制）	指令	指令结果
18	000000 00001 00011 11000 00000 101010	slt $24, $1, $3	$24 = 1
19	000000 01111 10000 11001 00000 101010	slt $25, $15, $16	$25 = 0
22	000100 10100 10110 0000000000000001	beq $20, $22, 1	go to line 22;
23	000000 01111 10000 10010 00000 101010	slt $18, $15, $16	(skipped)
24	000010 00000000000000000000011000	j 24	go to line 24
25	001000 01011 01011 0000000000000010	addi $11, $11, 2	(skipped)
26	000011 00000000000000000000011010	jal 26	go to line 26
27	000100 00101 00110 0000000000000010	beq $5, $6, 2	go to line 2
28	000000 11111 00000 00000 00000 001000	jr $31	go to line 25
29	001000 01011 01011 0000000000000010	addi $11, $11, 2	(skipped)
30	001000 01011 01011 0000000000000010	addi $11, $11, 2	$11 = $11 + 2 = 514

4.2 激励文件编写

激励文件中，主要分三个部分：

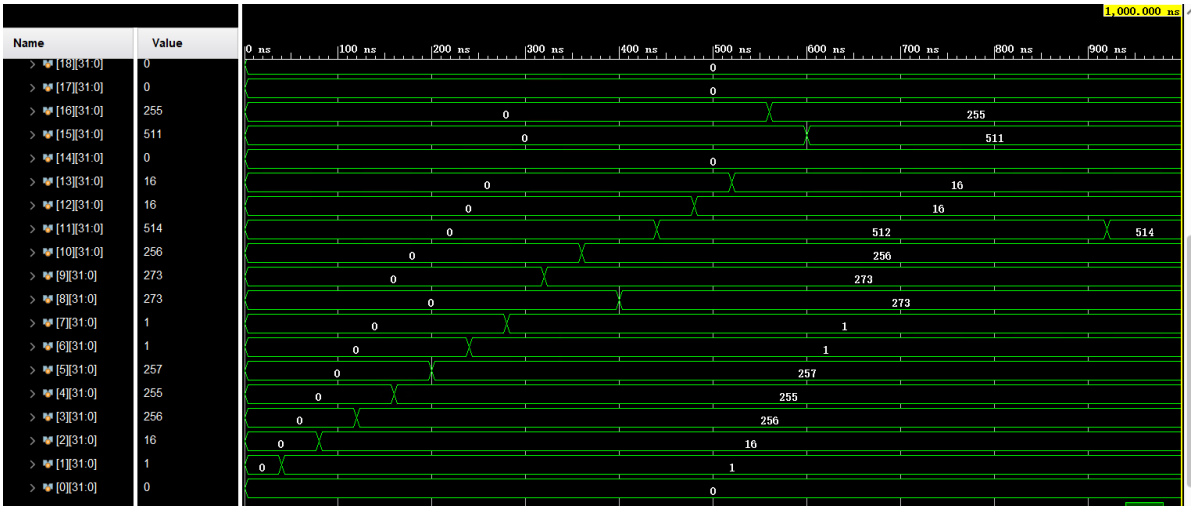
- 实例化Top顶层模块
- 初始化，包括初始化reset和clk信号，利用readmemb和readmemh函数读取外部数据和指令文件，对instFile和memFile进行初始化
- 设置时钟周期

测试代码如下：

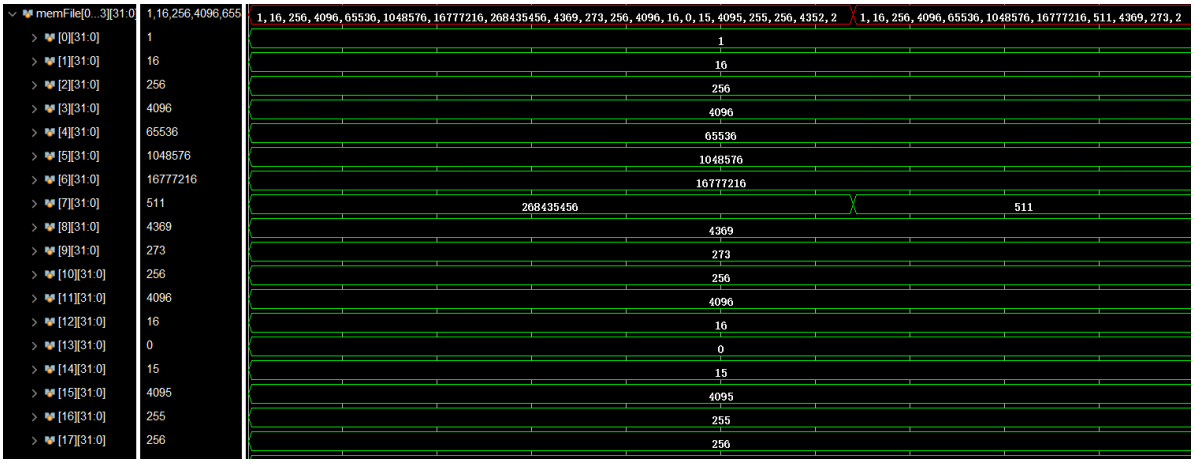
```
module top_tb(

    );
    reg clk;
    reg reset;

    top TOP(
        .clk(clk),
        .reset(reset)
    );
```

```
11
12      initial begin
13          reset = 1;
14          clk = 0;
15
        $readmemb("C:/Archlabs/lab05/mem_inst.txt",TOP.inst_memory.instFile);
16          $readmemh("C:/Archlabs/lab05/mem_data.txt",TOP.memory.memFile);
17
18          #10 reset = 0;
19      end
20
21      always #20 clk = ~clk;
22  endmodule
```

4.1.2 仿真图像



由图像可知，各寄存器值都符合我们预期



由图像可知，511的值顺利写入了MemFile中的对应位置，sw指令也成功实现

5 反思总结

5.1 difficulty

- 本实验的难度比较大，尤其是需要首先理清楚各个部件的功能和关系，相比前两个实验多了很多内容，所以一开始找不到合适的切入点，最终也依靠反复阅读实验指导书和结合系统结构课内所学的数据通路等的知识，完成了本实验。
- 本次实验中，由于对vivado不是很熟悉，调试时花费了很多时间，无论是连线的问题还是指令的问题都需要调试很久，最终是在各处增加了一些输出等等才加快了调试进度。此外，由于连线较为复杂，因此提前绘制一下电路图加以分析会比脑海中空想整个过程要好得多，这也帮助加快了我的实验进展
- 实验中，自己编写实验数据和一系列的32位二进制实验指令，模拟指令运算结果等花费了接近一天的时间，过程比较繁琐，也很容易出问题，比较考验耐心

5.2 summary

本实验在lab03和lab04实现的各模块以及几个新模块的基础上，实现了一个完整的支持 16 指令的单周期 MIPS 处理器，且能成功运行。通过本次实验，我对整个MIPS的处理器及其数据通路和运行逻辑都有了深刻、具体而微的认识，收获巨大。此外，通过本次实验，自主设计指令和数据的过程使得我对于MIPS的十六条相关指令也有了更加深刻的理解，并且培养了我耐心细致的实验素养

付出与收获成正比，本实验虽然难度较大，但也着实令我受益匪浅

6 附录

6.1 Ctr完整代码

```
module Ctr(
    input [5:0] opCode,
    output regDst,
    output aluSrc,
    output memToReg,
    output regWrite,
    output memRead,
```

```
    output memWrite,
    output branch,
    output [2:0] aluOp,
    output jump,
    output extSign,
    output jalSign
    );

    reg RegDst;
    reg ALUSrc;
    reg MemToReg;
    reg RegWrite;
    reg MemRead;
    reg MemWrite;
    reg Branch;
    reg [2:0] ALUOp;
    reg Jump;
    reg ExtSign;
    reg JalSign;

    always @(opCode)
    begin
        case(opCode)
        6'b000000: //R type
        begin
            RegDst = 1;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b101;
            Jump = 0;
            ExtSign=0;
            JalSign=0;
        end
        6'b100011: //lw
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 1;
            RegWrite = 1;
            MemRead = 1;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b000;
            Jump = 0;
            ExtSign=1;
            JalSign=0;
        end
        6'b101011: //sw
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
```

```
            MemWrite = 1;
            Branch = 0;
            ALUOp = 3'b000;
            Jump = 0;
            ExtSign=1;
            JalSign=0;
        end
        6'b000100: //beq
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 1;
            ALUOp = 3'b001;
            Jump = 0;
            ExtSign=1;
            JalSign=0;
        end
        6'b001000:      // addi
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b010;
            Jump = 0;
            ExtSign=1;
            JalSign=0;
        end
        6'b001100:      // andi
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b011;
            Jump = 0;
            ExtSign=0;
            JalSign=0;
        end
        6'b001101:      // ori
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
```

```
            ALUOp = 3'b100;
            Jump = 0;
            ExtSign=0;
            JalSign=0;
        end
        6'b000010: //jump
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b110;
            Jump = 1;
            ExtSign=0;
            JalSign=0;
        end
        6'b000011:      // jal
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b110;
            Jump = 1;
            ExtSign=0;
            JalSign=1;
        end
        default:
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 3'b111;
            Jump = 0;
            ExtSign=0;
            JalSign=0;
        end
        endcase
    end

    assign regDst = RegDst;
    assign aluSrc = ALUSrc;
    assign memToReg = MemToReg;
    assign regWrite = RegWrite;
    assign memRead = MemRead;
    assign memWrite = MemWrite;
    assign branch = Branch;
    assign aluOp = ALUOp;
```

```
    assign jump = Jump;
    assign extSign=ExtSign;
    assign jalSign=JalSign;

endmodule
```

6.2 ALUCtr完整代码

```
module ALUCtr(
    input [2 : 0] aluOp,
    input [5 : 0] funct,
    output [3 : 0] aluCtrOut,
    output shamtSign,
    output jrSign
    );

    reg [3 : 0] ALUCtrOut;
    reg ShamtSign;
    reg JrSign;

    always @ (aluOp or funct)
    begin
        ShamtSign=0;
        JrSign=0;
        casex ({aluOp, funct})
            9'b000xxxxxx:  // lw or sw: actually add
                ALUCtrOut = 4'b0010;
            9'b001xxxxxx:  // beq: actually sub
                ALUCtrOut = 4'b0110;
            9'b010xxxxxx:  // addi: actually add
                ALUCtrOut = 4'b0010;
            9'b011xxxxxx:  // andi: acutally and
                ALUCtrOut = 4'b0000;
            9'b100xxxxxx:  // ori: acutally or
                ALUCtrOut = 4'b0001;
            9'b101000000:  // sll: actually left-shift
            begin
                ALUCtrOut = 4'b0011;
                ShamtSign = 1;
            end
            9'b101000010:  // srl: actuall right-shift
            begin
                ALUCtrOut = 4'b0100;
                ShamtSign = 1;
            end
            9'b101001000:  // jr: actually not change
            begin
                ALUCtrOut = 4'b0101;
                JrSign=1;
            end
            9'b101100000:  // add: actually add
                ALUCtrOut = 4'b0010;
            9'b101100010:  // sub: actually sub
                ALUCtrOut = 4'b0110;
            9'b101100100:  // and: actually and
```

```
                ALUCtrOut = 4'b0000;
            9'b101100101:  // or: actually or
                ALUCtrOut = 4'b0001;
            9'b101101010:  // slt: actually set on less than
                ALUCtrOut = 4'b0111;
            9'b110xxxxxx:  // jump / jal: actually not change
                ALUCtrOut = 4'b0101;
        endcase
    end

    assign aluCtrOut = ALUCtrOut;
    assign shamtSign=ShamtSign;
    assign jrSign=JrSign;

endmodule
```

6.3 ALU完整代码

```
module ALU(
    input [31 : 0] inputA,
    input [31 : 0] inputB,
    input [3 : 0] aluCtrOut,
    output zero,
    output [31 : 0] aluRes
    );

    reg Zero;
    reg [31 : 0] ALURes;

    always @ (inputA or inputB or aluCtrOut)
    begin
        case (aluCtrOut)
            4'b0000:    // and
                ALURes = inputA & inputB;
            4'b0001:    // or
                ALURes = inputA | inputB;
            4'b0010:    // add
                ALURes = inputA + inputB;
            4'b0110:    // sub
                ALURes = inputA - inputB;
            4'b0111:    // set on less than
                ALURes = ($signed(inputA) < $signed(inputB));
            4'b1100:    // nor
                ALURes = ~(inputA | inputB);
            4'b0011:    // left-shift
                ALURes = inputB << inputA;
            4'b0100:    // right-shift
                ALURes = inputB >> inputA;
            4'b0101:    // not change
                ALURes = inputA;
            default:
                ALURes = 0;
        endcase
        if (ALURes == 0)
            Zero = 1;
```

```
        else
            Zero = 0;
    end

    assign zero = Zero;
    assign aluRes = ALURes;
endmodule
```

6.4 Registers完整代码

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

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

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

    always @ (negedge clk or reset)
    begin
        if(reset)
        begin
            for(i=0;i<32;i=i+1)
                regFile[i] = 0;
        end
        else begin
            if(regWrite)
                regFile[writeReg] = writeData;
        end
    end
endmodule
```

6.5 data memory完整代码

```
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 : 1023];
    reg [31 : 0] ReadData;

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

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

    assign readData=ReadData;
endmodule
```

6.6 signext完整代码

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

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