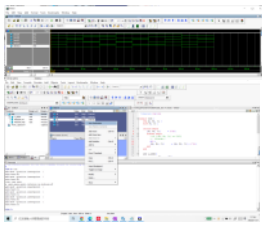
2022年11月3日 17:09

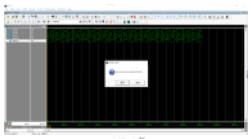
学习verilog
在菜鸟教程看了基本语法
下载了一个全加器的例程尝试quartus和modelsim的联合调试、
add.v文件
代码如下

```
module add(
        input  Ai, Bi, Ci,   //输入
        output  So, Co);   //输出

`ifdef ADDER_DESCRIPTION
        assign {Co, So} = Ai + Bi + Ci ;   //如果定义了ADDER_DESCRIPTION那么用这个语句
`else
        assign So = Ai ^ Bi ^ Ci ;            //如果没有定义用这个原理性语句
        assign Co = (Ai & Bi) | (Ci & (Ai | Bi));
`endif
```

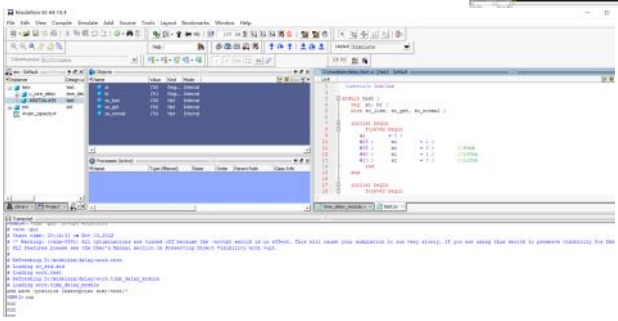


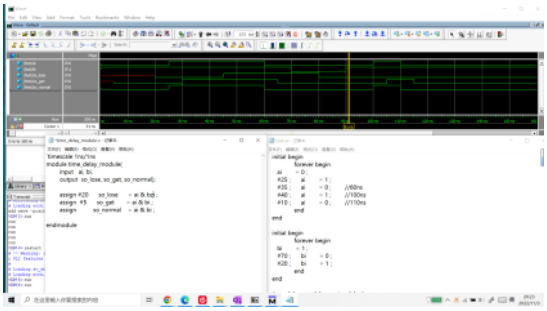
add_test.vt文件
代码如下

```
`timescale 1ns/1ns   //时间单位ns,精度ns

module test ;
  reg Ai, Bi, Ci ;        //寄存器型
  wire So, Co ;           //线与型

  initial begin          //初始化定义 Ai=0，Bi=0，Ci=0
    {Ai, Bi, Ci}     = 3'b0;
    forever begin       //一直做下面的语句
      //if ({Ai, Bi, Ci} == 3'h7)
      //  $finish ;
      #10 ;             //等待10ns
      {Ai, Bi, Ci}     = {Ai, Bi, Ci} + 1'b1;  //AiBiCi加一
    end
  end

  add u_adder(
            .Ai     (Ai),
            .Bi     (Bi),
            .Ci     (Ci),

            .So     (So),
            .Co     (Co));


  initial begin
    forever begin       //等待100ns
      #100;
      //$display("---gyc---%d", $time);
      if ($time >= 1000) begin   //如果时间超过1000ns停止程序
        $finish ;
      end
    end
  end
endmodule // test
```

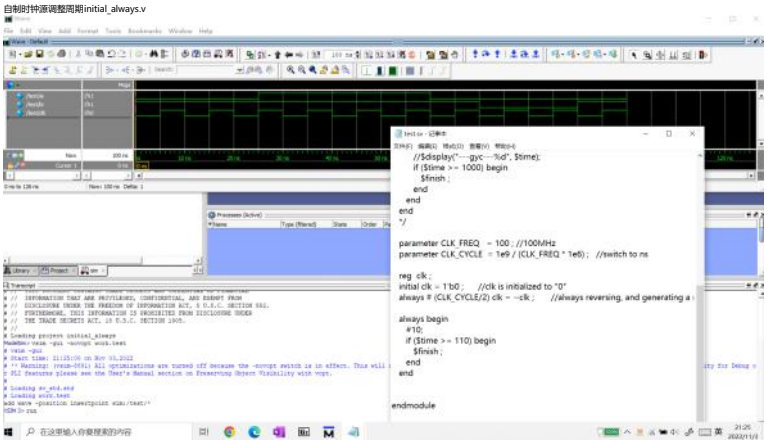


菜鸟教程给的例程test.sv里没有forever begin，导致simulation出来的objects没有值
Time_delay_module.v里忘写`timescale 1ns/ns了导致报错





自制时钟源调整周期initial_always.v



```
    //$display("---gyc---%d", $time);
    if ($time >= 1000) begin
      $finish ;
    end
  end
end
*/

parameter CLK_FREQ   = 100 ; //100MHz
parameter CLK_CYCLE  = 1e9 / (CLK_FREQ * 1e6) ;  //switch to ns

reg  clk ;
initial clk = 1'b0 ;    //clk is initialized to "0"
always # (CLK_CYCLE/2) clk = ~clk ;    //always reversing, and generating a

always begin
  #10;
  if ($time >= 110) begin
    $finish ;
  end
end


endmodule
```

阻塞赋值，顺序进行
非阻塞赋值，并行进行，使用的是并行进行前的旧值
一种奇妙的交换ab两值的方法

```
always @(posedge clk) begin
    a <= b ;
end

always @(posedge clk) begin
    b <= a;
end
```

```
`timescale 1ns/1ns

module test ;
  reg value_test ;
  reg value_general, value_embed, value_single ;

  //signal source
  initial begin
    value_test    = 0 ;
    #25 ;    value_test    = 1 ;
    #35 ;    value_test    = 0 ;    //absolute 60ns
    #40 ;    value_test    = 1 ;    //absolute 100ns
    #10 ;    value_test    = 0 ;    //absolute 110ns
  end
```

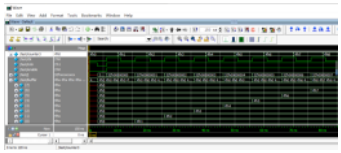
关键字 posedge 指信号发生边沿正向跳变，negedge 指信号发生负向边沿跳变，未指明跳变方向时，则 2 种情况的边沿变化都会触发相关事件

```
//信号clk只要发生变化，就执行q<=d，双边沿D触发器模型
always @(clk) q <= d ;
//在信号clk上升沿时刻，执行q<=d，正边沿D触发器模型
always @(posedge clk) q <= d ;
//在信号clk下降沿时刻，执行q<=d，负边沿D触发器模型
always @(negedge clk) q <= d ;
//立刻计算d的值，并在clk上升沿时刻赋值给q，不推荐这种写法
q = @(posedge clk) d ;
```

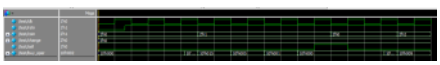
```
//signal source
initial begin
  value_test     = 0 ;
  #25;   value_test     = 1 ;
  #35;   value_test     = 0 ;    //absolute 60ns
  #40;   value_test     = 1 ;    //absolute 100ns
  #10;   value_test     = 0 ;    //absolute 110ns
end


//(1)general delay control
initial begin
  value_general         = 1;
  #10 value_general  = value_test ; //10ns, value_test=0,先等10ns，看看，再赋值
  #45 value_general  = value_test ; //55ns, value_test=1
  #30 value_general  = value_test ; //85ns, value_test=0
  #20 value_general  = value_test ; //105ns, value_test=1
end

//(2)embedded delay control
initial begin
  value_embed           = 1;
  value_embed  = #10 value_test ; //10ns, value_test=0，先看看，等10ns，再赋值
  value_embed  = #45 value_test ; //55ns, value_test=1
  value_embed  = #30 value_test ; //85ns, value_test=0
  value_embed  = #20 value_test ; //105ns, value_test=1
end

//(3)single delay control
initial begin
  value_single = 1;
  #10 ;
  value_single = value_test ; //10ns, value_test=0
  #45 ;
  value_single = value_test ; //55ns, value_test=1
  #30 ;
  value_single = value_test ; //85ns, value_test=0
  #20 ;
  value_single = value_test ; //105ns, value_test=1
end

 always begin
   #10;
   if ($time >= 150) begin
     $finish ;
   end
 end

endmodule



// vending-machine
// 2 yuan for a bottle of drink
// only 2 coins supported: 5 jiao and 1 yuan
// finish the function of selling and changing

module  vending_machine_p3  (
    input           clk ,
    input           rstn ,
    input  [1:0]    coin ,      //01 for 0.5 jiao, 10 for 1 yuan

    output [1:0]    change ,
    output          sell    //output the drink
    );

    //machine state decode
    parameter           IDLE    = 3'd0 ;          //所有可能的状态列出来
    parameter           GET05   = 3'd1 ;
    parameter           GET10   = 3'd2 ;
    parameter           GET15   = 3'd3 ;

    //machine variable
    reg [2:0]           st_next ;               //状态寄存器
    reg [2:0]           st_cur ;

    //(1) state transfer
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            st_cur      <= 'b0 ;                //复位，状态0
        end
        else begin
            st_cur      <= st_next ;            //复位完成，在时钟的上升沿，状态转变为下一状态
        end
    end

    //(2) state switch, using block assignment for combination-logic
    //all case items need to be displayed completely
    always @(*) begin
        //st_next = st_cur ;//如果条件选项考虑不全，可以赋初值消除latch
        case(st_cur)                    //条件选择，按照状态转移图画
            IDLE:
                case (coin)
                    2'b01:      st_next = GET05 ;
                    2'b10:      st_next = GET10 ;
                    default:    st_next = IDLE ;
                endcase
            GET05:
                case (coin)
                    2'b01:      st_next = GET10 ;
                    2'b10:      st_next = GET15 ;
                    default:    st_next = GET05 ;
                endcase

            GET10:
                case (coin)
                    2'b01:      st_next = GET15 ;
                    2'b10:      st_next = IDLE ;
                    default:    st_next = GET10 ;
                endcase
            GET15:
                case (coin)
                    2'b01,2'b10:
                                st_next = IDLE ;
                    default:    st_next = GET15 ;
                endcase
            default:    st_next = IDLE ;
        endcase
    end

    //(3) output logic, using non-block assignment
    reg  [1:0]   change_r ;
    reg          sell_r ;
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            change_r        <= 2'b0 ;               //复位
            sell_r          <= 1'b0 ;
        end
        else if ((st_cur == GET15 && coin ==2'b1)
                || (st_cur == GET10 && coin ==2'd2)) begin
            change_r        <= 2'b0 ;
            sell_r          <= 1'b1 ;
        end
        else if (st_cur == GET15 && coin == 2'b2) begin
            change_r        <= 2'b1 ;
            sell_r          <= 1'b1 ;
        end
        else begin
            change_r        <= 2'b0 ;
            sell_r          <= 1'b0 ;
        end
    end
    assign      sell    = sell_r ;
    assign      change  = change_r ;
`timescale 1ns/1ps

endmodule
module test ;
来自 <https://www.runoob.com/w3cnote/verilog-fsm.html>
    reg             clk ;
    reg             rstn ;
    reg [1:0]       coin ;
    wire [1:0]      change ;
    wire            sell ;

    //clock generating
    parameter   CYCLE_200MHz = 10 ;    //搞个时钟
    always begin
        clk = 0 ; #(CYCLE_200MHz/2) ;
        clk = 1 ; #(CYCLE_200MHz/2) ;
    end

    //motivation generating
    reg [9:0]     buy_oper ; //store state of the buy operation
    initial begin
        buy_oper    = 'b0 ;
        coin        = 2'b0 ;
        rstn        = 1'b0 ;
        #8 rstn     = 1'b1 ;
        @(negedge clk) ;

        //case(1) 0.5 -> 0.5 -> 0.5 -> 0.5
        #16 ;
        buy_oper    = 10'b00_0101_0101 ;
        repeat(5) begin
            @(negedge clk) ;
            coin        = buy_oper[1:0] ;
            buy_oper    = buy_oper >> 2 ;
```

```
always @(clk) q <= d ;
//在信号clk上升沿时刻，执行q<=d，正边沿D触发器模型
always @(posedge clk) q <= d ;
//在信号clk下降沿时刻，执行q<=d，负边沿D触发器模型
always @(negedge clk) q <= d ;
//立刻计算d的值，并在clk上升沿时刻赋值给q，不推荐这种写法
q = @(posedge clk) d ;
```

来自 <https://www.runoob.com/w3cnote/verilog-timing-control.html>



```
`timescale 1ns/1ns

module test ;

 //(1) while-loop sentense
 reg [3:0]   counter ;
 initial begin
   counter = 'b0 ;
   while (counter<=10) begin
     #10 ;
     counter = counter + 1'b1 ;
   end

 end

 //(2) for-loop sentense
 integer   i;
 reg [3:0]   counter2 ;
 initial begin
   counter2 = 'b0 ;
   for (i=0; i<=10; i=i+1) begin
     #10 ;
     counter2 = counter2 + 1'b1 ;
   end
 end

 //(3) repeat-loop sentense
 reg [3:0]   counter3 ;
 initial begin
   counter3 = 'b0 ;
   repeat (11) begin
     #10 ;
     counter3 = counter3 + 1'b1 ;
   end
 end

 //(3)
 reg       clk ;
 reg       rstn ;
 reg       enable ;
 reg [3:0]   buffer [7:0] ;
 integer     j ;

 initial begin
   clk    = 0 ;
   rstn   = 1 ;
   enable  = 0 ;
   #3;
   rstn   = 0 ;
   #3;
   rstn   = 1 ;
   enable  = 1 ;
   forever begin
     clk = ~clk ;
     #5 ;
   end
 end

 always @(posedge clk or negedge rstn) begin
   j = 0 ;
   if (!rstn) begin
     repeat (8) begin
       buffer[j] <= 'b0 ;
       j = j + 1 ;
     end
   end
   else if (enable) begin
     repeat (8) begin
       @(posedge clk) buffer[j]   <= counter3 ;
       j = j + 1 ;
     end
   end
 end



 //stop the simulation
 always begin
   #10 ;
   if ($time >= 1000) $finish ;
 end

endmodule
```



感觉学到这个售货机这里，已经有流水线工作的雏形了

工作顺序：读取硬币-记录状态-读取硬币-记录状态-读取硬币-记录状态-读取硬币-记录状态-确定sell-确定change

需要1：状态转移图

2：翻译为代码，使用三段式，1）确定需要记录的寄存器值

2）状态转移case

3）每个状态+输入=输出

```verilog
        end

        //case(2) 1 -> 0.5 -> 1, taking change
        #16 ;
        buy_oper  = 10'b00_0010_0110 ;
        repeat(5) begin
            @(negedge clk) ;
            coin      = buy_oper[1:0] ;
            buy_oper  = buy_oper >> 2 ;
        end

        //case(3) 0.5 -> 1 -> 0.5
        #16 ;
        buy_oper  = 10'b00_0001_1001 ;
        repeat(5) begin
            @(negedge clk) ;
            coin      = buy_oper[1:0] ;
            buy_oper  = buy_oper >> 2 ;
        end

        //case(4) 0.5 -> 0.5 -> 0.5 -> 1, taking change
        #16 ;
        buy_oper  = 10'b00_1001_0101 ;
        repeat(5) begin
            @(negedge clk) ;
            coin      = buy_oper[1:0] ;
            buy_oper  = buy_oper >> 2 ;
        end
    end

   //(1) mealy state with 3-stage
    vending_machine_p3    u_mealy_p3     (
        .clk              (clk),
        .rstn             (rstn),
        .coin             (coin),
        .change           (change),
        .sell             (sell)
        );

   //simulation finish
   always begin
      #100;
      if ($time >= 10000)  $finish ;
   end

endmodule // test
```

下面通过学习一个乘法器流水线例程理解一下流水线的工作方式

来自 <https://www.runoob.com/w3cnote/verilog-fsm.html>

首先对乘法器进行一个非流水线设计

```verilog
module    mult_low
    #(parameter N=4,              //设置参数：乘数与被乘数的位数
      parameter M=4)
     (
      input                     clk,          //时钟
      input                     rstn,         //复位
      input                     data_rdy ,  //数据输入使能
      input [N-1:0]             mult1,      //被乘数
      input [M-1:0]             mult2,      //乘数

      output                    res_rdy ,   //数据输出使能
      output [N+M-1:0]          res         //乘法结果
      );

    //calculate counter
    reg [31:0]           cnt ;
    //乘法周期计数器
    wire [31:0]          cnt_temp = (cnt == M)? 'b0 : cnt + 1'b1 ;//cnt_temp=只要没计到就加一
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            cnt    <= 'b0 ;
        end
        else if (data_rdy) begin    //数据使能时开始计数
            cnt    <= cnt_temp ;
        end
        else if (cnt != 0 ) begin  //防止输入使能端持续时间过短
            cnt    <= cnt_temp ;
        end
        else begin
            cnt    <= 'b0 ;
        end
    end

    //multiply
    reg [M-1:0]          mult2_shift ;
    reg [M+N-1:0]        mult1_shift ;
    reg [M+N-1:0]        mult1_acc ;
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            mult2_shift    <= 'b0 ;
            mult1_shift    <= 'b0 ;
            mult1_acc      <= 'b0 ;
        end
        else if (data_rdy && cnt=='b0) begin  //初始化，如果有数据输入且乘法计数器=0，意味着要开始乘了
            mult1_shift    <= {{(N){1'b0}}, mult1} << 1 ;  //被乘数左移一位。0000 1101 变成0001 1010
            mult2_shift    <= mult2 >> 1 ;  //乘数右移一位0101变成0010
            mult1_acc      <= mult2[0] ? {{(N){1'b0}}, mult1} : 'b0 ;//0000 1101，注意，这是并行执行，这三行用的都是
最初的原始数据
        end
        else if (cnt != M) begin
            mult1_shift    <= mult1_shift << 1 ;  //被乘数乘2
            mult2_shift    <= mult2_shift >> 1 ;  //乘数右移，方便判断
            //判断乘数对应为是否为1，为1则累加
            mult1_acc      <= mult2_shift[0] ? mult1_acc + mult1_shift : mult1_acc ;
        end
        else begin                       //四位都搞完了
            mult2_shift    <= 'b0 ;
            mult1_shift    <= 'b0 ;
            mult1_acc      <= 'b0 ;
        end
    end

    //results
    reg [M+N-1:0]        res_r ;
    reg                  res_rdy_r ;
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            res_r          <= 'b0 ;
            res_rdy_r      <= 'b0 ;
        end
        else if (cnt == M) begin
            res_r          <= mult1_acc ;  //乘法周期结束时输出结果
            res_rdy_r      <= 1'b1 ;
        end
        else begin
            res_r          <= 'b0 ;
            res_rdy_r      <= 'b0 ;
        end
    end

    assign res_rdy       = res_rdy_r;
    assign res           = res_r;

endmodule
```

来自 <https://www.runoob.com/w3cnote/verilog-pipeline-design.html>

单周期的testbench

好的，已经成功看懂，单周期运行与testbench代码了

```verilog
`timescale 1ns/1ns

module test ;
    parameter    N = 8 ;
    parameter    M = 4 ;
    reg          clk, rstn;

   //clock
    always begin
        clk = 0 ; #5 ;
        clk = 1 ; #5 ;
    end

   //reset
    initial begin
        rstn      = 1'b0 ;
        #8 ;      rstn      = 1'b1 ;
    end

    //no pipeline
    reg                  data_rdy_low ;
    reg [N-1:0]          mult1_low ;
    reg [M-1:0]          mult2_low ;
    wire [M+N-1:0]       res_low ;
    wire                 res_rdy_low ;

    //使用任务周期激励
    task mult_data_in ;
        input [M+N-1:0]   mult1_task, mult2_task ;
        begin
            wait(!test.u_mult_low.res_rdy) ;  //not
output state
            @(negedge clk ) ;
```

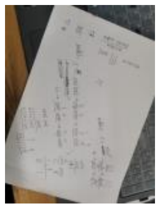
下面对乘法执行过程的中间状态进行保存，以便流水工作

只要使能就往下干，没有计数？？？那怎么停？看懂了但不理解，再看看下一个层次咋写，毕竟这是个cell

```verilog
module    mult_cell
    #(parameter N=4,
      parameter M=4)
     (
      input                     clk,
      input                     rstn,
      input                     en,
      input [M+N-1:0]           mult1,      //被乘数
      input [M-1:0]             mult2,      //乘数
      input [M+N-1:0]           mult1_acci, //上次累加结果

      output reg [M+N-1:0]      mult1_o,     //被乘数移位后保存
值
      output reg [M-1:0]        mult2_shift, //乘数移位后保存值
      output reg [N+M-1:0]      mult1_acco,  //当前累加结果
      output reg                rdy );

    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            rdy            <= 'b0 ;
            mult1_o        <= 'b0 ;
            mult1_acco     <= 'b0 ;
            mult2_shift    <= 'b0 ;
        end
        else if (en) begin
            rdy            <= 1'b1 ;
            mult2_shift    <= mult2 >> 1 ;
            mult1_o        <= mult1 << 1 ;
            if (mult2[0]) begin
                //乘数对应位为1则累加
                mult1_acco  <= mult1_acci + mult1 ;
            end
            else begin
                mult1_acco  <= mult1_acci ; //乘数对应位为1则保
持
            end
        end
        else begin
            rdy            <= 'b0 ;
            mult1_o        <= 'b0 ;
            mult1_acco     <= 'b0 ;
            mult2_shift    <= 'b0 ;
        end
    end

endmodule
```

这个应该才是和第一个单周期相提并论的模块

```verilog
module    mult_man
    #(parameter N=4,
      parameter M=4)
     (
      input                     clk,
      input                     rstn,
      input                     data_rdy ,
      input [N-1:0]             mult1,
      input [M-1:0]             mult2,

      output                    res_rdy ,
      output [N+M-1:0]          res );

    wire [N+M-1:0]       mult1_t [M-1:0] ;
    wire [M-1:0]         mult2_t [M-1:0] ;
    wire [N+M-1:0]       mult1_acc_t [M-1:0] ;//关键点应该在这里，把cell里的output reg型变成wire
型了
    wire [M-1:0]         rdy_t ;

    //第一次例化相当于初始化，不能用 generate 语句
    mult_cell      #(.N(N), .M(M))
    u_mult_step0
    (
      .clk              (clk),
      .rstn             (rstn),
      .en               (data_rdy),
      .mult1            ({{(M){1'b0}}, mult1}),
      .mult2            (mult2),
      .mult1_acci       ({(N+M){1'b0}}),
      //output
      .mult1_acco       (mult1_acc_t[0]),
      .mult2_shift      (mult2_t[0]),
      .mult1_o          (mult1_t[0]),
      .rdy              (rdy_t[0]) );

    //多次模块例化，用 generate 语句
    genvar               i ;
    generate
        for(i=1; i<=M-1; i=i+1) begin: mult_stepx          //只有cell乘法是停不下来的
            mult_cell      #(.N(N), .M(M))                 //在这里设置了for循环，乘法就能停下
来了，不对，乘法根本不用停下来
            u_mult_step
            (
              .clk              (clk),
              .rstn             (rstn),
              .en               (rdy_t[i-1]),
              .mult1            (mult1_t[i-1]),
              .mult2            (mult2_t[i-1]),
              //上一次累加结果作为下一次累加输入
              .mult1_acci       (mult1_acc_t[i-1]),
              //output
              .mult1_acco       (mult1_acc_t[i]),
              .mult1_o          (mult1_t[i]),   //被乘数移位状态传递
              .mult2_shift      (mult2_t[i]),   //乘数移位状态传递
              .rdy              (rdy_t[i]) );
        end
    endgenerate

    assign res_rdy       = rdy_t[M-1];
    assign res           = mult1_acc_t[M-1];

endmodule
```



看不懂了，对着图像理解一下

来自 <https://www.runoob.com/w3cnote/verilog-pipeline-design.html>

```verilog
    reg                  data_rdy ;
    reg [N-1:0]          mult1 ;
    reg [M-1:0]          mult2 ;
    wire                 res_rdy ;
    wire [N+M-1:0]       res ;

    //driver
    initial begin
        #55 ;
        @(negedge clk ) ;
        data_rdy  = 1'b1 ;
        mult1  = 25;      mult2      = 5;
        #10 ;      mult1  = 16;      mult2      = 10;
        #10 ;      mult1  = 10;      mult2      = 4;
        #10 ;      mult1  = 15;      mult2      = 7;
        mult2      = 7;   repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 1;   repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 15;  repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 3;   repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 11;  repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 4;   repeat(32)    #10   mult1    = mult1 + 1 ;
        mult2      = 9;   repeat(32)    #10   mult1    = mult1 + 1 ;
    end

    //对输入数据进行移位，方便后续校验
    reg  [N-1:0]   mult1_ref [M-1:0];     //把最近的M个乘数与被乘数的值记录下来了，
这有啥用
    reg  [M-1:0]   mult2_ref [M-1:0];
    always @(posedge clk) begin
        mult1_ref[0] <= mult1 ;
        mult2_ref[0] <= mult2 ;
    end

    genvar         i ;
    generate
        for(i=1; i<=M-1; i=i+1) begin
            always @(posedge clk) begin
            mult1_ref[i] <= mult1_ref[i-1];
            mult2_ref[i] <= mult2_ref[i-1];
            end
        end
    endgenerate

    //自校验
    reg  error_flag ;                       //啊把数记下来是为了自校验，那没事了
    always @(posedge clk) begin
        # 1 ;
        if (mult1_ref[M-1] * mult2_ref[M-1] != res && res_rdy) begin
            error_flag <= 1'b1 ;
        end
        else begin
            error_flag <= 1'b0 ;
        end
    end

    //module instantiation
    mult_man  #(.N(N), .M(M))
     u_mult
     (
      .clk              (clk),
      .rstn             (rstn),
      .data_rdy         (data_rdy),
      .mult1            (mult1),
```

明白了，主要是寄存器，把每一步的结果都进行寄存，然后按顺序进行输出

```
    wire                res_rdy_low ;

    //使用任务周期激励
    task mult_data_in ;
        input [M+N-1:0]    mult1_task, mult2_task ;
        begin
            wait(!test.u_mult_low.res_rdy) ;  //not
output state
            @(negedge clk ) ;
            data_rdy_low = 1'b1 ;
            mult1_low = mult1_task ;
            mult2_low = mult2_task ;
            @(negedge clk ) ;
            data_rdy_low = 1'b0 ;
            wait(test.u_mult_low.res_rdy) ; //test the
output state
        end
    endtask

    //driver
    initial begin
        #55 ;
        mult_data_in(25, 5 ) ;
        mult_data_in(16, 10 ) ;
        mult_data_in(10, 4 ) ;
        mult_data_in(15, 7) ;
        mult_data_in(215, 9) ;
    end

    mult_low  #(.N(N), .M(M))
    u_mult_low
    (
      .clk              (clk),
      .rstn             (rstn),
      .data_rdy         (data_rdy_low),
      .mult1            (mult1_low),
      .mult2            (mult2_low),
      .res_rdy          (res_rdy_low),
      .res              (res_low));

    //simulation finish
    initial begin
       forever begin
          #100;
          if ($time >= 10000)  $finish ;
       end
    end

endmodule // test
```

```
//module instantiation
mult_man  #(.N(N), .M(M))
 u_mult
 (
  .clk              (clk),
  .rstn             (rstn),
  .data_rdy         (data_rdy),
  .mult1            (mult1),
  .mult2            (mult2),
  .res_rdy          (res_rdy),
  .res              (res));
```

来自<https://www.runoob.com/w3cnote/verilog-pipeline-design.html>

ok，到此为止我认为对了verilog语言以及modelism模拟器有了初步的了解

下面正式开始作业