**P1**



**module splitter(**

**input [31:0] A,**

**output [7:0] O1,**

**output [7:0] O2,**

**output [7:0] O3,**

**output [7:0] O4**

**);**

**assign O1[7:0] = A[31:24];**

**assign O2[7:0] = A[23:16];**

**assign O3[7:0] = A[15:8];**

**assign O4[7:0] = A[7:0];**

**endmodule**

**搞反了 input [31:0] A变成错误的input A[31:0]**

在看教程的时候看到了case, casez与casex的语句，突然发现自己不记得三者的区别了...... 于是查了一下资料，重新温习了一下三者的区别。特记录于此，既帮助自己记忆，也方便大家~

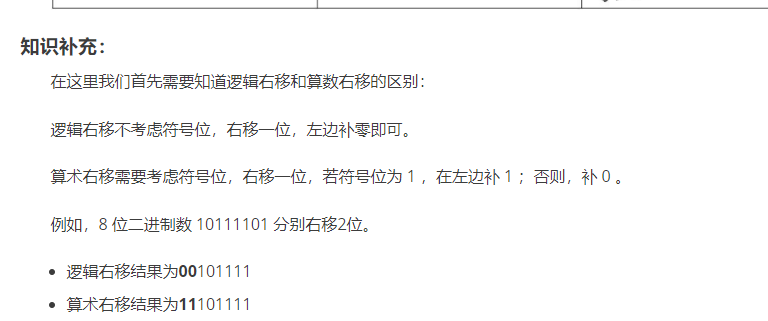
case，casez与casex三者的语法结构是一模一样的，区别在于三者匹配的方式不同。

对于case，四个值1, 0, x, z对应关系是一一对应，即在case的条件分支中，每个值只能与自身匹配。例如1000是不能与100x、100z等匹配的，因为末尾的0与x/z不匹配。

对于casez，四个值1, 0, x, z除了可以与自身匹配外，高阻态z还可以和1,0,x匹配。也就是说，z可以和任何值匹配。例如1000是可以与100z匹配的。

而对于casex，四个值1, 0, x, z除了可以与自身匹配外，高阻态z、不定值x都可以和任何值匹配。这是一个比较容易混淆的地方，因为casex不仅x能与任何值匹配，z也可以和任何值匹配。例如1000可以与100x、100z匹配。

以上就是case、casez与casex的区别，**主要是casex容易忘记z也可以和任何值匹配。**希望能对大家有所帮助~





**module alu(**

**input [31:0] A,**

**input [31:0] B,**

**input [2:0] ALUOp,**

**output reg[31:0] C**

**);**

**always @ (A or B or ALUOp) begin**

**case(ALUOp)**

**3'b000: C = A+B;**

**3'b001: C = A-B;**

**3'b010: C = A&B;**

**3'b011: C = A|B;**

**3'b100: C = A>>B;**

**3'b101: C = $signed(A) >>> B;**

**3'b110:;**

**3'b111:;**

**default:;**

**endcase**

**end**

**endmodule**

1. **注意case语句只能在模块中使用**
2. **最好设置成output是reg**
3. **注意生成锁存器，把所有情况都考虑进去**
4. **注意always @的@不要漏掉**



**善用讨论区**

**module ext(**

**input [15:0] imm,**

**input [1:0] EOp,**

**output reg [31:0] ext**

**);**

**always @ (imm or EOp) begin**

**case(EOp)**

**2'b00: begin**

**ext = $signed(imm); //用自带的拓展功能**

**end**

**2'b01: begin**

**ext = imm; //用自带的拓展功能**

**end**

**2'b10: begin**

**ext = {imm[15:0],{16{1'b0}}}; //用拼接符**

**end**

**2'b11: begin**

**ext = {{14{imm[15]}},imm[15:0],2'b00}; //用拼接符**

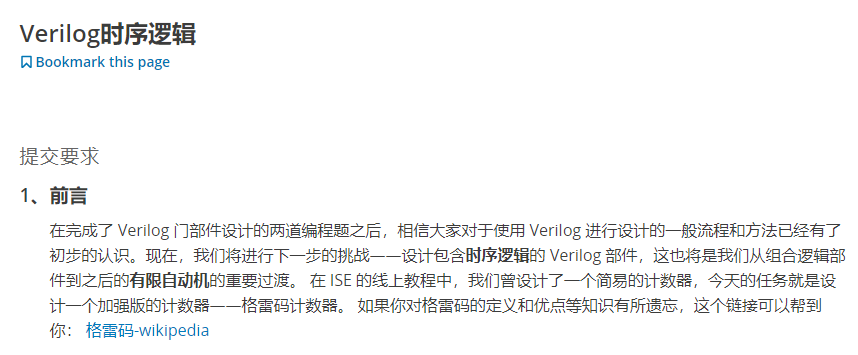
**end**

**default:;**

**endcase**

**end**

**endmodule**





**module gray(**

**input Clk,**

**input Reset,**

**input En,**

**output reg [2:0] Output,**

**output reg Overflow**

**);**

**reg [2:0] Binary;**

**initial begin**

**Output <= 0;**

**Overflow <= 0;**

**Binary <= 0;**

**end**

**always @ (posedge Clk)begin**

**if(Reset)begin**

**Output <= 0;**

**Overflow <= 0;**

**Binary <= 0;**

**end**

**else begin**

**if(En)begin //别忘了有使能信号**

**if(Binary == 3'b111)begin //先判断是否发生溢出**

**Binary <= 0;**

**Output <= 0; //不会自动清零**

**Overflow <= 1;**

**end**

**else begin**

**Binary = Binary + 1; //为什么这里取阻塞赋值呢，因为当这个else开始时，无论是二进制还是格雷码计数器都应该是+1的了，而格雷码从公式上看依赖于二进制的加1，所以用阻塞式赋值传递给下面**

**Output <= (Binary >> 1)^Binary;**

**end**

**end**

**end**

**end**

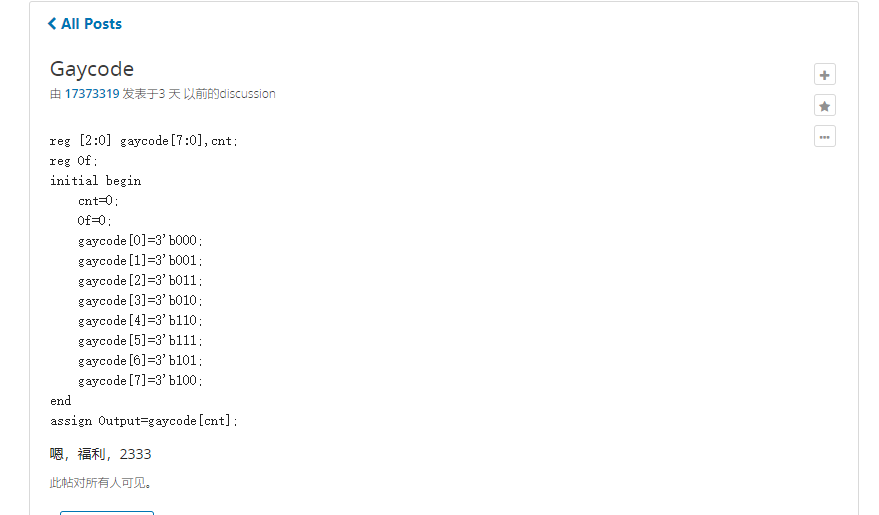
**endmodule**

**wire只能被assign赋值，initial只能在always和initial中赋值**

**注意看仿真波形！！！**

**非阻塞式赋值指的是那一个module所有<=右边的式子执行完后才赋值，而阻塞式赋值则会对下面造成影响。**



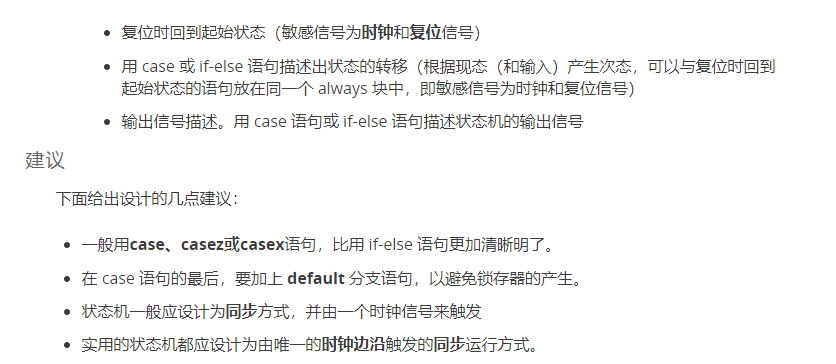
**打表法**

## 有限状态机的设计流程

对于状态机的概念和用法在理论课和前面部分的学习中接触到了很多，所以在此不再过多叙述。

       在 Verilog HDL 中可以用许多种方法来描述有限状态机，最常用的方法是用 ****always 语句****和 ****case 语句****。下面具体说明用 Verilog 设计有限状态机的一般步骤：

1. **逻辑抽象，得出状态转换图。**
2. **状态化简（该步骤可以省略）**，如果在状态转换图中出现两个一样的状态则需要将其合并为一个状态，以得到最简的状态转换图。
3. **状态分配，也就是状态编码。**通常有很多种编码方式，比如 Gray 编码、独热编码等。在实际电路中，需综合考虑电路复杂度与电路性能之间的折中。在这里的设计中没有用到特别复杂的电路逻辑，所以大家可以自行决定用哪种编码方式。（**不过推荐使用**独热**编码方式**，因为 FPGA 有丰富的寄存器资源，门逻辑相对缺乏，采用独热编码可以有效提高电路的速度和可靠性，也有利于提高器件资源的利用率。）



* 一般用****case、casez或casex****语句，比用 if-else 语句更加清晰明了。
* 在 case 语句的最后，要加上 ****default****分支语句**，以避免锁存器的产生。**
* 状态机一般应设计为****同步****方式，并由一个时钟信号来触发
* 实用的状态机都应设计为由唯一的****时钟边沿****触发的****同步****运行方式。

**在《Verilog数字系统设计教程》P57页上方也有缺少default生成锁存器的例子**

**FSM\_EXAMPLE**

**module fsm(Clock,Reset,A,K2,K1,state);**

**input Clock,Reset,A;**

**output K2,K1;**

**output [1:0] state;**

**reg K1,K2;**

**reg [1:0] state;**

**parameter Idle = 2'b00, //idle 闲置**

**Start = 2'b01,**

**Stop = 2'b10,**

**Clear = 2'b11;**

**always @ (posedge Clock)begin**

**if(!Reset)begin**

**state <= Idle;**

**K2 <=0;**

**K1 <=0;**

**end**

**else**

**case(state)**

**Idle: if(A) begin**

**state <= Start;**

**K1 <=0;**

**end**

**else begin**

**state <= Idle;**

**K2 <=0;**

**K1 <=0;**

**end**

**Start: if(!A) state <= Stop;**

**else state <= Start;**

**Stop: if(A) begin**

**state <= Clear;**

**K2 <=1;**

**end**

**else begin**

**state <= Stop;**

**K2 <= 0;**

**K1 <= 0;**

**end**

**Clear: if(!A) begin**

**state <= Idle;**

**K2 <= 0;**

**K1 <= 1;**

**end**

**else begin**

**state <= Clear;**

**K2 <= 0;**

**K1 <= 1;**

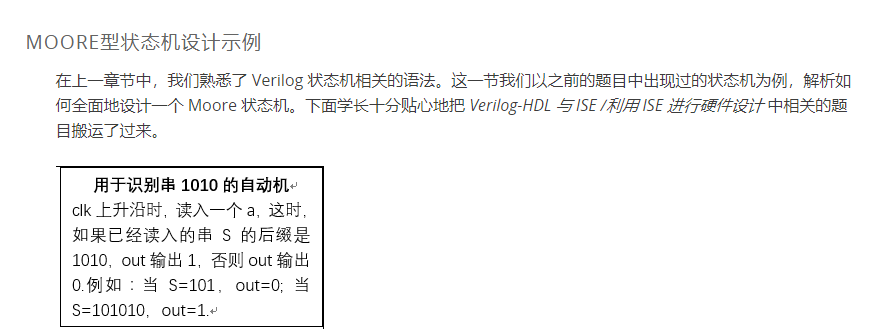
**end**

**default: state <= 2'bxx;**

**endcase**

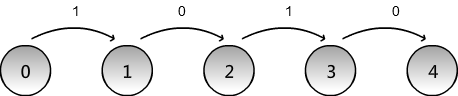
**end**

**endmodule**



**设计状态机的第一步，我们需要先分析问题，确定状态的数量和定义。**

这道题我们需要设计一个识别1010的状态机。要确定它的状态数量，我们可能直觉认为要设置5种状态，分别处理读入了∅,1,10,101,1010五种情况。这种想法是可行的，我们先根据这个直观的理解画出部分状态转换图，其中state=0,1,2,3,4分别对应“读入了∅,1,10,101,1010”。

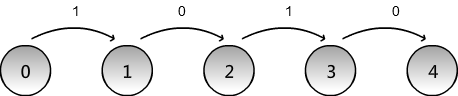


显然，如果从零时刻开始，串行输入1010，状态机会按照state=0->1->2->3->4的顺序转移，假如我们规定out=1当且仅当state=4，那么状态机可以正确识别1010的串。问题是，当匹配到101时，如果读入了一个1，得到了1011，我们该如何正确选择应该转移到哪个状态？为此，我们要用一种啰嗦的方式严格定义状态statei的意义。我们看到，所谓“读入了∅,1,10,101,1010”是一种很模糊的说法，它有歧义。例如字符流是1010，我们该说“读入了10”，还是“读入了1010”呢？因此，原来的说法不是良好的定义，因为它不能单单根据字符流来确定究竟是哪个状态。

****定义****：设目标字符串1010为str；状态机处于statei当且仅当：i>0, 设所有1<=j<=4,str[1:j]能匹配的输入字符流的后缀为****S****，str[1:i]匹配的输入字符流的后缀在****S****中最长; i=0, 当且仅当所有1<=j<=4,str[1:j]都不能匹配输入字符流的任何一个后缀。

str[1:i]的意义是str第1个字符到第i个字符的部分，比如str[1:4]就是****1010。****通过这个定义，我们就可以很好地直接根据字符流确定是哪个状态。例如读入了1010时，显然str[1:2], str[1:4]都可以匹配1010的后缀，前者对应state2，后者对应state4，然而后者匹配的后缀长度最长，因此我们确定这时应该是state=state4。

这样的定义是有好处的，有了它，我们就可以根据字符流判断是哪个状态。当然，这种状态的正确性的维护有赖于状态转移函数的选择。如果状态转移函数胡乱规定，我们就无法确定状态的正确性。

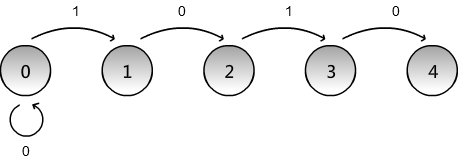


在这里，我们采用类似于动态规划的思路规划状态转移函数。思路是：每在状态之间加入一条或几条表示状态转移的边，都要保证这样的操作能维持一个****性质****——在外界输入保证变换只涉及当前已经加入的边的条件下，涉及到的状态不违背我们事先约定的定义。

现在我们有四条边，我们考虑在这四条边上进行状态转移的情况。初始状态为state=0。显然要走到4，只有一条路径0->1->2->3->4，对应输入1010。state=0时，我们回顾****定义****，输入的字符流是∅，这时谁也匹配不了，我们保持了状态定义的正确性；state=1，输入的字符流是1，str[1:i],i=1能匹配最长的后缀，而别的str[1:j]都不能匹配，我们再次保持了状态定义的正确性……接下来你可以自己推导，我们初步的状态转移设置确实维护了****定义的正确性****。

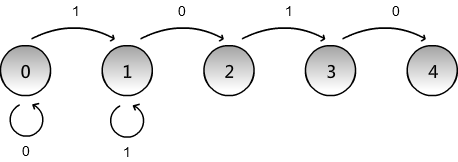
接下来，我们要补全状态转换图。当前的状态转换图是不完整的，我们在维持定义正确性的前提下补全每个状态下0,1两种输入对应的状态转移后，就得到了完整的正确的状态转换图。

首先从state=0开始考虑，我们欠缺了输入为0的情况。我们尝试加入一条边，再验证它的正确性。



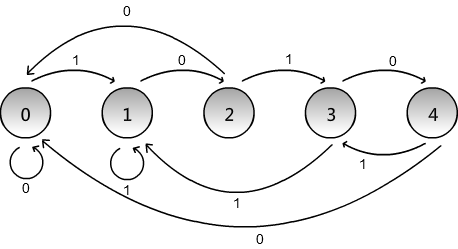
首先，我们根据前面的分析，知道假如状态转移只走之前加过的边，状态的定义是被保持的。现在我们考虑经过新加边的情况。当state=0，对任意1<=j<=4，输入字符流的末尾没有一个符合str[1:j]，新加边表示我们又读入了一个0，显然还是没有statej来匹配输入字符流的后缀。因此新加边没有影响定义的正确性。

接着我们看state=1。我们像上面一样，添加一个状态转移边并分析正确性。



回顾定义，state=1时，字符串"1"即str[1:1]匹配的输入字符流的后缀在所有状态中最长的。隐含的意思是，我们不需要考虑输入字符流的后缀有什么“101”的情况。这时我们读入了一个字符1，那么输入字符流的一个后缀是11，我们扫遍所有的statej，发现还是state1匹配的后缀最长，因此要保持在state=1。我们再次维护了上述的定义的正确性。

以此类推，我们容易得到正确的状态转移图：



最后，把状态转换图改写成对应的Verilog程序，并注意根据state维护out的值，我们就成功实现了一个能正确识别1010串的Moore自动机(学长又贴心地把程序搬运了过来)。

**module recognize0101(**

**input clk,**

**input in,**

**output reg out = 0**

**);**

**integer state = 0;**

**/\*state被期望能够正确表示当前累计读入的1010的前缀的长度。**

**比如未读入时state=0；读入到10时我们期望有state<=2\*/**

**always @(posedge clk) begin**

**case(state)**

**0:**

**begin**

**state <= in==1?1:0;**

**out <= 0;**

**end**

**1:**

**begin**

**state <= in==1?1:2;**

**out <= 0;**

**end**

**2:**

**begin**

**state <= in==1?3:0;**

**out <= 0;**

**end**

**3:**

**begin**

**state <= in==1?1:4;**

**out <= (in==0);**

**end**

**4:**

**begin**

**state <= in==1?3:0;**

**out <= 0;**

**end**

**default:;**

**endcase**

**end**

**endmodule**

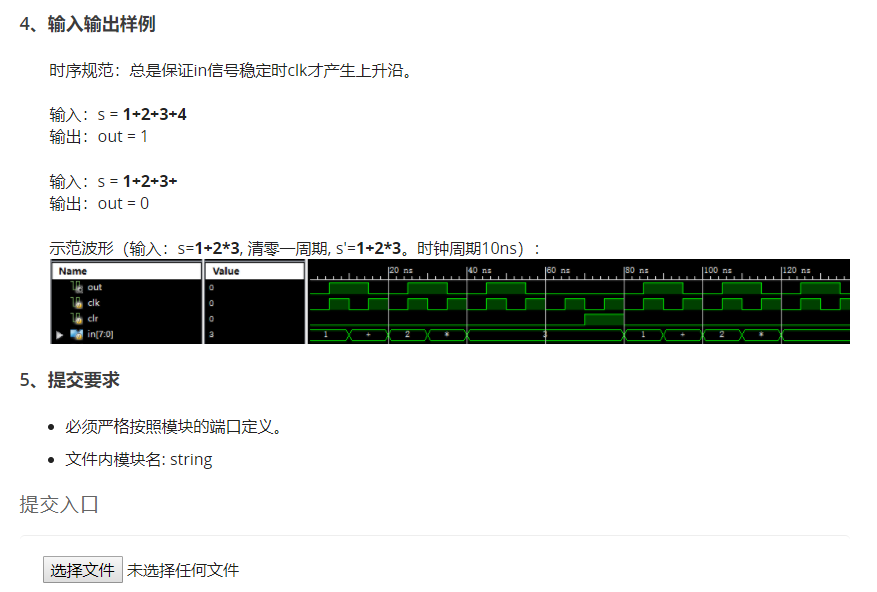
1. **分析问题，确定状态的数量和定义**

**逻辑推理，得出状态转换图**

**状态化简，合并相同的状态，注意定义好状态不要产生状态混淆**

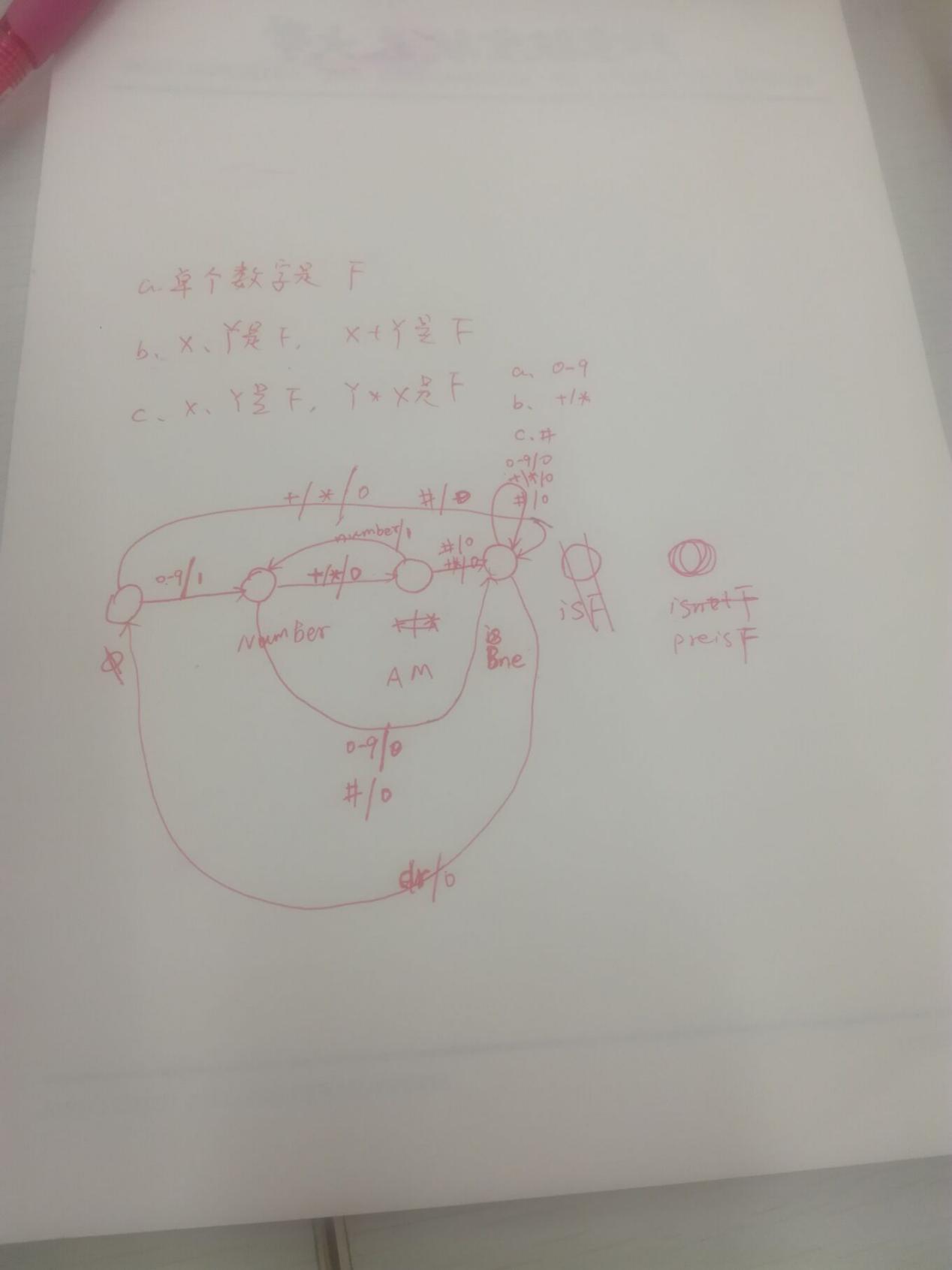
**编码分配，用parameter或者define，推荐用独热编码**

1. **补全状态图**
2. **用verilog语言描述状态图，推荐用case**





**用always检测输入信号如clk、clr的变化**

****

**module string(**

**input clk,**

**input clr,**

**input [7:0] in,**

**output reg out = 0**

**);**

**integer state = 0;**

**parameter null=0,**

**isdigit=1,**

**ischar=2,**

**illegal=3;**

**always @ (posedge clk or posedge clr)begin**

**if(clr)begin**

**state <= null;**

**out <= 0;**

**end**

**else begin**

**case(state)**

**null:**

**begin**

**if(in<=57 && 48<=in)begin**

**state <= isdigit;**

**out <= 1;**

**end**

**else begin**

**state <= illegal;**

**out <= 0;**

**end**

**end**

**isdigit:**

**begin**

**if(in==43 || in==42)begin**

**state <= ischar;**

**out <= 0;**

**end**

**else begin**

**state <= illegal;**

**out <= 0;**

**end**

**end**

**ischar:**

**begin**

**if(in<=57 && 48<=in)begin**

**state <= isdigit;**

**out <= 1;**

**end**

**else begin**

**state <= illegal;**

**out <= 0;**

**end**

**end**

**illegal: //这里用不用判断clr呢？存疑应该不用**

**begin**

**state <= illegal;**

**out <= 0;**

**end**

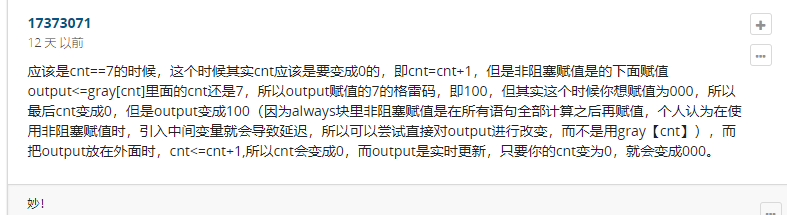
**default: state <= null;**

**endcase**

**end**

**end**

**endmodule**

1. **寄存器设置位数！！**
2. **搞反了 input [31:0] A变成错误的input A[31:0]**
3. **注意case语句只能在模块中使用**
4. **最好设置成output是reg**
5. **注意生成锁存器，把所有情况都考虑进去**
6. **注意always @的@不要漏掉**
7. **善用讨论区**
8. **wire只能被assign赋值，reg只能在always和initial中赋值**
9. **注意看仿真波形！！！**
10. **非阻塞式赋值指的是那一个module所有<=右边的式子执行完后才赋值，而阻塞式赋值则会对下面造成影响。**
11. **打表法**
12. **逻辑抽象，得出状态转换图。**
13. **状态化简（该步骤可以省略）**
14. **状态分配，也就是状态编码。**
15. **在 case 语句的最后，要加上 default 分支语句，以避免锁存器的产生。**
16. **用always检测输入信号如clk、clr的变化**
17. **习惯在每个if后加上else即使是保持原来的状态，然后case后也习惯加上default即使是不存在**
18. **记得在写的时候也把testbench写了，不然助教不让过**
19. **记得在copy相同部分代码时，把标签改了**
20. **记得parameter也是要设置位数的。**
21. **case中，值和值之间可以合并，但是default不能和任何值合并。**
22. **拼接符{4{w}}表明多少个单位外也要{}**
23. **注意表明位数的[:]里面是冒号而不是逗号，而且注意不要是中文冒号**
24. **记得初始化输出~**
25. **非阻塞赋值引入中加变量会有延迟！！此时改为阻塞赋值！**
26. **设置状态时考虑需求然后合并状态**
27. **verilog中的字符是双引号不是单引号**
28. **可以设置计数器**
29. **宏定义`define state 4’b0010 不用加分号**
30. 
31. **缩减运算符**
32. **$monitor一般只在initial块中调用。**
33. 此题是跑整个串，所以你需要额外加一个 fail 状态，表示之前的串已经不合法，之后的任何读入都还是不合法
34. 此题的状态数可以设计为4个 比较好画。

此题是异步复位，在logisim的时候，我就保持着同步复位器在异步复位的时候不取消的习惯，这里也一样。 可以写成

always @(posedge clr) /\* reset your status \*/

always @(posedge clk) begin

if (clr) /\* reset your status \*/

else /\* update \*/

用display可以检查某一段电路是否有访问过

使用测试电路的同学，在Initialize Inputs阶段，请把clr置为1，然后在初始置为0才能看到图示相同波形。

如果****不知道ASCII码****也想电路，在Verilog中可以使用“a”,"0"表示一个字符（Verilog的编译器会把字符串解析成bits）很良心的。

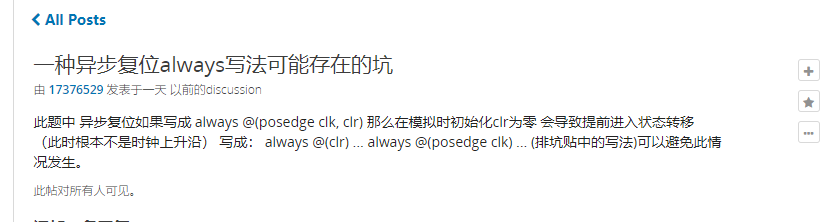
记得查看每个变量的名字和module的名字（有人被坑过）

parameter 也可以数组 （格雷码那个题有人居然直接开寄存器存常量，真的是浪费资源啊，好吧FPGA牛逼！）

parameter [7:0] lsfr\_taps [0 : 7] = '{8'd9, 8'd5, 8'd3, 8'h21, 8'd9, 8'd9, 8'd5, 8'd9};

****寄存器设置位数****，因为发现大家好像在logisim那边线路都会自动报错，如果位数不对的话。Verilog的CE信息看起来有点点小头大，所以一定要提前避免。

祝大家好运。



忘记给reg设置位数



**状态机的同步和异步复位**

**状态机的异步置位与复位：**

**异步置位与复位与时钟无关。因此把它们列入always块的事件控制括号内就能触发always块的执行。**

**always @ (posedge clk or posedge set or posedge reset)**

**begin**

**if(reset)**

**begin**

**/\*置输出为0\*/**

**end**

**else**

**if(set)**

**begin**

**/\*置输出为1\*/**

**end**

**else**

**begin**

**/\*与时钟同步的逻辑\*/**

**end**

**end**

**状态机的同步置位与复位：**

**同步置位与复位是指只有在时钟的有效跳边沿时刻置位或复位。因此不要把set和reset信号名列入always块的时间控制表达式。**

**always @ (posedge clk)**

**begin**

**if(reset)**

**begin**

**/\*置输出为0\*/**

**end**

**else**

**if(set)**

**begin**

**/\*置输出为1\*/**

**end**

**else**

**begin**

**/\*与时钟同步的逻辑\*/**

**end**

**end**