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;  
 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

#### 原理

|  |  |
| --- | --- |
|  | 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

`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

Verilog

#### 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

`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

Verilog

#### 测试代码

`timescale 1ns / 1ps  
  
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：

module full\_adder1(  
 input Ai,Bi,Ci,  
 output So,Co);  
   
 assign So=Ai^Bi^Ci;  
 assign Co=(Ai&Bi)|(Ci&(Ai|Bi));  
endmodule

Verilog

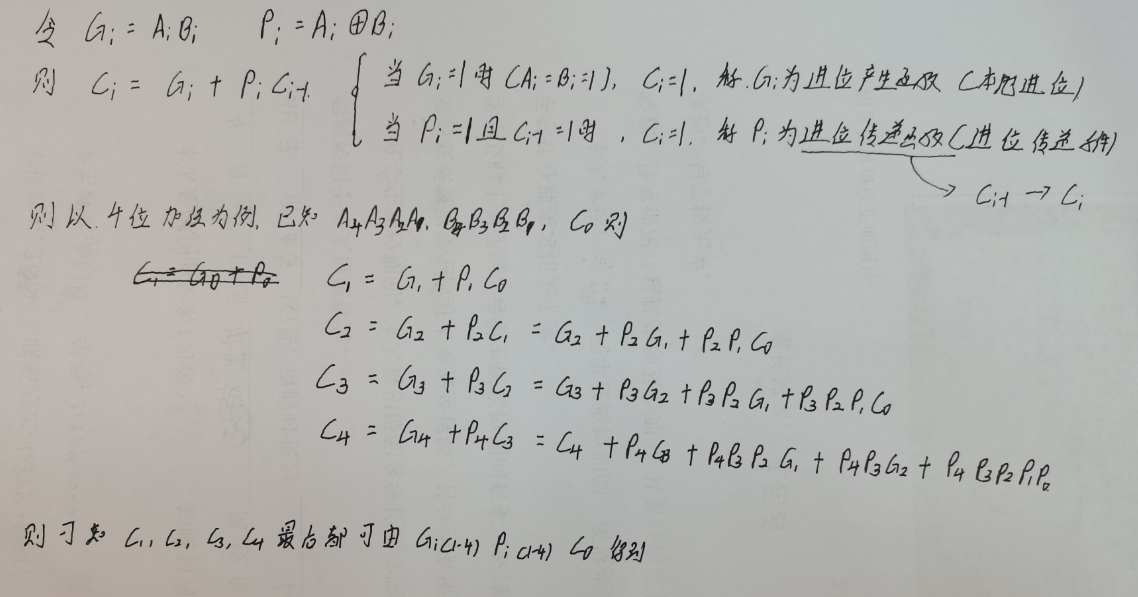
#### 四位全加器——用一位全加器嵌套实现

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](https://www.wolai.com/hxRrm2Kw3MnLxoW1E7Rq7b#cLFui4bLWpkt6Zu96Zjf69)



* 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](https://zhuanlan.zhihu.com/p/129850342) |

* 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

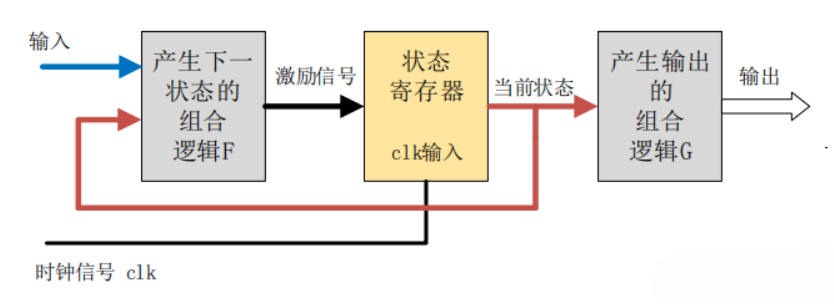
Verilog

### 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

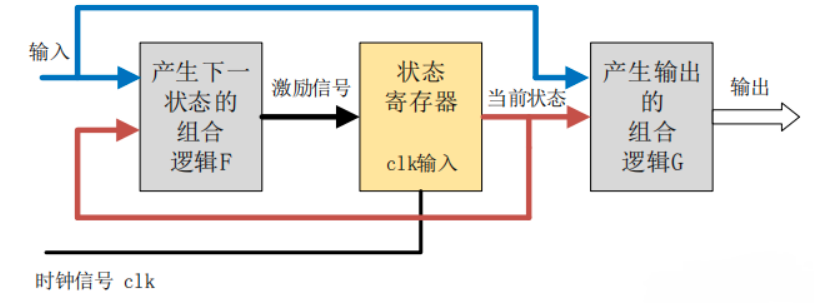


Moore 型状态机的**输出只与当前状态有关，与当前输入无关**

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：**

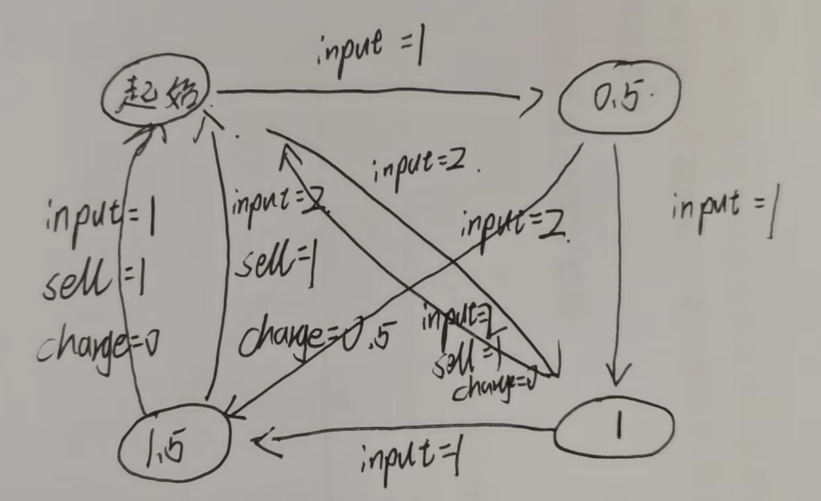
* 首先，根据状态机的个数确定状态机编码。利用编码给状态寄存器赋值，代码可读性更好
* 状态机第一段，**时序逻辑，非阻塞赋值<=，传递寄存器的状态**
* 状态机第二段，**组合逻辑，阻塞赋值=，根据当前状态和当前输入，确定下一个状态机的状态**
* 状态机第三代，**时序逻辑，非阻塞赋值<=，因为是 Mealy 型状态机，根据当前状态和当前输入，确定输出信号**

#### 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

1. 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

1. 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

1. 测试代码

`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
* 非循环左移
* 循环右移
* 非循环右移 shr
* 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，

condir=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，

con\_dir=1，

输出data\_out用条件选择符决定:

`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

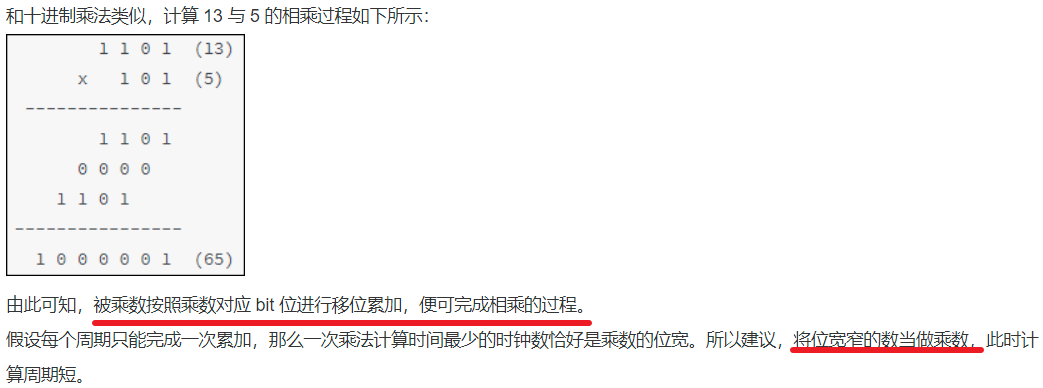
#### 常数的乘法可以用移位相加来实现

移n位是乘以，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



**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

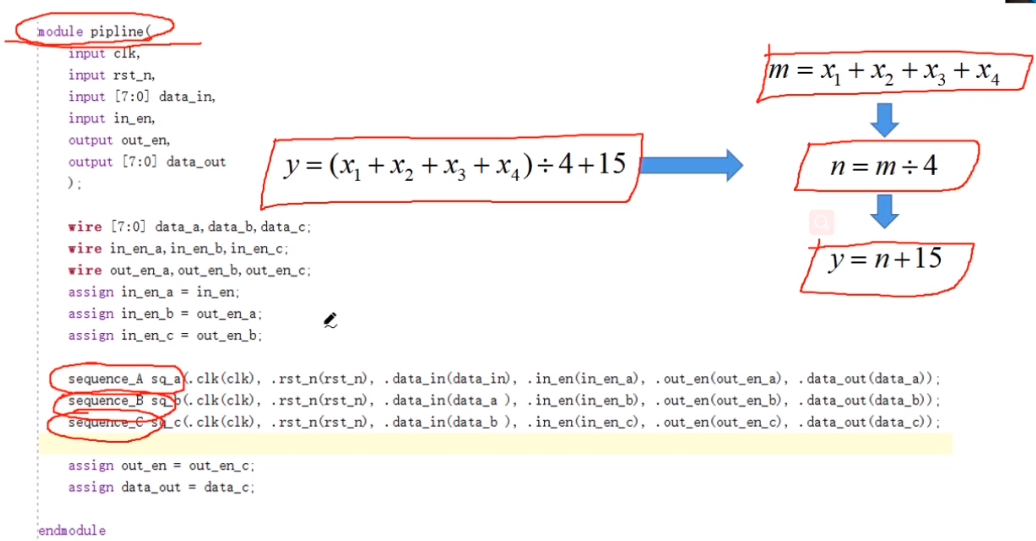
* 流水线设计思想

✍🏽**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



* 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,  
  
 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

Verilog

* + 顶层模块

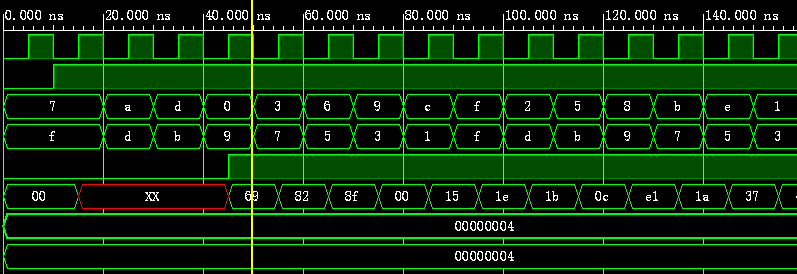
`timescale 1ns / 1ps  
module multiplypipe\_top #(parameter M =4 ,N=4)(  
 input clk,  
 input rst,  
 input [M-1:0]mul\_a,  
 input [N-1:0]mul\_b,  
 output out\_en,  
 output [M+N-1:0]res  
);   
 wire [M+N-1:0] mul\_a\_next[N-1:0];  
 wire [N-1:0] mul\_b\_next[N-1:0];  
 wire [M+N-1:0]res\_next[N-1:0];  
 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

* 1. 按照上述的无符号数操作进行计算
  2. 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

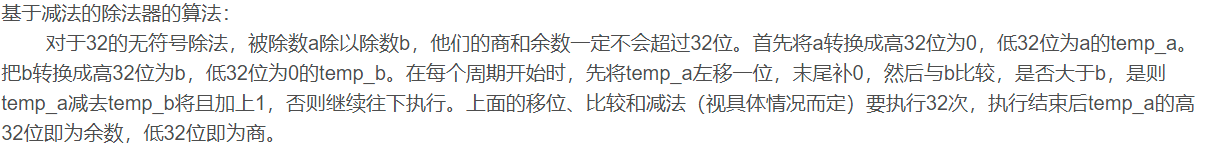
* 1. 顶层代码

`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



#### 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

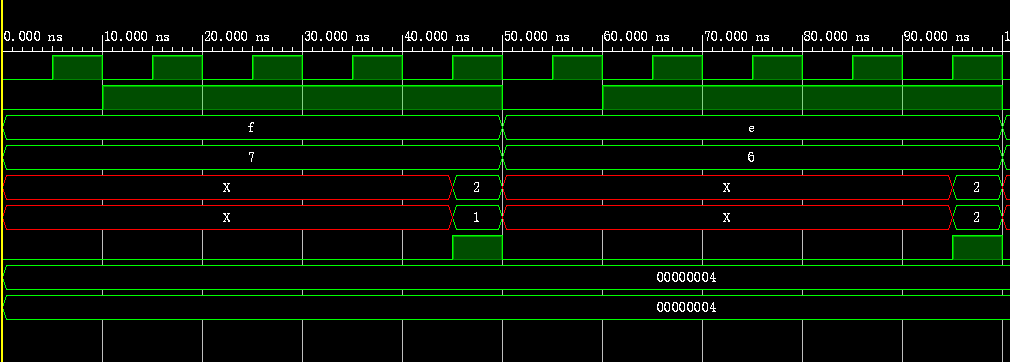
`timescale 1ns / 1ps  
module divider\_design #(parameter M=4,N=4)(  
 input clk,  
 input rst,  
  
 input [M-1:0]A,  
 input [N-1:0]B,  
 output reg out\_en,  
 output reg [M-1:0]res,  
 output reg [N-1:0]mod  
 );  
  
 reg [M+N-1:0]B\_temp;  
 reg [M+N-1:0]A\_temp;  
 integer i;  
 **always @**(posedge 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



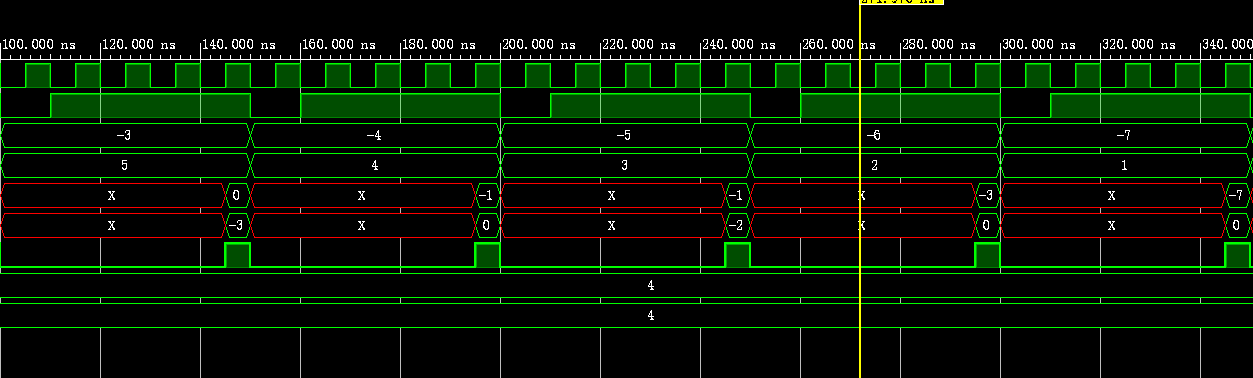
* 有符号数的除法——思想与乘法器处理一致

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：

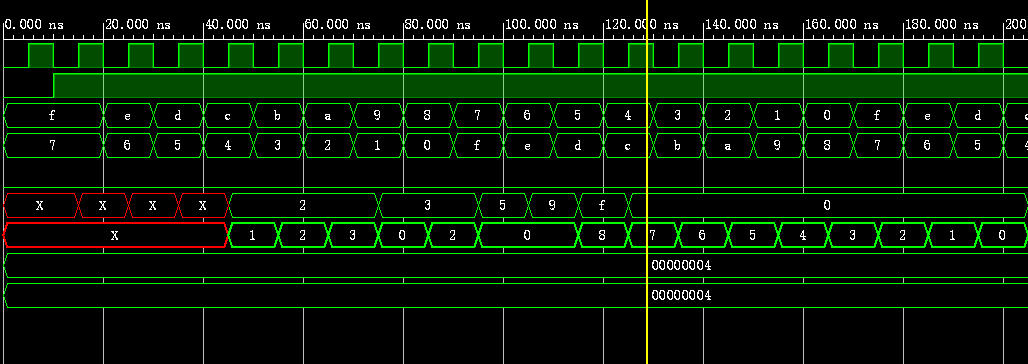
`timescale 1ns / 1ps  
  
  
module dividerpipe\_top#(parameter M=4,N=4)(  
 input clk,  
 input rst,  
  
 input [M-1:0]A,  
 input [N-1:0]B,  
 output out\_en,  
 output [M-1:0]res,  
 output [N-1:0]mod  
 );  
 wire [M+N-1:0]B\_temp={B,{M{1'b0}}};  
 wire [M+N-1:0]A\_temp=A;  
  
 wire out\_temp[N-1:0];  
 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：

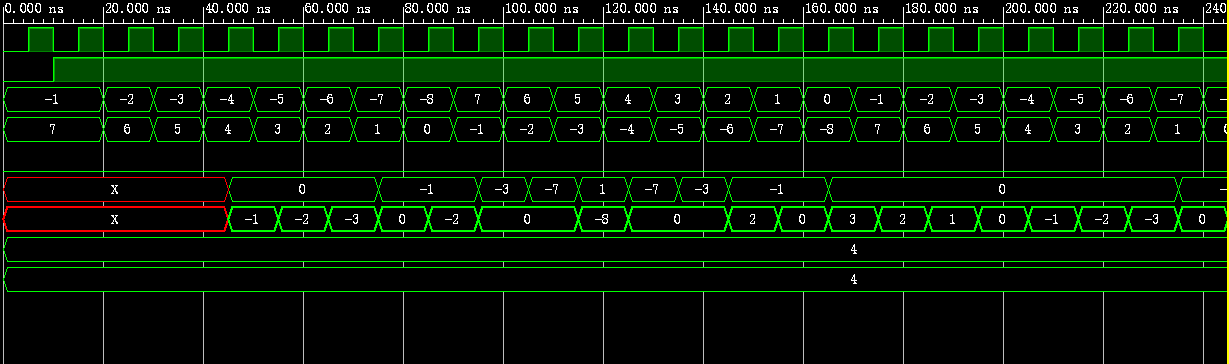
`timescale 1ns / 1ps  
  
module dividerpipe\_design #(parameter M=4,N=4)(  
 input clk,  
 input rst,  
  
 input [M+N-1:0]A,  
 input [M+N-1:0]B,  
 input i,  
 input A\_sign,  
 input B\_sign,  
  
 output reg [M+N-1:0] A\_next,  
 output reg [M+N-1:0] B\_next,  
 output reg out\_en,  
 output reg 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。