# 流水线CPU

余超逸 2018160701013

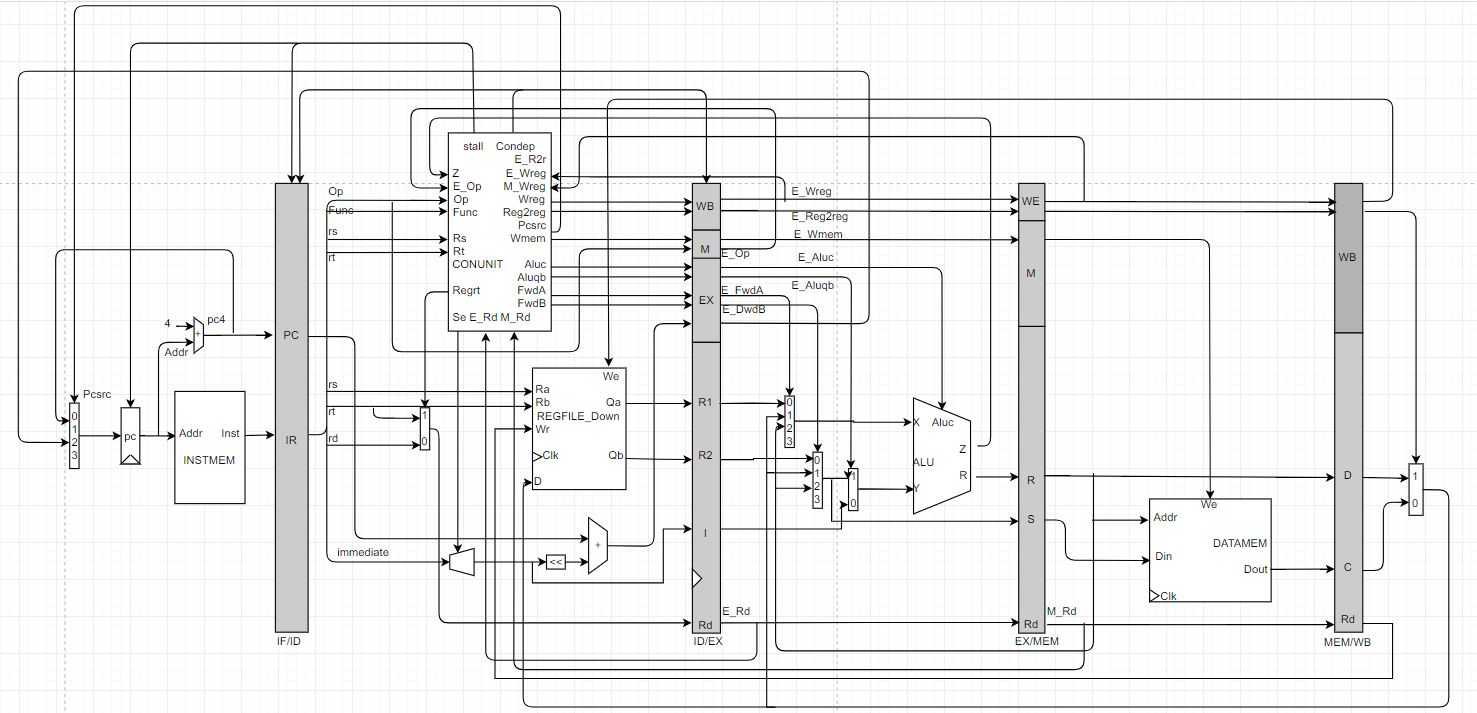
刘常乐 2018010801026

## 一、原理

### 1.流水线CPU

设计并实现一个具有数据前推机制的流水线CPU，具有处理数据冒险和控制冒险的功能。流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于高档CPU的架构中。根据MIPS处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级，对应多周期的五个处理阶段。一个指令的执行需要5个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

其总体框图如下（来自网络资料）



上图是一个能够在流水线上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令储存在指令储存器，数据储存在数据存储器。而且实现了数据冒险和控制冒险的解决方案。

根据实验原理中的流水线CPU总体结构图，我们可以清楚的知道流水线CPU的设计应包括五级。其中为了运行整个CPU还需要加入一个封装模块（MAIN）来调用这些模块，所以自然地，这些模块为封装模块MAIN的子模块。

### 2.冒险的解决

#### 控制冒险

在基本流水线中相邻两条指令的前一条指令还没有更新目的寄存器时，后一条指令就已经先读取了该寄存器的旧值，使得指令的计算结果出现错误。这样相关的问题就称为数据冒险。下面给出本实验所使用的解决方法。

##### 寄存器堆的写操作提前半个时钟周期

在未提前半周期时CPU中，指令在时钟周期结束时的上升沿将ID级寄存器的值锁存进ID/EX，或将ALU的计算结果更新寄存器的值。但是寄存器的读、写的操作时间实际上只有时钟周期的一半。因此可以把寄存器堆的写操作提前到时钟周期中间的下降沿完成。那么后半个时钟周期就可以将写入之后的值读出。这样做后，同一级的数据冒险得到解决。

##### 内部前推

运算指令的结果在EX级结束时，就已经锁存在EX/MEM的Rd中，然而寄存器的值在ID结束时就已经锁存在ID/EX，但在EX级才真正使用这些值。

所以为了支持内部前推，需在ALU的两个输入端之前，分别增加一个多路选择器和相应的数据通路，并检测处于EX级指令的两个源操作寄存器号是否和处于MEM级或WB级指令的目的寄存器号相等。下面给出检测条件。

[条件a] E\_Rs == EX/MEM.Rd，

判断EX级指令的rs字段是否和MEM级指令的目的寄存器号相同

[条件b] E\_Rs == MEM/WB.Rd，

判断EX级指令的rs字段是否和WB级指令的目的寄存器号相同

[条件c] (E\_Rt == EX/MEM.Rd) & ((E\_Inst == I\_add) | (E\_Inst == I\_sub) | (E\_Inst == I\_and) | (E\_Inst == I\_or) | (E\_Inst == sw) | (E\_Inst == beq) | (E\_Inst == bne))

判断EX级指令的rt字段是否和MEM级指令的目的寄存器号相同

[条件d] (E\_Rt == MEM/WB.Rd) & ((E\_Inst == I\_add) | (E\_Inst == I\_sub) | (E\_Inst == I\_and) | (E\_Inst == I\_or) | (E\_Inst == sw) | (E\_Inst == beq) | (E\_Inst == bne))

判断EX级指令的rt字段是否和WB级指令的目的寄存器号相同

某些指令可能不写回寄存器，例如sw和beq指令，或者某些指令的写信号被关闭。所以还需检测处于MEM级或WB级指令的寄存器堆写使能信号M\_Wreg或W\_Wreg是否有效。

寄存器$0始终为0，不必考虑在$0上产生的数据冒险，即第三条指令分别与第一条、第二条指令存在数据冒险。按照执行逻辑，当第三条指令处于EX级时应选择处于MEM级的第二条指令的前推。而不能选择第一条前推。所以在判断逻辑模块的代码实现时，应先判断相邻两条指令是否存在数据冒险。

##### LW指令的数据冒险

内部前推有显而易见的局限性，因为内部前推必须要求前一条指令在EX结束时更新，但是LW指令最早只能在WB级读出寄存器的值。因此无法及时提供给下一条指令的EX级使用。

分析流水线时序图，可以发现lw指令的下一条指令，需要阻塞一个时钟周期，才能确保该指令能获得正确的操作数值，下面给出具体解决方法。

检测是否存在lw指令的数据冒险。

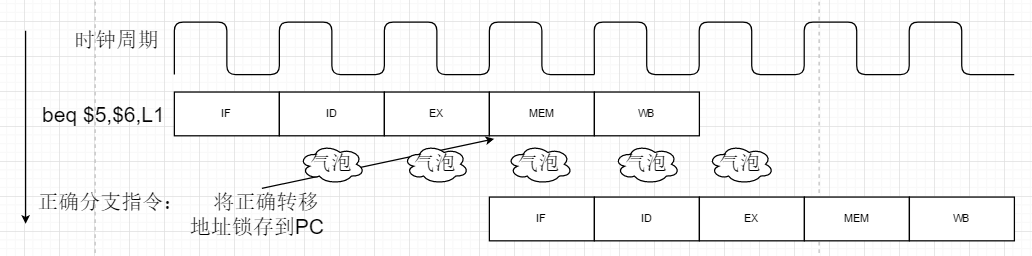
检测单元仍放置于CONUNIT部件内，且Reg2reg信号可以唯一区分lw指令和其他指令。检测条件可写为：

((Rs == E\_Rd) | (Rt == E\_Rd)) & (E\_Reg2reg == 0) & (E\_Rd != 0) & (E\_Wreg == 1)

#### 控制冒险

在处理beq和bne指令时，条件分支指令或者跳转指令的后续指令有可能在目标地址形成之前或分支条件形成之前就已经进入流水线。这样相关的问题就称为控制冒险。下面给出本实验所使用的解决方法。

缩短分支的延迟：



在条件分支指令处于EX级时，判断分支条件是否成立

若成立，则控制部件CONUNIT的STALL端口（流水线阻塞）输出高电平，IF/ID和ID/EX流水线寄存器组的Clrn端口输入为低电平

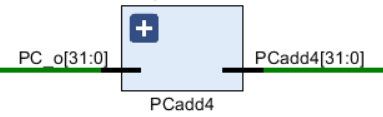
在该时钟周期结束时，IF/ID和ID/EX流水线寄存器组的内容清0，即变为两条空指令

这样在下一个时钟周期开始时，正确的目标指令处于IF级，ID级和EX级都是编码为全0的空指令，条件分支指令进入到MEM级

## 二、代码

### 1.取指（IF）部分

#### 1.1 PCAdd4



**模块功能**

作为PC寄存器的更新信号之一，PC地址+4

**引脚及控制信号**

IF\_Addr:当前指令地址，输入端口(IF内循环)

IF\_PCadd4:下一条指令地址，输出端口

主要实现代码

module PCadd4(PC\_o,PCadd4);

input [31:0] PC\_o;

//偏移量

output [31:0] PCadd4;

//新指令地址

CLA\_32 cla32(PC\_o,4,0, PCadd4, Cout);

endmodule

子模块 CLA\_32

module CLA\_32(X,Y,Cin,S,Cout);

input [31:0]X,Y;

input Cin;

output [31:0]S;

output Cout;

wire Cout0,Cout1,Cout2,Cout3,Cout4,Cout5,Cout6;

CLA\_4 add0(X[3:0],Y[3:0],Cin,S[3:0],Cout0);

CLA\_4 add1(X[7:4],Y[7:4],Cout0,S[7:4],Cout1);

CLA\_4 add2(X[11:8],Y[11:8],Cout1,S[11:8],Cout2);

CLA\_4 add3(X[15:12],Y[15:12],Cout2,S[15:12],Cout3);

CLA\_4 add4(X[19:16],Y[19:16],Cout3,S[19:16],Cout4);

CLA\_4 add5(X[23:20],Y[23:20],Cout4,S[23:20],Cout5);

CLA\_4 add6(X[27:24],Y[27:24],Cout5,S[27:24],Cout6);

CLA\_4 add7(X[31:28],Y[31:28],Cout6,S[31:28],Cout);

endmodule

子模块 CLA\_4

module CLA\_4(X,Y,Cin,S,Cout);

input [3:0]X,Y;

input Cin;

output Cout;

output [3:0]S;

wire [3:0]A,B,C;

wire [16:0]D;

and (A[0],Y[3],X[3]);

and (A[1],Y[2],X[2]);

and (A[2],Y[1],X[1]);

and (A[3],Y[0],X[0]);

or (B[0],Y[3],X[3]);

or (B[1],Y[2],X[2]);

or (B[2],Y[1],X[1]);

or (B[3],Y[0],X[0]);

not (C[0],A[0]);

not (C[1],A[1]);

not (C[2],A[2]);

not (C[3],A[3]);

//初始值

nand (D[1],B[0],A[1]);

nand (D[2],B[0],B[1],A[2]);

nand (D[3],B[0],B[1],B[2],A[3]);

nand (D[4],B[0],B[1],B[2],B[3],Cin);

nand (Cout,C[0],D[1],D[2],D[3],D[4]);

//Cout运算

and (D[5],C[0],B[0]);

nand (D[6],B[1],A[2]);

nand (D[7],B[1],B[2],A[3]);

nand (D[8],B[1],B[2],B[3],Cin);

nand (D[9],C[1],D[6],D[7],D[8]);

xor (S[3],D[5],D[9]);

//S[3]运算

and (D[10],C[1],B[1]);

nand (D[11],B[2],A[3]);

nand (D[12],B[2],B[3],Cin);

nand (D[13],C[2],D[11],D[12]);

xor (S[2],D[10],D[13]);

//S[2]运算

and (D[14],C[2],B[2]);

nand (D[15],B[3],Cin);

nand (D[16],C[3],D[15]);

xor (S[1],D[14],D[16]);

//S[1]运算

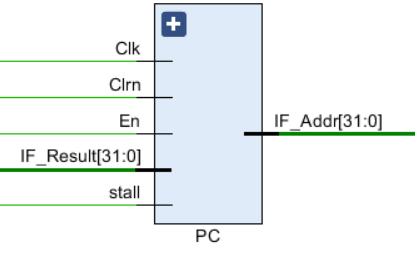
and (D[0],C[3],B[3]);

xor (S[0],D[0],Cin);

//S[0]运算

endmodule

#### 1.2 PC



**模块功能**

用于给出指令在指令储存器中的地址，且当发生数据冒险时，需要保持PC寄存器不变。

**引脚及控制信号**

En:控制PC寄存器的开启和关闭状态，外部输入信号。

stall:判断是否发生Lw指令数据冒险，输入信号。

Clk：时钟周期，输入信号

Clrn：当存在Lw指令的数据冒险和控制冒险时为零，输入信号。

IF\_Result目标地址，输入信号。

IF\_Addr：指令地址，输出信号。

**主要实现代码**

module PC(IF\_Result,Clk,En,Clrn,IF\_Addr,stall);

input [31:0]IF\_Result;

input Clk,En,Clrn,stall;

output [31:0] IF\_Addr;

wire En\_S;

assign En\_S=En&~stall;

D\_FFEC32 pc(IF\_Result,Clk,En\_S,Clrn,IF\_Addr);

endmodule

子模块D\_FFEC32

module D\_FFEC32(D,Clk,En,Clrn,Q,Qn);

input [31:0]D;

input Clk,En,Clrn;

output [31:0]Q,Qn;

D\_FFEC d0(D[0],Clk,En,Clrn,Q[0],Qn[0]);

D\_FFEC d1(D[1],Clk,En,Clrn,Q[1],Qn[1]);

D\_FFEC d2(D[2],Clk,En,Clrn,Q[2],Qn[2]);

D\_FFEC d3(D[3],Clk,En,Clrn,Q[3],Qn[3]);

D\_FFEC d4(D[4],Clk,En,Clrn,Q[4],Qn[4]);

D\_FFEC d5(D[5],Clk,En,Clrn,Q[5],Qn[5]);

D\_FFEC d6(D[6],Clk,En,Clrn,Q[6],Qn[6]);

D\_FFEC d7(D[7],Clk,En,Clrn,Q[7],Qn[7]);

D\_FFEC d8(D[8],Clk,En,Clrn,Q[8],Qn[8]);

D\_FFEC d9(D[9],Clk,En,Clrn,Q[9],Qn[9]);

D\_FFEC d10(D[10],Clk,En,Clrn,Q[10],Qn[10]);

D\_FFEC d11(D[11],Clk,En,Clrn,Q[11],Qn[11]);

D\_FFEC d12(D[12],Clk,En,Clrn,Q[12],Qn[12]);

D\_FFEC d13(D[13],Clk,En,Clrn,Q[13],Qn[13]);

D\_FFEC d14(D[14],Clk,En,Clrn,Q[14],Qn[14]);

D\_FFEC d15(D[15],Clk,En,Clrn,Q[15],Qn[15]);

D\_FFEC d16(D[16],Clk,En,Clrn,Q[16],Qn[16]);

D\_FFEC d17(D[17],Clk,En,Clrn,Q[17],Qn[17]);

D\_FFEC d18(D[18],Clk,En,Clrn,Q[18],Qn[18]);

D\_FFEC d19(D[19],Clk,En,Clrn,Q[19],Qn[19]);

D\_FFEC d20(D[20],Clk,En,Clrn,Q[20],Qn[20]);

D\_FFEC d21(D[21],Clk,En,Clrn,Q[21],Qn[21]);

D\_FFEC d22(D[22],Clk,En,Clrn,Q[22],Qn[22]);

D\_FFEC d23(D[23],Clk,En,Clrn,Q[23],Qn[23]);

D\_FFEC d24(D[24],Clk,En,Clrn,Q[24],Qn[24]);

D\_FFEC d25(D[25],Clk,En,Clrn,Q[25],Qn[25]);

D\_FFEC d26(D[26],Clk,En,Clrn,Q[26],Qn[26]);

D\_FFEC d27(D[27],Clk,En,Clrn,Q[27],Qn[27]);

D\_FFEC d28(D[28],Clk,En,Clrn,Q[28],Qn[28]);

D\_FFEC d29(D[29],Clk,En,Clrn,Q[29],Qn[29]);

D\_FFEC d30(D[30],Clk,En,Clrn,Q[30],Qn[30]);

D\_FFEC d31(D[31],Clk,En,Clrn,Q[31],Qn[31]);

endmodule

子模块 D\_FFEC

module D\_FFEC(D,Clk,En,Clrn,Q,Qn);

input D,Clk,En,Clrn;

output Q,Qn;

wire Y0,Y\_C;

MUX2X1 m0(Q,D,En,Y0);

and i0(Y\_C,Y0,Clrn);

D\_FF d0(Y\_C,Clk,Q,Qn);

endmodule

子模块 D\_FF

module D\_FF(D,Clk,Q,Qn);

input D,Clk;

output Q,Qn;

wire Clkn,Q0,Qn0;

not i0(Clkn,Clk);

D\_Latch d0(D,Clkn,Q0,Qn0);

//主锁存器

D\_Latch d1(Q0,Clk,Q,Qn);

//从锁存器

endmodule

子模块 D\_Latch

module D\_Latch(D,En,Q,Qn);

input D,En;

output Q,Qn;

wire Sn,Rn,Dn;

not i0(Dn,D);

nand i1(Sn,D,En);

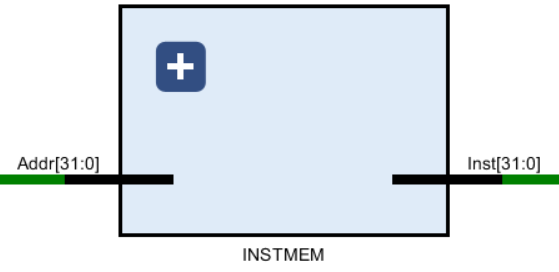
nand i2(Rn,En,Dn);

nand i3(Q,Sn,Qn);

nand i4(Qn,Q,Rn);

endmodule

#### 1.3 INSTMEM



**模块功能**

依据当前pc，读取指令寄存器中相对应地址Addr[6:2]的指令。

**引脚及控制信号**

IF\_Addr：指令地址，输入信号

IF\_Inst：指令编码，输出信号

**主要实现代码**

module INSTMEM(Addr,Inst);

input [31:0]Addr;

output [31:0]Inst;

wire [31:0]Rom[31:0];

assign Rom[5'h00]=32'h20010008;

//addi $1,$0,8 $1=8//001000 00000 00001 0000000000001000

assign Rom[5'h01]=32'h3402000C;

//ori $2,$0,12 $2=12//00110100000000100000000000001100

assign Rom[5'h02]=32'h00221820;//add $3,$1,$2 $3=20//数据冒险

assign Rom[5'h03]=32'h00412022;

//sub $4,$2,$1 $4=4//00000000010000010010000000100010

assign Rom[5'h04]=32'h00222824;//and $5,$1,$2

assign Rom[5'h05]=32'h00223025;//or $6,$1,$2

assign Rom[5'h06]=32'h14220006;

//bne $1,$2,6//00010100001000010000000000000110

assign Rom[5'h07]=32'h00221820;//add $3,$1,$2 $3=20

assign Rom[5'h08]=32'h00412022;//sub $4,$2,$1 $4=4

assign Rom[5'h09]=32'h10220002;// beq $1,$2,2

assign Rom[5'h0A]=32'h0800000D;// J 0D

assign Rom[5'h0B]=32'hXXXXXXXX;

assign Rom[5'h0C]=32'hXXXXXXXX;

assign Rom[5'h0D]=32'hAD02000A;// sw $2 10($8) memory[$8+10]=12

assign Rom[5'h0E]=32'h8D04000A;//lw $4 10($8) $4=12

assign Rom[5'h0F]=32'h10440002;//beq $2,$4,2//lw数据冒险

assign Rom[5'h10]=32'h20210004;

//addi $1,$1,4 //00100000001000010000000000000100

assign Rom[5'h11]=32'h00222824;//and $5,$1,$2

assign Rom[5'h12]=32'h14220006;//bne $1,$2,6

assign Rom[5'h13]=32'h30470009;//andi $2,9,$7//控制冒险

assign Rom[5'h14]=32'hXXXXXXXX;

assign Rom[5'h15]=32'hXXXXXXXX;

assign Rom[5'h16]=32'hXXXXXXXX;

assign Rom[5'h17]=32'hXXXXXXXX;

assign Rom[5'h18]=32'hXXXXXXXX;

assign Rom[5'h19]=32'hXXXXXXXX;

assign Rom[5'h1A]=32'hXXXXXXXX;

assign Rom[5'h1B]=32'hXXXXXXXX;

assign Rom[5'h1C]=32'hXXXXXXXX;

assign Rom[5'h1D]=32'hXXXXXXXX;

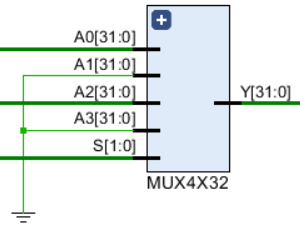
assign Rom[5'h1E]=32'hXXXXXXXX;

assign Rom[5'h1F]=32'hXXXXXXXX;

assign Inst=Rom[Addr[6:2]];

endmodule

#### 1.4 MUX4X32



**模块功能**

实现目标地址的选择

**引脚及控制信号**

F\_PCadd4：PC+4的地址，输入信号。

0：空位，输入信号。

mux4x32\_2：beq和bne指令的跳转地址，输入信号。

0：空位，输入信号。（本是）

Pcsrc：对地址进行选择的控制信号，输入信号

IF\_Result：目标地址，输出信号

**主要实现代码**

module MUX4X32 (A0, A1, A2, A3, S, Y);

input [31:0] A0, A1, A2, A3;

input [1:0] S;

output [31:0] Y;

function [31:0] select;

input [31:0] A0, A1, A2, A3;

input [1:0] S;

case(S)

2'b00: select = A0;

2'b01: select = A1;

2'b10: select = A2;

2'b11: select = A3;

endcase

endfunction

assign Y = select (A0, A1, A2, A3, S);

endmodule

#### 1.5 REG\_IF\_ID

**模块功能**

寄存IF级的输出指令，分割IF级和ID级的指令或控制信号，防止相互干扰，在IF级执行结束时将指令的控制信号传递至下一级

**引脚及控制信号**

IF\_PCadd4：下一条指令地址，输入信号。

IF\_Inst：本条指令编码，输入信号。

En：写使能信号，输入信号。

Clk：时钟信号，输入信号。

Clrn：清零信号，输入信号。

ID\_PCadd4：下条指令地址，输出信号。

ID\_Inst：本条指令编码，输出信号。

Stall：记录是否存在Lw数据冒险，输出信号。

condep：记录是否存在控制冒险，输出信号。

**主要实现代码**

module REG\_IF\_ID(D0,D1,En,Clk,Clrn,Q0,Q1,stall,condep);

input [31:0] D0,D1;

input En,Clk,Clrn;

input stall,condep;

output [31:0] Q0,Q1;

wire En\_S,Clrn\_C;

assign En\_S=En&~stall;

assign Clrn\_C=Clrn&~condep;

D\_FFEC32 q0(D0,Clk,En\_S,Clrn\_C,Q0);

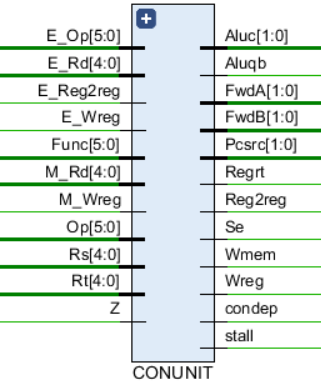
D\_FFEC32 q1(D1,Clk,En\_S,Clrn\_C,Q1);

endmodule

子模块同PC寄存器

### 2. 译码（ID）部分

#### 2.1 CONUNIT



**模块功能**

CONUNIT是作为CPU控制信号产生的器件，通过通过解析op得到该指令的各种控制信号，使其他器件有效或无效或选择输入信号。

**引脚及控制信号**

M\_Op;MEM级指令的OP字段，输入信号。

M\_Z;MEM级指令的Z值。stall

ID\_Inst[31:26]：Op，输入信号。

ID\_Inst[5:0]：Func，输入信号。

Z：零标志信号，对Pcsrc有影响，输入信号。

Regrt：控制输入寄存器的Wr端口，输出信号。

Se：控制扩展模块，输出信号。

Wreg：控制寄存器端的写使能信号，输出信号。

Aluqb：控制ALU的Y端口的输入值，输出信号。

Aluc：控制ALU的计算种类，输出信号。

Wmem：控制数据存储器的写使能信号，输出信号。

Pcsrc：控制目标指令地址，输出信号。

Reg2reg：控制REHFILE更新值的来源，输出信号。

ID\_Inst[25:21]：ID指令的rs字段，输入信号。

ID\_Inst[20:16]：ID指令的rt字段，输入信号。

E\_Rd[4:0]：EX级输出的目的寄存器号，输入信号。

M\_Rd[4:0]：MEM级输出的目的寄存器号，输入信号。

E\_Wreg：EX级的写使能信号，输入信号。

M\_Wreg：MEM级写使能信号，输入信号。

FwdA[1:0]：判断是否在rs寄存器发生数据冒险，输出信号。

FwdB[1:0]：判断是否在rt寄存器发生数据冒险，输出信号。

E：写使能信号，外部输入信号

**主要实现代码**

module CONUNIT(E\_Op,Op,Func,Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,Reg2reg,Rs,Rt,E\_Rd,M\_Rd,E\_Wreg,M\_Wreg,FwdA,FwdB,E\_Reg2reg,stall,condep);

input [5:0]Op,Func,E\_Op;

input Z;

input E\_Wreg,M\_Wreg,E\_Reg2reg;

input [4:0]E\_Rd,M\_Rd,Rs,Rt;

output Regrt,Se,Wreg,Aluqb,Wmem,Reg2reg,stall,condep;

output [1:0]Pcsrc,Aluc;

output reg [1:0]FwdA,FwdB;

wire R\_type=~|Op;

wire I\_add=R\_type&Func[5]&~Func[4]&~Func[3]&~Func[2]&~Func[1]&~Func[0];

wire I\_sub=R\_type&Func[5]&~Func[4]&~Func[3]&~Func[2]&Func[1]&~Func[0];

wire I\_and=R\_type&Func[5]&~Func[4]&~Func[3]&Func[2]&~Func[1]&~Func[0];

wire I\_or=R\_type&Func[5]&~Func[4]&~Func[3]&Func[2]&~Func[1]&Func[0];

wire I\_addi=~Op[5]&~Op[4]&Op[3]&~Op[2]&~Op[1]&~Op[0];

wire I\_andi=~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&~Op[0];

wire I\_ori=~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&Op[0];

wire I\_lw=Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];

wire I\_sw=Op[5]&~Op[4]&Op[3]&~Op[2]&Op[1]&Op[0];

wire I\_beq=~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&~Op[0];

wire I\_bne=~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&Op[0];

wire E\_beq=~E\_Op[5]&~E\_Op[4]&~E\_Op[3]&E\_Op[2]&~E\_Op[1]&~E\_Op[0];

wire E\_bne=~E\_Op[5]&~E\_Op[4]&~E\_Op[3]&E\_Op[2]&~E\_Op[1]&E\_Op[0];

wire I\_J=~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&~Op[0];

wire E\_Inst = I\_add|I\_sub|I\_and|I\_or|I\_sw|I\_beq|I\_bne;

assign Regrt = I\_addi|I\_andi|I\_ori|I\_lw|I\_sw|I\_beq|I\_bne|I\_J;

assign Se = I\_addi|I\_lw|I\_sw|I\_beq|I\_bne;

assign Wreg = I\_add|I\_sub|I\_and|I\_or|I\_addi|I\_andi|I\_ori|I\_lw;

assign Aluqb = I\_add|I\_sub|I\_and|I\_or|I\_beq|I\_bne|I\_J;

assign Aluc[1] = I\_and|I\_or|I\_andi|I\_ori;

assign Aluc[0] = I\_sub|I\_or|I\_ori|I\_beq|I\_bne;

assign Wmem = I\_sw;

assign Pcsrc[1] = (E\_beq&Z)|(E\_bne&~Z)|I\_J;

assign Pcsrc[0] = I\_J;

assign Reg2reg = I\_add|I\_sub|I\_and|I\_or|I\_addi|I\_andi|I\_ori|I\_sw|I\_beq|I\_bne|I\_J;

always@(E\_Rd,M\_Rd,E\_Wreg,M\_Wreg,Rs,Rt)begin

FwdA=2'b00;

if((Rs==E\_Rd)&(E\_Rd!=0)&(E\_Wreg==1))begin

FwdA=2'b10;

end else begin

if((Rs==M\_Rd)&(M\_Rd!=0)&(M\_Wreg==1))begin

FwdA=2'b01;

end

end

end

always@(E\_Rd,M\_Rd,E\_Wreg,M\_Wreg,Rs,Rt)begin

FwdB=2'b00;

if((Rt==E\_Rd)&(E\_Rd!=0)&(E\_Wreg==1))begin

FwdB=2'b10;

end else begin

if((Rt==M\_Rd)&(M\_Rd!=0)&(M\_Wreg==1))begin

FwdB=2'b01;

end

end

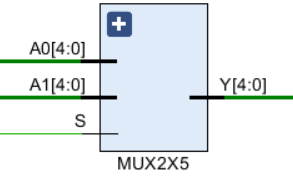
end

assign stall=((Rs==E\_Rd)|(Rt==E\_Rd))&(E\_Reg2reg==0)&(E\_Rd!=0)&(E\_Wreg==1);

assign condep=(E\_beq&Z)|(E\_bne&~Z);

endmodule

#### 2.2 MUX2X5



**模块功能**

R型指令和I行指令的Wr信号不同，所以需要一个5位二选一选择器进行选择。

**引脚及控制信号**

ID\_Inst[15:11],：R型指令的rd信号，输入信号

ID\_Inst[20:16]：I型指令的rt信号，输入信号

Regrt：选择指令的控制信号，输入信号

ID\_Wr：Wr信号，输出信号

**主要实现代码**

module MUX2X5(A0,A1,S,Y);

input [4:0]A0,A1;

input S;

output [4:0]Y;

function [4:0]select;

input [4:0]A0,A1;

input S;

case(S)

1'b0:select=A0;

1'b1:select=A1;

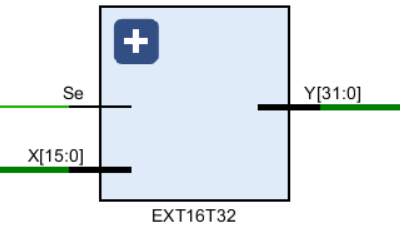
endcase

endfunction

assign Y=select(A0,A1,S);

endmodule

#### 2.3 EXT16T32



**模块功能**

I指令的addi需要对立即数进行符号拓展，andi和ori需要对立即数进行零扩展，所以需要一个扩展模块。

**引脚及控制信号**

ID\_Inst[15:0]：I型指令的立即数字段，输入信号。

Se：选择零扩展或是符号扩展的控制模块，输入信号。

ID\_EXTIMM：扩展后的立即数，输出信号。

**主要实现代码**

//数据扩展器 0零扩展 1符号扩展

module EXT16T32(X,Se,Y);

input [15:0]X;

input Se;

output [31:0]Y;

wire [31:0]E0,E1;

wire [15:0]e={16{X[15]}};

parameter z=16'b0;

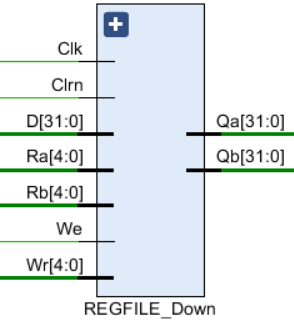
assign E0={z,X};

assign E1={e,X};

MUX2X32 i(E0,E1,Se,Y);

endmodule

#### 2.4 REGFILE\_Down



**模块功能**

给出要读取的两个寄存器编号和要写入的寄存器编号，然后由Qa和Qb端口更新Ra和Rb端口的输入编号分别输入其值。

**引脚及控制信号**

ID\_Inst[25:21]：读取寄存器编号1，输入信号。

ID\_Inst[20:16]：读取寄存器编号2或立即数，输入信号。

D：寄存器更新值，输入信号。

W\_Wr：写入寄存器编号3，输入信号。

W\_Wreg：写使能信号，为0的时候不能写入，D值不更新，为1的时候能写入，D值更新，输入信号。

Clk：时钟周期，输入信号。

Clrn：清零信号，输入信号。

ID\_Qa：输出寄存器1的值，输入信号。

ID\_Qb：输出寄存器2的值，输入信号。

**主要实现代码**

module REGFILE\_Down(Ra,Rb,D,Wr,We,Clk,Clrn,Qa,Qb);

input [4:0]Ra,Rb,Wr;

input [31:0]D;

input We,Clk,Clrn;

output [31:0]Qa,Qb;

wire [31:0]Y\_mux,Q31\_reg32,Q30\_reg32,Q29\_reg32,Q28\_reg32,Q27\_reg32,Q26\_reg32,Q25\_reg32,Q24\_reg32,Q23\_reg32,Q22\_reg32,Q21\_reg32,Q20\_reg32,Q19\_reg32,Q18\_reg32,Q17\_reg32,Q16\_reg32,Q15\_reg32,Q14\_reg32,Q13\_reg32,Q12\_reg32,Q11\_reg32,Q10\_reg32,Q9\_reg32,Q8\_reg32,Q7\_reg32,Q6\_reg32,Q5\_reg32,Q4\_reg32,Q3\_reg32,Q2\_reg32,Q1\_reg32,Q0\_reg32;

DEC5T32E dec(Wr,We,Y\_mux);

REG32\_Down i1(D,Y\_mux,Clk,Clrn,Q31\_reg32,Q30\_reg32,Q29\_reg32,Q28\_reg32,Q27\_reg32,Q26\_reg32,Q25\_reg32,Q24\_reg32,Q23\_reg32,Q22\_reg32,Q21\_reg32,Q20\_reg32,Q19\_reg32,Q18\_reg32,Q17\_reg32,Q16\_reg32,Q15\_reg32,Q14\_reg32,Q13\_reg32,Q12\_reg32,Q11\_reg32,Q10\_reg32,Q9\_reg32,Q8\_reg32,Q7\_reg32,Q6\_reg32,Q5\_reg32,Q4\_reg32,Q3\_reg32,Q2\_reg32,Q1\_reg32,Q0\_reg32);

MUX32X32 select1(Q31\_reg32,Q30\_reg32,Q29\_reg32,Q28\_reg32,Q27\_reg32,Q26\_reg32,Q25\_reg32,Q24\_reg32,Q23\_reg32,Q22\_reg32,Q21\_reg32,Q20\_reg32,Q19\_reg32,Q18\_reg32,Q17\_reg32,Q16\_reg32,Q15\_reg32,Q14\_reg32,Q13\_reg32,Q12\_reg32,Q11\_reg32,Q10\_reg32,Q9\_reg32,Q8\_reg32,Q7\_reg32,Q6\_reg32,Q5\_reg32,Q4\_reg32,Q3\_reg32,Q2\_reg32,Q1\_reg32,Q0\_reg32,Ra,Qa);

MUX32X32 select2(Q31\_reg32,Q30\_reg32,Q29\_reg32,Q28\_reg32,Q27\_reg32,Q26\_reg32,Q25\_reg32,Q24\_reg32,Q23\_reg32,Q22\_reg32,Q21\_reg32,Q20\_reg32,Q19\_reg32,Q18\_reg32,Q17\_reg32,Q16\_reg32,Q15\_reg32,Q14\_reg32,Q13\_reg32,Q12\_reg32,Q11\_reg32,Q10\_reg32,Q9\_reg32,Q8\_reg32,Q7\_reg32,Q6\_reg32,Q5\_reg32,Q4\_reg32,Q3\_reg32,Q2\_reg32,Q1\_reg32,Q0\_reg32,Rb,Qb);

endmodule

子模块基本同PC寄存器相同，只是将模块D\_FF改为

D\_FF\_Down，代码如下，即写操作时机修改为下降沿实现。

module D\_FF\_Down(D,Clk,Q,Qn);

input D,Clk;

output Q,Qn;

wire Clkn,Q0,Qn0;

not i0(Clkn,Clk);

D\_Latch d0(D,Clk,Q0,Qn0);

//主锁存器

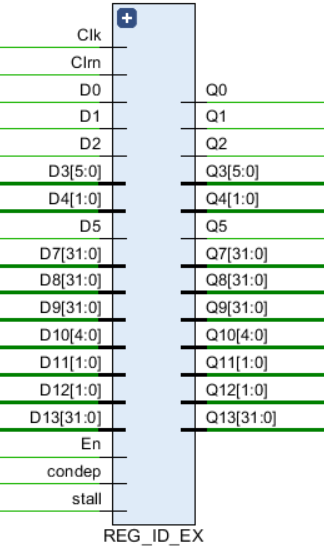
D\_Latch d1(Q0,Clkn,Q,Qn);

//从锁存器

//更改为下降沿进行写操作

endmodule

#### 2.5 REG\_ID\_EX



**模块功能**

寄存ID级的输出指令，分割ID级和EX级的指令或控制信号，防止相互干扰，在ID级执行结束时将指令的控制信号传递至下一级。

**引脚及控制信号**

Wreg,Reg2reg,Wmem,ID\_Inst[31:26],Aluc,Aluqb,ID\_PCadd4,ID\_Qa,ID\_Qb,ID\_EXTIMM,ID\_Wr,En,Clk,Clrn,E\_Wreg,E\_Reg2reg,E\_Wmem,E\_Op,E\_Aluc,E\_Aluqb,E\_PC,E\_R1,E\_R2,E\_I,E\_Rd,FwdA,FwdB,E\_FwdA,E\_FwdB,stall, condep

**主要实现代码**

module REG\_ID\_EX(D13,D0,D1,D2,D3,D4,D5,D7,D8,D9,D10,En,Clk,Clrn,Q0,Q1,Q2,Q3,Q4,Q5,Q7,Q8,Q9,Q10,D11,D12,Q11,Q12,stall,Q13,condep);

input [31:0] D7,D8,D9,D13;

input [5:0]D3;

input [4:0]D10;

input [1:0]D4,D11,D12;

input D0,D1,D2,D5;

input En,Clk,Clrn,stall,condep;

wire Clrn\_SC;

assign Clrn\_SC=Clrn&~stall&~condep;

output [31:0] Q7,Q8,Q9,Q13;

output [5:0] Q3;

output [4:0]Q10;

output [1:0]Q4,Q11,Q12;

output Q0,Q1,Q2,Q5;

D\_FFEC q0(D0,Clk,En,Clrn\_SC,Q0);

D\_FFEC q1(D1,Clk,En,Clrn\_SC,Q1);

D\_FFEC q2(D2,Clk,En,Clrn\_SC,Q2);

D\_FFEC6 q3(D3,Clk,En,Clrn\_SC,Q3);

D\_FFEC2 q4(D4,Clk,En,Clrn\_SC,Q4);

D\_FFEC q5(D5,Clk,En,Clrn\_SC,Q5);

// D\_FFEC32 q6(D6,Clk,En,Clrn\_SC,Q6);

D\_FFEC32 q7(D7,Clk,En,Clrn\_SC,Q7);

D\_FFEC32 q8(D8,Clk,En,Clrn\_SC,Q8);

D\_FFEC32 q9(D9,Clk,En,Clrn\_SC,Q9);

D\_FFEC5 q10(D10,Clk,En,Clrn\_SC,Q10);

D\_FFEC2 q11(D11,Clk,En,Clrn\_SC,Q11);

D\_FFEC2 q12(D12,Clk,En,Clrn\_SC,Q12);

D\_FFEC32 q13(D13,Clk,En,Clrn\_SC,Q13);

### 3.执行（EXE）部分

#### 3.1 SHIFTER32

**模块功能**

使用32位移位器SHIFTER32，固定左移两位

**引脚及控制信号**

E\_I：指令中的偏移量，输入信号

E\_I\_L2：偏移量左移后的结果，输出信号

**主要实现代码**

module SHIFTER32\_L2(X,Sh);

input [31:0] X;

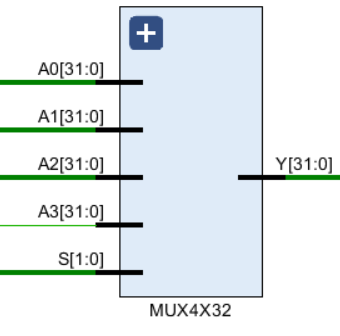
output [31:0] Sh;

parameter z=2'b00;

assign Sh={X[29:0],z};

endmodule

#### 3.2 MUX4X32



**模块功能**

在内部前推的工作中，ALU的两个输入端都需要选择来自

ID/EX寄存器或者EX/MEM寄存器或者MEM/WB寄存器所锁存的值

**引脚及控制信号**

1）

E\_R1;来自Qa端口的信号，输入信号。

D;来自WB级的信号，输入信号

M\_R；来自MEM级的信号，输入信号。

0；空信号。

E\_FwdA;选择信号。

Alu\_X：作为ALU中X端的信号，输出信号。

2）

E\_R2:来自Qb端口的信号，输入信号。

D:；来自WB级的信号，输入信号。

M\_R: 来自MEM级的信号，输入信号。

0:空信号

E\_FwdB:选择信号

E\_NUM:，选择的寄存器锁存的值和立即数做选择，后输作为ALU中Y端的信号，输出信号。

**主要实现代码**

module MUX4X32 (A0, A1, A2, A3, S, Y);

input [31:0] A0, A1, A2, A3;

input [1:0] S;

output [31:0] Y;

function [31:0] select;

input [31:0] A0, A1, A2, A3;

input [1:0] S;

case(S)

2'b00: select = A0;

2'b01: select = A1;

2'b10: select = A2;

2'b11: select = A3;

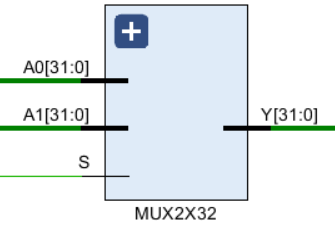
endcase

endfunction

assign Y = select (A0, A1, A2, A3, S);

endmodule

#### 3.3 MUX2X32



**模块功能**

ALU的Y端输入信号种类根据指令的不同而不同

**引脚及控制信号**

E\_I: 来自EXT16T32的信号，输入信号。

E\_NUM: 来自FwdB端口的信号，输入信号。

E\_Aluqb: 控制信号。

Y: 输入ALU进行后续计算的信号，输出信号。

**主要实现代码**

module MUX2X32(A0,A1,S,Y);

input [31:0]A0,A1;

input S;

output [31:0]Y;

function [31:0]select;

input [31:0]A0,A1;

input S;

case(S)

1'b0:select=A0;

1'b1:select=A1;

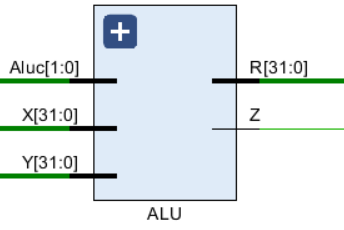
endcase

endfunction

assign Y=select(A0,A1,S);

endmodule

#### 3.4 ALU



**模块功能**

算数逻辑部件，需要实现加，减，按位与，按位或。

**引脚及控制信号**

Alu\_X：寄存器1的值。来自REG\_idex，输入信号。

Y：寄存器2的值或立即数，输入信号。

E\_Aluc：控制信号。

E\_R：输入寄存器端口D的计算结果，输出信号。

E\_Z：当值为1时代表两个输入信号值相等，当值为0时代表两个输入信号不等，输出信号。

**主要实现代码**

module ALU(X,Y,Aluc,R,Z);//ALU代码

input [31:0]X,Y;

input [1:0]Aluc;

output [31:0]R;

output Z;

wire[31:0]d\_as,d\_and,d\_or,d\_and\_or;

ADDSUB\_32 as(X,Y,Aluc[0],d\_as);

assign d\_and=X&Y;

assign d\_or=X|Y;

MUX2X32 select1(d\_and,d\_or,Aluc[0],d\_and\_or);

MUX2X32 seleted(d\_as,d\_and\_or,Aluc[1],R);

assign Z=~|R;

endmodule

子模块ADDSUB\_32

module ADDSUB\_32(X,Y,Sub,S,Cout);

input [31:0]X,Y;

input Sub;

output [31:0]S;

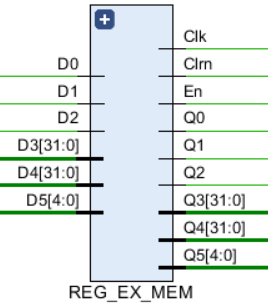
output Cout;

CLA\_32 ADD(X,Y^{32{Sub}},Sub,S,Cout);

endmodule

（其余子模块同模块PCadd4）

#### 3.5 REG\_EX\_MEM



**模块功能**

寄存EX级的输出指令，分割EX级和MEM级的指令或控制信号，防止相互干扰，在EX级执行结束时将指令的控制信号传递至下一级。

**引脚及控制信号**

E\_Wreg,E\_Reg2reg,E\_Wmem,E\_R,E\_NUM,E\_Rd,En,Clk,Clrn,M\_Wreg,M\_Reg2reg,M\_Wmem,M\_R,M\_S,M\_Rd

**主要实现代码**

module REG\_EX\_MEM(D0,D1,D2,D3,D4,D5,En,Clk,Clrn,Q0,Q1,Q2,Q3,Q4,Q5);

input D0,D1,D2;

input [31:0]D3,D4;

input [4:0]D5;

output En,Clk,Clrn;

output Q0,Q1,Q2;

output [31:0]Q3,Q4;

output [4:0]Q5;

D\_FFEC q0(D0,Clk,En,Clrn,Q0);

D\_FFEC q1(D1,Clk,En,Clrn,Q1);

D\_FFEC q2(D2,Clk,En,Clrn,Q2);

D\_FFEC32 q3(D3,Clk,En,Clrn,Q3);

D\_FFEC32 q4(D4,Clk,En,Clrn,Q4);

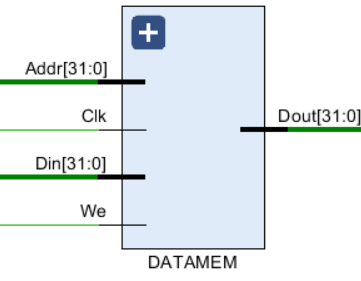
D\_FFEC5 q5(D5,Clk,En,Clrn,Q5);

endmodule

子模块同PC寄存器

### 4. 存储（MEM）访问部分

#### 4.1 DATAMEM



**模块功能**

数据存储器，通过控制信号，对数据寄存器进行读或者写操作

**引脚及控制信号**



当We为1时，进行sw指令操作，此时Din端口输入信号实际为rt，Addr端口输入信号为rs和偏移量相加的地址，在时钟周期上升沿将rt的值写入改地址的储存单元。

当We为0时，进行lw指令操作，此时Addr端口输入信号为rs和偏移量相加的地址，Dout为读取该地址储存器的内容。

**主要实现代码**

module DATAMEM(Addr,Din,Clk,We,Dout);

input [31:0]Addr,Din;

input Clk,We;

output [31:0]Dout;

reg [31:0]Ram[31:0];

//该指令存储器可存放32条指令

assign Dout=Ram[Addr[6:2]];

always@(posedge Clk)begin

if(We)Ram[Addr[6:2]]<=Din;

end

integer i;

initial begin

for(i=0;i<32;i=i+1)

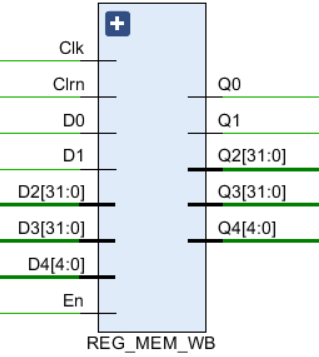
Ram[i]=0;

//所有32个数据的初值为全0

end

endmodule

#### 4.2 REG\_MEM\_WB



**模块功能**

寄存MEM级的输出指令，分割MEM级和WB级的指令或控制信号，防止相互干扰，在MEM级执行结束时将指令的控制信号传递至下一级。

**引脚及控制信号**

E\_Wreg,E\_Reg2reg,E\_Wmem,E\_Op,EX\_PC,E\_Z,E\_R,E\_R2,E\_Rd,En,Clk,Clrn,M\_Wreg,M\_Reg2reg,M\_Wmem,M\_Op,M\_PC,M\_Z,M\_R,M\_S,M\_Rd

**主要实现代码**

module REG\_MEM\_WB(D0,D1,D2,D3,D4,En,Clk,Clrn,Q0,Q1,Q2,Q3,Q4);

input D0,D1;

input [31:0] D2,D3;

input [4:0] D4;

input En,Clk,Clrn;

output Q0,Q1;

output [31:0] Q2,Q3;

output [4:0] Q4;

D\_FFEC q0(D0,Clk,En,Clrn,Q0);

D\_FFEC q1(D1,Clk,En,Clrn,Q1);

D\_FFEC32 q2(D2,Clk,En,Clrn,Q2);

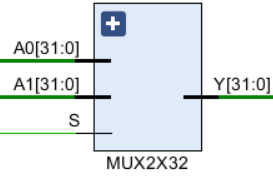
D\_FFEC32 q3(D3,Clk,En,Clrn,Q3);

D\_FFEC5 q4(D4,Clk,En,Clrn,Q4);

Endmodule

### 5. 寄存器堆写回部分（WB）

#### 5.1 MUX2X32



**模块功能**

对写入寄存器堆的数据进行选择

**引脚及控制信号**

W\_C：DATAMEM的输出值，输入信号

W\_D：ALU的输出值，输入信号

W\_Reg2reg：控制信号

D：写入R寄存器堆D端的信号，输出信号

**主要实现代码**

module MUX2X32(A0,A1,S,Y);

input [31:0]A0,A1;

input S;

output [31:0]Y;

function [31:0]select;

input [31:0]A0,A1;

input S;

case(S)

1'b0:select=A0;

1'b1:select=A1;

endcase

endfunction

assign Y=select(A0,A1,S);

endmodule

### 6. MAIN封装模块

**模块功能**

实现CPU的封装，设计输出信号使得在方正时便于观察其波形图

**主要实现代码**

module MAIN(Clk,En,Clrn,IF\_ADDR,EX\_X,EX\_Y,EX\_R,E\_Rd,M\_Rd,Rs,Rt,IF\_Inst,ID\_Inst,EX\_PC,Pcsrc);

input Clk,En,Clrn;

output[31:0] IF\_ADDR,EX\_R,EX\_X,EX\_Y,IF\_Inst,ID\_Inst,EX\_PC;

output [4:0] E\_Rd,M\_Rd,Rs,Rt;

output [1:0]Pcsrc;

wire [31:0] IF\_Result,IF\_Addr,IF\_PCadd4,D,ID\_Qa,ID\_Qb,ID\_PCadd4,ID\_Inst;

wire [31:0] E\_R1,E\_R2,E\_I,E\_I\_L2,Y,E\_R,M\_R,M\_S,Dout,W\_D,W\_C,ID\_EXTIMM,Alu\_X,E\_NUM,ID\_EXTIMM\_L2,ID\_PC;

wire[5:0] E\_Op;

wire [4:0] ID\_Wr,W\_Wr;

wire [1:0]Aluc,E\_Aluc,FwdA,FwdB,E\_FwdA,E\_FwdB;

wire Regrt,Se,Wreg,Aluqb,Reg2reg,Wmem,Z;

wire E\_Wreg,E\_Reg2reg,E\_Wmem,E\_Aluqb,Cout,M\_Wreg,M\_Reg2reg,M\_Wmem,W\_Wreg,W\_Reg2reg,stall,condep;

//IF

MUX4X32 mux4x32(IF\_PCadd4,0,EX\_PC,0,Pcsrc,IF\_Result);

PC pc(IF\_Result,Clk,En,Clrn,IF\_Addr,stall);

PCadd4 pcadd4(IF\_Addr,IF\_PCadd4);

INSTMEM instmem(IF\_Addr,IF\_Inst);

REG\_IF\_ID ifid(IF\_PCadd4,IF\_Inst,En,Clk,Clrn,ID\_PCadd4,ID\_Inst,stall,condep);

//ID

CONUNIT conunit(E\_Op,ID\_Inst[31:26],ID\_Inst[5:0],Z,Regrt,Se,Wreg,Aluqb,Aluc,Wmem,Pcsrc,Reg2reg,ID\_Inst[25:21],ID\_Inst[20:16],E\_Rd,M\_Rd,E\_Wreg,M\_Wreg,FwdA,FwdB,E\_Reg2reg,stall,condep);

MUX2X5 mux2x5(ID\_Inst[15:11],ID\_Inst[20:16],Regrt,ID\_Wr);

EXT16T32 ext16t32(ID\_Inst[15:0],Se,ID\_EXTIMM);//ID\_EXTIMM对应E\_I

REGFILE\_Down regfile(ID\_Inst[25:21],ID\_Inst[20:16],D,W\_Wr,W\_Wreg,Clk,Clrn,ID\_Qa,ID\_Qb);

SHIFTER32\_L2 shifter2(ID\_EXTIMM,ID\_EXTIMM\_L2);//控制冒险

CLA\_32 cla\_32(ID\_PCadd4,ID\_EXTIMM\_L2,0,ID\_PC,Cout);//ID\_PCadd4对应E\_PC

REG\_ID\_EX idex(ID\_PC,Wreg,Reg2reg,Wmem,ID\_Inst[31:26],Aluc,Aluqb,ID\_Qa,ID\_Qb,ID\_EXTIMM,ID\_Wr,En,Clk,Clrn,E\_Wreg,E\_Reg2reg,E\_Wmem,E\_Op,E\_Aluc,E\_Aluqb,E\_R1,E\_R2,E\_I,E\_Rd,FwdA,FwdB,E\_FwdA,E\_FwdB,stall,EX\_PC,condep);

//EX

//SHIFTER32\_L2 shifter2(E\_I,E\_I\_L2);

MUX4X32 mux4x32\_ex\_1(E\_R1,D,M\_R,0,E\_FwdA,Alu\_X);

MUX4X32 mux4x32\_ex\_2(E\_R2,D,M\_R,0,E\_FwdB,E\_NUM);

MUX2X32 mux2x32\_1(E\_I,E\_NUM,E\_Aluqb,Y);

ALU alu(Alu\_X,Y,E\_Aluc,E\_R,Z);

//CLA\_32 cla\_32(E\_PC,E\_I\_L2,0,EX\_PC,Cout);

REG\_EX\_MEM exmem(E\_Wreg,E\_Reg2reg,E\_Wmem,E\_R,E\_NUM,E\_Rd,En,Clk,Clrn,M\_Wreg,M\_Reg2reg,M\_Wmem,M\_R,M\_S,M\_Rd);

//MEM

DATAMEM datamem(M\_R,M\_S,Clk,M\_Wmem,Dout);

REG\_MEM\_WB memwb(M\_Wreg,M\_Reg2reg,M\_R,Dout,M\_Rd,En,Clk,Clrn,W\_Wreg,W\_Reg2reg,W\_D,W\_C,W\_Wr);

//WB

MUX2X32 mux2x32\_2(W\_C,W\_D,W\_Reg2reg,D);

assign IF\_ADDR=IF\_Addr;

assign EX\_R=E\_R;

assign EX\_X=Alu\_X;

assign EX\_Y=Y;

assign Rs=ID\_Inst[25:21];

assign Rt=ID\_Inst[20:16];

endmodule

### 7.test部分

**模块功能**

进行仿真，观察其波形图

**主要实现代码**

module testMAIN;

reg Clk;

reg En;

reg Clrn;

wire [31:0] IF\_ADDR;

wire [31:0] EX\_R;

wire [31:0] EX\_X;

wire [31:0] EX\_Y,IF\_Inst,ID\_Inst,EX\_PC;

wire [4:0] E\_Rd,M\_Rd,Rs,Rt;

wire [1:0]Pcsrc;

MAIN uut(.Clk(Clk),.En(En),.Clrn(Clrn),.IF\_ADDR(IF\_ADDR),.EX\_R(EX\_R),.EX\_X(EX\_X),.EX\_Y(EX\_Y),.E\_Rd(E\_Rd),.M\_Rd(M\_Rd),.Rs(Rs),.Rt(Rt),.IF\_Inst(IF\_Inst),.ID\_Inst(ID\_Inst),.EX\_PC(EX\_PC),.Pcsrc(Pcsrc));

initial begin

Clk=0;Clrn=0;En=1;

#10;

Clk=1;Clrn=0;

#10;

Clrn=1;

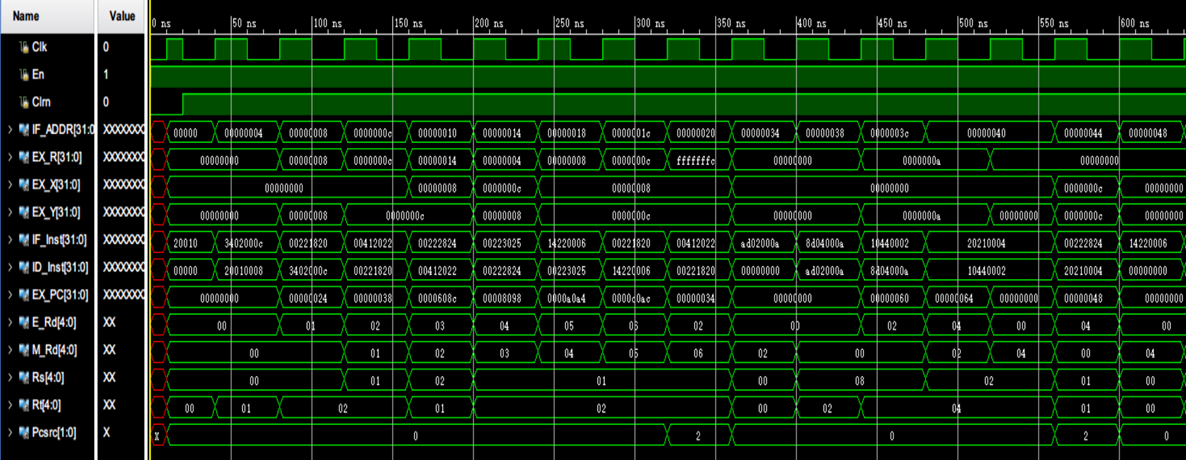
Clk=0;

forever #20 Clk=~Clk;

end

endmodule

## 三、运行结果

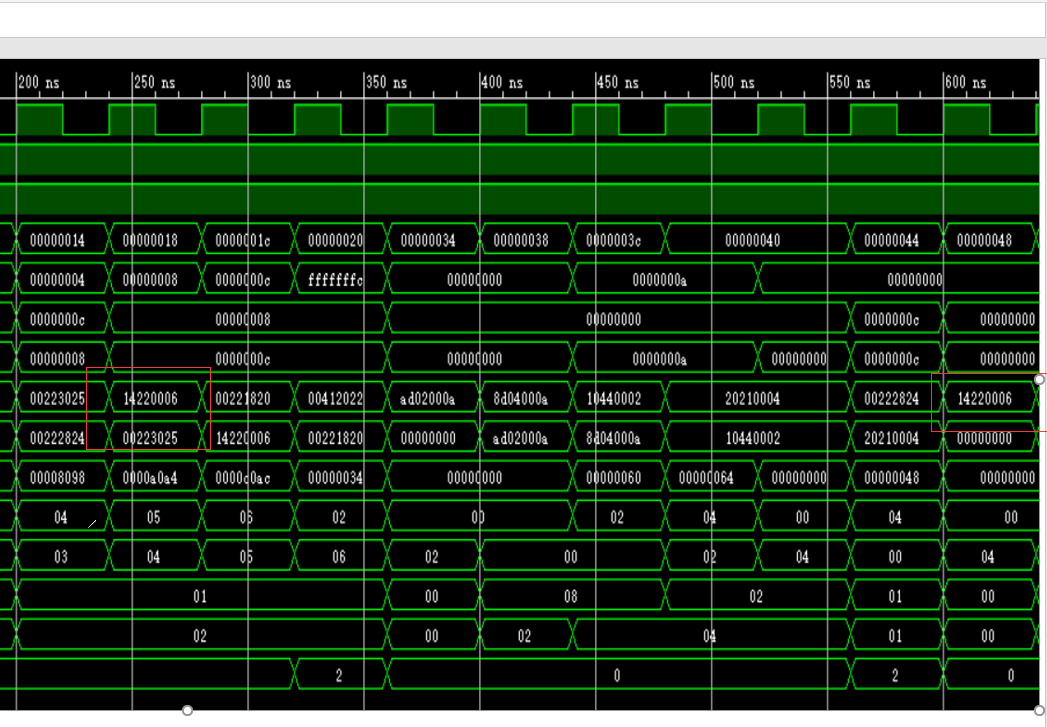


从图中可以看出，CPU基本功能基本实现

其中，我们观察00221820（在图中第三列数据），代表add $3,$1,$2，和前面的指令有数据冒险

结果：$3=$1+$2=20，输出值正确，代表一般的数据冒险得到解决。

观察14220006，图中红框，代表bne $1,$2,6

发生了跳转，说明18这条指令未执行，控制冒险得到解决。