Verilog学习补充

字符串

一个字符占八位（1Byte）,不同的定义方式有不同的填充顺序

```
reg [0:1023] data;
data="ab"
```

0	...	...	...	1008-1015	1016-1023
0	0	0	0	"a"	"b"

```
reg [1023:0] data;
data="ab"
```

1023	...	...	...	15-8	7-0
0	0	0	0	"a"	"b"

字符串类的tb新写法

以下内容皆为搬运

每次写字符串自动机的testbench都得写成这样

```
#2; reset = 0; in = "B";
#2; in = "e";
#2; in = "g";
#2; in = "I";
#2; in = "n";
#2; in = " ";
#2; in = " ";
#2; in = "e";
//...此处省略40+行
```

非常的讨厌，所以今天get到了简单用循环移位去做的简单写法（以BlockerChecker的Testbench为例）

```
`timescale 1ns / 1ps
module BlockChecker_tb;

    // Inputs
    reg clk;
    reg reset;
    reg [7:0] in;

    // Outputs
    wire result;
```

```
    reg [0:1023] data;
    integer i = 0;

    // Instantiate the Unit Under Test (UUT)
    BlockChecker uut (
        .clk(clk),
        .reset(reset),
        .in(in),
        .result(result)
    );

    always #1 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 1;
        in = 0;
        data = "begin enDbegin xyzz eNd BeGin begin end endbegin end ";
        while(!data[0:7]) data = data << 8;
        #2;
        reset = 0;
        for(i = 0; i < 53; i=i+1) begin
            in[7:0] = data[0:7];
            data = data << 8;
            #2;
        end
        $finish;
    end
endmodule
```

可以看到，定义了一个`reg [0:1023] data`（**注意，这里一定是从小端开始，否则你读到`in`端口里面的数据就是反过来的ASCII码就不对了**），然后给`data`用一串优雅的字符串直接赋上初值，**由于我们赋值默认被挤到了最右端，所以用`while(!data[0:7]) data = data << 8;`一直移位，直到找到我们想要的数据为止**，然后用一个for循环，直接开始把值打到`in`端口里面去，非常的方便。

END_OF_COPY

数字tb写法

```
reg data[0:1023]; //定义一个存数据的寄存器
initial begin
        // Initialize Inputs
        clk = 1;
        reset = 1;
        coin = 0;
        data=1024'b0101010101010010011001011010101001010111100; //有n组数据，这里是
21组，每组占两位
        #5;
    for(i=0;i<512-21;i=i+1)begin  // i<512-n 和上面一样也要把第一个有效数据移到最左边
        data=data<<2;             //这个移动的位数，根据每组数据的宽度决定
    end
```

```
    reset=0;

    for(i=0;i<21;i=i+1)begin // i<n
        coin=data[0:1];   //每次取开头两位
        data=data<<2;
        #10;
    end
```

这样每次写tb的时候只需按序在data中写即可。 00 01 10 11 就是稍微有点费眼睛（逃

连续输入判断定长字符串

```
reg [23:0] string;
initial begin
        // Initialize Inputs
        clk = 0;
        reset = 0;
        in = 0;
        data=data<<8;
        data[7:0]="i";
        data=data<<8;
        data[7:0]="n";
        data=data<<8;         //要先移位再添加字符
        data[7:0]="t";
        // Wait 100 ns for global reset to finish
    clk=(data=="int");

        // 此时的结果是clk=1

end
```

对于字符串的比较可以直接用==

循环更新字符串

```
reg [23:0] string;

string<=string<<8;  //先移动，再添字符
string[7:0]<=in;
//in分别输入 “i” “n” “t”
string的内容为分别为
_ _ i
_ i n
i n t
//如果还有输入就继续循环覆盖
```

定义string一定要大端开始[23:0] ,要先移位再给string[7:0]赋值。如果是先赋值再移位会导致输入三个字符时，第一个输入的字符被吞掉。即最后是n t _ 。

for循环

for循环的写法

```
integer i;
for (i = 0; i < MAX; i = i + 1) begin
    // do what u want
end
```

这里，我并不是要强调for循环的语法规则，
而是想说两个重点：

1. for循环一般在组合逻辑中作为一种函数出现，
   和**always(*)相伴**，如果不放在always块中，而是直接出现，
   会报错。
2. for循环一般用于计数，而计数时，会有中间变量，
   那么每次启动for循环用于计数时，
   要保证中间变量被清零了，即每次求和所需的integer或者reg。

当然，可以用函数来代替for循环，
不过，函数的for循环，
中间变量在loop中，也得清零，否则会累加。

向量赋值

Verillog 还支持指定 bit 位后固定位宽的向量域选择访问。

- **[bit+: width]** : 从起始 bit 位开始递增，位宽为 width
- **[bit-: width]** : 从起始 bit 位开始递减，位宽为 width

```
//下面 2 种赋值是等效的
A = data1[31-: 8];
A = data1[31:24];
//下面 2 种赋值是等效的
B = data1[0+: 8];
B = data1[0:7];
```

函数用法

```
function <返回值的类型或范围>(函数名);
    <端口说明语句>            // input XXX
    <变量类型说明语句>        // reg YYY
    ......
    begin
        <语句>
        ......
        函数名 = ZZZ;         // 函数名就相当于输出变量;
    end
endfunction
```

```
//大写变小写的函数做例子
function [7:0] lower;
    input [7:0] char;
    reg [7:0] out;
    begin
        if(char>="A"&&char<="Z")begin
            lower=char-8'd32; //函数默认声明了名字与函数名一致，位宽与函数声明定义位宽一直的寄存器作为返回值
        end
    end
endfunction
//函数调用
lower(8'd97);
```

① <返回值的类型或范围>这一项为**可选项**，如果缺失，则返回值为一位寄存器类型数据。

② 从函数的返回值：函数的定义蕴含声明了与函数同名、位宽一致的内部寄存器。例子中，lower被赋予的值就是调用函数的返回值。

③ 函数的调用：函数的调用是通过将函数作为表达式中的操作数来实现的。其调用格式：

<函数名> （<表达式> ,..., <表达式>）；

④ 函数使用的规则

1.函数定义不能包含有任何的时间控制语句，即任何用#、@、wait来标识的语句。

2.函数不能调用“task”。

3.定义函数时至少要有一个输入参数。

4.在函数的定义中**必须有一条赋值语句给函数中与函数名同名、位宽相同的内部寄存器赋值。**

5.verilog中的function只能用于组合逻辑;

多个模块利用for循环例化

不能直接在for循环里面连接模块，用下面的方法（`generate...end`）

```
full_add uut0(.A(a[0]),
              .B(b[0]),
              .C(c[0]),
              .Cin(cin==1'b1 ? 1'b1 : 1'b0),
              .Cout(temp[0])
             );
genvar i;                                          //要定义genvar，不能定义integer
generate
    for(i=1; i<8; i=i+1) begin: generate_adder      //for循环没有i++了，begin后面一定要加一个label,定义一个名字
        full_add uut1(.A(a[i]),                    //要有冒号
                      .B(b[i]),
                      .C(c[i]),
                      .Cin(temp[i-1]),
                      .Cout(temp[i])
                     );
    end
endgenerate
```

```
assign cout = temp[7];
```

一些条件判断

我一般习惯于写二段式的状态机，对于有些复杂的状态题，需要一些中间变量去作为判断的依据，如flag或者i的类型，对于flag或者i的改变要放在时钟信号来的时候，如果信号改变放在组合逻辑中即(always(*))块中，容易出现时序混乱的结果，和预期不符。

每次对于中间变量的改变还是放在时序电路逻辑中比较靠谱。

```
//integer i;
//reg flag; //可能会有中间变量作为标志辅助判断
reg [x:y]state,next_state;
always@(*)begin
    //状态转移
    case(state)
        xxx:
            next_state=XXX;
    endcase
end
always@(posedge clk,posedge reset)begin
    if(reset==1)begin
        state=XXX;
    end
    else begin
        state=next_state; //状态存储
        //对于中间变量的改变要放在这里，
        if(next_state==XXX)begin
            //do something
        end
        else
            //do something
    end
end
assign out=(state==target_state);//Moore
assign out=(state==target_state&&in==target_in);//Mealy
```

注意一定要给state和next_state的寄存器设置好位数，如果没设置，编译器和程序运行都不会报错

verilog默认自动截取部分填进去。。

对于要使用的整数变量integer，**在一开始的时候要赋初值**，可以在定义的时候直接赋值，否则会出bug，会出现一堆未知状态。

```
integer i=0;
```

寄存器状态的管理

题目要求

1.设计说明

请用Verilog编写一个有限状态机，实现一个饮料售卖机控制器。假设饮料价格为2元。投币器只能接受5角和1元的硬币。

当顾客投入的硬币没有达到饮料的价格(2元)时，顾客不能得到饮料，drink信号保持低电平；如果顾客按下了退币按钮，售卖机应当把顾客投入的钱返还给顾客，此时back信号应当变为相应的值，且保持一个周期。

当顾客投入的硬币大于或等于饮料的价格时，可以获得饮料，drink信号变为高电平，且保持一个周期；如果投入的硬币大于饮料价格，售卖机还应当向顾客返还相应的钱数，此时back信号应当变为相应的值，且保持一个周期。

注意：在出货的那个周期，仍然可以投币

2、模块规格

模块名：drink

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号，回归初始状态，所有输出为0
coin[1:0]	I	投入的硬币，2'b00代表未投币，2'b01代表投入5角，2'b10代表投入1元，2'b11代表按下退币按钮
drink	O	顾客是否能获得饮料，1代表是，0代表否
back[1:0]	O	售货机返还的钱数，2'b00代表不返还，2'b01代表5角，2'b10代表1元，2'b11代表1.5元

3、功能要求

每个时钟上升沿，状态机从coin中读入一个2位的二进制数字。需要在此时判断顾客是否可以获得饮料，以及售货机是否应当退还相应的钱。

由于每次输出drink或者找零信号都需要**保持一个周期**，那么就需要一个**寄存器去保存状态**，但是寄存容易出现状态改变但是没有给寄存器改变值得情况。

case

case一定要写default 否则会有些状态继承原来的值。

if

if一定要写else 否则会有些状态不会改变但它又是在那个情况下应该变的。

在这个题里面我使用的drink2(说个有意思的有时候**1和L的小写容易混淆 还de不出bug 少用1作为变量名吧**)和back1([1:0])去作为drink和back的寄存器

```
module drink(
    input clk,
    input reset,
    input [1:0] coin,
    output drink,
    output [1:0] back
    );
    parameter S0=3'b000,S1=3'b001,S2=3'b010,S3=3'b011,S4=3'b100;
    reg [2:0] state,next_state;
    reg drink2=0;
    reg [1:0] back1=2'b00;
    always@(*)begin
        case(state)
            S0:
            if(coin==2'b00)begin
                next_state=S0;
            end
            else if(coin==2'b01)begin
                next_state=S1;
            end
            else if(coin==2'b10)begin
                next_state=S2;
```

```
            end
            else if(coin==2'b11)begin
                next_state=S0;
            end
            S1:
            if(coin==2'b00)begin
                next_state=S1;
            end
            else if(coin==2'b01)begin
                next_state=S2;
            end
            else if(coin==2'b10)begin
                next_state=S3;
            end
            else if(coin==2'b11)begin
                next_state=S0;
            end
            S2:
            if(coin==2'b00)begin
                next_state=S2;
            end
            else if(coin==2'b01)begin
                next_state=S3;
            end
            else if(coin==2'b10)begin
                next_state=S0;
            end
            else if(coin==2'b11)begin
                next_state=S0;
            end
            S3:
            if(coin==2'b00)begin
                next_state=S3;
            end
            else if(coin==2'b01)begin
                next_state=S0;
            end
            else if(coin==2'b10)begin
                next_state=S0;
            end
            else if(coin==2'b11)begin
                next_state=S0;
            end
            default next_state=S0;
        endcase
    end
```

```
always@(posedge clk,posedge reset)begin
        if(reset==1)begin
            state=S0;
        end
        else begin
            if(coin==2'b11)begin
                case(state)
                S0: begin
```

```
                back1=2'b00;
                drink2=0;
                end
                S1: begin
                back1=2'b01;
                drink2=0;
                end
                S2: begin
                back1=2'b10;
                drink2=0;
                end
                S3: begin
                back1=2'b11;
                drink2=0;
                end
                default begin
                back1=2'b00;
                drink2=0;
                end
                endcase
            end
            else begin
                if(next_state==S0)begin
                    case(state)
                    S0: begin
                        drink2=0;
                        back1=2'b00;
                    end
                    S1: begin
                        drink2=0;
                        back1=2'b00;
                    end
                    S2: begin
                        drink2=1;
                        back1=2'b00;
                        end
                    S3: begin
                        drink2=1;
                        back1=(coin==2'b01)?2'b00:2'b01;
                    end
                    default begin
                    drink2=0;
                    back1=2'b00;
                    end
                    endcase
                end
                else begin
                drink2=0;
                back1=2'b00;
                end
            end
            state=next_state;
        end
    end
    assign drink=drink2;
```

```
    assign back=back1;
endmodule
```

状态转移就不说了，主要是在时钟上升沿控制的always块中，**所有的if-else的结果都要把两个寄存器带着清零！！！**

还有default**要有清零**，以及case中的一些无关case，**都要把两个寄存器带着一起赋值或者清零**，**不然就寄了！！！**

总结就是

- 所有的if都要有else
- 所有的case都要有default
- 所有的case的中间情况都要兼顾所有作为最终输出的寄存器