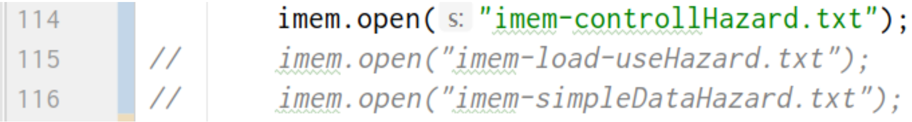
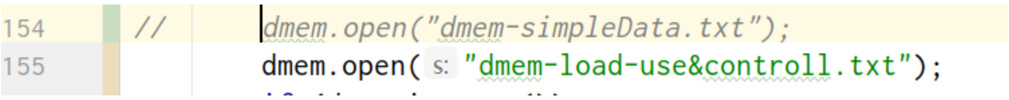
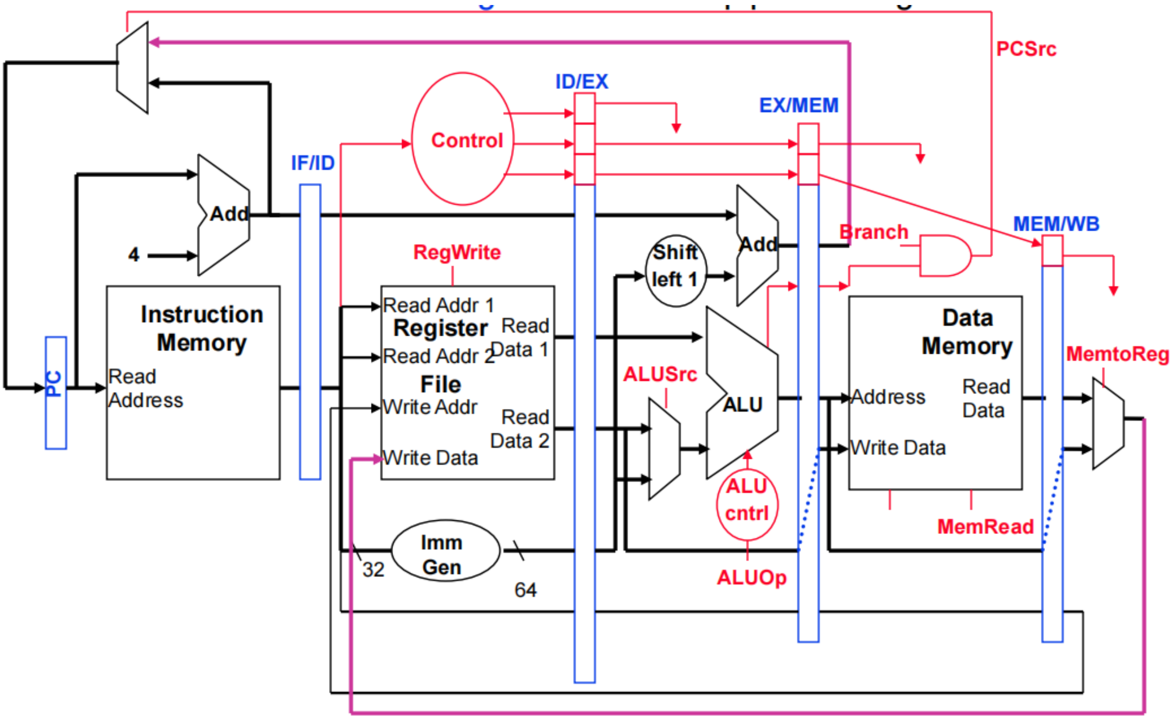
> - 由于简单RAW冒险程序、load-use冒险程序、控制冒险程序中包含不含任何RAW冒险的指令，故不单独列出不含任何RAW冒险的程序。
> - 运行程序时，可以通过选择INSMem和DataMem中的imem dmem选择不同的测试程序
>
> 
>
> ```
>      imem.open( s: "imem-controllHazard.txt");
> //   imem.open("imem-load-useHazard.txt");
> //   imem.open("imem-simpleDataHazard.txt");
> ```
>
> 
>
> ```
> //   dmem.open("dmem-simpleData.txt");
>      dmem.open( s: "dmem-load-use&controll.txt");
> ```

实验目的

在C++中为一个5阶段流水线的RISC-V处理器实现一个周期级精确的模拟器。该模拟器支持RISC-V指令集的一个子集，并且应该对每个指令的执行周期进行建模。

实验原理

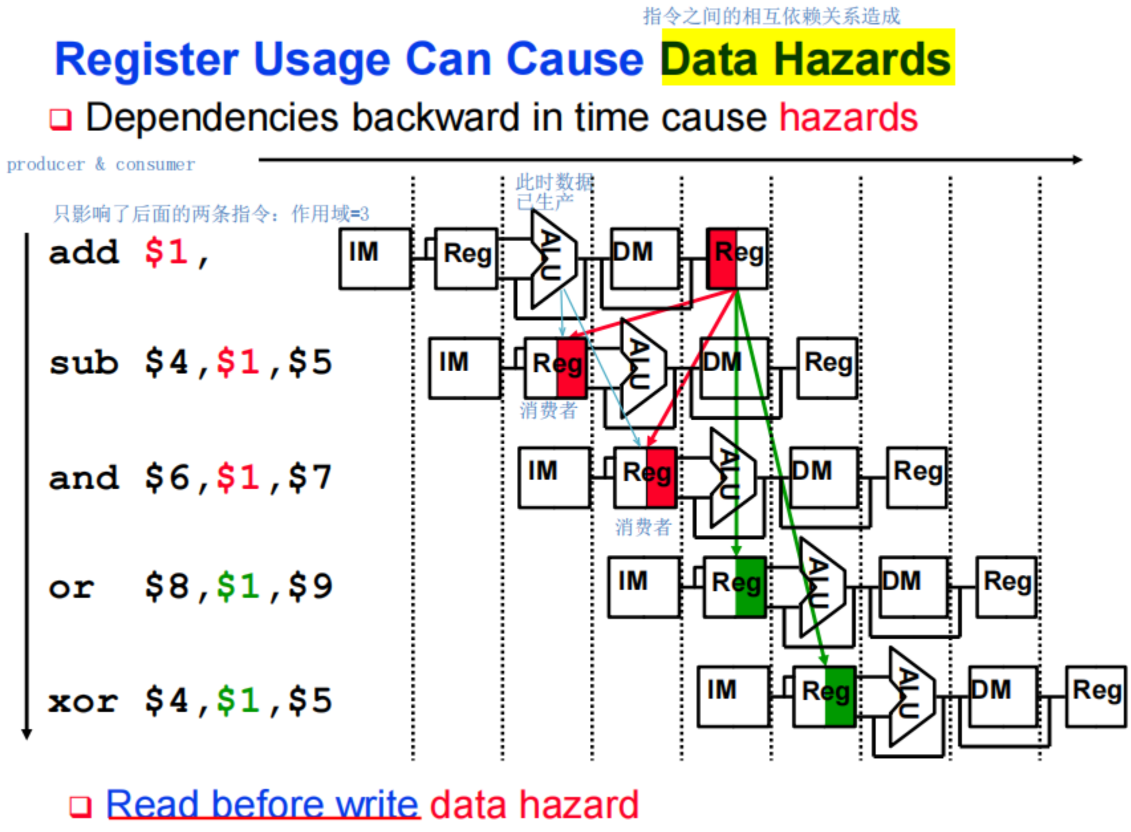
流水线



冒险

数据冒险

R类型



指令之间的相互依赖关系造成

Register Usage Can Cause **Data Hazards**

❑ Dependencies backward in time cause hazards

producer & consumer

只影响了后面的两条指令：作用域=3

此时数据已生产

add $1,

sub $4,$1,$5

and $6,$1,$7

or $8,$1,$9

xor $4,$1,$5

消费者

消费者

❑ Read before write data hazard

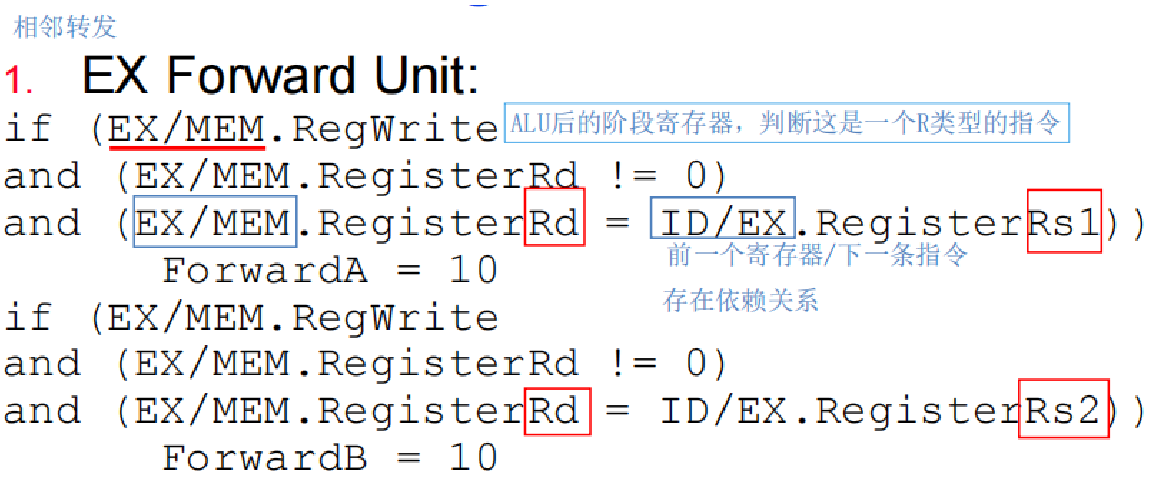
解决方法

由于第一条指令在EX阶段末尾即可得到Rd需要的结果，可以不需要等到MEM阶段再传递数据，直接在EX末尾转发。

ID/EX是新进来的指令，从EX/MEM或MEM/WB（相邻老指令或次相邻老指令）中获得Rd值

冒险条件

由于某些指令可能并没有写寄存器，所以判断一下regwrite



相邻转发

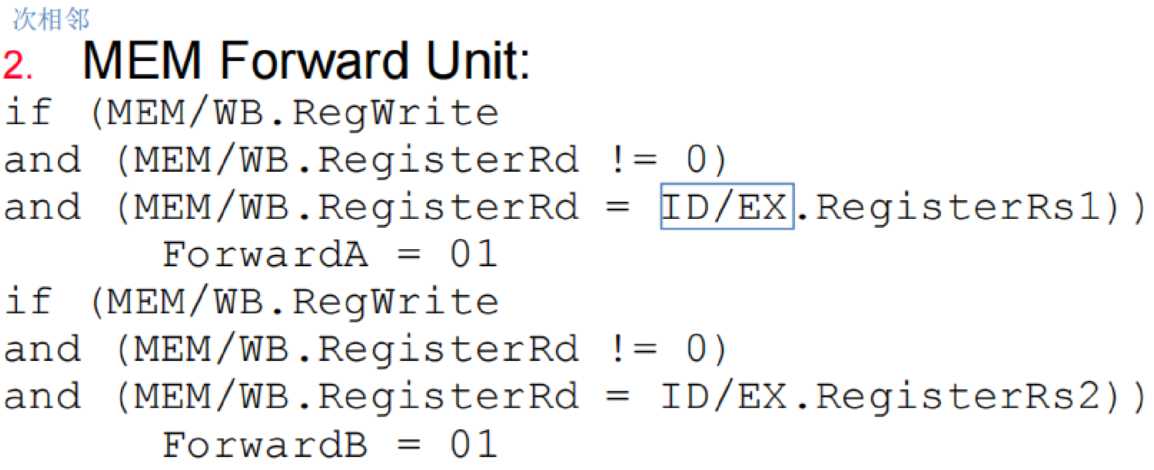
1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
      ForwardB = 10
```

ALU后的阶段寄存器，判断这是一个R类型的指令

前一个寄存器/下一条指令

存在依赖关系

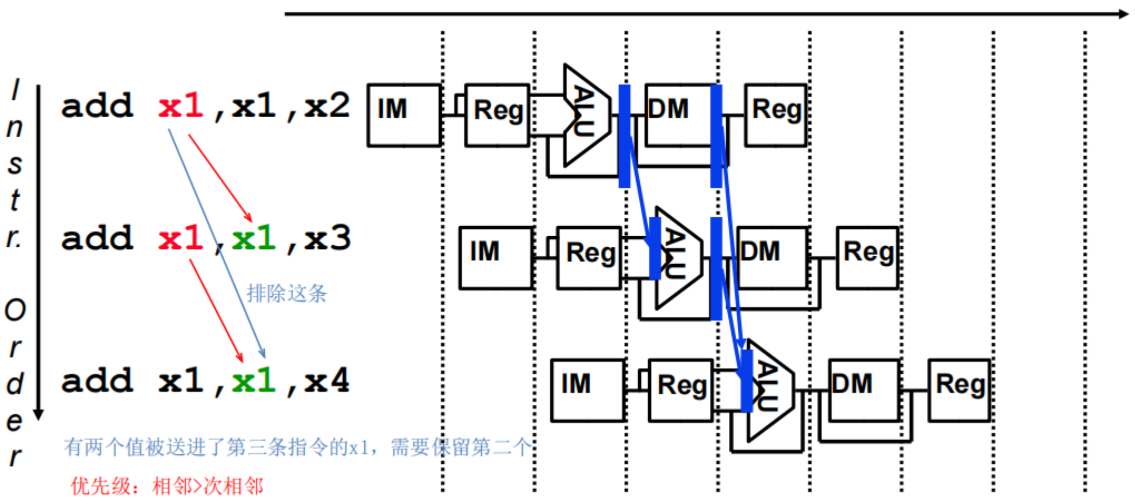


次相邻

2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
      ForwardB = 01
```

进一步问题

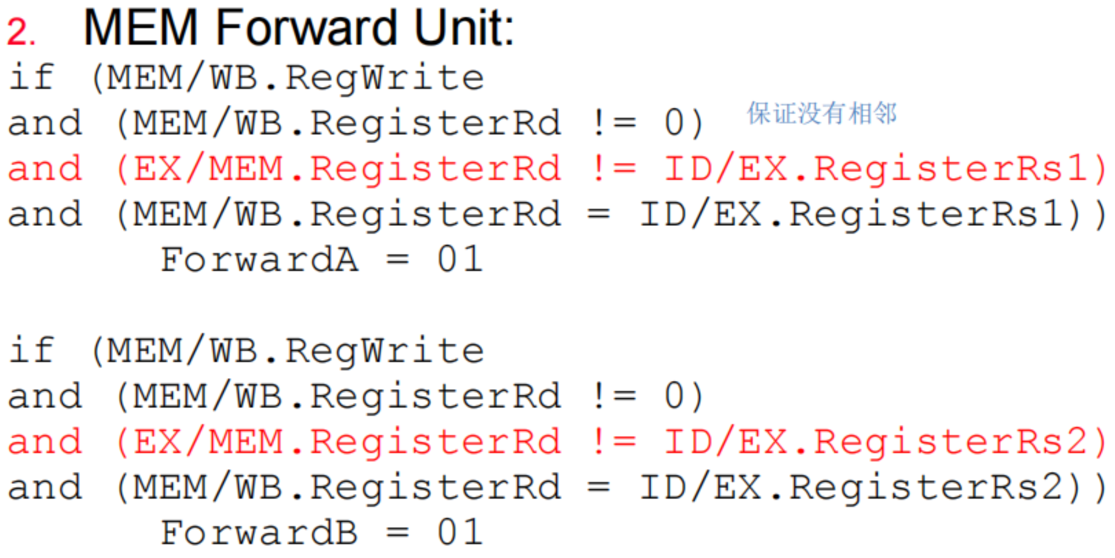


排除这条

有两个值被送进了第三条指令的x1，需要保留第二个

优先级：相邻>次相邻

此时需要修改**次相邻转发**条件判断，需要保证没有相邻情况



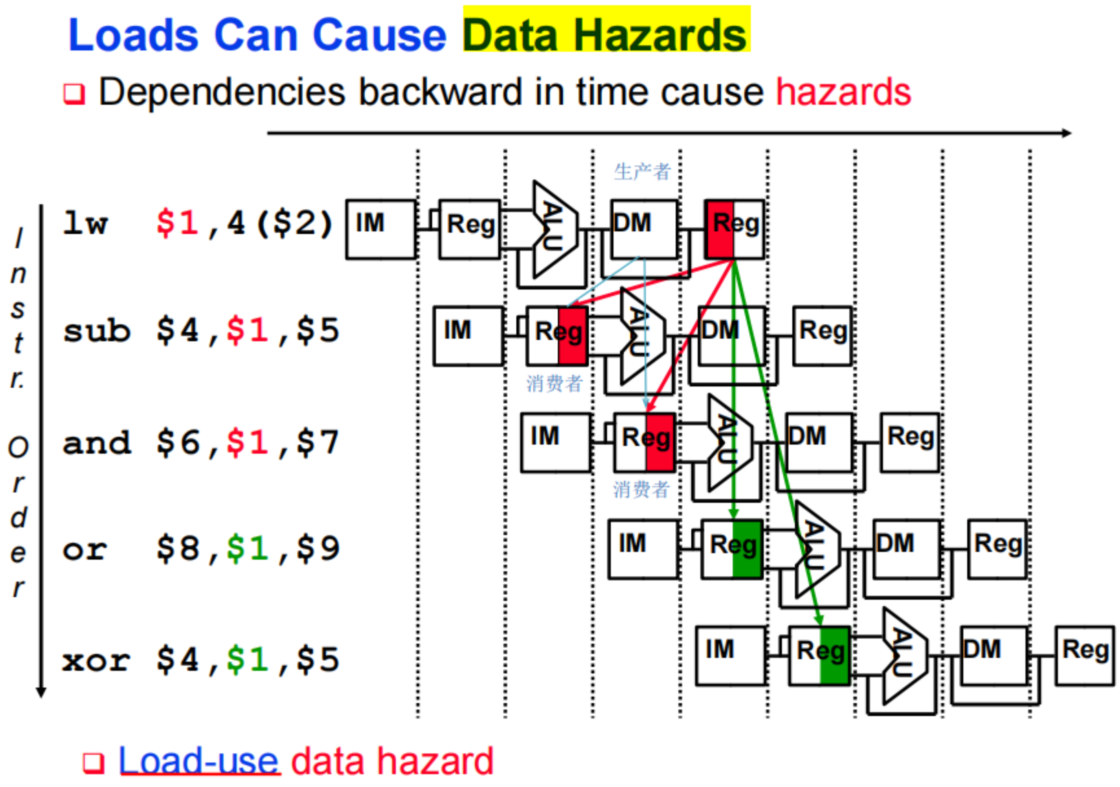
2. MEM Forward Unit:

```
if  (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd != 0)
and  (EX/MEM.RegisterRd != ID/EX.RegisterRs1)
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
          ForwardA = 01

if  (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd != 0)
and  (EX/MEM.RegisterRd != ID/EX.RegisterRs2)
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
          ForwardB = 01
```

保证没有相邻

load-use



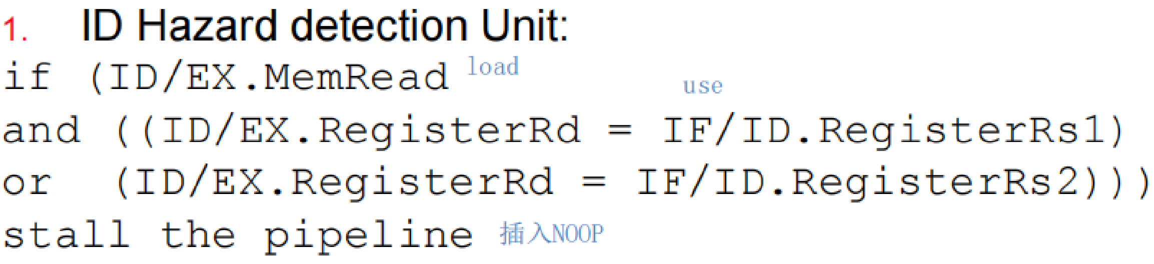
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

- Load-use data hazard

解决方法

冒险控制加在ID级，在load use之间加入noop



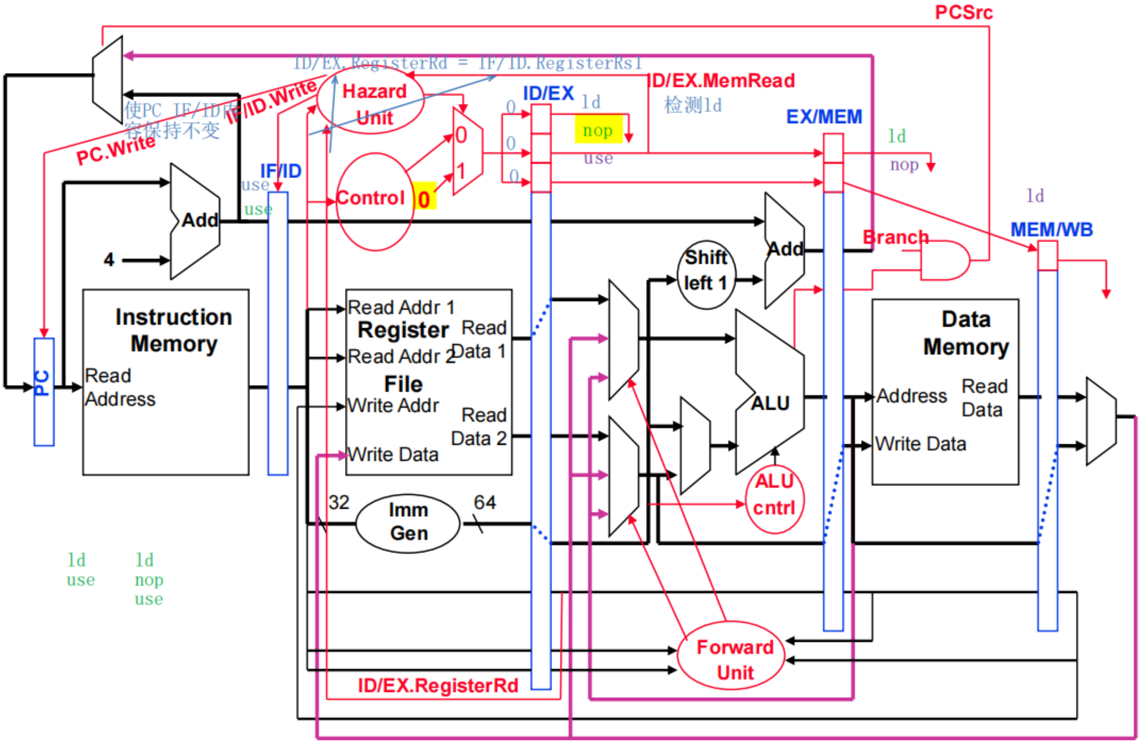
1. ID Hazard detection Unit:

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
or  (ID/EX.RegisterRd = IF/ID.RegisterRs2))
stall the pipeline
```

load

use

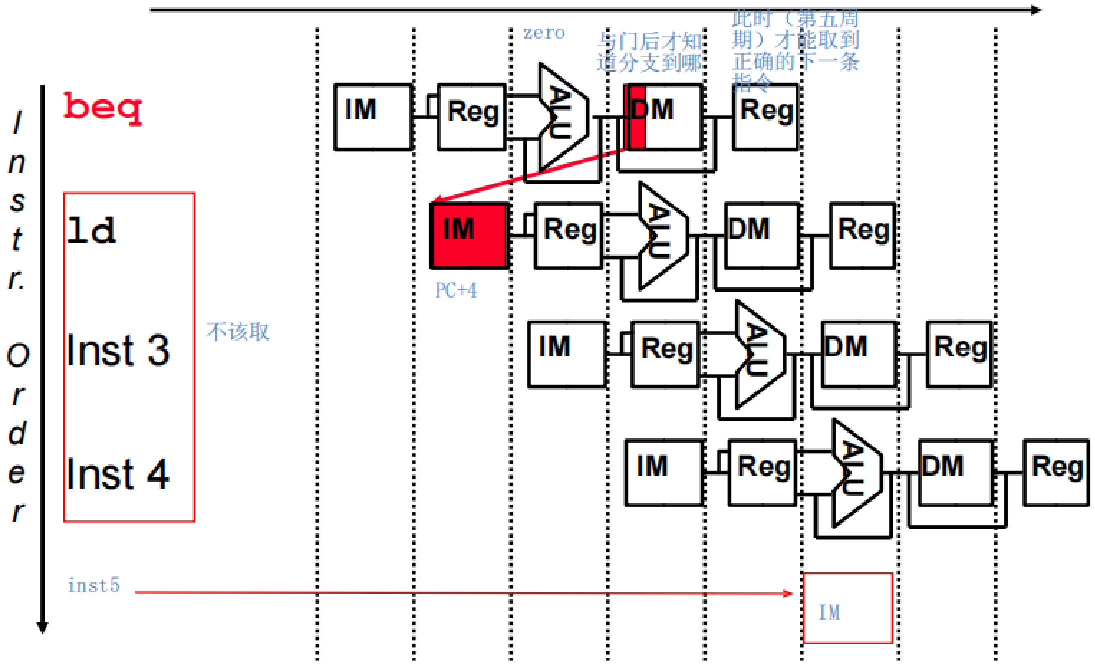
插入NOOP



控制冒险

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



解决方法

前移决策点，缩短分支的延迟

提前以下两个动作

- 计算分支目标地址

  由于在IF/ID中已经有了PC和Imm，分支地址计算可从EX提前到ID

分支目标地址对所有指令计算，但只有需要时才使用

- 判断分支条件

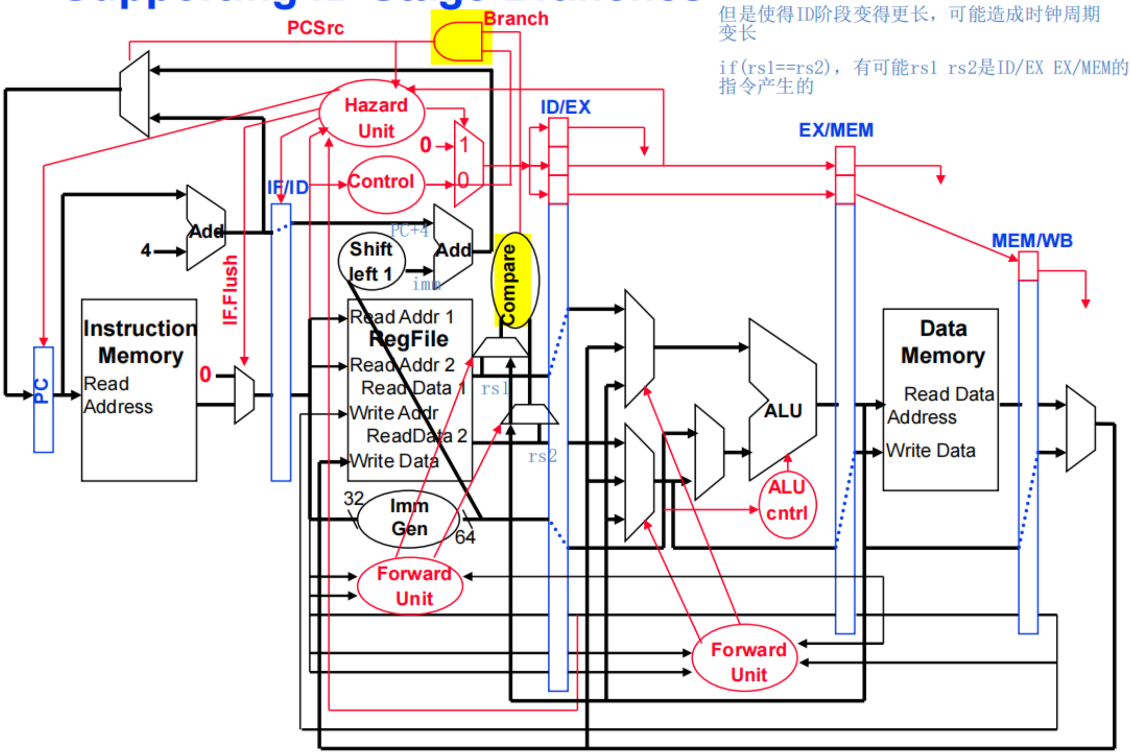
  比较从ID级取到的两个寄存器的值是否相等

Supporting ID Stage Branches

提前到ID决定，减少flush

但是使得ID阶段变得更长，可能造成时钟周期变长

if(rs1==rs2)，有可能rs1 rs2是ID/EX EX/MEM的指令产生的



实验步骤

流水线

WB

- 修改Reg file
- 承接上一级的相关信息

```
        if(!state.WB.nop){
            if(state.WB.wrt_enable){
                cout<<"writeRF:"<<state.WB.Wrt_reg_addr.to_ulong()<<'
'<<state.WB.Wrt_data.to_ulong()<<endl;
                myRF.writeRF(state.WB.Wrt_reg_addr,state.WB.Wrt_data);
            }
        }
        state.WB.nop = state.MEM.nop;
```

MEM

- 对ld和sd指令读/写内存
- 承接上一级的相关信息

```
        if(!state.MEM.nop)
        {
            // ld:rd<-(rs1+offset)
            // 64位数据->32位地址
            bitset<32> tmpALUResult=bitset<32>(state.MEM.ALUresult.to_string().substr(32,32));
            if(state.MEM.rd_mem)
            {
                state.WB.Wrt_data=myDataMem.readDataMem(tmpALUResult);
            }
            // R:rd<-(rs1+rs2/imm)
            else
                state.WB.Wrt_data=state.MEM.ALUresult;

            // sd:rs2->(rs1+offset)
            if(state.MEM.wrt_mem)
            {
                myDataMem.writeDataMem(tmpALUResult,state.MEM.Store_data);
                state.WB.Wrt_data=state.MEM.Store_data;
            }
            state.WB.Rs=state.MEM.Rs;
            state.WB.Rt=state.MEM.Rt;
            state.WB.Wrt_reg_addr=state.MEM.Wrt_reg_addr;
            state.WB.wrt_enable=state.MEM.wrt_enable;
        }
        state.MEM.nop=state.EX.nop;
```

EX

- 得到ALU结果
- 承接上一级相关信息
- **处理memory-to-memory copies**

```
        if(!state.EX.nop){
            bitset<64> data;
            data = state.EX.Read_data2;
            if(state.EX.is_I_type){
                data = state.EX.Imm;//直接将立即数读入
            }
            if(state.EX.wrt_mem)  //sd
            {
                data=state.EX.Imm;
            }

            //add
            if(state.EX.alu_op){
                state.MEM.ALUresult = bitset<64>(state.EX.Read_data1.to_ulong()+data.to_ulong());
            }
            //sub
            else{
                state.MEM.ALUresult = bitset<64>(state.EX.Read_data1.to_ulong() - data.to_ulong());
            }
        }
        state.MEM.Store_data = state.EX.Read_data2;
        state.MEM.Rt = state.EX.Rt;
        state.MEM.Rs = state.EX.Rs;
        state.MEM.Wrt_reg_addr = state.EX.Wrt_reg_addr;
        state.MEM.wrt_enable = state.EX.wrt_enable;
        state.MEM.rd_mem = state.EX.rd_mem;
        state.MEM.wrt_mem = state.EX.wrt_mem;

        //ld&sd相连
        if(state.MEM.Rt == state.WB.Wrt_reg_addr ){
            state.MEM.Store_data = state.WB.Wrt_data;
        }

        state.EX.nop = state.ID.nop;
```

ID

- 解码
- **处理RAW hazard, load-use hazard, controll hazard**
- 承接上一级相关信息

```
        if(!state.ID.nop){
            //判断是否是I-type
            // 不是
            if(state.ID.Instr.to_string().substr(25,7) != "0010011" && state.ID.Instr.to_string().substr(25,7) != "0000011"){
                state.EX.is_I_type = false;
                //确定rs1，rs2
                state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
                state.EX.Rt = bitset<5>(state.ID.Instr.to_string().substr(7,5));
                state.EX.Read_data1 = myRF.readRF(state.EX.Rs);
                state.EX.Read_data2 = myRF.readRF(state.EX.Rt);
                state.EX.rd_mem= false;
                state.EX.wrt_enable = true;
            }
            // 是
            else{
                state.EX.is_I_type = true;
                // rs1
                state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
                state.EX.Read_data1 = myRF.readRF(state.EX.Rs);
                // rd
                state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));
            }

            // ld
            if(state.ID.Instr.to_string().substr(25,7) == "0000011"){
                state.EX.rd_mem = true;
                state.EX.wrt_enable= true;
                state.EX.alu_op= true;
                state.EX.Imm = bitset<64>(state.ID.Instr.to_string().substr(0,12));//立即数
                if(state.EX.Imm[11]){//如果是负数
                    state.EX.Imm = bitset<64>(string(52,'1')+state.ID.Instr.to_string().substr(0,12));//立即数
                }
            }
            // sd
            if(state.ID.Instr.to_string().substr(25,7) == "0100011"){
                state.EX.Imm=bitset<64>(state.ID.Instr.to_string().substr(0, 7) +
                    
 state.ID.Instr.to_string().substr(20, 5));
                state.EX.wrt_mem = true;
                state.EX.alu_op= true;
            }
            // R
            if(state.ID.Instr.to_string().substr(25,7) == "0110011"){
                state.EX.wrt_enable = true;
                state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));//rd
                // add
                if(state.ID.Instr.to_string().substr(0, 7) == string("0000000"))
                    state.EX.alu_op= true;
                // sub
                if(state.ID.Instr.to_string().substr(0, 7) == string("0100000"))
                    state.EX.alu_op= false;
            }
            // branch
            if(state.ID.Instr.to_string().substr(25,7) == "1100011")
            {
                state.EX.Imm=bitset<64>(state.ID.Instr.to_string().substr(0,1)+state.ID.Instr.to_string().substr(24,1)+state.ID.Instr.to_string().substr(1,6)+state.ID.Instr.to_string().substr(20,4));
            }


            //处理raw hazard，不包括load-use 冒险
            if(!state.EX.rd_mem){
                int flag=0;  // 是否处理过相邻
                if(state.MEM.wrt_enable){//需要写回数据,相邻的优先级应该大于次相邻
                    if(state.EX.Rs == state.MEM.Wrt_reg_addr){
                        flag=1;
                        state.EX.Read_data1 = state.MEM.ALUresult;
                        cout<<"RAW11 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }
                    if(state.EX.Rt == state.MEM.Wrt_reg_addr){
                        flag=1;
                        state.EX.Read_data2 = state.MEM.ALUresult;
                        cout<<"RAW12 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }

                }
                if(state.WB.wrt_enable&&flag==0){//需要写回数据
                    if(state.EX.Rs == state.WB.Wrt_reg_addr){
                        state.EX.Read_data1 = state.WB.Wrt_data;
                        cout<<"RAW21 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }
                    if(state.EX.Rt == state.WB.Wrt_reg_addr){
                        state.EX.Read_data2 =state.WB.Wrt_data;
                        cout<<"RAW22 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }

                }
            }
            //ld指令
            else
            {
                int flag=0;
                //ld作为consumer
                if(state.EX.Rs == state.MEM.Wrt_reg_addr){
                    // x0不可能被写，只能是初始化值还未修改
                    if(state.MEM.Wrt_reg_addr.to_string()!="00000"){
                        flag=1;
                        state.EX.Read_data1 = state.MEM.ALUresult;
                        cout<<"RAW31 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }

                }
                if(state.EX.Rs == state.WB.Wrt_reg_addr&&flag==0){
                    if(state.WB.Wrt_reg_addr.to_string()!="00000"){
                        state.EX.Read_data1 = state.WB.Wrt_data;
                        cout<<"RAW32 hazard cycle:"<<cycle<<" reg:"<<state.MEM.Wrt_reg_addr<<endl;
                    }
                }
                // load-use
                if(state.EX.Wrt_reg_addr.to_string()==myInsMem.readInstr(state.IF.PC).to_string().substr(12,5)||
                state.EX.Wrt_reg_addr.to_string()==myInsMem.readInstr(state.IF.PC).to_string().substr(7,5))
                {
                    if(state.EX.Wrt_reg_addr.to_string()!="00000")
                    // x0不可能被写，只能是初始化值还未修改
                    {
                        lu_flag=1;
                        cout<<"load-use hazard cycle:"<<cycle<<" reg:"<<state.EX.Wrt_reg_addr<<endl;
                        state.ID.nop = true;//flush
                    }
                }
            }

            // branch
            if(state.ID.Instr.to_string().substr(25,7) == "1100011"){
                cout<<"branch: "<<state.EX.Rs<<' '<<state.EX.Rt<<' '<<endl;
                if(state.EX.Read_data1 != state.EX.Read_data2){//不相等需要跳转
                    cout<<"imm: "<<state.EX.Imm<<' '<<endl;
                    string s = state.ID.Instr.to_string();
                    bitset<32> addressExtend;
                    addressExtend = bitset<32>(s.substr(0,1)+s.substr(24,1)+s.substr(1,6)+s.substr(20,4));
                    cout<<"addressExtend: "<<addressExtend<<' '<<endl;
                    if(state.EX.Imm[11]){
                        addressExtend = bitset<32>(string(20,'1') + addressExtend.to_string().substr(20,12));//立即数
                        addressExtend.flip();
                        cout<<"addressExtend-after: "<<addressExtend<<' '<<endl;
                        state.IF.PC = bitset<32>(state.IF.PC.to_ulong()-(addressExtend.to_ulong()+1));//如果是负数
                    }
                    else{
                        state.IF.PC = bitset<32>(addressExtend.to_ulong()+state.IF.PC.to_ulong());
                    }
                    state.EX.nop = true;
                }
            }

        }
        // nop，清空所有控制信号
        else
        {
            state.EX.is_I_type= false;
            state.EX.rd_mem= false;
            state.EX.wrt_mem= false;
            state.EX.alu_op= false;
            state.EX.wrt_enable= false;

        }
        if(!lu_flag)
            state.ID.nop = state.IF.nop;
```

IF

- 取指
- 更新PC

```
        if(!state.IF.nop)
        {
            if(!lu_flag)
            {
                // 取指
                state.ID.Instr=myInsMem.readInstr(state.IF.PC);
                // 更新PC
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + 4);
            }
            else
            {
                lu_flag=0;
            }
            //判断是否需要终止
            if(state.ID.Instr.to_string()=="11111111111111111111111111111111")
            {
                state.IF.nop= true;
                state.ID.nop= true;
            }
        }
```

冒险

简单RAW冒险

对指令：`B[1] = A[i-j]`，涉及到

```
// data hazard, both EX forwarding and MEM forwarding
sub x30, x28, x29 // compute i-j
add x30, x30, x30 // multiply by 8 to convert the double word offset to a byte offset
add x30, x30, x30
add x30, x30, x30
add x10, x10, x30

// data hazard
add x10, x10, x30
ld x30, 0(x10) // load A[i-j]

// memory-to-memory copies
ld x30, 0(x10) // load A[i-j]
sd x30, 8(x12)  // store in B[1]
```

设计DMEM

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001  //j=1
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000010  //i=2
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00101000  //A的地址=5*8=40
00000000 00000000 00000000 00000000
00000000 00000000 00000000 01000000  //B的地址= 8*8=64  需要修改
11111111 11111111 11111111 11111111
01111111 11111111 11111111 11111110  //
11111111 11111111 11111111 11111111
01111111 11111111 11111111 11111110  //A[0]
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111  //A[I-J]=7/A[1]=7
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111  //A[2]=A[J]
00000000 00000000 00000000 00000000
00000000 00000000 00000000 11111111  //B[0]
00000000 00000000 00000000 00000000
00000000 00000000 00000000 11111111  //B[1]
11111111 11111111 11111111 11111111
11111111 11111111 11111111 11111111
```

设计汇编代码

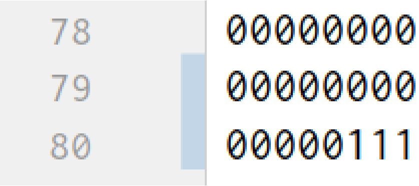
```
ld x29 0(x0)  // j
ld x28 8(x0)  // i
ld x10 16(x0)  //&A
ld x12 24(x0)  //&B
sub x30, x28, x29 // compute i-j
add x30, x30, x30 // multiply by 8 to convert the double word offset to a byte offset
add x30, x30, x30
add x30, x30, x30
add x10, x10, x30
ld x30, 0(x10) // load A[i-j]
sd x30, 8(x12)  // store in B[1]
```

设计二进制指令

```
00000000 00000000 00111110 10000011
00000000 10000000 00111110 00000011
00000001 00000000 00110101 00000011
00000001 10000000 00110110 00000011
01000001 11011110 00001111 00110011
00000001 11101111 00001111 00110011
00000001 11101111 00001111 00110011
00000001 11101111 00001111 00110011
00000001 11100101 00000101 00110011
00000000 00000101 00111111 00000011
00000001 11100110 00110100 00100011
11111111 11111111 11111111 11111111
```

实验结果

dmemresult.txt，地址=72处B[1]获得数据7



```
00000000
00000000
00000111
```

load-use冒险

对指令`i=3*j`

设计DMEM

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000100  /j=4，以下沿用
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00101000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 01000000
11111111 11111111 11111111 11111111
01111111 11111111 11111111 11111110
11111111 11111111 11111111 11111111
01111111 11111111 11111111 11111110
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000111
00000000 00000000 00000000 00000000
00000000 00000000 00000000 11111111
00000000 00000000 00000000 00000000
00000000 00000000 00000000 11111111
11111111 11111111 11111111 11111111
11111111 11111111 11111111 11111111
```

汇编指令

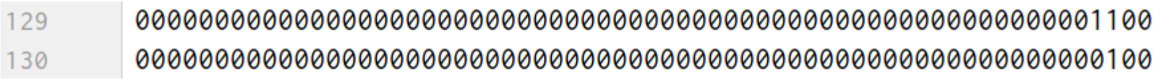
```
ld x29,0(x0)  // j
add x28,x29,x29  // i=2*j
add x28,x28,x29  // i=3*j
```

IMEM

```
00000000 00000000 00111110 10000011
00000001 11011110 10001110 00110011
00000001 11011110 00001110 00110011
11111111 11111111 11111111 11111111
```

实验结果

RFresult.txt中，(x28)=12，(x29)=4，即i=j*3=12



```
129  0000000000000000000000000000000000000000000000000000000000001100
130  0000000000000000000000000000000000000000000000000000000000000100
```

控制冒险

对指令

```
int i=1,j=4;
i*=2;
while(i!=j)
    i*=2;
j*=2;
```

设计DMEM

沿用load-use

汇编指令

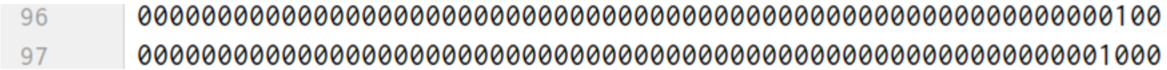
```
ld x29, 0(x0)
ld x28, 8(x0)
Loop: add x28, x28, x28
ld x10 16(x0)  // 为了不造成数据依赖
ld x12 24(x0)
bne x28,x29, Loop
add x29,x29,x29
```

IMEM

```
00000000 00000000 00111110 10000011
00000000 10000000 00111110 00000011
00000001 00000000 00110101 00000011
00000001 10000000 00110110 00000011
00000001 11001110 00001110 00110011
00000001 00000000 00110101 00000011
00000001 10000000 00110110 00000011
11111111 11011110 00000000 11100011
00000001 11011110 10001110 10110011
11111111 11111111 11111111 11111111
```

实验结果

在RFresult.txt中，x28=4, x29=8



```
96  0000000000000000000000000000000000000000000000000000000000000100
97  0000000000000000000000000000000000000000000000000000000000001000
```