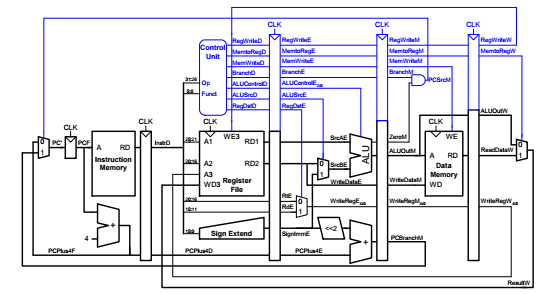
Lab4 五段流水MIPS CPU设计

## 设计概述

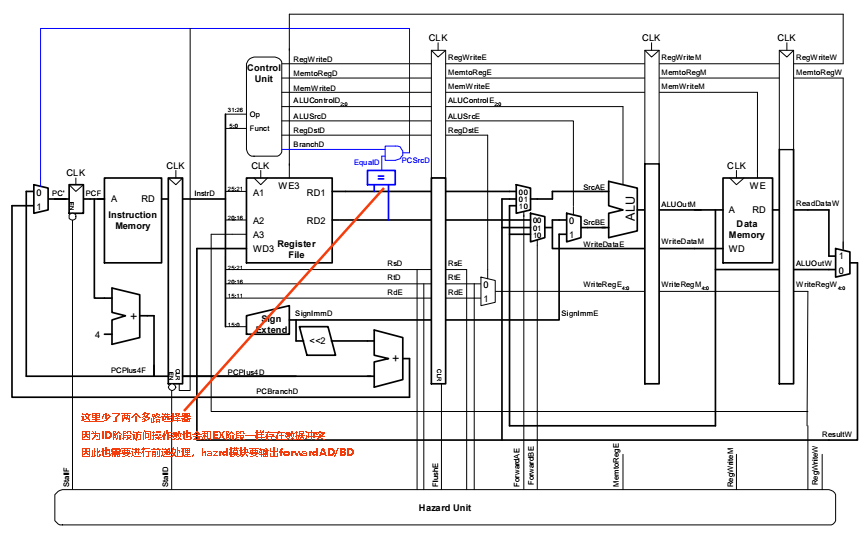
基本上各个最底层模块都可以复用之前的，只不过**数据通路以及控制器的集成需要拆分为五个阶段：“取指”、“译码”、“执行”、“访存”和“写回”**，此外**数据通路模块内需要集成冒险处理模块**

1. 带流水线的数据通路图如下：



图中少画了jump指令的处理流程

1. 带流水线的且利用前递、阻塞解决冒险的数据通路图如下



图中少画了jump指令的处理流程

**在原有的基础上增加了触发器、比较器、三选一器以及冒险模块**

### 触发器

各个阶段之间以触发器的方式间隔，触发器有Flopr、Floprc、Flopenr、Flopenrc三种

1. Flopr：clk和rst 实现PC

`timescale 1ns / 1ps  
  
//带有reset的D触发器  
module flopr(  
 input clk,  
 input rst,  
  
 input [31:0]i\_addr,  
 output reg [31:0]o\_addr  
);  
 **always @**(posedge clk) begin  
 if (~rst) begin  
 o\_addr<=0;  
 end  
 else begin  
 o\_addr<=i\_addr;  
 end  
 end  
endmodule

Verilog

1. Floprc：clk和rst和clear

`timescale 1ns / 1ps  
  
//带有reset和clear的D触发器  
module floprc(  
 input clk,  
 input rst,  
 input clear,  
  
 input [31:0]d,  
 output reg[31:0]q  
);  
 **always @**(posedge clk) begin  
 if(rst) begin  
 q <= 0;  
 end else if (clear)begin  
 q <= 0;  
 end else begin   
 q <= d;  
 end  
 end  
endmodule

Verilog

1. Flopenr：clk和enable和rst

`timescale 1ns / 1ps  
  
//带有enable、rst的D触发器  
module flopenr(  
 input clk,  
 input rst,  
 input enable,  
  
 input [31:0]d,  
 output reg[31:0]q  
);  
 **always @**(posedge clk) begin  
 if (~rst) begin  
 q<=0;  
 end  
 else if(enable)begin  
 q<=d;  
 end  
 end  
endmodule

Verilog

1. Flopenrc：clk和enable和rst和clear（没用到）

`timescale 1ns / 1ps  
  
module flopenrc(  
 input clk,rst,enable,clear,  
 input [31:0]d,  
 output reg[31:0]q  
);  
 **always @**(posedge clk) begin  
 if (~rst) begin  
 q<=0;  
 end  
 else if (clear) begin  
 q<=0;  
 end  
 else if (enable) begin  
 q<=d;  
 end  
 end  
  
endmodule

Verilog

### 比较器

`timescale 1ns / 1ps  
  
module beq(  
 input [31:0]A,  
 input [31:0]B,  
  
 output isBeq  
);  
 assign isBeq=(A==B)?1:0;  
endmodule

Verilog

### 三选一器

`timescale 1ns / 1ps  
  
module mux3to1(  
 input [31:0]A,  
 input [31:0]B,  
 input [31:0]C,  
 input [1:0]sel,  
  
 output[31:0]res  
);  
 assign res=(sel==2'b00)?A:  
 (sel==2'b01)?B:  
 (sel==2'b10)?C:32'h0;  
endmodule

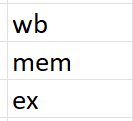
Verilog

### 冒险模块

#### 数据冒险→通过前递技术和阻塞解决RAW

1. 前递技术
   * 介绍

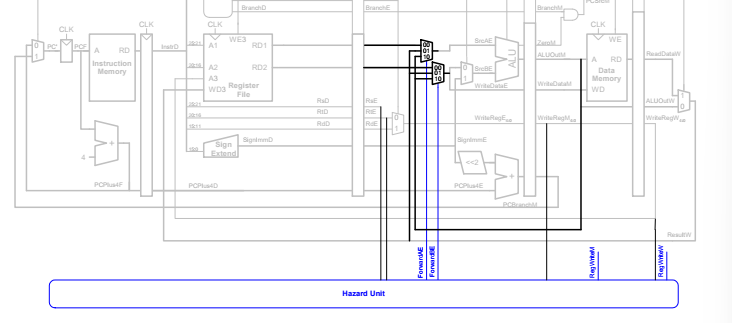
如下图流水所示，*ex*与上一条指令的*mem*、上上条指令的wb冲突，而***ex所读到的操作数可能来自于mem的alu计算结果、wb的访存结果***。**前递技术就是不需要再在*wb*写完寄存器，*ex*才能获取到数据，而是直接将所用到的数据送往ex，即*mem→ex、wb→ex***



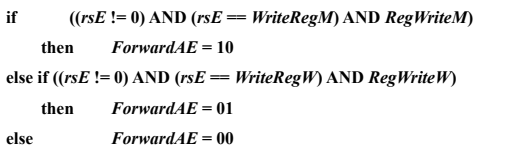
* + 实现

**通过比较当前指令的读寄存器地址*rsE、rtE*和上条指令的访存阶段的写寄存器地址*writeRegM*、上上条写回阶段的写寄存器地址*writeRegW*，若写寄存器标志有效*regWriteM/W*且地址有相同则直接送操作数到srcAE、srcBEtemp**

因此*rsE、rtE、writeRegM、writeRegW、regWriteM、regWriteW*均需送往*hazard阻塞模块*，阻塞模块的输出*forwardAE、forwardBE*（均两位）则送往三选一选择器进行*srcAE、srcBEtemp*的选择



*forwardAE、forwardBE*的产生逻辑如下：



1. 阻塞技术

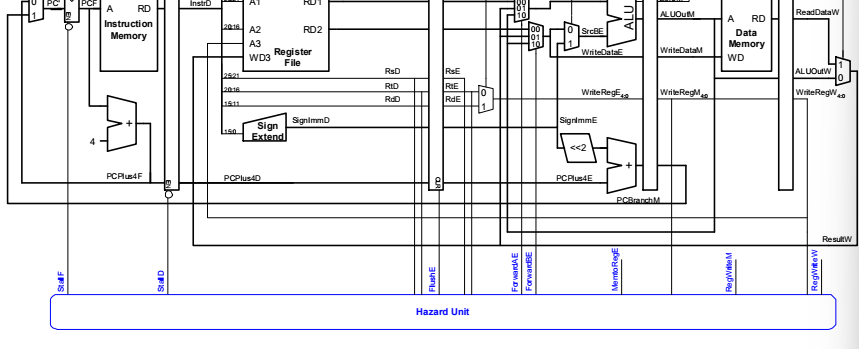
前递技术并不能解决所有的数据冒险，比如“*load延迟*”——如果相邻的上一条指令是load指令，且存在RAW那么此时**必须依靠阻塞或者重新安排指令顺序**来解决“load延迟”

阻塞方法如下：

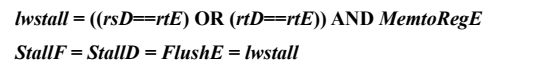
当某指令译码阶段访问寄存器地址***rsD、rtD***与上一条load指令（***memToRegE***有效）的写寄存器地址***writeRegE/rtE***相同时，则下一阶段开始延迟一个时钟周期，如下图：



因此需要把***rsD、rtD、memToRegE、writeRegE*** 送阻塞模块hazard，输出阻塞信号***stallF、stallD、freshE***，如下图



实现逻辑如下：



#### 控制冒险→提前地址的计算和分支的判断→减少分支延迟

|  |  |
| --- | --- |
|  | 分支地址的计算提前到*ID*译码阶段→ |

|  |  |
| --- | --- |
|  | 是否分支的判断提前到*ID*译码阶段→根据*inst[25:21],inst[20:16]*取寄存器操作数比较，结果是否为0的标志送*controler*生成*pcSrcD*（） |

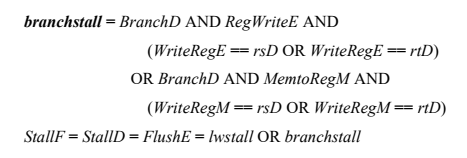
这里*ID*阶段的访问寄存器数同*EX*阶段一样涉及到了*RAW*数据冒险，仍按照前递和阻塞的技术处理如下：

前递*forwardAD/BD*：

|  |  |
| --- | --- |
|  | *EX*阶段的读取结果和*ID*一致，不需要处理。*MEM*阶段的*aluResult*和*WB*阶段的*resultW*则需要处理 |

阻塞*branchStall*：

1. 在*WB*阶段的无论是*alu*写寄存器还是*lw*写寄存器，*ID*阶段都可以无阻塞的进行前递读取结果
2. 如果上一条指令是一个写寄存器指令，无论是lw写还是alu写，均无法前递获得结果，只能通过阻塞，此时的阻塞条件是*regWriteE&(rsD==writeRegE|rtD==writeRegE)*
3. 如果上上条指令是一个写寄存器指令，若是*alu*写寄存器则*aluResult*已可以直接在*MEM*阶段送至*ID*，而若是*lw*写寄存器，则需要阻塞延迟一个时钟周期到*WB*才可前递送达，此时的阻塞条件是*memToRegM&(rsD==writeRegM|rtD==writeRegM)*



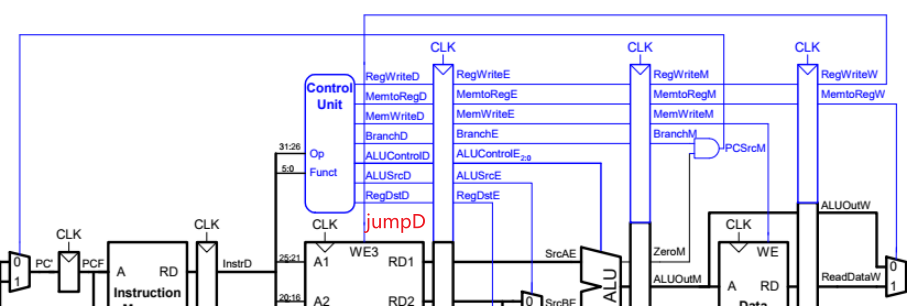
#### 代码实现

`timescale 1ns / 1ps  
  
module hazard(  
 input [4:0]rsD,rtD,rsE,rtE,  
 input memToRegE,memToRegM,  
 input branchD,  
 input regWriteE,regWriteM,regWriteW,  
 input [4:0]writeRegE,writeRegM,writeRegW,  
 output stallF,stallD,freshE,  
  
 output [1:0]forwardAE,forwardBE,  
 output [1:0]forwardAD,forwardBD  
);  
//todo:看AD、BD是否要判断WB；看branchStall是否有WB  
 //wb阶段已经写回去  
 wire lwStall,branchStall;  
 assign forwardAD=((rsD!=0)&(rsD==writeRegM)&regWriteM)?10:  
 ((rsD!=0)&(rsD==writeRegW)&regWriteW)?01:00;  
 assign forwardBD=((rtD!=0)&(rtD==writeRegM)&regWriteM)?10:  
 ((rtD!=0)&(rtD==writeRegW)&regWriteW)?01:00;  
  
 assign forwardAE=((rsE!=0)&(rsE==writeRegM)&regWriteM)?10:  
 ((rsE!=0)&(rsE==writeRegW)&regWriteW)?01:00;  
 assign forwardBE=((rtE!=0)&(rtE==writeRegM)&regWriteM)?10:  
 ((rtE!=0)&(rtE==writeRegW)&regWriteW)?01:00;  
   
 //稍微的处理延迟——便于观察  
 assign #1 lwStall=(rsD==rtE|rtD==rtE)&memToRegE;  
 //在clk下降沿写寄存器，上升沿读——先写后读，所以是1->0->1  
 assign #1 branchStall=branchD&(regWriteE&(rsD==writeRegE|rtD==writeRegE)  
 |memToRegM&(rsD==writeRegM|rtD==writeRegM));  
 assign #1 stallD=lwStall|branchStall;  
 assign #1 stallF=stallD;  
 assign #1 freshE=stallD;  
endmodule

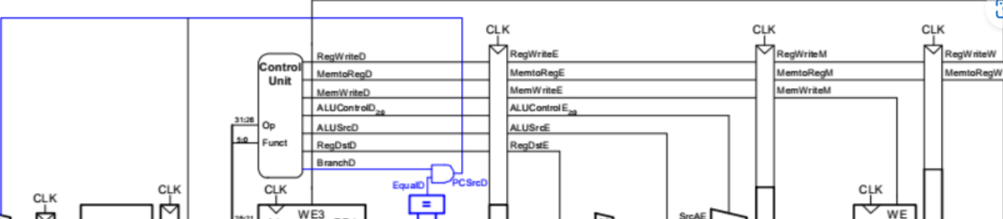
Verilog

## 控制器的集成

控制器集成的方法仍同之前的单周期，但是这里要结合各个流水段，将各个流水段需要的控制信号利用上面的四个触发器传递过去



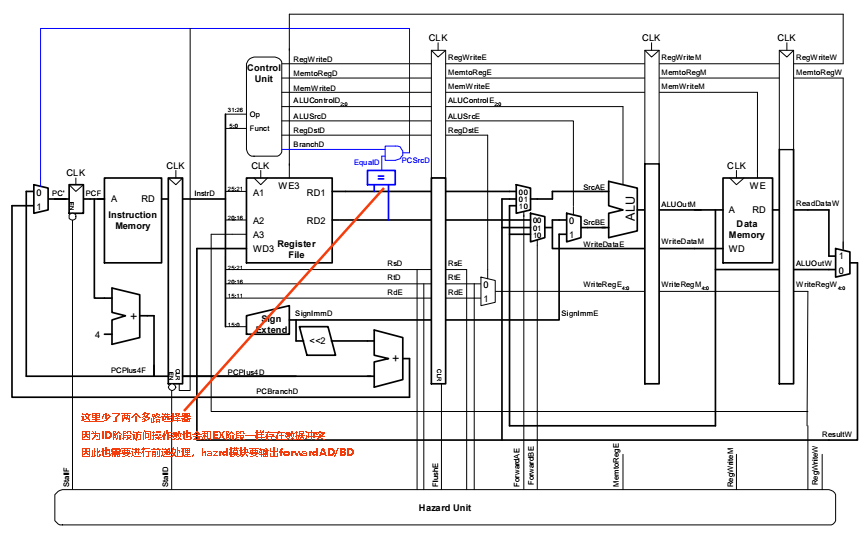
**注意为了缩短分支延迟，将分支的判断提前到ID段，分支地址的计算提前到IF段**



`timescale 1ns/1ps  
  
module controler (  
 input clk,rst,  
 input [5:0]op,alu\_funct,  
 input isBeq,  
 input freshE,//清除译码和执行中间的触发器  
   
 //pc\_next生成的二路选择信号  
 output pcSrcD,jumpD,  
   
 //hazard和datapath一些模块要的  
 output branchD,  
 output memToRegE,memToRegM,  
 output regWriteE,regWriteM,regWriteW,  
  
 //datapath执行阶段需要的控制信号——B操作数的选择、写回寄存器地址的选择、alu做什么运算  
 output alusrcE,regDstE,  
 output [2:0]aluControlE,  
  
 //输出给dataMem的写存信号  
 output memWriteM,  
  
 //输出给写寄存器数的二路选择信号  
 output memToRegW  
);  
 //译码阶段  
 wire memToRegD,memWriteD,alusrcD,regDstD,regWriteD;  
 wire [1:0]aluopD;  
 wire [2:0]aluControlD;  
  
 //执行阶段  
 wire memWriteE;  
  
 mainDecoder mainDecoder\_inst (  
 .op(op),  
 .memToReg(memToRegD),  
 .memWrite(memWriteD),  
 .branch(branchD),  
 .alusrc(alusrcD),  
 .regDst(regDstD),  
 .regWrite(regWriteD),  
 .jump(jumpD),  
 .aluop(aluopD)  
 );  
 aluDecoder aluDecoder\_inst (  
 .aluop(aluopD),  
 .alu\_funct(alu\_funct),  
 .aluControl(aluControlD)  
 );  
 assign pcSrcD=branchD&isBeq;//分支提前判断  
  
 parameter WIDTH\_DE = 8;  
 floprc # (.WIDTH(WIDTH\_DE)) floprc\_DE (  
 .clk(clk),  
 .rst(rst),  
 .clear(freshE),  
 .d({memToRegD,memWriteD,alusrcD,regDstD,regWriteD,aluControlD}),  
 .q({memToRegE,memWriteE,alusrcE,regDstE,regWriteE,aluControlE})  
 );  
 parameter WIDTH\_EM = 3;  
 flopr # (.WIDTH(WIDTH\_EM)) flopr\_EM (  
 .clk(clk),  
 .rst(rst),  
 .d({regWriteE,memToRegE,memWriteE}),  
 .q({regWriteM,memToRegM,memWriteM})  
 );  
 parameter WIDTH\_MW = 2;  
 flopr # (.WIDTH(WIDTH\_MW)) flopr\_MW (  
 .clk(clk),  
 .rst(rst),  
 .d({regWriteM,memToRegM}),  
 .q({regWriteW,memToRegW})  
 );  
endmodule

Verilog

### 数据通路的集成



✍🏽思路：

* 1. 模块定义思路：阻塞模块、产生***pc\_next*、取指、译码、执行、访存、写回**需要外部输入的信号，输出给外部的信号
  2. 定义模块内部变量：按照产生***pc\_next*、取指、译码、执行、访存、写回**的顺序分析
  3. 模块声明：*阻塞模块 hazard*、产生 *pc\_next* (*pcSrcD* 多选器、*jumpD* 多选器)、寄存器堆 *regFile*、取指（*pc* 触发器、*adder4*）、译码（触发器传递 *instrD* 和 *pcPlus4D*、立即数扩展 *signExtend*、左移 *shl2*、*adder* 生成 *pcBranchD*、*forwardAD/BD* 三选一选择器、*beq* 比较器、赋值*op*等信号）、执行（触发器传递 *regReadAE/BE*、*signImmE*、*rsE*、*rtE*、*rdE*，*forwardAE/BE* 三选一选择器， *aluSrcE* 二选一器，*alu*，*regDstE* 二选一器）、访存（触发器传递 *aluOutM*、*writeDataM*、*writeRegM*）、写回（触发器传递 *aluOutW*、*readDataW*、*writeRegW*，*memToReg*W 二选一器）

`timescale 1ns/1ps  
  
module datapath(  
 input clk,rst,  
  
 //阻塞模块  
 input branchD,  
 input memToRegE,memToRegM,  
 input regWriteE,regWriteM,regWriteW,  
 output freshE,  
  
 //产生pc\_next  
 input pcSrcD,jumpD,  
   
 //取指  
 input [31:0]instrF,  
 output [31:0]pcAddr,  
  
 //译码  
 output [5:0]op,alu\_funct,  
 output isBeq,  
   
 //执行  
 input regDstE,alusrcE,  
 input [2:0]aluControlE,  
  
  
 //访存  
 output [31:0]aluOutM,  
 output [31:0]writeDataM,  
 input [31:0]readDataM,  
  
 //写回  
 input memToRegW  
);  
 //产生pc\_next  
 //pcPlus4F pc+4，pcBranchD 分支跳转地址，pcTemp4 前两个地址的选择输出  
 wire [31:0]pcPlus4F,pcBranchD,pcTempF,pc\_next;  
   
 //取指  
 wire stallF;  
   
 //译码  
 wire stallD;  
 wire [1:0]forwardAD,forwardBD;  
 wire [31:0]instrD;  
 wire [31:0]pcPlus4D;  
 wire [31:0]signImmD,signImmD\_L2;  
 wire [4:0]rsD,rtD,rdD;  
 wire [31:0]readRegAD,readRegBD;  
 wire [31:0]beqAD,beqBD;  
  
 //执行  
 wire [1:0]forwardAE,forwardBE;  
 wire [4:0]rsE,rtE,rdE;  
 wire [31:0]readRegAE,readRegBE;  
 wire [31:0]aluOutE,srcAE,srcBE,writeDataE;  
 wire [4:0]writeRegE;  
 wire [31:0]signImmE;  
   
 //访存  
 wire [4:0]writeRegM;  
   
 //写回  
 wire [31:0]readDataW,aluOutW;  
 wire [4:0]writeRegW;  
 wire [31:0]resultW;  
   
 hazard hazard\_inst (//阻塞模块  
 .rsD(rsD),  
 .rtD(rtD),  
 .rsE(rsE),  
 .rtE(rtE),  
 .memToRegE(memToRegE),  
 .memToRegM(memToRegM),  
 .branchD(branchD),  
 .regWriteE(regWriteE),  
 .regWriteM(regWriteM),  
 .regWriteW(regWriteW),  
 .writeRegE(writeRegE),  
 .writeRegM(writeRegM),  
 .writeRegW(writeRegW),  
 .stallF(stallF),  
 .stallD(stallD),  
 .freshE(freshE),  
 .forwardAE(forwardAE),  
 .forwardBE(forwardBE),  
 .forwardAD(forwardAD),  
 .forwardBD(forwardBD)  
 );  
  
 //产生pc\_next  
 mux2to1 mux2to1\_select\_pcTempF (  
 .A(pcPlus4F),  
 .B(pcBranchD),  
 .sel(pcSrcD),  
 .res(pcTempF)  
 );  
 mux2to1 mux2to1\_select\_pcNext (  
 .A(pcTempF),  
 .B({pcPlus4D[31:28],instrD[25:0],2'b00}),//原pcAdder4后的高4位，指令的低26位左移两位(28位)  
 .sel(jumpD),  
 .res(pc\_next)  
 );  
  
 //寄存器模块  
 regfile regfile\_inst (  
 .clk(clk),  
 .regWrite(regWriteW),  
 .addrA(rsD),  
 .addrB(rtD),  
 .addrC(writeRegW),  
 .writeData(resultW),  
 .readDataA(readRegAD),  
 .readDataB(readRegBD)  
 );  
  
 //取指  
 pc pc\_inst (  
 .clk(clk),  
 .rst(rst),  
 .en(~stallF),  
 .d(pc\_next),  
 .q(pcAddr)  
 );  
 adder adder\_pcPlusF (  
 .a(pcAddr),  
 .b(32'h4),  
 .y(pcPlus4F)  
 );  
  
 //译码  
 //触发器传递  
 parameter WIDTH\_FD = 32;  
 flopenr # (.WIDTH(WIDTH\_FD)) flopenr\_ID2 (  
 .clk(clk),  
 .rst(rst),  
 .enable(~stallD),  
 .d(pcPlus4F),  
 .q(pcPlus4D)  
 );  
 flopenr # (.WIDTH(WIDTH\_FD)) flopenr\_ID1 (  
 .clk(clk),  
 .rst(rst),  
 .enable(~stallD),  
 .d(instrF),  
 .q(instrD)  
 );  
   
 //符号扩展->产生pcBranchD  
 signExtend signExtend\_instD (  
 .d(instrD[15:0]),  
 .q(signImmD)  
 );  
 shl2 shl2\_instD (  
 .a(signImmD),  
 .b(signImmD\_L2)  
 );  
 adder adder\_pcBranchD (  
 .a(signImmD\_L2),  
 .b(pcPlus4D),  
 .y(pcBranchD)  
 );  
  
 //比较器输出isBeq  
 mux3to1 mux2to1\_select\_beqAD (  
 .A(readRegAD),  
 .B(resultW),  
 .C(aluOutM),  
 .sel(forwardAD),  
 .res(beqAD)  
 );  
 mux3to1 mux2to1\_select\_beqBD (  
 .A(readRegBD),  
 .B(resultW),  
 .C(aluOutM),  
 .sel(forwardBD),  
 .res(beqBD)  
 );  
 beq beq\_instD (  
 .A(beqAD),  
 .B(beqBD),  
 .isBeq(isBeq)  
 );  
  
 //基本变量赋值  
 assign op=instrD[31:26];  
 assign alu\_funct=instrD[5:0];  
 assign rsD=instrD[25:21];  
 assign rtD=instrD[20:16];  
 assign rdD=instrD[15:11];  
   
 //执行  
 //触发器传递  
 parameter WIDTH\_EX\_DATA = 32;  
 floprc # (.WIDTH(WIDTH\_EX\_DATA)) floprc\_EX1 (  
 .clk(clk),  
 .rst(rst),  
 .clear(freshE),  
 .d(readRegAD),  
 .q(readRegAE)  
 );  
 floprc # (.WIDTH(WIDTH\_EX\_DATA)) floprc\_EX2 (  
 .clk(clk),  
 .rst(rst),  
 .clear(freshE),  
 .d(readRegBD),  
 .q(readRegBE)  
 );  
 floprc # (.WIDTH(WIDTH\_EX\_DATA)) floprc\_EX3 (  
 .clk(clk),  
 .rst(rst),  
 .clear(freshE),  
 .d(signImmD),  
 .q(signImmE)  
 );  
 parameter WIDTH\_EX\_RegAddr = 15;  
 floprc # (.WIDTH(WIDTH\_EX\_RegAddr)) floprc\_EX4 (  
 .clk(clk),  
 .rst(rst),  
 .clear(freshE),  
 .d({rsD,rtD,rdD}),  
 .q({rsE,rtE,rdE})  
 );  
   
 //SrcAE、SrcBE产生并进行ALU运算  
 mux3to1 mux3to1\_srcAE (  
 .A(readRegAE),  
 .B(resultW),  
 .C(aluOutM),  
 .sel(forwardAE),  
 .res(srcAE)  
 );  
 mux3to1 mux3to1\_WDE (  
 .A(readRegBE),  
 .B(resultW),  
 .C(aluOutM),  
 .sel(forwardBE),  
 .res(writeDataE)  
 );  
 mux2to1 mux2to1\_srcBE (  
 .A(writeDataE),  
 .B(signImmE),  
 .sel(alusrcE),  
 .res(srcBE)  
 );  
 alu alu\_inst (  
 .aluControl(aluControlE),  
 .A(srcAE),  
 .B(srcBE),  
 .aluResult(aluOutE)  
 );  
  
 //生成写寄存器地址  
 mux2to1 mux2to1\_WRegAddr (  
 .A(rtE),  
 .B(rdE),  
 .sel(regDstE),  
 .res(writeRegE)  
 );  
  
 //访存  
 //触发器传递  
 parameter WIDTH\_MEM\_DATA= 32;  
 flopr # (.WIDTH(WIDTH\_MEM\_DATA)) flopr\_MEM1 (  
 .clk(clk),  
 .rst(rst),  
 .d(aluOutE),  
 .q(aluOutM)  
 );  
 flopr # (.WIDTH(WIDTH\_MEM\_DATA)) flopr\_MEM2 (  
 .clk(clk),  
 .rst(rst),  
 .d(writeDataE),  
 .q(writeDataM)  
 );  
 parameter WIDTH\_MEM\_RegAddr = 5;  
 flopr # (.WIDTH(WIDTH\_MEM\_RegAddr)) flopr\_MEM3 (  
 .clk(clk),  
 .rst(rst),  
 .d(writeRegE),  
 .q(writeRegM)  
 );  
  
 //写回  
 //触发器传递  
 parameter WIDTH\_WB\_DATA= 32;  
 flopr # (.WIDTH(WIDTH\_WB\_DATA)) flopr\_WB1 (  
 .clk(clk),  
 .rst(rst),  
 .d(aluOutM),  
 .q(aluOutW)  
 );  
 flopr # (.WIDTH(WIDTH\_WB\_DATA)) flopr\_WB2 (  
 .clk(clk),  
 .rst(rst),  
 .d(readDataM),  
 .q(readDataW)  
 );  
 parameter WIDTH\_WB\_RegAddr = 5;  
 flopr # (.WIDTH(WIDTH\_WB\_RegAddr)) flopr\_WB3 (  
 .clk(clk),  
 .rst(rst),  
 .d(writeRegM),  
 .q(writeRegW)  
 );  
  
 //产生写寄存器结果  
 mux2to1 mux2to1\_WRegRes (  
 .A(aluOutW),  
 .B(readDataW),  
 .sel(memToRegW),  
 .res(resultW)  
 );  
endmodule

Verilog

### MIPS模块、顶层模块

`timescale 1ns / 1ps  
  
module mips(  
 input clk,rst,  
  
 input [31:0]instrF,readDataM,  
 output [31:0]pcAddr,aluOutM,writeDataM,  
 output memWriteM  
);  
 wire branchD,memToRegE,memToRegM,memToRegW,regWriteE,regWriteM,regWriteW;  
 wire pcSrcD,jumpD,isBeq,regDstE,alusrcE,freshE;  
 wire [5:0]op,alu\_funct;  
 wire [2:0]aluControlE;  
   
 controler controler\_inst (  
 .clk(clk),  
 .rst(rst),  
 .op(op),  
 .alu\_funct(alu\_funct),  
 .isBeq(isBeq),  
 .freshE(freshE),  
 .pcSrcD(pcSrcD),  
 .jumpD(jumpD),  
 .branchD(branchD),  
 .memToRegE(memToRegE),  
 .memToRegM(memToRegM),  
 .regWriteE(regWriteE),  
 .regWriteM(regWriteM),  
 .regWriteW(regWriteW),  
 .alusrcE(alusrcE),  
 .regDstE(regDstE),  
 .aluControlE(aluControlE),  
 .memWriteM(memWriteM),  
 .memToRegW(memToRegW)  
 );  
  
 datapath datapath\_inst (  
 .clk(clk),  
 .rst(rst),  
 .branchD(branchD),  
 .memToRegE(memToRegE),  
 .memToRegM(memToRegM),  
 .regWriteE(regWriteE),  
 .regWriteM(regWriteM),  
 .regWriteW(regWriteW),  
 .freshE(freshE),  
 .pcSrcD(pcSrcD),  
 .jumpD(jumpD),  
 .instrF(instrF),  
 .pcAddr(pcAddr),  
 .op(op),  
 .alu\_funct(alu\_funct),  
 .isBeq(isBeq),  
 .regDstE(regDstE),  
 .alusrcE(alusrcE),  
 .aluControlE(aluControlE),  
 .aluOutM(aluOutM),  
 .writeDataM(writeDataM),  
 .readDataM(readDataM),  
 .memToRegW(memToRegW)  
 );  
endmodule

Verilog

`timescale 1ns/1ps  
  
module top (  
 input clk,rst,  
 output [31:0]instrF  
);  
 wire [31:0]readDataM;  
 wire [31:0]pcAddr,aluOutM,writeDataM;  
 wire memWriteM;  
 mips mips\_inst (  
 .clk(clk),  
 .rst(rst),  
 .instrF(instrF),  
 .readDataM(readDataM),  
 .pcAddr(pcAddr),  
 .aluOutM(aluOutM),  
 .writeDataM(writeDataM),  
 .memWriteM(memWriteM)  
 );  
 instMem instMem\_inst(//clk下降沿读  
 .clka(~clk),  
 .addra(pcAddr),  
 .douta(instrF)  
 );  
 dataMem dataMem\_inst(//clk下降沿写存  
 .clka(~clk),  
 .wea(memWriteM),  
 .addra(aluOutM),  
 .dina(writeDataM),  
 .douta(readDataM)  
 );  
endmodule

Verilog