Lab4 乘除法器实验

乘法是主要的数据运算之一。在处理器、数字信号处理和数字通信等领域有着广泛的应用，乘法运算的性能往往影响整个系统的运行速度。如何实现快速高效的乘法器关系着整个系统的运算速度和资源效率。根据手工乘法运算的特性，通常采用移位相加的算法，将乘法运算变成加法，通过逐步移位相加实现乘法。这种算法需要使用寄存器保存每次加法后的部分积，花费时间较长。为了提升乘法运算执行速度，阵列乘法器只使用组合电路，通过加法器阵列来实现乘法运算。在 FPGA 芯片中乘法器通常作为 DSP 核的一部分可直接引用。

除法运算与乘法运算很相似，都是一种移位和加减运算的迭代过程，但比乘法运算更加复杂。

一、实验目的

1) 掌握无符号数和带符号数乘法器的设计方法。
2) 掌握快速乘法器的设计方法。
3) 掌握无符号和带符号数除法器的设计方法。
4) 掌握RV32M指令集的实现方法。

二、实验环境

1. Vivado 开发环境
2. Xilinx A7-100T 实验板

三、实验原理

1、无符号数乘法器设计

计算机中两个无符号数相乘，类似手算乘法。为了提高效率，做了相应改进。主要的改进措施有以下几个方面。

① 每次将乘数 Y 的一位乘以被乘数得 $X \times Y_i$ 后，就将该结果与前面所得的结果累加，得到 P_i，称之为

部分积。因为没有等到全部计算后一次求和，所以减少了保存每次相乘结果 $X \times Y_i$ 的开销。

② 在每次求得 $X \times Y_i$ 后，不是将它左移与前次部分积 P_i 相加，而是将部分积 P_i 右移一位与 $X \times Y_i$ 相加。

③ 对乘数中为 1 的位执行加法和右移运算，对为 0 的位只执行右移运算，而不需执行加法运算。

因为每次进行加法运算时，只需要将 X×Yi 与部分积中的高 n 位进行相加，低 n 位不会改变，因此，只需用 n 位加法器就可实现两个 n 位数的相乘。

部分积 Pi 和 X 进行无符号数相加，可能会产生进位，因而需要 1 位进位 C。整个迭代过程从乘数最低位 Yn 和 P0 = 0 开始，经过 n 次“判断–加法–右移”循环，直到求出 Pn 为止。Pn 就是最终的乘积。假定每次循环在一个时钟周期内完成，则 n 位乘法需要用 n 个时钟周期来完成。

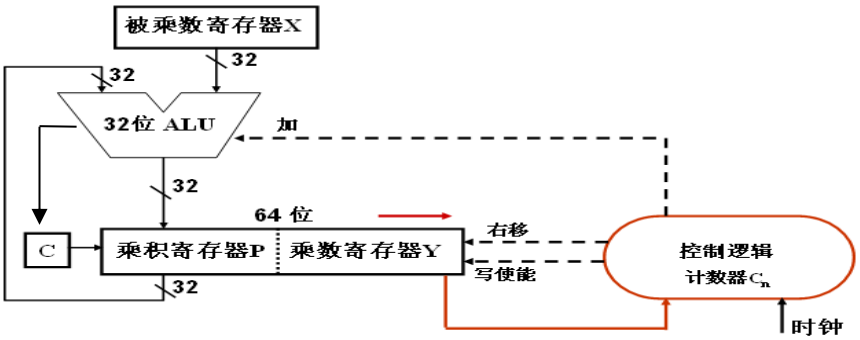


图 4.1　原码一位乘法的逻辑结构图

实现两个 32 位无符号数乘法的逻辑结构图如图 4.1 所示。被乘数寄存器 X 用于存放被乘数；乘积寄存器 P 开始时置初始部分积 P0 = 0，结束时存放的是 64 位乘积的高 32 位；乘数寄存器 Y 开始时置乘数，结束时存放的是 64 位乘积的低 32 位；进位 C 表示加法器的进位信号；计数器 Cn 存放循环次数，初值是 32，每循环一次，Cn 减 1，当 Cn = 0 时，乘法运算结束；加法器是乘法核心部件，在控制逻辑控制下，对乘积寄存器 P 和被乘数寄存器 X 的内容进行“加”运算。

每次循环都要对进位 C、“加”运算结果和乘数寄存器 Y 实现同步“右移”，此时，进位信号 C 移入寄存器 P 的最高位，“加”运算结果的最低位移入到寄存器 Y 的最高位，寄存器 Y 的最低位移出，在“写使能”控制下右移结果被更新到寄存器 P 和寄存器 Y。从最低位 Yn 开始，逐次把乘数的各个数位 Yn-i 移到寄存器 Y 的最低位上。因此，寄存器 Y 的最低位被送到控制逻辑以决定被乘数是否“加”到部分积上。

引用 32 位加法器模块，采用行为建模方式实现的 32 位无符号数乘法的模块代码参考设计如下：

```
module mul_32u(
    output [63:0] p,          //乘积
    output out_valid,         //高电平有效时，表示乘法器结束工作
    input clk,                //时钟
    input rst,                //复位信号
    input [31:0] x,           //被乘数
    input [31:0] y,           //乘数
    input in_valid            //高电平有效，表示乘法器开始工作
```

```
);
        reg [5:0] cn;                   //移位次数寄存器
        always @(posedge clk or posedge rst) begin
          if (rst) cn <= 0;
          else if (in_valid) cn <= 32;
          else if (cn != 0) cn <= cn - 1;
        end
        reg [31:0] rx, ry, rp;          //加法器操作数和部分积
        wire [31:0] Add_result;     //加法运算结果
        wire cout;                      //进位
       // adder32 是 32 位加法器模块的实例化，参见实验 3 的设计
        Adder32 my_adder(.f(Add_result),.cout(cout),.x(rp),.y(ry[0] ? rx : 0),.sub(0));
        always @(posedge clk or posedge rst) begin
          if (rst) {rp, ry, rx} <= 0;
          else if (in_valid) {rp, ry, rx} <= {32'b0, y, x};
          else if (cn != 0) {rp, ry} <= {cout, Add_result, ry} >> 1;
        end
        assign out_valid = (cn == 0);
        assign p = {rp, ry};
endmodule
```

2、无符号数除法器设计

除法运算与乘法运算很相似，都是一种移位和加减运算的迭代过程，但比乘法运算更加复杂。在进行定点数除法运算前，首先要对被除数、除数的取值和大小进行相应的预处理，以判断被除数是否为 0、商是否为 0、除数是否为 0 和商是否溢出等异常情形。预处理的操作步骤如下：

(1) 若被除数为 0、除数不为 0，或者定点整数除法时|被除数|＜|除数|，则说明商为 0，余数为被除数，不再继续执行。

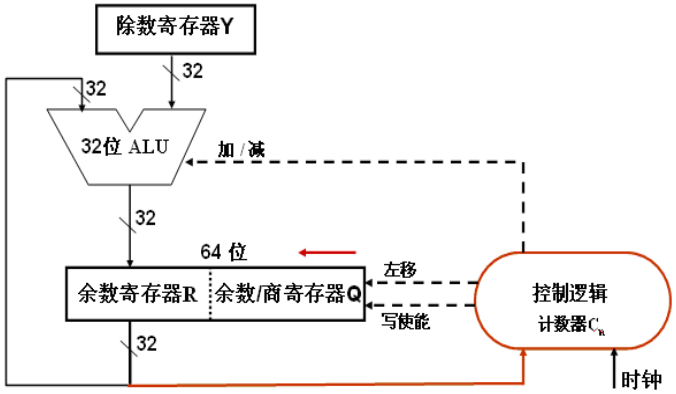
(2）若被除数不为 0、除数为 0，对于整数，则发生“除数为 0”异常；对于浮点数，则结果为无穷大。

(3）若被除数和除数都为 0，对于整数，则发生除法错异常；对于浮点数，则有些机器产生一个不发信号的 NaN，即“quiet NaN”。

只有当被除数和除数都不为 0，并且商也不可能溢出（例如，补码中最大负数除以-1 时会发生溢出）时，才进一步进行除法运算。

除法逻辑结构类似于乘法逻辑结构，如图 4.2 所示是一个 32 位除法逻辑结构示意图。除数寄存器 Y 存放除数；余数寄存器 R 开始时置被除数的高 32 位，作为初始中间余数 R0 的高位部分，结束时存放的是余数；余数/商寄存器 Q 开始时置被除数的低 32 位，作为初始中间余数 R0 的低位部分，结束时存放的是 32 位商，在运算过程中，Q 中存放的并不是商的全部位数，而是部分为被除数或中间余数，部分为商，只

有到最后一步才是商的全部位数；计数器 Cn 存放循环次数，初值是 32，每循环一次，Cn 减 1，当 Cn = 0 时，除法运算结束；ALU 是除法器核心部件，在控制逻辑控制下，对于余数寄存器 R 和除数寄存器 Y 的



内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器 R。

图 4.2　32 位无符号数除法运算逻辑结构框图

每次循环都要对寄存器 R 和 Q 实现同步“左移”，左移时，Q 的最高位移入 R 的最低位，Q 中空出的最低位上被上“商”。从低位开始，逐次把商的各个数位左移到 Q 中。每次由控制逻辑根据 ALU 运算结果的符号位来决定上商为 0 还是 1。

n 位定点数的除法，实际上是用一个 2n 位的数去除以一个 n 位的数，得到一个 n 位的商。在实现时需要对被除数的扩展，对于两个 n 位无符号整数相除时，只要将被除数 X 的高位添 n 个 0 即可。即 $X = x_{n-1}x_{n-2}\ldots x_1x_0$ 变成 $X = 00\ldots00\ x_{n-1}x_{n-2}\ldots x_1x_0$。对被除数预置时，R 寄存器中为全 0，Q 寄存器中为被除数 X。

两个无符号数除法的运算步骤和算法要点如下：

① 操作数预置：在确认被除数和除数都不为 0 后，将被除数（必要时进行 0 扩展）置于余数寄存器 R 和余数/商寄存器 Q 中，除数置于除数寄存器 Y 中。

② 做减法试商：根据 R–Y 得到的结果的符号来判断两数的大小。若结果为正，则上商 1，若结果为负，则上商 0。

③ 上商为 0 时恢复余数：把减掉的除数再加回来，恢复原来的中间余数。

④ 中间余数左移，重复第 2 步以便继续试商。

上述给出的算法要点③中，采用了“上商为 0 时恢复余数”的方式，这种方法称为“恢复余数法”。也可以在下一步运算时把当前多减的除数补回来，这种方法称为“不恢复余数法”，又称“加减交替法”。根据余数恢复方式的不同，有“恢复余数除法”和“不恢复余数除法”两种。

参照乘法器的设计，引用 32 位加减法器模块，采用行为建模方式实现的 32 位无符号数除法器的模块代码参考如下：

```
module div_32u(
    output   [31:0] Q,              //商
    output   [31:0] R,              //余数
    output out_valid,           //除法运算结束时，输出为 1
```

```
    output in_error,              //被除数或除数为 0 时，输出为 1
    input clk,                    //时钟
    input rst,                   //复位信号
    input [31:0] X,               //被除数
    input [31:0] Y,               //除数
    input in_valid               //输入为 1 时，表示数据就绪，开始除法运算
);
    reg [5:0] cn;
    reg [63:0] RDIV;
    reg [31:0] TempQ;
    reg temp_out_valid;
    wire [31:0] diff_result;
    wire cout;
    assign in_error = ((X == 0) || (Y == 0));   //预处理，除数和被除数异常检测报错
    assign out_valid=in_error;                          //如果检测异常，则结束运算
    always @(posedge clk or posedge rst) begin
      if (rst)    cn <= 0;
      else if (in_valid) cn <= 32;
      else if (cn != 0) cn <= cn - 1;
    end
   // adder32 是 32 位加法器模块的实例化，参见实验 3 的设计
    Adder32 my_adder(.f(diff_result),.cout(cout),.x(RDIV[63:32]),.y(Y),.sub(1)); //减法，当 cout=0 时，
表示有借位。
    always @(posedge clk or posedge rst) begin
      if (rst) RDIV = 0;
      else if (in_valid) begin RDIV = {32'b0, X};TempQ=32'b0; temp_out_valid=1'b0;end
      else if ((cn >= 0)&&(!out_valid)) begin
                          if(cout)    begin     //判断是否有借位，=1，表示够减，没有借位
                                RDIV[63:32] = diff_result[31:0]; //把差值赋值到中间被除数
的高 32 位
                                 TempQ[cn]=1'b1;              //商在该位置 1
                              end
                          if(cn>0)        RDIV=RDIV <<1;
                          else temp_out_valid=1'b1;
                         end
    end
    assign out_valid = temp_out_valid;
    assign Q = TempQ;
    assign R =RDIV[63:32];
endmodule
```

对于大多数处理器来说，除法运算都是相对较慢的操作。当除数为常量时，常采用近似算法来替代，例如除数为 2 的幂次的无符号除法可以用右移来代替，相当于通过乘以除数的倒数。对于非 2 的幂次的除数则需要修正积的高 32 位，使用这种方法可以优化除法的运算速度。

四、实验内容

1、补码 1 位乘法器设计

补码作为机器中带符号整数的表示形式，需要计算机能实现定点补码整数的乘法运算。根据每次部分积是一位相乘得到还是两位相乘得到，有补码一位乘法和补码两位乘法。

A.D.Booth 提出了一种补码相乘算法，可以将符号位与数值位合在一起参与运算，直接得出用补码表示的乘积，且正数和负数同等对待。这种算法被称为 Booth（布斯）算法。

计算机中操作数的长度都是字节的倍数，因而其位数应该是偶数。假定两个偶数位的带符号整数 x 和 y 的机器级表示分别为 $[x]_{补}$和 $[y]_{补}$，$[x\times y]_{补}$的 Booth 一位乘法运算规则如下：

① 乘数最低位增加一位辅助位 $Y_{-1}=0$。

② 根据 $Y_i\ Y_{i-1}$ 的值，决定下一步的运算：当 $Y_i\ Y_{i-1}$=00 或者=11 时，不操作；当 $Y_i\ Y_{i-1}$=01 时，部分积 P_i 加被乘数 X；当 $Y_i\ Y_{i-1}$=10 时，部分积 P_i 减被乘数 X。

③ 每次加减后，算术右移一位，得到部分积 P_{i+1}。

④ 重复第②和第③步 n 次，结果得 $[x\times y]_{补}$。

图 4.3 是实现 32 位补码一位乘法的逻辑结构图，和图 4.1 所示的无符号数乘法电路的逻辑结构很类似，只是部分控制逻辑不同。

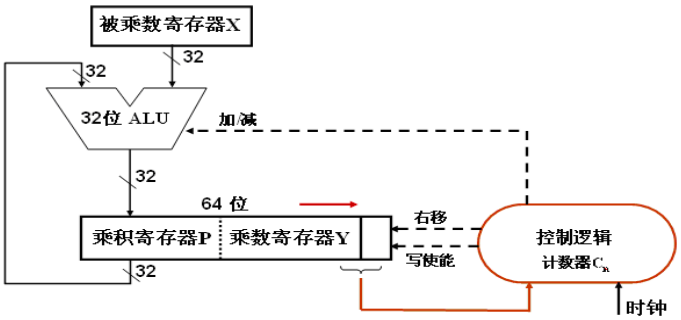


图 4.3　实现补码一位乘法的逻辑结构图

采用行为建模方式的设计的 32 位补码一位乘法器的模块源代码如下：

```
module mul_32b (
    output [63:0] p,          //乘积
    output out_valid,         //高电平有效时，表示乘法器结束工作
    input clk,                //时钟
    input rst,                //复位信号
    input [31:0] x,           //被乘数
```

```
    input [31:0] y,              //乘数
    input in_valid               //高电平有效，表示乘法器开始工作
);
  //add your code here
endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab4.zip里的mul_32b.v文件。

3、 点击添仿真测试文件，加入lab4.zip里的mul_32b_tb.v文件。

4、 根据实验要求，完成源码文件的设计。

5、 对工程进行仿真测试，运行足够的时间，分析验证输入输出时序波形和控制台信息。

6、 仿真通过后，进行综合、实现，分析电路的资源占用情况。

2、快速乘法器

在程序设计的运算中大约 1/3 是乘法运算，实现高速乘法运算有助于提升计算机系统的性能。一方面可以通过一次判断两位（多位）乘数来提高乘法速度，但是多位乘法运算的控制复杂度呈几何级数增长，实现难度很大。另一方面，随着大规模集成电路技术的飞速发展，出现了采用硬件叠加或流水处理的快速乘法器件，如阵列乘法器就是其中之一，采用全加器阵列来实现。

大多数组合乘法都是基于手算的移位-累加算法来实现。图 4.4 说明了 8*8 乘法器的基本思想，被乘数 X=x7x6x5x4x3x2x1x0，乘数 Y=y7y6y5y4y3y2y1y0,X 和 Y 都是无符号整数，每一行称为一个乘积分量，表示移位的被乘数，每个小方格表示位积 yi·xj，乘积分量相加得到乘积 P=p15p14… p2p1p0。

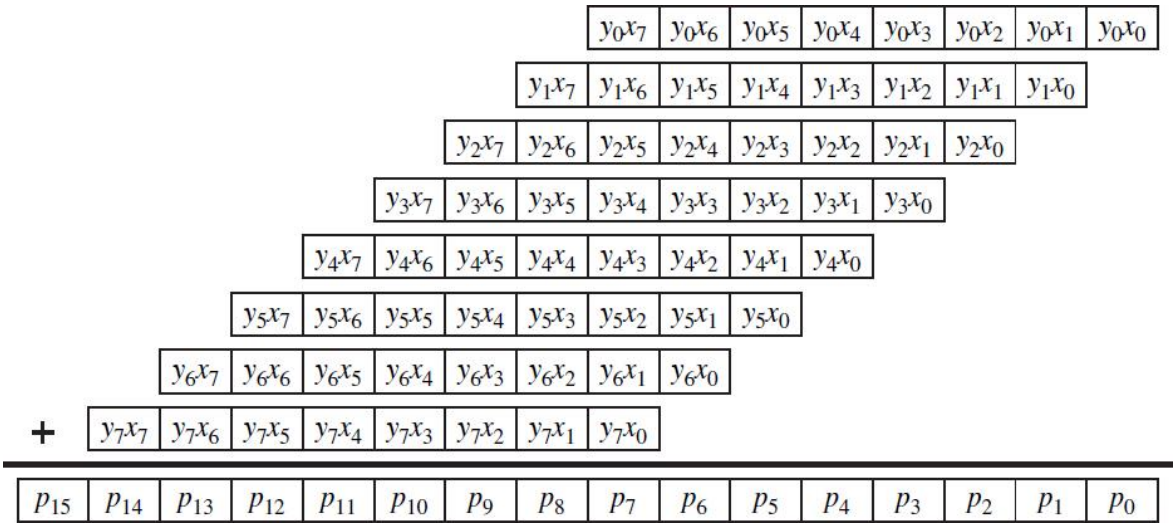


图 4.4 8*8 乘法器的部分积

图 4.5 表示加法阵列可改用基于 CSA（Carry Save Adder，进位保留加法器）方式的结构。CSA 将本级进位与本级和一样同时输出至下一级，而不是向前传递到本级的下一位，因而求和速度快，且向下级传递的速度与字长无关。

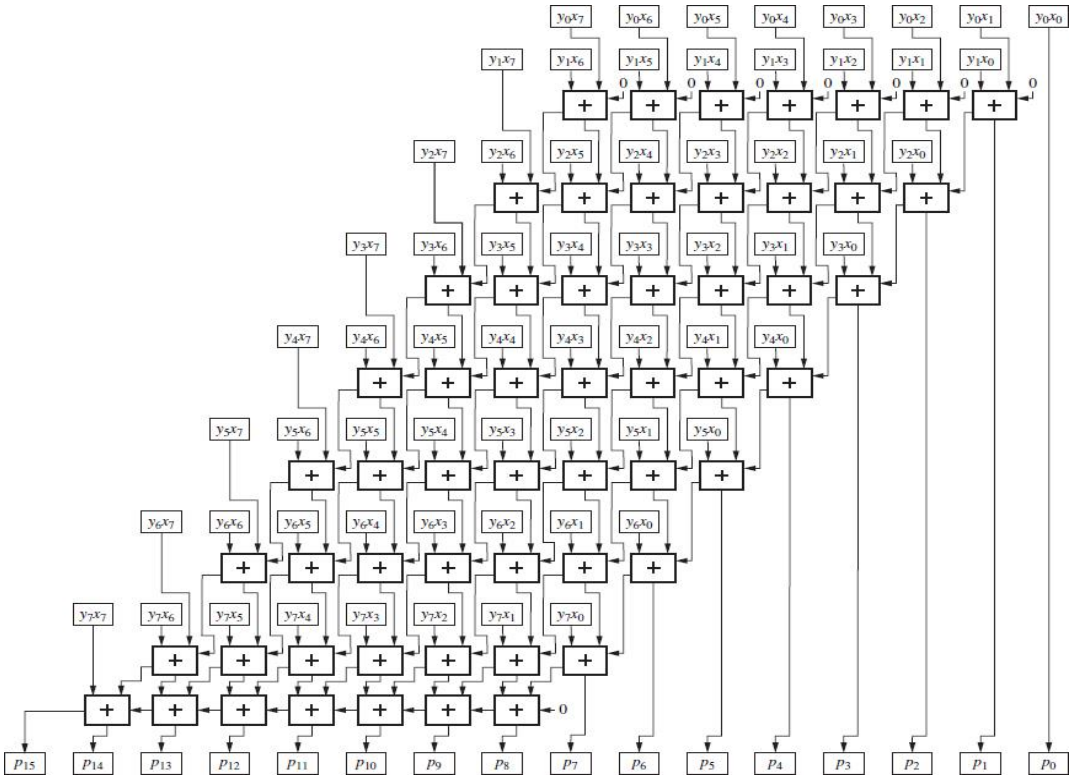


图 4.5 基于 CSA 的 8x8 阵列乘法器原理图

阵列乘法器结构规范，标准化程度高，有利于布局布线，适合用超大规模集成电路实现，且能获得较高的运算速度，其乘法速度仅取决于逻辑门和加法器的传输延迟。

设置数据位宽为 32 位，采用合适的阵列结构实现 32 位阵列乘法器 multi_32k，假设模块接口定义如下：

```
module mul_32k (
      output [63:0] p,
      input [31:0] x,
      input [31:0] y
   );
    //add your code here
endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab4.zip里的mul_32k.v文件。

3、 点击添仿真测试文件，加入lab4.zip里的mul_32k_tb.v文件。

4、 根据实验要求，完成源码文件的设计。

5、 对工程进行仿真测试，运行足够的时间，分析验证输入输出时序波形和控制台信息。

6、 仿真通过后，进行综合、实现，分析电路的资源占用情况。

3、补码除法器

补码作为带符号数的表示形式，与补码的其它运算一样，补码除法也可以将符号位和数值位合并在一起进行运算，而且商的符号位直接在除法运算中产生。对于两个 n 位二进制数补码除法，被除数需要进行 n 位符号扩展。若被除数为 2n 位，除数为 n 位，则被除数无须符号位扩展。

和无符号数一样，首先要对被除数和除数的取值、大小等进行相应的预处理，以确定除数是否为 0、商是否为 0、是否溢出等异常情况。

因为补码除法中被除数、中间余数和除数都是有符号的，所以不像无符号数除法那样可以直接用做减法来判断是否够减，而应该根据被除数（中间余数）与除数之间符号的异同或差值的正负来确定下次做减法还是加法，再根据加或减运算的结果来判断是否够减。表 4.1 给出了判断是否够减的规则。

表 4.1 补码除法判断是否够减的规则

中间余数 R	除数 Y	新中间余数：$R–Y$		新中间余数：$R+Y$	
		0	1	0	1
0	0	够减	不够减		
0	1			够减	不够减
1	0			不够减	够减
1	1	不够减	够减		

从表 4.1 可看出，当被除数（中间余数）与除数同号时做减法；异号时，做加法。若加减运算后得到的新余数与原余数符号一致（余数符号未变）则够减；否则不够减。

根据是否立即恢复余数，补码除法也分为恢复余数法和不恢复余数法两种。下面主要给出不恢复余数法的算法要点。

根据表 4.1 给出的 n 位补码除法判断是否够减的判断规则，可以得到如下不恢复余数除法的算法要点。

(1) 操作数的预置：除数装入除数寄存器 Y，被除数 X 符号位扩展后装入余数寄存器 R 和余数/商寄存器 Q。

(2) 根据以下规则求第一位商 Q_n。

若 X 与 Y 同号，则做减法，即 $R_1=R-Y$；否则，做加法，即 $R_1=R+Y$，并按以下规则确定商值 Q_n。

① 若新的中间余数 R_1 与 Y 同号，则 Q_n 置 1，转第（3）步。

② 若新的中间余数 R_1 与 Y 异号，则 Q_n 置 0，转第（3）步。

Q_n 用来判断是否溢出，而不是真正的商。以下情况下会发生溢出：X 与 Y 同号且上商 $Q_n=1$，或者，X 与 Y 异号且上商 $Q_n=0$。

(3) 对于 $i=1$ 到 n，按以下规则求出相应商。

① 若 R_i 与 Y 同号，则 Q_{n-i} 置 1，$R_{i+1}=2R_i–Y$，$i=i+1$。

② 若 R_i 与 Y 异号，则 Q_{n-i} 置 0，$R_{i+1}=2R_i+Y$，$i=i+1$。

(4) 商的修正：最后一次 Q 寄存器左移一位，将最高位 Q_n 移出，并在最低位置上商 Q_0。若被除数与除数同号， Q 中就是真正的商；否则，将 Q 中的商的末位加 1。

(5) 余数的修正：若余数符号同被除数符号，则不需修正，余数在 R 中；否则，按下列规则进行修正：当被除数和除数符号相同时，最后余数加除数；否则，最后余数减除数。

假设 32 位补码除法器的模块的接口定义如下：

```
module div_32b (
    output   [31:0] Q,              //商
    output   [31:0] R,              //余数
    output out_valid,           //除法运算结束时，输出为 1
    output in_error,            //被除数或除数为 0 时，输出为 1
    input clk,                  //时钟
    input rst,                  //复位信号
    input [31:0] X,             //被除数
    input [31:0] Y,             //除数
    input in_valid              //输入为 1 时，表示数据就绪，开始除法运算
   );
  //add your code here
endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab4.zip里的div_32b.v文件。

3、 点击添仿真测试文件，加入lab4.zip里的div_32b_tb.v文件。

4、 根据实验要求，完成源码文件的设计。

5、 对工程进行仿真测试，运行足够的时间，分析验证输入输出时序波形和控制台信息。

6、 仿真通过后，进行综合、实现，分析电路的资源占用情况。

4、RV32M 指令实现

RV32M 类型指令是 RISC-V 基础指令集的扩展，为 RISC-V 架构提供了一组特定的指令，提供了一系列整数乘除指令，包括无符号数和带符号数的乘除指令、取余数指令等。这些指令用于执行高效的整数乘除运算，实现快速、高效的数据处理操作，从而提高系统的性能和效率。

RV32M 的指令都是 R 型指令，指令操作码如图 4.6 所示，指令描述及功能如表 4.2 所示。

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

图 4.6　RV32M 指令操作码布局图

表 4.2 RV32M 指令功能

序号	指　令	描　述	功能
1	乘法 mul	x[rd]=x[rs1]×x[rs2]	把寄存器 x[rs2]乘到寄存器 x[rs1]上，乘积写入 x[rd]。忽略算术溢出。
2	高位乘 mulh	x[rd]=(x[rs1]$_s$×$_s$x[rs2])≫$_s$32	把寄存器 x[rs2]乘到寄存器 x[rs1]上，都视为 2 的补码，将乘积的高位写入 x[rd]。
3	高位有符号-无符号乘 mulhsu	x[rd]=(x[rs1]$_s$×$_u$x[rs2])≫$_s$32	把寄存器 x[rs2]乘到寄存器 x[rs1]上，x[rs1]为 2 的补码，x[rs2]为无符号数，将乘积的高位写入 x[rd]。
4	高位无符号乘 mulhu	x[rd]=(x[rs1]$_u$×$_u$x[rs2])≫32	把寄存器 x[rs2]乘到寄存器 x[rs1]上，x[rs1]、x[rs2]均为无符号数，将乘积的高位写入 x[rd]。
5	除法 div	x[rd] = x[rs1] ÷$_s$ x[rs2]	用寄存器 x[rs1]的值除以寄存器 x[rs2]的值，向零舍入，都视为 2 的补码，把商写入 x[rd]。
6	无符号除法 divu	x[rd] = x[rs1] ÷$_u$ x[rs2]	用寄存器 x[rs1]的值除以寄存器 x[rs2]的值，向零舍入，都视为无符号数，把商写入 x[rd]。
7	求余数 rem	x[rd]=x[rs1] %$_s$ x[rs2]	x[rs1]除以 x[rs2]，向 0 舍入，都视为 2 的补码，余数写入 x[rd]。
8	求无符号数的余数 remu	x[rd]=x[rs1] %$_u$ x[rs2]	x[rs1]除以 x[rs2]，向 0 舍入，都视为无符号数，余数写入 x[rd]。

要求引用上述乘除法设计模块，使用立即数代替指令中的寄存器赋值，通过 3 位功能选择码实现 RV32M 指令集。假设 rv32m 的模块端口定义如下：

```
module rv32m
(
    output   [31:0] rd,            //运算结果
```

```
    output out_valid,          //运算结束时，输出为 1
    output in_error,           //运算出错时，输出为 1
    input clk,                 //时钟
    input rst,                 //复位信号，低有效
    input [31:0] rs1,          //操作数 rs1
    input [31:0] rs2,          //操作数 rs2
    input [2:0] funct3，        //3 位功能选择码
    input in_valid             //输入为 1 时，表示数据就绪，开始运算
  );
  //add your code here
endmodule
```

为了能够在实验板进行验证，操作数使用重复的 4 位二进制数，输出通过 led 指示灯和七段数码管来表示。具体定义如下：拨档开关 0~3 表示操作数 x；拨档开关 4~7 表示操作数 y；拨档开关 12~15 表示功能选择 funct3；如果需要，拨档开关 9 表示数据就绪 in_valid，拨档开关 10 表示复位信号 rst。32 位的 rd 结果显示在 8 个七段数码管上，低 16 位显示在 16 个 led 指示灯上，指令运算结束 out_valid 显示在三色 led1 的绿色灯上，错误标志 in_error 显示在三色 led2 等红色灯上。

建立 rv32m 的父级模块 rv32m_top，其端口定义如下：

```
module rv32m_top
(
    output   [15:0] rd_l,      //运算结果的低 16 位
    output out_valid,          //运算结束时，输出为 1
    output in_error,           //运算出错时，输出为 1
    output [6:0] segs,         // 7 段数值
    output [7:0] AN,           //8 个数码管，显示 32 位运算结果 rd
    input clk,                 //时钟
    input rst,                 //复位信号，低有效
    input [3:0] x,             //操作数 1，重复 8 次后作为 rs1
    input [3:0] y,             //操作数 2，重复 8 次后作为 rs2
    input [2:0] funct3，        //3 位功能选择码
    input in_valid             //输入为 1 时，表示数据就绪，开始运算
  );
  //add your code here

endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab4.zip里的rv32m.v、rv32m_top.v文件。

3、 点击添仿真测试文件，加入lab4.zip里的rv32m_tb.v文件。

4、 点击添加约束文件，加入lab4.zip里的rv32m_top.xdc文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试，运行足够的时间，分析验证输入输出时序波形和控制台信息。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成比特流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1、分析 rv32m 模块的资源占用和性能特点。

2、阐述浮点数的乘除法如何实现？

3、分析当除数是一个常量时，如何通过乘以常量倒数的方法来得到近似的结果。与除法运算进行对比分析，比如常量是 3 或者 7 时。

4、通过查找资料，阐述提高乘除法器的运算性能的思路和方法。