Lab1 基本组件实现

简单 ALU 实现

> 题解:

给定操作数 a、b、操作码，执行对应的操作运算

按照 add、sub、multi、div、mod、and、or、xor、not、sll、srl、sra、slt、~^、~&、~|的指令顺序

模块设计代码

```
`timescale 1ns / 1ps

module simple_alu(
        input [31:0] a,
        input [31:0] b,
        input [3:0] func,
        output reg[31:0] res1,
        output reg[31:0] res2
        //加减乘数时的高位"溢出"
);

always @(*) begin//综合成组合逻辑，内部采用阻塞赋值
        case (func)
                4'b0000:{res1,res2}=a+b;//add
                4'b0001:{res1,res2}=a-b;//sub
                4'b0010:{res1,res2}=a*b;//multiply
                4'b0011:{res1,res2}=a/b;//divide
                4'b0100:res2=a%b;//mod
```

```
                4'b0101:res2=a&b;//and
                4'b0110:res2=a|b;//or
                4'b0111:res2=a^b;//xor
                4'b1000:res2=~a;//not
                4'b1001:res2=a<<b;//shl
                4'b1010:res2=a>>b;//shr
                4'b1011:res2=a>>>b;//sra
                4'b1100:res2=(a<b);//<
                4'b1101:res2=~(a^b);//同或
                4'b1110:res2=~(a&b);//与非
                4'b1111:res2=~(a|b);//或非
                default: {res1,res2}=0;
        endcase
end

endmodule
```

Verilog

测试实例

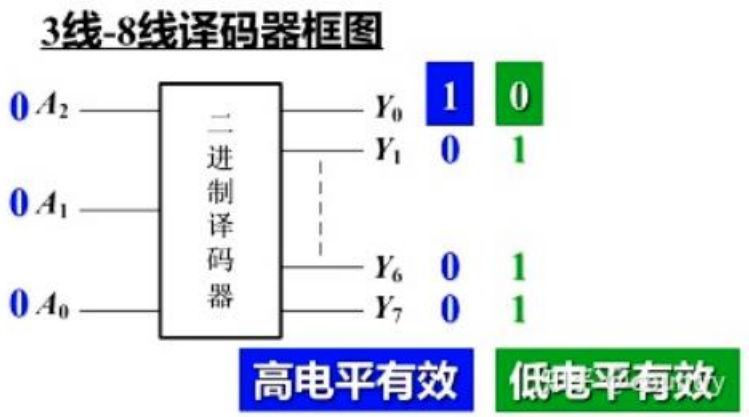
```
`timescale 1ns / 1ps
module alu_test();
    reg [31:0] a;
    reg [31:0] b;
    reg [3:0]func;
    wire [31:0]res1;
    wire [31:0]res2;

    initial begin
        a=2147483647;
        b=1111111;
        func=4'b0;
```

```verilog
        repeat (16)begin
            #10;
            func=func+4'b1;
        end
    end

    simple_alu alu(.a (a),.b(b),.func(func),.res1(res1),.res2(res2));

    always @(*) begin
        if ($time>=1000) begin
            #10;
            $finish;
        end
    end
endmodule
```

Verilog

$nrightarrow2^n$ 译码器

原理



高电平有效与低电平有效输出

译码器的构造如左图：输入若干根线，输入的二进制编码所对应的十进制数所对应的输出端有效，其余无效。如：000➡Y0, 001➡Y1……

设计源代码

```
`timescale 1ns / 1ps
/*
3-8 译码器,是输入 3 根线，输出 8 根线
比如输入 000，就输出的最底线有效——高有效或者低有效
*/


module decoder3_8(
        input [2:0] d_in,
        output  [7:0] d_out
);
genvar i;
generate for(i=0;i<=7;i=i+1)begin:generate_decoder3_8
    assign d_out[i]=(d_in==i);
end
endgenerate
endmodule

module decoder4_16(
    input[3:0]d_in,
    output[15:0]d_out
);
genvar j;
generate for(j=0;j<=15;j=j+1) begin:generate_decoder4_16
    assign d_out[j]=(d_in==j);
end
endgenerate
endmodule
```

Verilog

✍注意 generate 内的赋值为连续赋值，且 begin 后要设置名字

测试代码

```Verilog
`timescale 1ns / 1ps
module decoder_test();
    reg [2:0] d_in;
    reg [3:0] d_in2;
    wire [7:0] d_out;
    wire [15:0] d_out2;
    initial begin
        d_in=3'b000;
        d_in2=4'b0000;
        repeat (7) begin
            #10;
            d_in=d_in+1;
        end
        repeat (15) begin
            #10;
            d_in2=d_in2+1;
        end
    end

    decoder3_8 decoder3_8_test(.d_in (d_in),.d_out(d_out));
    decoder4_16 decoder4_16_test(.d_in(d_in2),.d_out(d_out2));

    always @(*) begin
        if ($time>=1000) begin
            #10;
            $finish;
        end
    end
endmodule
```

$2^n rightarrow$编码器

原理

以 8-3 编码为例，输入 8 根线，若第 i 根线为有效信号，则用二进制表示 i 并输出

设计代码

```Verilog
`timescale 1ns / 1ps
/*
编码器和译码器正相反
0->000
1->001
2->002
*/
module encoder8_3(
    input [7:0] d_in,
    output reg[2:0] d_out
);
always @(*) begin:for_loop
    integer i;
    for (i = 0; i<=7; i=i+1) begin
        if(d_in[i]==1'b1)begin
            d_out=i;
        end
    end
end
endmodule
//case 也可实现
```

测试代码

```Verilog
`timescale 1ns / 1ps
```

```verilog
module encoder_test();
    reg [7:0]d_in;
    wire [2:0] d_out;

    initial begin
        d_in=8'b1;
        forever begin
            #10;
            d_in=d_in<<1;
            if (d_in==0) begin
                d_in=8'b1;
            end
        end
    end

    encoder8_3 encoder8_3_test(.d_in(d_in),.d_out(d_out));

    always @(*) begin
        if ($time>=1000) begin
            #10;
            $finish;
        end
    end
endmodule
```

Verilog

多路选择器

原理

多路选择器既可以用每位 1 选，也可以用二进制编码选

每位 1：0001 选第一个，0010 选第二个，0100 选第三个……

二进制编码：001 选第一个，010 选第二个，011 选第三个……

设计源代码

```
`timescale 1ns / 1ps


module mux_bycode(//8位宽4选1
    input [7:0] in_1,
    input [7:0] in_2,
    input [7:0] in_3,
    input [7:0] in_4,
    input [1:0] sel,
    output [7:0] res
);
assign res=(sel==2'b00)?in_1:
           (sel==2'b01)?in_2:
           (sel==2'b10)?in_3:
           (sel==2'b11)?in_4:8'bx;
endmodule

module mux_bybit(
    input [7:0] in_1,
    input [7:0] in_2,
    input [7:0] in_3,
    input [7:0] in_4,
    input [3:0] sel,
    output [7:0] res
);
assign res=({8{sel[0]}}&in_1)|({8{sel[1]}}&in_2)|({8{sel[2]}}&in_3)|({8{sel[3]}}&in_4);
endmodule
```

Verilog

测试源代码

```
`timescale 1ns / 1ps
```

```
module mux_test();
    reg [7:0] in_1,in_2,in_3,in_4;
    reg [1:0] sel_1;
    reg [3:0] sel_2;
    wire [7:0] res;

    initial begin
        in_1=1;
        in_2=3;
        in_3=5;
        in_4=7;
        sel_1=2'b00;
        sel_2=4'b0001;
        forever begin
            #10;
            sel_1=sel_1+1;
            sel_2=sel_2<<1;
            if(sel_2==0) begin
                sel_2=4'b0001;
            end
        end
    end

    mux_bycode mux_bycode_test(.in_1 (in_1),.in_2(in_2),.in_3(in_3),.in_4(in_4),
                               .sel(sel_1),.res(res));
    mux_bybit mux_bybit_test(.in_1 (in_1),.in_2(in_2),.in_3(in_3),.in_4(in_4),
                               .sel(sel_2),.res(res));

    always @(*) begin
        if($time>=1000)begin
            #10;
            $finish;
        end
```

```
    end
endmodule
```
Verilog

全加器

一位全加器

✍一位全加器的代码实现：

$$So = Ai \oplus Bi \oplus Ci$$

$$Co = AiBi + Ci(Ai + Bi)$$

```
module full_adder1(
  input Ai,Bi,Ci,
  output So,Co);

  assign So=Ai^Bi^Ci;
  assign Co=(Ai&Bi)|(Ci&(Ai|Bi));
endmodule
```
Verilog

四位全加器——用一位全加器嵌套实现

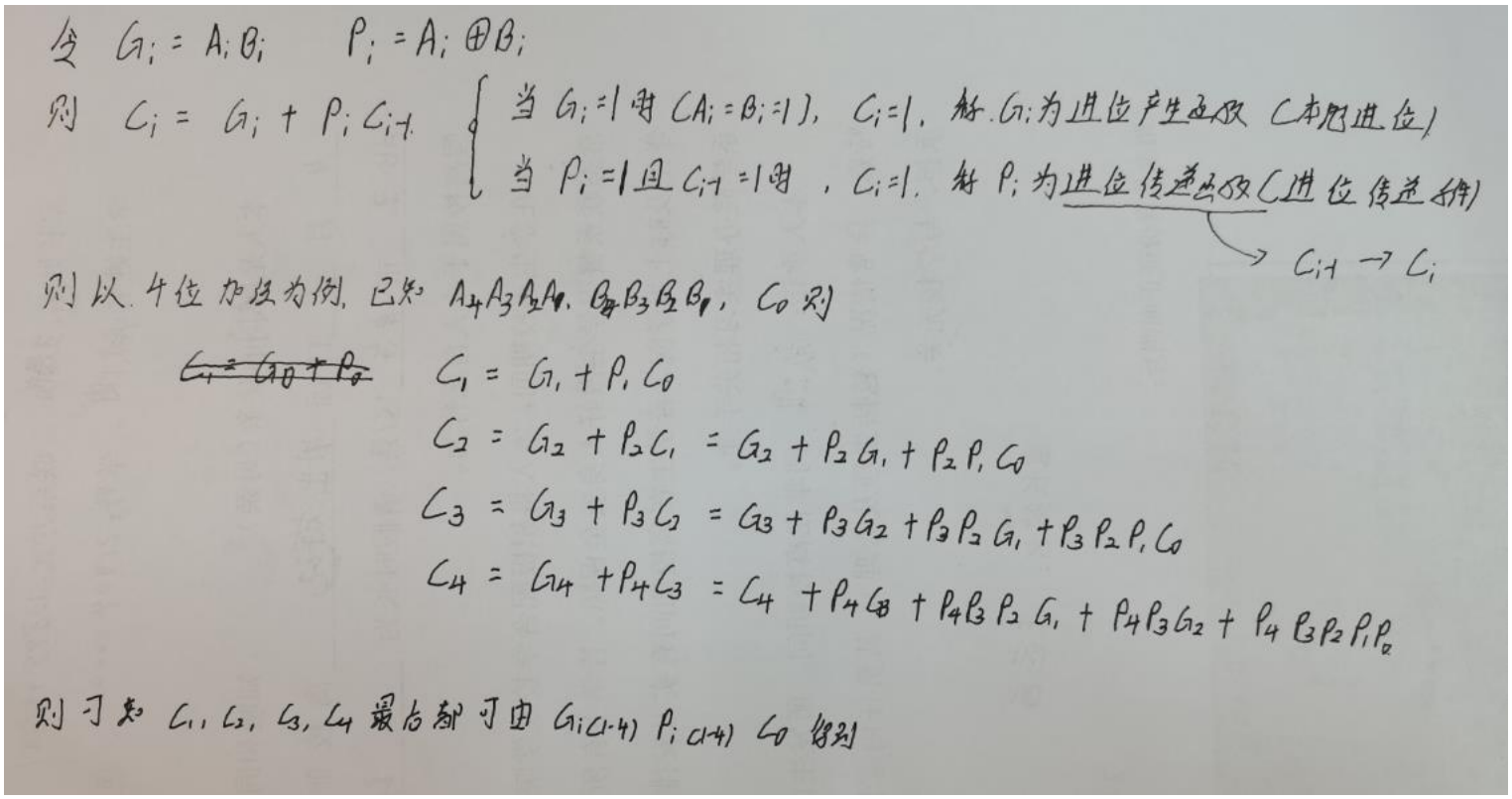
```
module full_adder4(
    input [3:0]   a ,   //adder1
    input [3:0]   b ,   //adder2
    input         c ,   //input carry bit

    output [3:0]  so ,  //adding result
    output        co    //output carry bit
    );

    wire [3:0]    co_temp ;
    //第一个例化模块一般格式有所差异，需要单独例化
```

```
    full_adder1  u_adder0(
        .Ai     (a[0]),
        .Bi     (b[0]),
        .Ci     (c),
        .So     (so[0]),
        .Co     (co_temp[0]));

    //generate 模块内的例化：每个实例变量用之前的变量
    genvar        i ;
    generate
        for(i=1; i<=3; i=i+1) begin: adder_gen
        full_adder1  u_adder(
            .Ai     (a[i]),
            .Bi     (b[i]),
            .Ci     (co_temp[i-1]), //上一个全加器的溢位是下一个的进位
            .So     (so[i]),
            .Co     (co_temp[i]));
        end
    endgenerate

    assign co    = co_temp[3] ;

endmodule
```

Verilog

超前进位全加器 CLA

利用下述的原理并行进位加法器



- 设计代码

```
`timescale 1ns / 1ps
//4 位 CLA 全加器
module cla_alu(
    input [3:0]A,
    input [3:0]B,
    input C0,
    output [4:0]S,
    output C
);
    wire [3:0]C_temp;
    wire [3:0]G;
    wire [3:0]P;
    genvar i;
    generate for ( i= 0; i<=3; i=i+1) begin:GP_Generate
        assign G[i]=A[i]&B[i];
        assign P[i]=A[i]^B[i];
```

```
    end
    endgenerate
    assign C_temp[0]=G[0]|(P[0]&C0);
    assign C_temp[1]=G[1]|(P[1]&G[0])|(P[1]&P[0]&C0);
    assign C_temp[2]=G[2]|(P[2]&G[1])|(P[2]&P[1]&G[0])|(P[2]&P[1]&P[0]&C0);
    assign
C_temp[3]=G[3]|(P[3]&G[2])|(P[3]&P[2]&P[1]&G[0])|(P[3]&P[2]&G[1])|(P[3]&P[2]&P[1]&P[0]&C0);
    assign C=C_temp[3];
    assign S[0]=A[0]^B[0]^C0;
    assign S[4]=C_temp[3];
    genvar j;
    generate for ( j= 1; j<=3; j=j+1) begin
        assign S[j]=A[j]^B[j]^C_temp[j-1];
    end

    endgenerate
endmodule
```

Verilog

- 测试代码

```
`timescale 1ns / 1ps

module cla_alu_test();

    reg [3:0]A,B;
    reg C0;
    wire [4:0]res;
    wire C;

    initial begin
        A=4'b0000;
        B=4'b1111;
```

```
        C0=0;
        forever begin
            #10;
            A=A+3;
            B=B-1;
            C0=~C0;
        end
    end

    cla_alu cla_test(.A(A),.B(B),.C0(C0),.S(res),.C(C));

    always @(*) begin
        if ($time>=1000) begin
            $finish;
        end
    end
endmodule
```

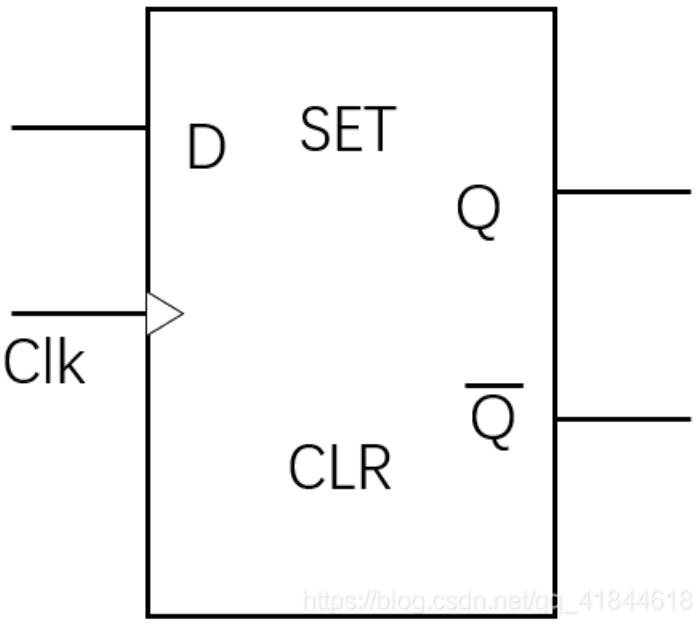
Verilog

D 触发器

D 触发器之后用于流水线 MIPS 中的流水段之间数据的传递

D 触发器均有有 clk 和复位功能，此外有的还支持 clear 和 enable，因此有 flopr,floprc,flopenr,flopenrc 四种

传统的 D 触发器



在 CLK 的上升沿或者下降沿（2 选 1），将 D 端输入送至 Q，将非 D 送至非 Q

此外还有带使能的 D 触发器，CLK 有效且使能有效才会打通 D 触发器

- 触发器的 setup 延迟[注释1]、hold 延迟[注释2]和 Clock-to-Q[注释3]延迟

  https://zhuanlan.zhihu.com/p/129850342 书签：深入理解 setup time 和 hold time

- 设计代码

```
`timescale 1ns / 1ps

module dflipflop(
    input [31:0]D,
    input clk,
    input rst,
    input en,
    output reg [31:0]P,
    output reg [31:0]_P
);
    always @(posedge clk) begin
        if(rst)begin//rst 高电平复位
            P<=32'b0;
            _P<=32'hffff_ffff;
        end
        else if(en)begin
            P<=D;
            _P<=~D;
```

```
        end
        else begin
        end
    end
endmodule
```

Verilog

- 测试代码

```
`timescale 1ns / 1ps

module dflipflop_test();
    reg [31:0] D;
    reg clk,rst,en;
    wire [31:0]P,_P;

    initial begin
        D=32'b1;
        clk=0;
        rst=0;
        en=0;
        forever begin
            clk=~clk;
            en=~en;
            #10;
            D=D<<1;//1~2~4~8~16...
            if($time%20==0)begin//每 20ns 复位一次，持续 20ns
                rst=~rst;
            end
        end
    end

    dflipflop dflipflop_test1(.D(D),.clk(clk),.rst(rst),.en(en),.P(P),._P(_P));
```

```
    always @(*) begin
        if ($time>=1000) begin
            #10;
            $finish;
        end
    end
endmodule
```

Verilog

用于流水线的 flop 系列触发器

设计代码中的时序逻辑采用非阻塞赋值<=

- flopr

```
`timescale 1ns / 1ps

module flopr #(parameter  WIDTH=32)(
    input clk,rst,
    input [WIDTH-1:0]d,
    output reg[WIDTH-1:0]q
);
    always @(posedge clk) begin
        if (rst) begin//rst 高电平复位
            q<=0;
        end else begin
            q<=d;
        end
    end
endmodule
```

Verilog

- floprc

```
`timescale 1ns/1ps

module floprc #(parameter  WIDTH=32)(
    input clk,rst,clear,
    input [WIDTH-1:0]d,
    output reg[WIDTH-1:0]q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else if (clear) begin
            q<=0;
        end else begin
            q<=d;
        end
    end
endmodule
```

Verilog

- flopenr

```
`timescale 1ns/1ps

module flopenr #(parameter  WIDTH=32)(
    input clk,rst,enable,
    input [WIDTH-1:0]d,
    output reg[WIDTH-1:0]q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else if (enable) begin
            q<=d;
```

```
        end
    end
endmodule
```

Verilog

- flopenrc

```
`timescale 1ns/1ps

module flopenrc #(parameter  WIDTH=32)(
    input clk,rst,clear,enable,
    input [WIDTH-1:0]d,
    output reg[WIDTH-1:0]q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else if (clear) begin
            q<=0;
        end else if (enable) begin
            q<=d;
        end
    end
endmodule
```

Verilog

寄存器堆

✍MIPS CPU 要求寄存器堆中有 32 个寄存器，其中 0 号寄存器[注释4] 始终为 0；寄存器堆有 2 个读端口，1 个写端口；写使能且一般在 clk 上升沿[1]，读 Always Enabled

```
module regfile(
    input         clk,
    // READ PORT 1
```

```
    input  [ 4:0] raddr1,
    output [31:0] rdata1,
    // READ PORT 2
    input  [ 4:0] raddr2,
    output [31:0] rdata2,
    // WRITE PORT
    input         we,       //write enable, HIGH valid
    input  [ 4:0] waddr,
    input  [31:0] wdata
);
reg [31:0] rf[31:0];

//WRITE
always @(posedge clk) begin//下降沿写
    if (we) begin
        rf[waddr] = wdata;//zero 寄存器可以写，但不管写入什么，始终返回 0
    end
end

//READ OUT 1
assign rdata1 = (raddr1==5'b0) ? 32'b0 : rf[raddr1];//zero 寄存器不管写入什么，始终返回 0

//READ OUT 2
assign rdata2 = (raddr2==5'b0) ? 32'b0 : rf[raddr2];//zero 寄存器不管写入什么，始终返回 0

endmodule
```

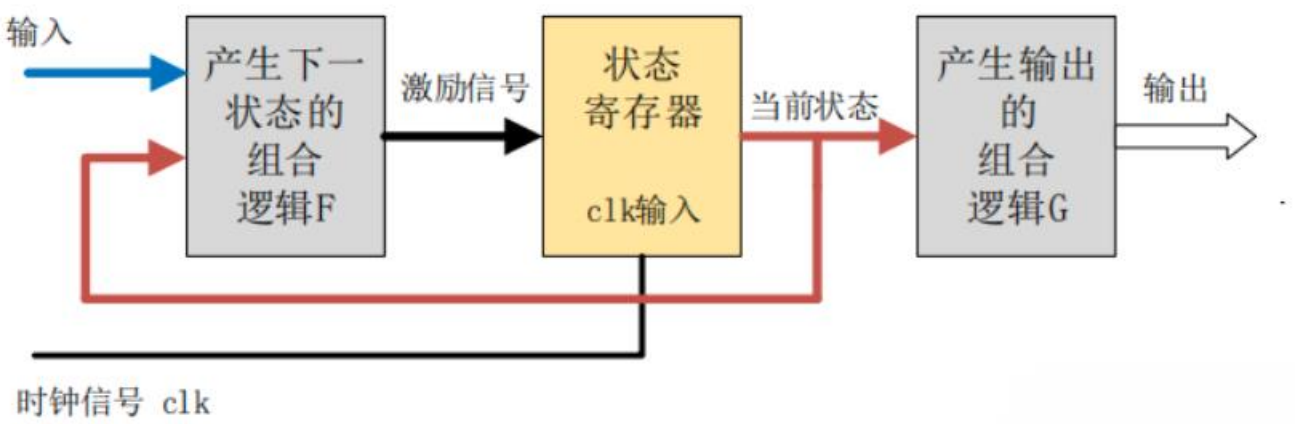
Verilog

状态机

状态机的类型

Verilog 中**状态的切换方向不但取决于各个输入值，还取决于当前的所在状态**

- Moore 型状态机

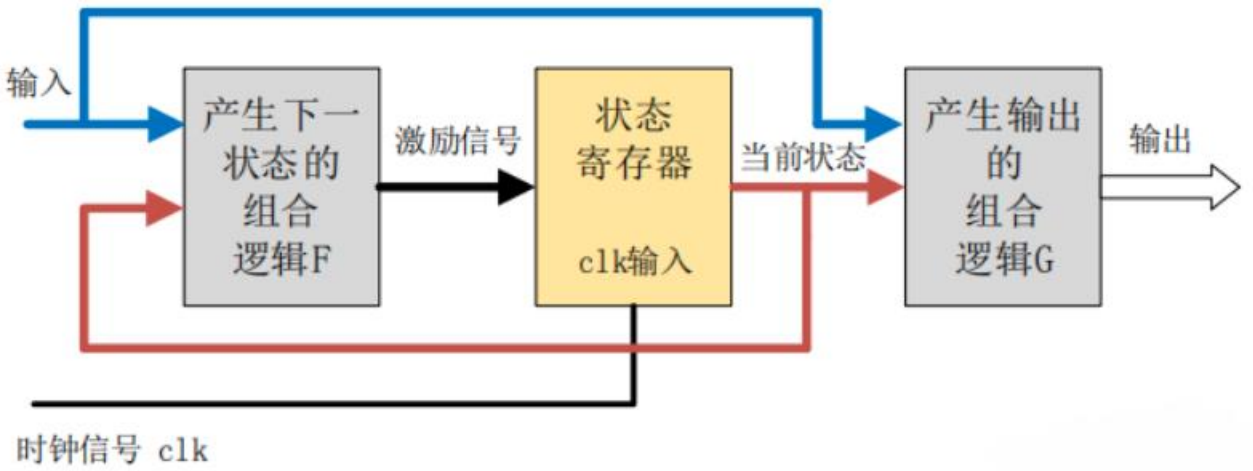


Moore 型状态机的**输出只与当前状态有关，与当前输入无关**

输出会在一个完整的时钟周期内保持稳定，即使此时输入信号有变化，输出也不会变化

输入对输出的影响要到下一个时钟周期才能反映出来。这也是 Moore 型状态机的一个重要特点：**输入与输出是隔离开来的。**

- Mealy 型状态机



Mealy 型状态机的输出，**不仅与当前状态有关，还取决于当前的输入信号**

Mealy 型状态机的**输出是在输入信号变化以后立刻发生变化**，且输入变化可能出现在任何状态的时钟周期内。因此，同种逻辑下，Mealy 型状态机输出对输入的响应会比 Moore 型状态机早一个时钟周期

状态机的设计流程

根据设计需求画出**状态转移图**，确定使用状态机类型，并标注出各种输入输出信号，更有助于编程。**一般使用最多的是 Mealy 型 3 段式状态机**

状态机设计如下：

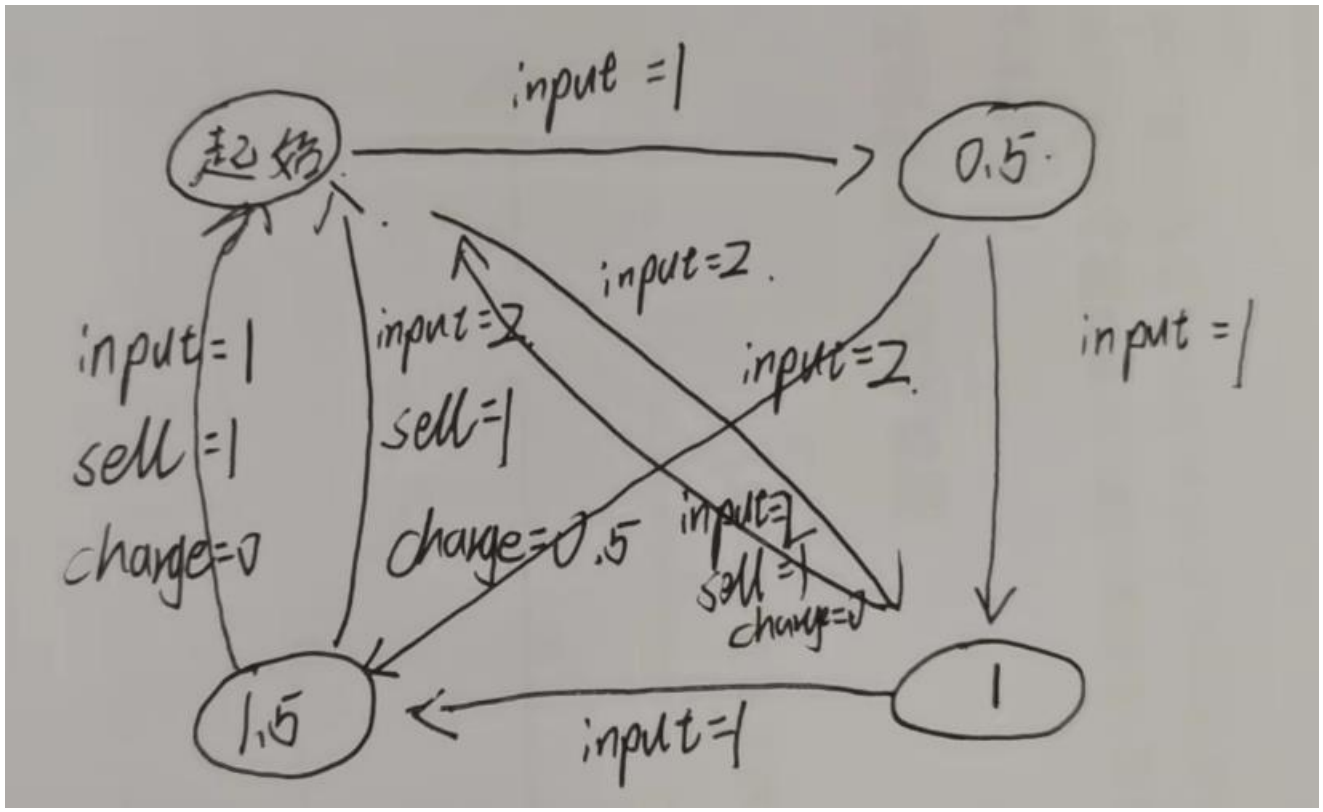
- 首先，根据状态机的个数确定状态机编码。利用编码给状态寄存器赋值，代码可读性更好
- 状态机第一段，**时序逻辑，非阻塞赋值<=，传递寄存器的状态**
- 状态机第二段，**组合逻辑，阻塞赋值=，根据当前状态和当前输入，确定下一个状态机的状态**
- 状态机第三代，**时序逻辑，非阻塞赋值<=，因为是 Mealy 型状态机，根据当前状态和当前输入，确定输出信号**

自动售卖机

饮料单价 2 元，该售卖机只能接受 0.5 元、1 元的硬币。考虑找零和出货。投币和出货过程都是一次一次的进行，不会出现一次性投入多币或一次性出货多瓶饮料的现象。每一轮售卖机接受投币、出货、找零完成后，才能进入到新的自动售卖状态

- 题目分析

设置状态的输入是 input，input=1 表示输入 0.5，input=2 表示输入 1



则共有 4 个状态，用 00、01、10、11，输入是 input，输出是 sell 和 charge

设置两个 reg 变量，一个表示当前状态，一个表示下一阶段状态

状态机的第一段：时序非阻塞赋值，传递寄存器的状态——有复位则是初始状态，无复位则将下一阶段的状态传递到当前阶段

状态机的第二段：组合阻塞赋值，根据当前的寄存器状态和当前的输入来确定下一阶段寄存器的状态

状态机的第三段：时序非阻塞赋值，根据当前的寄存器状态和当前的输入确定输出

- 设计代码

```
`timescale 1ns / 1ps

module vending_machines(
    input clk,
    input rst,
    input coin,//0 表示付 0.5，1 表示付 1
    output reg charge,//0 表示不用找零，1 表示找 0.5
    output reg sell
);
reg [1:0] cur_state=2'b00;
reg [1:0] next_state=2'b00;

//第一段，时序逻辑的非阻塞赋值
always @(posedge clk or negedge rst) begin//clk 上升沿传递状态，rst 下降沿复位
    if (rst) begin
        cur_state<=2'b00;
    end
    else begin
        cur_state<=next_state;
    end
end

//第二段，组合逻辑的阻塞赋值
always @(*) begin
    case (coin)
        0:begin
            case (cur_state)
                2'b00:next_state=2'b01;
                2'b01:next_state=2'b10;
```

```
                2'b10:next_state=2'b11;
                default: next_state=2'b00;
            endcase
        end
        1: begin
            case (cur_state)
                2'b00:next_state=2'b10;
                2'b01:next_state=2'b11;
                2'b10:next_state=2'b00;
                default: next_state=2'b00;
            endcase
        end
        default:
            cur_state=2'bx;
    endcase
end

//第三段，时序逻辑设置输出
always @(posedge clk or negedge rst) begin
    if(rst)begin
        charge<=0;
        sell<=0;
    end
    else begin
        case (cur_state)
            2'b00,
            2'b01:begin//多个同样的，用，隔开
                if (coin==0) begin
                    charge<=0;
                    sell<=0;
                end
                else begin
```

```
                charge<=0;
                sell<=0;
            end
        end
        2'b10:begin
            if (coin==0) begin
                charge<=0;
                sell<=0;
            end
            else begin
                charge<=0;
                sell<=1;
            end
        end
        default: begin
            if (coin==0) begin
                charge<=0;
                sell<=1;
            end
            else begin
                charge<=1;//找零
                sell<=1;
            end
        end
    endcase
  end
end
endmodule
```

Verilog

- 测试代码

```verilog
`timescale 1ns / 1ps

module test_vm();
    reg clk;
    reg rst;
    reg coin;
    wire charge;
    wire sell;

    reg [7:0] in=8'b01000000;//二进制右移代表投递：0.5 0.5 0.5 0.5...
    initial begin
        clk=0;
        rst=0;
        coin=0;
        repeat (8)begin
            clk=~clk;
            coin=in[0];
            in=in>>1;
            #10;
        end
    end
    vending_machines vending_machines_test(clk,rst,coin,charge,sell);
    always @(*) begin
        if($time>=80) begin
            #10;
            $finish;
        end
    end
endmodule
```

Verilog

移位寄存器

实现循环左移、非循环左移、循环右移、非循环右移这几种移位寄存器，此外还有串行输入并行输出移位寄存器和并行输入串行输出移位寄存器

- 循环左移

$$data[n-1:0]rightarrow\{data[n-2:0],data[n-1]\}$$

- 非循环左移

$$data[n-1:0]rightarrow\{data[n-2:0],0\}$$

- 循环右移

$$data[n-1:0]rightarrow\{data[0],data[n-1:1]\}$$

- 非循环右移 shr

$$data[n-1:0]rightarrow\{0,data[n-1:1]\}$$

- 串入并出

设定数据位宽 WIDTH，输入方向 con_dir（0 左 1 右）

设定整数变量 i=0，当 i 值为 WIDTH 时，输出数据 data_out（用 data_temp 暂存数据）

而 con_dir 决定的是输入的位数据如何存放：

condir=0，$data\ \ temp[WIDTH-1:0]leftarrow\{data\ \ temp[WIDTH-2:0],data\ \ in\}$

condir=1，$data\ \ temp[WIDTH-1:0]leftarrow\{data\ \ in,data\ \ temp[WIDTH-1:1],\}$

- 并入串出

设定数据位宽 WIDTH，输入方向 con_dir（0 左 1 右），data_temp 暂存输入数据

con_dir=0，$data\ \ temp[WIDTH-1:0]leftarrow\{data\ \ temp[WIDTH-2:0],0\}$

con_dir=1，$data\ \ temp[WIDTH-1:0]leftarrow\{0,data\ \ temp[WIDTH-1:1]\}$

输出 data_out 用条件选择符决定：

$$data\ \ out=(con\ \ dir)?data\ \ temp[0]:data\ \ temp[WIDTH-1]$$

```
`timescale 1ns / 1ps


 //设置带参数的模块，模块名#(parameter 参数=值)(端口);
module cycle_leftShift #(parameter WIDTH = 4)(
    input clk,
    input rst,
    input [WIDTH-1:0] d_in,
    input en,
```

```
    output reg[WIDTH-1:0] d_out
);
    always @(posedge clk or negedge rst) begin
        if (rst) begin
            d_out=0;
        end
        else begin
            if(en) begin
                d_out=d_in;
            end
            else begin
                d_out={d_out[WIDTH-2:0],d_out[WIDTH-1]};
            end
        end
    end
endmodule
```

```
module leftShift #(parameter WIDTH = 4)(
    input clk,
    input rst,
    input [WIDTH-1:0] d_in,
    input en,
    output reg[WIDTH-1:0] d_out
);

    always @(posedge clk or negedge rst) begin
        if (rst) begin
            d_out=0;
        end
        else begin
            if(en) begin
                d_out=d_in;
            end
```

```
            else begin
                d_out={d_out[WIDTH-2:0],1'b0};
            end
        end
    end
endmodule

module cycle_rightShift #(parameter WIDTH = 4)(
    input clk,
    input rst,
    input [WIDTH-1:0] d_in,
    input en,
    output reg[WIDTH-1:0] d_out
);
    always @(posedge clk or negedge rst) begin
        if (rst) begin
            d_out=0;
        end
        else begin
            if(en) begin
                d_out=d_in;
            end
            else begin
                d_out={d_out[0],d_out[WIDTH-1:1]};
            end
        end
    end
endmodule

module rightShift #(parameter WIDTH = 4)(
    input clk,
    input rst,
    input [WIDTH-1:0] d_in,
```

```
    input en,
    output reg[WIDTH-1:0] d_out
);

    always @(posedge clk or negedge rst) begin
        if (rst) begin
            d_out=0;
        end
        else begin
            if(en) begin
                d_out=d_in;
            end
            else begin
                d_out={1'b0,d_out[WIDTH-1:1]};
            end
        end
    end
endmodule

module pIn_sOut #(parameter WIDTH=4)(
    input clk,
    input rst,
    input con_dir,//0左1右
    input en,
    input [WIDTH-1:0] d_in,
    output  d_out
);
    reg [WIDTH-1:0] d_temp;
    always @(posedge clk or negedge rst) begin
        if (rst) begin
            d_temp=0;
        end
        else begin
```

```
            if(en) begin
                d_temp=d_in;
            end
            else begin
                if (con_dir) begin//右移
                    d_temp={1'b0,d_temp[WIDTH-1:1]};
                end
                else begin
                    d_temp={d_temp[WIDTH-2:0],1'b0};
                end
            end
        end
    end
    assign d_out=(con_dir)?d_temp[0]:d_temp[WIDTH-1];
endmodule

module sIn_pOut #(parameter WIDTH=4)(
    input clk,
    input rst,
    input con_dir,//0左1右
    input en,
    input d_in,
    output reg[WIDTH-1:0] d_out
);
reg [WIDTH-1:0] d_temp=0;
integer i=0;
always @(posedge clk or negedge rst) begin
   if (rst) begin
        d_temp=0;
   end
   else begin
        if (en) begin
```

```
            if (con_dir) begin
                d_temp={1'b0,d_temp[WIDTH-1:1]};
                d_temp[WIDTH-1]=d_in;
                i=i+1;
            end
            else begin
                d_temp={d_temp[WIDTH-2:0],1'b0};
                d_temp[0]=d_in;
                i=i+1;
            end
        end
        if (i==WIDTH) begin
            d_out=d_temp;
        end
    end
end
endmodule
```

Verilog

计数器

每次 clk 上升沿计数一次，记到 n 时重新从 0 开始：0~n-1 的 n 进制计数

```
`timescale 1ns / 1ps


//N 进制计数 0~N-1
module count_design #(parameter N=10,WIDTH=$clog2(N)+1)(
    input clk,
    input rst,
    output reg [WIDTH-1:0]d_out
);
    always @(posedge clk or negedge rst) begin
        if (rst) begin
```

```
            d_out=0;
        end
        else begin
            if(d_out==N-1)begin
                d_out=0;
            end
            else begin
                d_out=d_out+1;
            end
        end
    end
endmodule
```

Verilog

乘法器

常数的乘法可以用移位相加来实现

移 n 位是乘以2^n，基本的构造方式是将常数拆分成若干个被乘数的和，从而利用加法结合律产生出若干个移位后的被乘数的相加，也可以用多的相减

```
1*A = A
2*A=A<<1
4*A=A<<2
8*A=A<<3...
```

Verilog

乘法器的设计原理

和十进制乘法类似，计算 13 与 5 的相乘过程如下所示：

```
        1 1 0 1  (13)
      x   1 0 1  (5)
 ---------------
        1 1 0 1
      0 0 0 0
    1 1 0 1
---------------
  1 0 0 0 0 0 1  (65)
```

由此可知，被乘数按照乘数对应 bit 位进行移位累加，便可完成相乘的过程。

假设每个周期只能完成一次累加，那么一次乘法计算时间最少的时钟数恰好是乘数的位宽。所以建议，将位宽窄的数当做乘数，此时计算周期短。

无符号数乘法时，结果变量位宽应该为 2 个操作数位宽之和

✍即需要有乘数个“被乘数所左移得到的数”相加，每个加数的位宽是（被乘数位宽+乘数位宽-1）

- 设计代码

```
`timescale 1ns / 1ps

module multiply_design #(parameter M=4,N=4)(//M 是被乘数的位宽，N 是乘数的位宽
    input clk,
    input rst,
    input [M-1:0]mul_a,
    input [N-1:0]mul_b,
    output reg isOut,//通知测试文件，计算完了，可以进行下一个输入
    output reg[M+N-1:0]res
  );
  wire  [M+N-1:0] add_temp[N-1:0];//存储每一个乘数位所对应的加数
  reg [M+N-1:0]res_temp=0;//存储累加结果，当加完所有加数时，将值赋为 res
  genvar i;

  generate for ( i= 0; i<N; i=i+1)
    begin
      assign add_temp[i]=(mul_b[i]==1)?mul_a<<i:0;
```

```
    end
endgenerate

integer j=0;
always @(posedge clk or negedge rst)
begin:begin_clock
  if(~rst)//rst 低电平复位
  begin
    res_temp=0;
    res=0;
    isOut=0;
    j=0;
  end
  else begin:add_to_res
       res_temp=res_temp+add_temp[j];
       j=j+1;
       if (j==N) begin
          res=res_temp;
          isOut=1;
       end
    end
  end
endmodule
```

Verilog

- 测试代码

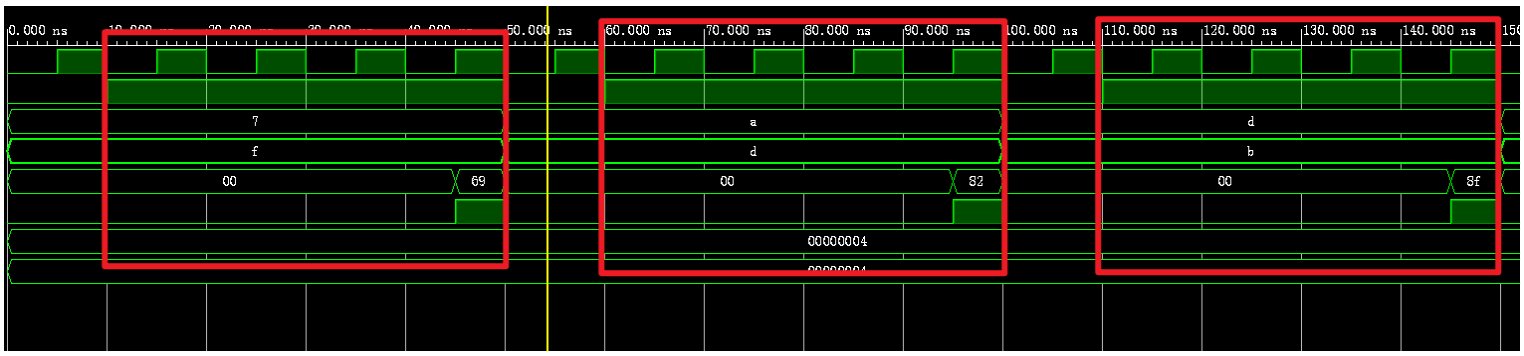
```
`timescale 1ns / 1ps

module multiply_design_tb;

  // Parameters
  localparam  M = 4;
```

```
localparam  N = 4;

//Ports
reg  clk;
reg  rst;
reg [M-1:0] mul_a;
reg [N-1:0] mul_b;
wire [M+N-1:0] res;
wire isOut;

initial begin
  clk=0;
  rst=0;
  mul_a=7;
  mul_b=15;
  #10;
  rst=1;
end
always @(*) begin
  if (isOut==1) begin
      #5;
      rst=0;
      mul_a=mul_a+3;
      mul_b=mul_b-2;
      #10;
      rst=1;
  end
end
multiply_design # (
                  .M(M),
                  .N(N)
                )
                multiply_design_inst (
```

```
                    .clk(clk),
                    .rst(rst),
                    .mul_a(mul_a),
                    .mul_b(mul_b),
                    .res(res),
                    .isOut(isOut)
                );
  always #5 clk=~clk;
endmodule
```

Verilog

乘数 N 位是需要加 N 步，即一个结果输出需要 N 个周期，如下图



下面实现流水线乘法器，实现一个周期输出一个结果（除去流水线的通过时间+排出时间）

流水线乘法器

- 流水线设计思想

  ✍**需要一个顶层模块，顶层模块的输入输出和第一个流水子模块一致**

  有几个流水段就设计几个子模块

  这里以 3 个子模块 a, b, c 为例

  顶层模块的输入给子模块 a

  子模块 a 的输出给子模块 b

  子模块 b 的输出给子模块 c

  子模块 c 的输出给顶层模块的输出

```
module pipline(
    input clk,
    input rst_n,
    input [7:0] data_in,
    input in_en,
    output out_en,
    output [7:0] data_out
    );

    wire [7:0] data_a,data_b,data_c;
    wire in_en_a,in_en_b,in_en_c;
    wire out_en_a,out_en_b,out_en_c;
    assign in_en_a = in_en;
    assign in_en_b = out_en_a;
    assign in_en_c = out_en_b;

    sequence_A sq_a(.clk(clk), .rst_n(rst_n), .data_in(data_in), .in_en(in_en_a), .out_en(out_en_a), .data_out(data_a));
    sequence_B sq_b(.clk(clk), .rst_n(rst_n), .data_in(data_a ), .in_en(in_en_b), .out_en(out_en_b), .data_out(data_b));
    sequence_C sq_c(.clk(clk), .rst_n(rst_n), .data_in(data_b ), .in_en(in_en_c), .out_en(out_en_c), .data_out(data_c));

    assign out_en = out_en_c;
    assign data_out = data_c;

endmodule
```

$y=(x_1+x_2+x_3+x_4)\div 4+15$

$m=x_1+x_2+x_3+x_4$

$n=m\div 4$

$y=n+15$

- 流水线乘法器的设计思想

乘数有多少位就有多少段流水，声明 res_temp[M+N-1]

流水段输入的操作数 A 位宽是（M+N-1）位，B 位宽是 1 位，res 位宽是（M+N-1）位

第一级流水线：输入 A 是顶层输入的被乘数（高 N 位补 0），B 是乘数，res 是 res_temp

流水线中 res_temp=res_temp+A*B[0]，然后 A<<1，B>>1，若移位后 B 为 0 则运算结束

之后的流水线，输入 A 是上一级流水线的左移输出 A，B 是上一级输出的乘数，res 是 res_temp

最后赋值 out_en 和 res

- 设计代码
  - 流水段

```
`timescale 1ns / 1ps
module multiplypipe_design #(parameter  M=4,N=4)(
    input clk,
    input rst,
    input [M+N-1:0]mul_a,
    input [N-1:0]mul_b,
    input [M+N-1:0]cur_res,
```

```Verilog
    output reg[M+N-1:0]mul_a_next,
    output reg[N-1:0]mul_b_next,
    output reg out_en,
    output reg[M+N-1:0]res
);
    always @(posedge clk or negedge rst) begin
        if (~rst) begin
            out_en=0;
            res=0;
        end
        else begin
            res=(mul_b[0]==1)?(cur_res+mul_a):cur_res;
            mul_a_next=mul_a<<1;
            mul_b_next=mul_b>>1;
            if (mul_b_next==0) begin//这里也可以计数管理，判断 0 可以提前终止
                out_en=1;
            end
        end
    end
endmodule
```

● 顶层模块

```Verilog
`timescale 1ns / 1ps
module multiplypipe_top #(parameter M =4 ,N=4)(
    input clk,
    input rst,
    input [M-1:0]mul_a,
    input [N-1:0]mul_b,
    output out_en,
    output [M+N-1:0]res
);
```

```
    wire [M+N-1:0] mul_a_next[N-1:0];
    wire [N-1:0] mul_b_next[N-1:0];
    wire [M+N-1:0]res_next[N-1:0];
    wire out_next[N-1:0];

    multiplypipe_design # (
        .M(M),
        .N(N)
    )
    multiplypipe_design_1 (
        .clk(clk),
        .rst(rst),
        .mul_a({{N{0}},mul_a}),
        .mul_b(mul_b),
        .cur_res({M+N-1{0}}),
        .mul_a_next(mul_a_next[0]),
        .mul_b_next(mul_b_next[0]),
        .out_en(out_next[0]),
        .res(res_next[0])
    );
    genvar i;
    generate for ( i= 1; i<N; i=i+1) begin
        multiplypipe_design # (
            .M(M),
            .N(N)
        )
        multiplypipe_design_next (
            .clk(clk),
            .rst(rst),
            .mul_a(mul_a_next[i-1]),
            .mul_b(mul_b_next[i-1]),
            .cur_res(res_next[i-1]),
            .mul_a_next(mul_a_next[i]),
```

```
            .mul_b_next(mul_b_next[i]),
            .out_en(out_next[i]),
            .res(res_next[i])
        );
    end
    endgenerate

    assign out_en=out_next[N-1];
    assign res=res_next[N-1];

endmodule
```

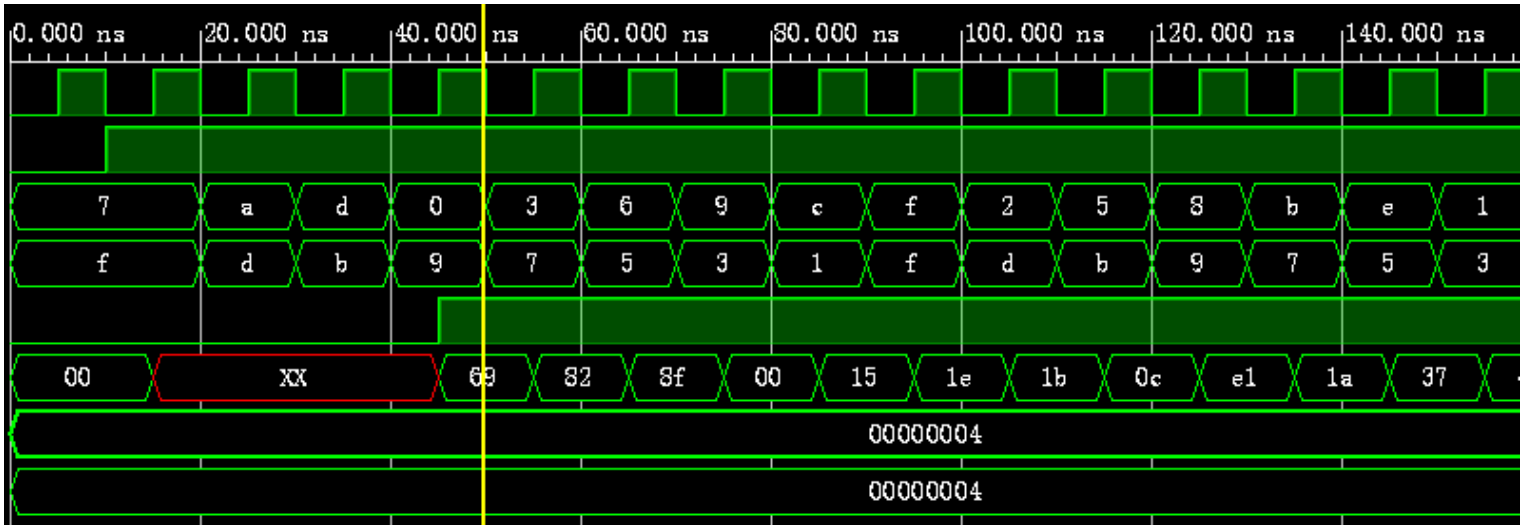
Verilog

- 测试代码

```
`timescale 1ns / 1ps

module multiply_pipe_test();
  // Parameters
  localparam  M = 4;
  localparam  N = 4;

  //Ports
  reg  clk;
  reg  rst;
  reg [M-1:0] mul_a;
  reg [N-1:0] mul_b;
  wire  out_en;
  wire [M+N-1:0] res;
  initial
  begin
    clk=0;
```

```
    rst=0;
    mul_a=7;
    mul_b=15;
    #10;
    rst=1;
    forever begin//相比无流水线的测试代码，去掉了检测 isOut 和复位的处理
        #10;
        mul_a=mul_a+3;
        mul_b=mul_b-2;
    end
end
multiplypipe_top # (
                    .M(M),
                    .N(N)
                )
                multiplypipe_top_inst (
                    .clk(clk),
                    .rst(rst),
                    .mul_a(mul_a),
                    .mul_b(mul_b),
                    .out_en(out_en),
                    .res(res)
                );

  always #5  clk = ! clk ;
endmodule
```

Verilog



✍实现带符号数的计算：

上述的例子是实现的无符号数，带符号数的计算可以当作无符号数的计算再加上一些特殊的步骤（只需要更改顶层代码）

1. 首先对操作数进行处理：正数不变，负数变成其绝对值——多了符号的传递

   流水线段需要加入各输入输出 A 和 B 符号位

   以 A 为例，若 A[N-1]=0,则 A 不处理，反之 A=~A+1

2. 按照上述的无符号数操作进行计算
3. 最后输出时再结合传递过来的符号

   输出若 A 和 B 符号位异或是 1，那么结果取反+1，反之不变

- 顶层代码

```
`timescale 1ns / 1ps
module multiplypipe_top #(parameter M =4 ,N=4)(
    input clk,
    input rst,
    input [M-1:0]mul_a,
    input [N-1:0]mul_b,
    output out_en,
    output  [M+N-1:0]res
);
//无符号：mul_a 补 0,处理 mul_b 位宽次

//有符号：先按无符号算，然后再算上符号输出
    wire [M+N-1:0] mul_a_next[N-1:0];
```

```
    wire [N-1:0] mul_b_next[N-1:0];
    wire [M+N-1:0]res_next[N-1:0];
    wire out_next[N-1:0];

    wire [M-1:0]t_mul_a=(mul_a[M-1]==0)?mul_a:~mul_a+1;
    wire [N-1:0]t_mul_b=(mul_b[N-1]==0)?mul_b:~mul_b+1;

    wire sign_a[N-1:0];
    wire sign_b[N-1:0];
    multiplypipe_design # (
        .M(M),
        .N(N)
    )
    multiplypipe_design_1 (
        .clk(clk),
        .rst(rst),
        .mul_a({{N{0}},t_mul_a}),
        .mul_b(t_mul_b),
        .cur_res({M+N-1{0}}),
        .sign_a(mul_a[M-1]),
        .sign_b(mul_b[N-1]),

        .nsign_a(sign_a[0]),
        .nsign_b(sign_b[0]),
        .mul_a_next(mul_a_next[0]),
        .mul_b_next(mul_b_next[0]),
        .out_en(out_next[0]),
        .res(res_next[0])
    );
    genvar i;
    generate for ( i= 1; i<N; i=i+1) begin
        multiplypipe_design # (
            .M(M),
```

```
            .N(N)
        )
        multiplypipe_design_next (
            .clk(clk),
            .rst(rst),
            .mul_a(mul_a_next[i-1]),
            .mul_b(mul_b_next[i-1]),
            .cur_res(res_next[i-1]),
            .sign_a(sign_a[i-1]),
            .sign_b(sign_b[i-1]),

            .nsign_a(sign_a[i]),
            .nsign_b(sign_b[i]),
            .mul_a_next(mul_a_next[i]),
            .mul_b_next(mul_b_next[i]),
            .out_en(out_next[i]),
            .res(res_next[i])
        );
    end
    endgenerate
    assign out_en=out_next[N-1];
    assign res=(sign_a[N-1]^sign_b[N-1]==1)?~res_next[N-1]+1:res_next[N-1];
endmodule
```

Verilog

除法器

除法的基本原理

基于减法的除法器的算法：

对于32的无符号除法，被除数a除以除数b，他们的商和余数一定不会超过32位。首先将a转换成高32位为0，低32位为a的temp_a。把b转换成高32位为b，低32位为0的temp_b。在每个周期开始时，先将temp_a左移一位，末尾补0，然后与b比较，是否大于b，是则temp_a减去temp_b将且加上1，否则继续往下执行。上面的移位、比较和减法（视具体情况而定）要执行32次，执行结束后temp_a的高32位即为余数，低32位即为商。

无流水线除法器的设计

- 设计思想

除法结果商变量位宽是被除数位宽，余数位宽是除数位宽

按照上述的思想，被除数 A[M-1:0],除数[N-1:0],商[M-1:0],余数[N-1:0]

先比较 M 和 N:

若 M<N 则商为 0，余数为 A

若 M≥N，则先将被除数、除数均补为 M+N 位 A_temp,B_temp

被除数是{{N{0}},被除数}，除数是{除数,{M{0}}}

然后进行计算，按下面的方法计算

首先左移一位 A_temp，然后和 B_temp 比较，若大于等于则 A_temp=A_temp-B_temp+1

否则 A_temp=A_temp

重复计算 N 步结束，此时 A_temp 的高 N 位是余数，低 M 位是商

- 设计代码

```
`timescale 1ns / 1ps
module divider_design #(parameter M=4,N=4)(
    input clk,
    input rst,

    input [M-1:0]A,
    input [N-1:0]B,
    output reg out_en,
    output reg [M-1:0]res,
    output reg [N-1:0]mod
);

reg [M+N-1:0]B_temp;
reg [M+N-1:0]A_temp;
integer i;
always @(posedge clk or negedge rst) begin:func_block
  if (~rst) begin
      i=1;
      out_en=0;
```

```
        A_temp=A;//reg 默认无符号
        if (M>=N) begin
            B_temp={B,{M{1'b0}}};//注意这里是 1'b0
        end
        res={{M{1'bx}}};
        mod={{N{1'bx}}};
    end
    else begin
        if (M<N) begin
            out_en=1;
            res=0;
            mod=A;
        end
        else begin
            if (i<=M) begin
                A_temp={A_temp[N-2:0],1'b0};
                if (A_temp[M+N-1:M]>=B_temp[M+N-1:M]) begin
                    A_temp=A_temp-B_temp+1;
                end
                i=i+1;
                if (i==M+1) begin
                    out_en=1;
                    mod=A_temp[M+N-1:M];
                    res=A_temp[M-1:0];
                end
            end
        end
    end
  end


endmodule
```

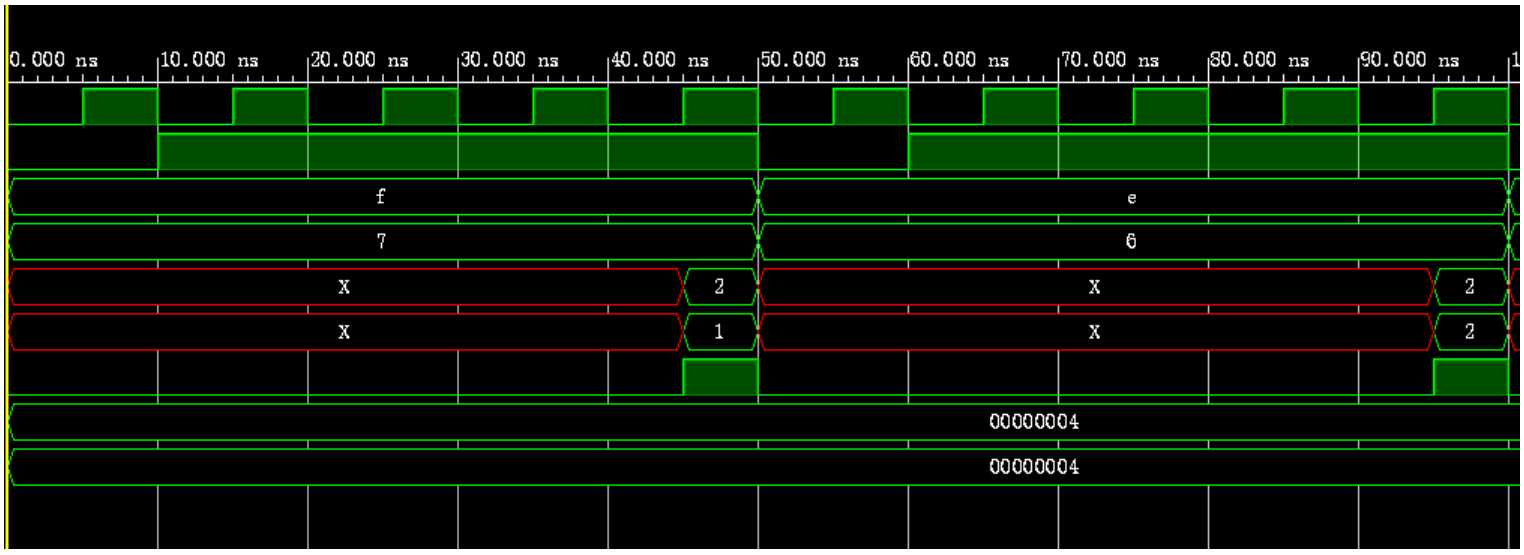
Verilog

- 测试代码

```
`timescale 1ns / 1ps


module divider_test();
  // Parameters
  localparam  M = 4;
  localparam  N = 4;

  //Ports
  reg  clk;
  reg  rst;
  reg [M-1:0] A;
  reg [N-1:0] B;

  wire [M-1:0] res;
  wire [N-1:0]mod;
  wire isOut;

  initial
  begin
    clk=0;
    rst=0;
    A=15;
    B=7;
    #10;
    rst=1;
  end
  always @(*)
  begin
```

```
    if (isOut==1)
    begin
      #5;
      rst=0;
      A=A-1;
      B=B-1;
      #10;
      rst=1;
    end
  end
  divider_design # (
                  .M(M),
                  .N(N)
                )
                divider_design_inst (
                  .clk(clk),
                  .rst(rst),

                  .A(A),
                  .B(B),

                  .res(res),
                  .mod(mod),
                  .out_en(isOut)
                );
    always #5 clk=~clk;
endmodule
```

Verilog



- 有符号数的除法——思想与乘法器处理一致

将输入的被除数和除数取其绝对值再扩充 M+N 位，最后的结果结合符号位处理：**商符号取被除数符号和除数符号异或，余数符号和被除数符号一致**

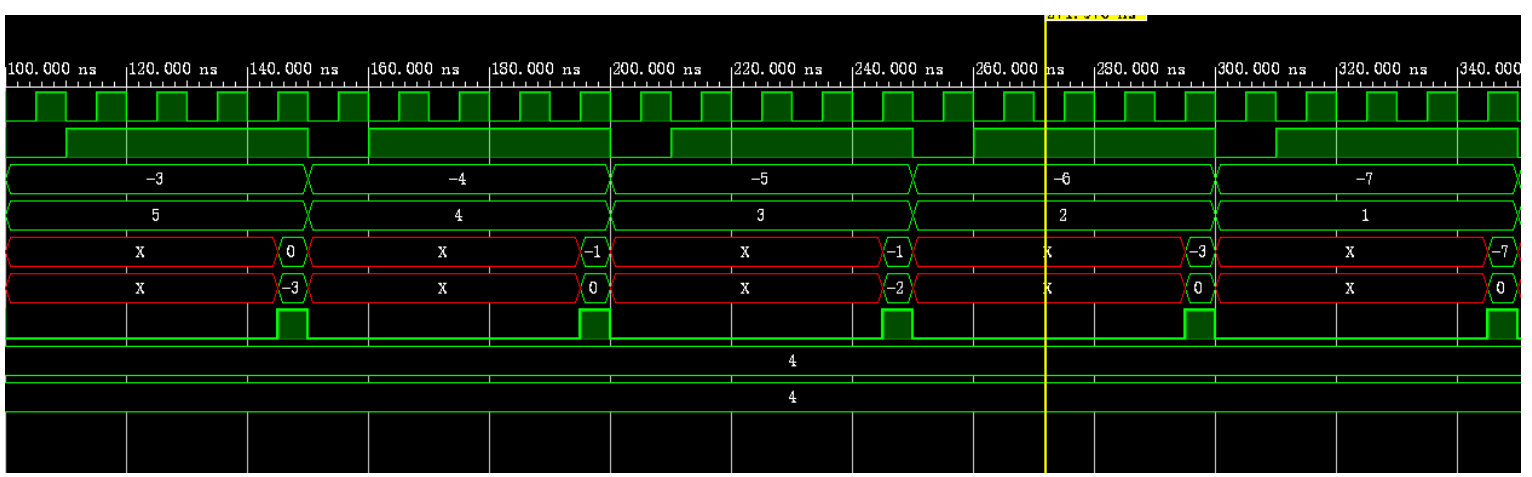
顶层代码如下：

```
`timescale 1ns / 1ps
module divider_design #(parameter M=4,N=4)(
  input clk,
  input rst,

  input [M-1:0]A,
  input [N-1:0]B,
  output reg out_en,
  output reg [M-1:0]res,
  output reg [N-1:0]mod
);
reg [M-1:0]A_;
reg [N-1:0]B_;
reg [M+N-1:0]B_temp;
reg [M+N-1:0]A_temp;
integer i;
always @(posedge clk or negedge rst) begin:func_block
  if (~rst) begin
```

```
        i=1;
        out_en=0;
        A_=(A[M-1])?~A+1:A;
        A_temp=A_;//reg 默认无符号
        if (M>=N) begin
            B_=(B[N-1])?~B+1:B;
            B_temp={B_,{M{1'b0}}};//这里是 1'b0
        end
        res={{M{1'bx}}};
        mod={{N{1'bx}}};
    end
    else begin
        if (M<N) begin
            out_en=1;
            res=0;
            mod=A;
        end
        else begin
            if (i<=M) begin
                A_temp=A_temp<<1;
                if (A_temp[M+N-1:M]>=B_temp[M+N-1:M]) begin
                    A_temp=A_temp-B_temp+1;
                end
                i=i+1;
                if (i==M+1) begin
                    out_en=1;
                    mod=(A[M-1]==A_temp[M+N-1])?A_temp[M+N-1:M]:~A_temp[M+N-1:M]+1;
                    res=(A[M-1]^B[N-1]==0)?A_temp[M-1:0]:~A_temp[M-1:0]+1;
                end
            end
        end
    end
```

```
    end


endmodule
```

Verilog



有流水线的除法器设计

- 设计思想

  N 位除数则有 N 级流水线

  每一级流水输入 A_temp[M+N-1],B_temp[M+N-1],输出 A_next,B_next,计数 i，当 i 计数到 N 时，则 out_en 有效进行输出

- 设计代码

  流水段：

```
`timescale 1ns / 1ps

module dividerpipe_design #(parameter  M=4,N=4)(
    input clk,
    input rst,

    input [M+N-1:0]A,
    input [M+N-1:0]B,
    input i,

    output reg [M+N-1:0] A_next,
```

```
    output reg [M+N-1:0] B_next,
    output reg out_en
);
    wire next_i=i+1;

    always @(posedge clk or negedge rst) begin
        if (~rst) begin
            out_en=0;
        end
        else begin
            B_next=B;
            A_next=A<<1;
            if (A_next[M+N-1:M]>=B[M+N-1:M]) begin
                A_next=A_next-B_next+1;
            end
            if (next_i==N) begin
                out_en=1;
            end
        end
    end
endmodule
```

Verilog

顶层：

```
`timescale 1ns / 1ps


module dividerpipe_top#(parameter M=4,N=4)(
    input clk,
    input rst,

    input [M-1:0]A,
```

```
    input [N-1:0]B,
    output  out_en,
    output [M-1:0]res,
    output [N-1:0]mod
);
    wire [M+N-1:0]B_temp={B,{M{1'b0}}};
    wire [M+N-1:0]A_temp=A;

    wire out_temp[N-1:0];
    wire [M+N-1:0]next_A_temp[N-1:0];
    wire [M+N-1:0]next_B_temp[N-1:0];

    dividerpipe_design # (
                        .M(M),
                        .N(N)
                    )
                    dividerpipe_design_inst (
                        .clk(clk),
                        .rst(rst),
                        .A(A_temp),
                        .B(B_temp),
                        .i(0),
                        .A_next(next_A_temp[0]),
                        .B_next(next_B_temp[0]),
                        .out_en(out_temp[0])
                    );
genvar i;
generate for ( i= 1; i<N; i=i+1) begin
    dividerpipe_design # (
        .M(M),
        .N(N)
      )
      dividerpipe_design_pipe (
```

```
            .clk(clk),
            .rst(rst),
            .A(next_A_temp[i-1]),
            .B(next_B_temp[i-1]),
            .i(i),
            .A_next(next_A_temp[i]),
            .B_next(next_B_temp[i]),
            .out_en(out_temp[i])
        );
    end
    endgenerate

    if (M<N) begin
        assign out_en=1;
        assign res=0;
        assign mod=A;
    end
    else begin
        assign res=next_A_temp[N-1][M-1:0];
        assign mod=next_A_temp[M-1][M+N-1:M];
        assign out_en=out_temp[N-1];
    end
endmodule
```

Verilog

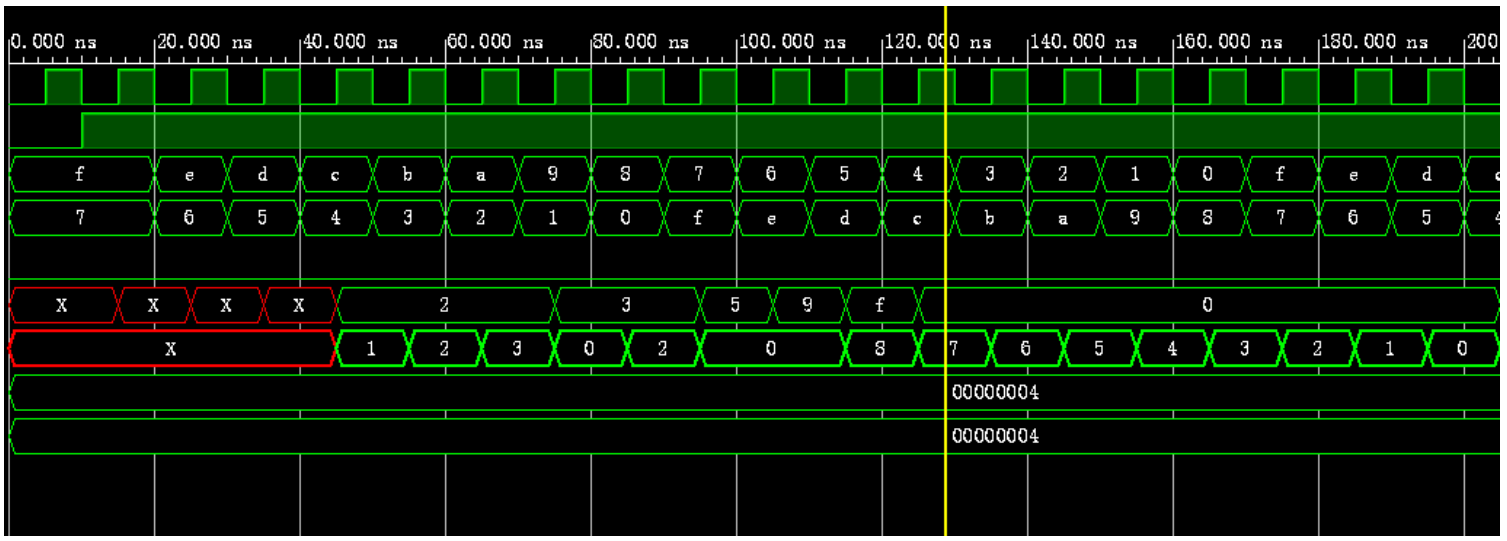
- 测试代码

```
`timescale 1ns / 1ps


module divider_pipe_test();

    // Parameters
```

```
    localparam  M = 4;
    localparam  N = 4;

    //Ports
    reg  clk;
    reg  rst;
    reg [M-1:0] A;
    reg [N-1:0] B;
    wire  out_en;
    wire [M-1:0] res;
    wire [N-1:0] mod;

    initial begin
        clk=0;
        rst=0;
        A=15;
        B=7;
        #10;
        rst=1;
        forever begin
            #10;
            A=A-1;
            B=B-1;
        end
    end

    dividerpipe_top # (
        .M(M),
        .N(N)
    )
    dividerpipe_top_inst (
        .clk(clk),
        .rst(rst),
```

```
        .A(A),
        .B(B),
        .out_en(out_en),
        .res(res),
        .mod(mod)
    );

    always #5  clk = ! clk ;

endmodule
```

Verilog



- 有符号的流水线除法器

思想同有符号的无流水除法器

改动如下：

流水段：

```
`timescale 1ns / 1ps

module dividerpipe_design #(parameter  M=4,N=4)(
    input clk,
    input rst,
```

```
    input [M+N-1:0]A,
    input [M+N-1:0]B,
    input i,
    input A_sign,
    input B_sign,

    output reg [M+N-1:0] A_next,
    output reg [M+N-1:0] B_next,
    output reg out_en,
    output reg A_next_sign,
    output reg B_next_sign
);
    wire next_i=i+1;

    always @(posedge clk or negedge rst) begin
        if (~rst) begin
            out_en=0;
        end
        else begin
            A_next_sign=A_sign;
            B_next_sign=B_sign;
            B_next=B;
            A_next=A<<1;
            if (A_next[M+N-1:M]>=B[M+N-1:M]) begin
                A_next=A_next-B_next+1;
            end
            if (next_i==N) begin
                out_en=1;
            end
        end
    end
endmodule
```

Verilog

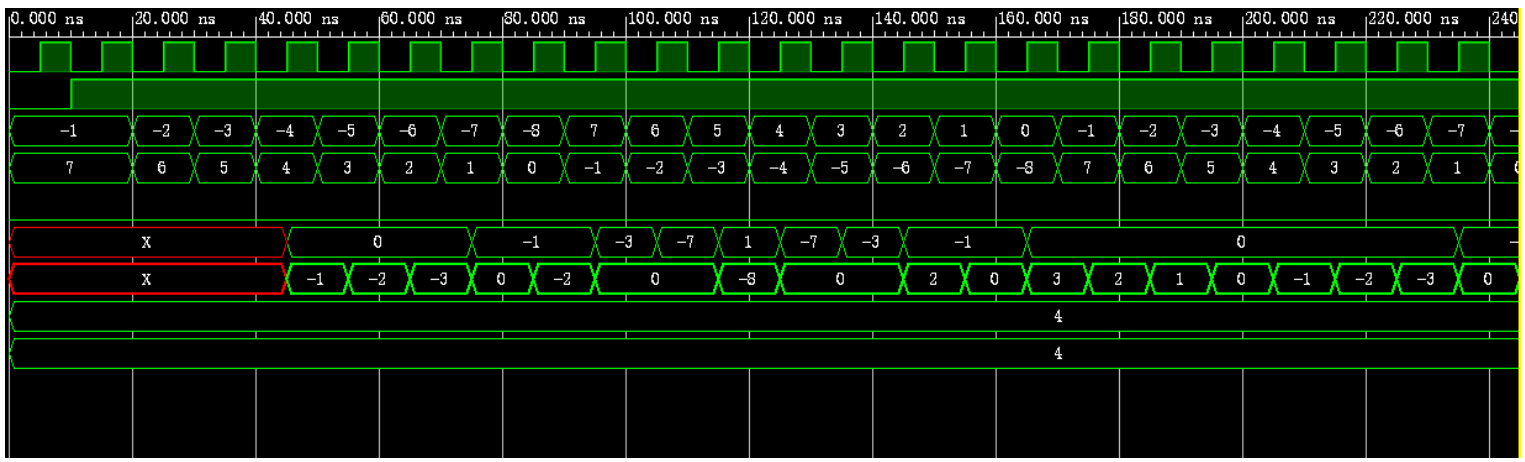
顶层：

```
`timescale 1ns / 1ps


module dividerpipe_top#(parameter M=4,N=4)(
    input clk,
    input rst,

    input [M-1:0]A,
    input [N-1:0]B,
    output  out_en,
    output [M-1:0]res,
    output [N-1:0]mod
  );
    wire [M-1:0]B_=(B[N-1]==0)?B:(~B+1);
    wire [M-1:0]A_=(A[M-1]==0)?A:(~A+1);
    wire [M+N-1:0]B_temp={B_,{M{1'b0}}};
    wire [M+N-1:0]A_temp=A_;

    wire out_temp[N-1:0];
    wire [M+N-1:0]next_A_temp[N-1:0];
    wire [M+N-1:0]next_B_temp[N-1:0];
    wire A_next_sign[N-1:0];
    wire B_next_sign[N-1:0];

    dividerpipe_design # (
                      .M(M),
                      .N(N)
                    )
                    dividerpipe_design_inst (
```

```
                        .clk(clk),
                        .rst(rst),
                        .A(A_temp),
                        .B(B_temp),
                        .i(0),
                        .A_sign(A[M-1]),
                        .B_sign(B[N-1]),
                        .A_next(next_A_temp[0]),
                        .B_next(next_B_temp[0]),
                        .out_en(out_temp[0]),
                        .A_next_sign(A_next_sign[0]),
                        .B_next_sign(B_next_sign[0])
                    );
    genvar i;
    generate for ( i= 1; i<N; i=i+1) begin
        dividerpipe_design # (
            .M(M),
            .N(N)
        )
        dividerpipe_design_pipe (
            .clk(clk),
            .rst(rst),
            .A(next_A_temp[i-1]),
            .B(next_B_temp[i-1]),
            .i(i),
            .A_sign(A_next_sign[i-1]),
            .B_sign(B_next_sign[i-1]),
            .A_next(next_A_temp[i]),
            .B_next(next_B_temp[i]),
            .out_en(out_temp[i]),
            .A_next_sign(A_next_sign[i]),
            .B_next_sign(B_next_sign[i])
        );
```

```
    end
    endgenerate

    if (M<N) begin
        assign out_en=1;
        assign res=0;
        assign mod=A;
    end
    else begin
        assign res=(A_next_sign[N-1]^B_next_sign[N-1]==0)?next_A_temp[N-1][M-
1:0]:~next_A_temp[N-1][M-1:0]+1;
        assign mod=(A_next_sign[M-1]==next_A_temp[N-1][M+N-1])?next_A_temp[N-1][M+N-
1:M]:~next_A_temp[N-1][M+N-1:M]+1;
        assign out_en=out_temp[N-1];
    end
endmodule
```

Verilog



[注释 1] 输入信号应先于 clk 动作到达的时间

[注释 2] clk 动作到达后，输入信号仍需保存的时间

[注释 3] clk 动作到达后，到触发器输出新的状态稳定建立所需要的时间

[注释 4] 不管往里写入什么，总是返回 0

[1] Verilog中的寄存器堆通常是在时钟上升沿进行写入操作。在时钟下降沿写入寄存器并不是常见的做法，因为在时钟下降沿进行数据写入可能会导致稳定性和可靠性问题。

在时钟下降沿写入寄存器可能会导致时序问题，因为寄存器的存储元件需要足够的时间来稳定并接收正确的数据。如果在时钟下降沿写入数据，可能会导致数据在存储元件稳定之前发生变化，这将导致不确定的结果。

因此，通常建议在时钟上升沿写入寄存器，以确保数据稳定地传递到寄存器中。