除法指令的实现

一、除法指令的说明

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	rt	00000	00000	DIV 011010	div指令
SPECIAL 000000	rs	rt	00000	00000	DIVU 011011	divu指令

- 当功能码是 6'b011010 时，表示是 div 指令，有符号除法运算。

指令用法为：div rs, rt。

指令作用为：{HI, LO} <- rs / rt，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值，作为有符号数进行除法运算，将商保存到寄存器 LO，余数保存到寄存器 HI。

- 当功能码是 6'b011011 时，表示是 divu 指令，无符号除法运算。

指令用法为：divu rs, rt。

指令作用为：{HI, LO} <- rs / rt，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值，作为无符号数进行除法运算，将商保存到寄存器 LO，余数保存到寄存器 HI。

二、实现思路

2.1 试商法

试商法就是我们小学学习的简单除法，只不过这里全都换成了二进制，反而还简单了。在 OpenMIPS 中，对于 32 位除法，至少需要 32 个时钟周期才能得到结果，具体步骤见下：

设被除数是 m，除数是 n，商保存在 s 中，被除数的位数是 k，其计算步骤如下（为了便于说明，在此处将所有数据的最低位称为第 1 位，而不称为第 0 位）。

1、取出被除数的最高位 m[k]，使用被除数的最高位减去除数 n，如果结果大于等于 0，则商的 s[k]为 1，反之为 0；

2、如果上一步得出的结果是 0，表示当前的被减数小于除数，则取出被除数剩下的值的最高位 m[k-1]，与当前被减数组合做为下一轮的被减数；如果上一步得出的结果是 1，表示当前的被减数大于除数，则利用上一步中减法的结果与被除数剩下的值的最高位 m[k-1]组合做为下一轮的被减数。然后，设置 k 等于 k-1；

3、新的被减数减去除数，如果结果大于等于 0，则商的 s[k]为 1，否则 s[k]为 0，后面的步骤重复 2-3，直到 k 等于 1。

举例见表：

表 7-4 使用试商法计算 1101/0010（都是二进制）

步骤		minuend	minuend-n	k	s	说明
置初值				4	0000	置被除数 m 为 1101，除数 n 为 0010，k 为 4，同时 s 清零
第 0 步	开始时	1	小于 0	4	0000	(1-0010) < 0 , s[4]=0
	结束时	11		3		新的 minuend=（minuend，m[k-1]）
第 1 步	开始时	11	大于 0	3	0100	(11-0010) > 0, s[3]=1
	结束时	10		2		新的 minuend=（11-0010，m[k-1]）=10
第 2 步	开始时	10	等于 0	2	0110	(10-0010) = 0, s[2]=1
	结束时	01		1		新的 minuend=（10-0010，m[k-1]）=01
第 3 步	开始时	01	小于 0	1	0110	(01-0010) < 0, s[1]=0
结束					0110	最终得到商为 0110

或者我们可以用一个流程图表示：

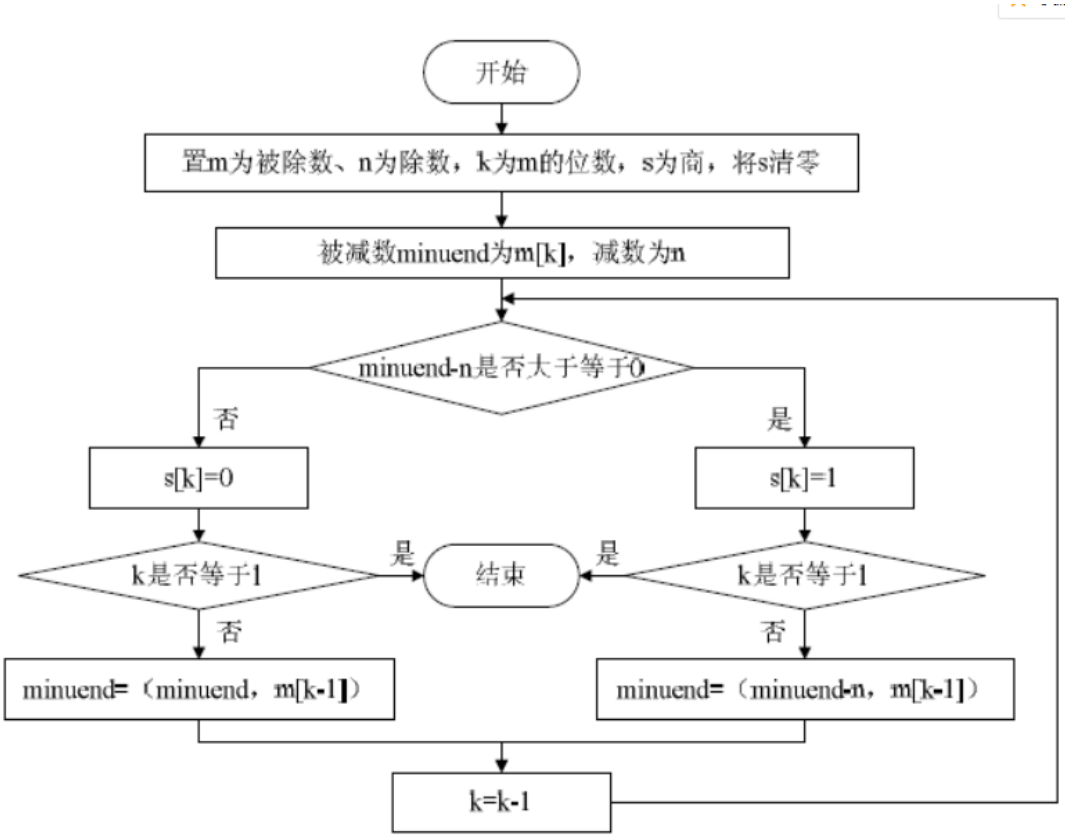


图 7-16 试商法的运算过程

我们考虑新增加一个 DIV 模块，用来单独执行除法指令，在此之前要先执行流水线暂停。

2.2 系统结构修改

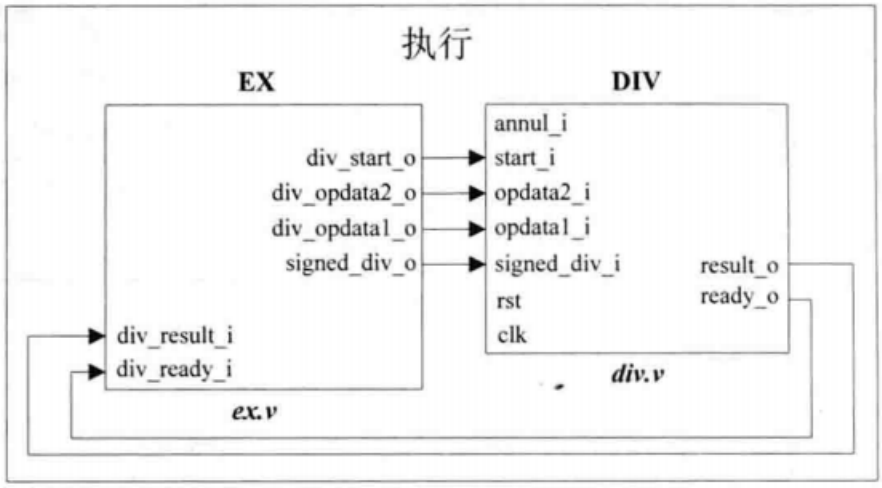


图 7-17 为了实现除法指令而对系统结果做的修改

当程序执行到 EX 模块时，执行流水暂停，将寄存器中的被除数和除数的值传入 DIV 模块的 opdata1_i 和 opdata2_i，用 signed_div_i 判断是否是有符号除法，除法指令通过 start_i 进行控制。除法指令执行完后，通过 ready_o 接口将状态传递到 EX 模块，并通过 result_o 接口将最终结果传递回 EX 模块。其中 result_o 高 32 位是余数，低 32 位是商，这些最终都将传递到 HILO 寄存器。

三、具体实现

3.1 增加 DIV 模块

接口描述如下表：

表 7-5 DIV 模块的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	signed_div_i	1	输入	是否有符号除法，为 1 表示有符号除法
4	opdata1_i	32	输入	被除数
5	opdata2_i	32	输入	除数
6	start_i	1	输入	是否开始除法运算
7	annul_i	1	输入	是否取消除法运算，为 1 表示取消除法运算
8	result_o	64	输出	除法运算结果
9	ready_o	1	输出	除法运算是否结束

DIV 模块的主要部分是一个状态机，共有四个状态，状态转换图如下：

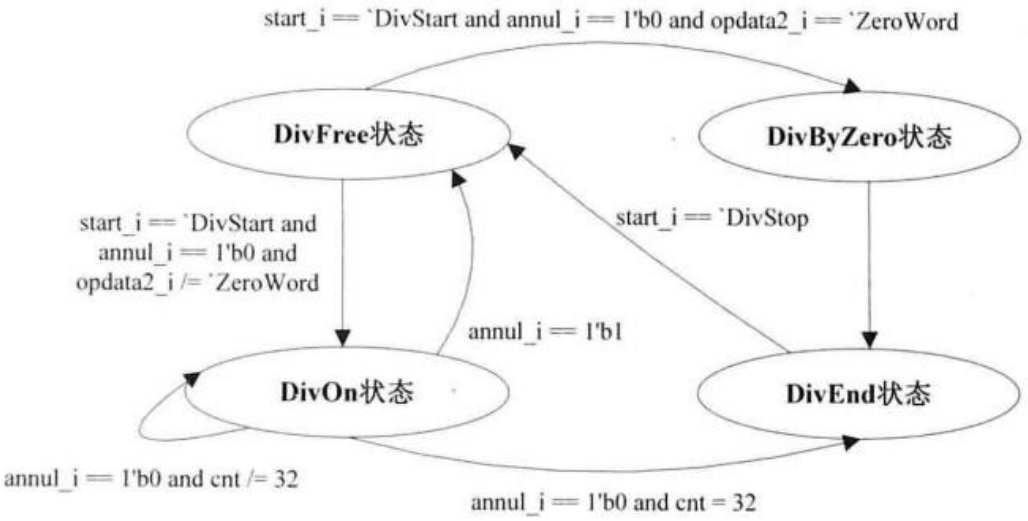


图 7-18　DIV 模块内部的状态转换图

复位的时候，DIV 模块处于 DivFree 状态，当开始信号 start_i 为 DivStart 时，并且输入信号 annul_i 为 0 时，除法操作开始。

当除数为 0 时，状态机进入 DivByZero 状态，直接将除法结果的商和余数记为 0，然后进入 DivEnd 状态，将 ready_o 设置为 DivResultReady，EX 模块会设置 Div 模块的 start_i 为 DivStop，除法结束。

当除数不为 0，进入 DivOn 状态，使用试商法，经过 32 个时钟周期得出结果，然后进入 DivEnd 状态，之后操作同上。

这个状态机有四个状态，而且 DIV 模块有开始和停止信号 start_i，这也是一个状态。但是这个状态和状态机的状态不一样。

关于这两种状态，后面在仿真时会具体说明。

3.2 修改 ID 模块

ID 模块增加了两条除法指令，修改相应的端口，没什么难点。

3.3 修改 EX 模块

增加的接口如下表：

表 7-6　EX 模块新增的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	signed_div_o	1	输出	是否是有符号除法，为 1 表示是有符号除法
2	div_opdata1_o	32	输出	被除数
3	div_opdata2_o	32	输出	除数
4	div_start_o	1	输出	是否开始除法运算
5	div_result_i	64	输入	除法运算结果
6	div_ready_i	1	输入	除法运算是否结束

该段代码分为三部分理解：

（1） 当指令传过来时，并且 DIV 模块没有进行到第 32 个周期，也就是 div_ready_i 为 DivResultNotReady，那么传递被除数和除数，div_start_o 信号设置为 Divstart，并且设置 stall 暂停信号，转向 DIV 模块进行运算。当 DIV 模块已经进行到第 32 周期时，即 DIV 模块中 state 为 DivEnd 状态，设置 div_ready_i 为 DivResultReady，除法运算停止，撤掉暂停命令 stallreq_for_div；

（2） 暂停指令包括乘累加、乘累减和现在的除法，因此他们之间用逻辑或连接；

（3） 最终 div_result_i 就是除法运算的结果，将高 32 位的余数存入 HI 寄存器，低 32 位的商存入 LO 寄存器。

3.4 修改 OpenMIPS 模块

增加接口的例化，注意哪些值是相等的，很简单，细心即可。

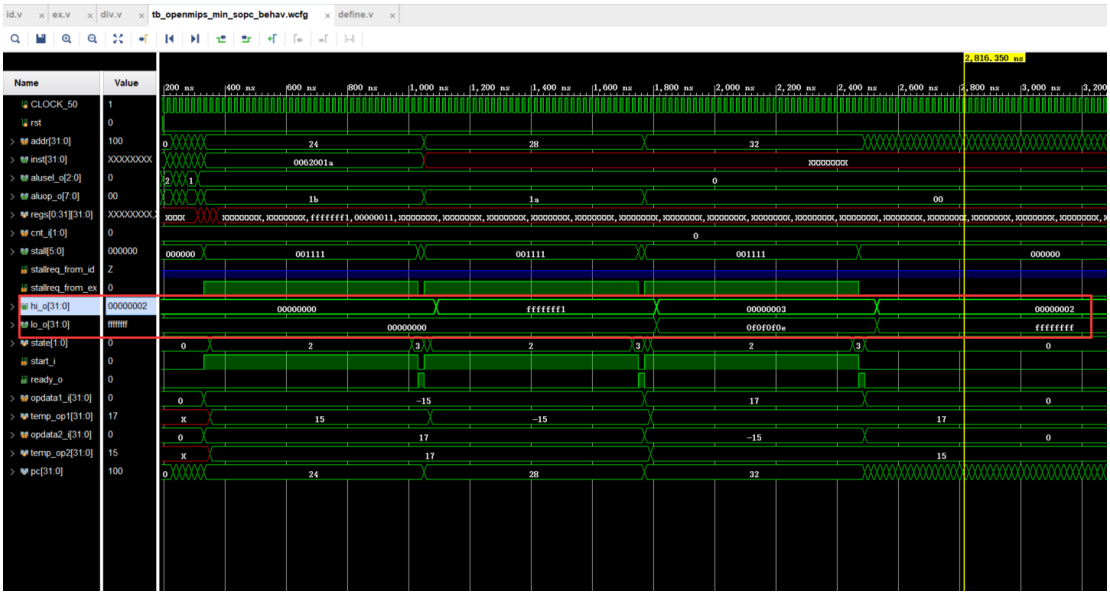
有一点需要注意，DIV 模块的输入接口 annul_i 目前为常数 0，表示不会有取消除法指令的情况发生，但是在后面实现异常处理的时候，会重新确定 DIV 模块的输入接口 annul_i 的值。

四、测试及分析

我使用了如下的汇编测试代码：

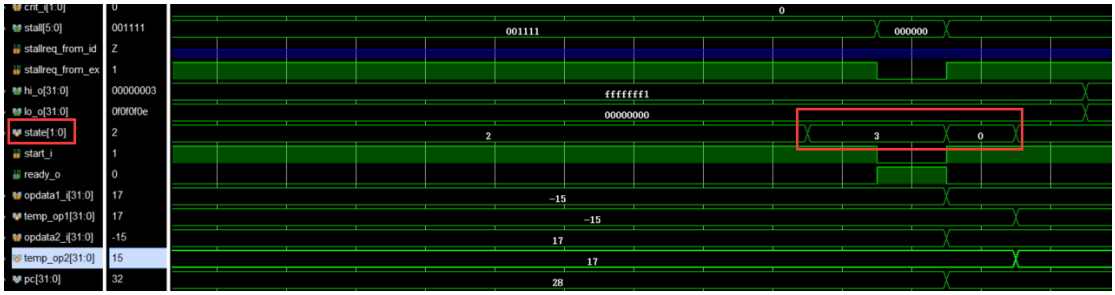
```
    .org 0x0
    .global _start
_start:
    ori   $2,$0,0xffff
    sll   $2,$2,16
    ori   $2,$2,0xfff1          # $2 = -15
    ori   $3,$0,0x11            # $3 = 17

    div $zero,$2,$3             # hi = 0xfffffff1
                                # lo = 0x0
    divu $zero,$2,$3            # hi = 0x00000003
                                # lo = 0x0f0f0f0e

    div   $zero,$3,$2           # hi = 2
                                # lo = 0xffffffff
```

最终得到的仿真结果为：



结果正确。

我们看一下 state 的值：



state 为 DivEnd 状态是两个周期，而后面的 DivFree 状态只有一个周期，为什么呢？这就是和之前说的状态机的状态和 Div 模块的状态的区别了。

当 32 个周期执行完后，状态机状态从 DivOn 转换为 DivEnd，但是 Div 模块还处于 DivStart 状态，因此还需要一个周期，通过 EX 模块将 Div 模块的状态变为 DivStop 才能彻底终止 Div 除法模块的运算，因此 state 会在 DivEnd 状态停留两个周期。

另外，我在写代码的时候，把红框框里的 temo_op2 的值赋值成了 opdata1_i，但是只有最后一个结果的 LO 寄存器值不正确，找了很长时间才发现问题所在。

```
`DivFree: begin
  if (start_i == `DivStart && annul_i == 1'b0) begin
    if (opdata2_i == `ZeroWord) begin
      state <= `DivByZero;
    end else begin
      state <= `DivOn;
      cnt <= 6'b000000;
      if (signed_div_i == 1'b1 && opdata1_i[31] == 1'b1) begin
        temp_op1 = ~opdata1_i + 1;
      end else begin
        temp_op1 = opdata1_i;
      end
      if (signed_div_i == 1'b1 && opdata2_i[31] == 1'b1) begin
        temp_op2 = ~opdata2_i + 1;
      end else begin
        temp_op2 = opdata2_i;
      end
      dividend <= {`ZeroWord, `ZeroWord};
      dividend[32:1] <= temp_op1;
      divisor <= temp_op2;
    end
  end else begin
    ready_o <= `DivResultNotReady;
    result_o <= {`ZeroWord, `ZeroWord};
  end
```

实际上就是测试数据比较特殊，正号负号的判断问题，有些数值没有进入我写错的那个判断语句。

在增加完算数操作后的数据流图如下：

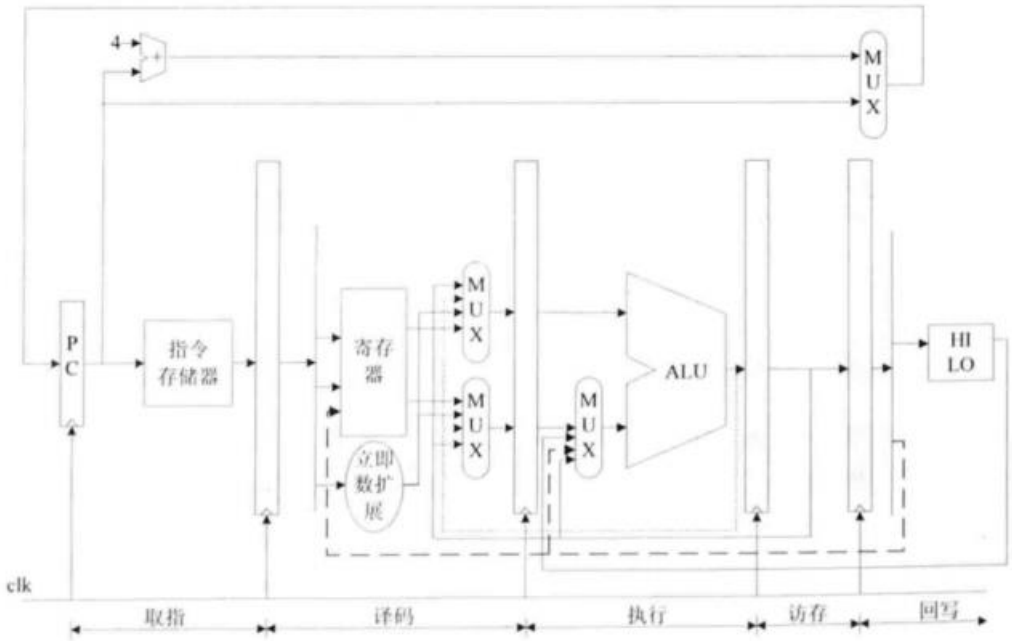


图 7-21 增加算术操作后的数据流图

五、一些思考

在 Div 模块中，我们定义了一个 65 位的 dividend 和 33 位的 div_temp，是因为一开始赋值的时候将被除数赋值给了 dividend 的[32:1]，因此第 0 位没有用到，所以最终结果 $result_o <= \{dividend[64:33], dividend[31:0]\}$也是中间空出了一位（32bit），这一位就是一开始的 0bit，因此我有一个想法，若始终定义为 64 位和 32 位，而不是 65 位和 33 位，是否也可以实现呢？书上这样设计是否只是为了看上去结构更清晰还是有别的考虑？