MIPS32 处理器设计文档

组号：**30**

组员及分工：

薛双百 061221126：组长，整体框架设计，两个流水段寄存器，冒险检测单元，乘法器（基 2-booth 编码+wallace 树（4-2 压缩&3-2 压缩）+超前进位加法），处理器顶层通路的搭建，文档汇总；并和张家军同学共同完成了**调试工作。**

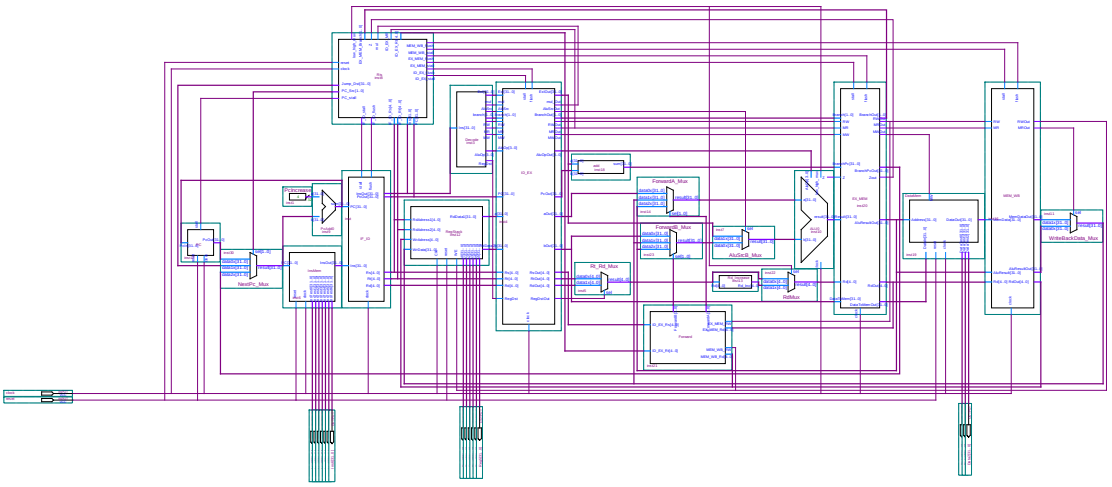
严磊 061221127：两个流水段寄存器，转发单元，ALU 模块，文档编写。

张家军 061221139：超前进位加法器（32 位），处理器顶层通路中的**一个定制的 5 位加法器，译码单元；调试。**

钟鹏 061221147：寄存器堆，指令存储器，数据存储器，部分文档。

目录

图 1　MIPS32 处理器框图



摘要：

我们组设计并实现的是一个五级流水线的 MIP32 处理器。处理器的设计图纸如上图 1 所示，**大图见图片文件夹下的图片《MIP32 最终版》**。另附调试时的图纸，及除去各测试针脚后的版本。为了便于在设计中引入针脚观察内部状态，测试时的版本结构比较松散。最终版只是在测试版的基础上除去测试针脚，并重新布局，使设计图看上去更为紧凑。

我们实现的指令集包括：绝大多数 R-Type 指令（目前没有实现除法），分支指令（beq,bne），跳转指令（jump），I-Type 立即数型指令，访存指令（load word/save word）。对于其中的移位指令，没有做移位器，故只能移 1 位，不是标准的 MIPS 指令定义。分支和跳转指令按照教材上的定义来实现，没有设置分支延迟槽，这点也和工业标准的 MIPS 指令不同。

流水线的五级分别为：取指，译码，执行，访存，写回。指令在译码阶段生成所有的控制信号；4 个流水段寄存器用于在指令执行的各阶段间传递必要的数据和控制信息。转发单元保证进入 ALU 参与运算的数据总是最“新”的。冒险检测单元在必要的时刻阻塞流水线，或者清除保存于流水段寄存器中的指令；该单元还根据流水线的状态决定下条指令的地址。

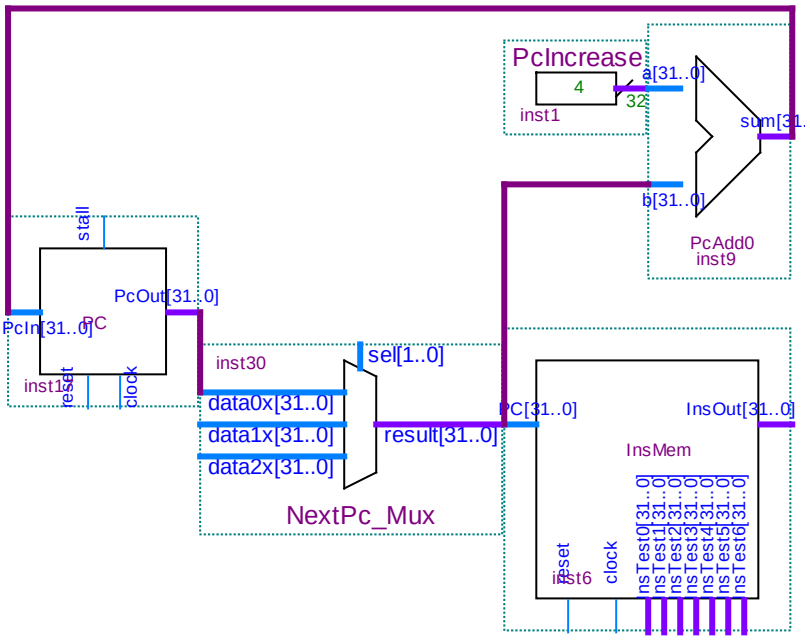
在核心部件 ALU 中，加法部件为超前进位加法器，9 级门延迟（以异或门算两级计）；乘法器中用到的技术有：基 2 布斯重编码，wallace 树压缩（4-2 压缩及 3-2 压缩），超前进位加法。乘法运算经 3 个周期得到积的低 32 位，第 4 个周期得到积的高 32 位。

在图中，有两个输入针脚，其中 clock 为时钟；reset 为复位信号，用于驱动存储器写入初始的测试数据。另有 16 个输出针脚，分别来自指令存储器（7 个），寄存器堆（6 个）和数据存储器（3 个）。这 16 个针脚全部用于测试，因为在 Quartus 中仿真时，对输出没有影响的寄存器会被优化掉，故将相应寄存器的内容引出作为输出，以便观察和调试。

各模块功能及其接口的定义见后文的详细设计报告。调试方法见设计报告第 8 章。

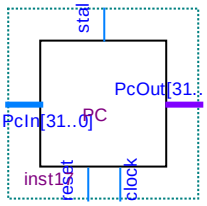
1.取指

图 1.0　取指单元



1.1 程序计数器

图 1.1　程序计数器



在调试中发现之前设计的取指逻辑存在问题（原设计见周报 7），故修改成图 2 所示的结构。在此结构下，PC 中存储的是刚取出的指令的下条指令的地址，即 PC+4，作为下次取指的备选指令地址。

PC 模块的核心是一个 32 位寄存器。对外的接口有：PcIn,PcOut,stall,reset,Clock。其中：

PcIn、PcOut 是 32 位寄存器 PC 的输入和输出；

stall 是阻塞信号，该信号有效时，PC 的值保持不变；

reset 为复位信号，该信号有效时，PC 的值清 0；

clock 为时钟信号，PC 寄存器在时钟正延触发。

该模块的代码如下：

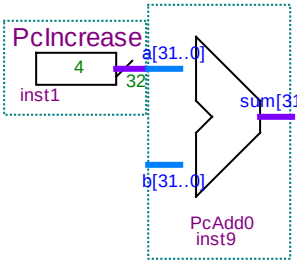
```
////////////////////////////////////////////////////////////////////////////////////////////////
module PC(reset,clock,stall,PcIn,PcOut);

input reset;
```

```
input clock;
input stall;
input [31:0] PcIn;
output reg [31:0] PcOut;

always@(posedge clock)begin
   if(reset)PcOut<=0;

   else if(~stall)begin
      PcOut<=PcIn;
      end
end

endmodule
```

//

1.2 下地址专用加法器

图 1.2　下地址专用加法器

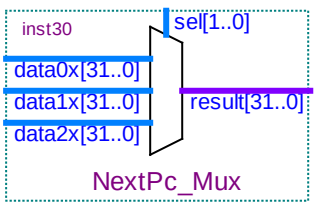


一个 32 位超前进位加法器，输入为当前 PC（指实际用于取指的指令地址，可能为 PC 寄存器中的值，也可能为分支地址或者跳转地址）和常数 4，输出为 PC+4，故其结果为实际用于取指的指令地址的下地址，该值被送到 PC 单元作为输入，如图 1.0 所示。该模块的实现细节见 ALU 的加法器章节。

实际上，**该加法器只要 30 位就够了**，因为不论是输入还是输出，其最低两位都是 0，所以完全可以只处理高 30 位，再在低两位补 0 输出。但那样就需要重新定制一个 30 位加法器，由于时间紧迫，在此不予实现，直接复用 32 位加法器模块。

1.3 下地址选择(NextPc_Mux)

图 1.3　下地址多路选择器



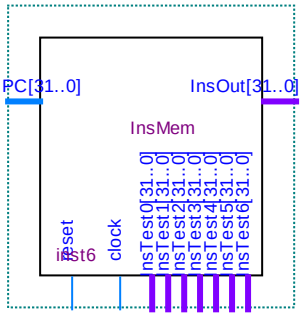
一个多路选择器，三输入，两位控制信号。输入输出都是 32 位。控制信号来自冒险检测单元，其功能如下表：

表 1　下地址选择

控制信号 sel	选择数据	输出结果
0	Data0	PC+4
1	Data1	跳转地址
2	Data2	分支地址

1.4 指令存储器 InsMem

图 1.4 指令存储器



接口有 PC、InsOut、reset 和 clock。7 个 InsTest 用于调试时观察指令寄存器的状态，与该模块的实际功能无关。各接口定义如下：

PC 是要读取的指令的地址；

InsOut 是读取的指令；

Reset 是复位信号，在该信号驱动下，完成指令存储器的初始化，即写入测试指令。不会写 testbench，故用此法。

Clock 是时钟信号，指令存储器的写操作在时钟的正延触发。

需要特别指出的是，我们的设计本身是打算将指令存储器的大小设为 4K 字节，即 1K 条指令，但那样会严重加长编译时间，给调试带来很大麻烦。所以我们将其尺寸设为几十个字节。

附指令寄存器的代码如下：

```
//////////////////////////////////////////////////////////////////////////////////
module InsMem(reset,clock,PC,InsOut,
                InsTest0,InsTest1,InsTest2,InsTest3,InsTest4,InsTest5,InsTest6);

input reset;
input clock;
input [31:0] PC;

output [31:0] InsOut;
output [31:0] InsTest0,InsTest1,InsTest2,InsTest3,InsTest4,InsTest5,InsTest6;

reg [7:0] Ins [35:0];//
```

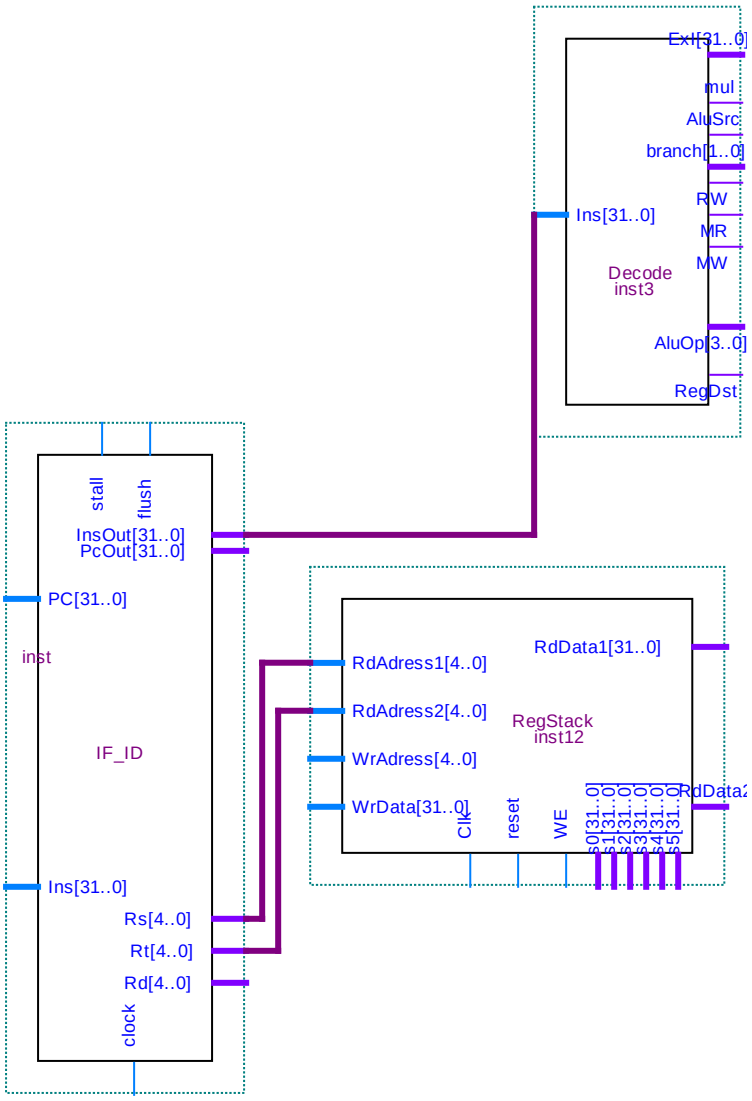
```
always@(posedge clock)begin
  if(reset)begin   //复位信号，用于写入测试指令
    Ins[0]<=8'b10001100;//lw R2,0(R4);
    Ins[1]<=8'b10000010;
    Ins[2]<=8'b00000000;
    Ins[3]<=8'b00000000;

    Ins[4]<=8'b00000000;//add R2,R1,R2;
    Ins[5]<=8'b00100010;
    Ins[6]<=8'b00010000;
    Ins[7]<=8'b00100000;

    Ins[8]<=8'b00000000;//add R2,R1,R2;
    Ins[9]<=8'b00100010;
    Ins[10]<=8'b00010000;
    Ins[11]<=8'b00100000;

    Ins[12]<=8'b00100000;//addi R2,R2,2;
    Ins[13]<=8'b01000010;
    Ins[14]<=8'b00000000;
    Ins[15]<=8'b00000010;

    Ins[16]<=8'b00010000;//beq R2,R3,4;
    Ins[17]<=8'b01000011;
    Ins[18]<=8'b00000000;
    Ins[19]<=8'b00000100;

    Ins[20]<= 8'b00001000;//j 4;
    Ins[21]<= 8'b00000000;
    Ins[22]<= 8'b00000000;
    Ins[23]<= 8'b00000001;

    Ins[24]<=8'b10101100;//sw R2,4(R4);
    Ins[25]<=8'b10000010;
    Ins[26]<=8'b00000000;
    Ins[27]<=8'b00000100;

    Ins[28]<=8'b00000000;//mult R4,R2,R3;
    Ins[29]<=8'b01000011;
    Ins[30]<=8'b00100000;
    Ins[31]<=8'b00011000;

    end
end
```

```
assign InsOut={Ins[PC],Ins[PC+1],Ins[PC+2],Ins[PC+3]};
assign InsTest0={Ins[0],Ins[1],Ins[2],Ins[3]};
assign InsTest1={Ins[4],Ins[5],Ins[6],Ins[7]};
assign InsTest2={Ins[8],Ins[9],Ins[10],Ins[11]};
assign InsTest3={Ins[12],Ins[13],Ins[14],Ins[15]};
assign InsTest4={Ins[16],Ins[17],Ins[18],Ins[19]};
assign InsTest5={Ins[20],Ins[21],Ins[22],Ins[23]};
assign InsTest6={Ins[24],Ins[25],Ins[26],Ins[27]};
endmodule
```

//

2.译码

图 2.0 译码阶段



2.1 IF_ID(取指/译码)流水段寄存器

接口定义如下：

Ins，32 位指令，取自指令存储器。

PC，程序计数器，实际上是当前指令的下条指令地址，即 PC+4。将此信号沿数据通路向前传递，用于分支和跳转指令。

Stall，阻塞信号，使流水段寄存器的内容保持不变，高电平有效。

Flush，清除信号，将流水段寄存器内的指令清除，实际实现时，只是将寄存器堆和数据存储器的写使能信号清 0。有些情况下这种逻辑会出错，对此我们在介绍冒险检测单元时再作说明。

Rs、Rt、Rd，取自 Ins 的相应字段，分别为：Ins[25:21],Ins[20:16],Ins[15:11]。这三个信号是访问寄存器堆时所用地址。

其中 clock、stall 和 flush 信号在每个流水段寄存器上都有设置，功能完全相同，后文将不再重复说明这三个信号。

2.2 译码单元 Decode

译码单元以 32 位指令作为输入，其逻辑编码完全遵照 MIPS32 指令集设计。该单元产生的控制信号有：

RegDst，用于确定寄存器堆写回地址，其值为 1 则写回地址选 Rd，为 0 则选 Rt。

表 2.1　寄存器堆写回地址选择

RegDst	地址选择
0	Rt
1	Rd

RW(RegisterWrite)，寄存器堆的写使能信号。

MR(MemoryRead)，数据存储器读标志。仅当指令为 lw(load word)时该信号有效。实际上存储器的读出并不需要使能信号，该标志用于在写回阶段选择将要写回寄存器堆的数据。该信号无效，则写回数据取 ALU 的运算结果；有效则写回从数据存储器读出的内容。

MW(MemoryWrite)，数据存储器的写使能信号。仅当指令为 sw(store word)时该信号有效。

Branch，宽度两位的分支标志。0 表非分支指令；1 表 beq(相等则跳转)；2 表 bne(不等则跳转)。

表 2.2　分支信号

指令高 6 位（31：26）	Branch（1：0）
000100（beq）	01
000101（bne）	10
其他	00

AluSrc，选择 ALU 的第二个操作数。其值为 0 则选寄存器堆的第二个输出数据；为 1 则选经扩展的立即数 Ins[15:0]，即指令的低 16 位。

ExI，经扩展的立即数。组原书上的实现，是设置了独立的符号扩展单元，但考虑到逻辑并不复杂，将其一并集成到译码逻辑中。

Mul ，乘法标志。该信号保证冒险检测单元在必要的时刻发现乘法指令，以便及时阻塞流水线，等待乘法指令的执行。

AluOp(ALU Operation)，ALU 的操作码。其功能编码将在介绍 ALU 时再行讨论，详见 3.6 节。

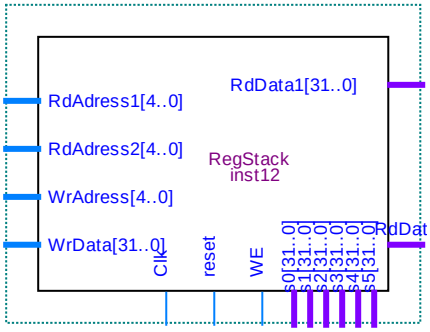
最后，附译码逻辑表如下：

表 2.3 指令译码逻辑

31:26	十进制	5:0	十进制	指令	AluOp	AluSrc	MW	RW	MR	Sign	RegDst
000000	0	**R-Type**									
		00 0000	0	**sll**	1000						
		00 0010	2	**srl**	1001						
		00 0011	3	**sra**	1010						
		01 1000	24	**mult**	0010						
		01 1001	25	**multu**	0010						
		01 1010	26	**div**	0011						
		01 1011	27	**divu**	0011						
		10 0000	32	**add**	0000						
		10 0001	33	**addu**	0000	0	0	1	0	x	1
		10 0010	34	**sub**	0001						
		10 0011	35	**subu**	0001						
		10 0100	36	**and**	0100						
		10 0101	37	**or**	0101						
		10 0110	38	**xor**	0110						
		10 0111	39	**nor**	0111						
		10 1010	42	**slt**	1011						
		10 1011	43	**sltu**	1100						
000010	2			**j**	xxxx	x					
000100	4			**beq**	0001	0	0	0	0	x	x
000101	5			**bne**	0001	0					
001000	8			**addi**	0000					1	
001001	9			**addiu**	0000					0	
001010	10			**slti**	1011					1	
001011	11			**sltiu**	1100	1	0	1	0	0	0
001100	12			**andi**	0100					1	
001101	13			**ori**	0101					1	
001110	14			**xori**	0110					1	
100011	35			**lw**	0000	1	0	1	**1**	1	0
101011	43			**sw**	0000	1	**1**	0	0	1	

2.3 寄存器堆

图 2.1 寄存器堆



接口定义如下：

RdAddress1、RdAddress2，两个读地址。读出数据分别为 RdData1、RdData2.

Clk，时钟信号，写数据采用时钟正沿触发。

WE(Write Enable)，写使能信号。

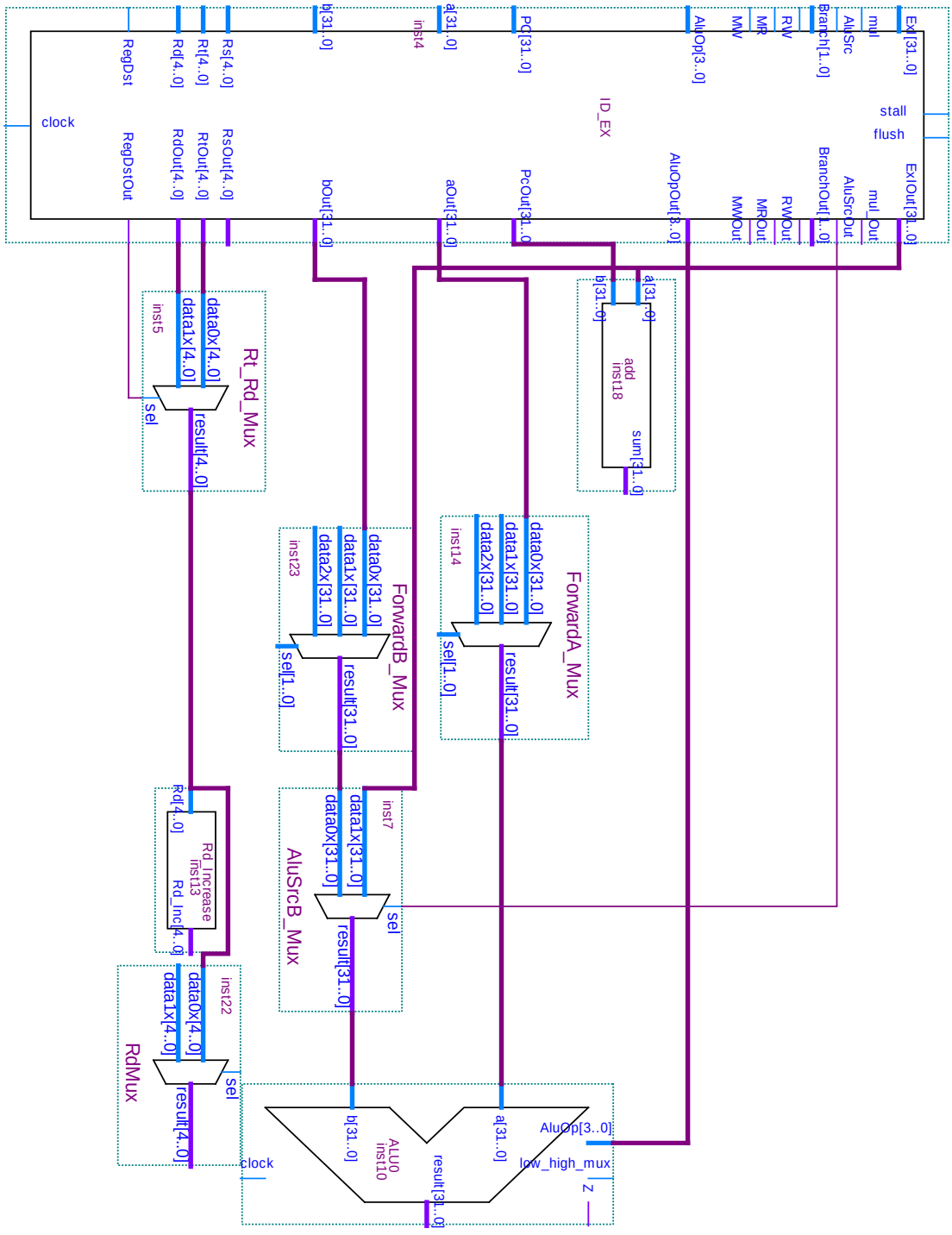
我们在调试时发现，转发单元中只考虑让 ALU 获得“最新”的数据，却忽略了译码阶段。在译码阶段，**如果将要从寄存器堆读取的数据与即将写入的数据的寄存器号相同，则显然欲写入的数据比寄存器堆中的现有数据更“新”，这时就应该将待写入的数据作为读取结果。**修正后的寄存器堆模块代码如下，其中加黑部分解决了上述数据冒险。

```
//////////////////////////////////////////////////////////////////////////////////
module RegStack
(    reset,WrAdress, Clk, WE, RdAdress1, RdAdress2, WrData, RdData1, RdData2,
     s0,s1,s2,s3,s4,s5
);

      input reset;
     input [4:0] WrAdress;
     input Clk;
     input WE;
     input [4:0] RdAdress1;
     input [4:0] RdAdress2;
     input [31:0] WrData;
     output [31:0] RdData1;
     output [31:0] RdData2;
     output [31:0] s0,s1,s2,s3,s4,s5;

     wire t1,t2;
      reg [31:0] s [0:31];

      always @ (posedge Clk)begin
         if(reset)begin
            s[0]=0;
```

```
        s[1]=1;
        s[2]=2;
        s[3]=13;
        s[4]=4;
        s[5]=5;
        end
    else begin
        if(WE)s[WrAdress]<=WrData;
        end
    end

    assign t1=(~|(WrAdress^RdAdress1))&WE;//将要写入且写地址与第一个读
                                          //地址相同
    assign t2=(~|(WrAdress^RdAdress2))&WE;//将要写入且写地址与第二个读
                                          //地址相同
    assign RdData1={{32{t1}}&WrData}|{{32{~t1}}&s[RdAdress1]};
    assign RdData2={{32{t2}}&WrData}|{{32{~t2}}&s[RdAdress2]};
    assign s0=s[0];
    assign s1=s[1];
    assign s2=s[2];
    assign s3=s[3];
    assign s4=s[4];
    assign s5=s[5];

endmodule
////////////////////////////////////////////////////////////////////////////////
```

3.执行

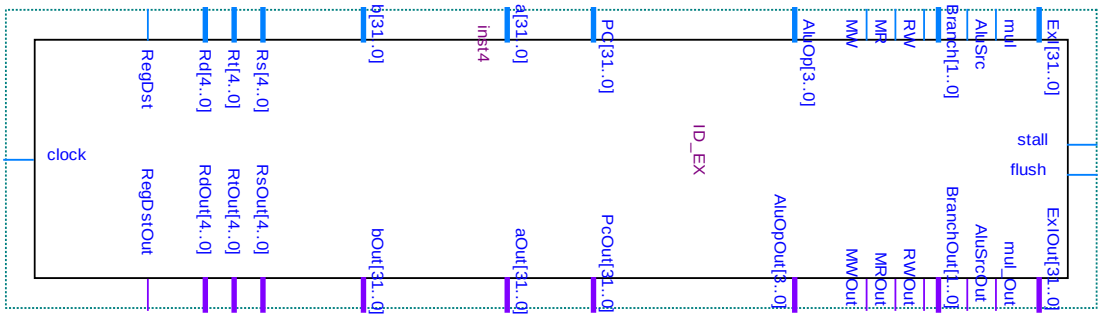
图 3.0　执行阶段



图中的两个 Forward_Mux 是为了实现数据转发而引入的，详见第 6 章。

3.1 ID_EX(译码/执行)流水段寄存器

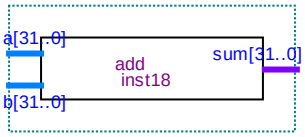
图 3.1　ID_EX(译码/执行)流水段寄存器



如图，右边的 8 个输入输出(ExI,mul,AluSrc,Branch,RW,MR,MW,AluOp)及最左边的 RegDst 来自译码单元，详见 2.2 节。中间的 a、b 为从寄存器堆读取的 32 位数据。PC 为当前指令的下一条指令的地址。

3.2 计算分支地址专用加法器

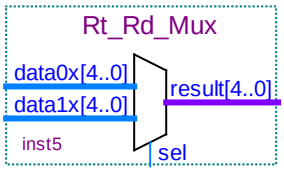
图 3.2　分支地址专用加法器



输入为 PC 和符号扩展的立即数，作加法运算，得分支地址。该单元直接调用 ALU 中设计并实现的 32 位超前进位加法器。与取指阶段计算 PC+4 的加法器类似，该加法器实际只要 30 位，因为在字节寻址情况下，每条指令 4 个字节，其地址低两位必都为 0。

3.3 写回地址 Rd 选择

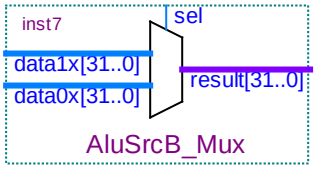
图 3.3　写回地址选择



两个数据输入为 Rt(Data0)、Rd(Data1)，控制输入为 RegDst(sel)。输出的选择见 2.2 节的译码单元。对于 R-Type 指令，寄存器堆写回地址应选 Rd；I-Type 和 lw(访存)指令时则选 Rt。

3.4 ALU 第二个操作数选择

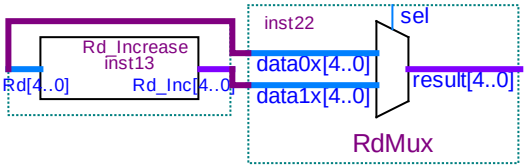
图 3.4



两个备选数据分别为 寄存器输出 b(Data0) 和 扩展成 32 位的立即数 ExI (Data1)。控制信号为 AluSrc(sel)，由译码单元生成。

3.5 为乘法结果高 32 位设置的写回地址生成逻辑

图 3.5



虽然在 3.3 中已经作了寄存器堆写回地址 Rd 的选择，但寄存器堆只有一个写口，一次只能写入 32 位数据，而乘法器的输出为 64 位。为了能将乘法器的运算结果完整的写入寄存器堆，我们采取的策略是：将结果的低 32 位写入 Rd 指定的寄存器，高 32 位的结果则写入(Rd+1)所指定的寄存器。故在此处定制了一个 5 位的超前进位加法器，并取模块名为 Rd_Increase。RdMux 通过控制信号 Low_high_mux 来决定写回地址取 Rd(Data0)还是 Rd+1(Data1)。其中控制信号 Low_high_mux 由冒险检测单元生成，详见 7.2 节。

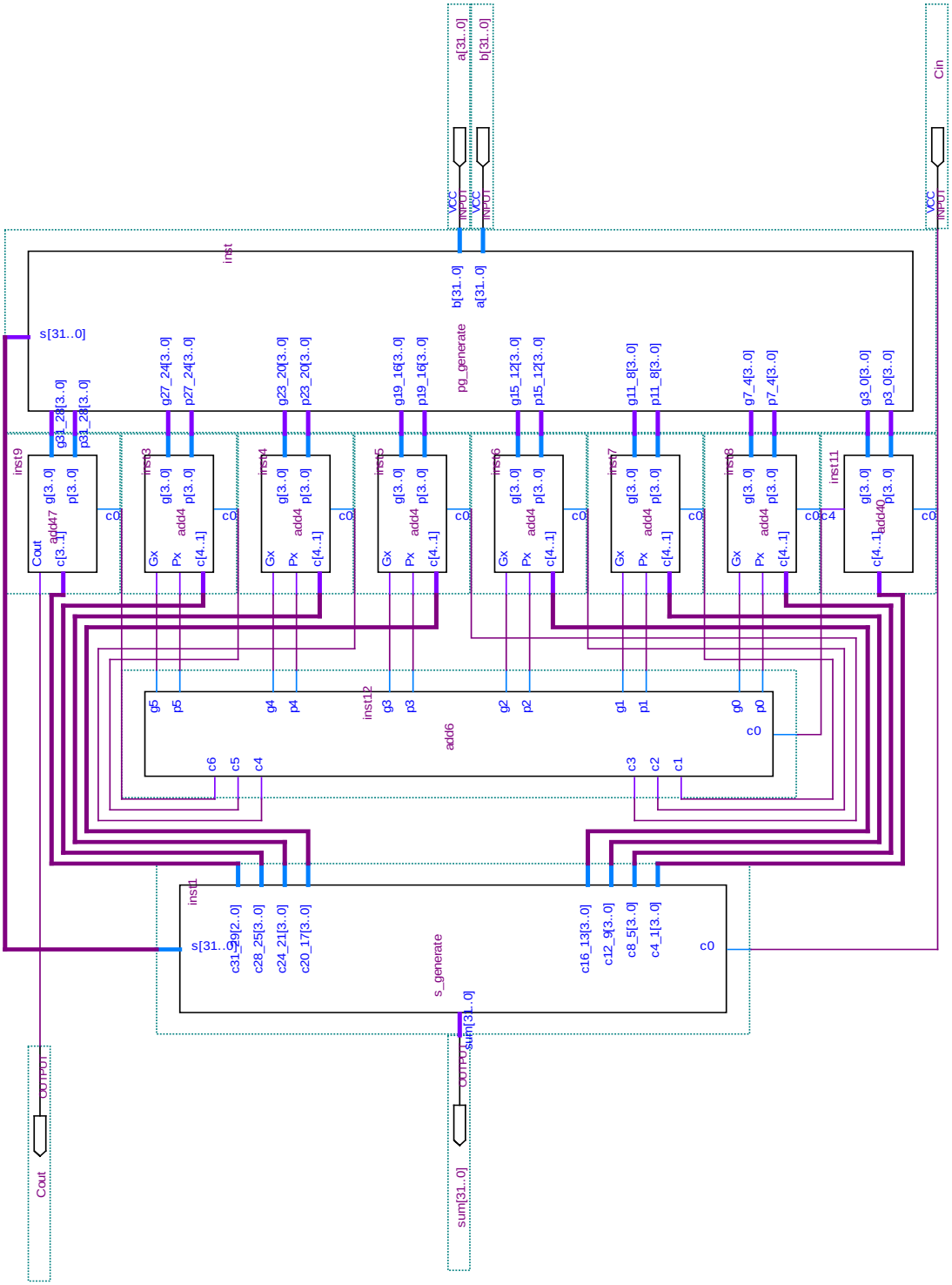
还有一种可选的方案是，指定一个专用寄存器，用于存放乘法结果的高 32 位。但这存在诸多问题，比如连续的乘法运算，后算的高 32 位结果将会把先算的乘法指令的高 32 位结果冲洗掉。

在我们选用的方案中，也有许多潜在的危险，比如程序员书写的乘法指令的 Rd 字段如果是奇数，就很不受欢迎。还有就是如果(Rd+1)所指向的寄存器为某个专用寄存器，这样很可能会导致系统的崩溃。这时要么改进设计，引入更复杂的控制逻辑，要么寄希望于编译器，使可能导致错误的指令不会出现。

3.6 算数逻辑部件 ALU

3.6.1 加法器

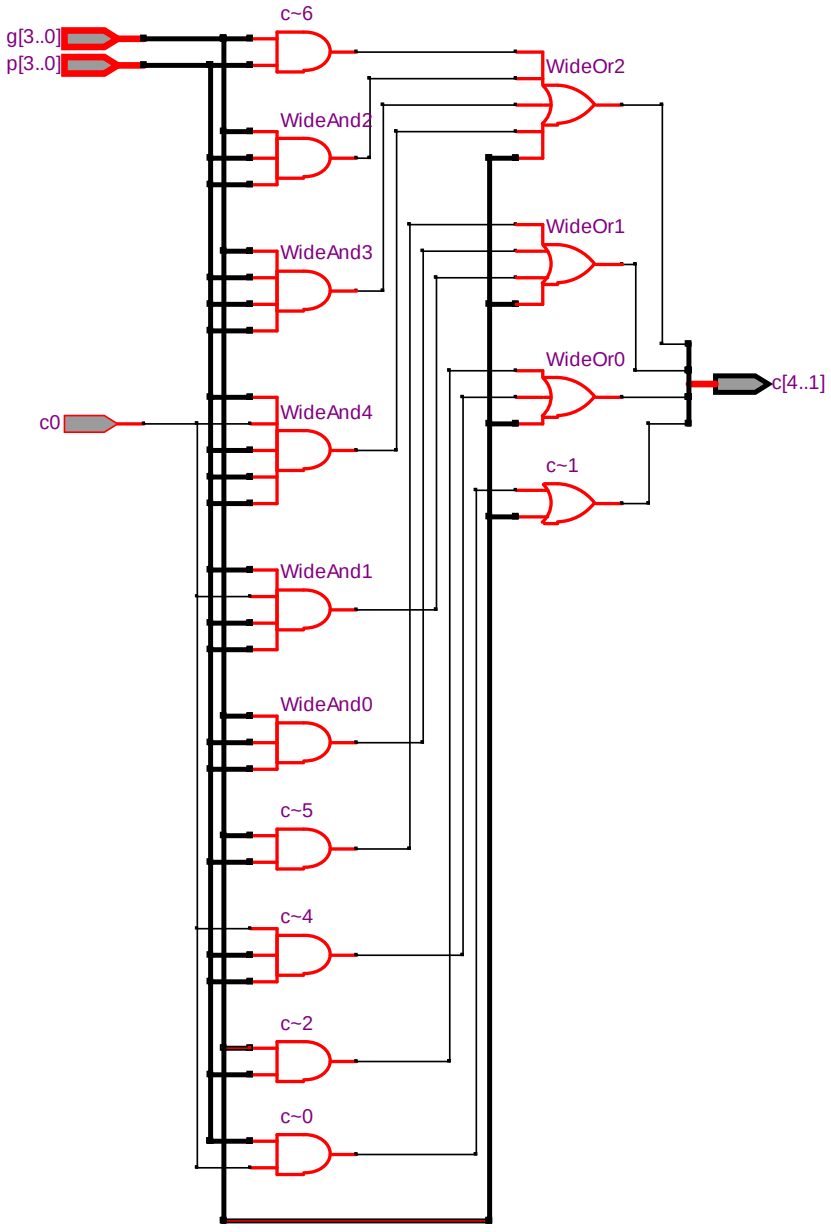
图 3.6.1 超前进位加法器



为了生成超前进位，首先要生成信号 p 和 g，p=a|b，g=a&b。
然后，我们构建了一个 4 位的超前进位逻辑单元，代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////////////
assign c[1]=g[0]|(p[0]&c0);
assign c[2]=|{g[1],p[1]&g[0],&{p[1],p[0],c0}};
assign c[3]=|{g[2],p[2]&g[1],&{p[2],p[1],g[0]},&{p[2],p[1],p[0],c0}};
assign c[4]=|{g[3],p[3]&g[2],&{p[3],p[2],g[1]},&{p[3],p[2],p[1],g[0]},&{p[3],p[2],p[1],p[0],c0}};
////////////////////////////////////////////////////////////////////////////////////////////////
```

综合出来的电路图如下。由图可见，在 4 位的超前进位链中，扇入系数最高为 4+1=5，时延为 2 级门。



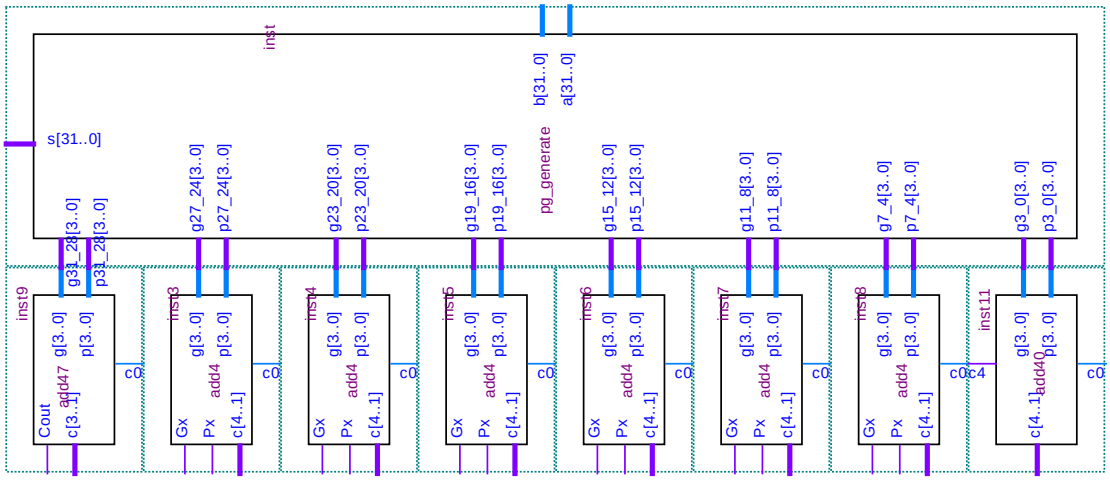
经推导可得：以 4 位超前进位链为一个单元，则单元间的进位也可由超前进位逻辑生成。（具体推导见周报 8，此处不再重复）。为了生成单元间的进位，为 4 位超前进位单元另加两个输出 Gx 和 Px，得代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////////////
module add4(g,p,c0,c,Gx,Px);
```

```
input [3:0] g,p;
input c0;
output [4:1] c;
output Gx,Px;

assign c[1]=g[0]|(p[0]&c0);
assign c[2]=|{g[1],p[1]&g[0],&{p[1],p[0],c0}};
assign c[3]=|{g[2],p[2]&g[1],&{p[2],p[1],g[0]},&{p[2],p[1],p[0],c0}};
assign c[4]=|{g[3],p[3]&g[2],&{p[3],p[2],g[1]},&{p[3],p[2],p[1],g[0]},&{p[3],p[2],p[1],p[0],c0}};
assign Gx=|{g[3],p[3]&g[2],&{p[3],p[2],g[1]},&{p[3],p[2],p[1],g[0]}};
assign Px=&p;

endmodule
```

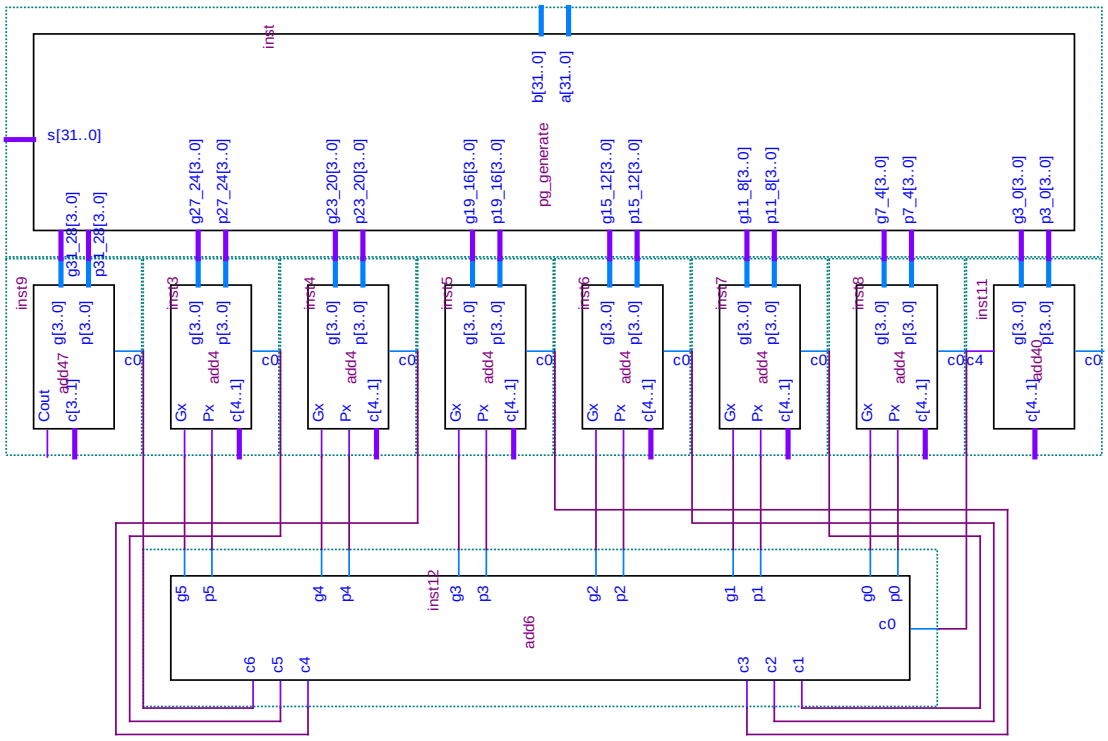
//

在 32 位加法中，共用到 8 个 4 位的超前进位单元，如下图：



其中最低 4 位的进位单元（最右边）的进位 C0 即位整个加法器的输入 Ci。第 2 低位的单元，其进位输入 C4 只要引入最低位单元的进位输出 C4 即可。对于其余 6 个单元的进位输入，要引入一个 6 位的超前进位逻辑单元，其进位输入为 C4，输出的 6 个进位依次引向 6 个 4 位进位单元的进位输入端。该 6 位超前进位单元的扇入系数为 6+1=7，时延为 2 级门。

加入 6 位超前进位单元后如下图：



图中，最低位的 4 位进位单元为 2 级门延迟，可得 C4，将其引至 6 位进位单元和次低位的 4 位进位单元。再经 2 级门延迟，6 位进位单元生成其余 6 个 4 位单元的进位输入 C8,C12,C16,C20,C24,C28，并送往相应单元。最后再经 2 级门延迟，其余 6 个 4 位单元的输出全部产生，即全部 32 位进位全部生成，将其与伪和 S 作按位异或，既得加法结果 sum。

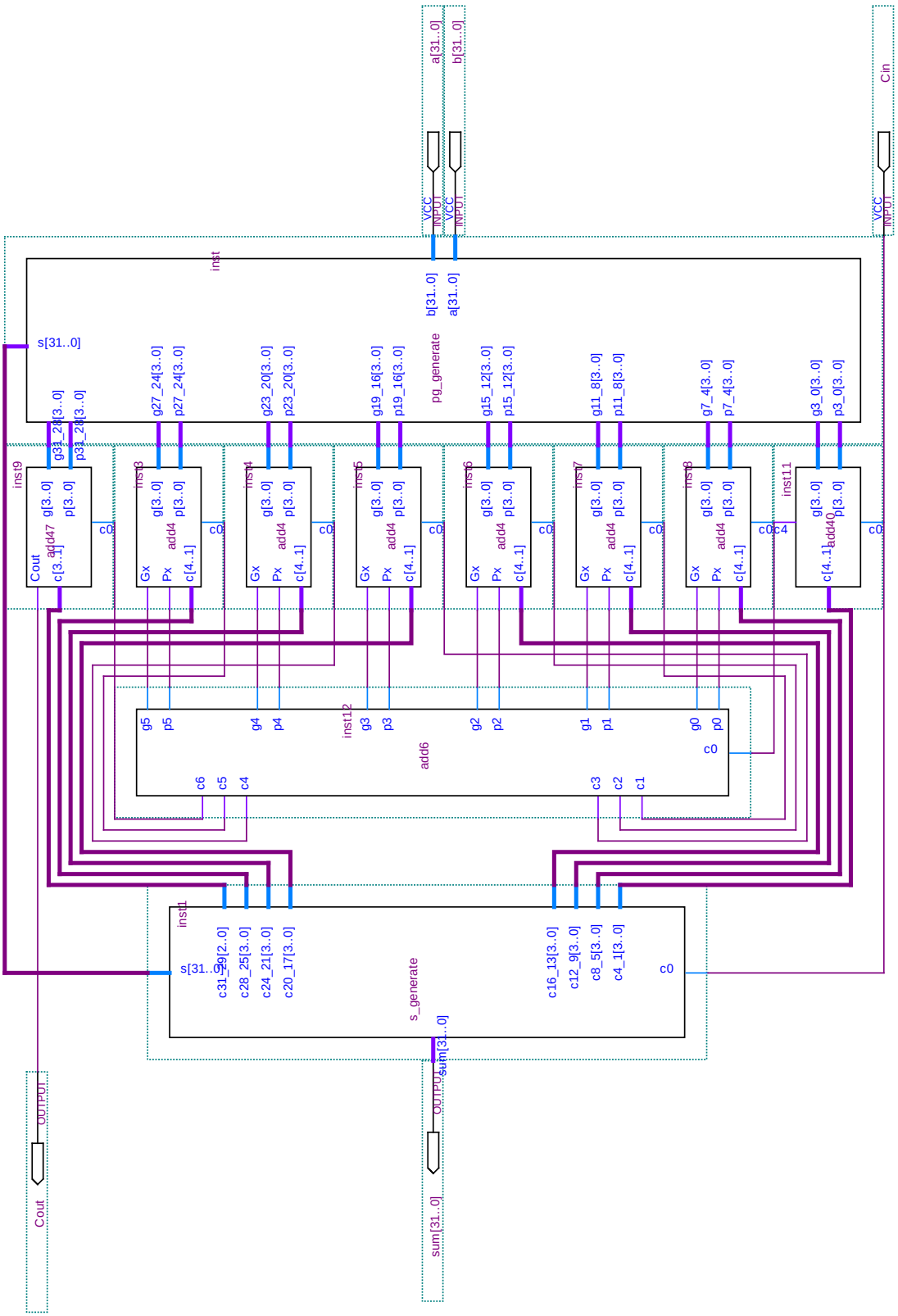
由于 6 位的超前进位单元需要输入 Gx 和 Px，而 Gx 和 Px 由其上 6 个 4 位超前进位单元生成，要 2 级门延迟，故如果将 C0 直接引入 6 位超前进位单元(即将其扩展为 7 位超前进位单元)，并不能加快各 4 位单元的进位输入的生成速度。所以将最低位的 4 位单元与 6 位单元简单级联，当 Gx 与 Px 生成的时候，C4，即 6 位单元的进位输入也刚好生成。

对于我们的 6 位单元，如果引入最高位的 4 位单元的 Gx 与 Px，将其向高位扩展成 7 位单元，也是没有必要的。因为进位 C32 可以和其余进位同时生成，而没有必要比其他的进位(这当然不包括 C8,C12,C16,C20,C24,C28)提前 2 级门延迟生成。

通过以上两个段落的讨论可见，将 4 位单元间的进位逻辑做成 6 位是最为合理的。

最后，将伪和 s(=a^b，在最初产生 p 和 g 的单元里生成)和 8 个 4 位单元输出的 32 位超前进位作异或(sum=s^c)，即得结果。

综上，得加法器设计图如下：(即本节起始处给出的图 3.6.1)


a[31..0]
b[31..0]
Cin
VCC
INPUT
inst
s[31..0]
pg_generate
g27_24[3..0]
p27_24[3..0]
g23_20[3..0]
p23_20[3..0]
g19_16[3..0]
p19_16[3..0]
g15_12[3..0]
p15_12[3..0]
g11_8[3..0]
p11_8[3..0]
g7_4[3..0]
p7_4[3..0]
g3_0[3..0]
p3_0[3..0]
inst9
inst3
inst4
inst5
inst6
inst7
inst8
inst11
g[3..0]
p[3..0]
Cout
add47
add4
add40
c[3..1]
c[4..1]
Gx
Px
c0
c4
inst12
add6
g5
p5
g4
p4
g3
p3
g2
p2
g1
p1
g0
p0
c6
c5
c4
c3
c2
c1
inst1
s[31..29]
c31_29[2..0]
c28_25[3..0]
c24_21[3..0]
c20_17[3..0]
c16_13[3..0]
c12_9[3..0]
c8_5[3..0]
c4_1[3..0]
s_generate
sum[31..0]
OUTPUT
Cout
sum[31..0]

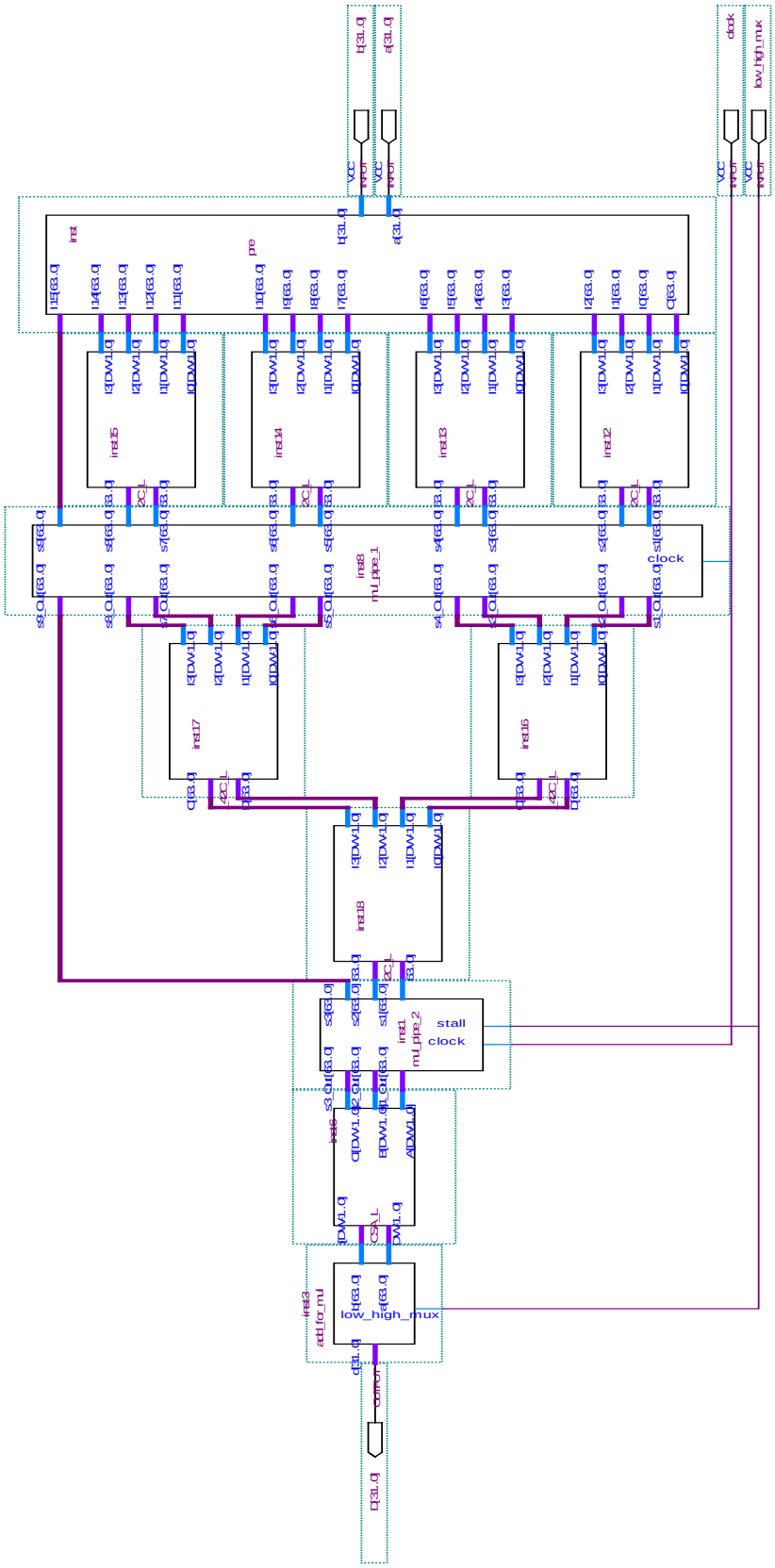
对于减法运算，采用加补码实现。所谓补码，即取反加 1。为此，给加减法单元设置一个标志 a_s，其值为 0 表加法，为 1 表减法。**将该信号扩展成 32 位，与 b(加数/减数)异或**，即可实现：加法时加数取原码，减法时减数取反码。**将标志 a_s 作为加法单元的进位输入**，即可实现：加法时只加原码，减法时加减数的反码再加 1。

综上，得加减法单元的代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////////////
module add_sub(result,c_out,a,b,a_s);

   output [31:0] result;
   output c_out;
   input [31:0] a;
   input [31:0] b;
   input a_s;

   wire c_out1;
   wire [31:0] b1;
   assign b1=b^{32{a_s}};
   add ad(a_s,a,b1,c_out1,result);
   assign c_out=c_out1^a_s;

endmodule
////////////////////////////////////////////////////////////////////////////////////////////////
```

3.6.2 乘法器

图 3.6.2 乘法器



我们的乘法器采用的技术有：基 4-booth 重编码，wallace 树压缩，和超前进位加法器。

所谓 booth 编码，即将一个二进制数的乘权相加表达式中的每一项拆分，然后将相邻项合并。重复合并，即可得高基的 booth 编码。Booth 编码的效果是，**将序列中连续的 1 全换成 0**，然后将连续 1 的左边第一个 0 换成 1，并在连续 1 的最低位处减 1。即 0111…11=1000…00 - 1。这有效减少了部分积的个数。

基 4-booth 重编码如下表：

表 3.1 基 4-booth 编码

A[2n+1]	A[2n]	A[2n-1]	E[n]
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

应用基 4-booth 编码，部分积数目减半，共 16 个。但在编码表中可以看到，E[n]的值可能为负，这时部分积要取被乘数的补码，即取反加 1。取反很容易，但加 1 的代价太大，以我们实现的 32 位超前进位加法器算，要 9 级门延迟。为此，我们**设置第 17 个部分积，专门用于存放其余 16 个部分积可能产生的“加 1”**，16 个部分积只是 被乘数 的原码或反码。

综上，给出 booth 编码的代码如下：

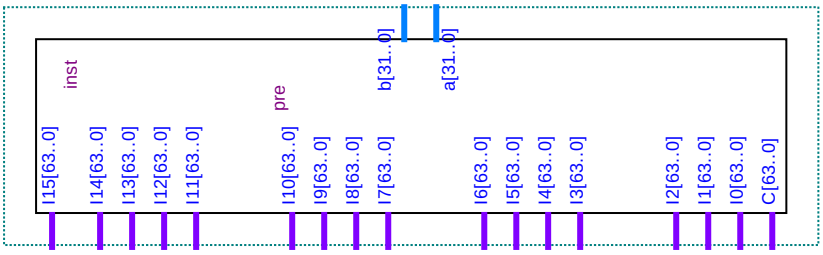
```
////////////////////////////////////////////////////////////////////////////////////
    module booth (
                Encode,
                Source,
                Result,
                Carry
               );
    parameter DW = 32;
    input [2:0]    Encode;
    input [DW-1:0]    Source;
    output[DW:0]    Result;
    output[1:0]    Carry;

    wire                Add_Sub,// add(0) or sub(1)
                        Once_Valid,// once is valid if it is '1' else zero
                        Twice_Enable;// twice is valid when it is '1' else zero
    assign Add_Sub=Encode[2];
    assign Once_Valid=Encode[1]^Encode[0];
    assign Twice_Enable = ((Encode == 3'b011) | (Encode == 3'b100));
```

```
    assign Result = (({Source[DW-1],Source} ^ {(DW+1){Add_Sub}}) & {(DW+1){Once_Valid}}) |
                    (({Source,1'b0} ^ {(DW+1){Add_Sub}}) & {(DW+1){Twice_Enable}});
    assign Carry = {1'b0,(Add_Sub & (Once_Valid | Twice_Enable))};
endmodule
```

//

在乘法器的 pre 模块中，调用 booth 模块，生成 17 个部分积，并将其扩展为 64 位。实际上，扩展成 64 位并不是必须的。这一步主要是为了简化 wallace 树的设计。如果不将部分积统一为 64 位，则后期压缩时各 4-2 或 3-2 压缩模块就要考虑输入输出的位数问题，扩展位数的逻辑也将异常复杂，且容易出错。扩展时，高位补符号，低位补 0。

Pre 模块的框图如下，输入为被乘数和乘数，输出为经 booth 编码的 17 个部分积。代码冗长而没有实质性的复杂逻辑，此处略去。



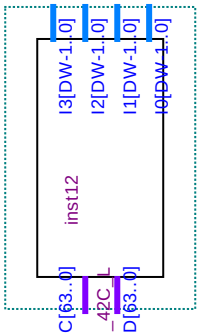
下面说明 wallace 树。

Walllace 树的基本构件是压缩器，有 4-2 压缩和 3-2 压缩。

所谓 4-2 压缩，就是将 4 个加数转化为 2 个加数，且这 2 个加数的和与原来 4 个加数的和相同。3-2 压缩器原理与此相同。

具体实现时，全加器即为 3-2 压缩器；两个全加器级联，得 1 位 5-3 压缩器，再将 5-3 压缩器的低位 Cout 与高位 Cin 相连，即得任意位数的 4-2 压缩器。

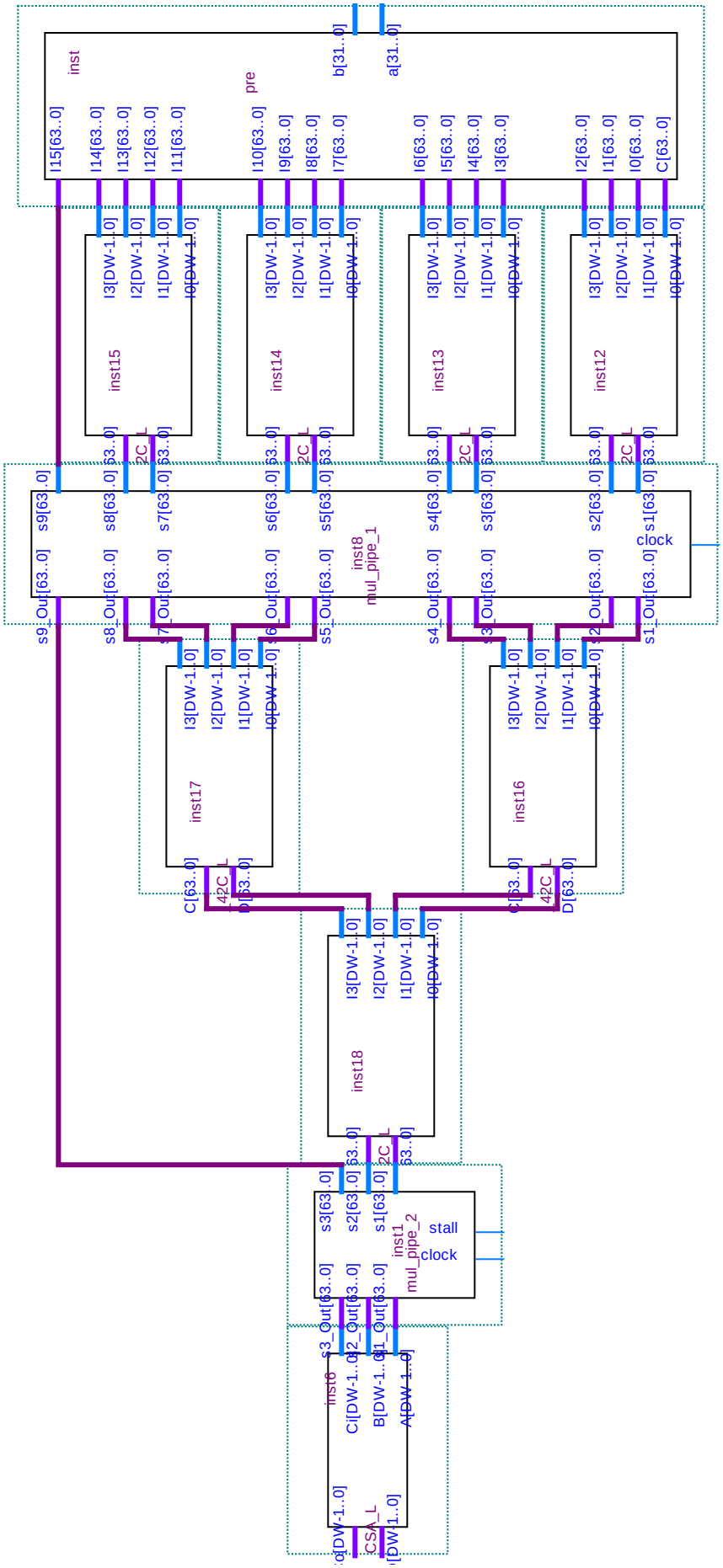
以下是 4-2 压缩器的框图和代码，输入为 4 个加数 I0~I3，输出为两个加数 C 和 D。需要特别说明的是，**要压缩加数的个数，必然要扩展加数的位数**，因为 4 个加数的和可能会向高位进 2 位，而两个加数的和至多能进 1 位，所以两个加数的位数应该比 4 个加数要多 1~2 位。但在我们的实现中，却并没有将 64 位的输入在输出端扩展为 65 或 66 位。这是因为：**两个 32 位数相乘，积不会超过 64 位。**而高位的加数不会影响到低位的结果。所以**压缩中产生的加数中超过 64 位的部分，全部可以忽略**。



```
////////////////////////////////////////////////////////////////////////////////////////
    module _42C_L(I0,I1,I2,I3,D,C);
    parameter DW=64;

    input [DW-1:0] I0,I1,I2,I3;
    output [63:0] D;
    output [63:0] C;

    wire [DW:0] D1;
    wire [DW+1:1] C1;

    wire [DW-1:0] TXR,TAO,TOA;

    assign TXR=I0^I1^I2^I3;
    assign TAO=(I0&I1)|(I2&I3);
    assign TOA=(I0|I1)&(I2|I3);

    assign D1={TXR[DW-1],TXR}^{TOA,1'b0};
    assign
C1=({TXR[DW-1],TXR}&{TOA,1'b0})|((~{TXR[DW-1],TXR})&{TAO[DW-1],TAO});

    assign D=D1[63:0];
    assign C[63:1]=C1[63:1];
    assign C[0]=1'b0;

    endmodule
////////////////////////////////////////////////////////////////////////////////////////
```

3-2 压缩的原理与 4-2 压缩类似，此处略去。

压缩模块构建完毕，下面步入正题，来看我们的 wallace 树，如下图。


inst
pre
b[31..0]
a[31..0]
I15[63..0]
I14[63..0]
I13[63..0]
I12[63..0]
I11[63..0]
I10[63..0]
I9[63..0]
I8[63..0]
I7[63..0]
I6[63..0]
I5[63..0]
I4[63..0]
I3[63..0]
I2[63..0]
I1[63..0]
I0[63..0]
C[63..0]
inst15
inst14
inst13
inst12
I3[DW-1..0]
I2[DW-1..0]
I1[DW-1..0]
I0[DW-1..0]
s8[63..0]
s7[63..0]
s6[63..0]
s5[63..0]
s4[63..0]
s3[63..0]
s2[63..0]
s1[63..0]
s9[63..0]
2C_L
inst8
mul_pipe_1
clock
s9_Out[63..0]
s8_Out[63..0]
s7_Out[63..0]
s6_Out[63..0]
s5_Out[63..0]
s4_Out[63..0]
s3_Out[63..0]
s2_Out[63..0]
s1_Out[63..0]
inst17
inst16
C[63..0]
D[63..0]
42C_L
inst18
s3[63..0]
s2[63..0]
s1[63..0]
inst1
mul_pipe_2
stall
clock
s3_Out[63..0]
s2_Out[63..0]
s1_Out[63..0]
inst6
Ci[DW-1..0]
B[DW-1..0]
A[DW-1..0]
CSA_L
o[DW-1..0]
[DW-1..0]

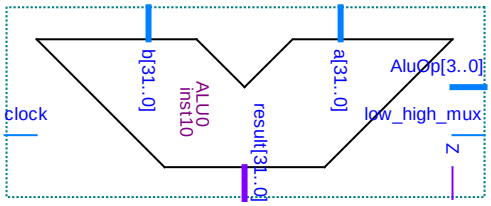
图中两个 mul_pipe 模块是流水段寄存器，关于乘法运算的流水化，稍后再作讨论。

我们使用的 wallace 树几乎全部是 4-2 压缩器，只在最后 1 级使用 3-2 压缩。4-2 压缩有 6 级门延迟，3-2 压缩为 4 级门延迟。整个 Wallace 树共 22(=6*3+4)级门延迟。

我们本打算采用**跳跃式 wallace 树**的结构，可以减少 2 级门延迟。但跳跃式结构的基本原理是将先产生的信号输出，后产生的信号送到更下层的模块。这样**不利于乘法器的流水化**。虽然我们实际上并没有实现乘法单元的流水化，因为时间不足，但我们为其后的改进预留了空间。实际上，乘法本身已经部分实现流水化，但在流水线上传递的不应该只有乘法的操作数，还应该有整条指令的所有控制信号。所以还称不上乘法流水化。

我们为乘法单元分配了 4 个周期，其中第 3 个周期末就已经算出结果的低 32 位，第 4 个周期末算出结果的高 32 位。对这 64 位结果的写回，具体技巧见 7.2 节。

3.6.3 ALU 模块综合



ALU 模块有 5 个输入，定义如下：

a 和 b 是两个 32 位操作数。

AluOp 用于指定运算类型，具体如下表：

表 3.2　ALU 功能编码

AluOp	运算功能选择
0000	ADD
0001	SUB
0010	MUL
0011	DIV
0100	AND
0101	OR
0110	XOR
0111	NOR
1000	SLL
1001	SRL
1010	SRA
1011	SLT
1100	SLTU

Clock 和 low_high_mux 是为乘法器特设的。Clock 是时钟，用于触发乘法器的流水段寄存器。当乘法进行到第 3 个周期末，low_high_mux 为 0，输出低 32 位加法器的结果。第 4 个周期末，low_high_mux 为 1，输出高 32 位加法器的结

果。

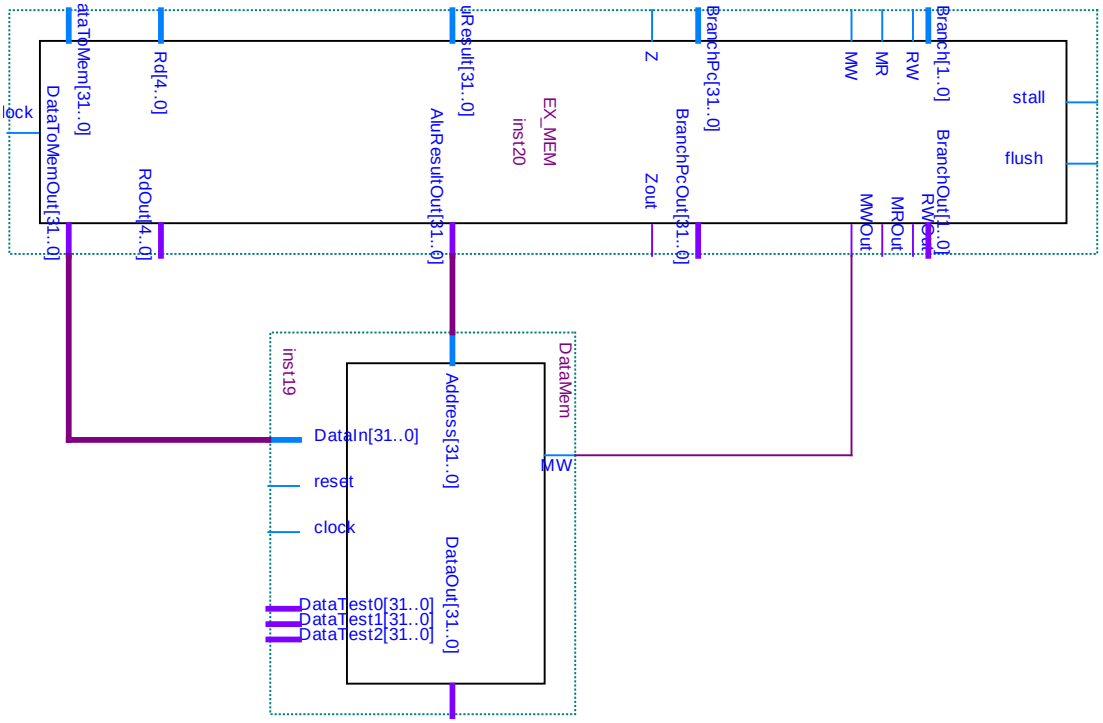
ALU 的两个输出端口定义如下：

Result 为 32 位运算结果。

Z 为零标志，用于分支指令。该信号有效表 ALU 的运算结果为 0。

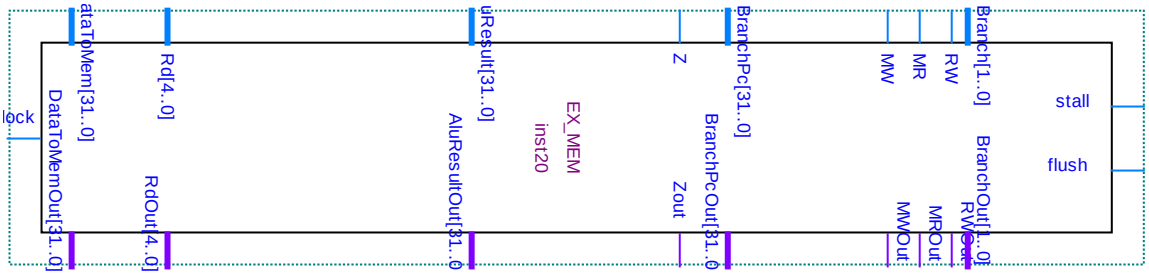
4.访存

图 4.0 访存阶段



4.1 EX_MEM(执行/访存)流水段寄存器

图 4.1 EX_MEM(执行/访存)流水段寄存器



传递的各信号定义如下：

Branch，分支标志。

RW，寄存器写使能。

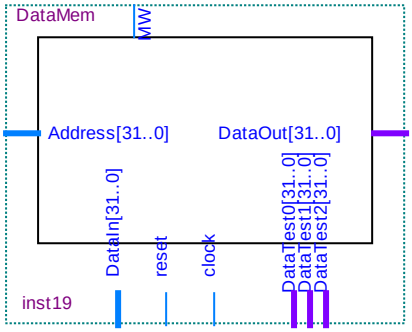
MR，存储器读标志。

MW，存储器写使能

BranchPc，分支地址。
Z，零标志。
AluResult，ALU 的运算结果。
Rd，寄存器堆写回地址。
DataToMem，要写入存储器的数据。

4.2 数据存储器

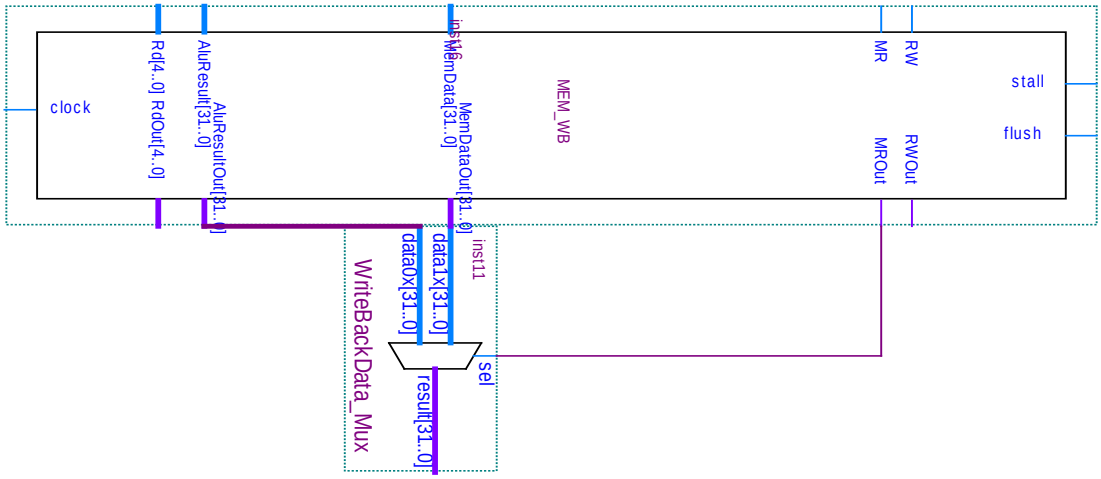
图 4.2 数据存储器



端口定义如下：
MW，存储器写使能。
Address，地址，读或写。
DataIn，待写入的数据。
Reset，复位，用于测试，在该信号驱动下写入初始测试数据。
Clock，时钟信号，存储器的写采用时钟正延触发。
DataOut，读出的数据。

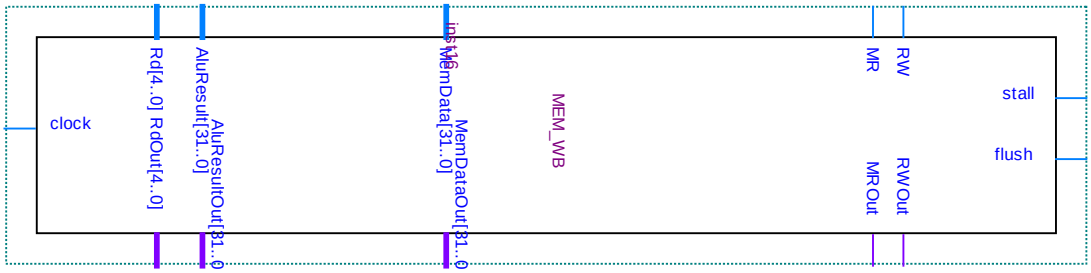
5.写回

图 5.0 写回阶段



5.1 MEM_WB(访存/写回)流水段寄存器

图 5.1　MEM_WB 流水段寄存器



传递的信号定义如下：

RW，寄存器堆写使能。

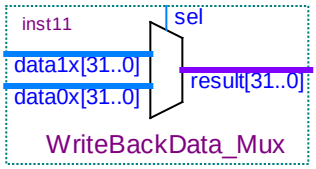
MR，存储器读标志，用于决定写回存储器读取的数据，还是 ALU 的运算结果。

MemData，从存储器读取的数据。

Aluresult，ALU 的运算结果。

Rd，寄存器堆写回地址。

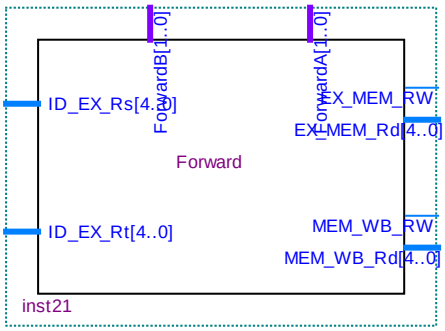
5.2 写回数据选择



输入是 MemData(data1)和 AluResult(data0)，控制信号是 MR(sel)，输出是将要写回寄存器堆的数据。

6.转发

图 6.0 转发单元



转发单元用于确保处于执行阶段的指令，其操作数是“最新的”。如果处于访存或写回阶段的指令，其 RW(寄存器堆写使能)有效，且写回地址和 ALU 某操作数的来源寄存器相同，则显然将要写回的数据比 ID_EX 流水段寄存器中存储的

数据更“新”，这时就应该进行数据的转发。

需要注意的是，如果处于访存阶段和处于写回阶段的指令都可用于转发，且转发的对象相同，则由于处于访存阶段的数据比处于写回阶段的更“新”，这时应该转发处于访存阶段的指令的写回数据。即访存阶段的写回数据优先级更高。

考虑这样一种情况，执行阶段的指令数据依赖于访存阶段指令的访存结果。实际上这种情况是不会发生的，因为冒险检测单元在访存指令处于 EX 阶段时就可检测出它和其后一条指令的数据依赖(load-use 冒险)，并在这两条指令之间插入一个 NOP。这样，当 use 指令进入执行阶段时，访存指令已经访存完毕，进入写回阶段，需要的数据可以顺利转发。

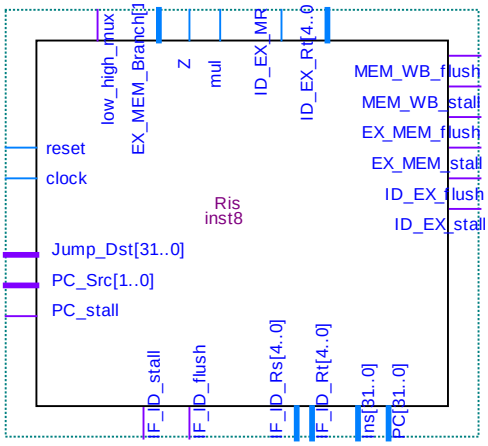
综上，得转发单元的代码如下：

```
//////////////////////////////////////////////////////////////////////////////////
module
Forward(EX_MEM_RW,EX_MEM_Rd,MEM_WB_RW,MEM_WB_Rd,ID_EX_Rs,ID_EX_Rt,ForwardA,ForwardB );

input EX_MEM_RW;
input [4:0] EX_MEM_Rd;
input MEM_WB_RW;
input [4:0] MEM_WB_Rd;
input [4:0] ID_EX_Rs;
input [4:0] ID_EX_Rt;

output reg [1:0] ForwardA;
output reg [1:0] ForwardB;

always@(*)begin
  if(EX_MEM_RW
      &&EX_MEM_Rd!=0
      &&EX_MEM_Rd==ID_EX_Rs)
      ForwardA=2'b10;
  else if(MEM_WB_RW
            &&MEM_WB_Rd!=0
            &&MEM_WB_Rd==ID_EX_Rs)
            ForwardA=2'b01;
  else ForwardA=0;

  if(EX_MEM_RW
      &&EX_MEM_Rd!=0
      &&EX_MEM_Rd==ID_EX_Rt)
      ForwardB=2'b10;
  else if(MEM_WB_RW
            &&MEM_WB_Rd!=0
            &&MEM_WB_Rd==ID_EX_Rt)
            ForwardB=2'b01;
```

```
    else ForwardB=0;
end
endmodule
//////////////////////////////////////////////////////////////////////////////////
```

7. 冒险检测

图 7.0 冒险检测单元



冒险检测单元的功能是，监控流水线状态，决定下条指令的地址来源，并且在必要的时刻向流水线中加入空指令 nop，或阻塞流水线。

其输入端口定义如下：

Reset，复位信号，用于初始化寄存器 mul_state，该寄存器用于乘法运算，相当于一个计数的状态机，记录流水线已经为乘法指令等待(阻塞)了多少个周期。

Clock，时钟信号，mul_state 采用时钟正沿触发。

mul，乘法运算标志，来自 ID_EX 流水段寄存器，告知冒险检测单元：处于执行阶段的指令是否是一条乘法指令。

EX_MEM_Branch，分支信号，来自 EX_MEM 流水段寄存器。因为零标志 Z 在此时才算出来。

Z，零标志，来自 EX_MEM 流水段寄存器。

IF_ID_Rs，来自 IF_ID 流水段寄存器，是其相对应指令的寄存器堆第一个读地址。

IF_ID_Rt，来自 IF_ID 流水段寄存器，是其相对应指令的寄存器堆第二个读地址。

ID_EX_MR，处于译码阶段的指令的存储器读标志。

ID_EX_Rt，处于译码阶段的指令的写回地址(如果这是一条读存储器的指令的话)。

Ins，处于译码阶段的指令，用于判断其是否是跳转指令，并计算跳转地址。

PC，用其高 4 位和 Ins 的低 26 位拼接，并左移两位(即低两位补 0)，构成跳

转地址。

该模块的输出就是各流水段寄存器的阻塞(stall)和冲洗(flush)信号，此处不逐条列出。

另有几个输出的定义如下：

Pc_Src，下条指令的地址选择信号。可选的地址来源有：PC+4，跳转地址和分支地址。

Jump_Dst，跳转地址。

Low_high_mux，用于乘法指令，如果不是乘法指令，则该信号置 0。如果是乘法指令，则在乘法指令执行的前 3 个周期置 0，第 4 个周期置 1。该信号置 0 则写回地址选指令中指定的写回地址。置 1 则选指定的地址加 1 的结果。

7.1 分支冒险

分支冒险的检测处于流水线的最前沿，故其优先级也最高。若果检测到处于访存阶段的指令是分支指令，且据 Z 标志得知确实需要分支，则流水线中处于取指、译码和执行阶段的指令都应该清除，且下条指令的地址应取分支地址。其逻辑如下：

```
////////////////////////////////////////////////////////////////////////////////////
if((EX_MEM_Branch==1&&Z)||    //beq
                (EX_MEM_Branch==2&&(~Z)))begin      //bne
    Jump_Dst=0;
    PC_Src=2;
    PC_stall=0;

    IF_ID_flush=0;
    IF_ID_stall=0;

    ID_EX_flush=1;
    ID_EX_stall=0;
    EX_MEM_flush=1;
    EX_MEM_stall=0;

    MEM_WB_flush=0;
    MEM_WB_stall=0;
    low_high_mux=0;
    end
////////////////////////////////////////////////////////////////////////////////////
```

7.2 等待乘法单元

如果检测到处于执行阶段的指令是乘法指令，则对该指令的执行周期数进行计数。

在乘法运算的前 2 个周期，阻塞 EX_MEM 及其之前的所有流水段寄存器，其后的指令允许其向前行进，但当其向前传递后，原流水段寄存器应该被冲洗。

当乘法运算在 ALU 中执行 3 个周期，低 32 位的结果有效，让该乘法指令向前行进，写回地址取指令中指定的寄存器号，但同时阻塞 EX_MEM 及其之前的流水段寄存器，从而产生乘法指令的一个副本。

在乘法运算的第 4 个周期，释放整个流水线，但乘法指令的写回地址选择原地址加 1，即将积的高 32 位写回到指令指定的寄存器的后 1 号寄存器。

此逻辑写成代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////////
if(mul)begin//mul
    case (mul_state)
      0:begin
        Jump_Dst=0;
        PC_stall=1;
        PC_Src=0;
        IF_ID_stall=1;
        IF_ID_flush=0;
        ID_EX_stall=1;
        ID_EX_flush=0;
        EX_MEM_stall=0;
        EX_MEM_flush=1;
        MEM_WB_stall=0;
        MEM_WB_flush=0;
        low_high_mux=0;
        end
      1:begin
        Jump_Dst=0;
        PC_stall=1;
        PC_Src=0;
        IF_ID_stall=1;
        IF_ID_flush=0;
        ID_EX_stall=1;
        ID_EX_flush=0;
        EX_MEM_stall=1;
        EX_MEM_flush=0;
        MEM_WB_stall=0;
        MEM_WB_flush=1;
        low_high_mux=0;
        end
      2:begin
        Jump_Dst=0;
        PC_stall=1;
        PC_Src=0;
        IF_ID_stall=1;
```

```
            IF_ID_flush=0;
            ID_EX_stall=1;
            ID_EX_flush=0;
            EX_MEM_stall=0;
            EX_MEM_flush=0;
            MEM_WB_stall=0;
            MEM_WB_flush=0;
            low_high_mux=0;
            end
          3:begin
            Jump_Dst=0;
            PC_stall=0;
            PC_Src=0;
            IF_ID_stall=0;
            IF_ID_flush=0;
            ID_EX_stall=0;
            ID_EX_flush=0;
            EX_MEM_stall=0;
            EX_MEM_flush=0;
            MEM_WB_stall=0;
            MEM_WB_flush=0;
            low_high_mux=1;
            end
      endcase
      end
////////////////////////////////////////////////////////////////////////////////////////
```

7.3 Load-use 冒险

如果处于执行阶段的指令是读存储器(load)指令，且其写回地址与其后的处于译码阶段的指令的读寄存器堆地址相同，则产生 load-use 冒险，这是应该在这两条指令之间引入一个 nop，使得当 use 指令进入执行阶段时，load 指令已经访存完毕，数据可以被转发单元转发。

实现代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////
if(ID_EX_MR&&          //load-use
            ((ID_EX_Rt==IF_ID_Rs)
           ||(ID_EX_Rt==IF_ID_Rt))
              )begin
                 //PC_Src=2'b00;
      Jump_Dst=0;
      PC_stall=1;
      PC_Src=0;
```

```
    IF_ID_flush=0;
    IF_ID_stall=1;

    ID_EX_flush=1;
    ID_EX_stall=0;

    EX_MEM_flush=0;
    EX_MEM_stall=0;

    MEM_WB_flush=0;
    MEM_WB_stall=0;
    low_high_mux=0;
    end
////////////////////////////////////////////////////////////////////////////////////////
```

7.4 Jump(跳转)冒险

如果处于译码阶段的指令是一条跳转指令，则下条指令的地址应选冒险单元算出的跳转地址。代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////
if(Ins[31:26]==2)begin//jump
    Jump_Dst={PC[31:28],Ins[25:0],2'b00};
    PC_Src=1;
    PC_stall=0;
    IF_ID_stall=0;
    IF_ID_flush=0;
    ID_EX_stall=0;
    ID_EX_flush=0;
    EX_MEM_stall=0;
    EX_MEM_flush=0;
    MEM_WB_stall=0;
    MEM_WB_flush=0;
    low_high_mux=0;
    end
////////////////////////////////////////////////////////////////////////////////////////
```

综上，给出冒险检测单元的代码如下：

```
////////////////////////////////////////////////////////////////////////////////////////
module Ris(reset,clock,
            mul,low_high_mux,//mul
            EX_MEM_Branch,Z,//branch
            IF_ID_Rs,IF_ID_Rt,ID_EX_MR,ID_EX_Rt,//load-use
            Ins,PC,//jump
            PC_stall,
            IF_ID_stall,IF_ID_flush,
```

```
                ID_EX_stall,ID_EX_flush,
                EX_MEM_stall,EX_MEM_flush,
                MEM_WB_stall,MEM_WB_flush,
                PC_Src,Jump_Dst
                );

input reset;
input clock;
input mul;//mul

input [1:0] EX_MEM_Branch; //branch
//input C;
input Z;

input [4:0] IF_ID_Rs;           //load-use
input [4:0] IF_ID_Rt;
input ID_EX_MR;
input [4:0] ID_EX_Rt;

input [31:0] Ins;       //jump
input [31:0] PC;

output reg [1:0] PC_Src;
output reg [31:0] Jump_Dst;
output reg PC_stall;
output reg IF_ID_stall;
output reg IF_ID_flush;
output reg ID_EX_stall;
output reg ID_EX_flush;
output reg EX_MEM_stall;
output reg EX_MEM_flush;
output reg MEM_WB_stall;
output reg MEM_WB_flush;
output reg low_high_mux;

reg [1:0] mul_state;

/*initial begin

   ID_EX_flush=1;
   EX_MEM_flush=1;
   MEM_WB_flush=1;

end*/
```

```
always@(posedge clock)begin
   if(reset)
      mul_state<=0;
   else if(mul)begin
      case(mul_state)
         0:mul_state<=1;
         1:mul_state<=2;
         2:mul_state<=3;
         3:mul_state<=0;
         endcase
      end
   end


always@(reset,mul,EX_MEM_Branch,Z,IF_ID_Rs,IF_ID_Rt,ID_EX_MR,ID_EX_Rt,Ins,PC
)begin
   if(reset)begin
      Jump_Dst=0;
      PC_stall=1;
      PC_Src=0;
      IF_ID_stall=0;
      IF_ID_flush=1;
      ID_EX_stall=0;
      ID_EX_flush=1;
      EX_MEM_stall=0;
      EX_MEM_flush=1;
      MEM_WB_stall=0;
      MEM_WB_flush=1;
      low_high_mux=0;
      end

   else if((EX_MEM_Branch==1&&Z)||    //beq
                  (EX_MEM_Branch==2&&(~Z)))begin      //bne
      Jump_Dst=0;
      PC_Src=2;
      PC_stall=0;

      IF_ID_flush=0;
      IF_ID_stall=0;

      ID_EX_flush=1;
      ID_EX_stall=0;
      EX_MEM_flush=1;
      EX_MEM_stall=0;
```

```
    MEM_WB_flush=0;
    MEM_WB_stall=0;
    low_high_mux=0;
    end

else if(mul)begin//mul
    case (mul_state)
        0:begin
            Jump_Dst=0;
            PC_stall=1;
            PC_Src=0;
            IF_ID_stall=1;
            IF_ID_flush=0;
            ID_EX_stall=1;
            ID_EX_flush=0;
            EX_MEM_stall=0;
            EX_MEM_flush=1;
            MEM_WB_stall=0;
            MEM_WB_flush=0;
            low_high_mux=0;
            end
        1:begin
            Jump_Dst=0;
            PC_stall=1;
            PC_Src=0;
            IF_ID_stall=1;
            IF_ID_flush=0;
            ID_EX_stall=1;
            ID_EX_flush=0;
            EX_MEM_stall=1;
            EX_MEM_flush=0;
            MEM_WB_stall=0;
            MEM_WB_flush=1;
            low_high_mux=0;
            end
        2:begin
            Jump_Dst=0;
            PC_stall=1;
            PC_Src=0;
            IF_ID_stall=1;
            IF_ID_flush=0;
            ID_EX_stall=1;
            ID_EX_flush=0;
```

```
            EX_MEM_stall=0;
            EX_MEM_flush=0;
            MEM_WB_stall=0;
            MEM_WB_flush=0;
            low_high_mux=0;
            end
          3:begin
            Jump_Dst=0;
            PC_stall=0;
            PC_Src=0;
            IF_ID_stall=0;
            IF_ID_flush=0;
            ID_EX_stall=0;
            ID_EX_flush=0;
            EX_MEM_stall=0;
            EX_MEM_flush=0;
            MEM_WB_stall=0;
            MEM_WB_flush=0;
            low_high_mux=1;
            end
        endcase
        end

    else if(ID_EX_MR&&          //load-use
              ((ID_EX_Rt==IF_ID_Rs)
            ||(ID_EX_Rt==IF_ID_Rt))
                )begin
                    //PC_Src=2'b00;
        Jump_Dst=0;
        PC_stall=1;
        PC_Src=0;

        IF_ID_flush=0;
        IF_ID_stall=1;

        ID_EX_flush=1;
        ID_EX_stall=0;

        EX_MEM_flush=0;
        EX_MEM_stall=0;

        MEM_WB_flush=0;
        MEM_WB_stall=0;
        low_high_mux=0;
```

```
        end
    else if(Ins[31:26]==2)begin//jump
        Jump_Dst={PC[31:28],Ins[25:0],2'b00};
        PC_Src=1;
        PC_stall=0;
        IF_ID_stall=0;
        IF_ID_flush=0;
        ID_EX_stall=0;
        ID_EX_flush=0;
        EX_MEM_stall=0;
        EX_MEM_flush=0;
        MEM_WB_stall=0;
        MEM_WB_flush=0;
        low_high_mux=0;
        end
    else begin
        Jump_Dst=0;
        PC_Src=0;
        PC_stall=0;
        IF_ID_stall=0;
        IF_ID_flush=0;
        ID_EX_stall=0;
        ID_EX_flush=0;
        EX_MEM_stall=0;
        EX_MEM_flush=0;
        MEM_WB_stall=0;
        MEM_WB_flush=0;
        low_high_mux=0;
        end
end
endmodule
//////////////////////////////////////////////////////////////////////////////////
```

8.仿真验证

我们使用 reset 信号驱动，向各存储单元内写入了测试数据和指令。测试指令如下：其中括号左边为指令地址，括号内为指令序号

0(0)： lw R2,0(R4);
4(1)： add R2,R1,R2;
8(2)： add R2,R1,R2;
12(3)：addi R2,R2,2;
16(4)：beq R2,R3,1;

20(5)：j 1;

24(6)：sw R2,4(R4);

28(7)：mult R4,R2,R3;

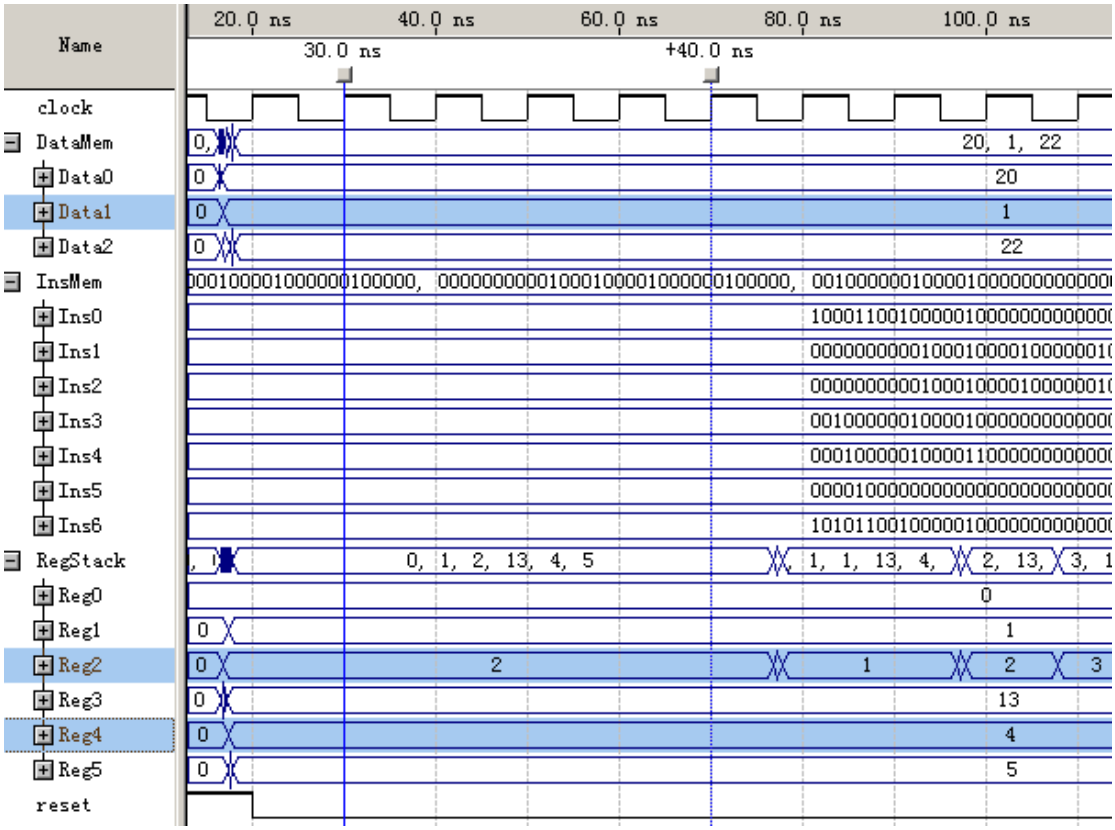
这段指令实现的功能是：从存储器读取数据存到 R2，增加(R2)的值，直到(R2)的值和(R3)相同，这时将 R2 的值写入存储器，最后计算(R2)和(R3)的积，结果存入 R4(及 R5)。

这段指令没有什么实际意义，但其中包含了各种类型的指令，以及各种类型的冒险。

其中 0 号指令为 load 指令，和其后的 1 号指令形成 load-use 冒险；1、2、3 号指令有两种类型的转发；3 号指令读取 R2 时，恰逢 0 号指令写回 R2；3 号指令为 I-Type(立即数型)指令。4 号为分支指令，构成分支冒险；5 号为跳转指令；6 号为 store word 指令；28 号为乘法指令。

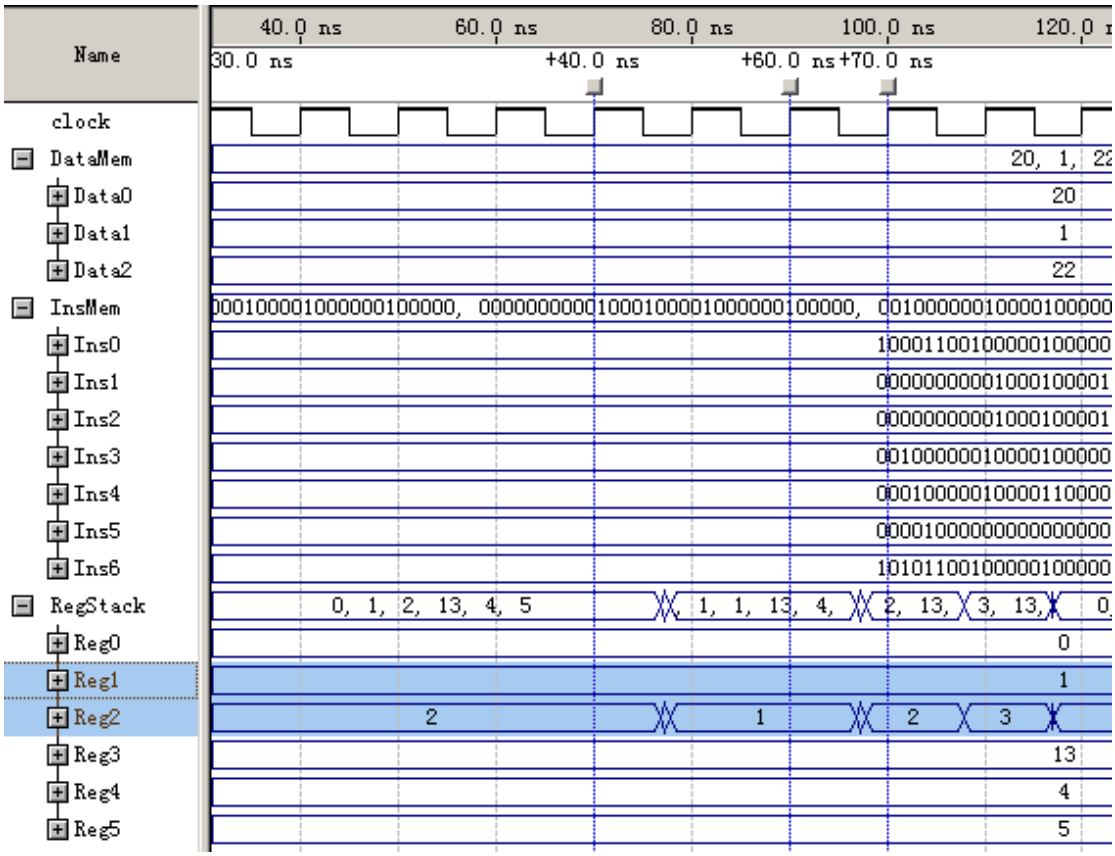
仿真结果：（时钟周期 10ns）

图 8.1



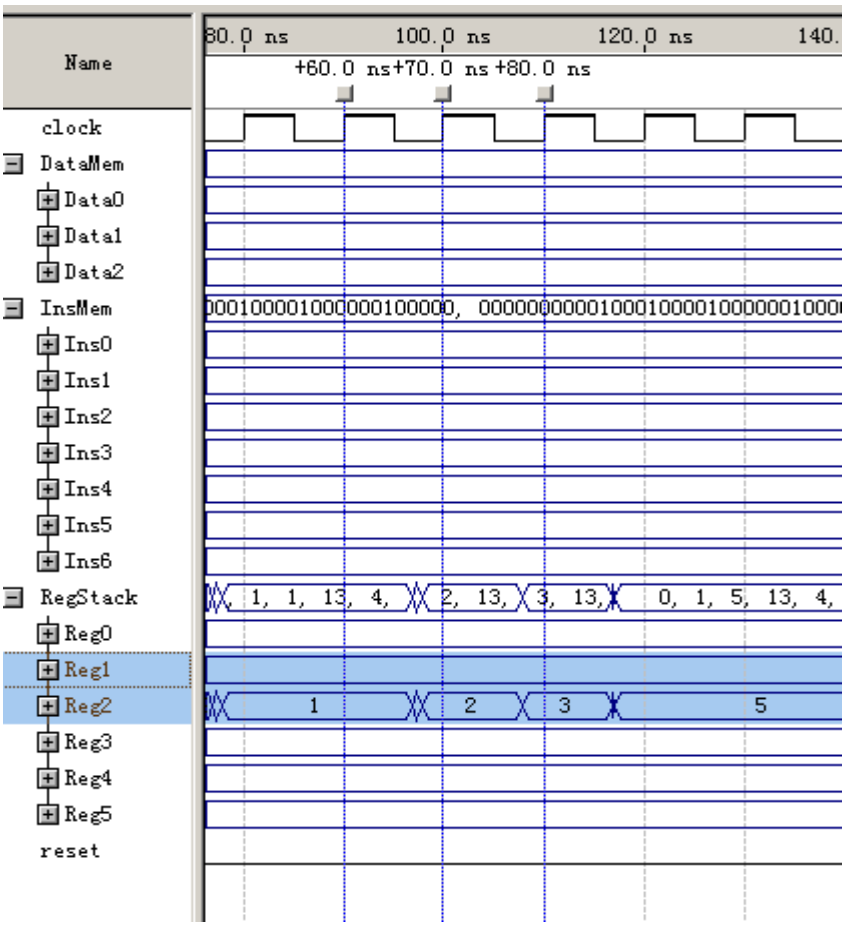
如图 8.1 所示，复位信号 reset 在时间轴 20ns 末端失效，故 30ns 处（第一条竖线标记处）的时钟 clock 正沿 0 号指令（lw R2,0(R4)）开始取指，40ns 处的时钟正沿该指令开始译码，50ns 处的正沿开始执行，60ns 处的正沿访存，70ns 处（第二条竖线标记处）的正沿开始写回，在 80ns 处的正沿到来之前，从内存（此处只是 cache）中取出的数据（即 Data1，值为 1）被正确写入到 R2（R2 的值在 80ns 之前由 2 变成从内存中读出的 1）。

图 8.2



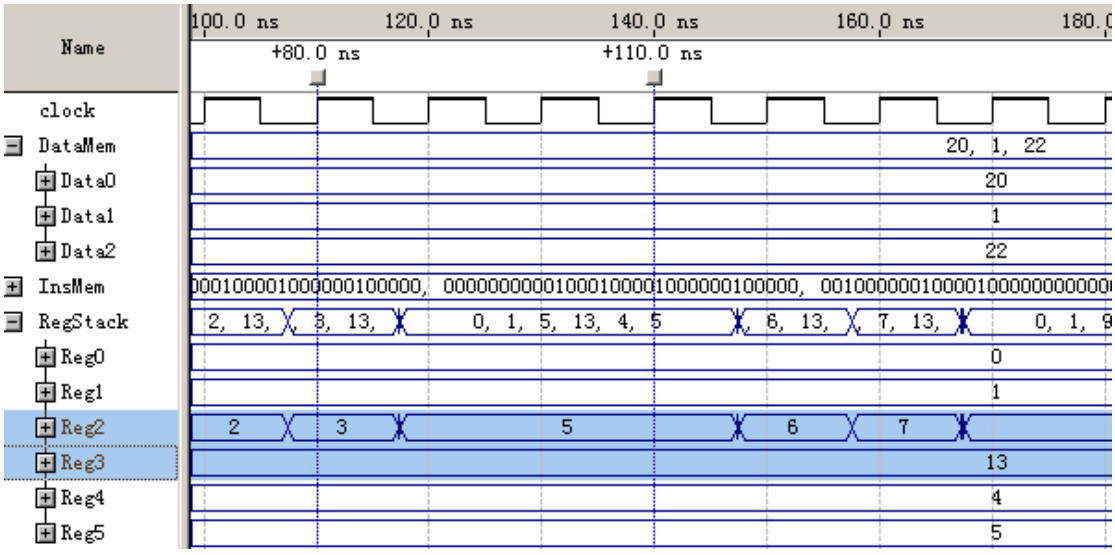
如图 8.2，1 号指令（add R2,R1,R2;）的执行结果（为 2）在 90ns 处的正沿开始写入，在下一个正沿之前（100ns 处）写入完成。其开始写入的时间比第一条指令晚了 20ns（即 2 个周期）。顺序执行情况下，本该只晚一个周期，此处多了一个周期，这是因为该指令和其前第一条指令（lw R2,0(R4);）之间存在 load-use 冒险，于是该指令在译码阶段被冒险检测单元阻塞一个周期。

图 8.3



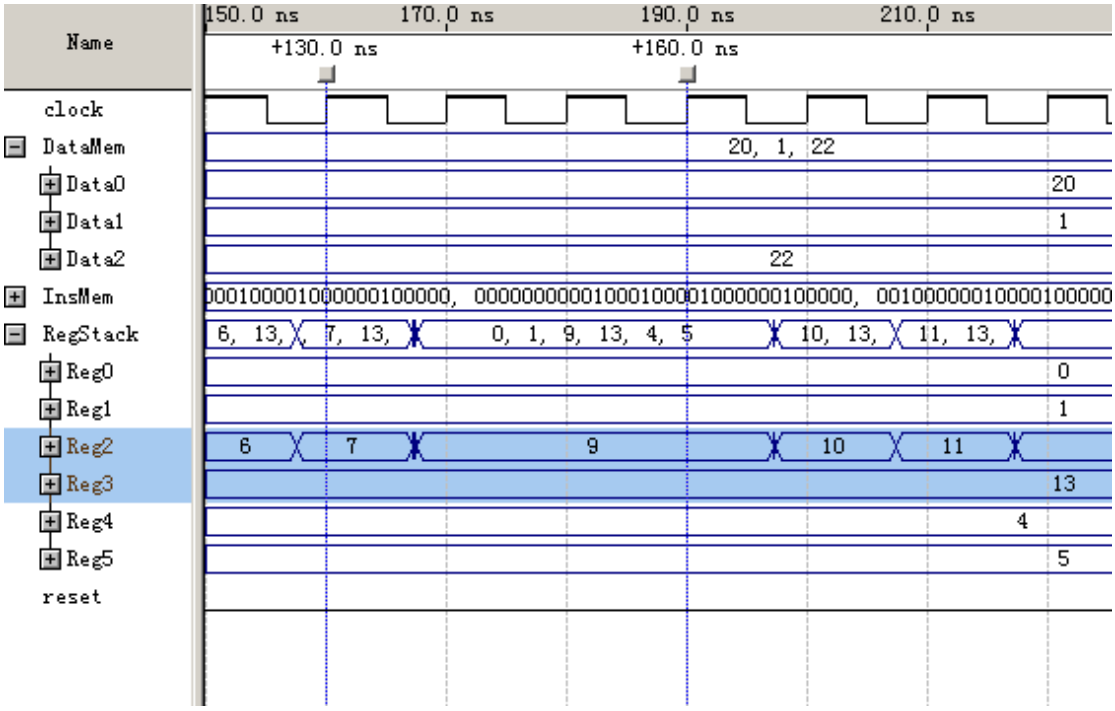
如图 8.3，2 号指令（add R2,R1,R2）的结果（值为 3）在随后的 100ns 处的时钟正沿开始写入（在 110ns 处的正沿到来前完成写入），比 1 号指令的开始回写的时间晚 10ns，即一个时钟周期。3 号指令（addi R2,R2,2）的结果（值为 5）在 110ns 处的时钟正沿开始写入（在 120ns 处的正沿之前完成写入），比 2 号指令的开始回写时间晚一个周期。

图 8.4



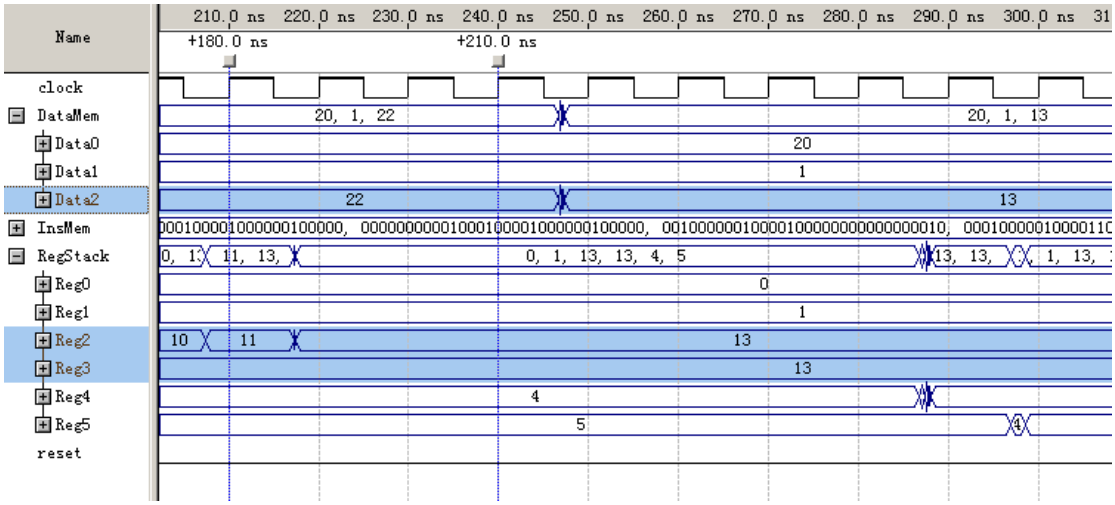
4 号指令（beq R2,R3,1）是分支指令，分支指令在其第 4 个周期（访存阶段）开始时检测分支条件是否满足，此次不满足，故其后指令有效。5 号指令为跳转指令（j 1），跳转到 1 号指令处（add R2,R1,R2）。该指令的结果在 140ns 处的正沿开始回写，这比前一次循环时 3 号指令开始回写的时间晚了 3 个周期。这 3 个周期中，由于该指令在上次循环中的 3 号之后执行，故自然落后 1 个周期，另外分支和跳转分别占用 1 个周期，故总共落后 3 个周期。

图 8.5



如图 8.5，第 2 次循环结束时，R2 的值变成 9.

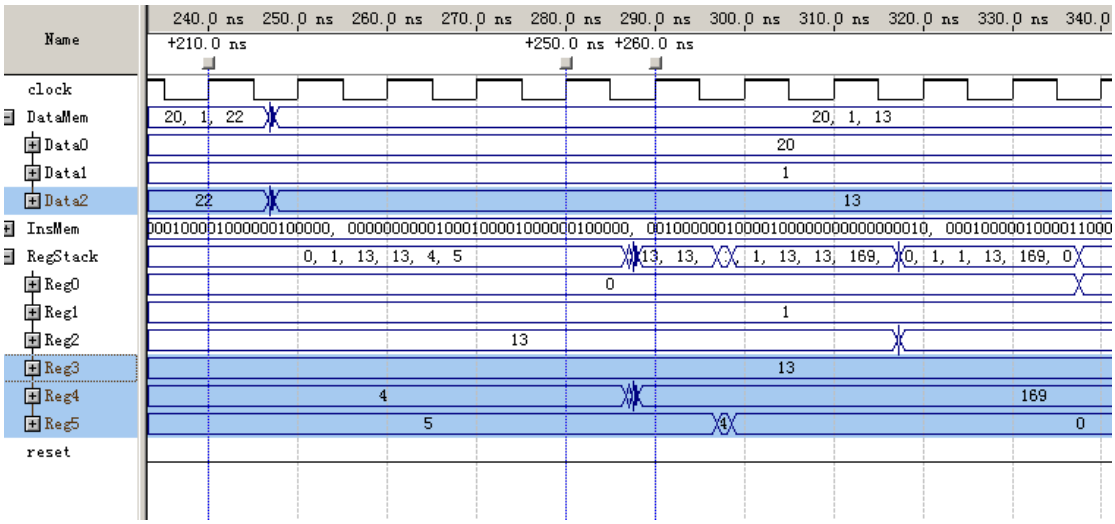
图 8.6



如图 8.6，第 3 次循环结束时，R2 是值变为 13，和 R3 的值相同，故分支成功，执行 6 号指令（sw R2,4(R4)）。6 号指令在 240ns 处的正沿开始访存，并在 250ns 处的正沿到来之前访存完毕。其开始访存的时间比上次循环中的 3 号指令的开始回写时间晚了 3 个周期。这是因为分支占去 1 个周期，分支指令又清除了其后的两条指令，即损失两个周期，而 6 号本身的顺序落后 1 个周期，即 6 号指

令总共比前一次循环的 3 号指令落后 4 个周期，但 6 号访存指令不回写寄存器组，而是将数据写入到内存，这一步操作是在其访存阶段做的，而不是回写阶段，故该动作的开始时间比 3 号指令的回写动作的开始时间晚 3 个周期。图 8.6 中标出的两条竖线印证了这一点。

图 8.7



6 号指令之后是 7 号指令（mult R4,R2,R3），执行结果是 169，回写到 R4。7 号指令的回写动作比 6 号的回写动作自然落后 1 个周期，故其回写动作比 6 号指令的访存动作自然落后 2 个周期。但是 7 号指令的执行阶段（第 3 个指令周期开始）为 3 个周期，不是正常情况下的 1 个周期，故此处多出 2 个周期，加上自然落后的 2 个周期，故 7 号指令的回写动作比 6 号指令的访存动作落后共 4 个周期。如图 8.7 中两条竖线所示，7 号指令回写动作的开始时间是 280ns（图中第 2 条竖线）处的正沿，比其前的 6 号指令访存动作的开始时间（图中第 1 条竖线）晚了 4 个周期，与前面的分析相符。7 号指令回写动作占 2 个周期，第 1 个周期回写结果的低 32 位，第 2 个周期回写结果的高 32 位。如图，290ns 处，高 32 位开始回写（R5），300ns 处的正沿之前，高 32 位回写完毕。

综上，仿真测试的结果与预期相符，处理器的各项功能正确无误。

仿真结果请查看 MIP32 文件夹下的 MIPS32.sim.cvwf。为了便于查看该文件，我们还另建了文件夹“仿真结果”，其中存放了仿真报告 MIPS32.sim.cvwf 的副本。

导入工程请双击 MIP32/MIP32.qdf 文件。

修改调试指令，请打开 MIPS32/mem/InsMem.v 文件，在有注释处修改指令，然后编译工程，并执行时序仿真，在随后弹出的时序仿真报告中查看波形结果。