结构化建模

结构化建模的过程类似于将元件连接起来形成一个完整的电路模块的过程。

在verilog中声明一个元件需要指明模块的输入输出端口和对应的信号。最常见的语法如下：

```
模块名  实例名（端口信号映射）
```

端口信号映射格式有两种：

1. 位置映射：模块名 实例名（信号1，信号2，... ），其中信号n对应为模块被实例化后的第n个端口。
2. 名映射：模块名 实例名（ .端口名a（信号1），.端口名b（信号2）... ），其中信号n对应其前的端口名。

Verilog中为我们准备好的一系列基本功能元件称为原语，主要包括：

and,or,nand,nor,xor,nxor这些n输入原语和**not**这一n输出原语。

这两种原语都**只能通过位置映射**来进行连接，其中n输入原语对应的端口顺序为（输出，输入1，输入2，... ）而n输出原语对应的端口顺序为（输出1，输出2，...，输出n，输入）。即**输出在前，输入在后**。

在实例化元件时，wire类型的信号可以连接到任何端口上，而reg类型的信号只能被连接到输入端口上；对应的，在声明元件时可以将任何端口声明为wire类型，而只能将输出端口声明为reg类型。

例如实例化一个加法器：

```
module Adder(
  input a;
  input b;
  input cin;
  output sum;
  output overflow;
  );
  wire s1,s2,s3;
  xor xor1(sum,a,b,cin);
  and and1(s1,a,b);
  and and2(s2,a,cin);
  and and3(s3,b,cin);
  or or1(overflow,s1,s2,s3);
endmodule
```

行为级描述

当我们使用行为级描述时，我们不再关注具体到门级电路的实现，而是只关注输入和输出之间的逻辑关系。而具体的电路实现则交给软件去完成，这个过程就叫做综合（synthesis），在这个层级，我们更加关注电路要干什么，而不是怎么做。

行为级描述的方法主要有以下两种：

1. 通过assign语句对电路连续赋值。
2. 通过initial结构，always结构和过程控制语句描述电路。

assign连续赋值语句

其常见形式为`assign signal = expression;`

其中signal必须是wire型数据，而expression则是数据和运算符组成的表达式。

所谓连续赋值，即为当expression的值发生变化时，signal的值也会发生变化。一般来说，assign语句综合出来的电路是右侧表达式化简之后形成的逻辑门组合。

过程控制语句与有关结构

一个电路的输出可以由当前的输入信号决定，也可以由输入信号和电路当前的状态共同决定。我们可以通过**reg**变量来存储输入信号或者是电路当前的状态，通过描述不同条件下这些变量的变化规律来描述这个电路。

值得注意的是，在综合过程中，每个reg变量不一定对应一个寄存器，具体的电路实现将有软件考虑决定。在这里的reg只是一个抽象的变量，而非现实中的一个寄存器。

一般情况下，我们可以有两种方法来将变量和模块的输出建立起关系

1. 使用assign语句将模块的输出赋值给变量（wire类型）
2. 直接在output端口声明为reg型

过程控制语句不能存在于模块当中，大部分情况下，他们存在于always或者initial结构当中。而一个模块中可以存在多个此种结构，在执行时将同时执行。

两种基本结构和语句块

Initial结构

形式为`initial 语句（块）`，在仿真时刻为0时开始执行，并只执行一次，通常用于电路的初始化。

always结构

形式为`always@（敏感条件列表） 语句（块）`，从仿真的0时刻开始执行。

敏感条件列表通常由多个敏感条件通过or或者，连接而成，当**其中任意一个敏感条件被触发**时，将执行always中的语句（块）。

敏感条件分为两种：即边沿敏感或电平敏感，一般不可以混用。

其中，边沿敏感包括`posedge 信号名`和`negedge 信号名`，分别表示信号上升沿和下降沿时会执行always中的语句，一般用于时序逻辑。电平敏感条件的格式为信号名，表示在该信号的电平发生变化时**（不是变成高电平时）**执行always中的语句，一般用于组合逻辑。

在描述组合逻辑时，可以使用`always@(*)`来表示**当always包含的语块中涉及到的任意一个驱动信号的电平发生变化**时都执行always中的语句。

如果省略掉了`@()`，则表示无条件的执行always中的语句，通常用于配合延迟语句在testbench中编写时钟，即：

```
always #5 clk = ~clk;
```

表示时钟的周期为10个时间周期。

不要在多个always块中对一个变量进行赋值

块语句分为两种：顺序块和并行块。

其中顺序块以begin作为开始，end作为结尾（相当于C语言中的大括号），其中的语句逐条执行，包括延迟语句也是相对于上一条语句执行结束的仿真时间

并行块则以fork作为开始，join作为结尾，其中的语句并行执行。

块语句可以嵌套。

例：

```
initial begin
  // do something
end
always@(posedge clk) begin
  // do something
end
```

常见的结构控制语句同C语言：

if-else结构，while结构，for结构，注意大括号需要改变为begin和and（fork和join）

常用数据类型

wire型

wire型数据可类比于电路中的导线，多用于表示组合逻辑信号。本身不存储数据，而是随着输入信号的改变而改变。**一般使用assign语句对wire型数据进行驱动**

wire型的数据分为标量（1位）和向量（多位）两种。**可以在声明过程中使用范围指示器指明位数**，例如`wire[31:0] a;`。冒号两侧分别表示最高有效位和最低有效位。在访问时，可以使用`a[7:4]`的方式取出a的第7-4位数据。

一般地，在使用wire型数据之前应当先声明它。但是如果在模块实例的端口信号列表中使用了一个未声明的变量，则会将其默认定义为1位的wire变量。

注意，与C语言不同，对于wire型变量a，`assign a = a + 1;`是不合法的。

reg型

reg型是寄存器数据类型，具有存储功能。也分为**标量**和**向量**。一般在always块中使用reg型变量，通过赋值语句改变寄存器的值。

需要注意的是，**reg型变量不能使用assign赋值**，而且reg型变量也并**不一定被综合成寄存器**。

通过reg数据类型建模存储器

例如`reg[31:0] mem[0:1023];`，其中前面的中括号内为位宽，后面的中括号内为存储器数量。如果需要访问的时候，例如`mem[2]`即可访问mem中的第三个元素。

verilog中没有多维数组

数字字面量

verilog中数字字面量可以按二进制（b），八进制（o），十六进制（h），十进制（d）表示。

数字的完整表达是`<位宽>'<进制><值>`,例如10'd100。省略位宽是采用默认位宽（与机器有关，一般为32位），省略进制时**默认为十进制**，值部分可以用下划线分开提高可读性，如`16'b1010_1011_1111_1010。`

verilog中除了普通的数字以外，还有两个特殊的值：`x`和`z`。x为不定值，当某一二进制位的值不能确定时出现，变量的默认初始值时`x`。`z`时高阻态，代表没有连接到有效输入上，对于位宽大于1的数据类型，`x`和`z`均**只可在部分位上**出现。

integer型

一般为32位，与C语言中的int类型类似。

parameter型

`parameter`类型用于**在编译时确认值的常量**，通过形如`parameter 标识符 = 表达式;`的形式进行定义，例如`parameter width = 8;`在实例化模块时，可以通过参数传递改变在被引用模块实例中已定义的参数（模块的实例化将在后面的章节进行介绍）。`parameter`虽然看起来可变，但是属于常量（在编译时就有一个确定的值）。

`parameter`可以用于在模块实例化时指定数据位宽等参数，便于在结构相似、位宽不同的模块之间实现代码复用。

组合建模常见语法

assign语句

assign语句是连续赋值语句，作用是**用一个信号来驱动另一个信号**。其中assign语句被赋值的信号**要求为wire类型（也可以通过位拼接得到）**。

assign语句不能在initial和always块中使用

assign语句经常与三目运算符配合使用建模组合逻辑

常用的运算符

1. 大部分的运算符和C语言都没有区别，这里只讲一些有区别的地方，其余部分由不再赘述
2. **逻辑右移运算符>>与算数右移运算符>>>**

   他们的区别在于前者在最高位补0，后者在最高位补符号位（无符号时与逻辑右移相同）。
3. **相等比较运算符`==`和`===`，以及`!=`与`!==`**

   `==`和`!=`可能由于不定值x和高阻值z的出现导致结果为不定值x，而`===`和`!==`的会将x与z也参与比较，因此结果一定是确定的0或1。
4. **阻塞赋值`=`和非阻塞赋值`<=`**

   不同于assign语句，这两种赋值方式被称为过程赋值，通常会出现在initial和always块中，位**reg型**变量赋值。这种赋值类似于C语言中的赋值，仅在一个时刻执行。注意，**在描述时许逻辑时要使用非阻塞式赋值`<=`**。

5. **位拼接运算符{}**

   这个运算符可以将几个信号的某些位拼接起来，例如`{a,b[3:0],w,3'b101};`可以简化重复的表达式，例如`{4{w}};`，还可以嵌套，比如`{b,{3{a,b}}};`。

6. **缩减运算符**

   运算符&，｜，^可以作为单目运算符使用，是对操作数的每一位进行汇总运算，如对于`reg[31:0] B;`中的B来说，`&B`代表将B的每一位进行与运算得到的最终结果。

时序建模常见语法

always语句和initial语句

同一个模块中的顺序块将会并行执行。

`always@(*)`表示当always语块中的任意一个驱动信号发生变化时执行always中的语句。

if语句

if语句只能出现在顺序块当中，其后的分支也只能是语句或顺序块；为了避免意料之外的锁存器的生成而导致错误，**要为所有的if语句都写出对应的else分支**。

case语句

case语句也只能出现在顺序块当中，其中的分支也只能是语句或者顺序块。并且case语句在分支结束之后不会落入下一个分支，而会自动退出。

例如

```
always @(posedge clk)begin
  case(data)
    0: out <= 4;
    1: out <= 5;
    2: out <= 2;
    3: begin
        out <= 1;
    end
    default: ;
  endcase
end
```

for语句和while语句

for语句中的循环变量通常为integer类型。其他与C语言中的实现无异，两种循环的样例如下：

```
module vote7(
    input [6:0] vote,
    output reg pass
    );
```

```
    reg[2:0] sum; // sum为reg型变量，用于统计赞成的人数
    integer i; // 循环变量
    always @(vote) begin // 此处使用always建模组合逻辑
        sum = 3'b000; // sum初值为0
        for (i = 0;i < 7;i = i + 1) begin // for语句
            if (vote[i]) sum = sum + 1; // 只要有人投赞成票，则sum加1
        end
        if (sum >= 3'd4) pass = 1'b1; // 若大于等于4人赞成，则表决通过
        else pass = 1'b0;
    end
endmodule
```

```
module count1s_while(
    input clk,
    input [7:0] rega,
    output reg [3:0] count
    );

     always @(posedge clk) begin: count1 // 命名顺序块，建模时序逻辑
        reg[7:0] tempreg; // 用作循环执行条件表达式
        count = 0; // count初值为0
        tempreg = rega; // tempreg初值为rega
        while (tempreg) begin // 若tempreg非0，则执行以下语句
            if (tempreg[0]) count = count + 1; // 只要trmpreg最低位为1，则count加1
            tempreg = tempreg >> 1; // 逻辑右移1位
        end
    end
endmodule
```

时间控制语句

时间控制语句常常用来编写testbench。这个语句通过#来实现延迟，格式为`#时间`，当延时语句出现在顺序块当中时，它会在上一条语句执行结束之后开始延时，它后面的语句会在延时结束之后继续执行。

例如：

```
#3;          // 延迟 3 个时间单位
#5 b = a;    // b 为 reg 型，延迟 5 个时间单位后执行赋值语句
always #5 clk = ~clk;    // 每过 5 个时间单位触发一次，时钟信号反转，时钟周期为 10 个时间单位
assign #5 b = a;         // b 为 wire 型，将表达式右边的值延时 5 个时间单位后赋给 b
```

Verilog其他语法

子模块的调用

在调用子模块的时候，需要将其实例化并指明端口信号。

例如，我们已经有了一个Adder模块：Adder（input a,input b,input c,output cout);

那么在调用该模块时需要`Adder adder1(x,y,z,r);`

对于这种顺序命名方式，每个接口均按顺序连接，即x对应a，y对应b……。并且不可以有空闲的端口。

这样的方式是不利于我们添加或删减模块的输入输出端口的。因此更推荐以下的命名方式：

```
Adder adder1(.a(x),.cout(r),.b(y));
```

这种方式不需要保证顺序和数量的一一对应。

阻塞赋值和非阻塞赋值

阻塞赋值`==`按顺序执行，类似于C语言中的赋值。

非阻塞赋值`<=`是并行执行的语句，当执行至第一个非阻塞执行语句中，系统会将所有阻塞赋值语句当前状态下的右值均记录下来，并在语块结束之后同时进行赋值。

有符号数的处理方法

在verilog中，`wire`和`reg`数据类型默认为无符号，当需要进行符号操作时需要使用`$signed()`。

例如，如果我们要比较两个数的大小，当存在负数的时候，我们就需要使用该语句，否则verilog将会把用补码存储的数字看作为纯二进制数字。

值得注意的是，在对无符号数和符号数同时进行操作的时候，verilog会自动地做数据类型匹配。**将符号数向无符号数**进行转化。因此在进行有符号数的比较时，需要将两者均加上`$signed()`。

特殊的，对于移位运算符而言，其右侧的操作数总是被视为无符号数，并且不会对运算结果的符号性产生任何影响。

来看一个逆天符号判定

```
module a2(
    input clk,
    input reset,
    input [3:0] a,
    input [3:0] b,
    output [3:0] ans1,
    output [3:0] ans2,
    output [3:0] ans3
    );
     assign ans1 = (1'b1==1'b1) ? a>>>b : 0;
     assign ans2 = (1'b1==1'b1) ? $signed(a)>>>b : 0;
     assign ans3 = (1'b1==1'b1) ? $signed(a)>>>b : 4'b0;

endmodule
```

Testbench部分内容如下所示：

```
    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 0;
        a = 0;
        b = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        a=3;
        b=1;
        #2;
        a=-2;
        b=1;

    end
```

T1. 在101ns和103ns时，ans1的值分别为（ ）。

A 4'b0001;4'b0111 B 4'b0001;4'b1111 C 4'b1001;4'b0111 D 4'b1001;4'b1111

T2. 在101ns和103ns时，ans2的值分别为（ ）。

A 4'b0001;4'b0111 B 4'b0001;4'b1111 C 4'b1001;4'b0111 D 4'b1001;4'b1111

T3. 在101ns和103ns时，ans3的值分别为（ ）。

A 4'b0001;4'b0111 B 4'b0001;4'b1111 C 4'b1001;4'b0111 D 4'b1001;4'b1111

请输入三道题的答案选项：（例："ABC"，无需输入引号）

可以发现ans2和ans3是不一致的。

这是因为verilog非常逆天的符号判定与强制转换。具体如下：

同学你好，针对符号，表达式的最终计算主要有两步：① `符号确定` ② `向内传播` 。

根据 `IEEE 1364-2005` Verilog语言标准第5.1节、第5.5节：

- 移位运算符(<<、>>、>>>等）生成的表达式（形如 `i >>> j` ）的符号由 `i` 的符号决定， `j` 恒为无符号且自决定。
- 三目运算符生成的表达式（形如 `i ? j : k` ）中 `i` 为自决定表达式

> — **Decimal numbers** are **signed**.
> — **Based_numbers** are **unsigned**, except where the s notation is used in the base specifier (as in "4'sd12").

可知 `$signed(a)>>>b` 、 `0` **有符号，** `4b'0` **无符号。**

> — The sign and size of any **self-determined operand** are determined by the operand itself and **independent of the remainder of the expression**.
> — For **nonself-determined operands**, the following rules apply:
> — **If any operand is unsigned, the result is unsigned**, regardless of the operator.
> — If all operands are signed, the result will be signed, regardless of operator, except when specified otherwise.

由上得，简单来说，如果一个表达式除了自决定子表达式外，剩下的子表达式中任意一个被确定为无符号，那么整个表达式都被确定为无符号，否则有符号（与 `向内传播` 相反， `符号确定` 这一过程是**由内向外**的）。易得 `condition ? $signed(a)>>>b : 0` **整体**被确定为有符号、 `condition ? $signed(a)>>>b : 4'b0` **整体**被确定为无符号。

我们已经完成了第①步 `符号确定` ，然后开始 `向内传播` ：

> — Propagate the type and size of the expression (or self-determined subexpression) back down to the **context-determined operands** of the expression. In general, any context-determined operand of an operator shall be the same type and size as the result of the operator.
> — When propagation reaches a **simple operand** as defined in 5.2 (a primary as defined in A.8.4), then that operand shall **be converted to the propagated type and size**. If the operand must be extended, then it shall be sign-extended only if the propagated type is signed.

对确定为无符号的表达式 `condition ? $signed(a)>>>b : 4'b0` ，这种无符号的符号性被向内传播至 `$signed(a)>>>b` ，再传播至原子表达式 `$signed(a)` ，于是 `$signed(a)` 被强制转换成无符号。最终便得到了结果。

宏定义的简单使用

类似 C 语言，Verilog HDL 也提供了编译预处理指令。下面我们对其中的宏定义部分作一简要介绍。

在 Verilog HDL 语言中，为了和一般的语句相区别，编译预处理命令**以符号 `（反引号，backtick）开头**（位于主键盘左上角，其对应的上键盘字符为 ~。注意这个符号不同于单引号）。这些预处理命令的有效作用范围为定义命令之后到本文结束或到其他命令定义替代该命令之处。

宏定义用一个指定的标识符(即名字)来代表一个字符串，它的一般形式为：`define 标识符(宏名) 字符串(宏内容)`。它的作用是指定用标识符来代替字符串，在编译预处理时，把程序中该命令以后所有的标识符都替换成字符串。举例如下：

```
`define WORDSIZE 8
// 省略模块定义
 reg[1:`WORDSIZE] data;
// 相当于定义 reg[1:8] data;
```

注意，**引用宏名时也必须在宏名前加上符号 `**，以表明该名字是经过宏定义的名字。

Verilog：设计、开发与调试

编写可综合的Verilog代码

Verilog语言最终需要综合为由门电路组成的硬件，才是有意义的。但是在设计之初，Verilog语言是被当作一门仿真语言，因此其中的一些特性导致了最后写出的代码不可以被综合成实际电路。因此，在编写过程中，如果我们需要最终生成可综合的代码，就应该避免使用这种特性。

值得注意的是，**testbench不需要被转化为硬件，因此以下建议不适用于testbench的编写。**

不要使用initial块、不要为寄存器赋初值

如果想要初始化，请添加一个reset复位信号（类似你在logisim里做的那样），例如：

```
reg v,m;
always@(posedge clk) begin
  if (reset) begin
    v <= 6;
    m == 1;
  end
end
```

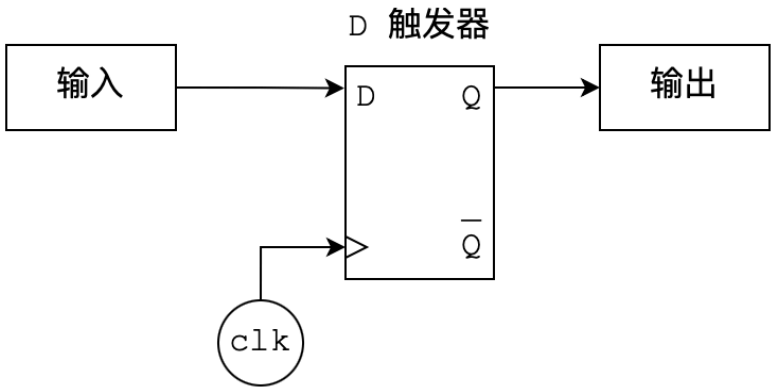
对应的，我们需要在testbench的编写中进行初始化，例如：

```
reg clk = 0;
reg reset = 0;
always #5 clk = ~clk;
initial begin
  reset = 1;
  # 10; // 延时一个时间周期
  reset = 0;
  // do something
end
```

如果我们在module中使用了initial块，或为寄存器赋初值，那么综合时均将被忽略掉。

一个寄存器只能在一个always块中赋值一次

Verilog综合时，寄存器通常会被综合为D触发器：



因此只有一个时钟输入和数据输入。所以每个寄存器只能属于一个时钟域（指只能由一个时钟信号进行驱动）。例如以下代码会使寄存器属于两个时钟域，因此是不可综合的：

```
reg a;
wire b,c;
always @(posedge clk_1) begin
  a <= b;
end

always @(posedge clk_2) begin
  a <= c;
end
```

除了注意时钟域的归属外，我们也需保证在每个时间周期中，寄存器至多被赋值一次，不能重复赋值。例如，以下的代码也是不可综合的：

```
reg n;

always @(posedge clk) begin
  if (a)
    n <= 1'b1;

  // do something

  n <= 1'b0;
end
```

组合逻辑相关注意事项

一般将代码分为时序逻辑和组合逻辑两个部分。一般情况下对于always块的敏感条件列表：时序逻辑`@(posedge clk)`，而组合逻辑使用`@(*)`来表达。一般情况下可以通过遵循以下准则。

1. 在时序逻辑中使用非阻塞赋值(<=)；在组合逻辑中，永远使用阻塞赋值（=）；
2. 每个组合逻辑运算结果仅在一个`always @(*)`中修改；
3. 在`always @(*)`中，为每个运算结果赋初值，避免latch的产生。

一段示例代码如下。

```
// 注意以下 count_n 并不是一个寄存器，而是由组合逻辑生成的运算结果；count 才是实际存放计数值的寄存器。
reg [4:0] count_n, count;
wire add, set;
wire [4:0] set_value;

always @(*) begin
    count_n = count; // 修改了 count_n，因此先赋初值
    if (set)
        count_n = set_value;
    if (add)
        count_n = count_n + 1;
    // 阻塞赋值类似于 C 语言，按顺序执行，以最后赋值的为准
end

always @(posedge clk) begin
    if (reset)
        count <= 0;
    else
        count <= count_n;
end
```

尽量使用位运算来代替乘除法

- 乘以2^n
  - 左移n位，例如`a * 8`可替换为`a << 3`；
  - 在变量后面拼接n个0，例如`a * 8`可替换为`{a,3'b0}`。
- 除以2^n
  - 右移n位，例如`a / 8`可替换为`a >> 3`；
  - 取变量的高位，例如`a / 8`可替换为`a[7:3]`（若a一共有8位）
- 求模2^n 取n位以后的低位，例如：`a % 8`可以替换为`a[2:0]`。

在使用移位运算符时，请注意移位运算符的优化级问题。

Verilog代码规范与实例技巧

传送门

该传送门通往介绍代码风格的文档，本部分主要为笔者平时编写verilog程序的一些积累与总结

用位运算代替乘法

众所周知，乘法是非常消耗时间的，而位运算则要快得多。那么对于任意一个数字，该如何进行乘法运算?

例如要求a*114的结果。将114用二进制表示为0b1110010之后我们可以轻易地发现，114等于（0b1000000+0b100000+0b10000+0b10），而最终的结果也可以顺理成章的表示为（a<<6+a<<5+a<<4+a<<1)

任何整数乘法都可以像这样进行操作

位运算判断一个数是否为2的幂次或二的幂次的倍数

如果一个数是2的幂数，那么转换为二进制之后一定有且仅有一位为1。那么我们可以采取***减一做与运算***的方式来进行判断。如果一个数和它减一的结果与运算为0的话，那么这个数就是2的幂次

例如8的二进制表示为0b1000，减一之后为0b111，经过与运算之后结果为0.

而7的二进制表示为0b111，减一后为0b110，经过与运算为0b110，即6。

令该数（二的幂次）为C，则C有且仅有一个数位为1，如果被判断数在该数位的低位均为0，那么就可以表示为这个数的整数倍。

也就是说我们可以以一个非常简单的方法***将C减一（低位全部置1）再和被判断数做与运算***，若为0即为C的整数倍，若不为0则说明被判断数的低位存在1，不是C的倍数。

有限状态机的三段式描述方法

就我个人而言，更喜欢一段式的写法（因为更贴近C语言的思维）。但如果按硬件思维来考虑的话，三段式更加容易修改，并且更加简洁明了。

三段式分别指：

1. 状态跳转逻辑：根据输入信号以及当前状态确定状态的次态状态

2. 触发器实现：在时钟边沿实现状态寄存器的跳变以及状态复位
3. 输出逻辑：根据当前状态（输入）进行输出

例如：

```
module top_module (
  input clk,
  input in,
  input areset,
  output out
);
  parameter A = 0,B = 1;
  reg state;
  reg next;
  //用一个next变量来存储下一个状态
  always @(*) begin
    case (state)
      A: next = in ? A:B;
      B: next = in ? B:A;
    endcase
  end
  //状态转移逻辑
  always @(posedge clk,posedge areset) begin
    if(areset)  state <= B;
    else  state <= next;
  end
  //触发器
  assign out = (state == B);
  //输出逻辑
endmodule
```

很容易地发现，上述代码描述了一个Moore型状态机。

但是我们知道Mealy型状态机是提前一个时间周期进行输出的。那么我们可以非常简单在输出逻辑当中加入input的判断来进行输出（换句话讲在输出逻辑中进行一次状态转移再进行输出），这样的话输出就会和Mealy型状态机一致。

注意！：上述转换为Mealy型状态机的方法只是投机取巧，并没有按照Mealy型状态机的原理进行设计，不要对理论知识产生影响