单周期MIPS设计 实验报告

宁晨然 17307130178

1. 实验概述

本次实验完成了单周期`MIPS`设计，实现单周期处理器的微体系结构。单周期体系结构意味着，在一个时钟周期内执行完一条完整指令(CPI=1)，时钟周期以最长的指令所花的时间为准。控制比较简单，但速度比较慢、成本比较高。利用verilog硬件设计语言。

2. 实验展示

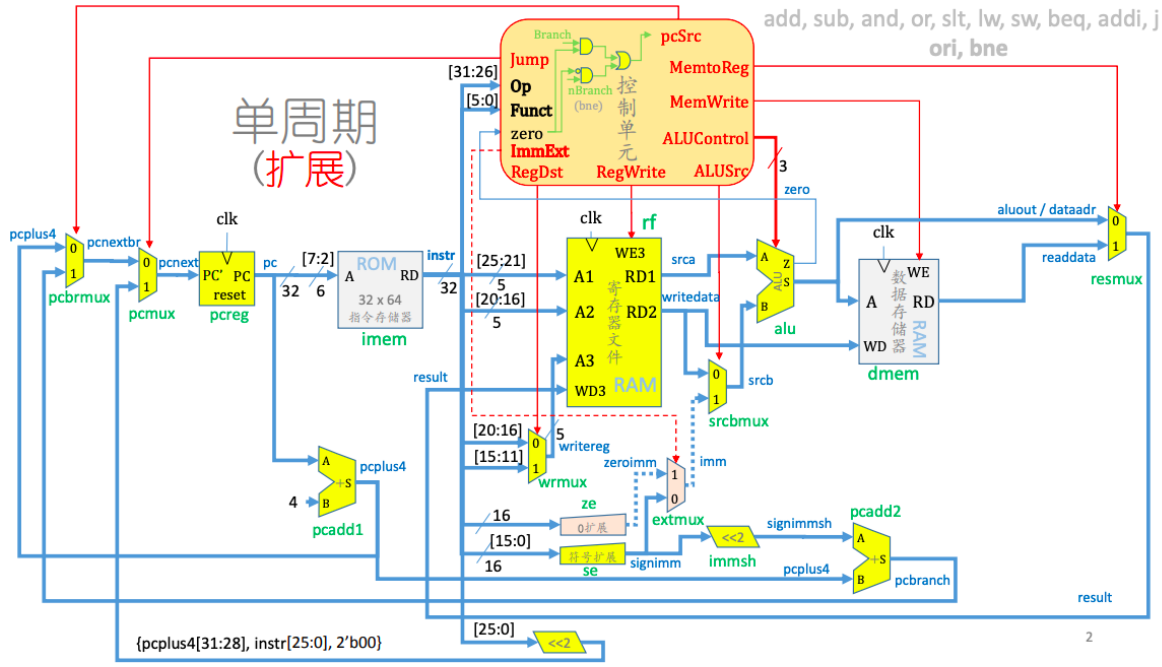
实验完成MIPS单周期处理器的设计，并且增加了指令，修改代码、修改逻辑图，并且最终通过了测试。

2.1 实验亮点

本人认为该实验可以达到**A水平**，在此列出本次实验作业的亮点：

- 实现了**MIPS单周期处理器的设计**
- 增加了很多指令，一共实现了**24条指令**！（具体指令在第五章讲解）
- 实现了**模块化测试和整合仿真测试，并且通过了最终的测试**
- 代码结构清晰，每个函数有**清晰易懂的注释**；**文件结构干净整洁**
- **实验报告清晰易懂，涵盖要点**、提纲挈领

2.2 实验原理



上图是单周期mips设计的模样，整个单周期处理器的构成由三个板块构成：

- MIPS主部件：主要部件由控制通路和数据通路组成，数据通路根据寄存器和储存器中的数据以及控制信号完成指令，控制通路负责根据指令调整控。
- dmem：是256行的32位数据存储器，用于存放数据。
- imem：指令存储器，用于存放所有即将执行的指令，根据pc值读取下一个需要执行的指令。

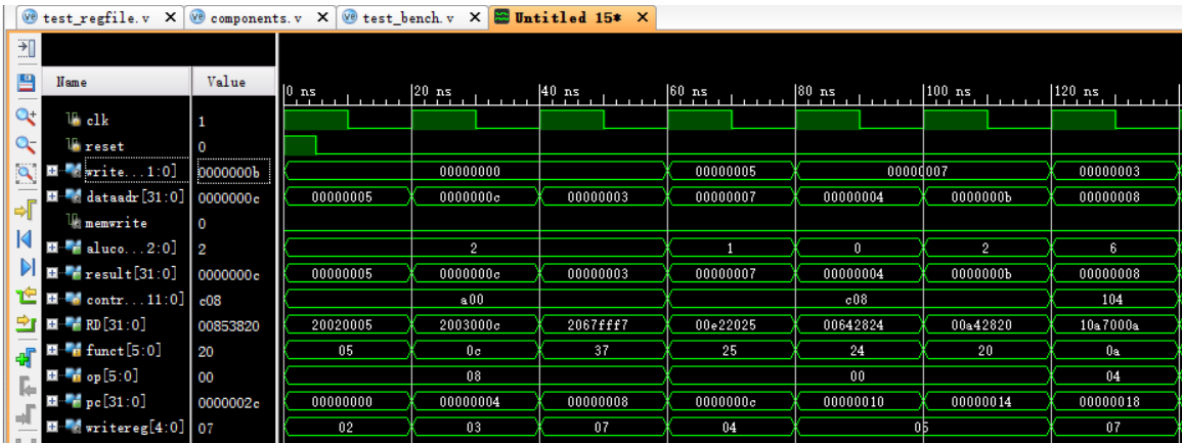
2.3 实验结果

具体测试在后面章节会讲到。

测试文件memfile通过了测试

```
run 200 ns
read data begin
done
run 200 ns
Simulation succeeded
```

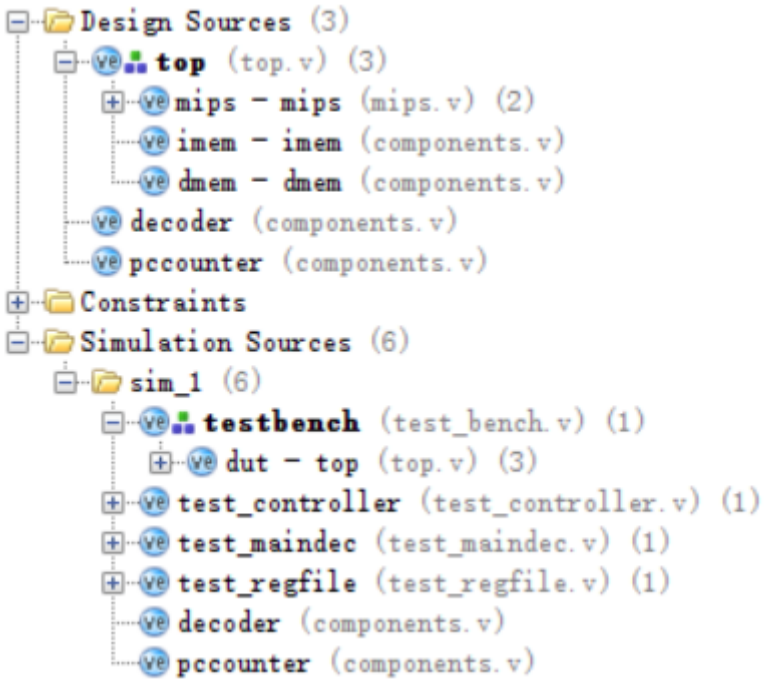
通过仿真一一核对了变量的正确运转



3. 实验代码讲解

先简单介绍本次实验的步骤：

- 第一步：构建mips单周期的流程，包括搭建`imem`/`dmem`/`mips`三个部分，并且由`top`文件调用
- 第二步：实现数据通路datapath，其包括在mips中，datapath实现包括操作元件和状态元件
- 第三步：实现controller，主要实现了maindecoder和aludecoder两个译码器，用于译码操作
- 第四步：实现单元测试，单独对各种模块进行测试，测试无误后再整合文件
- 第五步：实现整合测试，使用test_bench对整个mips的top文件进行测试，测试文件为memfile
- 第六步：添加指令，重复测试流程



下面先简单介绍文件结构。本单元部分介绍代码如何实现和原理，指令的讲解在第五章实验指令详解。

3.1 datapath

数据通路由操作元件和状态元件组合而成。通过分散方式或总线方式连接而成，进行数据存储、处理、传送的路径。下面先分板块讲解。

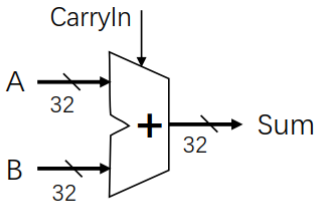
- 所有数据通路需要调用的元件均写在`components.v`文件中，便于**测试和管理**
- 所有元件均进行了**代码标注**，清晰易懂
- 下面解释的元件，均从**原理/输入输出/注意事项/代码实现讲解**

3.1.1 操作元件

实现的操作元件有以下几种：

加法器

- 原理：将两个数相加



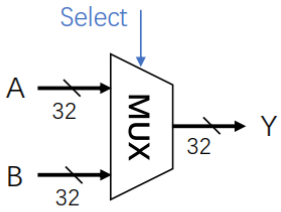
加法器 Adder

- 输入：这里只采用输入a,b；输出加法结果
- 注意：这里不考虑符号问题
- ```
  // Operation Component

  // 1. Adder
  module adder(
      input [31:0] a,b,
      output [31:0] sum
  );
  assign sum = a+b;
  endmodule
  ```
  ```

2路选择器

- 原理：通过selector选择输入的两个数据A或者B作为输出

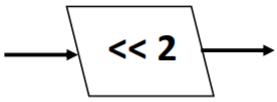


多路选择器

- 输入：两个数据和select值
- 输出：选择的值
- 注意：本次实验只需要使用二路选择器即可
- ```
  // 2. MUX2
  // choice 1 from 2
  module mux2#(parameter WIDTH=8)
  (
      input [WIDTH-1:0] a,b,
      input s,
      output [WIDTH-1:0] y
  );
  assign y = s?b:a;
  endmodule
  ```

移位器

- 原理：将输入的值左移两位

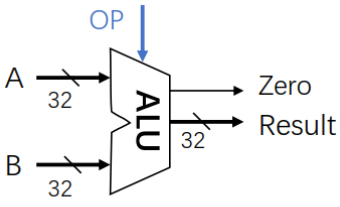


移位器

- 输入输出：输入值后，输出左移两位后的结果
- ```
  // 3. shift left
  // shift left 2 bit
  module sl2(
      input [31:0] a,
      output [31:0] y
  );
  assign y = a << 2;
  endmodule
  ```

ALU

- 原理：在实验一中已经实现

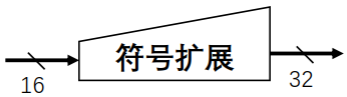


算术逻辑部件

- 注意：在实验添加指令后，alu部件需要修改
- ```
  // 4. ALU
  module alu(
      input [31:0] a,b,
      input [2:0] alucont,
      output reg [31:0] result,
      output zero
      );
  always @(*)
  case(alucont)
      3'b000:result <= a&b;
      3'b001:result <= a|b;
      3'b010:result <= a+b;
      3'b100:result <= a & ~b;
      3'b101:result <= a|~b;
      3'b110:result <= a-b;
      3'b111:result <= a<b ;
      default:result = 32'bz;
  endcase
  assign zero=result?0:1;
  endmodule
  ```

符号扩展

- 原理：根据16位数据的最高位扩展成32位



扩展器 3/

- 输入：16位数据；输出：32位扩展数据
- 注意：分为符号扩展和零扩展两种方法

```
// 6. signed number extender
// extend signed number
module signext(
    input [15:0] x,
    output [31:0] y
);
assign y = {{16{x[15]}},x};
endmodule

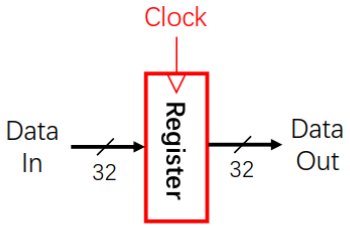
// 7. zero extender
module zeroext(
    input [15:0] x,
    output [31:0] y
    );
assign y = {16'b0,x};
endmodule
```

3.1.2 状态元件

实现的状态元件有下面几种：

寄存器
```

- 原理：当没有reset的时候，一直循环赋值新的寄存器值



寄存器

- 输入：时钟、reset、存入的值；输出：寄存器值
- 注意：数据通路中使用这个结构来专门存放PC值
- ```
  // State Component

  // 1. Register
  // stored data in register
  module flopr #(parameter WIDTH=8)(input clk,reset,
              input   [WIDTH-1:0] data_in,
              output reg [WIDTH-1:0] data_out);
  always @(posedge clk, posedge reset)
      if (reset) data_out<=0;
      else data_out<=data_in;
  endmodule
  ```

寄存器组

- 原理：组织寄存器，提供两个读接口和一个写接口
- 输入：时钟，是否写，读地址两个，写地址一个，写数据；输出：读取的两个数据
- 注意：不能修改寄存器0的值。
- ```
  // 7. Registers
  // 2 read 1 write
  module regfile(
      input clk, wen,
      input [4:0] address1,address2,address3,
      input [31:0] wdata,
      output [31:0] rd1,rd2
  );
  // 32 32-bit registers
  reg [31:0] rf [31:0];
  integer i;

  // initialization
  initial
  begin
      for(i=0;i<31;i=i+1)
      rf[i] = 32'b0;
  end

  // write 1
  always @(posedge clk)
  begin
      if( wen && (address3!=0)) // write and register 0 cannot be changed
          rf[address3] <= wdata;
  end
  ```

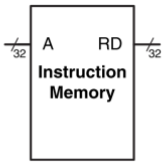
```
// read two at same time
assign rd1 = (address1!=0) ? rf[address1] : 0;
assign rd2 = (address2!=0) ? rf[address2] : 0;

endmodule
```

3.2 IMEM和DMEM

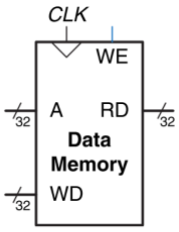
用于实现指令存储器和数据存储器。

- imem原理：imem初始化从memfile.dat中读取所需要的机器操作。根据输入的地址，寻找ram中的对应值。用于存放每次需要执行的指令。



指令存储器 (IM)

- dmem原理：dmem初始化为0。根据输入的地址，寻找ram中的对应值。



数据存储器 (DM)

- 注意：在top中才使用。

```
// 5. Instruction Memory
// read instruction every time
// A, RD
module imem(
    input [5:0] A,
    output reg [31:0] RD
);

// read from file
reg [31:0] RAM [63:0];
initial
    begin
        $display("read data begin");
        $readmemh("D:/memfile.dat",RAM);
        $display("done");
    end

// assign instruction
always @(A)
    RD = RAM[A];
endmodule

// 6. Data Memory
//
module dmem(
    input clk, we,
```

```
    input [31:0] a, WD,
    output [31:0] RD
);
reg [31:0] RAM[255:0];

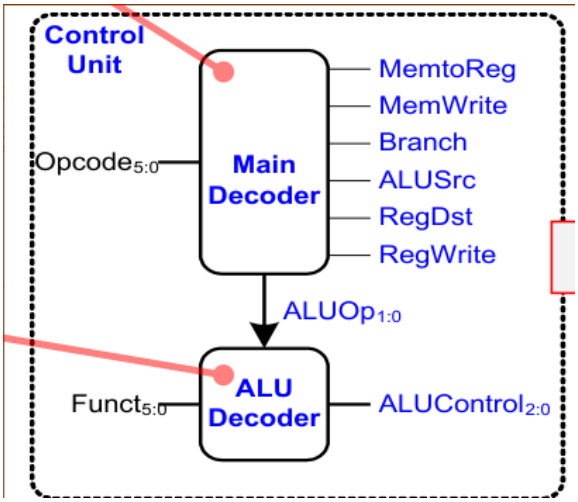
assign RD = RAM[a[31:2]];

always @(posedge clk)
begin
    if(we)
        RAM[a[31:2]] <= WD;
end

endmodule
```

3.3 controller

控制器由两个译码器构成，通过`op`，`funct`，`zero`的值译码得出所有操作的所需要的值。



3.3.1 maindecoder

主译码器主要用于译码各种控制的内容，根据操作码产生对应的所需操作信息。

注意：添加指令也是在此地方！！！

```
// main decoder
module maindec(
    input [5:0] op,
    output memtoreg, memwrite,
    output branch,alusrc,
    output regdst,regwrite,
    output [2:0] aluop,
    output jump,
    output immext,
    output nbranch
);
reg [11:0] controls;
assign
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop,immext,nbranch} =
controls;
always @(*)
    case(op)
```

```
        6'b000000:controls <= 12'b110000001000; //R-type
        6'b100011:controls <= 12'b101001000000; //lw
        6'b101011:controls <= 12'b001010000000; //sw
        6'b001000:controls <= 12'b101000000000; //addi
        6'b000100:controls <= 12'b000100000100; //beq
        6'b000101:controls <= 12'b000000000101; //bne
        6'b001101:controls <= 12'b101000001110; //ori
        6'b001100:controls <= 12'b101000010010; //andi
        6'b000010:controls <= 12'b000000100000; //j
        default:  controls <= 12'bxxxxxxxxxxx;
    endcase
endmodule
```

3.3.2 alucdecoder

进一步通过aluop的值，确定alu真正所需要的操作方式。此处可能会对R-type的类型的指令进行进一步译码。

```
// alu decoder
module aludec(
    input [5:0] funct,
    input [2:0] aluop,
    output reg [2:0] alucontrol
);
always @(*)
    case(aluop)
        3'b000: alucontrol <= 3'b010;       //add => addi,lw,sw
        3'b001: alucontrol <= 3'b110;       //sub => beq,bne
        3'b011: alucontrol <= 3'b001;       //or => ori
        3'b100: alucontrol <= 3'b000;       //and => andi
        default:
        begin
            case(funct)        //R-type
            6'b100000:alucontrol <= 3'b010;  //add
            6'b100010:alucontrol <= 3'b110;  //sub
            6'b100100:alucontrol <= 3'b000;  //and
            6'b100101:alucontrol <= 3'b001;  //or
            6'b101010:alucontrol <= 3'b111;  //slt
            default:  alucontrol <= 3'bxxx;
            endcase
        end
    endcase
endmodule
```

3.3.3 controller实例化

在controller中实例化maindec和aludec，并且将其中接线起来，使得其能够通过总线将操作码传递到datapath中去。

```
// controller
module controller(
    input [5:0] op,funct,
    input zero,
    output memtoreg,memwrite,
    output pcsrc,alusrc,
```

```
    output regdst,regwrite,
    output jump,
    output [2:0]alucontrol,
    output immext
    );
wire [2:0] aluop;
wire branch,nbranch;

maindec
md(op,memtoreg,memwrite,branch,alusrc,regdst,regwrite,aluop,jump,immext,nbranch)
;
aludec  ad(funct,aluop,alucontrol);

//assign pcsrc = (branch & zero)|(~branch & ~zero);   //beq && bne
assign pcsrc = (branch & zero)|(nbranch & ~zero);
endmodule
```

3.4 mips

然后在整合mips部件，实例化controller和datapath，并且正确接线。

```
// mips
module mips(
    input  clk,reset,
    output [31:0] pc,
    input  [31:0] instr,
    output memwrite,
    output [31:0] aluout,writedata,
    input  [31:0] readdata
    );
wire memtoreg,branch,pcsrc,zero,alusrc,regdst,regwrite,jump,immext;
wire [3:0] alucontrol;
controller c(instr[31:26],instr[5:0],zero,memtoreg,memwrite,pcsrc,
                alusrc,regdst,regwrite,jump,alucontrol,immext);
datapath dp(clk,reset,memtoreg,pcsrc,alusrc,regdst,regwrite,jump,
               alucontrol,zero,pc,instr,aluout,writedata,readdata,immext);
endmodule
```

3.5 top

最后添加top文件，将mips，imem，dmem全部实例化，接线起来。

```
// above all, top file
module top(
    input clk,reset,
    output [31:0] writedata,dataadr,
    output memwrite
    );
wire [31:0] pc,instr,readdata;

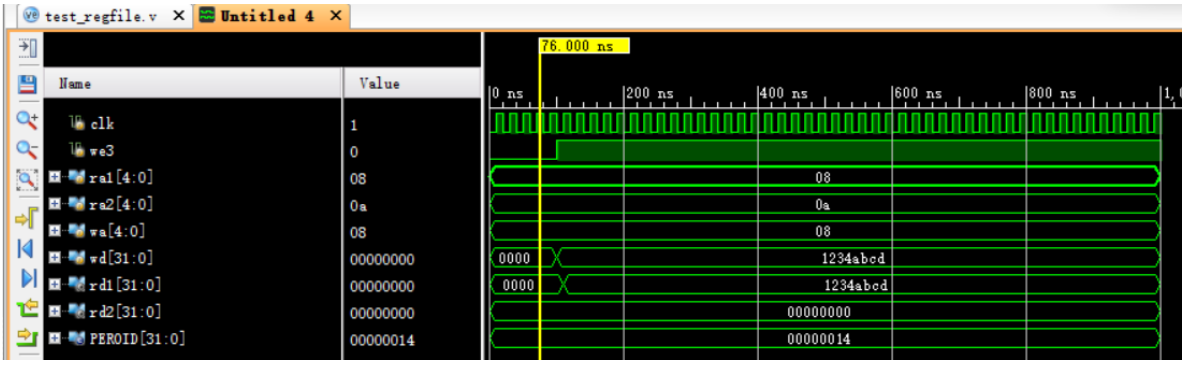
mips mips(clk,reset,pc,instr,memwrite,dataadr,writedata,readdata);
imem imem(pc[7:2],instr);
dmem dmem(clk,memwrite,dataadr,writedata,readdata);
endmodule
```

4. 实验测试

按照软件开发的原理，所有的部件应该先模块化测试再进行整合测试。本次实验延续了这种分开再整合测试的原理，贯穿整个实验。首先进行了`regfile`寄存器组的单元测试，再进行了`maindec`的单元测试。最终使用`test_bench`进行整合测试。

4.1 `test_regfile`

- 目的：用于模拟测试`regfile`模块的正确运转
- `regfile`：提供两个读取接口，可同时读取；提供一个写入接口，可同时写入。
- 实现方法：
  - 下图展示了仿真结果，其中100ns时写入寄存器
  - 写入后读出的结果也为寄存器的值
  - 另外`rd2`读出的值一直为0



- 代码讲解：
  - 先实例化`regfile`
  - 初始化时全部设置为0，100ns时写入寄存器
  - rd1读取写入寄存器8，rd2读取寄存器10

```
module test_regfile();
reg clk, we3;
    reg [4:0] ra1,ra2,wa;
reg [31:0] wd;
wire [31:0] rd1,rd2;

regfile MUT(clk,we3,ra1,ra2,wa,wd,rd1,rd2);

// initialization to zero
initial begin
    clk = 0;
    we3 = 0;
    wa = 0;
    wd = 0;
    ra1 = 0;
    ra2 = 0;

    #100;
    we3 = 1;
    wd = 32'h1234abcd;
end

// simulate clk
```

```
parameter PEROID = 20;
always begin
    clk = 1'b0;
    # (PEROID/2) clk = 1'b1;
    # (PEROID/2);
end

always begin
    wa = 8;
    ra1 = 8;
    ra2 = 10;
    #PEROID;
end
endmodule
```

4.2 `test_maindec`

- 目标：用于模拟`maindec`模块的正确运转
- `maindec`：用于产生所有控制单元的命令，再发送给总线，用于控制单周期中所有操作。
- 实现方法：
  - 下图展示了`maindec`测试结果，状况良好，正确产生了命令。
  - 一共12位的控制字节符合预期

- 预期结果为12位数字结果，如图所示，保证了正确性。

```
case(op)
    6'b000000:controls <= 12'b110000001000; //R-type
    6'b100011:controls <= 12'b101001000000; //lw
    6'b101011:controls <= 12'b001010000000; //sw
    6'b001000:controls <= 12'b101000000000; //addi
    6'b000100:controls <= 12'b000100000100; //beq
    6'b000101:controls <= 12'b000000000101; //bne
    6'b001101:controls <= 12'b101000001110; //ori
    6'b001100:controls <= 12'b101000010010; //andi
    6'b000010:controls <= 12'b000000100000; //j
    default:  controls <= 12'bxxxxxxxxxxx;
endcase
```

- 代码解释：
  - 先实例化
  - 再每个20ns设置一个操作数op

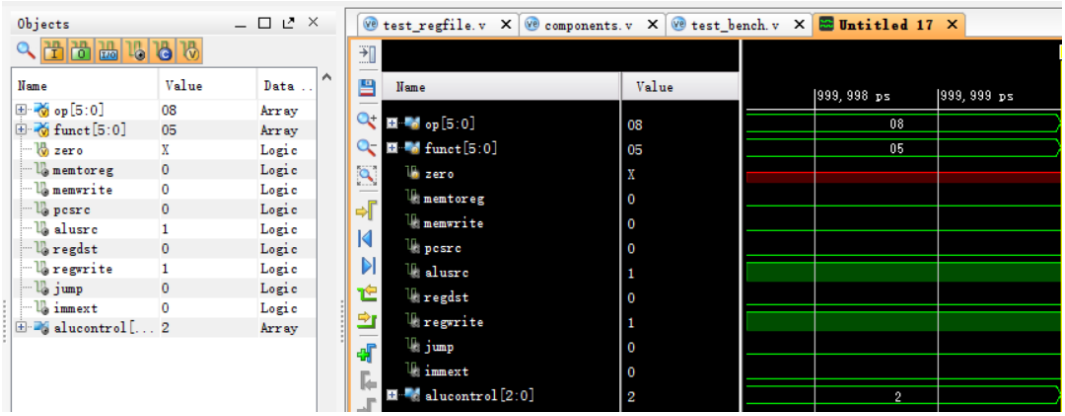
```
module test_maindec();
reg [5:0] op;
```

```
wire mem2reg,memwrite;
wire branch,alusrc,nbranch;
wire regdst,regwrite,jump,immext;
wire [2:0] aluop;

maindec MUT(op,mem2reg,memwrite,branch,alusrc,regdst,
                regwrite,aluop,jump,immext,nbranch);
initial begin
    op=0;
    #20 op=6'b000000;
    #20 op=6'b100011;
    #20 op=6'b101011;
    #20 op=6'b001000;
    #20 op=6'b000100;
    #20 op=6'b000101;
    #20 op=6'b001101;
    #20 op=6'b001100;
    #20 op=6'b000010;
end
endmodule
```

4.3 `test_controller`

在跑了`test_bench`之后发现`controller`中的`aluop`并没有接线，发现出现了`bug`，`maindec`输出的`aluop`并没有传参到`aludec`中的`aluop`输入中，出现了接线错误。在纠结很久`debug`调试过程中，没有找到明显问题，后来干脆重新写一个单元测试测试`controller`模块，果然此模块也出现了接线问题。

- 目的：用于测试`controller`模块的正确性
- 实现方法：
  - 传入`addi`对应的`op`和`funct`参数，查看输出结果是否正确
  - 

  - 实验结果如图所示，测试完毕后输出`alucontrol`结果为2，为理想实验结果。
- 困难点：
  - 由于`maindec`和`aludec`中间存在接线，一共有12位参数需要传，在写代码的时候，正好写反了两位参数，导致接线错误。
  - 打算之后写代码的时候直接复制函数体的参数内容，一个一个对照的写，避免参数过多出错。
- 代码解释：
  - 首先实例化`controller`
  - 传入`addi`的`op`和`funct`

- 查看仿真结果的正确性

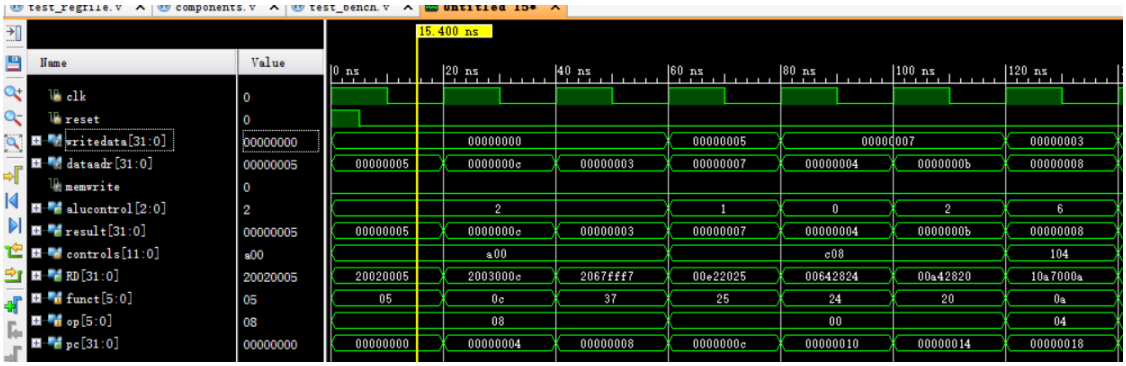
```
module test_controller();
reg [5:0] op,funct;
reg zero;
wire memtoreg,memwrite,pcsrc,alusrc,regdst,regwrite;
wire jump,immext;
wire [2:0]alucontrol;

controller
cont(op,funct,zero,memtoreg,memwrite,pcsrc,alusrc,regdst,regwrite,jump,alucontro
l,immext);

initial begin
   op = 6'b001000;
   funct = 6'b000101;
end

endmodule
```

4.4 `test_bench`

- 目的：用于测试单周期设计是否正确
- 实验结果：**通过了测试文件memfile**
- ```
  run 200 ns
  read data begin
  done
  run 200 ns
  Simulation succeeded
  ```
- 实现方法：
  - 实例化`top`后，输入汇编代码的机器码，查看是否运转正常
  - 示范输出如下
- 

- 由于`imem`会读入`memfile`文件中的数据，作为机器码输入指令，需要修改`memfile`查看其如何运转
-
  ```

memfile.dat

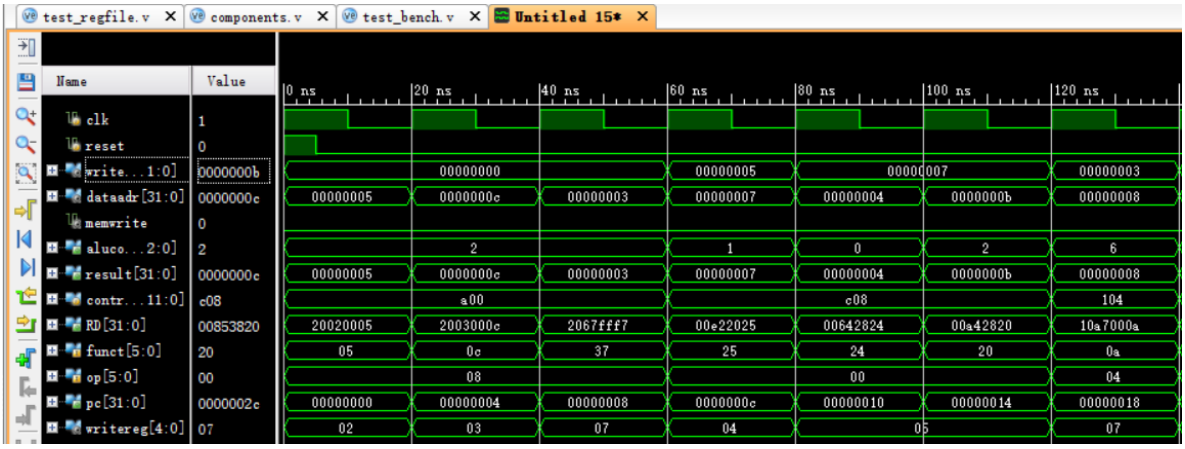
```
20020005
2003000C
2067FFF7
00E22025
00642824
00A42820
10A7000A
0064202A
10800001
20050000
00E2202A
00853820
00E23822
AC670044
8C020050
08000011
20020001
AC020054
```

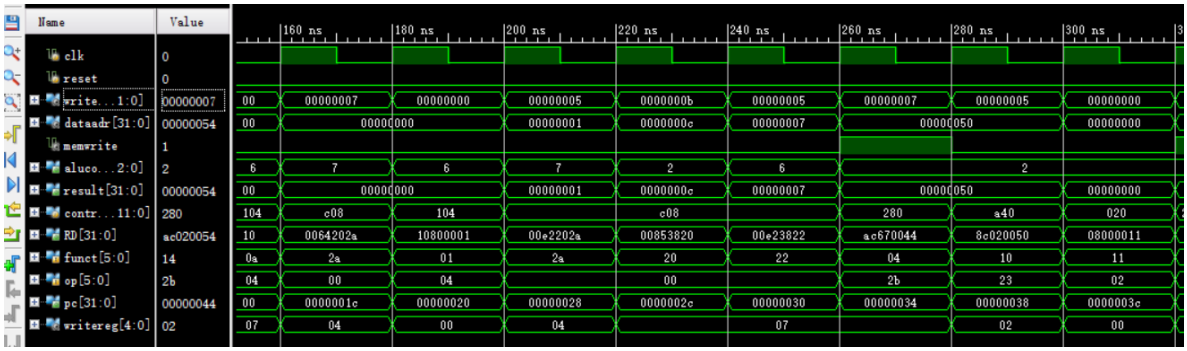
- 上述机器码的实际指令为

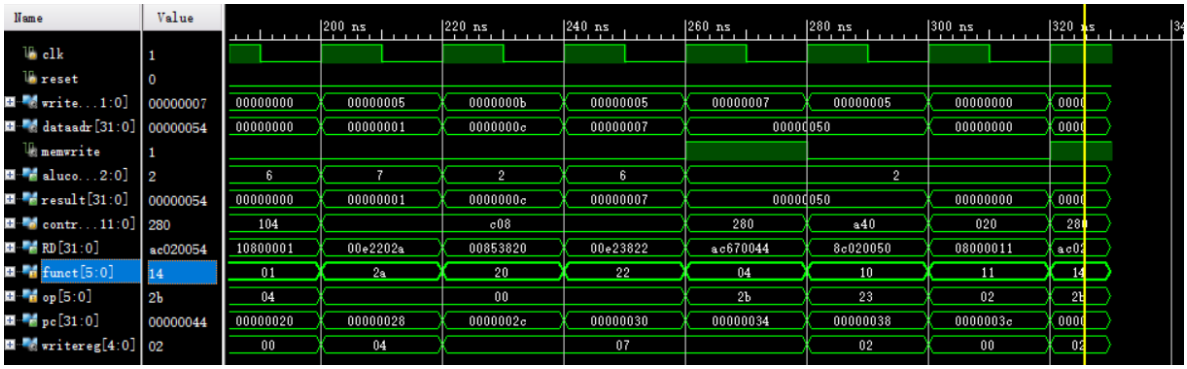
```
#         Assembly              Description              Address    Machine
main:     addi $2, $0, 5        # initialize $2 = 5      0          20020005
          addi $3, $0, 12       # initialize $3 = 12     4          2003000c
          addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
          or   $4, $7, $2       # $4 = (3 OR 5) = 7      c          00e22025
          and  $5, $3, $4       # $5 = (12 AND 7) = 4    10         00642824
          add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
          beq  $5, $7, end      # shouldn't be taken     18         10a7000a
          slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
          beq  $4, $0, around   # should be taken        20         10800001
          addi $5, $0, 0        # shouldn't happen       24         20050000
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
          add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
          sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
          sw   $7, 68($3)       # [80] = 7               34         ac670044
          lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
          j    end              # should be taken        3c         08000011
          addi $2, $0, 1        # shouldn't happen       40         20020001
end:      sw   $2, 84($0)       # write mem[84] = 7      44         ac020054
```

4.4.1 一一核对结果

- 根据验证，测试文件的所有指令与结果**全部正确**
- 通过了memfile的测试，`console`显示了`Simulation succeeded`。
- 测试成功结束



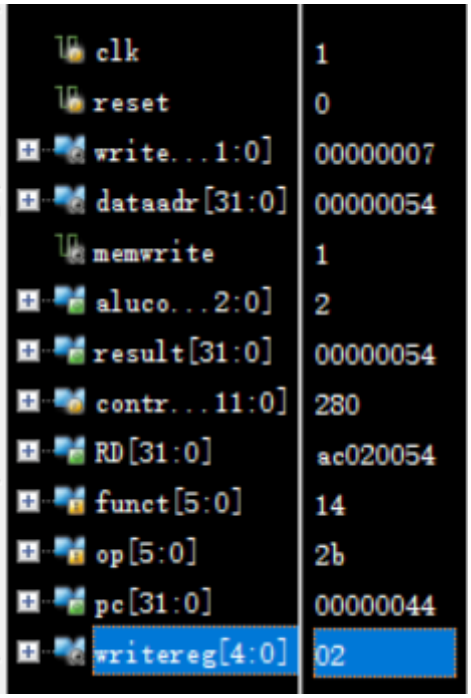




指令	作用	检测
addi	初始化寄存器2 = 5	观察写回的寄存器`writereg`分别为2/3/7；写回的值为`result`分别为5/12/3
addi	初始化寄存器3 = 12	
addi	初始化寄存器7 = 3	
or	寄存器4 = 7	观察写回寄存器为4；result为7
and	寄存器5 = 4	观察写回寄存器为5；result为4
add	寄存器5 = 11	观察写回寄存器为5；result为11；alucont为2（加法）
beq	5==7?	alucont为6（减法）；pc没跳转
slt	寄存器4 = 0	观察写回寄存器为4；result为0；alucont为7(a<b)
beq	4==0?	alucont为6（减法）；pc没跳转
slt	寄存器4 = 1	观察写回寄存器为4；result为1
add	寄存器7 = 12	观察写回寄存器为7；result为12
sub	寄存器7 = 7	观察写回寄存器为7；result为7；alucont为6（减法）
sw	寄存器7的值存入内存	观察memwrite=1；写入内存地址`dataaddr`为80；写入的寄存器值为7
lw	将80内存的值取出到寄存器2	观察写回寄存器为2；result为80
j	跳转到end	观察pc值下一个变成了44;
sw	写入内存84值寄存器7	观察memwrite=1；写入内存地址`dataaddr`为84；写入的寄存器值为7

4.4.2 每个变量解释

用于测试的时候，选取了以下变量进行观测，查看是否符合预期，下面解释一下最关键核心的部分。



变量名	解释
`writedata`	写入内存的数据
`dataaddr`	写入内存的地址
`memwrite`	是否写入内存
`alucont`	控制alu进行某个操作
`result`	需要写回寄存器的值
`writereg`	写回寄存器的号码

- 根据上述核心变量，可以检测整个mips周期的核心过程
- 上面核对结果时，就是使用了这个变量的检测方法

5. 实验指令详解

实验中最重要的板块就是实现所有的mips指令。本次实验实现了**24条指令**！（MIPS总共31条指令）

5.1 指令扩展

本次实验扩展指令的时候，根据需要扩展了更多的控制位数，用于控制数据的流通。

nbranch

原先的branch变量用于控制beq跳转时，是否跳转。然而有时候我们不是a==b跳转，还有a!=b的时候的跳转，比如bne。

所以需要控制一个在“非zero”的情况下跳转。

```
assign pcsrc = (branch & zero)|(nbranch & ~zero);
```

zeroext

原先只描述了signext，即符号扩展，再后来为了描述零扩展，先在component里面加了（上面有代码）zeroextender，为了实现零扩展的指令情况。为了选择zeroext还是signext，添加了变量**immext**用于控制选择signed extend还是zero extend。

```
signext     se(instr[15:0],signimm);
zeroext     ze(instr[15:0],zeroimm);
mux2 #(32)  extmux(signimm,zeroimm,immext,imm);
```

alu

原先的alu只能有3bit的aluop操作，远远少于实际alu的操作内容，所以重新撰写了alu的功能，使用了4位的aluop。添加了左移右移异或等复杂操作。(此处参考了网上代码的功能解释)

真值表对应如下代码中parameter中的名词表示，将所有功能与真值表一一对照了起来。

所以也需要修改一下aluop位数和alucontrols的位数，修改为4。

```
// updated alu
module alu(input [31:0] A,B,
           input [3:0] alucont,
           output [31:0] result,
           output zero);

reg [32:0] temp;
wire signed [31:0] SrcA = A, SrcB = B;

parameter ADDU = 4'b0000,
          SUBU = 4'b0001,
          ADD  = 4'b0010,
          SUB  = 4'b0011,
          OR   = 4'b0100,
          AND  = 4'b0101,
          XOR  = 4'b0110,
          NOR  = 4'b0111,
          SLTU = 4'b1000,
          SLT  = 4'b1001,
          SLL  = 4'b1010,
          SRL  = 4'b1011,
          SRA  = 4'b1100,
          LUI  = 4'b1101;

always @(*)
case(alucont)
    ADDU: temp = A + B; // addu
    SUBU: temp = A - B; // subu
    ADD : temp = SrcA + SrcB;  // r = a + b (signed) add
    SUB : temp = SrcA - SrcB;  // r = a - b (signed) sub
    AND : temp = A & B;  // r = a & b and
    OR  : temp = A | B;  // r = a | b or
    XOR : temp = A ^ B;  // r = a ^ b xor
    NOR : temp = ~(A | B); // r = ~(a | b) nor
    SLT : temp = (SrcA < SrcB) ? 1 : 0; // slt
    SLTU: temp = (A < B) ? 1 : 0;
    SLL : temp = B << A; //sll
    SRL : begin
```

```
            if(A == 0)
            {temp[31:0], temp[32]} = {B, 1'b0};
            else
            {temp[31:0],temp[32]} = B >> (A - 1); // srl
          end
    SRA : begin
              if(A == 0)
              {temp[31:0], temp[32]} = {B, 1'b0};
              else {temp[31:0], temp[32]} = SrcB >>> (A - 1); // sra
            end
    LUI : begin
          {temp[31:0], temp[32]} = {B[15:0], 17'b0};
          end
    default:temp = A + B;

endcase
assign  result = temp[31:0] ;
assign  zero = (result == 32'b0) ? 1 : 0;

endmodule
```

5.2 指令分析

经过扩展功能后，controls位数变成了13位。

```
assign
{regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop,immext,nbranch} =
controls;
```

- regwrite：是否需要写回register
- regdst：写回寄存器是rt还是rd
- alusrc：srcb要加的数是rt还是imm
- branch：是否需要根据zero跳转
- memwrite：是否需要写入存储器
- memtoreg：选择到result的值是aluout还是readdata
- jump：是否跳转
- aluop：alu操作控制
- immext：选择signext还是zeroext
- nbranch：是否需要根据!zero跳转

```
case(op)
    6'b000000:controls <= 13'b1100000111100; //R-type

    // I-type
    6'b001000:controls <= 13'b1010000001000; //addi
    6'b001001:controls <= 13'b1010000000010; //addiu
    6'b001100:controls <= 13'b1010000010110; //andi
    6'b001101:controls <= 13'b1010000010010; //ori
    6'b001110:controls <= 13'b1010000011010; //xori
    6'b100011:controls <= 13'b1010010001000; //lw
    6'b101011:controls <= 13'b0010100001000; //sw
    6'b000100:controls <= 13'b0001000000100; //beq
    6'b000101:controls <= 13'b0000000000101; //bne
    6'b001010:controls <= 13'b0011000100100; //slti
    6'b001011:controls <= 13'b0011000100010; //sltiu

    // J-type
    6'b000010:controls <= 13'b0000001000000; //j
    default:  controls <= 13'bxxxxxxxxxxxx;
endcase
```

根据译码的op，可以得到每个controls位上的值。

mips指令中，分为**R-type,I-type,J-type三种类型。**

R-type

这类操作，一般是对寄存器的操作。本次实验中实现的内容是，所有需要存入regsiter中的操作。下图是示例。

instr	regwrite	regdst	alusrc	branch	memwrite	memtoreg	jump	aluop	immext	nbranch
R-type										
add	1	1	0	0	0	0	0	1111	0	0

Bit #	31..26	25..21	20..16	15..11	10..6	5..0			
R-type	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中rs=$2，rt=$3，rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中rs=$2，rt=$3，rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ；其中rs=$2，rt=$3，rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs - rt ；其中rs=$2，rt=$3，rd=$1,无符号数

一般是，对rs和rt有一定alu中的操作后，存入rd中。所以需要regwrite=1。

所以对于op case的分支中，rtype的内容一样，需要写入的寄存器regdst=1，表示写入rd寄存器。aluop使用1111，用于跳转default状态。（进一步根据funct判断）剩下的内容不需要，均为0。

下面是aludec，alu译码器，生成alucontrol内容，此时根据alu的不同的操作，填充进去每个操作需要的alu操作。

这样就实现好了**下面12个指令。**

```
case(funct)        //R-type
6'b100000:alucontrol <= 4'b0010;  //add
6'b100001:alucontrol <= 4'b0000;  //addu
6'b100010:alucontrol <= 4'b0011;  //sub
6'b100011:alucontrol <= 4'b0001;  //subu
6'b100100:alucontrol <= 4'b0101;  //and
6'b100101:alucontrol <= 4'b0100;  //or
6'b100110:alucontrol <= 4'b0110;  //xor
6'b100111:alucontrol <= 4'b0111;  //nor
6'b101010:alucontrol <= 4'b1001;  //slt
6'b101011:alucontrol <= 4'b1000;  //sltu
6'b000100:alucontrol <= 4'b1010;  //sllv
6'b000110:alucontrol <= 4'b1011;  //srlv
default:  alucontrol <= 4'bxxxx;
endcase
```

I-type

itype的内容一般与imm立即数有关，有下面**11个指令已经实现。**参照mips设计文档可以填充每个真值表。

```
// I-type
6'b001000:controls <= 13'b1010000001000; //addi
6'b001001:controls <= 13'b1010000000010; //addiu
6'b001100:controls <= 13'b1010000010110; //andi
6'b001101:controls <= 13'b1010000010010; //ori
6'b001110:controls <= 13'b1010000011010; //xori
6'b100011:controls <= 13'b1010010001000; //lw
6'b101011:controls <= 13'b0010100001000; //sw
6'b000100:controls <= 13'b0001000000100; //beq
6'b000101:controls <= 13'b0000000000101; //bne
6'b001010:controls <= 13'b0011000100100; //slti
6'b001011:controls <= 13'b0011000100010; //sltiu
```

I-type	op	rs	rt	immediate			
addi	001000	rs	rt	immediate	addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ； 其中rt=$1,rs=$2
addiu	001001	rs	rt	immediate	addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate ； 其中rt=$1,rs=$2
andi	001100	rs	rt	immediate	andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ； 其中rt=$1,rs=$2
ori	001101	rs	rt	immediate	andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ； 其中rt=$1,rs=$2
xori	001110	rs	rt	immediate	andi $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate ； 其中rt=$1,rs=$2

- addi：需要写回寄存器、使用立即数、操作为有符号加法，所以regwrite=alusrc=1,aluop=0001
- addiu：同上，使用无符号加法，regwrite=alusrc=1,aluop=0000，使用零扩展immext=1
- andi：写回寄存器，使用立即数，零扩展。aluop=0101。
- ori：写回寄存器，使用立即数，零扩展。aluop=0100。
- xori：写回寄存器，使用立即数，零扩展。aluop=0110。
- lw：写回寄存器，使用立即数，需要读取存储器中值。aluop为加法。
- sw：写回存储器memwrite=1，aluop为加法。alusrc=1。
- beq：branch跳转，无符号加法aluop。
- bne：nbranch跳转，无符号加法aluop。
- slti：选择imm立即数alusrc=1，branch跳转，aluop使用slt操作。
- sltiu：选择imm立即数，branch跳转，aluop使用sltu操作。

J-type

```
// J-type
6'b000010:controls <= 13'b0000001000000; //j
```

这里实现了一个jtype就是jump指令，这个很简单，就是将jump设置为1即可。

5.3 指令测试

修改verilog代码结构后，添加了各种指令的代码后，也成功达到了仿真成功。

```
run 200 ns
read data begin
done
run 200 ns
Simulation succeeded
```

6. 实验感受

实验过程中，让我了解到了mips单周期处理器的整个处理流程，从一步一步的小部件，比如加法器，alu设计，组合成数据通路，在到设计控制通路，分配每一个指令应该有的步骤，再到合并为mips部件，合并imem和dmem两个部件，一起设计实现了一个大的mips完整的单周期处理器。这样的流程让我充分体会到了，mips的单个模块到整合的步骤。

实验过程中不乏遇到困难，比如在verilog上十分难debug，我使用了vscode编译器写verilog，能够避免大部分变量名的错误，和标点出错。然后再在vivado上编译测试，不断修改语法错误后，尝试在top文件的test_bench上测试整个部件的运转良好性。我也学会了分模块测试的方法，先测试不同的模块，比如小部件regfile/maindec/controller。再整合起来测试整个大的mips，test_bench运行的时候我就遇到了一个问题，就是maindec和controller中传的aluop/alucontrols的值竟然不同，后来单元测试也是对的，就是整合出了问题，后来仔细核对发现其实原因出在了实例化的时候，传入的参数和函数的参数位置不同，两个变量名写反了，出现了值传递错误的问题。这个问题也应该在之后好好避免一下。

实验中也获得了很多知识，巩固了ics中学的取指译码等流程，知道了op和funct用于控制译码的过程，根据译码的不同指令，确定其中指令需要进行哪些操作。也了解清楚了单周期的含义，我发现不同指令所需要的clk周期是不同的，有的指令很快就能执行完，有的指令需要执行好几个周期才能结束。

希望在下次实验中好好反省，学习自己实验过程中的优点，吸取教训。