REDC3329模块说明

1. 模块要求

根据任务要求实现一个12-bit位宽的流水线快速模乘器模块。模乘器输入a、b两个12-bit位宽的乘数，给定模数q=3329，在固定周期内输出a*b mod q。模块IO接口如表1所示，要求当模乘器输入使能有效时，a、b两个乘数输入，随后模乘器进行计算，在模乘器计算过程中，busy信号指示高电平为忙状态，否则为低电平指示空闲状态。在经过固定周期后，done信号由低变高，指示计算完成，同时输出结果。

2. 模块简介

模块名称：REDC3329_pipeline
输出延迟：4

端口	方向	位宽	描述
`clk`	Input	1	系统时钟信号。
`rst_n`	Input	1	低电平有效的异步复位信号。
`en`	Input	1	使能信号。当 `en` 为高时，模块在下一个时钟周期锁存输入 `a` 和 `b`。
`a`	Input	12	输入操作数 A。
`b`	Input	12	输入操作数 B。
`busy`	Output	1	忙信号。当模块正在处理数据时（从 `en` 拉高到 `done` 拉高之间），该信号为高。
`done`	Output	1	完成信号。当一次模乘计算完成，且输出 `r` 有效时，该信号拉高一个时钟周期。
`r`	Output	12	计算结果，即 `(a * b) mod 3329`。

3. 模块实现

3.1. Montgomery模乘算法

可以观察到题目中所给出的3329是一个质数，所以可以使用蒙哥马利算法简化取模的计算过程。算法的基本原理是通过一系列的转换将本来对一个普通数N的取模转换为对一个2的幂次R的取模过程，从而可以通过移位算法节省大量计算资源。为了达到对一个数T转换模数的目的，先构造REDC(a,b,R)函数解决$(ab)\cdot R^{-1}(modN)$的问题,这个函数是存在简便算法的。然后反复调用这个函数来实现模乘的过程。具体步骤如下：

3.1.1. REDC函数的设计

构造函数的基本思路是将大数T加上一个适当的数值m，使其能够被R整除，并且结果能够落在0到2N的范围内，并且仍然保证对N同余的特性。这样就可以直接使用简单的数据选择器实现对于N的取模。具体步骤如下：
首先构造加数:

$$m = N[T(modR)N'](modR)$$

其中：

$$N\cdot N' = -1(modR)$$

相加之后得到t：

$$t = T + m = T + N[T(modR)N'](modR) = T + [(NN')(modR)][(T(modR))](modR) = T - T(modR)$$

将t再modR的情况下进行化简：

$$t(modR) = T + [(NN')(modR)][(T(modR))](modR) = T - T(modR)$$

从上面的变换中，不难看出这个t是可以被R整除的。接下来将t除以R得到：

$$u = \frac{T+m}{R}$$

现在来估算u的范围：

$$T < N \cdot R$$

$$N[T(modR)N'](modR) < N \cdot R$$

$$0 \leq t < 2N$$

因此可以直接通过比较t和N的大小来实现对N的取模：

3.1.2. REDC函数的乘法性质

因为直接调用REDC函数会带有R的逆元，无法直接得到结果，所以通过多次调用转换来实现模乘的过程。具体步骤如下：
1.转换输入a,b为蒙哥马利域下的数：

$$a_{mont} = REDC(a, R^2)$$

$$b_{mont} = REDC(b, R^2)$$

2.计算a,b的乘积：

$$t = REDC(a_{mont}, b_{mont})$$

3.将结果转换回普通域下的数：

$$r = REDC(t, 1)$$

这样就刚好通过两部约去了最开始引入的R的次方影响，得到了正确的结果。

3.2. 算法验证

为了验证算法的正确性，编写了一个简单的python脚本来对比蒙哥马利模乘和直接模乘的结果。代码如下：

```
def REDC(a,b):
    t=a*b
    m=((t&0xFFF)*3327)&0xFFF
    u=(t+m*3329)>>12
    if u>=3329:
        return u-3329
    else:
        return u

def MENG(A,B):
    A1 = REDC(A, 2385)
    B1 = REDC(B, 2385)
    C1 = REDC(A1, B1)
    C2 = REDC(C1, 1)
    return C2

for i in range(4096):
```

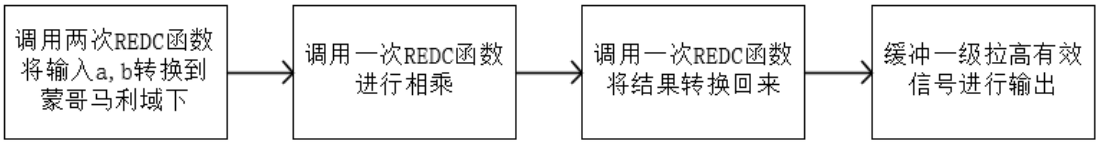
```
    for j in range(4096):
        if MENG(i,j) != (i*j)%3329:
            print("Error at i=",i," j=",j)
        else:
            if(i%1000==0 and j%1000==0):
                print("i=",i," j=",j," OK")
```

运行结果显示在所有的12-bit位宽输入下，蒙哥马利模乘和直接模乘的结果完全一致，验证了算法的正确性。

3.3. 硬件实现

3.3.1. 硬件流程图

根据上面的算法设计，编写了对应的硬件描述语言代码实现该功能，这也刚好对应了流水线的四级：



3.3.2. 硬件设计

REDC模块的设计如下：

```
module REDC_unit_comb(
    input [11:0] a,
    input [11:0] b,
    output [11:0] r
);
    parameter WIDTH = 12;
    parameter MOD = 3329;
    parameter MOD_INV = 3327;
    wire [2*WIDTH-1:0] T;
    wire [WIDTH-1:0] m;
    assign T=a*b;
    assign m=((T[WIDTH-1:0])*MOD_INV)&({WIDTH{1'b1}});
    wire [2*WIDTH-1:0] result_raw;
    assign result_raw=(T+m*MOD)>>WIDTH;
    wire [12:0] r_raw;
    assign r_raw=(result_raw[12:0]>=MOD)?(result_raw[12:0]-MOD):result_raw[12:0];
    assign r=r_raw[11:0];
endmodule
```

顶层模块的设计如下：

```
module REDC3329_pipeline(
    input clk,
    input rst_n,
    input en,
    input [11:0] a,
    input [11:0] b,
    output busy,
    output done,
    output reg [11:0] r
);
```

```
    parameter WIDTH = 12;
    parameter MOD = 3329;
    parameter MOD_INV = 3327;
    parameter R2_MOD = 2385;

    reg [WIDTH-1:0] A,B;
    reg [WIDTH-1:0] A_MONT, B_MONT;
    reg [WIDTH-1:0] r_MONT;
    wire [WIDTH-1:0] A_MONT_buffer,B_MONT_buffer;
    wire [WIDTH-1:0] r_MONT_buffer;
    reg  [WIDTH-1:0] r_buffer;
    reg [3:0] data_valid;

    always_ff@(posedge clk or negedge rst_n)begin
        if(!rst_n)begin
            A<=0;
            B<=0;
        end else begin
            data_valid[0]<=en;
            if(en)begin
                A<=a;
                B<=b;
            end
            data_valid[1]<=data_valid[0];
            data_valid[2]<=data_valid[1];
            data_valid[3]<=data_valid[2];
            A_MONT<=A_MONT_buffer;
            B_MONT<=B_MONT_buffer;
            r_MONT<=r_MONT_buffer;
            r<=r_buffer;
        end
    end

    assign busy=|data_valid;
    assign done=data_valid[3];

    REDC_unit_comb REDC_unit_comb_inst1(
        .a(A),
        .b(R2_MOD),
        .r(A_MONT_buffer)
    );
    REDC_unit_comb REDC_unit_comb_inst2(
        .a(B),
        .b(R2_MOD),
        .r(B_MONT_buffer)
    );
    REDC_unit_comb REDC_unit_comb_inst3(
        .a(A_MONT),
        .b(B_MONT),
        .r(r_MONT_buffer)
    );
    REDC_unit_comb REDC_unit_comb_inst4(
        .a(r_MONT),
        .b(1),
        .r(r_buffer)
    );
endmodule
```

4. 验证方案

因为12bit输入范围较小，可以通过穷举所有输入的方式来验证设计的正确性。编写了一个简单的testbench来实现该功能。将theroy信号进行四个周期的打拍，然后和r进行对比，如果不相等则说明设计有误。仿真结果显示在所有的12-bit位宽输入下，设计的结果和理论结果完全一致，验证了设计的正确性。部分仿真代码如下：

```
    initial begin
        rst_n = 0;
        en = 0;
        a = 0;
        b = 0;
        #10 rst_n = 1;
        for (int i = 0; i < 4096; i = i + 1) begin
            for (int j = 0; j < 4096; j = j + 1) begin
                a = i;
                b = j;
                theory = (a * b) % 3329;
                @(negedge clk);
                //wait(done);
                $display("a=%d, b=%d, r=%d, theory_buf=%d", a, b, r, theory_buf);
            end
            @(posedge clk);
            en=0;
            repeat(1000) @(negedge clk);
            en=1;
        end
        $finish;
    end
    wire wrong;
    assign wrong = (done && (r != theory_buf));
```


Name
Value
a[11:0]
4095
b[11:0]
fff
clk
0
rst_n
1
en
1
r[11:0]
354
busy
0
done
0
theory[11:0]
354
theory_shift[3:0][11:0]
354,354,354,354
theory_buf[11:0]
354
wrong
0
0. 000000000 ms
100. 000000000 ms
Name
Value
a[11:0]
24
b[11:0]
003
clk
1
rst_n
1
en
1
r[11:0]
000
busy
1
done
1
theory[11:0]
048
theory_shift[3:0][11:0]
000,018,030,048
theory_buf[11:0]
000
wrong
0
data_valid[3:0]
f
[3]
1
[2]
1
[1]
1
[0]
1
1, 223, 075. 000 ns
1, 223, 050. 000 ns
1, 223, 100. 000 ns
Name
Value
a[11:0]
23
b[11:0]
fff
clk
1
rst_n
1
en
0
r[11:0]
3cd
busy
0
done
0
theory[11:0]
3cd
theory_shift[3:0][11:0]
3cd,3cd,3cd,3cd
theory_buf[11:0]
3cd
wrong
0
data_valid[3:0]
0
[3]
0
[2]
0
[1]
0
[0]
0
1, 213, 050. 000 ns
1, 213, 100. 000 ns