计算机系统结构实验 Lab5实验报告

- 姓名：夏鸿驰
- 学号：520021910965

- 日期：2022年4月26日

目录

1. 实验概述

1.1 实验名称

- 简单的类 MIPS 单周期处理器的实现 – 整体调试

1.2 实验目的

1. 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部件间的互联定义、逻辑选择关系)
2. 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**设计**、**实现**与**调试**

2. 实验步骤

2.1 构建调试MIPS-9指令CPU

2.2.1 构建各单元模块文件

- 首先将lab3,lab4中构建成功的ALU,ALUCtr,Ctr,DataMemory,Registers,SignExt加载进入新创建的lab5工程文件。
- 创建新模块文件InstrMemory.v,作为指令内存使用，输入以下代码：

```
module InstrMemory(
    input [31:0] Address,
    output [31:0] ReadInstr
    );
    reg [31:0] MemFile[0:31];
    reg [31:0] rd;
    always @ (Address)
```

```
        begin
           rd = MemFile[Address>>2];
        end

    assign ReadInstr = rd;

endmodule
```

其中MemFile即为指令内存本体。
注意到MIPS指令均为等长的4字节，所以用32bit连续储存，并且在用PC访问的时候自动对齐字节：*Address>>2*。

- 创建新模块文件PC.v，作为PC使用,输入以下代码:

```
module PC(
    input [31:0] NextPC,
    input Clk,
    input reset,
    output [31:0] CurPC
    );
    reg [31:0] PCReg;


    always @ (posedge Clk)
        begin
            PCReg = NextPC;
        end
    assign CurPC = PCReg;

    always @ (posedge reset)
        begin
            PCReg = 0;
        end
endmodule
```

其中NextPC是下个时钟周期更新的PC,Clk为时钟信号,reset为重置信号用于初始化。
输出CurPC即为当前指令对应的PC值。

- 至此，所有在MIPS-9指令实现中所需的模块文件全部构建完成。

2.2.2 各模块文件的实例化与连接组装

- 首先，先创建Top.v用于作为顶层模块文件
  定义其输入输出端口如下:

```
module Top(
    input reset,
    input Clk
    );
```

reset用于初始化信号，Clk为CPU总时钟信号。

- 其次，先定义模块实例化所需的信号线，再实例化各个模块
  - PC模块：

```
//PC
wire [31:0] NextPC;
wire [31:0] CurPC;
PC PC(
    .NextPC(NextPC),
    .Clk(Clk),
    .reset(reset),
    .CurPC(CurPC)
);
```

定义NextPC和CurPC两条wire即可。
加入Clk,reset即可实例化PC。

- InstrMemory模块:

```
//InstrMemory
wire [31:0] InstrAddress;
wire [31:0] Instr;
InstrMemory InstrMemory(
    .Address(InstrAddress),
    .ReadInstr(Instr)
);
```

定义变量名称即为其用途，不多解释

- Ctr主控制模块:

```
//Ctr
wire RegDst;
wire ALUSrc;
wire MemToReg;
wire RegWrite;
wire MemRead;
wire MemWrite;
wire Branch;
wire [1:0] ALUOp;
wire Jump;
Ctr Ctr(
    .OpCode(Instr[31:26]),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemToReg(MemToReg),
    .RegWrite(RegWrite),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .Branch(Branch),
    .ALUOp(ALUOp),
    .Jump(Jump)
);
```

定义各个输出的控制信号线，变量名称即为其用途。
利用在InstrMemory取得的Instr的slice[31:26]来直接作为Opcode实例化Ctr。

- Registers寄存器文件模块:

```
//Registers
```

```
wire [25:21] readReg1;
wire [20:16] readReg2;
wire [4:0] writeReg;
wire [31:0] writeData;
wire [31:0] readData1;
wire [31:0] readData2;
Registers Registers(
    .Clk(Clk),
    .readReg1(readReg1),
    .readReg2(readReg2),
    .writeReg(writeReg),
    .writeData(writeData),
    .regWrite(RegWrite),
    .reset(reset),
    .readData1(readData1),
    .readData2(readData2)
);
```

加入时钟信号Clk和reset信号，并定义各读写端口及控制信号（定义类型意义同lab4中一致，不再赘述）。
加入实例化即可。

- SignExt符号拓展模块:

```
//SignExt
wire [31:0] SignExtImm;
SignExt SignExt(
    .Instr(Instr[15:0]),
    .Data(SignExtImm)
);
```

符号拓展I型指令的Instr[15:0]，故直接使用其作为实例化即可。
输出为SignExtImm。

- ALUCtr模块:

```
//ALUCtr
wire [3:0] aluCtr;
ALUCtr ALUCtr(
    .ALUOp(ALUOp),
    .Funct(Instr[5:0]),
    .ALUCtr(aluCtr)
);
```

ALUCtr利用Funct位Instr[5:0]和之前得到的ALUOp进行二阶译码控制，加入实例化即可。

- ALU模块:

```
//ALU
wire [31:0] input1;
wire [31:0] input2;
wire zero;
wire [31:0] ALURes;
ALU ALU(
    .input1(input1),
    .input2(input2),
    .ALUCtr(aluCtr),
    .zero(zero),
    .ALURes(ALURes)
);
```

实例化所需的aluCtr由ALUCtr输出，其他自己定义信号线，再实例化即可。

- DataMemory模块:

```
//DataMemory
wire [31:0] DataAddress;
wire [31:0] WriteData;
wire [31:0] ReadData;
DataMemory DataMemory(
    .Clk(Clk),
    .Address(DataAddress),
    .WriteData(WriteData),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .ReadData(ReadData)
);
```

利用产生的控制信号MemWrite，MemRead和时钟信号Clk并自定义其他剩余所需信号线即可。

- **至此，已经完成了所有模块的实例化，注意到模块实例化input所用的其他模块的output对应的wire其实就是相当于连接了两个模块的组合逻辑，但现在还剩余许多组合逻辑信号尚未连接设计，所以继续进行第三步。**

- 最后，按照组合逻辑发生的时间顺序和功能不同进行完整的连接。
  - IF阶段:

```
//IF
assign InstrAddress = CurPC;
//IF-PC
wire [31:0] NextSeqPC;
assign NextSeqPC = CurPC + 4;
```

    - 将CurPC与InstrAddress连接，即用当前PC值取指令。
    - 计算下一个顺序执行的指令地址NextSeqPC。
  - ID阶段:

```
//ID
assign readReg1 = Instr[25:21];
assign readReg2 = Instr[20:16];
//ID-PC
wire [31:0] JumpAddr;
assign JumpAddr[27:0] = Instr[25:0] << 2;
assign JumpAddr[31:28] = NextSeqPC[31:28];
wire [31:0] BranchAddr;
assign BranchAddr = NextSeqPC + (SignExtImm << 2);
```

- 先不考虑具体指令类型，做统一的工作，再用MUX选择即是。
- 分别将Instr[25:21](rs位)和Instr[20:16](rt位)与寄存器文件的readReg1和readReg2进行连接，即读取rs和rt。
- 计算JumpAddr：将Instr[25:0] << 2与NextSeqPC[31:28]连接即可。
- 计算BranchAddr：将NextSeqPC(PC+4)和SignExtImm << 2相加即可。

- EX阶段：

```
//EX
assign input1 = readData1;
assign input2 = ALUSrc? SignExtImm:readData2;
wire [31:0] PreNextPC;
assign PreNextPC = (Branch&&zero)? BranchAddr:NextSeqPC;
assign NextPC = Jump? JumpAddr:PreNextPC;
```

- ALU不论指令如何均用readData1(rs)作为input1，连接即可。
- 根据ALUSrc选择SignExtImm或readData2(rt)作为input2, 简单的实现MUX选择即可。
- 根据ALU计算结果zero和Ctr译码结果Branch决定是否选择分支（条件：Branch&&zero），简单的实现MUX选择即可。
- 根据Ctr译码结果Jump决定是否进行无条件跳转，简单的实现MUX选择即可。

- MEM阶段：

```
//MEM
assign DataAddress = ALURes;
assign WriteData = readData2;
```

- 将DataAddress与ALURes相连，并由MemRead决定是否读数据内存地址DataAddress。
- 将readData2与WriteData相连，并由MemWrite决定是否写修改数据内存地址DataAddress。

- WB阶段：

```
//WB
assign writeData = MemToReg? ReadData:ALURes;
assign writeReg = RegDst? Instr[15:11]:Instr[20:16];
```

- 根据Ctr产生的信号MemToReg选择写回内存读出的数据ReadData还是ALU计算结果ALURes。
- 根据Ctr产生的信号RegDst选择写回rt还是rd。
- 当然前提是Ctr产生的信号RegWrite=1才会启动寄存器写。

- **至此，已经完成了MIPS-9指令的所有设计实现工作，接下来仿真验证即可。**

2.2.3 MIPS-9指令CPU的仿真验证

- 首先，先设计测试集，覆盖9条指令，如下：
  - 汇编代码：

```
lw $1,12($0) //测试lw,从地址12出读数据进寄存器$1
lw $2,17($0) //测试lw,从地址17出读数据进寄存器$2
add $3,$1,$2 //测试add
sub $4,$1,$3 //测试sub
and $5,$2,$3 //测试and
or $6,$4,$5  //测试or
slt $7,$1,$2 //测试slt
slt $8,$2,$1
beq $1,$2,1  //测试beq 此处应不跳转 导致$9被赋值
add $9,$0,$1
beq $1,$1,1  //测试beq 此处应跳转 导致$10被复制1次，为$1的值
add $10,$0,$2
add $10,$0,$1
sw $5,-4($2) //测试sw及SignExt功能是否正常 若正常则M[0]被赋值$1
j 19         //测试j 若成功则$11未被更新
or $11,$0,$1
or $11,$0,$1
or $11,$0,$1
or $11,$0,$1
or $12,$0,$1
```

  - 对应的MIPS机器代码:

```
10001100000000010000000000001100
10001100000000010000000000010001
00000000001000100001100000100000
00000000001000110010000000100010
00000000010000110010100000100100
00000000010000101001100000100101
00000000001000100011100000101010
00000000010000101010000000101010
00010000001000100000000000000001
00000000000000001010010000100000
00010000001000010000000000000001
00000000000000001001010000100000
00000000000000001010100000100000
10101100010001011111111111111100
00001000000000000000000000010011
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001011000000100101
```

  - 设计的数据内存:

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

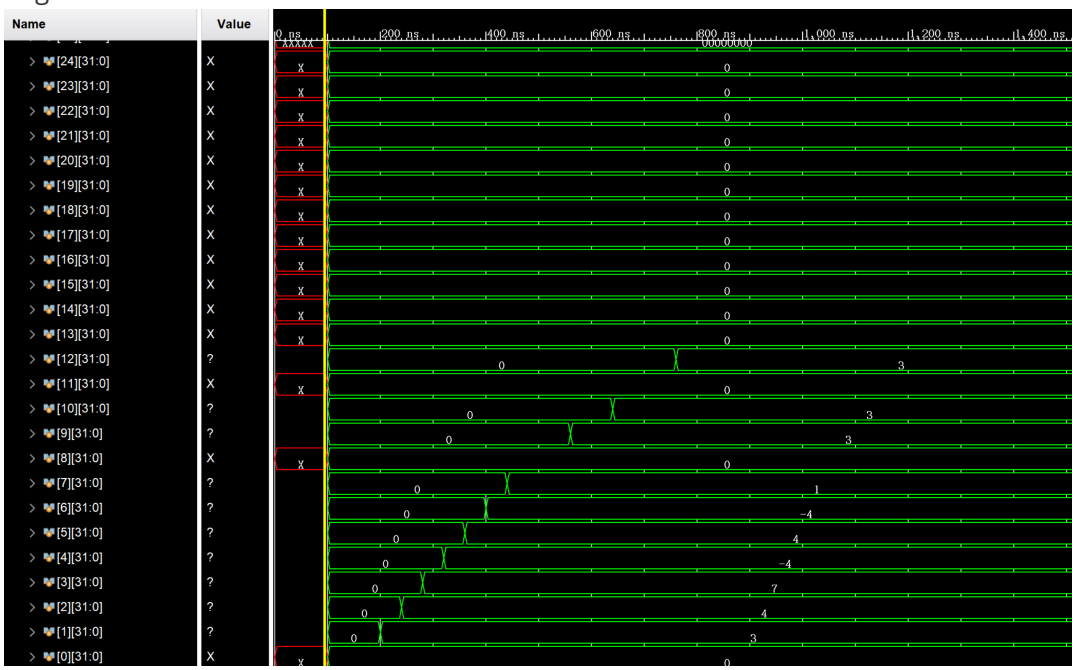
```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

- 其次，创建激励文件如下：

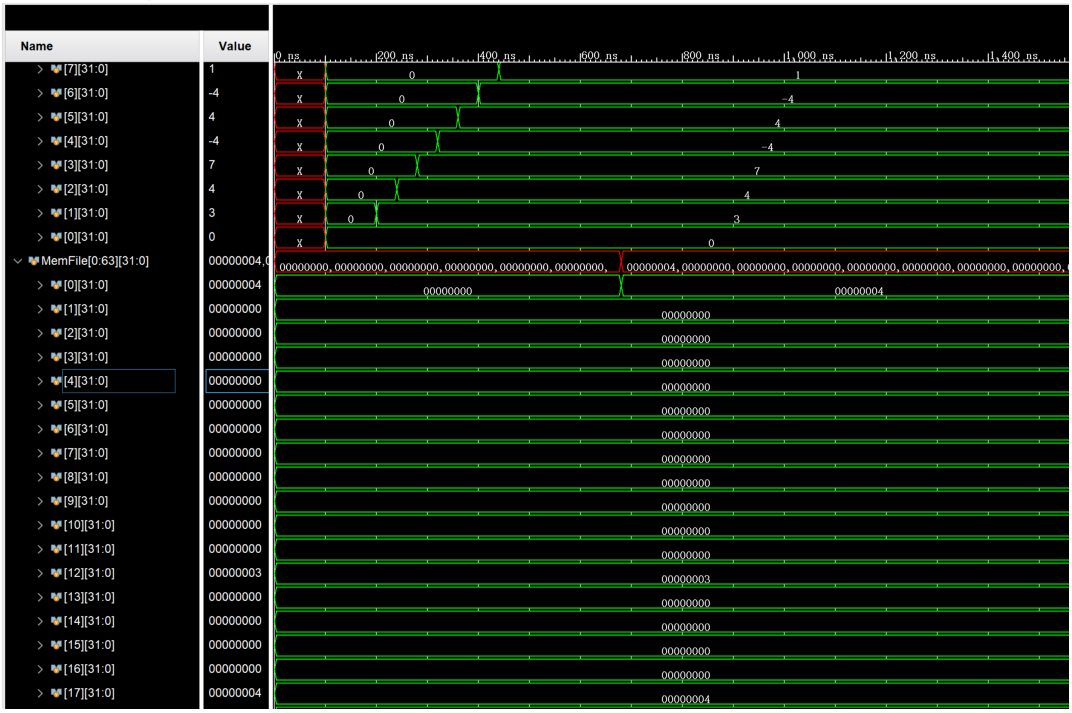
```
module Top_tb(

    );
    reg Clk;
    reg reset;
    Top Top(
        .Clk(Clk),
        .reset(reset)
    );

    always #20 if($time>=200) Clk = !Clk;
    initial begin
        Clk = 1;
        reset = 0;
        $readmemb("Instruction.txt",Top.InstrMemory.MemFile);
        $readmemb("Data.txt",Top.DataMemory.MemFile);
        #100;
        reset = 1;
        #80;
        reset = 0;
        #20;
    end
endmodule
```

- 首先实例化Top
- 再设置时钟周期为40ns
- 用前200ns进行reset初始化及读取指令内存和数据内存的数据
- 开始运行仿真
- 最后，得到Registers和DataMemory波形如下:

- Registers:



- DataMemory:



- 不难验证，得到的结果与预期一致, MIPS-9指令CPU构建成功!

2.2 构建调试MIPS-16指令CPU

2.2.1 创建新模块文件，修改现有模块文件

- 为了实现MIPS-16指令集，需要添加ZeroExt.v，作为零拓展模块，具体代码如下:

```
module ZeroExt(
    input [15:0] Instr,
    output [31:0] Data
    );
    assign Data[31:0] = Instr[15:0];
endmodule
```

- MIPS-16需要更加强大的译码能力，需要修改Ctr,ALUCtr部分信号及相关实现，如下:

- Ctr.v文件更新:

```
module Ctr(
    input [5:0] OpCode,
    output [1:0] RegDst, //new 00-rt 01-rd 10-31(for jal)
    output [1:0] ALUSrc1, //new 00-rs 01-rt 10-pc
    output [1:0] ALUSrc2, //new 00-rt 01-signext 10-zeroext 11-8
    output MemToReg,
    output RegWrite,
    output MemRead,
    output MemWrite,
    output Branch,
    output [2:0] ALUOp, //new 000-add 001-sub 010-Rtype 011-and
100-or
    output Jump
    );

    reg [1:0] regDst;
    reg [1:0] aluSrc1;
    reg [1:0] aluSrc2;
    reg memtoReg;
    reg regWrite;
    reg memRead;
    reg memWrite;
    reg branch;
    reg [2:0] aluOp;
    reg jump;

    always @ (OpCode)
    begin
        case(OpCode)
            6'b000000: //R type
            begin
                regDst = 2'b01;
                aluSrc1 = 2'b00;
                aluSrc2 = 2'b00;
                memtoReg = 0;
                regWrite = 1;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluOp = 3'b010;
                jump = 0;
            end

            6'b100011: //lw
            begin
                regDst = 2'b00;
                aluSrc1 = 2'b00;
                aluSrc2 = 2'b01;
                memtoReg = 1;
                regWrite = 1;
                memRead = 1;
                memWrite = 0;
                branch = 0;
                aluOp = 3'b000;
                jump = 0;
```

```
            end

            6'b101011: //sw
            begin
                regDst = 2'b00;
                aluSrc1 = 2'b00;
                aluSrc2 = 2'b01;
                memtoReg = 0;
                regWrite = 0;
                memRead = 0;
                memWrite = 1;
                branch = 0;
                aluOp = 3'b000;
                jump = 0;
            end

            6'b000100: //beq
            begin
                regDst = 0;
                aluSrc1 = 2'b00;
                aluSrc2 = 2'b00;
                memtoReg = 0;
                regWrite = 0;
                memRead = 0;
                memWrite = 0;
                branch = 1;
                aluOp = 3'b001;
                jump = 0;
            end

            6'b000010: //J
            begin
                regDst = 0;
                aluSrc1 = 0;
                aluSrc2 = 0;
                memtoReg = 0;
                regWrite = 0;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluOp = 2'b000;
                jump = 1;
            end

            6'b001000://addi
            begin
                regDst = 2'b00;
                aluSrc1 = 2'b00;
                aluSrc2 = 2'b01;
                memtoReg = 0;
                regWrite = 1;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluOp = 3'b000;
                jump = 0;
            end

```

```
113             6'b001100://andi
114             begin
115                 regDst = 2'b00;
116                 aluSrc1 = 2'b00;
117                 aluSrc2 = 2'b10;
118                 memtoReg = 0;
119                 regWrite = 1;
120                 memRead = 0;
121                 memWrite = 0;
122                 branch = 0;
123                 aluOp = 3'b011;
124                 jump = 0;
125             end
126 
127             6'b001101://ori
128             begin
129                 regDst = 2'b00;
130                 aluSrc1 = 2'b00;
131                 aluSrc2 = 2'b10;
132                 memtoReg = 0;
133                 regWrite = 1;
134                 memRead = 0;
135                 memWrite = 0;
136                 branch = 0;
137                 aluOp = 3'b100;
138                 jump = 0;
139             end
140 
141             6'b000011://jal
142             begin
143                 regDst = 2'b10;
144                 aluSrc1 = 2'b10;
145                 aluSrc2 = 2'b11;
146                 memtoReg = 0;
147                 regWrite = 1;
148                 memRead = 0;
149                 memWrite = 0;
150                 branch = 0;
151                 aluOp = 3'b000;
152                 jump = 1;
153             end
154 
155             default:
156             begin
157                 regDst = 0;
158                 aluSrc1 = 0;
159                 aluSrc2 = 0;
160                 memtoReg = 0;
161                 regWrite = 0;
162                 memRead = 0;
163                 memWrite = 0;
164                 branch = 0;
165                 aluOp = 0;
166                 jump = 0;
167             end
168         endcase
169     end
170 
```

```
assign RegDst = regDst;
assign ALUSrc1 = aluSrc1;
assign ALUSrc2 = aluSrc2;
assign MemToReg = memtoReg;
assign RegWrite = regWrite;
assign MemRead = memRead;
assign MemWrite = memWrite;
assign Branch = branch;
assign ALUOp = aluOp;
assign Jump = jump;

endmodule
```

- 其中需注意到：

```
output [1:0] RegDst, //new 00-rt 01-rd 10-31(for jal)
output [1:0] ALUSrc1, //new 00-rs 01-rt 10-pc(for jal)
output [1:0] ALUSrc2, //new 00-rt 01-signext 10-zeroext(for andi,ori) 11-8(for jal)
output [2:0] ALUOp, //new 000-add 001-sub 010-Rtype 011-and 100-or(new for ori)
```

四行为更新的控制信号，注释内有其详细用法.
可见jal和ori,andi的加入使控制信号RegDst,ALUSrc,ALUOp必须增加bit长度。

- ALUCtr.v文件更新如下:

```
module ALUCtr(
    input [2:0] ALUOp,
    input [5:0] Funct,
    output [3:0] ALUCtr,
    output jr,
    output shift
    );
    reg [3:0] aluCtr;
    reg JR;
    reg SHIFT;
    always @ (ALUOp or Funct)
    begin
        casex ({ALUOp, Funct})
            9'b000xxxxxx: begin
                aluCtr = 4'b0010;//add
                JR = 0;
                SHIFT = 0;
            end
            9'b001xxxxxx: begin
                aluCtr = 4'b0110;//sub
                JR = 0;
                SHIFT = 0;
            end
            9'b011xxxxxx: begin
                aluCtr = 4'b0000;//and
                JR = 0;
                SHIFT = 0;
            end
            9'b100xxxxxx: begin
                aluCtr = 4'b0001;//or
```

```
                JR = 0;
                SHIFT = 0;
            end
            //R-type
            9'b010100000: begin
                aluCtr = 4'b0010;//add
                JR = 0;
                SHIFT = 0;
            end
            9'b010100010: begin
                aluCtr = 4'b0110;//sub
                JR = 0;
                SHIFT = 0;
            end
            9'b010100100: begin
                aluCtr = 4'b0000;//and
                JR = 0;
                SHIFT = 0;
            end
            9'b010100101: begin
                aluCtr = 4'b0001;//or
                JR = 0;
                SHIFT = 0;
            end
            9'b010101010: begin
                aluCtr = 4'b0111;//slt
                JR = 0;
                SHIFT = 0;
            end
            9'b010100111: begin
                aluCtr = 4'b1100;//nor
                JR = 0;
                SHIFT = 0;
            end
            9'b010000000: begin
                aluCtr = 4'b0011;//sll
                JR = 0;
                SHIFT = 1;
            end
            9'b010000010: begin
                aluCtr = 4'b0100;//srl
                JR = 0;
                SHIFT = 1;
            end
            9'b010001000: begin
                aluCtr = 4'b1000;//jr
                JR = 1;
                SHIFT = 0;
            end
            default: aluCtr = 4'b0000;
        endcase
    end
    assign ALUCtr = aluCtr;
    assign jr = JR;
    assign shift = SHIFT;
endmodule
```

值得注意的是ALUOp增长为3bit,输出信号增加了jr和shift，目的是：

- ALUOp适配更新的Ctr.v
- jr和shift的出现缘由在jr/sll/srl为R-Type指令，但是其的控制信号和之前的R-Type指令有不同，需要修正之前产生的控制信号的用途，如下：

- ALUSrc1控制ALU-input1

```
    always @(ALUSrc1 or shift or readData1 or readData2 or CurPC) begin
        case (ALUSrc1)
            2'b00: begin
                if(!shift) input1 = readData1; //此处shift对R-Type的通用控制信号进行修正
                else input1 = readData2;
            end
            2'b01: input1 = readData2;
            2'b10: input1 = CurPC;
            default: input1 = 0;
        endcase
    end
```

- ALUSrc2控制ALU-input2

```
    always @(ALUSrc2 or shift or readData2 or SignExtImm or ZeroExtImm or Instr) begin
        case (ALUSrc2)
            2'b00: begin
                if(!shift) input2 = readData2; //此处shift对R-Type的通用控制信号进行修正
                else input2 = Instr[10:6];
            end
            2'b01: input2 = SignExtImm;
            2'b10: input2 = ZeroExtImm;
            2'b11: input2 = 8;
            default: input2 = 0;
        endcase
    end
```

- jr控制新加入的指令jr的实现

```
wire [31:0] PreNextPC1;
assign PreNextPC1 = (Branch&&zero)? BranchAddr:NextSeqPC;
wire [31:0] PreNextPC2;
assign PreNextPC2 = jr? readData1:PreNextPC1;
assign NextPC = Jump? JumpAddr:PreNextPC2;
```

- RegDst由于加入的jr需要做如下更新

```
always @(RegDst or Instr) begin
    case (RegDst)
        2'b00: writeReg = Instr[20:16];
        2'b01: writeReg = Instr[15:11];
        2'b10: writeReg = 5'b11111;
        default: writeReg = 0;
    endcase
end
```

- 在Registers.v中，因为jr的加入需要做如下更新:

```
module Registers(
    input Clk,
    input [25:21] readReg1,
    input [20:16] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    input regWrite,
    input jr,
    //此处加入了jr信号，用于R-Type指令一直令RegWrite=1的补救
    input reset,
    output [31:0] readData1,
    output [31:0] readData2
    );



    reg [31:0] regFile[0:31];
    reg [31:0] rd1;
    reg [31:0] rd2;
    integer i;

    always @(readReg1 or readReg2 or writeReg)
        begin
            rd1 = regFile[readReg1];
            rd2 = regFile[readReg2];
        end

    assign readData1 = rd1;
    assign readData2 = rd2;

    always @(negedge Clk)
        begin
            if(regWrite&&!jr&&writeReg!=0) regFile[writeReg] =
writeData;
            //在此处保证写寄存器文件时不为jr指令；同时让$0为硬件0，无法修改。
        end

    always @(posedge reset)
        begin
            for(i=0;i<32;i=i+1) begin
                regFile[i]=0;
            end
        end

endmodule
```

2.2.2 各模块文件的实例化与连接组装

- 许多内容与MIPS-9CPU中一致，不再指出复述
- 更新内容也在2.2.1节中进行了介绍
- 具体代码见工程文件Top.v

2.2.3 MIPS-16指令CPU的仿真验证

- 向原有的测试集添加新加入的7条指令的测试指令，完整内容如下:
  - 汇编代码:

```
lw $1,12($0)
lw $2,17($0)
add $3,$1,$2
sub $4,$1,$3
and $5,$2,$3
or $6,$4,$5
slt $7,$1,$2
slt $8,$2,$1
beq $1,$2,1
add $9,$0,$1
beq $1,$1,1
add $10,$0,$2
add $10,$0,$1
sw $5,-4($2)
j 19
or $11,$0,$1
or $11,$0,$1
or $11,$0,$1
or $11,$0,$1
or $12,$0,$1
# 从此开始为新加入的指令：
addi $13,$8,100  //测试addi
addi $14,$9,-40
andi $15,$10,57  //测试andi
andi $16,$10,-32
ori $17,$12,63   //测试ori
ori $18,$12,-43
sll $19,$13,14   //测试sll
srl $20,$14,9    //测试srl
jal 32           //测试jal 成功则$21未被赋值，$22被赋值，$31被赋值为PC+8
or $21,$0,$1
or $21,$0,$1
or $21,$0,$1
or $22,$0,$2
ori $23,$0,144   //用于测试jr，先载入需要的跳转地址进入寄存器$23
jr $23           //测试jr, 若成功则$25被赋值而$24未被赋值
or $24,$0,$1
or $25,$0,$2
```

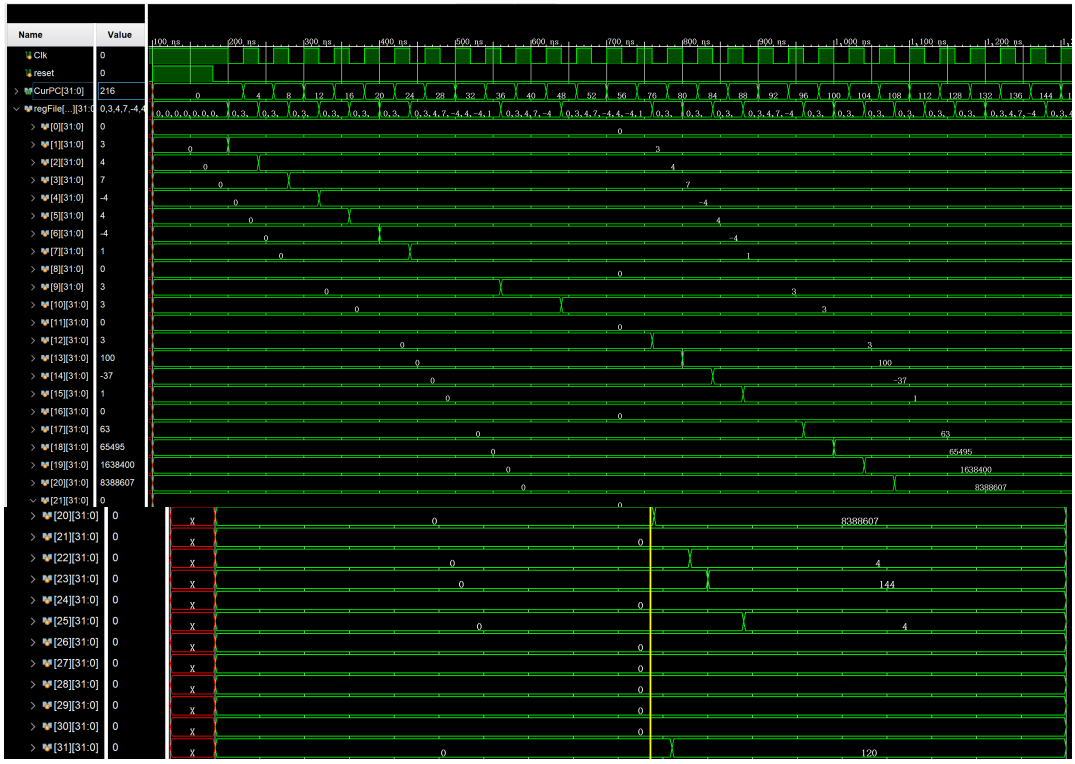
  - 对应的机器代码:

```
10001100000000010000000000001100
10001100000000010000000000010001
00000000001000100001100000100000
00000000001000110010000000100010
```

```
00000000010000110010100000100100
00000000100001010011000000100101
00000000001000100011100000101010
00000000010000010100000000101010
00010000001000100000000000000001
00000000000000001010010000100000
00010000001000010000000000000001
00000000000000001001010000000100000
00000000000000001010100000000100000
10101100001001011111111111111100
00001000000000000000000000010011
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001010110000100101
00000000000000001011000000100101
00100001000011010000000001100100
00100001001011101111111111011000
00110001010011111000000000111001
00110001010100001111111111100000
00110101100100010000000000111111
00110101100100101111111111010101
00000000000011011001101110000000
00000000000011101010001001000010
00001100000000000000000000100000
00000000000000001101010000100101
00000000000000001101010000100101
00000000000000001101010000100101
00000000000000001010110000100101
00110100000101110000000010010000
00000010111000000000000000001000
00000000000000001110000000100101
00000000000000001011001000100101
```

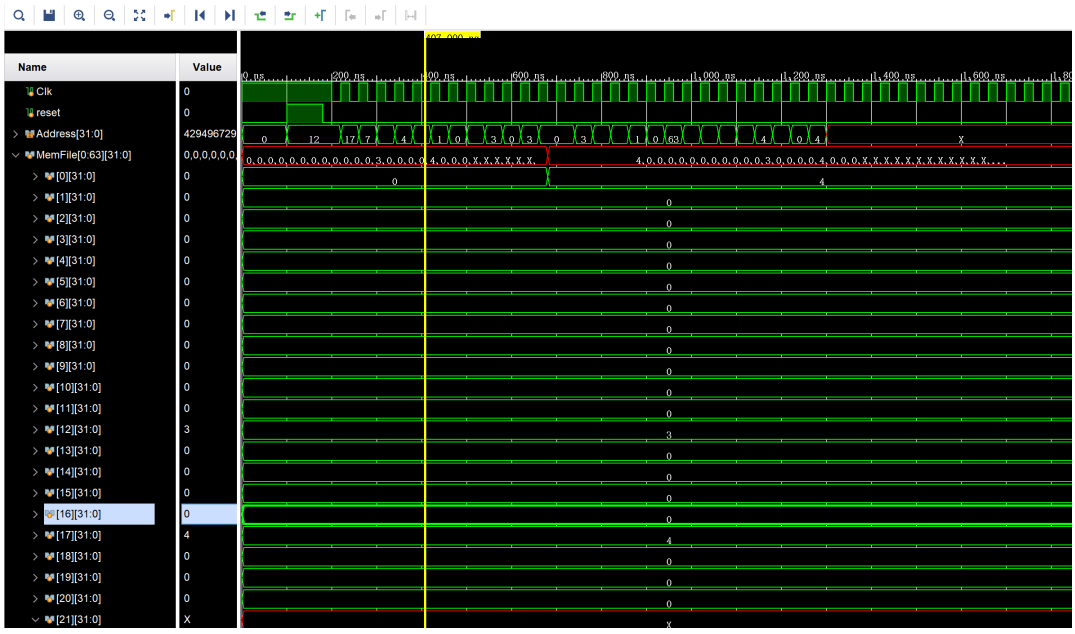
    - DataMemory初始化内容不变。
- 编写激励文件：与MIPS-9CPU一致，见上，不再赘述
- 开始仿真：

- 得到Registers的波形图如下：



  - 容易验证与预想相一致，新添加的7条指令实现成功，原有的9条指令除sw外也验证仍然正常工作中。

- 得到DataMemory的波形图如下：



  - sw仍然正常工作。

所以MIPS-16CPU整体设计实现调试圆满成功！

3. 实验心得

3.1 实验重难点

~~我觉得MIPS-9CPU还是很简单的，主要是MIPS-16CPU需要修改原有的指令译码信号，需要我们举一反三，有创新精神。此外，MIPS汇编代码的编写也是一个复杂的点。~~

这个实验的重难点在于“从零”开始构建一个完整的CPU。但是也不是完全的从零开始，因为之前也已经设计过许多可用的模块了，这里的难点就在模块的组装，考察到了对整个处理器执行逻辑的理解。解决难点的方式就是一步一步从IF,ID,EX,MEM到WB一步一步写，把基本框架搭好，把各个语句的执行逻辑调试正确。
到MIPS-16时，困难的地方就在于我们需要对Ctr,ALUCtr和ALU和众多控制信号进行集体的修改，让增加的7条指令能顺利的嵌入进原本的执行逻辑。

3.2 实验感想

做出来这个MIPS-16CPU还是很开心的，很有成就感。也让我真正地体会到自己成功组装出一个CPU的过程。感谢老师在微信群里面及时的回复和帮助。

4. 参考资料

【1】 2022计算机系统结构实验指导书lab5