设计报告

1_D089

一、设计思路概述

本次题目要求是设计一个简易的电子计算器电路，可实现整数（0～99999999）的加、减、乘、除和开方的算数运算，运算的结果可以保存在运算结果处理电路中。故本次设计主要分成三个模块：数据输入处理电路、算数运算电路及运算结果处理电路模块。以下分别就三个模块的设计思路及发挥部门两个模块进行**概述**说明。

基本要求部分

（一）数据输入处理电路

本模块完成的主要功能是将从键盘输入的数据变为存储在寄存器中的便于后续计算的数据。

主要思路是：通过运算结果处理电路反馈信号 BUSY_P 和算数运算电路反馈信号 BUSY_A，判断后两个模块的运算是否结束。若结束进行数据输入处理。设置一个标志寄存器区分输入的是第一个数据还是第二个数据。在第一个数据输入结束前，标志寄存器值不变，进入计算程序并将结果存储在 DATA_1 寄存器中；读入运算符后，将运算符对应二进制数存入 ARITH 寄存器中，并改变标志寄存器值，表明开始输入第二个数据，进入计算程序并将结果存储在 DATA_2 寄存器中。以上所提计算程序部分将在后续部分具体阐述。

（二）算数运算电路

本模块完成的主要功能是将存储在寄存器 DATA_1 和 DATA_2 中的数据按照 ARITH 寄存器中存储的相应运算符的指示，完成相应的加减乘除及开方操作。通过 case 语句进入不同的运算模式，并最后将计算结果保存在 DATA 寄存器中。

（三）运算结果处理电路

本模块完成的主要功能是将存储在寄存器中的十六进制运算结果转换成十进制结果输出；由于存储在寄存器中的十六进制数实际上是二进制数，故本模块的核心思想在于将二进制数转换成 BCD 码。通过判断输入数据的最高位确定被处理数是正数还是负数，对正负数进行分类讨论处理。核心转换部分采用“移位

加 3”算法实现。将移位及值修正后结果存储在中间变量寄存器中。当所有移位操作结束后将最终结果通过中间变量寄存器传递给 BCD 码寄存器，得到最终结果。

发挥部分

（一）防抖动模块

按键开关在闭合时不会马上稳定接通，在断开时也不会立刻断开。因而在闭合及断开的瞬间均伴随有一连串的抖动，导致按键输入不稳定。本模块设计的目的就是为了消除这种抖动，保证运算准确进行。

本模块的主要思想是：如果数据输入处理电路模块采用边沿触发，在输入有抖动的时候，相当于 KEY 相应位输出一段时间短且高频的脉冲波形。这样每来一个有效边沿就触发一次转换，相当于只按下一次按键，但数据输入电路却读入多个重复的数据，造成错误。

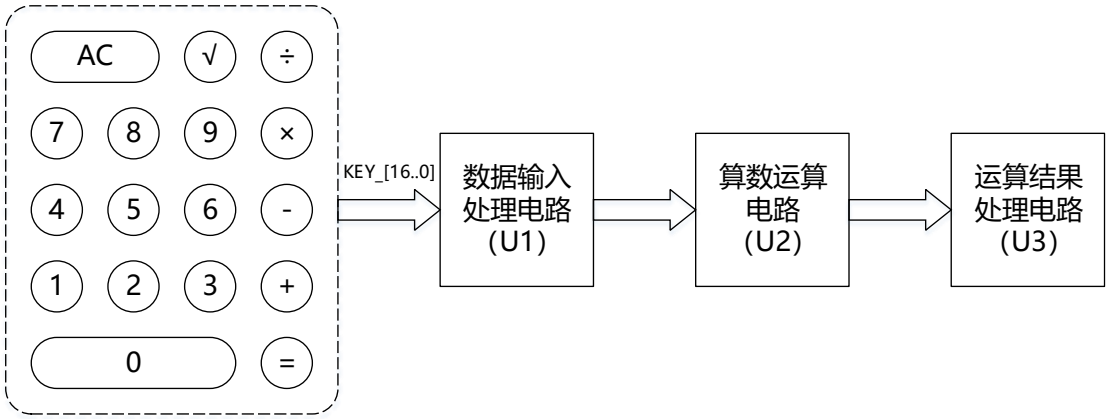
如果该边沿触发方式为查询方式，间隔一定时间对 KEY 对应位波形进行查询，避开多次边沿触发造成的误差，达到消除抖动的作用

（二）为除法运算和开方运算增加余数输出模块

Verilog 自带了求余运算符%，因此只要再加一路输出，同时输出余数和商即可。开方结果的余数为用原始数据减去开方计算结果的平方得到。

二、总体设计框图及详细说明

该电子计算器的总体功能连接框图如下所示：



图：电子计算器总体功能框图

其中键盘输入模块由仿真过程中写好的波形激励文件实现。需要变成设计的部分主要为：数据输入处理模块、算数运算电路模块及运算结果处理模块。接下

来对以上三个模块的具体设计思路及方法进行详细说明

基本要求部分

（一）数据输入处理电路

本模块完成的主要功能是将按键输入数据转换在存储在数据寄存器（DATA_1 及 DATA_2）中的，便于后续进行算数运算的二进制数据。其中还兼将输入的运算符转换成便于后续识别后续运算的、存储在寄存器（ARITH）中的二进制数据。其主要思路如下所示：

首先，对运算结果处理电路的反馈信号 BUSY_P 和算数运算电路的反馈信号 BUSY_A 的状态进行检测。只要两者有一个处在高电平，证明其中有一个模块正在执行相应操作，处在非空闲状态。无法输入数据。只有当两者均处在空闲状态，即两个反馈信号都为低电平时，才可以输入数据。

设置一个标志寄存器 **second flag** 来标记输入数据是属于第一个数据还是第二个数据。首先设标志寄存器 **second flag 为 0**，表明输入第一个数据。输入的数据按照题目中提供的‘表 1，独立按键定义’存储在寄存器 KEY 中。每来一个时钟上升沿，代表按下一个按键，相应的对寄存器 KEY 的值进行判断，通过 case 语句判断是代表哪个按键被按下。

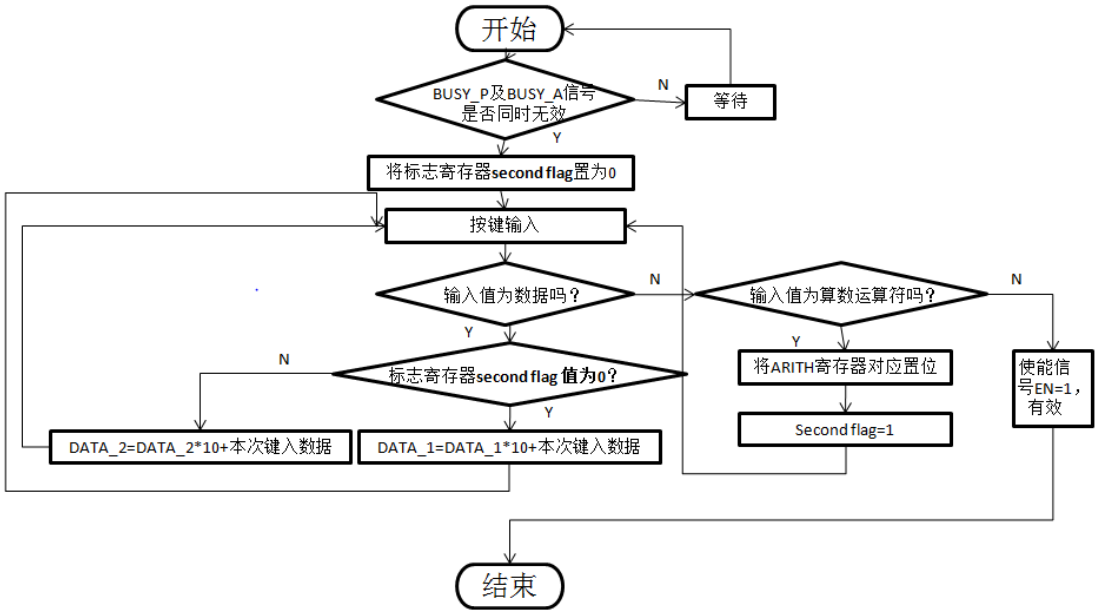
如果输入的是数据，则对标志寄存器 **second flag** 的值进行判断，若值为 0，代表输入的是第一个数据中的某一位，将 DATA_1 中原来存储的数据乘 10 并加上本次输入的数据；若值为 1，代表输入的是第二个数据中的某一位，将 DATA_2 中原来存储的数据称 10 并加上本次输入的数据。

如果输入的是运算符，则按照运算符设定规则（如下表 1 所示），将寄存器 ARITH 中赋予对应的值。并将标志寄存器 **second flag** 置 1，表明接下来要输入第二个数据。

运算符	对应二进制数
ADD（+）	000
SUB（—）	001
MUL（×）	010
DIV（÷）	011
SQRT（开方）	100

当检测到输入‘=’号，数据处理核心程序部分结束，将使能信号 EN 置为 1，对后续电路使能，表明后续电路可以开始工作。

以上算法的具体流程如下所示：



（二）算数运算电路

本模块完成的主要功能是将存储在寄存器 DATA_1 和 DATA_2 中的数据按照 ARITH 寄存器中存储的相应运算符的指示，完成相应的加减乘除及开方操作。

首先，令 BUSY_P 反馈信号置 1，表明此时 U2 模块正忙。接下来判断上级电路使能信号 EN 是否有效，若有效说明数据输入模块完成数据转换，可以开始进行运算模式。

进入主程序部分。首先，利用 case 语句判断当前进行的运算类型。

若运算为加法、乘法或除法，将两个数据寄存器中的数据进行简单相加、相乘、相除运算，并将最后运算结果存储在数据寄存器 DATA 中。

若运算为减法，首先判断两个运算数的相对大小。若运算数 1 大于等于运算数 2，运算数 1 做被减数，运算数 2 做减数，进行简单相减运算并将最后结果存储在数据寄存器 DATA 中。若运算数 1 小于运算数 2，则运算数 2 做被减数，运算数 1 做减数，进行简单减法运算并将最后结果存储在数据寄存器 DATA 中，并将最高位置 1，表明该寄存器中存储的是负数对应的绝对值。

若运算为开方运算，利用整数平方根算法实现。

（三）运算结果处理电路

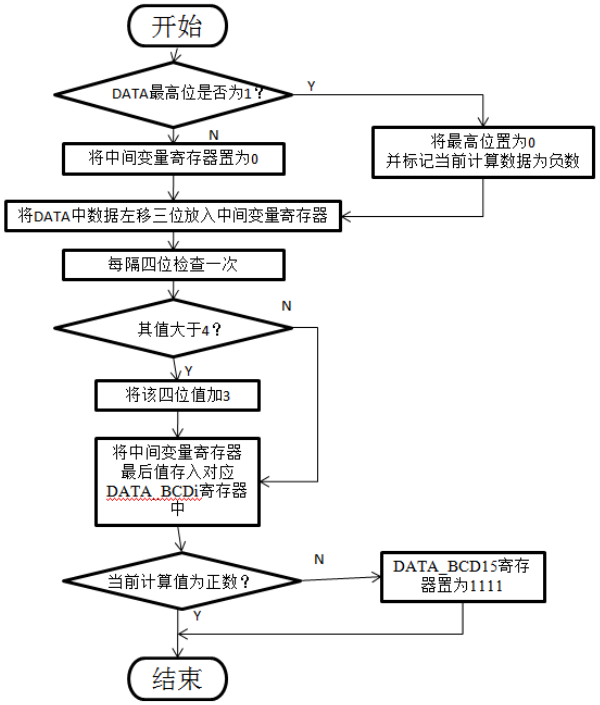
本模块完成的主要功能是将存储在寄存器中的十六进制运算结果转换成十进制结果输出；由于存储在寄存器中的十六进制数实际上是二进制数，故本模块的核心思想在于将二进制数转换成 BCD 码。

本模块采用“移位加三算法”实现。算数运算电路 U2 模块向本模块输入一个 54 位二进制数。首先将二进制数左移一位，将移位后结果存储在中间变量寄存器中。每次移位后，都要对中间变量寄存器中的内容进行检查。从第 54 位起，每四位检查一次。如果其值大于 4，就将其该四位值加 3；否则值不变。

因为只有移位至少三次后才会出现值大于 4 的情况。故首先对输入信号移位三位，之后每次循环移位一次。每移位一次都要按照上述内容进行检查修正。总共移位 54 次后，结束循环。最后，将中间寄存器中从第 54 位起，每四位分别赋值给输入 BCD 码寄存器中（DATA_BCD0~DATA_BCD15）。结束运算结果处理。

值得注意的一点，因为本次运算的结果既有正数又有负数。故在正数和负数的 BCD 码转换中要分别处理。按照题目设计规则，在转换前首先对 DATA[53]位进行检查，如果为零证明是正数，直接按照上述步骤处理即可；如果为 1，证明是负数，则首先将该位清零，并标记此时被转换数为负数。因为 DATA 中存储的是负数的绝对值，故对结果直接进行上文所述转换即可。最后按照题目规则，将 DATA_BCD15 中的每一位置为 1，代表此时的转换结果是负数。

以上算法的具体计算流程如下所示：



发挥部分

（一）防抖动模块

按键开关在闭合时不会马上稳定接通，在断开时也不会立刻断开。因而在闭合及断开的瞬间均伴随有一连串的抖动，导致按键输入不稳定。本模块设计的目的就是为了消除这种抖动，保证运算准确进行。

本模块的主要思想是：如果数据输入处理电路模块采用边沿触发，在输入有抖动的时候，相当于 KEY 相应位输出一段时间短且高频的脉冲波形。这样每来一个有效边沿就触发一次转换，相当于只按下一次按键，但数据输入电路却读入多个重复的数据，造成错误。

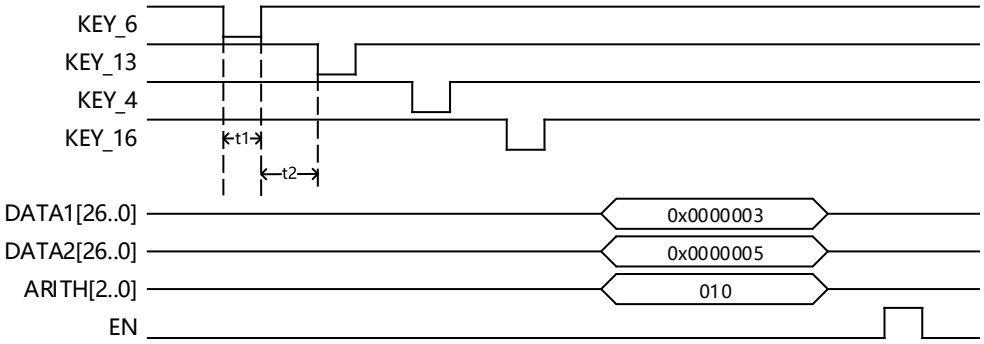
如果该边沿触发方式为查询方式，间隔一定时间对 KEY 对应位波形进行查询，避开多次边沿触发造成的误差，达到消除抖动的作用

（二）为除法运算和开方运算增加余数输出模块

Verilog 自带了求余运算符%，因此只要再加一路输出，同时输出余数和商即可。开方结果的余数为用原始数据减去开方计算结果的平方得到。

三、时序说明

本次设计总体时序图如下图所示：



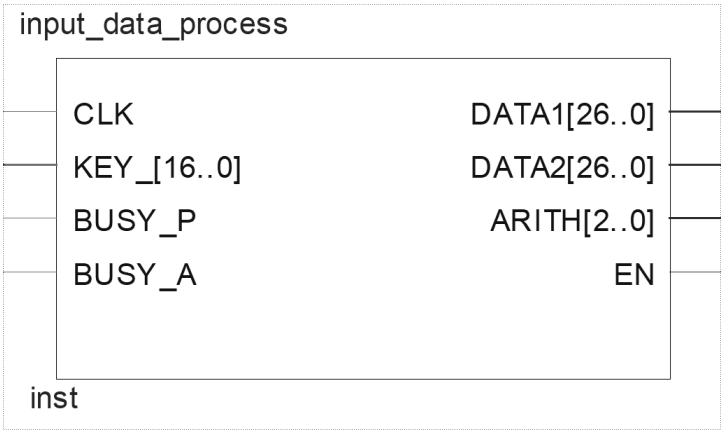
本次电路的时序大致如上图所示。在接下来的仿真结果中可以看到仿真时序符合设计时序要求。

四、模块设计框图、引脚说明、相关时序

基础部分

（一）数据输入处理模块：

首先，数据输入处理模块的引脚框图如下所示：



图：数据输入处理模块框图

详细引脚说明：

CLK：时钟输入

KEY_[16:0]：独立按键输入

BUSY_P：运算结果处理电路反馈信号

BUSY_A：算数运算电路反馈信号

DATA1[26:0]：第一个运算数输出

DATA2[26:0]：第二个运算数输出

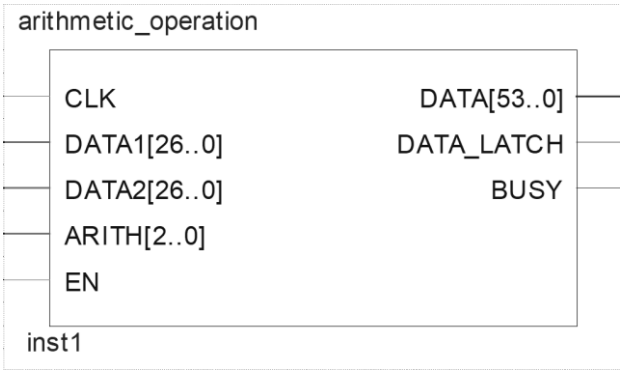
ARITH[2:0]：运算控制（表明当前运算符号）

EN：数据使能信号

其中，BUSY_P、BUSY_A 及 EN 都是高电平有效

（二）算数运算模块

首先，算数运算模块的引脚框图如下所示：



图：算术运算模块框图

详细引脚说明：

CLK：时钟输入

DATA1[26:0]：第一个运算数输入

DATA2[26:0]：第二个运算数输入

ARITH[2:0]：运算控制（表明当前运算符号）

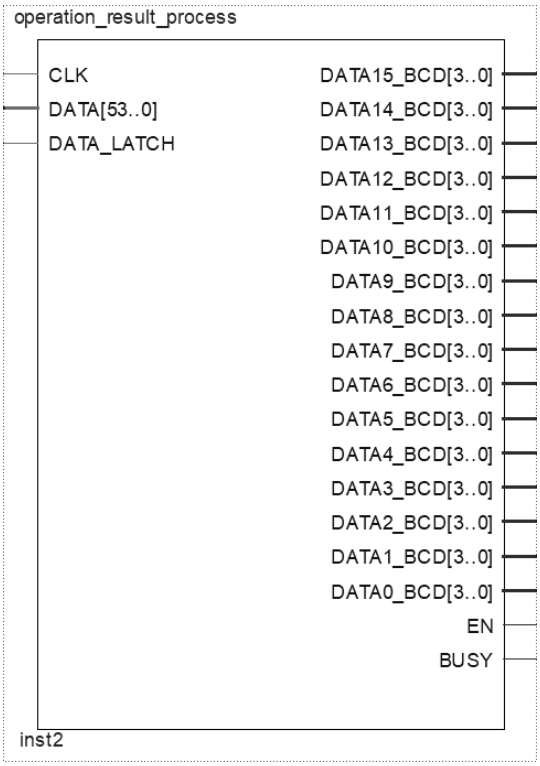
EN：数据使能信号输入

DATA[53..0]为运算结果输出

DATA_LATCH 为数据锁存信号（上升沿锁存）

BUSY 为工作状态反馈信号（高电平有效）

（三）运算结果处理模块：



图：运算结果处理模块框图

详细引脚说明：

CLK：时钟输入

DATA[53:0]：运算结果输入

DATA_LATCH：锁存信号（上升沿锁存）

DATA_BCD15[3:0]~DATA_BCD0[3:0]：16 位 BDC 码输出

EN：使能信号输出

BUSY：工作状态反馈信号

其中，EN 和 BUSY 都是高电平有效的信号

发挥部分

U2 增加的 MOD 引脚为余数的输出引脚，U3 增加的 MOD 为 U2 结果中余数的输入引脚，MOD_BCD 为余数部分的 BCD 输出引脚。

五、代码及必要注释

基本要求

数据输入处理电路代码：

```
`timescale 1ns/1ns

module input_data_process(
input wire CLK,
```

```
input wire [16:0] KEY,
input wire BUSY_P,
input wire BUSY_A,
output reg [26:0] DATA_1,
output reg [26:0] DATA_2,
output reg [2:0] ARITH,
output reg EN
);

parameter ADD = 3'b000;
parameter SUB = 3'b001;
parameter MUL = 3'b010;
parameter DIV = 3'b011;
parameter SQRT = 3'b100;                    //对基本运算符的定义

reg [16:0] temp1;
reg [16:0] temp2;
reg [16:0] result;

reg second_flag;
reg sqrt_flag;

initial
begin
      DATA_1 = 0;
       DATA_2 = 0;
       second_flag = 1'b0;
end


always @(posedge CLK)
begin
      if(!(BUSY_A || BUSY_P))                  //反馈工作信号都有效，可以进行按键输入
       begin

      EN = 1'b0;
       case(KEY)
            17'b11111110111111111: //0
             begin
                  if(second_flag == 1'b0)
                  DATA_1 = DATA_1 * 10;
                     else
                     DATA_2 = DATA_2 * 10;
             end
```

```
17'b11111111011111111: //1
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 1;
      else
      DATA_2 = DATA_2 * 10 + 1;
end
17'b11111111101111111: //2
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 2;
      else
      DATA_2 = DATA_2 * 10 + 2;
end
17'b11111111110111111: //3
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 3;
      else
      DATA_2 = DATA_2 * 10 + 3;
end
17'b11111111111011111: //4
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 4;
      else
      DATA_2 = DATA_2 * 10 + 4;
end
17'b11111111111101111: //5
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 5;
      else
      DATA_2 = DATA_2 * 10 + 5;
end
17'b11111111111110111: //6
begin
    if(second_flag == 1'b0)
    DATA_1 = DATA_1 * 10 + 6;
      else
      DATA_2 = DATA_2 * 10 + 6;
end
17'b11111111111111011: //7
begin
```

```
        if(second_flag == 1'b0)
        DATA_1 = DATA_1 * 10 + 7;
          else
          DATA_2 = DATA_2 * 10 + 7;
end
17'b11111111111111101: //8
begin
        if(second_flag == 1'b0)
        DATA_1 = DATA_1 * 10 + 8;
          else
          DATA_2 = DATA_2 * 10 + 8;
end
17'b11111111111111110: //9
begin
        if(second_flag == 1'b0)
        DATA_1 = DATA_1 * 10 + 9;
          else
          DATA_2 = DATA_2 * 10 + 9;
end
17'b11111110111111111: //AC
begin
        DATA_1 = 0;
          DATA_2 = 0;
          ARITH = 0;
          second_flag = 1'b0;
end
17'b10111111111111111: //ADD
begin
        second_flag = 1'b1;
          ARITH = ADD;
end
17'b11011111111111111: //SUB
begin
        second_flag = 1'b1;
          ARITH = SUB;
end
17'b11101111111111111: //MUL
begin
        second_flag = 1'b1;
          ARITH = MUL;
end
17'b11110111111111111: //DIV
begin
        second_flag = 1'b1;
```

```
                    ARITH = DIV;
                end
                17'b11111011111111111: //SQRT
                begin
                    sqrt_flag = 1'b1;
                    ARITH = SQRT;
                end
                17'b01111111111111111: //EQU
               begin
                    EN = 1'b1;
                end
                default:;
        endcase
         end
end
endmodule
```

算数运算电路代码：

```
module arithmetic_operation(
input wire CLK,
input wire [26:0] DATA1,
input wire [26:0] DATA2,
input wire [2:0] ARITH,
input wire EN,
output reg [53:0] DATA,
output reg DATA_LATCH,
output reg BUSY
);

parameter ADD = 3'b000;
parameter SUB = 3'b001;
parameter MUL = 3'b010;
parameter DIV = 3'b011;
parameter SQRT = 3'b100;

reg [13:0] square;
reg [27:0] delta;
reg sqrt_first_flag;
reg sqrt_run_flag;
reg sqrt_finish_flag;

initial
begin
        square = 1;
```

```
        delta = 3;
        sqrt_finish_flag = 1'b0;
        sqrt_run_flag = 1'b0;
end

always @(posedge CLK)
begin
      if(EN)
      begin
        BUSY = 1'b1;
        DATA_LATCH = 1'b0;


        case(ARITH)
              ADD:
               begin
                    DATA = DATA1 + DATA2;
                       sqrt_finish_flag = 1'b1;
               end
               SUB:
               begin
                    if(DATA1 >= DATA2)
                            DATA = DATA1 - DATA2;
                       else
                       begin
                            DATA = DATA2 - DATA1;
                             DATA[53] = 1;
                       end
                       sqrt_finish_flag = 1'b1;
               end
               MUL:
               begin
                    DATA = DATA1 * DATA2;
                       sqrt_finish_flag = 1'b1;
               end
               DIV:
               begin
                    DATA = DATA1 / DATA2;
                       sqrt_finish_flag = 1'b1;
               end
              SQRT:
               begin
                    sqrt_run_flag = 1'b1;
```

```
                    if((square > DATA1) && sqrt_run_flag)
                       begin
                            DATA = delta[27:1] - 1;
                             sqrt_finish_flag = 1'b1;
                             sqrt_run_flag = 0;
                       end
                       else if(sqrt_run_flag)
                       begin
                            square = square + delta;
                             delta = delta + 2;
                       end

                end
        endcase

        if(sqrt_finish_flag)
        begin
              BUSY = 1'b0;
              DATA_LATCH = 1'b1;
        end
        end
end
endmodule
```

运算结果处理电路代码：

```
`timescale 1ns/1ns

module operation_result_process(
input wire CLK,
input wire [53:0] DATA,
input wire DATA_LATCH,
output reg    [3:0] DATA_BCD0,
output reg    [3:0] DATA_BCD1,
output reg    [3:0] DATA_BCD2,
output reg    [3:0] DATA_BCD3,
output reg    [3:0] DATA_BCD4,
output reg    [3:0] DATA_BCD5,
output reg    [3:0] DATA_BCD6,
output reg    [3:0] DATA_BCD7,
output reg    [3:0] DATA_BCD8,
output reg    [3:0] DATA_BCD9,
output reg    [3:0] DATA_BCD10,
output reg    [3:0] DATA_BCD11,
```

```
output reg    [3:0] DATA_BCD12,
output reg    [3:0] DATA_BCD13,
output reg    [3:0] DATA_BCD14,
output reg    [3:0] DATA_BCD15,
output reg EN,
output reg BUSY
);

//中间变量
reg [117:0]z;
reg flag;
integer i;

initial
begin
      flag = 0;
end

always@(posedge DATA_LATCH)
begin
      EN = 1'b0;
       BUSY = 1'b1;
      for(i=0;i<=117;i=i+1)
      z[i]=0;

       if(DATA[53])
       begin

              flag = 1;
              z[55:3] = DATA[52:0];
       end
       else
       begin
            z[56:3]=DATA;
      end

      repeat(51)
      begin
           if(z[57:54]>4)
             z[57:54]=z[57:54]+3;
             if(z[61:58]>4)
             z[61:58]=z[61:58]+3;
             if(z[65:62]>4)
             z[65:62]=z[65:62]+3;
```

```
        if(z[69:66]>4)
          z[69:66]=z[69:66]+3;
          if(z[73:70]>4)
          z[73:70]=z[73:70]+3;
          if(z[77:74]>4)
          z[77:74]=z[77:74]+3;
          if(z[81:78]>4)
          z[81:78]=z[81:78]+3;
          if(z[85:82]>4)
          z[85:82]=z[85:82]+3;
          if(z[89:86]>4)
          z[89:86]=z[89:86]+3;
          if(z[93:90]>4)
          z[93:90]=z[93:90]+3;
          if(z[97:94]>4)
          z[97:94]=z[97:94]+3;
          if(z[101:98]>4)
          z[101:98]=z[101:98]+3;
          if(z[105:102]>4)
          z[105:102]=z[105:102]+3;
          if(z[109:106]>4)
          z[109:106]=z[109:106]+3;
          if(z[113:110]>4)
          z[113:110]=z[113:110]+3;
          if(z[117:114]>4)
          z[117:114]=z[117:114]+3;

          z[117:1]=z[116:0];

    end

     DATA_BCD0=z[57:54];
     DATA_BCD1=z[61:58];
     DATA_BCD2=z[65:62];
     DATA_BCD3=z[69:66];
     DATA_BCD4=z[73:70];
     DATA_BCD5=z[77:74];
     DATA_BCD6=z[81:78];
     DATA_BCD7=z[85:82];
     DATA_BCD8=z[89:86];
     DATA_BCD9=z[93:90];
     DATA_BCD10=z[97:94];
     DATA_BCD11=z[101:98];
     DATA_BCD12=z[105:102];
```

```
        DATA_BCD13=z[109:106];
        DATA_BCD14=z[113:110];
        if(!flag)
        DATA_BCD15=z[117:114];
        else
        DATA_BCD15=4'b1111;

        BUSY = 1'b0;
        EN = 1'b1;
        #10 EN = 1'b0;
        end
endmodule
```

发挥部分

防抖动部分代码展示

```
`timescale 1ms/1ms

module input_data_process(
input wire CLK,
input wire [16:0] KEY,
input wire BUSY_P,
input wire BUSY_A,
output reg [26:0] DATA_1,
output reg [26:0] DATA_2,
output reg [2:0] ARITH,
output reg EN
);

parameter ADD = 3'b000;
parameter SUB = 3'b001;
parameter MUL = 3'b010;
parameter DIV = 3'b011;
parameter SQRT = 3'b100;

reg [16:0] temp1;
reg [16:0] temp2;
reg [16:0] result;

reg second_flag;
reg sqrt_flag;

initial
begin
      DATA_1 = 0;
```

```
        DATA_2 = 0;
        second_flag = 1'b0;
end


always @(posedge CLK)
begin
       if(!(BUSY_A || BUSY_P))
        begin

        temp1 = KEY;
        #150 temp2 = KEY;
        if(temp1 == temp2)
        result = temp2;
        else
        result = 17'b11111111111111111;

       EN = 1'b0;
        case(result)
              17'b11111110111111111: //0
               begin
                     if(second_flag == 1'b0)
                     DATA_1 = DATA_1 * 10;
                        else
                        DATA_2 = DATA_2 * 10;
               end
               17'b11111111011111111: //1
               begin
                     if(second_flag == 1'b0)
                     DATA_1 = DATA_1 * 10 + 1;
                        else
                        DATA_2 = DATA_2 * 10 + 1;
               end
               17'b11111111101111111: //2
               begin
                     if(second_flag == 1'b0)
                     DATA_1 = DATA_1 * 10 + 2;
                        else
                        DATA_2 = DATA_2 * 10 + 2;
               end
               17'b11111111110111111: //3
               begin
                     if(second_flag == 1'b0)
                     DATA_1 = DATA_1 * 10 + 3;
```

```
            else
            DATA_2 = DATA_2 * 10 + 3;
        end
        17'b11111111111011111: //4
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 4;
            else
            DATA_2 = DATA_2 * 10 + 4;
        end
        17'b11111111111101111: //5
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 5;
            else
            DATA_2 = DATA_2 * 10 + 5;
        end
        17'b11111111111110111: //6
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 6;
            else
            DATA_2 = DATA_2 * 10 + 6;
        end
        17'b11111111111111011: //7
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 7;
            else
            DATA_2 = DATA_2 * 10 + 7;
        end
        17'b11111111111111101: //8
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 8;
            else
            DATA_2 = DATA_2 * 10 + 8;
        end
        17'b11111111111111110: //9
        begin
            if(second_flag == 1'b0)
            DATA_1 = DATA_1 * 10 + 9;
            else
            DATA_2 = DATA_2 * 10 + 9;
```

```
            end
            17'b11111110111111111: //AC
            begin
                DATA_1 = 0;
                  DATA_2 = 0;
                  ARITH = 0;
                  second_flag = 1'b0;
            end
            17'b10111111111111111: //ADD
            begin
                second_flag = 1'b1;
                  ARITH = ADD;
            end
            17'b11011111111111111: //SUB
            begin
                second_flag = 1'b1;
                  ARITH = SUB;
            end
            17'b11101111111111111: //MUL
            begin
                second_flag = 1'b1;
                  ARITH = MUL;
            end
            17'b11110111111111111: //DIV
            begin
                second_flag = 1'b1;
                  ARITH = DIV;
            end
            17'b11111011111111111: //SQRT
            begin
                sqrt_flag = 1'b1;
                  ARITH = SQRT;
            end
            17'b01111111111111111: //EQU
           begin
                EN = 1'b1;
            end
            default:;
      endcase
       end
end
endmodule
```

为除法运算和开、开方运算增加余数输出功能

U2 模块求余程序

```
module arithmetic_operation(
input wire CLK,
input wire [26:0] DATA1,
input wire [26:0] DATA2,
input wire [2:0] ARITH,
input wire EN,
output reg [53:0] DATA,
output reg [53:0] MOD,
output reg DATA_LATCH,
output reg BUSY
);

parameter ADD = 3'b000;
parameter SUB = 3'b001;
parameter MUL = 3'b010;
parameter DIV = 3'b011;
parameter SQRT = 3'b100;

reg [13:0] square;
reg [27:0] delta;
reg sqrt_first_flag;
reg sqrt_run_flag;
reg sqrt_finish_flag;

initial
begin
        square = 1;
        delta = 3;
        sqrt_finish_flag = 1'b0;
        sqrt_run_flag = 1'b0;
end

always @(posedge CLK)
begin
      if(EN)
      begin
        BUSY = 1'b1;
        DATA_LATCH = 1'b0;


        case(ARITH)
              ADD:
                begin
                      DATA = DATA1 + DATA2;
```

```
                sqrt_finish_flag = 1'b1;
        end
        SUB:
        begin
            if(DATA1 >= DATA2)
                DATA = DATA1 - DATA2;
            else
            begin
                DATA = DATA2 - DATA1;
                 DATA[53] = 1;
            end
            sqrt_finish_flag = 1'b1;
        end
        MUL:
        begin
            DATA = DATA1 * DATA2;
            sqrt_finish_flag = 1'b1;
        end
        DIV:
        begin
            DATA = DATA1 / DATA2;
            MOD = DATA1 % DATA2;
            sqrt_finish_flag = 1'b1;
        end
       SQRT:
        begin
            sqrt_run_flag = 1'b1;

            if((square > DATA1) && sqrt_run_flag)
            begin
                DATA = delta[27:1] - 1;
                 MOD = DATA1 - DATA * DATA;
                 sqrt_finish_flag = 1'b1;
                 sqrt_run_flag = 0;
            end
            else if(sqrt_run_flag)
            begin
                square = square + delta;
                 delta = delta + 2;
            end
        end
    endcase

    if(sqrt_finish_flag)
```

```
        begin
            BUSY = 1'b0;
            DATA_LATCH = 1'b1;
        end
        end
end

endmodule
```

U3 模块显示程序

```
module operation_result_process(
input wire CLK,
input wire [53:0] DATA,
input wire [53:0] MOD,
input wire DATA_LATCH,
output reg    [3:0] DATA_BCD0,
output reg    [3:0] DATA_BCD1,
output reg    [3:0] DATA_BCD2,
output reg    [3:0] DATA_BCD3,
output reg    [3:0] DATA_BCD4,
output reg    [3:0] DATA_BCD5,
output reg    [3:0] DATA_BCD6,
output reg    [3:0] DATA_BCD7,
output reg    [3:0] DATA_BCD8,
output reg    [3:0] DATA_BCD9,
output reg    [3:0] DATA_BCD10,
output reg    [3:0] DATA_BCD11,
output reg    [3:0] DATA_BCD12,
output reg    [3:0] DATA_BCD13,
output reg    [3:0] DATA_BCD14,
output reg    [3:0] DATA_BCD15,

output reg    [3:0] MOD_BCD0,
output reg    [3:0] MOD_BCD1,
output reg    [3:0] MOD_BCD2,
output reg    [3:0] MOD_BCD3,
output reg    [3:0] MOD_BCD4,
output reg    [3:0] MOD_BCD5,
output reg    [3:0] MOD_BCD6,
output reg    [3:0] MOD_BCD7,
output reg    [3:0] MOD_BCD8,
output reg    [3:0] MOD_BCD9,
output reg    [3:0] MOD_BCD10,
output reg    [3:0] MOD_BCD11,
```

```
output reg    [3:0] MOD_BCD12,
output reg    [3:0] MOD_BCD13,
output reg    [3:0] MOD_BCD14,
output reg    [3:0] MOD_BCD15,
output reg EN,
output reg BUSY
);

//中间变量
reg [117:0] z;
reg [117:0] z1;
reg flag;
integer i;

initial
begin
      flag = 0;
end

always@(posedge DATA_LATCH)
begin
      EN = 1'b0;
       BUSY = 1'b1;
      for(i=0;i<=117;i=i+1)
       begin
      z[i]=0;
       z1[i]=0;
       end

       if(DATA[53])
       begin
              flag = 1;
              z[55:3] = DATA[52:0];
       end
       else
       begin
            z[56:3]=DATA;
      end
       z1[56:3] = MOD;

      repeat(51)
      begin
           if(z[57:54]>4)
            z[57:54]=z[57:54]+3;
```

```
  if(z[61:58]>4)
  z[61:58]=z[61:58]+3;
  if(z[65:62]>4)
  z[65:62]=z[65:62]+3;
 if(z[69:66]>4)
  z[69:66]=z[69:66]+3;
  if(z[73:70]>4)
  z[73:70]=z[73:70]+3;
  if(z[77:74]>4)
  z[77:74]=z[77:74]+3;
  if(z[81:78]>4)
  z[81:78]=z[81:78]+3;
  if(z[85:82]>4)
  z[85:82]=z[85:82]+3;
  if(z[89:86]>4)
  z[89:86]=z[89:86]+3;
  if(z[93:90]>4)
  z[93:90]=z[93:90]+3;
  if(z[97:94]>4)
  z[97:94]=z[97:94]+3;
  if(z[101:98]>4)
  z[101:98]=z[101:98]+3;
  if(z[105:102]>4)
  z[105:102]=z[105:102]+3;
  if(z[109:106]>4)
  z[109:106]=z[109:106]+3;
  if(z[113:110]>4)
  z[113:110]=z[113:110]+3;
  if(z[117:114]>4)
  z[117:114]=z[117:114]+3;
  z[117:1]=z[116:0];

    if(z1[57:54]>4)
  z1[57:54]=z1[57:54]+3;
  if(z1[61:58]>4)
  z1[61:58]=z1[61:58]+3;
  if(z1[65:62]>4)
  z1[65:62]=z1[65:62]+3;
 if(z1[69:66]>4)
  z1[69:66]=z1[69:66]+3;
  if(z1[73:70]>4)
  z1[73:70]=z1[73:70]+3;
  if(z1[77:74]>4)
  z1[77:74]=z1[77:74]+3;
```

```
        if(z1[81:78]>4)
        z1[81:78]=z1[81:78]+3;
        if(z1[85:82]>4)
        z1[85:82]=z1[85:82]+3;
        if(z1[89:86]>4)
        z1[89:86]=z1[89:86]+3;
        if(z1[93:90]>4)
        z1[93:90]=z1[93:90]+3;
        if(z1[97:94]>4)
        z1[97:94]=z1[97:94]+3;
        if(z1[101:98]>4)
        z1[101:98]=z1[101:98]+3;
        if(z1[105:102]>4)
        z1[105:102]=z1[105:102]+3;
        if(z1[109:106]>4)
        z1[109:106]=z1[109:106]+3;
        if(z1[113:110]>4)
        z1[113:110]=z1[113:110]+3;
        if(z1[117:114]>4)
        z1[117:114]=z1[117:114]+3;
        z1[117:1]=z1[116:0];



end

  DATA_BCD0=z[57:54];
  DATA_BCD1=z[61:58];
  DATA_BCD2=z[65:62];
  DATA_BCD3=z[69:66];
  DATA_BCD4=z[73:70];
  DATA_BCD5=z[77:74];
  DATA_BCD6=z[81:78];
  DATA_BCD7=z[85:82];
  DATA_BCD8=z[89:86];
  DATA_BCD9=z[93:90];
  DATA_BCD10=z[97:94];
  DATA_BCD11=z[101:98];
  DATA_BCD12=z[105:102];
  DATA_BCD13=z[109:106];
  DATA_BCD14=z[113:110];
  if(!flag)
  DATA_BCD15=z[117:114];
  else
```

```
        DATA_BCD15=4'b1111;

        MOD_BCD0=z1[57:54];
        MOD_BCD1=z1[61:58];
        MOD_BCD2=z1[65:62];
        MOD_BCD3=z1[69:66];
        MOD_BCD4=z1[73:70];
        MOD_BCD5=z1[77:74];
        MOD_BCD6=z1[81:78];
        MOD_BCD7=z1[85:82];
        MOD_BCD8=z1[89:86];
        MOD_BCD9=z1[93:90];
        MOD_BCD10=z1[97:94];
        MOD_BCD11=z1[101:98];
        MOD_BCD12=z1[105:102];
        MOD_BCD13=z1[109:106];
        MOD_BCD14=z1[113:110];
        MOD_BCD15=z1[117:114];




        BUSY = 1'b0;
        EN = 1'b1;
        #10 EN = 1'b0;
        end
endmodule
```

六、仿真结果

基本要求

（一）数据输入处理电路（U1）

这一部分完成的功能主要是将按键输入的数据转化成实际存储在寄存器中的、用于实际计算的二进制数据。并将运算符转换成便于后续程序识别定义的二进制数据。这里为了验证该部分电路的正确性，我们选择了两个测试用例。

在本模块程序的设计中，我们对运算符的定义如下所示：

运算符	对应二进制数
ADD（+）	000
SUB（—）	001

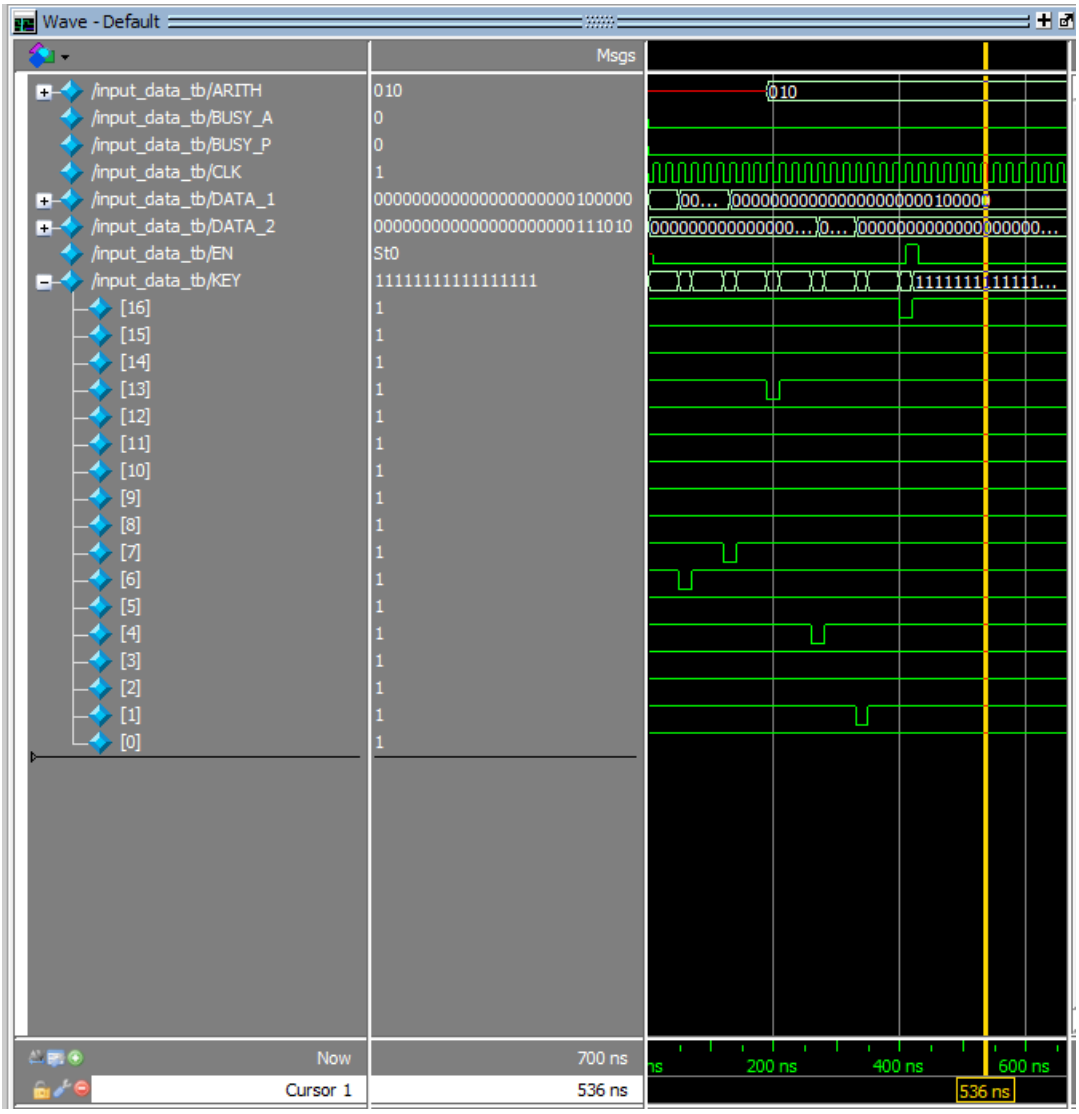
MUL（×）	010
DIV（÷）	011
SQRT（开方）	100

测试用例一

计算式：32×58

输入数据/运算符	对应二进制数
32	100000
×	010
58	111010

结果：



在 testbench 中，设计第一个输入数据为 32，运算符为×，第二个输入数据为 58.这里的设计（激励波形）由 KEY 实现。

观察上图仿真波形中 DATA_1、DATA_2 及 ARITH 中存储数据：

寄存器名	实际存储二进制数据	对应十进制数据
DATA_1	100000	32
DATA_2	111010	58
ARITH	010	X

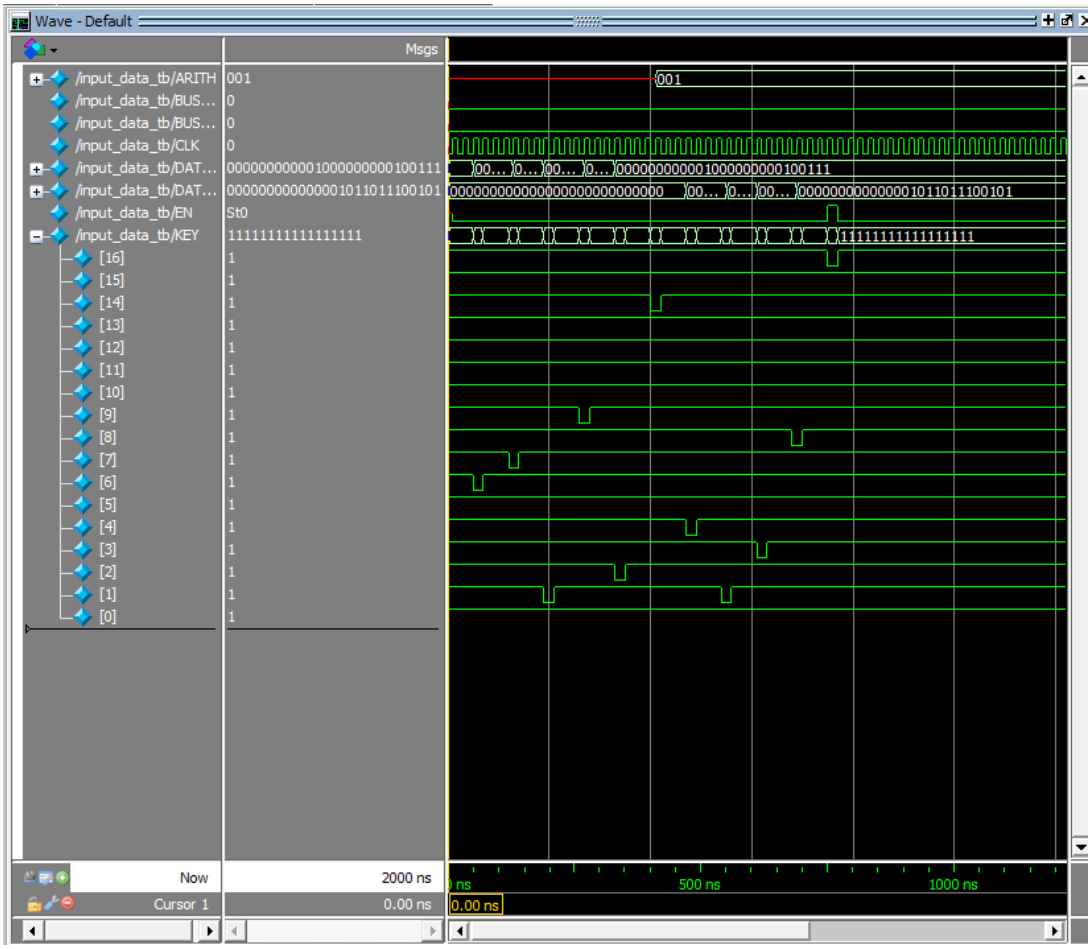
显然，仿真结果与预期一致。

测试用例二

计算式：32807-5861

输入数据/运算符	对应二进制数
32807	1000000000100111
—	001
5861	1011011100101

结果：



观察上图仿真波形中 DATA_1、DATA_2 及 ARITH 中存储数据：

寄存器名	实际存储二进制数据	对应十进制数据
DATA_1	1000000000100111	32807
DATA_2	1011011100101	5861
ARITH	001	—

显然，仿真结果与预期一致。

综上，两个测试用例的仿真结果都与预期一致，说明本模块（数据输入处理电路 U1）的程序编写达到预期效果。

（二）算数运算电路（U2）

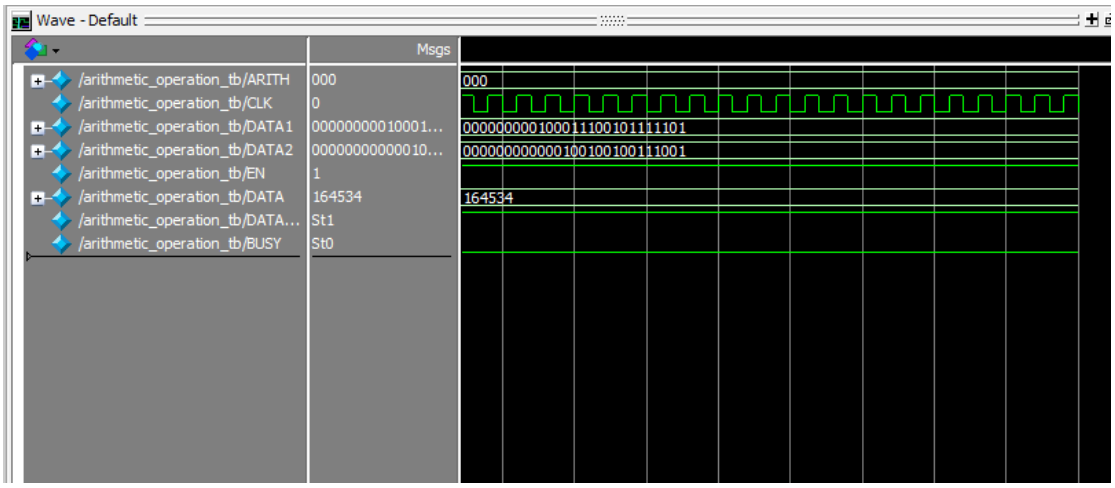
这一部分完成的功能主要是将实际存储在寄存器中的、用于实际计算的二进制数据，进行相应的加减乘除及开方运算，并将运算结果存储在寄存器 DATA 中。这里为了验证该部分电路的正确性，我们选择了五个测试用例，涵盖题目中要求的五个运算。

测试用例一

计算式：145789+18745

输入数据/运算符	对应二进制数
145789	100011100101111101
+	000
18745	100100100111001

结果：



观察上图仿真波形中 DATA 中存储数据：

十进制
164534

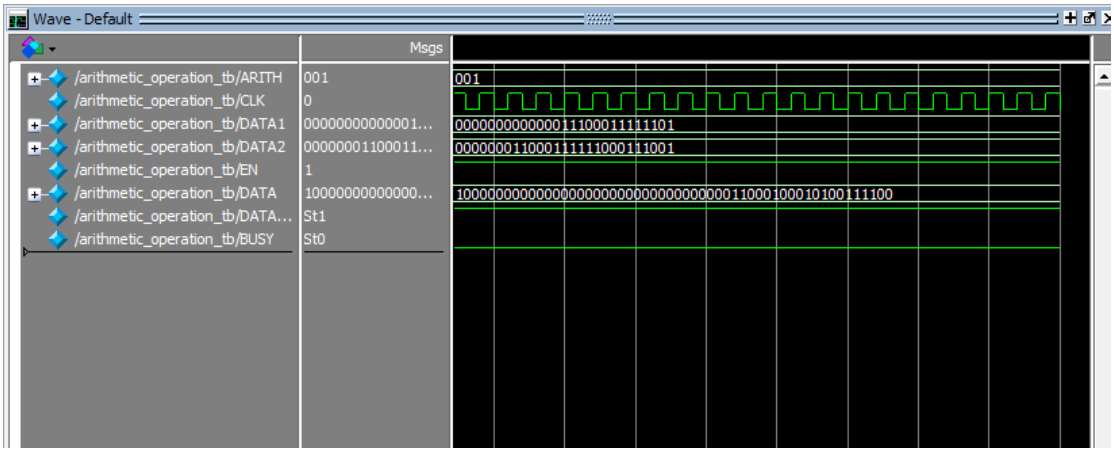
显然，145789+18745=164534，仿真结果与预期一致。

测试用例二

计算式：14589-818745

输入数据/运算符	对应二进制数
14589	11100011111101
—	001
818745	11000111111000111001

结果：



观察上图仿真波形中 DATA 中存储数据：

二进制
100…00110001000101001111100

其中，省略号部分均为 0，上述数据是一个 54 位的数据。最高位为 1.按照设计规则，最高位为 1 代表运算结果为负数，后面的结果代表真实运算结果的绝对值，其值对应十进制数为 804156.也即程序计算出的 14589-818745=—804156。

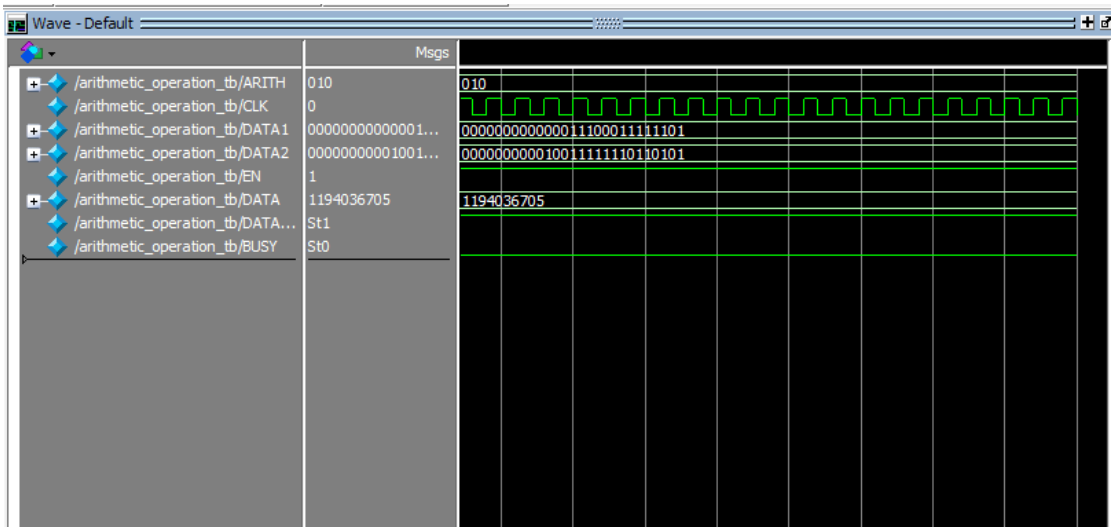
显然，仿真结果与预期一致。

测试用例三

计算式：14589×81845

输入数据/运算符	对应二进制数
14589	11100011111101
X	010
81845	10011111110110101

结果：



观察上图仿真波形中 DATA 中存储数据：

十进制
1194036705

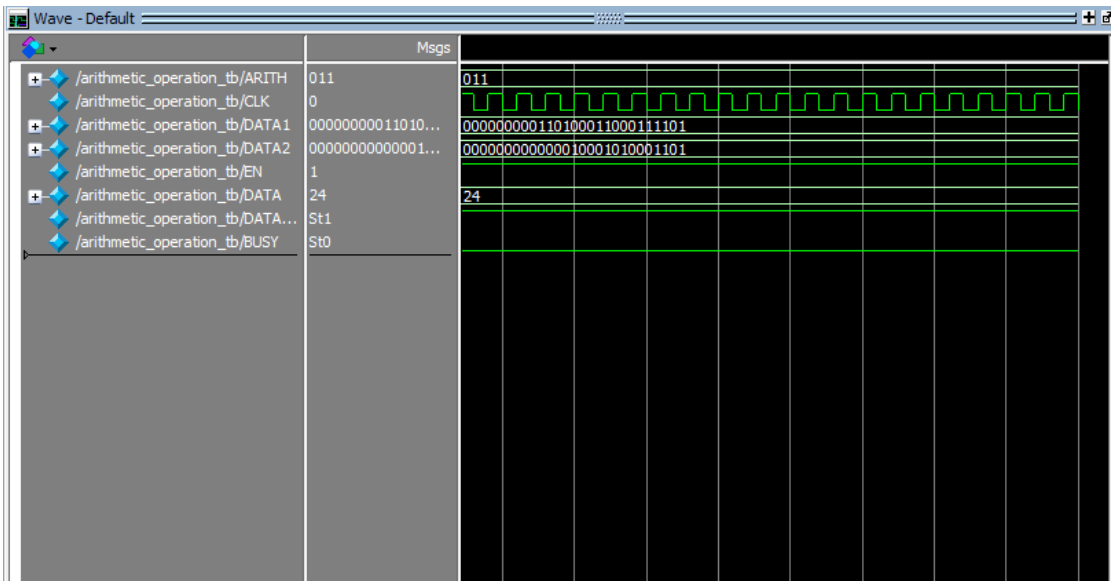
显然，14589×81845=1194036705，仿真结果与预期一致。

测试用例四

计算式：214589÷8845

输入数据/运算符	对应二进制数
214589	110100011000111101
÷	011
8845	10001010001101

结果：



观察上图仿真波形中 DATA 中存储数据：

十进制
24

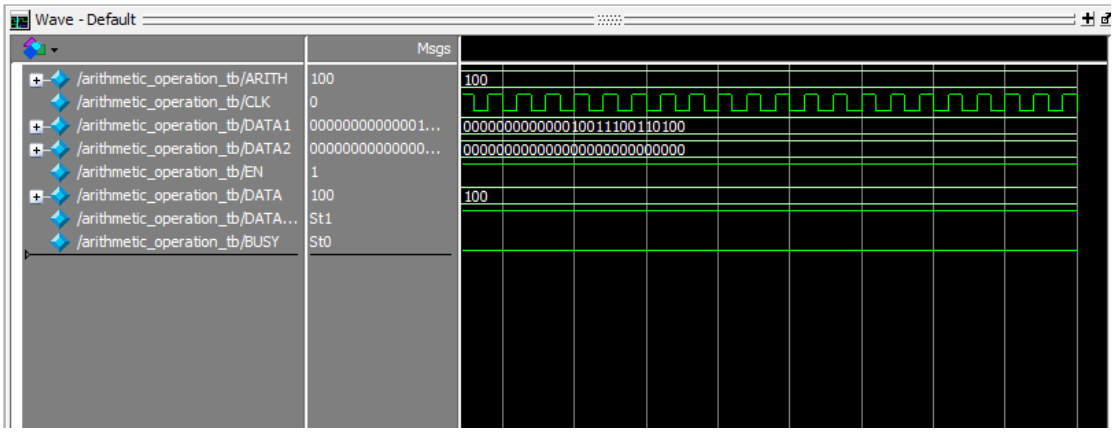
显然，214589÷8845=24，仿真结果与预期一致。

测试用例五

计算式：对 10036 开方

输入数据/运算符	对应二进制数
10036	10011100110100
—	100

结果：



观察上图仿真波形中 DATA 中存储数据：

十进制
100

显然，对 10036 开方值为 100 仿真结果与预期一致。

综上，五个测试用例的仿真结果都与预期一致，说明本模块（算数运算电路 U2）的程序编写达到预期效果。

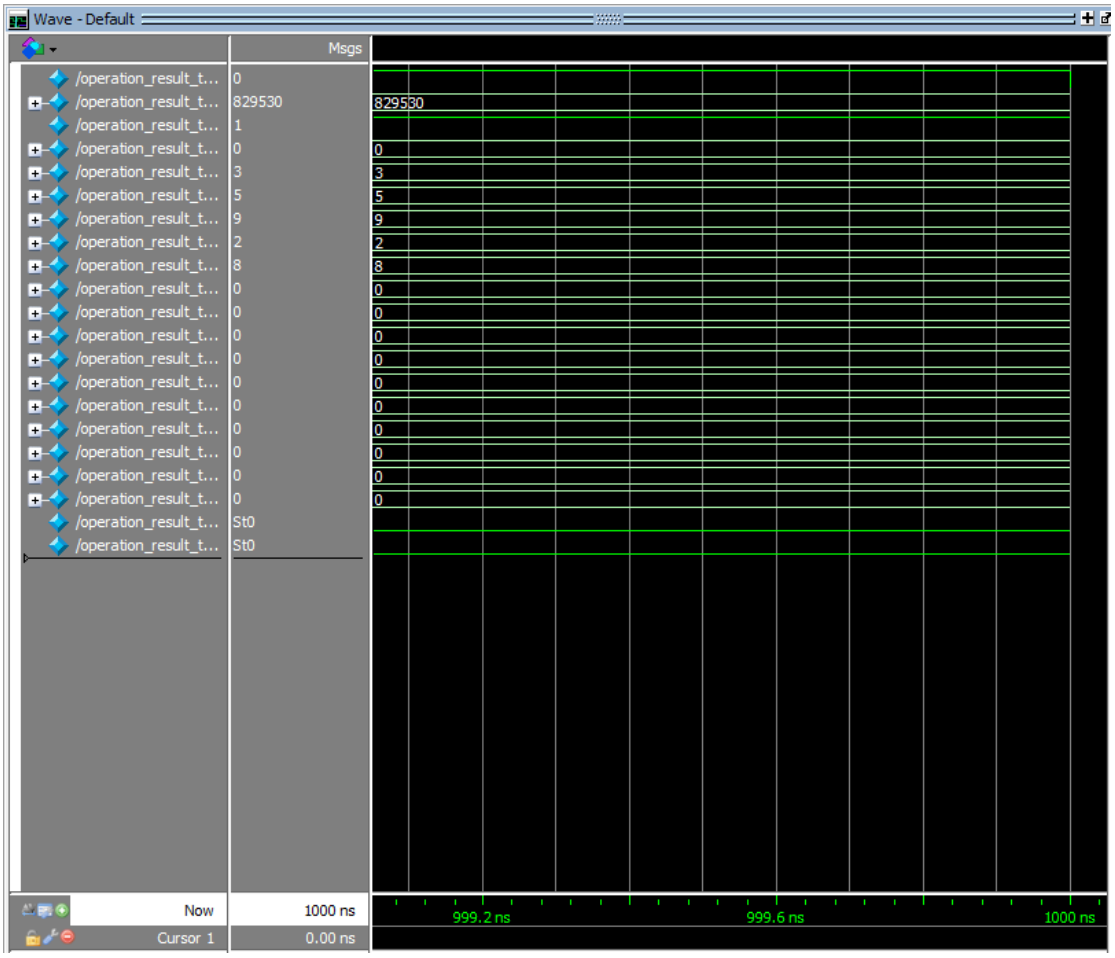
（三）运算结果处理电路（U3）

这一部分完成的功能主要是将实际存储在寄存器（DATA）最终计算结果的二进制数据，转换成相应的 BCD 码输出。这里为了验证该部分电路的正确性，我们选择了两个测试用例，涵盖正数与负数两种数 BCD 码的转换。

测试用例一

输入数据（二进制）	对应 BCD 码
11001010100001011010	0000000000829530

结果：

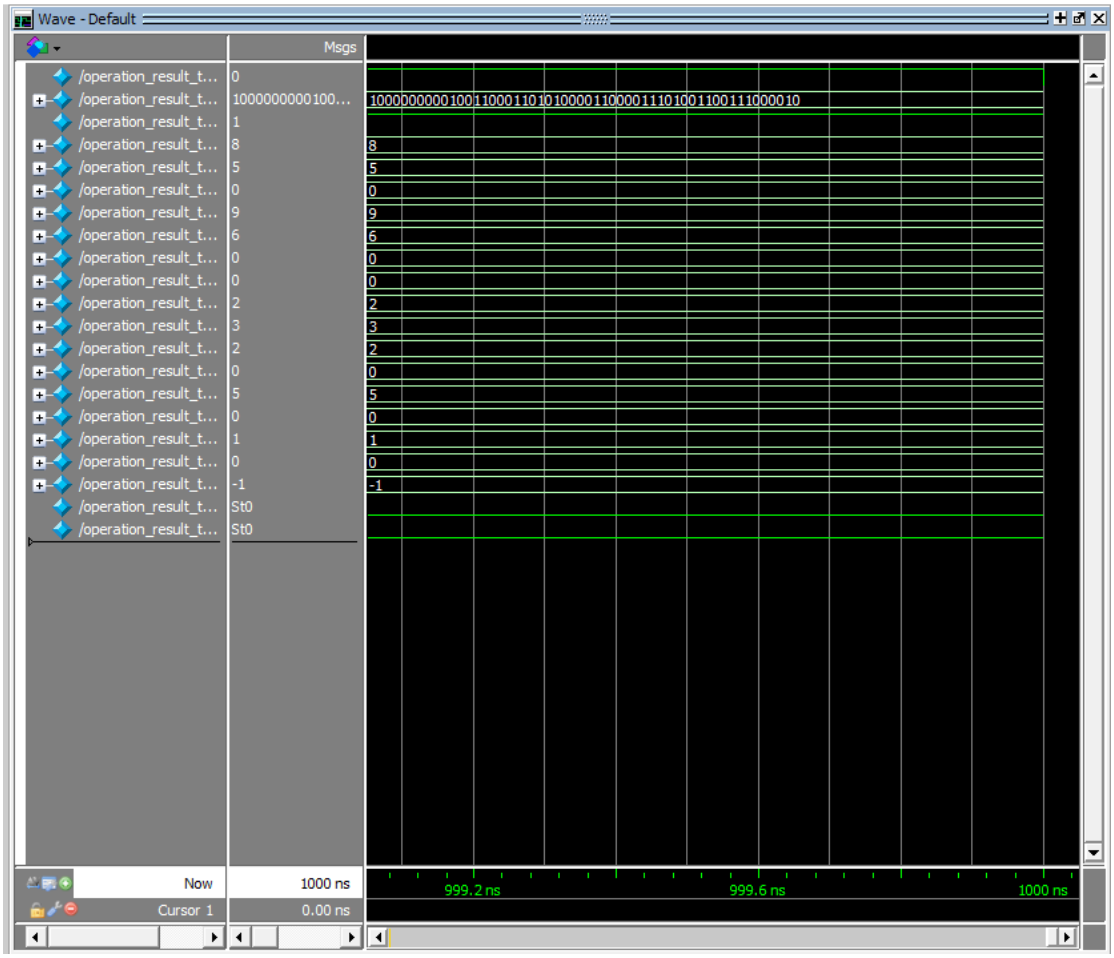


显然，仿真结果与预期一致。

测试用例二

输入数据（二进制）	对应 BCD 码
10…00111101010000001100101001111101011100001O	-0010502320069058

结果：



显然，仿真结果表明当输入是正数的时候，最高位寄存器 DATA_BCD15 值全为 1111，且后几位标识 BCD 码与预期一致。

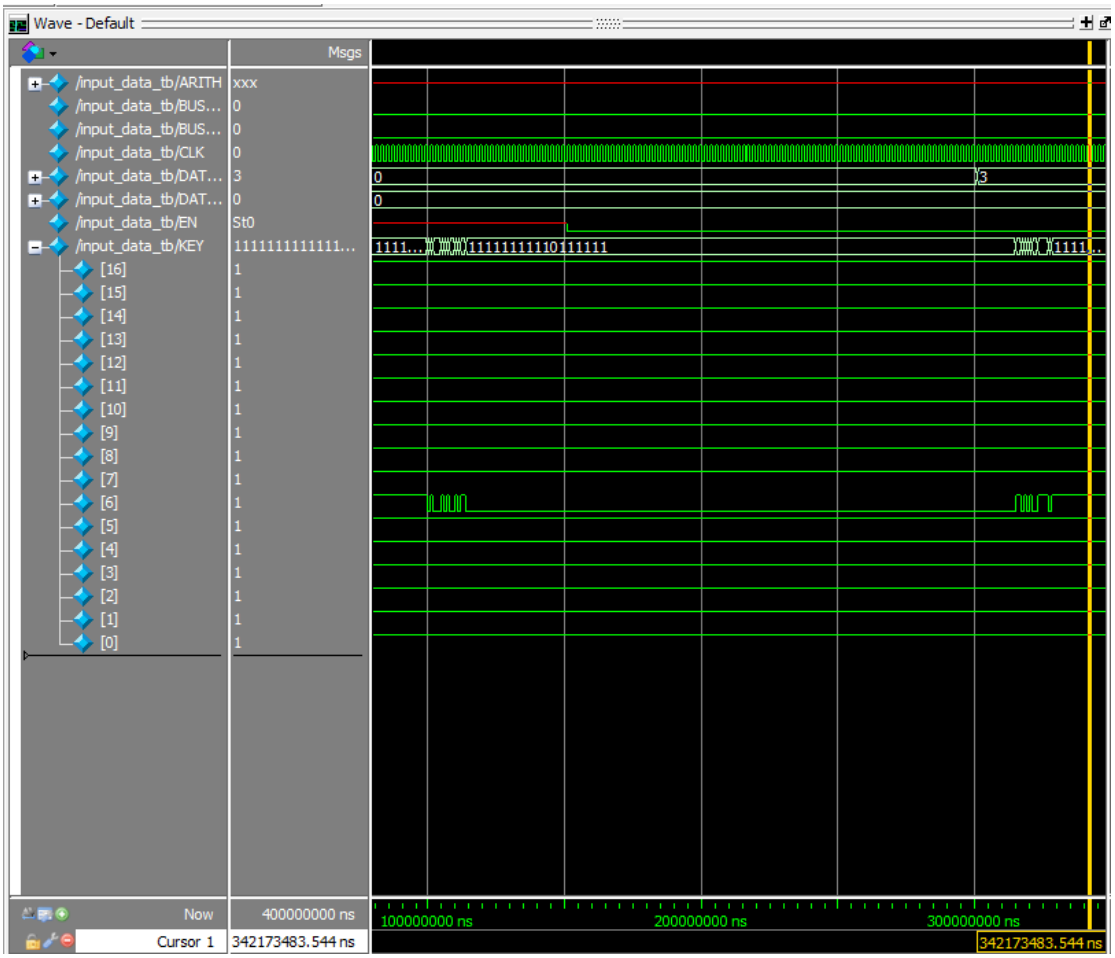
综上，两个测试用例的仿真结果都与预期一致，说明本模块（运算结果处理电路 U3）的程序编写达到预期效果。

发挥部分

（一）数据输入处理电路加入按键去抖电路

本模块设计的目的就是为了消除按键输入抖动，保证运算准确进行。

仿真结果：



仿真结果说明：

上图表明按下按键‘3’，观察上图 KEY[6]位输出波形中，有两段预期之外的密集矩形波，代表按键时的抖动。如果没有加入防抖动代码，在 DATA_1 中将出现连续个 3.但是因为加入了防抖动代码，DATA_1 只在预期的位置出现了 3，说明抖动被消除。

综上，该测试用例的仿真结果都与预期一致，说明本模块（数据输入处理电路加入按键去抖电路）的程序编写达到预期效果。

（二）为除法运算和开方运算增加余数输出功能

本模块的主要功能在于为 U2 和 U3 模块实现除法和开方运算的余数输出。具体仿真结果如下所示：

U2 求余部分仿真结果：

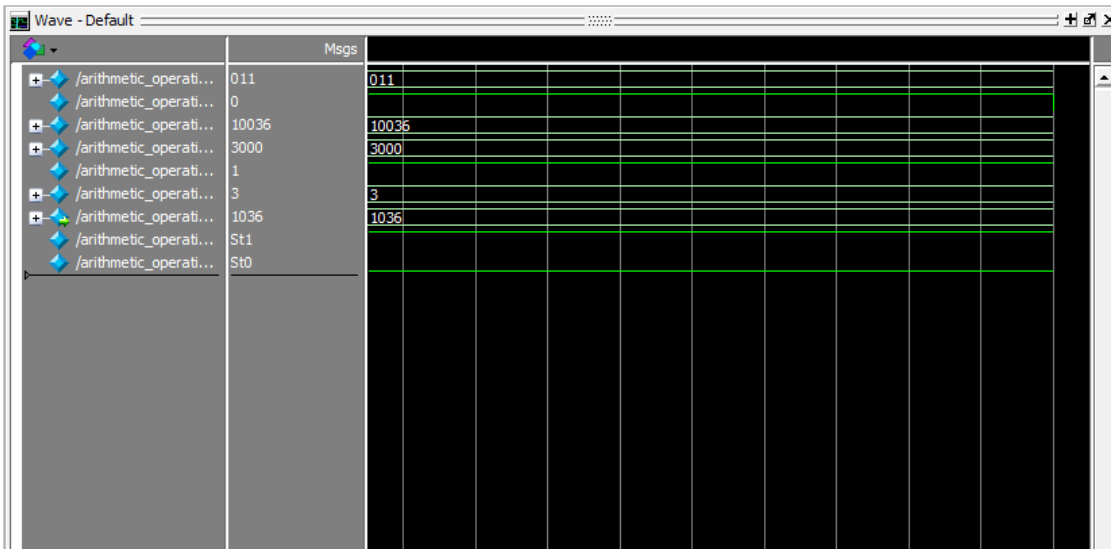
测试用例 1：除法运算

计算式：

输入数据/运算符

10036
÷
3000

结果：



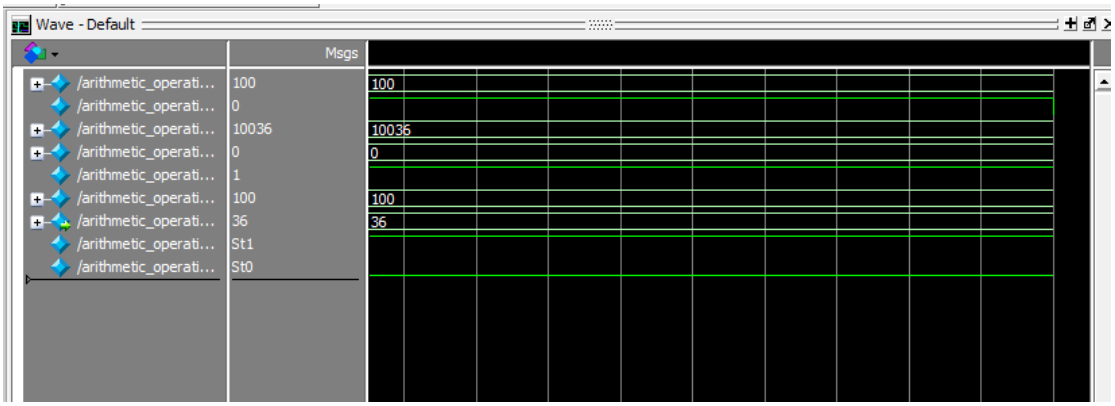
其中，上图显示余数计算结果为 1036，商为 3，与实际结果相符。

测试用例 2：开方运算

计算式：

输入数据/运算符
10036
开方

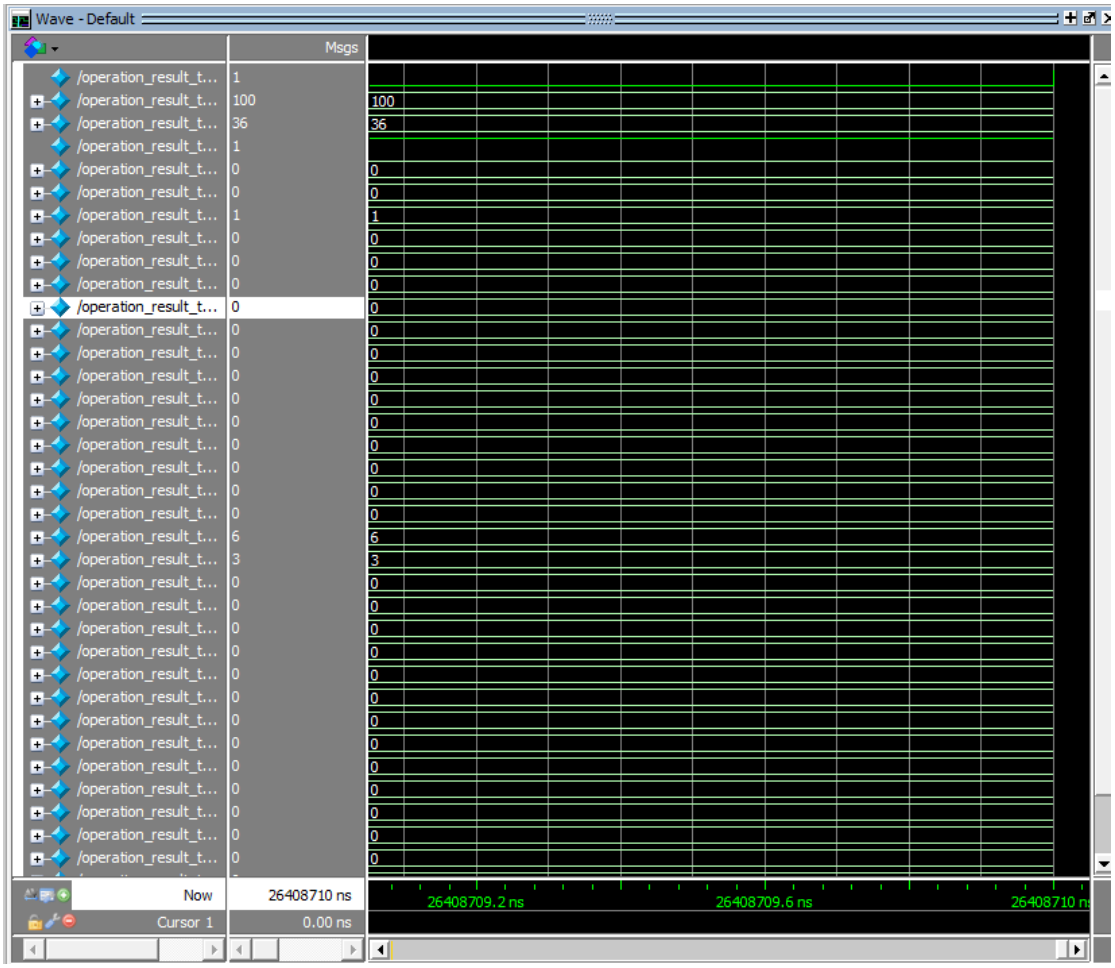
结果：



其中，上图显示余数计算结果为 36，开方结果为 100，与实际结果相符。

U3 余数显示部分仿真结果：

仿真结果：



仿真结果说明：

上图中，前两行数据分别为 100 和 36，分别代表 U2 电路中的商和余数。下面的数据则代表相应位的 BCD 码显示。显然，本仿真显示程序设计完成了对商和余数的 BCD 码显示。

最后，器件综合结果如下图所示：

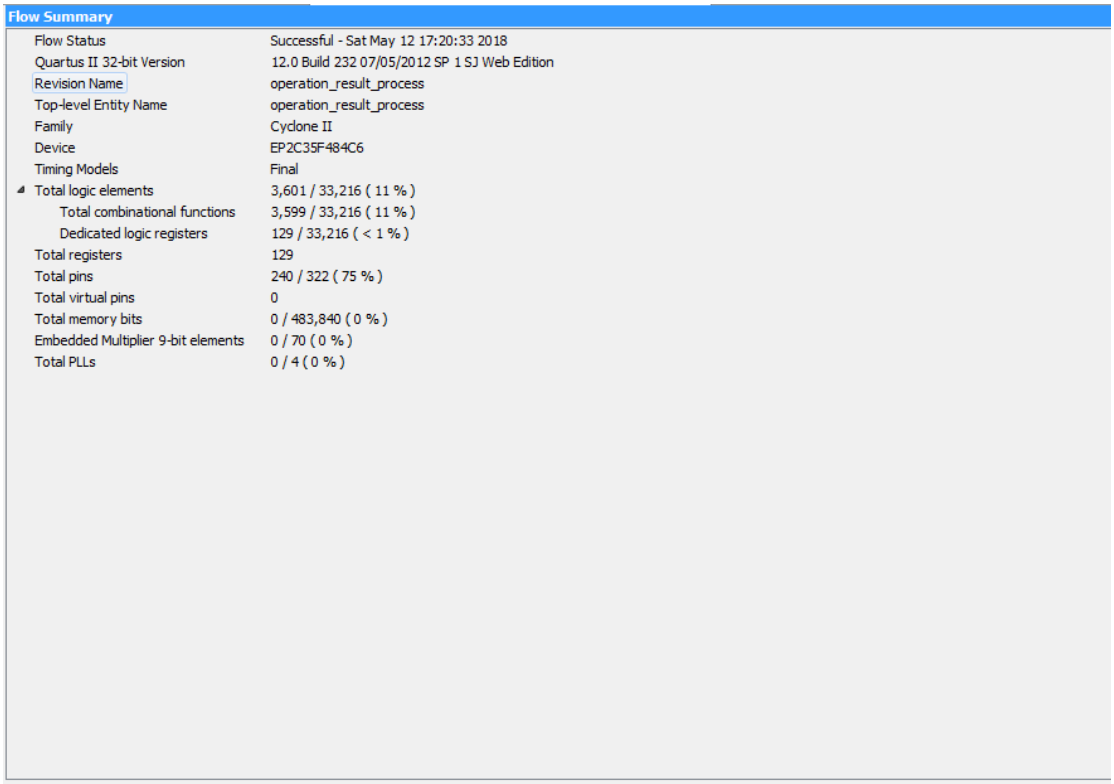
U1:

Flow Summary	
Flow Status	Successful - Sat May 12 17:17:32 2018
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Web Edition
Revision Name	input_data_process
Top-level Entity Name	input_data_process
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	484 / 33,216 (1 %)
Total combinational functions	484 / 33,216 (1 %)
Dedicated logic registers	59 / 33,216 (< 1 %)
Total registers	59
Total pins	78 / 475 (16 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

U2:

Flow Summary	
Flow Status	Successful - Sat May 12 17:19:15 2018
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Web Edition
Revision Name	arithmetic_operation
Top-level Entity Name	arithmetic_operation
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	2,130 / 33,216 (6 %)
Total combinational functions	2,115 / 33,216 (6 %)
Dedicated logic registers	150 / 33,216 (< 1 %)
Total registers	150
Total pins	169 / 475 (36 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	15 / 70 (21 %)
Total PLLs	0 / 4 (0 %)

U3:



七、结论

综合上述仿真结果及展示，本次设计通过合理的算法，圆满的完成了设计的全部基本要求和发挥部分。