同济大学计算机系

计算机组成原理实验报告

学　　号 **2151140**

姓　　名 王谦

专　　业 信息安全

授课老师 张冬冬

一、实验内容

32 位除法器

实验介绍：

通过本次试验，了解除法器的实现原理，并学习如何实现一个除法器，本实验将实现 32 位无符号除法器和 32 位带符号除法器

实验目标：

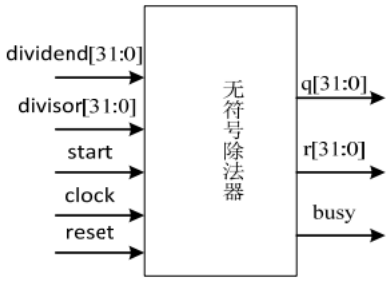
（1）了解 32 位有符号、无符号除法器的实现原理

（2）使用 Verilog 实现一个 32 位有符号除法器和一个 32 位无符号除法器

实验原理：

1） 无符号除法器

无符号除法器功能为将两个 32 位无符号数相除，得到一个 32 位商和 32 位余数。本实验分别实现 32 位有符号和无符号除法器，结果为 32 位商 quotient 和 32 位余数 remainder，分别存放在 CPU 的专用寄存器 LO 和 HI 中。除法器时钟信号下降沿时检查 start 信号，有效时开始执行，执行除法指令时，busy 标志位置 1。在执行除法指令时，任何情况下不产生算数异常，当除数为 0 时，运算结果未知，对除法器除数为 0 和溢出情况的发生通过汇编指令中其他指令进行检查和处理

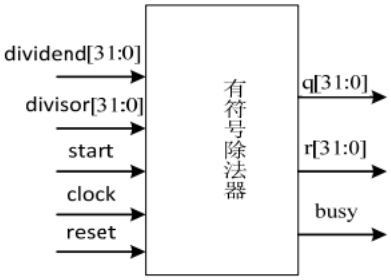


● 接口定义：

```
module DIVU(
    input [31:0]dividend,        //被除数
    input [31:0]divisor,         //除数
    input start,                 //启动除法运算
    input clock,
    input reset,
    output [31:0]q,              //商
    output [31:0]r,              //余数
    output busy                  //除法器忙标志位
    );
```

2） 带符号除法器

带符号除法器功能为：将两个 32 带无符号数相除，得到一个 32 位商和余数，基本和无符号除法器类似，注意余数符号与被除数符号相同。



● 接口定义：

```
module DIV(
    input [31:0]dividend,        //被除数
    input [31:0]divisor,         //除数
    input start,                 //启动除法运算
    input clock,
    input reset,
    output [31:0]q,              //商
    output [31:0]r,              //余数
    output busy                  //除法器忙标志位
    );
```

3） 参考思路

基于移位、减法的恢复余数除法器：

对于 32 位无符号除法，可将被除数 a 转换成高 32 位为 0 低 32 位是 a 的数 temp_a，在每个周期开始时 temp_a 向左移动一位，最后一位补零，然后判断 temp_a 的高 32 位是否大于等于除数 b，如是则 temp_a 的高 32 位减去 b 并且加 1，得到的值赋给 temp_a，如果不是则直接进入下一步，执行结束后 temp_a 的高 32 位即为余数，低 32 位即为商。对于 32 位有符号除法，可先将有符号数转换成无符号数除法，根据被除数和除数的符号判断商的符号，被除数是负数时余数为负，否则为正。

不恢复余数除法器：

不恢复余数即不管相减结果是正还是负，都把它写入 reg_r，若为负，下次迭代不是从中减去除数而是加上除数。

实验步骤：

1.新建工程

2.编写有符号除法器模块

3.编写无符号除法器模块

4.用 ModelSim 仿真测试各模块

二、硬件逻辑图

（实验步骤中要求用 logisim 画图的实验，在该部分给出 logisim 原理图，否则该部分在实验报告中不用写）

——本次实验不要求——

三、模块建模

（该部分要求对实验中建模的所有模块进行功能描述，并列出各模块建模的 verilog 代码）

无符号 DIVU：

定点数的除法运算

与定点数的乘法运算类似，定点数的出发也分为原码除法运算和补码除法运算。我们在做除法时可以一眼看出来够不够除，但是计算机却做不到，所以它需要先做加减法（被除数与除数做加减法），如果是负数说明不够除，是正数就说明够除。如果不够除就需要恢复成原来的余数以便进行下一步运算，我们把这种方法称为恢复余数法；当然也可以不恢复余数，那么这种方法就称为不恢复余数法（原码加减交替法）。我们就依次来了解它们的基本思想和运算步骤吧！

● 原码除法运算

原码除法主要采用原码不恢复余数法，特点是商符（结果的符号位）和商值（结果的数值位）是分开进行的，商符由操作数的符号位“异或”形成。

还记得原码乘法运算吗，它的规则也是符号位由操作数的符号位异或而成，数值部分是两个操作数的绝对值积。那么同样的，求商值的规则也是如此：两个操作数相除，商的符号由两个操作数的符号位异或而成，商的数值的绝对值由两个操作数的绝对值相除而成。

下面来看看原码除法运算的基本运算规则：

1) 符号位不参与运算。

2) 先用被除数减去除数（|X| - |Y| = |X| +(-|Y|)=|X| + [-|Y|]补），当余数为正（代表能除够），商为1，余数和商左移一位，再减去除数；当余数为负时，商上0，余数和商左移一位，再加上除数。

3) 当第n+1步余数为负，需要加上|Y|得到第n+1步正确的余数（余数与被除数同号）。

运算规则看似有些复杂，不过仔细观察其实是将我们日常的除法思维移植到计算机上而已。我们在做除法时第一步也是需要判断够不够除，比如3/5，我们就很明显知道不够除应该商0；不过计算机就需要做一次减法（被除数3减去除数5）得到结果为负数后再商0。在得到第一步结果后我们需要做一次运算来更正接下来的结果（也就是商1左移后减去除数，商0左移后加上除数），需要注意的最后我们需要判断第n+1步的余数是否需要修正。

```
`timescale 1ns / 1ps

module DIVU (dividend, divisor, start, clock, reset, q, r, busy);
input [31:0]dividend;         //被除数
input [31:0]divisor;           //除数
```

```
input start;                //启动除法运算
input clock;
input reset;
output [31:0]q;             //商
output [31:0]r;             //余数
output reg busy;               //除法器忙标志位

reg [63:0]div_a;
reg [31:0]diver, div_b, div_q, div_r;

always@(posedge clock or posedge reset)
begin
    if(reset)
    begin
        diver <= 32'b0;
        div_a <= 64'b0;
        div_b <= 32'b0;
        busy <= 0;
    end

    else if(start)

    begin
        busy = 1;
        diver = 32'b0;
        div_a = dividend;
        div_b = divisor;

        if(busy)
        begin
        repeat(32)
            begin
                if(div_a[63:32] >= div_b[31:0])
                begin
                    diver = diver + {31'b0, 1'b1};
                    div_a = div_a - {div_b[31:0], 32'b0};
                end
                div_a = div_a << 1;
                diver = diver << 1;
            end
        end

        if(div_a[63:32] >= div_b[31:0])
        begin
```

```
            diver = diver + {31'b0, 1'b1};
            div_a = div_a - {div_b[31:0], 32'b0};
        end

        busy = 0;
    end
end

assign q = diver;
assign r = div_a[63:32];

endmodule
```

有符号 DIV：

补码整数的不恢复余数除法

0. 首先给出算法核心的原理:

在得到第 i+1 次的部分余时，在试商的时候我们是通过 $R_{i+1}=2R_i-Y$ 来得到这次的部分余的，如果发现不够减，我们又要加回去，但是我们可以把这个恢复的步骤和下一次的试商步骤合到一起，比如如果 $2R_i-Y$ 算出来是不够减的，那么在算第 i+2 次的部分余时我们应该用 $2(R_{i+1}+Y)-Y=2R_{i+1}+Y$ 来计算。这样就把恢复和试商并成了一步，提高了效率。

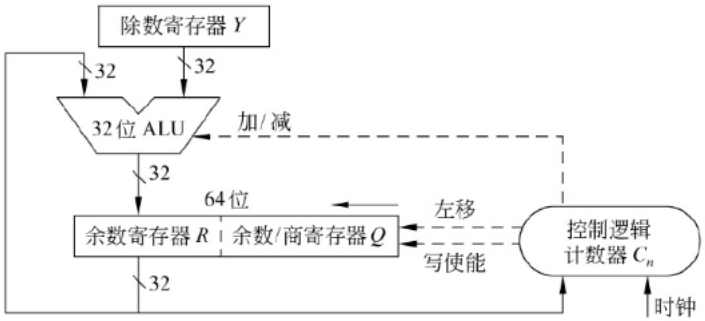


图 3.15　32 位除法运算逻辑结构

1. 算法要点

(1) 操作数的预置：除数装入 Y，被除数符号扩展之后装入 R 和 Q。

(2) 根据以下规则确定第一位商Qn:

若 R，Y 同号，则做减法，$R_1=R-Y$,否则做加法，$R_1=R+Y$ 。

①若 R_1 和 Y 同号，那么 Qn = 1，转到第(3) 步。

②否则 Qn = 0，转到第 (3) 步。

(3)

for i = 1 to n-1:

如果 R_i 和 Y 异号，那么 $R_{i+1} = 2R_i + Y$，否则 $R_{i+1} = 2R_i - Y$。

如果新的余数 R_{i+1} 和 Y 同号，则上商为 1，否则上商为 0.

(4) 第 n 次，按照同样的规则上商之后仅左移 Q。

(5) 修正商：如果被除数和除数异号，商加一。

(6)修正余数：如果余数和被除数同号，不需要修正，否则，如果被除数和除数同号，余数加上被除数，否则余数减去被除数。

但是在不同符号数相除的情形当中仍然有一个 bug，那就是正数除以负数且能整除的情况。比如 -8/2，我们会得到结果商为 -3,余数为 -2。试了一下，**当被除数为负数，除数为正数，且可以整除的时候**会出现这种情况。这时需要加一个特判修正，如果余数加上除数为0，那么商减一余数置 0 。

```
`timescale 1ns / 1ps

module DIV(dividend, divisor, start, clock, reset, q, r, busy);
input [31:0]dividend;          //被除数
input [31:0]divisor;           //除数
input start;                  //启动除法运算
input clock;
input reset;
output [31:0]q;               //商
output [31:0]r;               //余数
output reg busy;                 //除法器忙标志位


reg [63:0] div_a;
reg [31:0] div_b, div_c, div_d;

always@(posedge clock or posedge reset)
begin
    if(reset)
    begin
        div_a <= 64'b0;
        div_b <= 32'b0;
        busy <= 0;
    end

    else if(start)

    begin
        busy = 1;
```

```
        div_b = divisor[31:0];

        if(dividend[31])
        begin
            div_a[63:0] = {32'b1111_1111_1111_1111_1111_1111_1111_1111,
dividend[31:0]};
        end
        else
        begin
            div_a[63:0] = {32'b0000_0000_0000_0000_0000_0000_0000_0000,
dividend[31:0]};
        end

        if(busy)
        begin
            if(div_a[63] == div_b[31])
            begin
                div_a = div_a[63:0] - {div_b[31:0], 32'b0};
            end
            else if(div_a[63] != div_b[31])
            begin
                div_a = div_a[63:0] + {div_b[31:0], 32'b0};
            end

            repeat(32)
            begin
                if(div_a[63] == div_b[31])
                begin
                    div_a = div_a <<< 1;
                    div_a = {div_a[63:1], 1'b1};
                    div_a = div_a[63:0] - {div_b[31:0], 32'b0};
                end
                else if(div_a[63] != div_b[31])
                begin
                    div_a = div_a <<< 1;
                    div_a = div_a[63:0] + {div_b[31:0], 32'b0};
                end
            end
        end

        div_c = div_a[63:32];
        div_d = div_a[31:0];
        div_d = div_d <<< 1;
        if(div_c[31] == div_b[31])
```

```
        begin
            div_d[31:0] = div_d[31:0] + 1'b1;
        end

        if(dividend[31] != divisor[31])
        begin
            div_d = div_d[31:0] + 1'b1;
        end

        if(div_c[31] != dividend[31])
        begin
            if(dividend[31] != divisor[31])
            begin
                div_c = div_c[31:0] - divisor[31:0];
            end
            else
            begin
                div_c = div_c[31:0] + divisor[31:0];
            end
        end

        if(!(divisor[31:0] + div_c[31:0]))
        begin
            div_d = div_d - 1;
            div_c = 32'b0;
        end

        busy = 0;
    end
end

assign q = div_d[31:0];
assign r = div_c[31:0];

endmodule
```

四、测试模块建模

（要求列写各建模模块的 test bench 模块代码）

两者采用相近的 tb 数据，但是结果会有所不同：

```
`timescale 1ns / 1ps

module DIV_tb();
reg [31:0]dividend;         //被除数
reg [31:0]divisor;          //除数
reg start;                 //启动除法运算
reg clock;
reg reset;
wire [31:0]q;              //商
wire [31:0]r;              //余数
wire busy;                 //除法器忙标志位


initial
begin
    reset <= 0;
end

initial
clock = 0;
always #10 clock = ~clock;

initial
begin
    start = 1;
    dividend <= 32'b0001_1011_1101;
    divisor <= 32'b10101;
    #50
    start = 1;
    dividend <= 32'h00000000;
    divisor <= 32'hffffffff;
    #50
    start = 1;
    dividend <= 32'hffffffff;
    divisor <= 32'hffffffff;
    #50
    start = 1;
    dividend <= 32'hffffffff;
    divisor <= 32'haaaaaaaa;
    #50
    start = 1;
    dividend <= 32'haaaaaaaa;
    divisor <= 32'hffffffff;
    #50
    start = 1;
```

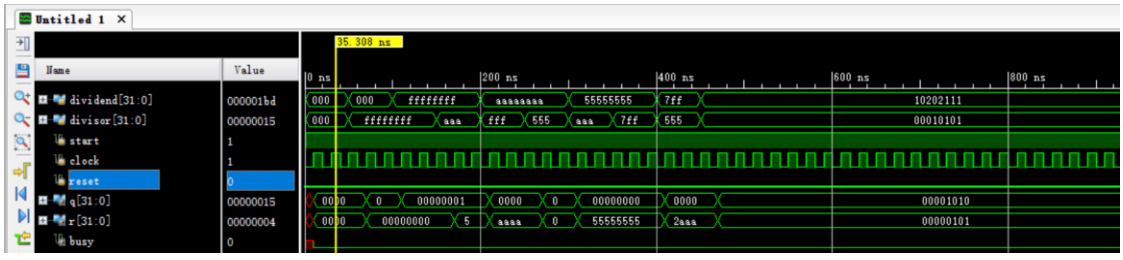
```
    dividend <= 32'haaaaaaaa;
    divisor <= 32'h55555555;
    #50
    start = 1;
    dividend <= 32'h55555555;
    divisor <= 32'haaaaaaaa;
    #50
    start = 1;
    dividend <= 32'h55555555;
    divisor <= 32'h7fffffff;
    #50
    start = 1;
    dividend <= 32'h7fffffff;
    divisor <= 32'h55555555;
    #50
    start = 1;
    dividend <= 32'h10202111;
    divisor <= 32'h10101;
    #50
    start = 1;
    dividend <= 32'b1111_1111_1111_1111_1111_1111_1111_0111;
    divisor <= 32'b0000_0000_0000_0000_0000_0000_0000_0010;
    #50
    start = 1;
    dividend <= 32'b1111_1111_1111_1111_1111_1111_1111_1001;
    divisor <= 32'b0000_0000_0000_0000_0000_0000_0000_0011;
    #50
    start = 1;
    dividend <= 32'b1111_1111_1111_1111_1111_1111_1111_1000;
    divisor <= 32'b0000_0000_0000_0000_0000_0000_0000_0010;
    #50
    start = 1;
    dividend <= 32'b0101_0101_0101_0101_0101_0101_0101_0101;
    divisor <= 32'b1010_1010_1010_1010_1010_1010_1010_1010;
    #50
    start = 1;
    dividend <= 32'b1010_1010_1010_1010_1010_1010_1010_1010;
    divisor <= 32'b0101_0101_0101_0101_0101_0101_0101_0101;
    #50
    start = 1;
    dividend <= 32'b0000_0000_0000_0000_0000_0000_0000_1001;
    divisor <= 32'b0000_0000_0000_0000_0000_0000_0000_0010;
    #50
    start = 1;
```

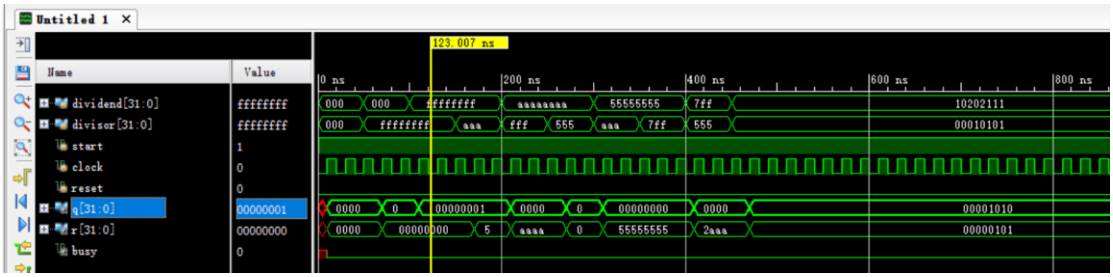
```
    dividend <= 32'b0000_0000_0000_0000_0000_0000_0000_1001;
    divisor <= 32'b1111_1111_1111_1111_1111_1111_1111_1110;
    #50
    start = 1;
    dividend <= 32'b1111_1111_1111_1111_1111_1111_1111_1000;
    divisor <= 32'b1111_1111_1111_1111_1111_1111_1111_1110;
    #50
    start = 1;
    dividend <= 32'b1111_1111_1111_1111_1111_1111_1111_1001;
    divisor <= 32'b0000_0000_0000_0000_0000_0000_0000_0010;

end

DIV DIV_init(
    .dividend(dividend),
    .divisor(divisor),
    .start(start),
    .clock(clock),
    .reset(reset),
    .q(q),
    .r(r),
    .busy(busy)
);

endmodule
```

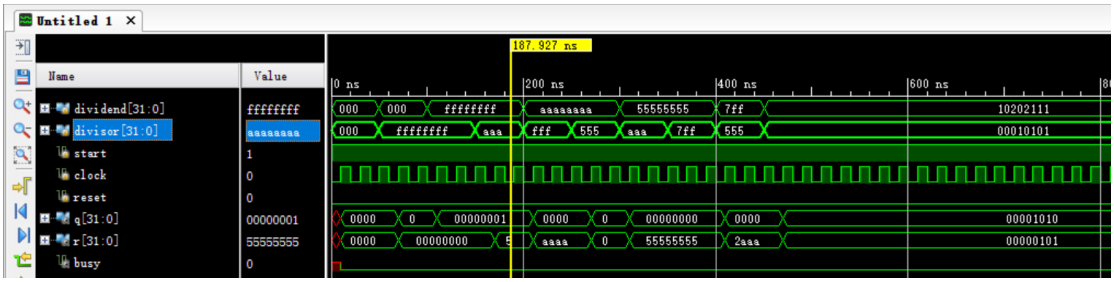
五、实验结果

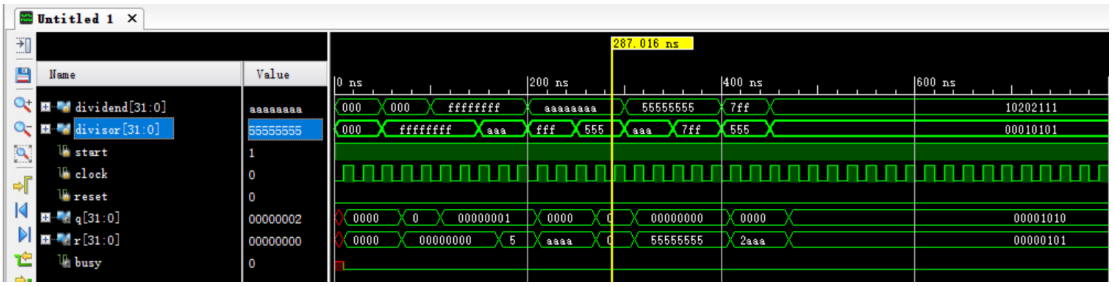
（该部分可截图说明，要求 logisim 逻辑验证图、modelsim 仿真波形图、以及下板后的实验结果贴图（实验步骤中没有下板要求的实验，不需要下板贴图））

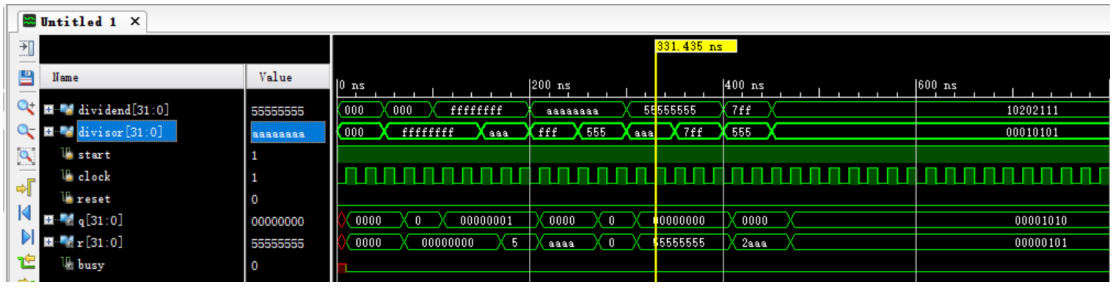
无符号 DIVU：

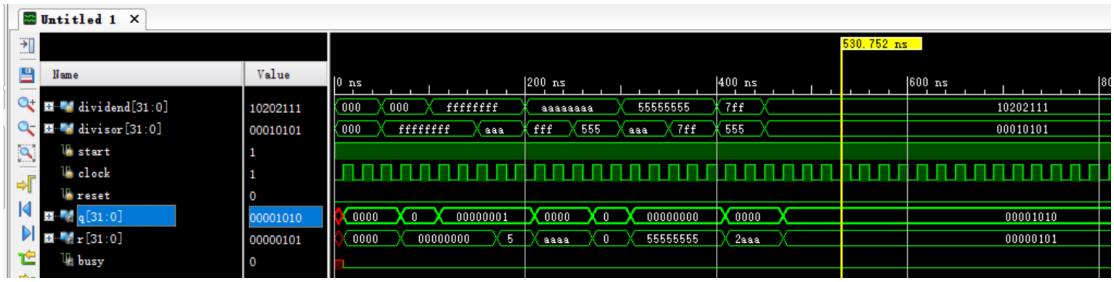



Untitled 1
123.007 ns
Name
Value
dividend[31:0]
ffffffff
divisor[31:0]
ffffffff
start
1
clock
0
reset
0
q[31:0]
00000001
r[31:0]
00000000
busy
0

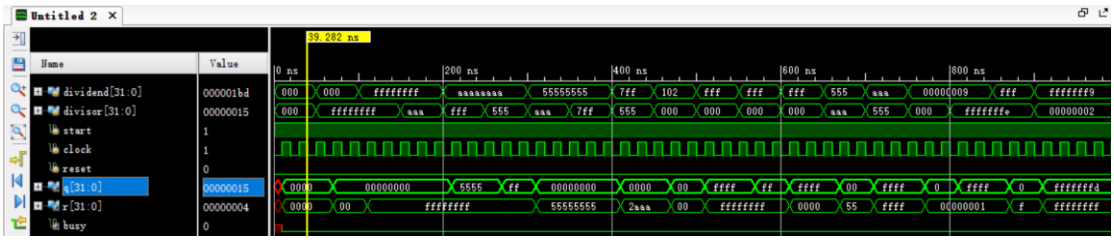

Untitled 1
187.927 ns
Name
Value
dividend[31:0]
ffffffff
divisor[31:0]
aaaaaaaa
start
1
clock
0
reset
0
q[31:0]
00000001
r[31:0]
55555555
busy
0

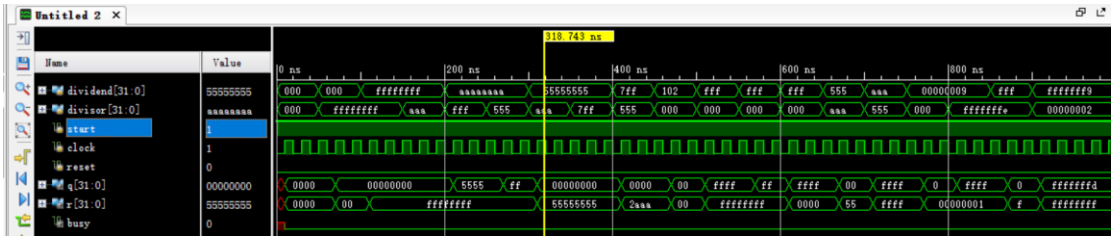

Untitled 1
287.016 ns
Name
Value
dividend[31:0]
aaaaaaaa
divisor[31:0]
55555555
start
1
clock
0
reset
0
q[31:0]
00000002
r[31:0]
00000000
busy
0

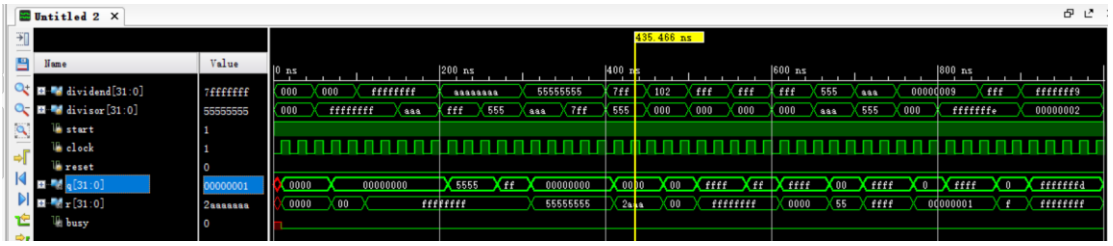

Untitled 1
331.435 ns
Name
Value
dividend[31:0]
55555555
divisor[31:0]
aaaaaaaa
start
1
clock
1
reset
0
q[31:0]
00000000
r[31:0]
55555555
busy
0

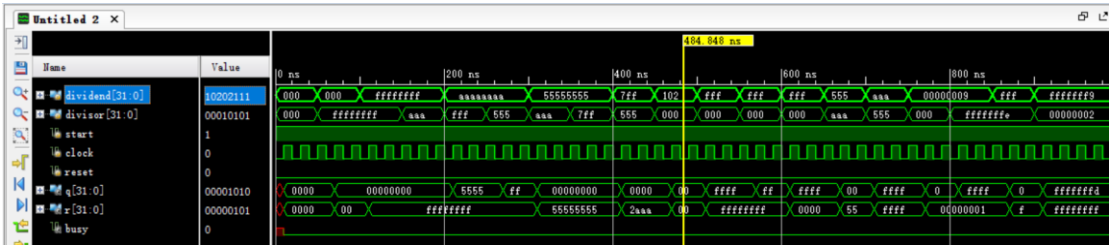

Untitled 1
530.752 ns
Name
Value
dividend[31:0]
10202111
divisor[31:0]
00010101
start
1
clock
1
reset
0
q[31:0]
00001010
r[31:0]
00000101
busy
0

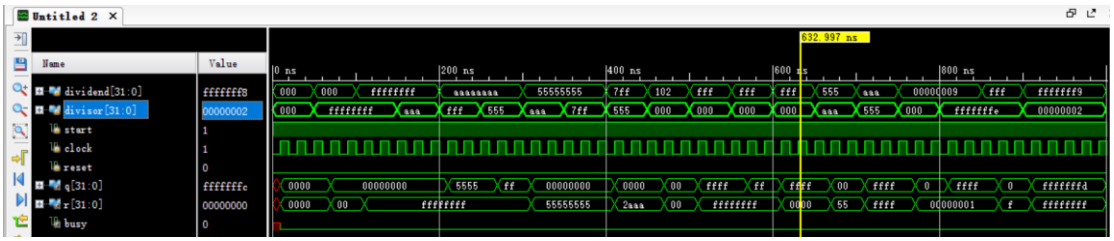
有符号 DIV：

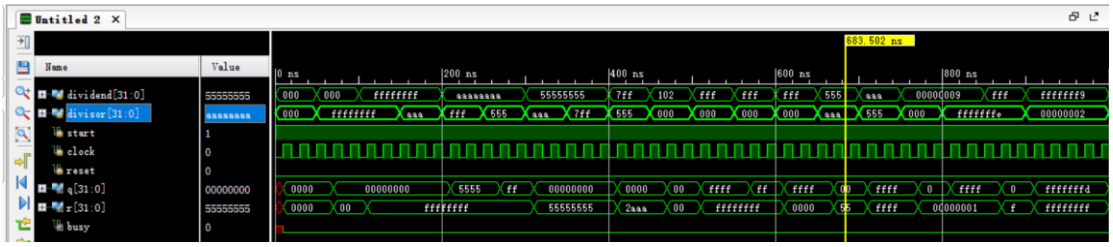

Untitled 2
39.282 ns
Name
Value
dividend[31:0]
000001bd
divisor[31:0]
00000015
start
1
clock
1
reset
0
q[31:0]
00000015
r[31:0]
00000004
busy
0

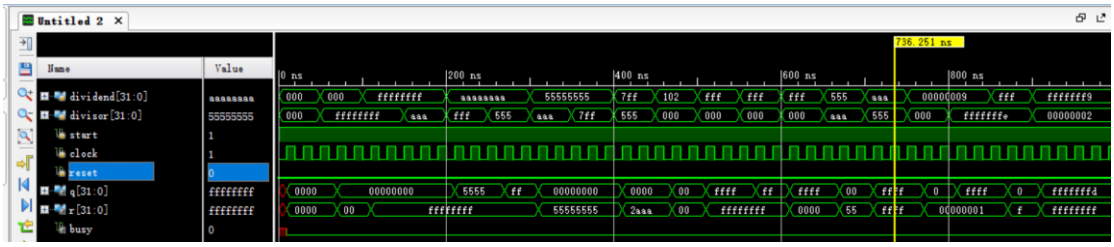

Untitled 2
318.743 ns
dividend[31:0] 55555555
divisor[31:0] aaaaaaaa
start 1
clock 1
reset 0
q[31:0] 00000000
r[31:0] 55555555
busy 0

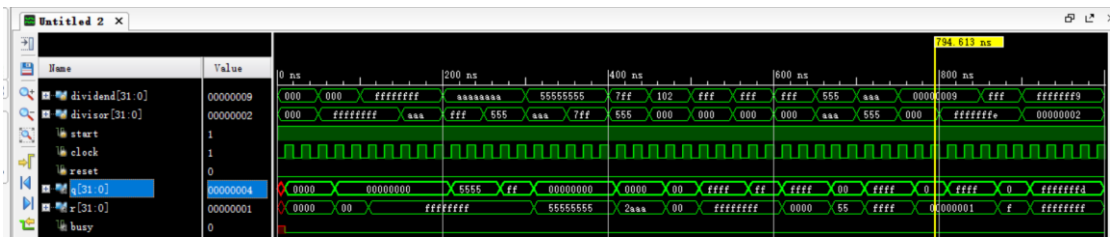

Untitled 2
435.466 ns
dividend[31:0] 7fffffff
divisor[31:0] 55555555
start 1
clock 1
reset 0
q[31:0] 00000001
r[31:0] 2aaaaaaa
busy 0


Untitled 2
484.848 ns
dividend[31:0] 10202111
divisor[31:0] 00010101
start 1
clock 0
reset 0
q[31:0] 00001010
r[31:0] 00000101
busy 0


Untitled 2
632.997 ns
dividend[31:0] fffffff8
divisor[31:0] 00000002
start 1
clock 1
reset 0
q[31:0] fffffffc
r[31:0] 00000000
busy 0


Untitled 2
683.502 ns
dividend[31:0] 55555555
divisor[31:0] aaaaaaaa
start 1
clock 0
reset 0
q[31:0] 00000000
r[31:0] 55555555
busy 0


Untitled 2
736.251 ns
dividend[31:0] aaaaaaaa
divisor[31:0] 55555555
start 1
clock 1
reset 0
q[31:0] ffffffff
r[31:0] ffffffff
busy 0


Untitled 2
794.613 ns
dividend[31:0] 00000009
divisor[31:0] 00000002
start 1
clock 1
reset 0
q[31:0] 00000004
r[31:0] 00000001
busy 0