1 实验内容

MIPS 架构 CPU 的传统流程可分为取指、译码、执行、访存、回写 (Instruction Fetch, Decode,Execution,Memory Request,Write Back),五阶段。实验一完成了 ALU 设计并掌握了存储器 IP 的使用;实验二实现了单周期 CPU 的取指、译码阶段,完成了 PC、控制器的设计。在实验一与实验二的基础上,单周期 CPU 的设计的各模块已经具备,再引入数字逻辑课程中所实现的多路选择器、加法器等门级组件,通过对原理图的理解,分析单条(单类型)指令在数据通路中的执行路径,依次连接对应端口,即可完成单周期 CPU。

阅读实验原理实现以下模块:

(1) Datapath,其中主要包含 alu(实验一已完成),PC(实验二已完成),adder、mux2、signext、sl2(其中 adder、mux2 数字逻辑课程已实现,signext、sl2 参见实验原理),
(2) Controller(实验二已完成),其中包含两部分,分别为 main_decoder,alu_decoder。
(3) 指令存储器 inst_mem(Single Port Ram),数据存储器 data_mem(Single Port Ram);使用 Block Memory Generator IP 构造指令,注意考虑 PC 地址位数统一。(参考实验二)
(4) 参照实验原理,将上述模块依指令执行顺序连接。实验给出 top 文件,需兼容 top 文件端口设定。
(5) 实验给出仿真程序,最终以仿真输出结果判断是否成功实现要求指令。

2 实验设计

2.1 控制器

32 位 MIPS 指令在不同类型指令中分别有不同结构,但 IR[31:26] 表示的 opcode,IR[5:0] 表示的 funct 为译码阶段明确指令控制信号的主要字段。Controller 模块将输入的 opcode 码和 funct 码解码,输出 memtoreg、memwrite、pcsrc、alusrc、regdst、regwrite、branch、jump 和 aluop,ALU decoder 和 alu control 信号。具体接口和设计过程见实验二。

2.2 数据通路

2.2.1 功能描述

Controller 将解码得到的控制信号传入 datapath(其中 memwrite 传给 Data memory,其他信号均传入 datapath),并接收 Data memory 输入的 data 和 Instruction memory 输入的 IR,输出 ALU 计算结果、写回数据、pc 信号以及 zero 信号。

datapath 模块包含许多模块,主要可以分为 PC 和 Data 两个部分:

PC->PC+4-> 判断 branch(IR[15:0]«2+PC+4)-> 判断 jump({PCPlus4[31:28],IR[25:0]«2})-> 更新 PC

IR-寄存器堆->RD1、RD2-ALU->ALUReult、WriteData-DataMem->ReadData、ALUResult->Result

具体连线图如图 1：

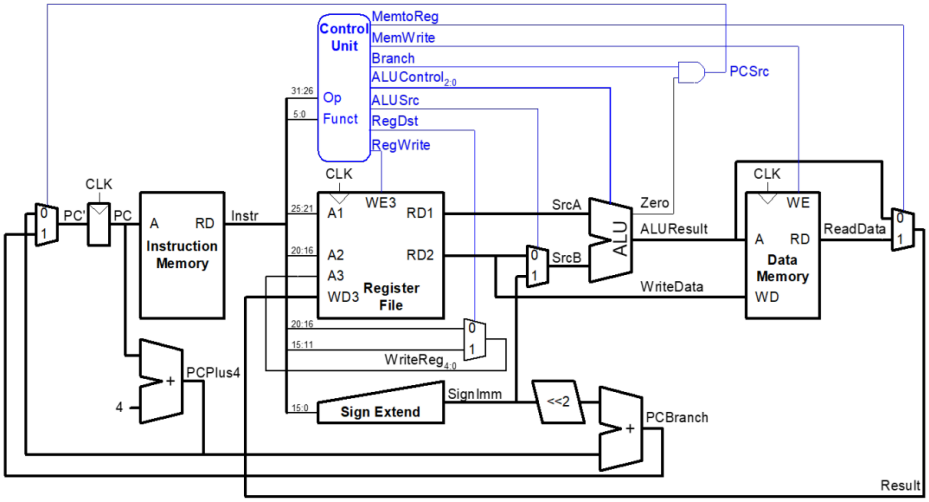

图 1: datapath 连线图

2.2.2 接口定义

如表 1。

2.3 存储器

2.3.1 指令存储器

指令存储器类型选择 Single Port ROM，端口宽度选择 32bits，深度选择 1024，如图 2。使能端设置为 Always Enabled，不设复位信号。具体端口参数选项如图 3。其他选项需加载 coe 文件，如图 4。参数选择完成后生成 ROM 如图 5。

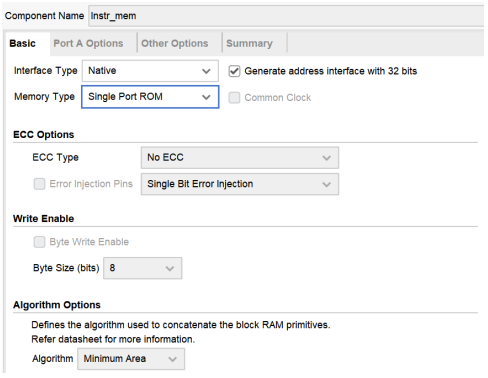

图 2: 指令存储器参数设置

信号名	方向	位宽	功能描述
clk	Input	1-bit	时钟信号
rst	Input	1-bit	重置信号
aluctrl	Input	3-bit	ALU 控制信号
memtoreg	Input	1-bit	判断写回寄存器堆的是 ALU 的计算结果还是 lw 读取的数据
alusrc	Input	1-bit	判断 SrcB 是 RD2 还是 SignImm
regdst	Input	1-bit	判断写寄存器号
regwrite	Input	1-bit	写寄存器的使能信号
pcsrc	Input	1-bit	判断是否执行 branch
jump	Input	1-bit	判断是否执行 jump
readdata	Input	32-bit	lw 从 Data memory 中读取的数据
IR	Input	32-bit	指令
aluresult	Output	32-bit	ALU 运算结果
writedata	Output	32-bit	sw 写进 Data memory 的数据
pc_out	Output	32-bit	当前 PC 值
zero	Output	1-bit	ALU 计算结果是否为 0

表 1: datapath 接口定义

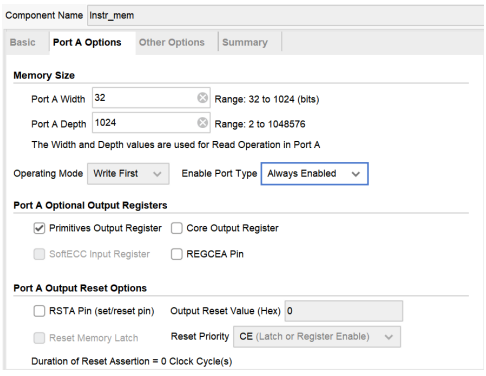

图 3: 指令存储器参数设置

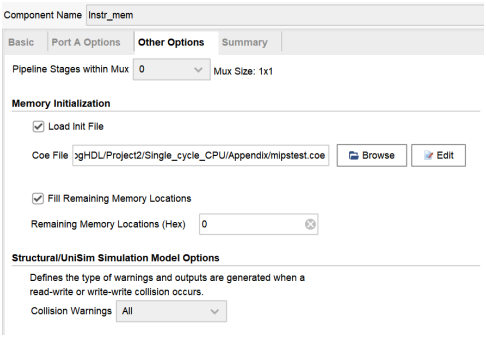

图 4: 指令存储器参数设置

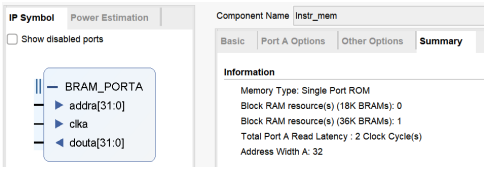

图 5: 指令存储器参数设置

2.3.2 数据存储器

数据存储器类型选择 Single Port RAM，端口宽度选择 32bits，深度选择 1024，如图 6。使能端设置为 Always Enabled，不设复位信号。具体端口参数选项如图 7。其他选项不必加载 coe 文件，如图 8。参数选择完成后生成 RAM 如图 9。

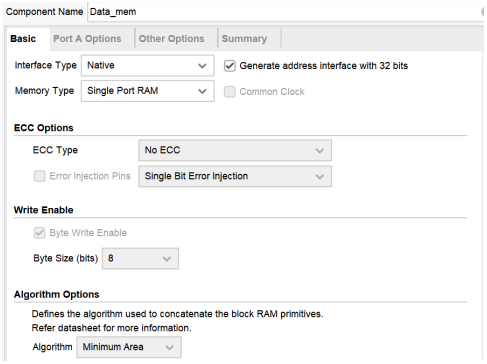

图 6: 数据存储器参数设置

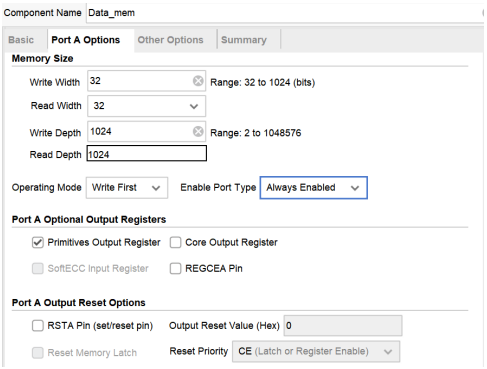

图 7: 数据存储器参数设置

2.4 MIPS 处理

将 controller 模块和 datapath 模块连通。controller 为纯组合逻辑，不包含任何存储功能，datapath 为时序逻辑，需要时钟信号控制。因为本实验是单周期 CPU，将 datapath 视为一个整体(由同一个时钟信号控制)，指令寄存器取指令、datapath 数据传输、Data memory 存取数据都需要在同一个时钟周期内完成。因此 Instruction memory 和 Data memory 均采用下降沿触发，Datapath 采用上升沿触发。当 clk 上升沿到来时，Datapath 将传入的 IR 和各控制信号进行运算，输出 PC、WriteData 和 ALUOut；clk 下降沿来临时，Data memory 将 WriteData 写回 (sw)/ALUOut 取出 (lw)/

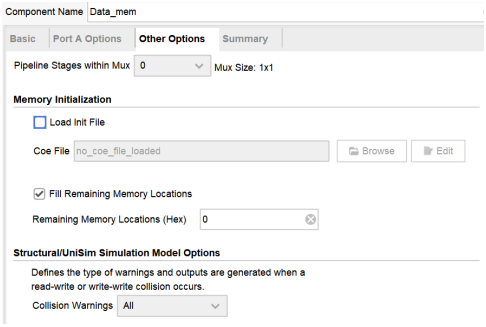

图 8: 数据存储器参数设置

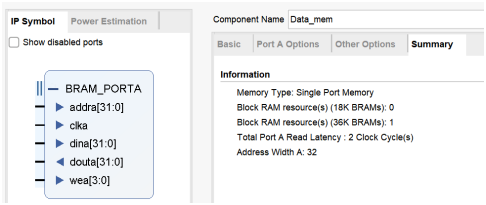

图 9: 数据存储器参数设置

ALUOut 传出 (R), 同时 Instruction memory 取指令传出; 等到下一个 clk 上升沿来临, Datapath 接收传回的 ReadData(lw)/ALUOut(R) 写回寄存器堆。但此处还会有一个冲突, 即下一条指令需要用到上一条写回的寄存器堆内数据, 本实验暂时不考虑此冲突。

具体连线图如图 10:

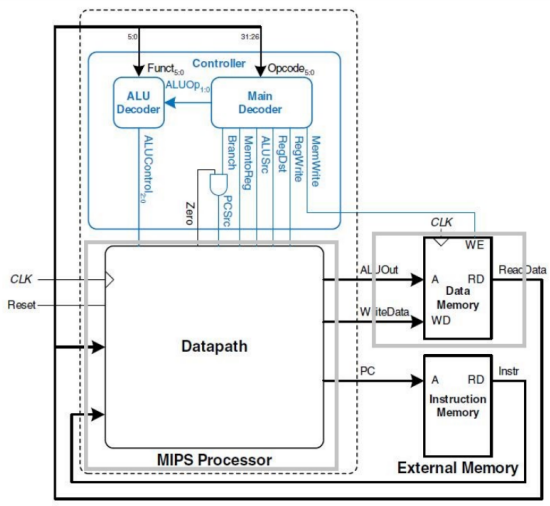

图 10: MIPS 模块连线图

3 实验过程记录

3.1 问题 1:RAM 的时钟信号问题

问题描述: 该问题是实验二的历史遗留问题, ROM 接受传入的 addr, 输出对应的数据, 应该是组合逻辑, 为什么需要时序逻辑? 能否不接受 clk 和 rst 信号?

解决方案: 组合逻辑只有逻辑门, 即根据输入给出输出, 没有记忆和存储功能。任意时刻的状态由该时刻的输入信号决定, 无输入信号时则输出为空。因此存储器均采用时序逻辑, 需要 clk 控制触发器, 信号边沿触发时改变状态。

3.2 问题 2:PC 模块无输入功能

问题描述: 实验二中设计的 PC 模块只接受 clk 和 rst 为输入信号, 通过 clk 边沿触发将输出 pc 自增。但实验三需要 PC 模块接受 PC' 信号输入, 以实现 branch、jump 等功能。

解决方案: 将 PC 模块修改为接受 PC' 作为输入, 并根据 clk 输出 PC 的时序逻辑。PC+4 的功能放入 adder 中实现, 逻辑图如下。

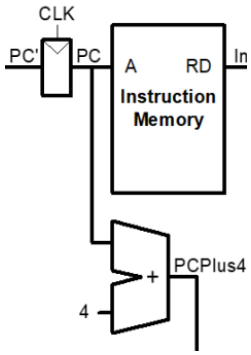

图 11: PC 模块图

3.3 问题 3: 实验一、二与三端口不兼容

问题描述: 由于实验三的很多模块是直接从实验一、二导入的, 因此存在端口不兼容的问题。如实验二中的 PC 缺少输入, 实验一中的 ALU 的操作数 a 只有 8 位, b 内置为 1 且无 zero 信号等。

解决方案: 将 ALU 模块的 a 操作数改为 32 位, 将 b 添加为输入, 增加 zero 信号; PC 模块增加输入。

3.4 问题 4:Controller 和 Datapath 之间的信号传递

问题描述: 设计 Controller 模块时没有做接收 zero 信号的处理, 而 Datapath 接收的判断 branch 跳转信号为 PCSrc(branch & zero) 信号, 两个模块无法直接连接。

解决方案: 原本考虑将 Zero 传入 Controller 模块, 修改原设计文件, 使得两个模块可以直接

传递。但为了尽可能保留原设计文件，因此在 MIPS 模块将两个模块相连，其中加入一个 wire 型信号做 branch & zero 再传入 Datapath。连接如图：

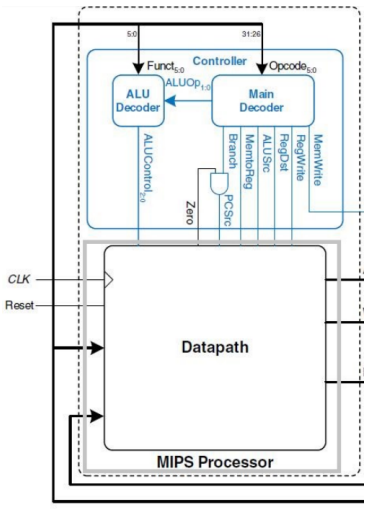

图 12: PCSrc 接口

3.5 问题 5: 出入参数位宽与模块定义不一致

问题描述：在设计 mux2 模块时，原先直接将端口定义为 32 位，但仿真发现多出为 X 或 Z。检查发现判断写寄存器信号时传入的参数都是 5 位宽 (IR[20:16]、IR[15:11])。

解决方案：将 mux2 模块改为可变参数模块即可。

3.6 问题 6:Datapath 仿真多出信号为 X 或 Z

问题描述：在仿真 datapath 时发现多处信号为 X 和 Z，连线也没问题，从 jump 信号向后就都出了问题。

解决方案：仿真文件实例化模块时遗漏了参数 aluctrl（这个问题确实很蠢...）。调试过程中发现，由于 datapath 模块信号较多，实例化模块时经常出现遗漏参数、顺序错误等问题，使得仿真出现大量 X 和 Z 信号。建议采用实参对应形参的方式，并且保证实例化端口数量与设计文件一致。

3.7 问题 7: 调用 IP 核连线位宽不匹配

问题描述：综合 top 模块生成电路图发现 Data memory 和 Instruction memory 的输出都没有连接到网线上。

解决方案：将输入信号扩展到相应位数再输入 IP 核。

3.8 问题 8:PC 与 IR 不同步

问题描述:仿真发现指令与 PC 值之间存在延迟,指令变化落后于 PC 一个周期。

解决方案:由于信号在 datapath 中传递需要时间,如果 datapath 和指令存储器都用上升沿触发的话,指令寄存器在上升沿收到的输入信号相当于上一个周期的地址。因此将指令寄存器改成下降沿触发即可。

3.9 问题 9:J 型指令地址跳转出错

问题描述:仿真发现 J 型指令跳转不到预期地址。

解决方案:J 型指令为伪直接寻址,跳转地址为 PC 高四位拼接 j 型指令 26 位地址,再左移两位。

4 实验结果及分析

4.1 仿真结果

仿真截图和控制台输出如下:

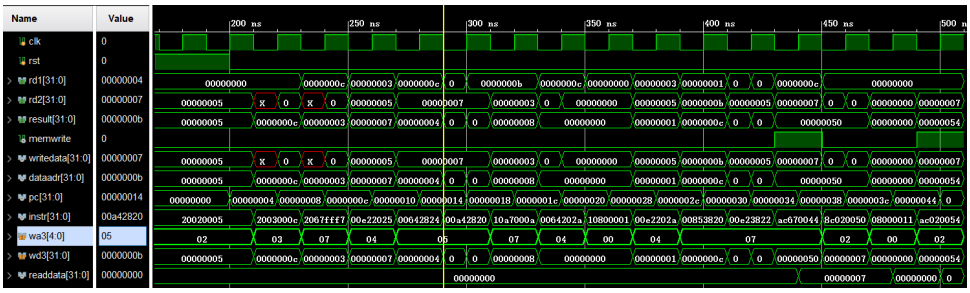

图 13: 仿真波形图

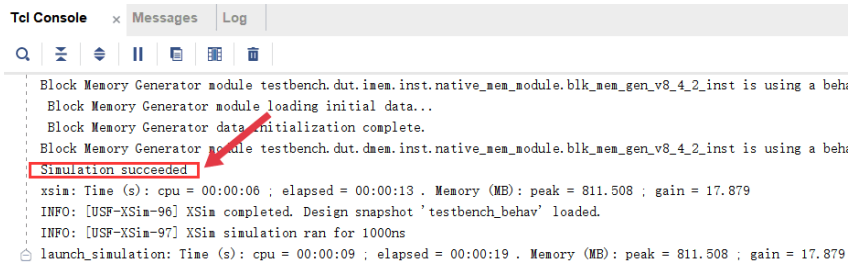

图 14: 控制台输出

4.2 不同指令在数据通路中的信号传递

4.2.1 R 型指令

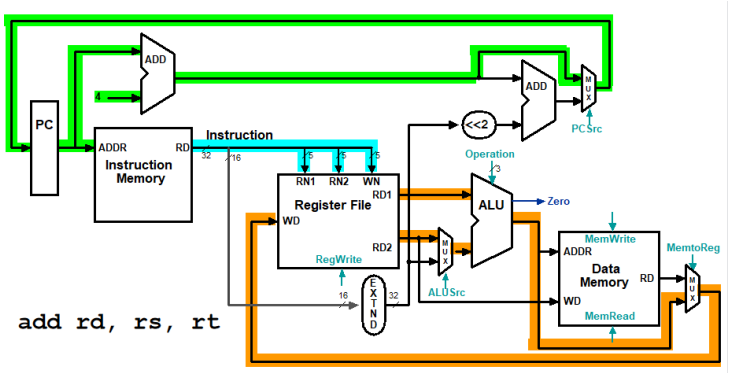

图 15: R 型指令数据通路

4.2.2 lw 指令

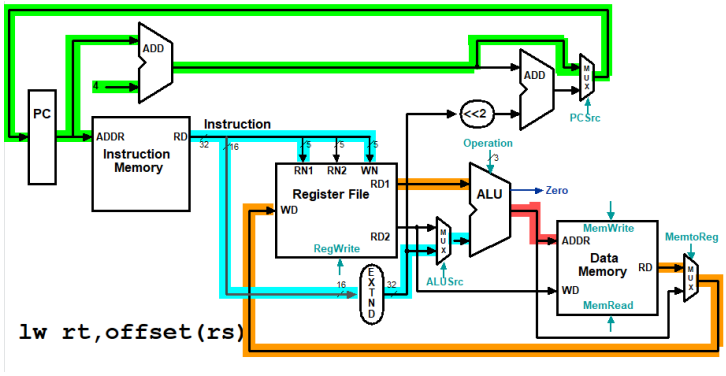

图 16: lw 指令数据通路

4.2.3 sw 指令

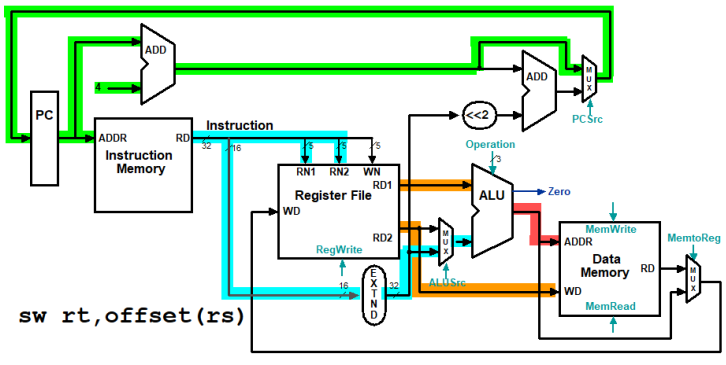

图 17: sw 指令数据通路

4.2.4 branch 指令

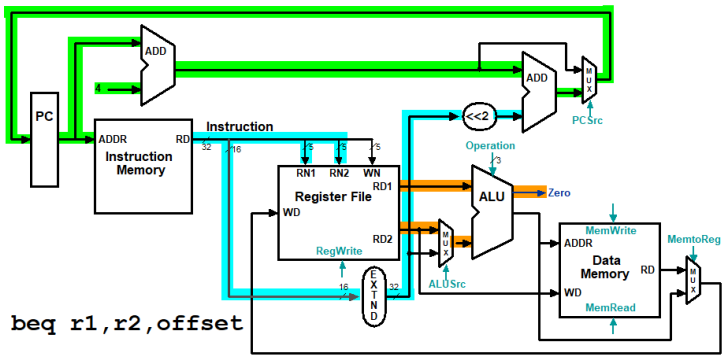

图 18: branch 指令数据通路

A Datapath 代码

```
‘timescale 1ns / 1ps
/*数据通路模块：将Controller解码的控制信号传入datapath（Controller输出的
    memwrite传给Data Memory，其他的均传入datapath），并接收Data memory输入的data
    和Instruction memory输入的IR，输出ALU计算结果、写回数据、pc信号以及zero信
    号。datapath模块包含许多模块：主要可以分为PC和Data两部分。
PC部分：PC->PC+4->判断branch(IR[15:0]<<2+PC+4)->判断jump({PCPlus4[31:28],IR
    [25:0]<<2})->更新PC
Data部分：IR-寄存器堆->RD1、RD2(与SignImm选择)-ALU->ALUReult、WriteData-DataMem
    ->ReadData、ALUResult->Result
*/
module datapath(
    input clk,rst,
    input[2:0] aluctrl,
    input memtoreg,alusrc,regdst,regwrite,pcsrc,jump,    //pcsrc=branch&zero，从
        外部传入
    input[31:0] readdata,                //lw指令从Data memory读出的指令
    input[31:0] IR,
    output[31:0] aluresult,
    output[31:0] writedata,
    output[31:0] pc_out,
    output zero
    );
    //PC部分
    wire[31:0] PC_1,PC_2,PC,PCPlus4;     //PC_1为判断branch后的地址信号，PC_2为
        判断jump后的地址信号
    assign pc_out=PC;
    pc gate1(clk,rst,PC_2,PC);           //PC模块
    adder gate2(PC,32'h00000004,PCPlus4);       //PC+4

    wire[31:0] SignImm;
    wire[15:0] tmp;
    assign tmp=IR[15:0];
    signext gate3(tmp,SignImm);     //lw、sw、addi、beq指令需要对IR[15:0]进行扩
        展
```

```
    wire[31:0] PCBranch_in;
    wire[31:0] PCBranch_out;
    sl2 gate4(SignImm,PCBranch_in);
    adder gate5(PCBranch_in,PCPlus4,PCBranch_out);       //计算branch跳转指令

    mux2 gate6(PCPlus4,PCBranch_out,pcsrc,PC_1);         //判断是否执行branch
    mux2 gate7(PC_1,{PCPlus4[31:28],IR[25:0],2'b00},jump,PC_2);     //判断是否执
        行jump

    //Data部分
    wire[4:0] WriteReg;                  //写寄存器号
    wire[31:0] Result;                   //写回寄存器数据
    wire[31:0] RD1,RD2;                  //寄存器堆读出数据
    assign writedata=RD2;
    mux2 #(5) gate8(IR[20:16],IR[15:11],regdst,WriteReg);     //判断写寄存器号
    regfile gate9(clk,regwrite,IR[25:21],IR[20:16],WriteReg,Result,RD1,RD2);

    wire[31:0] SrcB,ALUResult;
    assign aluresult=ALUResult;
    mux2 gate10(RD2,SignImm,alusrc,SrcB);                //判断RD2 or SignImm
    alu gate11(aluctrl,RD1,SrcB,ALUResult,zero);         //ALU运算

    mux2 gate12(ALUResult,readdata,memtoreg,Result);     //判断写回寄存器堆的是
        ALU的计算结果or lw读取的data

endmodule
```

B PC 代码

```
`timescale 1ns / 1ps
/*pc模块：对实验二中PC模块加以修改，接受PC'作为输入，并根据clk输出PC。PC+4的功
    能放入adder中实现,否则无法实现branch、jump功能
注意：addr不赋初值仿真会出现问题，第一条指令会被吞掉
*/
module pc(
    input clk,rst,
    input[31:0] pc,
    output reg[31:0] pc_new=0
);
    always@(posedge clk)begin
        if(rst)begin
            pc_new=0;
        end
        else begin
           pc_new=pc;
        end
```

```
    end
endmodule
```

C Signext 代码

```
‘timescale 1ns / 1ps
/*符号扩展模块：将输入16位二进制数扩展为32位*/
module signext(
    input [15:0] a,
    output [31:0] y
    );
    assign y={{16{a[15]}},a};    //有符号数扩展
endmodule
```

D Regfile 代码

```
‘timescale 1ns / 1ps
/*寄存器堆(32*32bits)模块：用于读写ALU运算时用到的数据，其中读取为组合逻辑，写
    入为时序逻辑*/
module regfile(
  input wire clk,
  input wire we3,                  //写使能信号
  input wire[4:0] ra1,ra2,wa3,     //读端口1,2，写端口3
  input wire[31:0] wd3,            //写数据
  output wire[31:0] rd1,rd2        //读数据
    );

  reg [31:0] rf[31:0];             //寄存器堆

  always @(posedge clk) begin
    if(we3) begin
       rf[wa3] <= wd3;
    end
  end

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;      //约定0号寄存器只存储0
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

E MIPS 代码

```
‘timescale 1ns / 1ps
/*MIPS指令处理模块：接收clk、rst、指令和数据
```

```
注意：controller为纯组合逻辑，不包含任何存储功能，datapath为时序逻辑，需要时钟
    信号控制。因为本实验是单周期CPU，将datapath视为一个整体(由同一个时钟信号控制
    )，指令寄存器取指令、datapath数据传输、Data memory存取数据都需要在同一个时钟
    周期内完成。因此Instruction memory和Data memory均采用下降沿触发，Datapath采
    用上升沿触发。当clk上升沿到来时，Datapath将传入的IR和各控制信号进行运算，输
    出PC、WriteData和ALUOut；clk下降沿来临时，Data memory将WriteData写回(sw)/
    ALUOut取出(lw)/ALUOut传出(R)，同时Instruction memory取指令传出；等到下一个
    clk上升沿来临，Datapath接收传回的ReadData(lw)/ALUOut(R)写回寄存器堆。但此处
    还会有一个冲突，即下一条指令需要用到上一条写回的寄存器堆内数据，本实验暂时不
    考虑此冲突。
*/
module mips(
  input wire clk,rst,
  input wire[31:0] IR,
  output wire[31:0] pc_out,
  output wire memwrite,
  output wire[31:0] aluout,writedata,
  input wire[31:0] readdata
    );
  wire[2:0] aluctrl;
  wire memtoreg,alusrc,regdst,regwrite,branch,jump,pcsrc,zero;
  assign pcsrc=branch&zero;

  controller gate15(IR[31:26],IR[5:0],aluctrl,memtoreg,memwrite,alusrc,regdst,
      regwrite,branch,jump);
  datapath gate16(clk,rst,aluctrl,memtoreg,alusrc,regdst,regwrite,pcsrc,jump,
      readdata,IR,aluout,writedata,pc_out,zero);

endmodule
```