版本历史

文档更新记录	文档名:	LEC05_硬件乘除法器实现
	版本号	V0.1
	创建人:	计算机体系结构研讨课教学组
	创建日期 :	2017-10-19

更新历史

序号	更新日期	更新人	版本号	更新内容
1	2017/10/19	邢金璋	V0.1	初版。

手册信息反馈: xingjinzhang@loongson.cn

1 硬件乘除法器实现

通过本章节的学习，你将获得：

(1) 32 位支持有符号、无符号乘法的乘法器编码实现指导。

(2) 32 位支持有符号、无符号除法的除法器编码实现指导。

1.1 乘法器的实现

1.1.1 简单补码乘法器

移位乘法器和平常列竖式算乘法的方式相同，将 32 位乘法转化为 32 个 64 位数的加法，再计算出结果。值得注意的是两个数的补码的积并不等于积的补码。如果[Y]补=$y_{31}y_{30}\ldots y_1y_0$，则：

$$[X\times Y]_{补}=[X]_{补}\times(-y_{31}\times 2^{31}+y_{30}\times 2^{30}+\ldots+y_1\times 2^1+y_0\times 2^0)$$

也就是说，$[X\times Y]_{补}$并不等于$[X]_{补}\times[Y]_{补}$，而是等于把[Y]补的最高位取反，其它位不变的数和[X]补相乘的结果。具体推到请参考课本《计算机体系结构基础》。

下面以一个 4 位的乘法器举例：0101(5) ×1001(-7) =1011101(-35)，简单补码乘法计算如下(每个部分积均要做符号扩展)：

				0	1	0	1
			×	1	0	0	1
+	0	0	0	0	1	0	1
+	0	0	0	0	0	0	
+	0	0	0	0	0		
-	0	1	0	1			
	1	0	1	1	1	0	1

在补码乘法运算过程中，只要对 Y 的最高位乘积项做减法，对其他位乘积项做加法即可。但在加法和减法操作时，一定要将符号位扩充，使乘积项对齐。

可以使用 1 位 Booth 算法进行补码乘法，将各部分积统一成相同的形式。

1.1.2 2 位 Booth 编码

对 32 位乘法而言，无论是上述简单的补码乘法器还是 1 位 Booth 算法实现的乘法器，都需要将 32 个部分积相加才能得到结果，其延迟和硬件的开销都很大。引入 2 位 Booth 编码可以显著减少加法的次数。

对$(-y_{31}\times 2^{31}+y_{30}\times 2^{30}+\ldots+y_1\times 2^1+y_0\times 2^0)$做如下变换：

$$(-y_{31}\times 2^{31}+y_{30}\times 2^{30}+\ldots+y_1\times 2^1+y_0\times 2^0)$$

$$=(y_{29}+y_{30}-2\times y_{31})\times 2^{30}+(y_{27}+y_{28}-2\times y_{29})\times 2^{28}+\ldots+(y_1+y_2-2\times y_3)\times 2^2+(y_0-2\times y_1)\times 2^0$$

$$=(y_{29}+y_{30}-2\times y_{31})\times 2^{30}+(y_{27}+y_{28}-2\times y_{29})\times 2^{28}+\ldots+(y_1+y_2-2\times y_3)\times 2^2+(y_{-1}+y_0-2\times y_1)\times 2^0 \text{ (其中 } y_{-1}=0)$$

根据上式，可以先将 Y 的-1 位补 0，然后每次扫描 Y 的 3 位来确定乘积项，乘积项减少了一半，32 位的乘法只需 15 次加法。2 位 Booth 编码规则如下表所示：

Y_{i+1}	y_i	Y_{i-1}	操作
0	0	0	0
0	0	1	$+[X]_{补}$
0	1	0	$+[X]_{补}$

0	1	1	+2[X]补
1	0	0	-2[X]补
1	0	1	-[X]补
1	1	0	-[X]补
1	1	1	0

比如 0101(5)×1001(-7)，其中[X]补=+0101，2[X]补=+01010，-[X]补=-0101=+1011，-2[X]补=-01010=+10110：

				0	1	0	1		
			×	1	0	0	1	0	(补 y_{-1})
+	0	0	0	0	1	0	1		(010,+[X]补)
+	1	0	1	1	0				(100,-2[X]补)
	1	0	1	1	1	0	1		(结果为-35 补码)

可以看到，对 4 位乘法使用 2 位 Booth 编码之后，部分积由原来的 4 个减少为 2 个，需要加法的次数由原来的 3 次减少为 1 次，很好地提高了效率。

1.1.3 保留进位加法器

对于 32 位乘法，即使使用了 2 位 Booth 编码，16 个部分积仍然需要进行 15 次接近 64 位的加法操作。一般一次 65 位加法已经延迟很大了。如果 15 次加法，使用累加的形式，则效率会更低。对于多个数的相加，可以考虑使用不需要等待进位信号的保留进位加法器。

保留进位加法器就是使用全加器将三个加数的加法转化成两个加数的加法的装置，其中转化后的两个加数一个是所有的本地和组成的，一个是所有的向高位的进位信号组成的。运算过程如下：

	1	0	1	1	0	1	0	1
	1	1	1	0	1	1	1	0
+	1	0	1	1	1	1	0	0
1	1	1	1	0	0	1	1	1
+ 1	0	1	1	1	1	0	0	0

10110101(-75)+11101110(-18)+10111100(-68)=111100111(-25)+101111000(-136)

保留进位加法器的每一位的计算都是独立的，不会依赖其他位的信息，所以不会有进位延迟。例子中的最低位的 3 个加数为 1、0、0，结果为 1，进位为 0；最高位的 3 个加数为 1、1、1，结果为 1，进位为 1，其他位也是一样，最后还需将进位结果左移一位。

1.1.4 华莱士树

对于要将很多个加数相加得到一个结果的运算，可以先使用一层保留进位加法器(也就是一组全加器)将其转化为约 2/3 个加数相加，再使用一层转换为约 4/9 个加数相加，直到最后转化为 2 个加数相加。这样的构造叫做华莱士树。

使用 2 位 Booth 编码后，32 位乘法被转化为 16 个 64 位的部分积相加。可以使用六层华莱士树将加数减少为 2 个，下图是 16 个部分积中，针对某一 bit 搭建的华莱士树

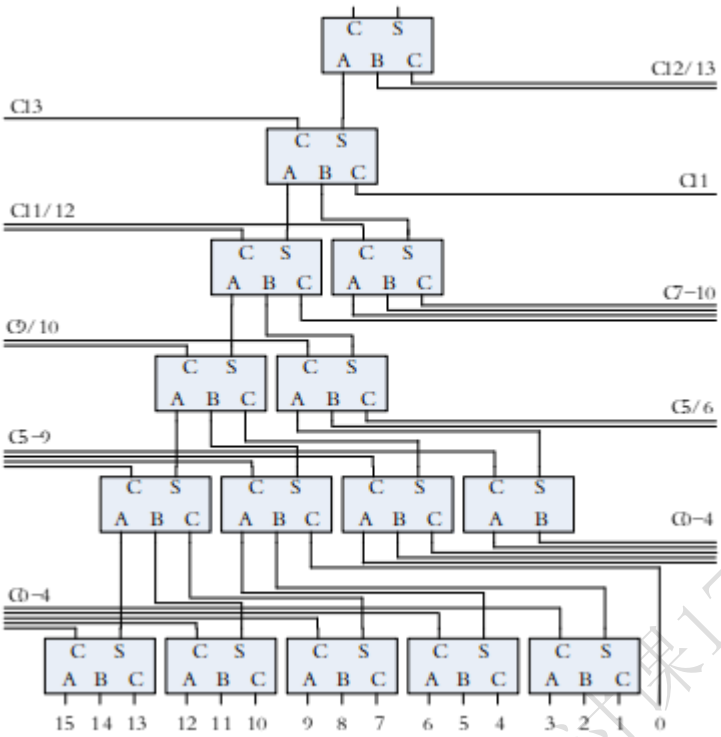


图 1　16 个数相加的 1 位华莱士树

1.1.5　使用 2 位 Booth 编码+华莱士树的 32 位补码乘法器

在 Booth 编码和华莱士树的基础上不难设计出 32 位定点补码乘法器。其结构如图 2 所示：

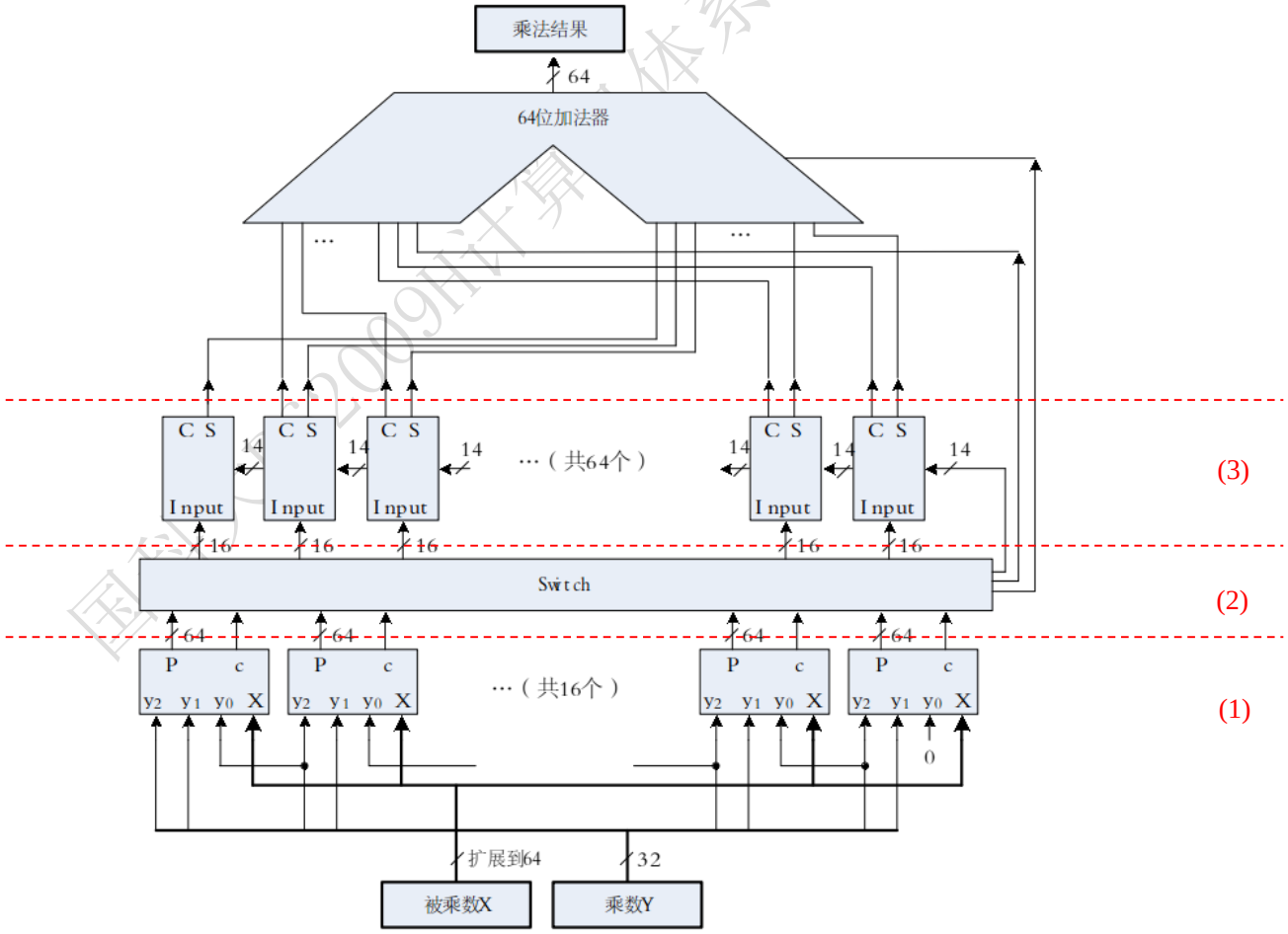


图 2　32 位补码乘法器结构

上图中第(1)部分为采用 2 位 booth 算法得到 16 个部分积，其中 P 为 64 位，为部分积的主体，c 为 1 位，为对被乘数取反的表示。

第(2)部分类似矩阵转置，将 16 个 64 位部分积转置为 64 个 16 位等数位的数，用作华莱士每 bit 处的输入，这一部分在电路中的体现就是连线，没有任何多余的电路单元。

第(3)部分为华莱士树，为 64 个 1bit 的华莱士的集合，需要注意的是每 1bit 的华莱士树有来自低 bit 的进位信号，而最高 bit 处的华莱士数向高位的进位则被忽略掉了。这是因为 32 位有符号数乘以 32 位有符号数，能得到的最大的数位：$-2^{31}\times-2^{31}=2^{62}$，用 64bit 有符号数已经能完全容下了，故就算将最高 bit 处的华莱士树想高位的进位带人运算，得到的也是符号位的扩展，故可以直接省略。

1.1.6 对乘法器的进一步的优化

(1) 用定点补码乘法器实现无符号乘法

上述几节都是介绍的补码乘法器，显然这是针对有符号乘法的，而在指令集里除了有符号乘法，还有无符号乘法。故实现的乘法器还需要支持无符号乘法。

所谓有符号乘法和无符号乘法，是指进行相乘的源操作数是被当做有符号数还是无符号数，比如 32 位寄存器里存放的数为 0x80000000，如果该数被看作有符号数则是-2^{31}(计算机里存储的有符号数均为补码形式)，被看作无符号数则是 2^{31}。

虽然看起来无符号数与有符号数差别很大，但是只要在无符号数最高位前再补一个 0，就可以把它看做是符号位为 0 的有符号数；同时对于有符号数，在起最高位前再扩展一位符号位，则显示表示的数值不变。通过这种最高位前再补一位的方法就可以将有符号数和无符号数统一，相应 32 位数就扩展成了 33 位数。

因而可以实现一个 33 位的乘法器，从而使得它既可以运算有符号乘法，又可以运算无符号乘法。每次运算前需要将源操作数扩展为 33 位，有符号数就在最高位前补符号位，无符号数就在最高位前补 0：比如 32 位数 0x8000_0000，当作无符号数时需扩展为 33 位 0x0_8000_0000，当作有符号数时需扩展为 0x1_8000_0000。需要注意的是 33 位数经过 2 位 Booth 编 码后会产生 17 个部分积，相应的华莱士树的结构也要做调整：

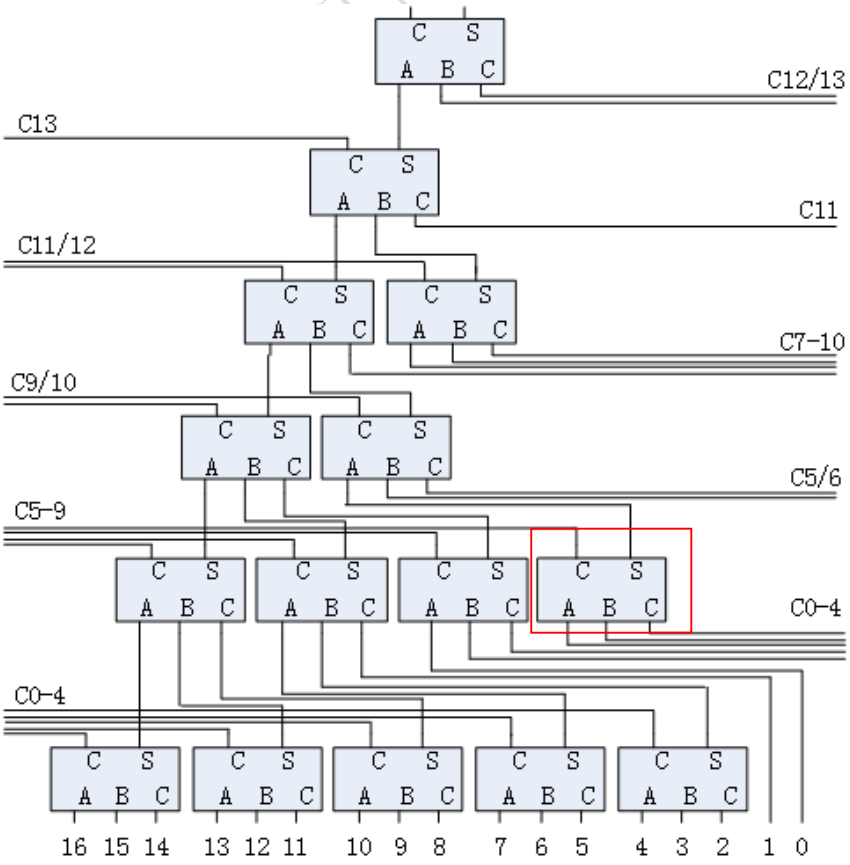


图 3 17 个数相加的 1 位华莱士树

(2) 切分流水线

按照上面的描述设计出的单周期 32 位补码乘法器，经过综合工具综合得到的最长延迟时间为约 36ns，延时过长，所以可以考虑切分流水线。

所谓流水线，就是在某个地方将单周期的乘法器一分为二，在两部分的数据通路上加一个寄存器，并将寄存器中的数据作为后一部分的输入信号。在乘法器中引入时钟，每当时钟上升沿到来之际，就将前一部分的数据写入寄存器。这样可以将乘法拆分成两个周期完成。由于切分流水线之后的两部分可以并行，所以平均下来还是大约每拍执行一条乘法，同时切分流水线后可以使乘法器以更高的频率运行。

切分流水线要尽量做到均衡，即被流水线分隔的几个部分的延迟时间差距不能太大。

本实验要求将乘法器切成两级流水结构，并且使得切流水之后的最长延迟时间接近单周期时的一半。所以应该先根据乘法器的结构来确定流水线的位置，切分完成后再使用综合工具进行评估。两级流水线的乘法器可以在 CPU 中占据执行级和访存级的位置，并不会拖延整个 CPU 的运行节奏。

本次实验虽然要求乘法器切分两级，但建议乘法器还是写为一个单独模块，这样如果想替换整个乘法器也很方便，对乘法器进行模块级验证也很方便。乘法器模块大致接口如下：

信号	位宽	方向	功能
mul_clk	1	input	乘法器模块时钟信号
resetn	1	input	复位信号，低电平有效
mul_signed	1	input	控制有符号乘法和无符号乘法
x	[31:0]	input	被乘数
y	[31:0]	input	乘数
result	[63:0]	output	乘法结果，高 32 写入 HI，低 32 位写入 LO

切分的流水级也包含在乘法器模块中，该模块的输入信号均来自执行级，输出信号在访存级被获取，带到写回级写入 HI/LO 寄存器。故乘法器模块内部只有一组流水级间的寄存器。

1.1.7 乘法器模块级验证

在乘法模块实现后，最好对其进行模块级随机验证，确保功能正确。

对于切分两级流水的乘法器模块，可以使用以下 testbench：

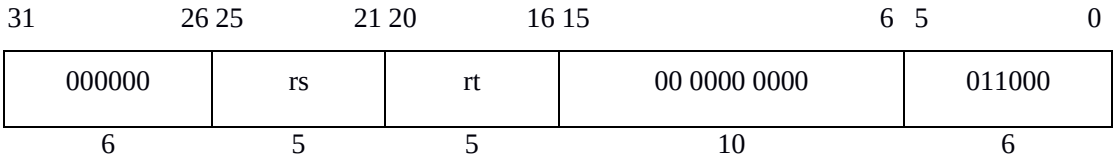
```
`timescale 1ns / 1ps

module mul_tb;

	// Inputs
	reg mul_clk;
	reg resetn;
	reg mul_signed;
	reg [31:0] x;
	reg [31:0] y;

	// Outputs
	wire signed [63:0] result;

	// Instantiate the Unit Under Test (UUT)
	mul uut (
	  .mul_clk(mul_clk),
	  .resetn(resetn),
	  .mul_signed(mul_signed),
	  .x(x),
	  .y(y),
	  .result(result)
	);

	initial begin
	  // Initialize Inputs
	  mul_clk = 0;
	  resetn = 0;
```

```
        mul_signed = 0;
        x = 0;
        y = 0;
        #100;
        resetn = 1;
      end
      always #5 mul_clk = ~mul_clk;

//产生随机乘数和有符号控制信号
always @(posedge mul_clk)
begin
    x          <= $random;
    y          <= $random; //$random 为系统任务，产生一个随机的 32 位有符号数
    mul_signed <= {$random}%2; //加了拼接符，{$random}产生一个非负数，除 2 取余得到 0 或 1
end

//寄存乘数和有符号乘控制信号，因为是两级流水，故存一拍
reg [31:0] x_r;
reg [31:0] y_r;
reg          mul_signed_r;

always @(posedge mul_clk)
begin
    if (!resetn)
    begin
        x_r          <= 32'd0;
        y_r          <= 32'd0;
        mul_signed_r <= 1'b0;
    end
    else
    begin
        x_r          <= x;
        y_r          <= y;
        mul_signed_r <= mul_signed;
    end
end

//参考结果
wire signed [63:0] result_ref;
wire signed [32:0] x_e;
wire signed [32:0] y_e;
assign x_e        = {mul_signed_r & x_r[31],x_r};
assign y_e        = {mul_signed_r & y_r[31],y_r};
assign result_ref = x_e * y_e;
assign ok         = (result_ref == result);

//打印运算结果
initial begin
    $monitor("x     =     %d,     y     =     %d,     signed     =     %d,     result
= %d,OK=%b",x_e,y_e,mul_signed_r,result,ok);
end

//判断结果是否正确
always @(posedge mul_clk)
begin
      if (!ok)
    begin
          $display("Error: x  =  %d,  y  =  %d,result  =  %d,  result_ref  =  %d,
OK=%b",x_e,y_e,result,result_ref,ok);
          $finish;
      end
end

endmodule
```

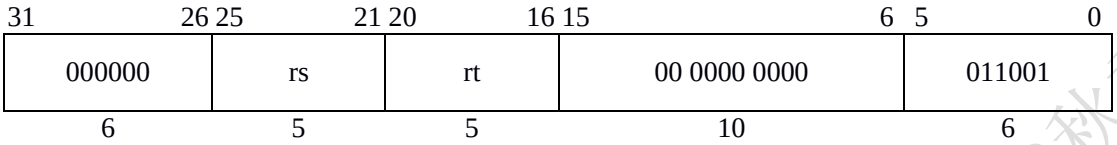
注意声明和计算有关的变量时必需写上 signed 来表示有符号数，在 testbench 中如果显示 OK=1，则说明结果正确，否则说明运算有错，同时会仿真会停止。

1.1.8 将乘法器集成到自己的 CPU 中

有乘法器之后就可以实现 MULT（有符号乘法）和 MULTU（无符号乘法）指令了，MULT 指令编码如下：



MULTU 指令编码如下：



在自己的五级流水 CPU 的译码级增加 MULT 和 MULTU 指令的项，经乘法器执行两拍之后，在写回级将结果的高 32 位写入 HI 寄存器，将结果的低 32 位写入 LO 寄存器。

MULT 和 MULTU 指令均不会产生例外。

1.2 除法器的实现

除法器依据依据是否将源操作数转换为绝对值分为：绝对值除法器和补码除法器。通常，绝对值除法器最后得到的是商和余数的绝对值，故最后需要计算商和余数的补码；而补码除法器，虽然得到的结果是补码，却计算过程可能存在多减除数的情况，所以也需要对余数进行调整。

1.2.1 迭代除法器

除法器通常都是需要迭代进行的，迭代除法也为试商法，32 位除法的商最多也为 32 位，故可以依次从商的第 31 位到第 0 位，试 0 或 1，这和我们笔算除法的方法类似。

依据迭代过程中，在不够减时（商为 0），是否恢复余数分为：恢复余数法（循环减法），不恢复余数法（加减交替）。加减交替就是在不够减时，其实是减多了，但并不恢复余数，下一次迭代改为加法，以补回多减的，之所以可以这样做，是因为，假设第二次加法为 r_2+x，其中 r_2 为前一次减法得到的结果，就是 r_1-2x，所以也就相当于是 r_1-x，与恢复余数是一样的。具体为什么前一次减法减去的是 2x，是后一次的两倍，可以假设被除数是四位 abcd，除数是二位 yz，那么从高位向低位迭代，开始是 ab-yz，相当于是 ab00-yz00，第二次是 bc-yz，相当于是 bc0-yz0，也就是第一次是 2x，第二次是 x。这也是迭代除法实现时使用移位器的道理。

其实所有迭代除法器，所谓绝对值、补码除法器，或者循环减法、加减交替等分类，都是基于一个原理：判断剩余被除数和除数的大小，确定商，更新剩余被除数。只是在具体实施中，依据处理方法不同，得到上述的除法器分类。

比如，补码除法器，对有符号除法，假设被除数为 0xffff_ffff，除数为 0x1111_1111。也就是被除数是负数，除数时正数，显然比较被除数和除数的大小，是需要使用加法操作的。这时候，如果采用恢复余数法，那就是循环加法补码除法器；如果不恢复余数法，迭代过程中，如果将余数加成了正数，那就说明加多了（绝对值减多了），就需要改为减法，那就是加减交替补码除法器。

以 1 位恢复余数绝对值迭代除法器为例，运算过程分为三步：

(1) 根据被除数和除数确定商和余数的符号，并计算被除数和除数的绝对值：

我们预先规定余数的符号要和被除数的符号保持一致，这样商和余数的符号就可以由下表确定：

被除数	除数	商	余数
正	正	正	正

正	负	负	正
负	正	负	负
负	负	正	负

对于无符号数，计算机里存的数就是其原码，故绝对值即为该数。对于有符号数：如果为正数，计算机里存的数为其补码也是其原码，故绝对值就是该数；对于负数，则计算机里存的为其补码，直接对该补码(含符号位)取反加 1 即得到其绝对值。

(2) 迭代运算得到商和余数的绝对值：

使用绝对值迭代运算的过程如下：首先将 32 位的被除数前面补 32 个 0，记为 A[63:0]，记除数为 B[31:0]，得到的商记为 S[31:0]，余数即为 R[31:0]，第一次迭代，取 A 的高 33 位，即 A[63:31]，与 B 高位补 0 的结果{1'b0,B[31:0]}做减法，如果结果为负数，则商的相应位(S[31])为 0，如果结果为正数，则将被除数的相应位(A[63:31])更新为减法的结果，商的相应位记为 1。随后进行第二次迭代，此时就需要取 A[62:30]与{1'b0,B[31:0]}作减法，依据结果更新的到 S[30]上，并更新 A[62:30]。依此类推，直到算完第 0 位。

(3) 调整最终的商和余数：

(2)中得到的商和余数为对应的绝对值，依据(1)中确定的符号数将商和余数转化为有符号数，提交结果。

下面以 8 位补码除法 10010101(-107)÷00011101(29)为例介绍迭代除法：

首先，因为被除数为负，除数为正，所以确定商为负，余数为负，并且计算得到被除数的绝对值为 01101011，除数的绝对值为 00011101。之后进入迭代减法的操作。

第一次迭代，得到商的最高位：

								0								
	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	（被除数扩展 8 个 0）
-	0	0	0	0	1	1	1	0	1								（减除数）
	1	1	1	1	0	0	0	1	1								（不够减，商上 0）
	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	（调整剩余被乘数）

上图中第 1 行为商的绝对值，第 2 行为第一拍执行之前被除数的绝对值，第 3 行为除数的绝对值，第 4 行为减法的结果，最后一行是第一拍执行之后被除数的绝对值。

在前六迭代中被除数的绝对都没有发生改变，所以略去。直接看第七次迭代：

								0	0	0	0	0	0	1		
	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	
-							0	0	0	0	1	1	1	0	1		（减除数）
							0	0	0	0	1	1	0	0	0		（够减，商上 1）
	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	（调整剩余被乘数）

第八次迭代，得到最终商和余数的绝对值：

								0	0	0	0	0	0	1	1	
	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	
-								0	0	0	0	1	1	1	0	1	（减除数）
								0	0	0	0	1	0	1	0	0	（够减，商上 1）
	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	（调整剩余被乘数）

于是得到商的绝对值为 00000011 和余数的绝对值为 00010100(取低 8 位)。

最后，将商和余数转为补码。因为前面已经确定了商为负数，余数也为负数，所以得到商 11111101(-3)和余数 11101100(-20)。

从上述计算过程可以看到，除法结果商的寄存器是从高位到低位依次得到，因为等效一个 32 位左移寄存器，每次迭代得到的商放在第 0 位上；被除数从高到低依次取 33 位域除数进行想减，并更新该 33 位，因而等效于一个 64 位左移寄存器，每次取高 33 位与除数进行想减判断并更新该 33 位。

可以依据以上所述画出迭代除法的结果示意图。

1.2.2 用迭代除法器进行无符号除法

如果仅仅让迭代除法器执行第二步的操作，它就是一个无符号除法器了。所以只需在迭代除法器中加入一个控制信号，使除法在第一、三步中不做数值调整，这样得到的结果就是无符号除法的结果。这样除法器就可以进行无符号运算。

1.2.3 迭代除法器中的控制信号

迭代除法器的第一步为获取乘数和控制信号，第三步为输出结果到下一流水级，各需要一拍。第二步为迭代运算，需要 32 拍。故一次除法需要 34 拍，可以在除法器中内置一个计数器，该计数器在除法开始时从 0 开始计数，当计到 33 时，将除法完成信号置为 1 并停止计数，直到下一个除法开始。

同乘法器一样，建议除法器也单独封装为一个模块，如下：

信号	位宽	方向	功能
div_clk	1	input	除法器模块时钟信号
resetn	1	input	复位信号，低电平有效
div	1	input	除法运算命令，在除法完成后，如果外界没有新的除法进入，必须将该信号置为 0
div_signed	1	input	控制有符号除法和无符号除法的信号
x	[31:0]	input	被除数
y	[31:0]	input	除数
s	[31:0]	output	除法结果，商
r	[31:0]	output	除法结果，余数
complete	1	output	除法完成信号，除法内部 count 计算达到 33

1.2.4 除法器进一步优化

首先，以上举例，除法都是一位试商，但其实可以考虑二位试商，其代价就是时序变差、资源消耗更多。

其次，可以观察以下除法：

(1) 0x7777_7777/0x7777_7776，会发现前面 31 位商都是 0。

(2) 0x2000_0001/0x2,会发现，后面 30 位商都是 0。

(3) 0x2000_0003/0x2 会发现，中间 29 位商是 0。

会发现，迭代除法中经常会出现一堆零，在软件程序中也是如此的，特别是商的首部出现一堆 0 的情况是十分普遍的。所以，可以考虑提前开始或提前结束的除法，关于中间一堆 0 的情况，也可以加速迭代，但需要一定的设计技巧。

注意，所有的除法实现方法，都是考虑硬件上好实现，很多我们认为很简单的操作，硬件实现却比较复杂，比如在一个 32 位数中，确定首部连续 0 的个数，也就是查找一个 32 位数的前导 0，看起来很简单，但硬件实现却是比较费资源的。

1.2.5 除法器模块级验证

在除法模块实现后，最好对其进行模块级随机验证，确保功能正确。可以使用以下 testbench：

```
`timescale 1ns / 1ps

module div_tb;
```

```
      // Inputs
      reg div_clk;
      reg resetn;
      wire div;
      reg div_signed;
      reg [31:0] x;
      reg [31:0] y;

      // Outputs
      wire [31:0] s;
      wire [31:0] r;
      wire complete;

      // Instantiate the Unit Under Test (UUT)
      div uut (
        .div_clk(div_clk),
        .resetn(resetn),
        .div(div),
        .div_signed(div_signed),
        .x(x),
        .y(y),
        .s(s),
        .r(r),
        .complete(complete)
      );

      initial begin
        // Initialize Inputs
        resetn = 0;
      #100;
        resetn = 1;
      end
initial
begin
    div_clk = 1'b0;
    forever
    begin
            #5 div_clk = 1'b1;
            #5 div_clk = 1'b0;
       end
end

//产生除法命令，正在进行除法
reg div_is_run;
integer wait_clk;
initial
begin
    div_is_run <= 1'b0;
    forever
    begin
            @(posedge div_clk);
        if (!resetn || complete)
            begin
                div_is_run <= 1'b0;
              wait_clk <= {$random}%4;
            end
            else
            begin
                repeat (wait_clk)@(posedge div_clk);
                div_is_run <= 1'b1;
                wait_clk <= 0;
            end
    end
end
//随机生成有符号乘法控制信号和乘数
assign div = div_is_run;
always @(posedge div_clk)
begin
    if (!resetn || complete)
    begin
          div_signed <= 1'b0;
          x          <= 32'd0;
```

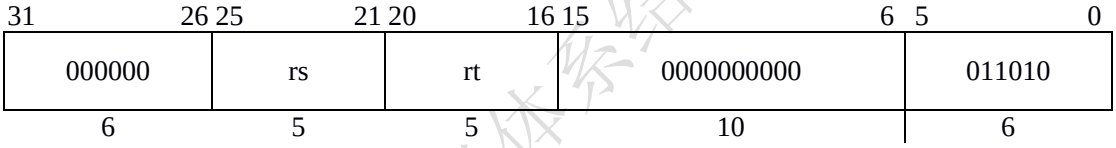
```
          y          <= 32'd1;
    end
    else if (!div_is_run)
       begin
          div_signed <= {$random}%2;
          x          <= $random;
          y          <= $random;  //被除数随机产生 0 的概率很小，基本可忽略
       end
end

//-----{计算参考结果}begin
//第一步，求 x 和 y 的绝对值，并判断商和余数的符号
wire x_signed = x[31] & div_signed;                //x 的符号位，做无符号时认为是 0
wire y_signed = y[31] & div_signed;                //y 的符号位，做无符号时认为是 0
wire [31:0] x_abs;
wire [31:0] y_abs;
assign x_abs = ({32{x_signed}}^x) + x_signed;      //此处异或运算必须加括号，
assign y_abs = ({32{y_signed}}^y) + y_signed;      //因为 verilog 中+的优先级更高
wire s_ref_signed = (x[31]^y[31]) & div_signed;//运算结果商的符号位，做无符号时认为是 0
wire r_ref_signed = x[31] & div_signed;         //运算结果余数的符号位，做无符号时认为是
0

//第二步，求得商和余数的绝对值
reg [31:0] s_ref_abs;
reg [31:0] r_ref_abs;
always @(div_clk)
begin
    s_ref_abs <= x_abs/y_abs;
    r_ref_abs <= x_abs-s_ref_abs*y_abs;
end

//第三步，依据商和余数的符号位调整
wire [31:0] s_ref;
wire [31:0] r_ref;
//此处异或运算必须加括号，因为 verilog 中+的优先级更高
assign s_ref = ({32{s_ref_signed}}^s_ref_abs) + {30'd0,s_ref_signed};
assign r_ref = ({32{r_ref_signed}}^r_ref_abs) + r_ref_signed;
//-----{计算参考结果}end

//判断结果是否正确
wire s_ok;
wire r_ok;
assign s_ok = s_ref==s;
assign r_ok = r_ref==r;
reg [5:0] time_out;

//输出结果,将各 32 位(不论是有符号还是无符号数)扩展成 33 位有符号数，以便以 10 进制形式打印
wire signed [32:0] x_d     = {div_signed&x[31],x};
wire signed [32:0] y_d     = {div_signed&y[31],y};
wire signed [32:0] s_d     = {div_signed&s[31],s};
wire signed [32:0] r_d     = {div_signed&r[31],r};
wire signed [32:0] s_ref_d = {div_signed&s_ref[31],s_ref};
wire signed [32:0] r_ref_d = {div_signed&r_ref[31],r_ref};
always @(posedge div_clk)
begin
    if (complete && div) //除法完成
    begin
            if (s_ok && r_ok)
          begin
              $display("[time@%t]: x=%d, y=%d, signed=%d, s=%d, r=%d, s_OK=%b,
r_OK=%b",
                       $time,x_d,y_d,div_signed,s_d,r_d,s_ok,r_ok);
          end
          else
          begin
              $display("[time@%t]Error:  x=%d,  y=%d,  signed=%d,  s=%d,  r=%d,
s_ref=%d, r_ref=%d, s_OK=%b, r_OK=%b",

$time,x_d,y_d,div_signed,s_d,r_d,s_ref_d,r_ref_d,s_ok,r_ok);
               $finish;
          end
```

```
    end
end
always @(posedge div_clk)
begin
    if (!resetn || !div_is_run || complete)
       begin
           time_out <= 6'd0;
       end
    else
    begin
           time_out <= time_out + 1'b1;
    end
end
always @(posedge div_clk)
begin
    if (time_out == 6'd34)
       begin
          $display("Error: div no end in 34 clk!");
          $finish;
       end
end

endmodule
```

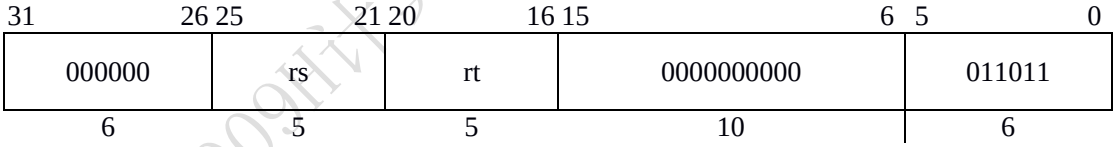
注意 testbench 里很多语法都是不可综合的，比如系统任务$signed(),循环语句 forever 等

1.2.6 将除法器集成到自己的 CPU 中

除法器验证无误后需要用除法器在 CPU 中实现 DIV（有符号除法）和 DIVU（无符号除法）指令，DIV 指令编码如下：



DIVU 指令编码如下：



这两个除法指令都要求用 rs 除以 rt，将商放到 LO 寄存器，将余数放在 HI 寄存器。

这两个除法指令都没有例外，所以编译器在编译除法指令时一般都会加上额外的语句来排除除零例外和溢出例外(有符号除法的 0x80000000÷0xffffffff 会产生溢出例外)。所以我们在设计除法器时不必考虑以上的情况。

由于除法要迭代执行，所以除法迭代的那级流水级(执行级)在除法未完成时就必须要阻塞流水线。

1.3 思考题

1、可以对 2 位 booth 编码器进行更进一步的优化吗？

2、为什么在迭代除法器中，用被除数的绝对值除以除数的绝对值就可以得到商的绝对值和余数的绝对值？这和对余数符号的规定有什么关系？。

3、迭代除法可以提前提前开始/结束吗？