第二章 行为建模

1. 模块
   1. 基本结构

output是wire的子类型，不允许对他用<=或=赋值，只能用assign；

如果需要用<=或=赋值，可以额外定义一个同名的寄存器

|  |
| --- |
| module MODULE\_NAME(PORT1, PORT2, PORT3, PORT4);  input [1:0] PORT1;  output PORT2;  inout PORT3;  output reg PORT4; // 同时声明output端口并定义reg  reg PORT2; // output端口声明和reg定义分开  // ....  endmodule // 注意没分号 |

或

|  |
| --- |
| module MODULE\_NAME(  input [1:0] PORT1,  output PORT2,  inout PORT3,  output reg PORT4,  );  reg PORT2;  // ...  endmodule // 注意没分号 |

* 1. 实例化（同面向对象的编程）

|  |
| --- |
| module MODULE\_NAME;  OTHER\_MODULE\_NAME inst\_OTHER\_MODULE\_NAME(  .PARAM1( WIRE1 ),  .PARAM2( WIRE2 )  );  endmodule // 注意没分号 |

1. always和initial
   1. always结构反复执行其中的语句，永远不会退出或终止

initial结构只执行一次

* 1. 单个always结构或initial结构内的语句是顺序执行的
  2. 不同always结构或initial结构的语句是并发执行的

注意：initial和always也是并发的，并不是先执行initial初始化再启动always

* 1. 当出现@、#、wait时，always和initial会挂起，直到@指定的事件发生、#的延迟达到、wait的表达式为真
  2. 规范—always语句中描述硬件行为，initial语句中描述模拟（仿真）的初始化工作

1. 赋值
   1. 持续赋值assign
   2. 非阻塞赋值<=
   3. 阻塞赋值=
2. 条件语句if – then – else
   1. 基本结构

|  |
| --- |
| if (CONDITION1) begin  // ...  end else if (CONDITION2) begin  // ...  end else begin  // …  end |

* 1. else子句为可选项，与最近的不完整if配对（同C语言）

1. 多分支语句case
   1. 基本结构

|  |
| --- |
| case (ir[15:14]) // 注意不是begin-end  2'b00 : begin // 多条执行语句用begin-end  // ...  end  2'b01, 2'b10 : // 指定多个条件，单个执行语句不需要begin-end  default : // 其他  endcase // 注意没分号 |

* 1. case & casex & casez

case枚举的条件如果出现了x（不确定）或z（高阻态），则需要完全匹配；

casez把z看作无关值；

casex把z和x都看作无关值

* 1. full case：将case指定为完备，告知综合器不要生成保持器

|  |
| --- |
| case (c\_state) //synthesis full\_case  2’b00 : n\_state <= state\_1;  2’b01 : n\_state <= state\_2;  endcase |

等价于

|  |
| --- |
| case (c\_state)  2’b00 : n\_state <= state\_1;  2’b01 : n\_state <= state\_2;  default : n\_state <= 2’bxx;  endcase |

* 1. parallel case：告知综合器去除冗余的优先级逻辑生成

否则如果case中出现多重匹配则只会选择第一个匹配的条件

|  |
| --- |
| case (CONDITION) // synthesis parallel\_case  // …  endcase |

1. 条件操作符（同C语言）

REG = CONDITION ? IF\_TRUE : IF\_FALSE;

1. 循环语句
   1. 四种循环
      1. repeat循环指定次数

|  |
| --- |
| repeat (REPEAT\_TIMES) begin  // ...  end |

* + 1. for循环（同C语言）：索引一般用integer类型

|  |
| --- |
| for (ASSIGNMENT1; CONDITION; ASSIGNMENT2) begin  // ...  end |

* + 1. while循环（同C语言）

|  |
| --- |
| while (CONDITION) begin  // ...  end |

* + 1. forever循环：无休止的循环

|  |
| --- |
| forever begin  // …  end |

* 1. 循环条件不能使用所在的always块或initial块外部的变量
  2. 循环的继续与退出

Verilog不直接支持像C语言的continue和break语句，但可以借助命名块和disable语句实现——

|  |
| --- |
| begin : break  while (CONDITION) begin : continue  // ...  disable continue; // 继续，即跳过本次循环  // ...  disable break; // 退出循环  end // continue  end // ~~break |

1. 连接运算符{}

可以将多组连线合并为一个总线进行表示，如{A,B}把A、B两组线合并在一起（A在高位，B在低位）

1. 任务和函数
   1. 基本结构
      1. 任务

|  |
| --- |
| task TASK\_NAME; // 注意有分号，而且不需要列举端口  // 端口声明  input PORT1; // 注意是分号  output PORT2; // task中不需要显式地给output定义reg就可以对他赋值  inout PORT3;  // 寄存器定义  reg A,B,C;  // 程序段  begin  // ...  end  endtask // 注意没分号 |

或

|  |
| --- |
| task TASK\_NAME(  // 端口声明  input PORT1, // 注意是逗号  output PORT2, // task中不需要显式地给output定义reg就可以对他赋值  inout PORT3 // 最后一项不能有逗号  ); // 注意有分号  // 寄存器定义  reg A,B,C;  // 程序段  begin  // ...  end  endtask // 注意没分号 |

* + 1. 函数

|  |
| --- |
| function [1:0] FUNC\_NAME; // 注意有分号，且不需要声明端口  // 端口声明  input PORT1; // 注意是分号  input PORT2; // function的端口只能是input  // 寄存器定义  reg A,B,C;  // 程序段  begin  // ...  FUNC\_NAME = 2'b00; // 返回值  end  endfunction // 注意没分号 |

或

|  |
| --- |
| function [1:0] FUNC\_NAME(  // 端口声明  input PORT1, // 注意是逗号  input PORT2, // function的端口只能是input  input PORT3 // 最后一项不能有逗号  ); // 注意有分号  // 寄存器定义  reg A,B,C;  // 程序段  begin  // ...  FUNC\_NAME = 2'b00; // 返回值  end  endfunction // 注意没分号 |

* 1. 任务与函数的比较

|  |  |  |
| --- | --- | --- |
|  | 任务 | 函数 |
| 调用 | 单独的过程语句，不能在持续赋值语句中调用 | 作为表达式的一个操作数，可以在过程和持续赋值语句中调用 |
| 输入和输出 | 通过参数输入输出，零个或多个各种类型的输入输出 | 通过参数输入，参数不能为output，至少有一个输入；通过函数名输出，不能将inout类型作为输出 |
| 定时和事件控制（#、@、wait） | 可以包含 | 不可以包含 |
| 调用其他任务或函数 | 可以调用其他任务和函数 | 可以调用其他函数，但不能调用其他任务 |

1. 标识符的作用域
   1. 超前引用和非超前引用

模块、任务、函数和命名块中的标识符都可以超前引用；

寄存器reg和线网wire变量必须先定义后引用

* 1. 层次名称：引用非本层次的标识符

|  |
| --- |
| module top;  reg r; // top.r  wire w; // top.w  always begin : y // top.y  reg q; // top.y.q  end  task t;  begin : c // top.t.c  reg q; // top.t.c.q  disable y; // 使前边的always块失效  end  endtask  endmodule // 注意没分号 |

直接引用一个名称：只能引用跟自己同一层次的，或者上一层次的，不能引用下一层次或隔壁层次的；如top.t.c可以直接引用top.y，但不能直接引用top.y.q

第三章 并发进程【仅考察概念，填空题】

1. 事件（@）
   1. 事件控制的两种基本形式
      1. 监视数值的改变（电平触发或边沿触发）
      2. 有名事件：监视事件这一抽象信号
   2. 事件控制语句

边沿触发：@( negedge clock ) q <= data;

电平触发：@( ~clock ) q <= data;

* 1. 或逻辑的事件组合

@( negedge clock or posedge rst ) q <= data;

@( negedge clock, posedge rst ) q <= data;

* 1. 上升沿和下降沿
     1. 下降沿negedge：1到0、1到x、x到0
     2. 上升沿posedge：0到1、0到x、x到1
  2. 有名事件

|  |
| --- |
| module top;  A a();  B b();  endmodule  module A;  event go; // 声明事件go  // ...  #50 -> go; // ->表示触发事件，这里事件名称为go  // ...  endmodule  module B;  // ...  @a.go // 等待实例a的go事件触发  abc <= 1'b0;  // ...  endmodule |

1. 等待语句（wait）

只对电平敏感，不对电平的变化敏感；

等待直到表达式a为True：wait(a)

1. 有名块的终止disable

disable除了终止有名块外，还将终止块中被调用的函数和任务，但不会终止其他的监听本块的事件控制

1. 内部赋值
   1. a = #25 b;

先读出b，然后延时25个单位时间，然后把前边读到的值赋给a；

（#25 a = b，先延时25个单位时间，然后读出b并立即赋给a）

* 1. q = @( posedge w ) r;

先读出r，在等到w的上升沿，然后赋值给q

* 1. w = repeat( 2 ) @( posedge clock ) t;

先读出t，然后等待两个clock的上升沿，再赋值给w

* 1. 同一个begin内的<=

|  |
| --- |
| begin  a <= #50 1;  b <= #100 2;  #100 c <= 1;  end |

a在0采样，50赋值；

b在0采样，100赋值；

c在100采样并赋值

第四章 逻辑级建模

1. 基元逻辑门
   1. 多输入单输出：and, nand, or, nor, xor, xnor

将输出作为第一个参数，其他作为输出；

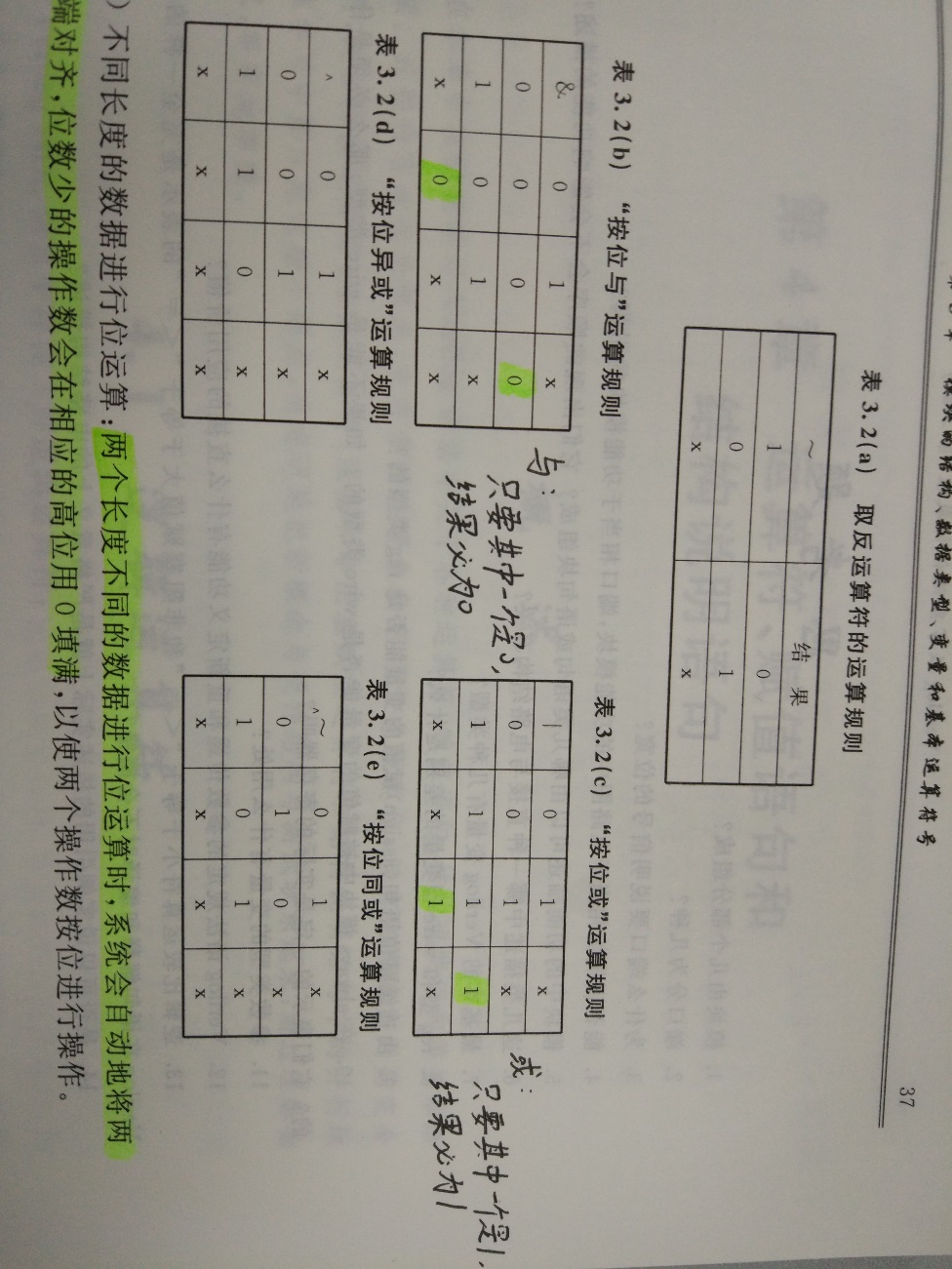
* 1. 单输入多输出：not, buf

将最后一个参数作为输入，其他作为输出

* 1. 三态单输入多输出：bufif0, bufif1, notif0, notif1

最后一个参数为使能端，倒数第二个参数为输入端

1. 逻辑值
   1. 0
   2. 1
   3. x，未知
   4. z，高阻态
   5. 逻辑运算规则



1. 线网
   1. 普通线网wire，带线与的线网wand
   2. 基本定义

wire w1; // 无延时的线网

wire #(3,5) w2; // 带延时的线网，上升沿3个单位时间，下降沿5个单位时间

1. 持续赋值

只有线网wire类型能被持续赋值；

assign sum = a\_i ^ b\_i ^ c\_i; // 注意必须是=而不是<=

wire AXorB = a ^ b; // 声明线网的时候也可以直接持续赋值

1. 延迟（上升沿延迟、下降沿延迟、关断延迟）

not #5 (ndata, data); // 匿名的非门

nand #(12, 15) qQ(q, nq, wa);

bufif1 #(3, 7, 13) qDriver(qOut, q, enable);

线网同理

指定最小延迟、典型延迟、最大延迟：

#(d1\_min : d1\_typ : d1\_max, d2\_min : d2\_typ : d2\_max, d3\_min : d3\_typ : d3\_max)

1. 时间单位定义

`timescale <time\_unit>/<time\_precision>

分别定义单位时间和时间精度

时间单位可以从fs到s

第六章 逻辑综合

1. 使用门和组合逻辑

assign out = (sel) ? A : B;

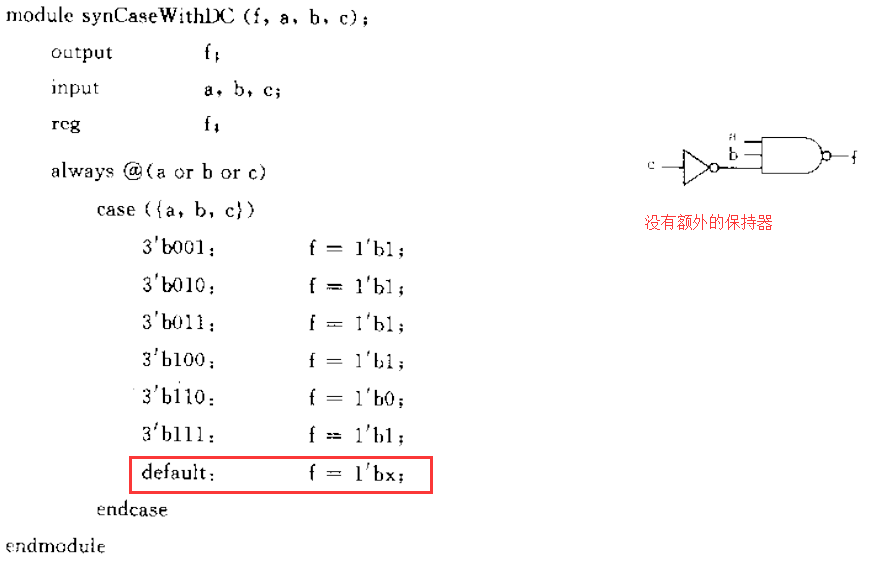
assign f = (a & b & c) | (a & ~b & ~c) | (~a & (b |c));

而不是逐一定义逻辑门和连线

1. 使用过程语句来说明组合逻辑

|  |
| --- |
| always @ (a or b or c) // 电平敏感  if ( a== 1 )  f = b; // 阻塞式赋值  else  f = c; |

1. Case语句中default直接输出x表示无关项，否则如果case中没有列举出所有情况，综合器会刻意去为未列举出的情况加一个保持器；



1. Case枚举的数值可以出现x、z、?，都表示该条件无关
2. 循环语句repeat是不可综合的
3. 时序元件的推断
   1. 锁存器（电平触发）

两种情况：显式的电平触发条件下的赋值、隐式的锁存器推断

推断always语句中的锁存器：

* + 1. 至少存在一条不对输出进行赋值的控制路径
    2. 敏感列表不允许包含任何边沿敏感的说明

如：

|  |
| --- |
| always @(g or d or reset)  if (~reset)  Q= 0; |

敏感列表不包含边沿敏感的说明，if语句中只说明了reset==0时的操作，没有说明reset==1时的操作，此时综合器会为reset保存上一次的数值（即额外布置了锁存器）

应尽量避免使用锁存器，容易产生毛刺，时序不确定，而且FPGA往往综合成一个逻辑门加一个触发器，浪费资源；

* 1. 触发器（边沿触发）

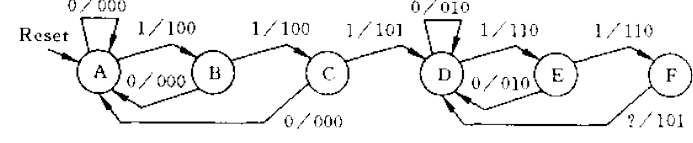
两种情况：显式的边沿触发条件下的赋值、隐式的推断

触发器推断类似锁存器，只不过是在边沿触发的always块中

1. always时序逻辑
   1. 敏感列表中每个信号都要指定边沿，只能包括时钟、复位、置位条件
   2. 第一个语句必须是if语句
   3. 必须先对置为和复位条件进行测试
   4. 赋值必须用非阻塞是赋值，尽管正规的阻塞式赋值也能正确的综合，但它无法被正确的模拟，所以应当避免
   5. 不能在同一个always块中既描述时序逻辑又描述组合逻辑
2. 三态器件的推断：使用了高阻态
3. 有限状态机（FSM）描述
   1. 应用场景
      1. 电路有时序规律或逻辑顺序
      2. 电路输出与特定状态有关
   2. 摩尔（Moorc）状态机：当前状态决定输出

米利（Mealy）状态机：当前状态和输入共同决定输出

* 1. 状态转换图

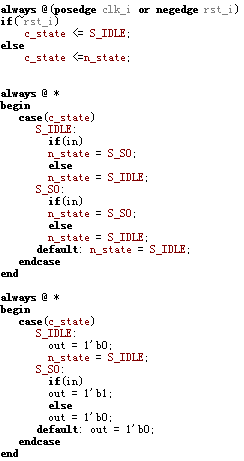


* 1. 状态编码
     1. 二进制
     2. 格雷码
     3. 独热码：每个状态只有一位为高电平

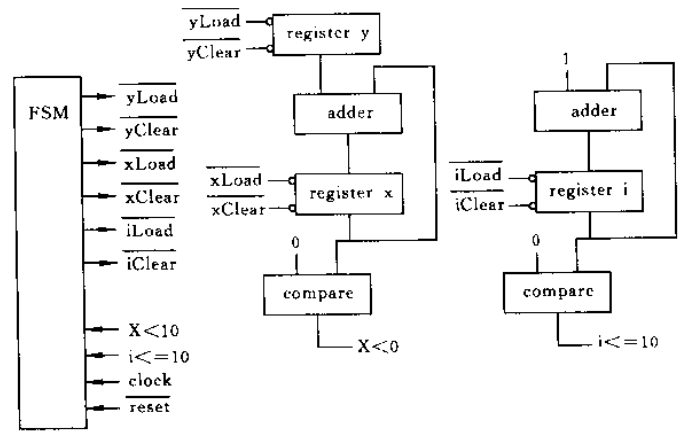
减少组合逻辑的使用量，减少毛刺，但占用比较多的触发器；

* 1. 三段式描述

状态推进、状态改变、输出控制



1. 数据通道，RTL建模



将系统分解为输入接口、数据通道、输出接口；

输入接口直接把系统输入保存到寄存器当中；

数据通道进一步按功能分解成若干个小模块，各个小模块处理某一部分或某一阶段的数据，从输入接口或上一级的模块获取输入，并传递给下一级；

输出接口则负责对外输出；

以上工作的流程一般用FSM来控制协调；

杂七杂八

1. 算数运算符和比较运算符只要操作数中出现了x或z，则运算结果为x
2. 连接操作{}中常数必须指定宽度
3. 敏感列表指定为\*表示对always块中所有条件、右值用到的量都敏感
4. 常见的不可综合语句
   1. initial
   2. 延时#
   3. while、repeat、forever
   4. 带延时的task
5. 一个变量只能在一个always里赋值
6. 模块的输入输出端口
   1. 输入端口只能是wire类型，可以由wire或reg驱动
   2. 输出端口可以是wire或reg类型，只能由wire驱动
7. 参数
   1. parameter PARAM = 8;

全局参数，上层文件也可以修改这个参数

* 1. localparam PARAM = 8;

局部参数，上层文件不能修改这个参数

1. 宏定义

`define M 5

在整个编译期间无论在哪个文件M都是有效的

1. 系统任务与函数
   1. $display：按格式控制符指定方式打印输出，每次执行都会打印
   2. $monitor：输出变化时做出响应，监测变量，变量改变时打印
   3. $time：显示当前仿真时间
   4. $stop：暂停仿真，执行run指令后可以继续仿真
   5. $finish：结束仿真
2. reg变量在声明时可以进行初始化（对应initial的0时刻）；

功能模块不要作这个初始化，应当在reset信号到来时进行初始化

1. 有符号数
   1. Reg和wire可以声明为带符号变量

reg signed [31:0] data\_reg;

* 1. 常数可以指定为有符号数：8’sh1b（八位有符号十六进制表达）
  2. 算数移位：>>>和<<<
  3. 系统函数$signed和$unsigned实现有符号数和无符号数的转换
  4. 可变向量域

vector[31-:8]等价于vector[31:24],

vector[16+:8]等价于vector[23:16]

1. 总线和数组

定义：reg [31:0] bram[255:0];

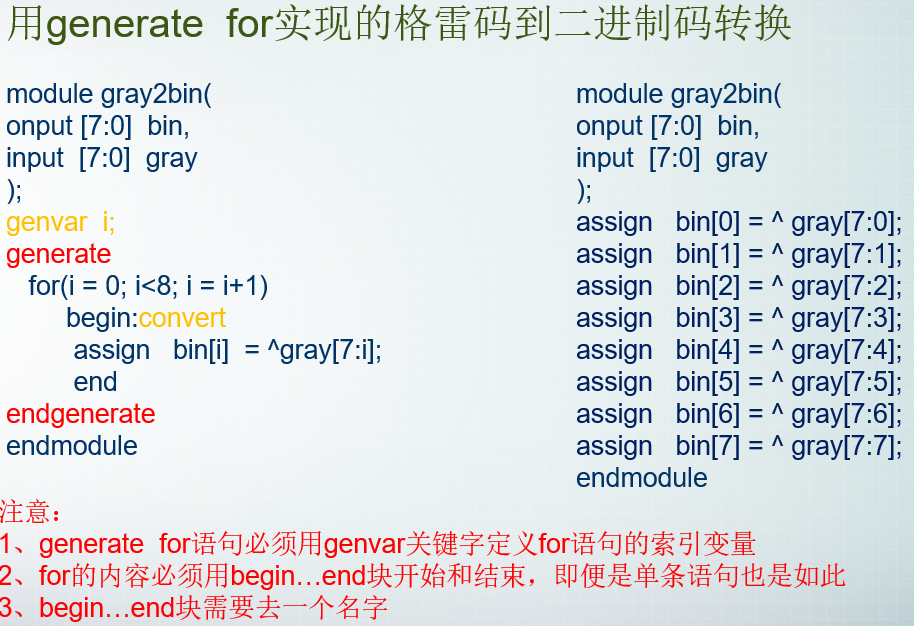
引用：（先指定数组索引再指定某一位）bram[200][30]

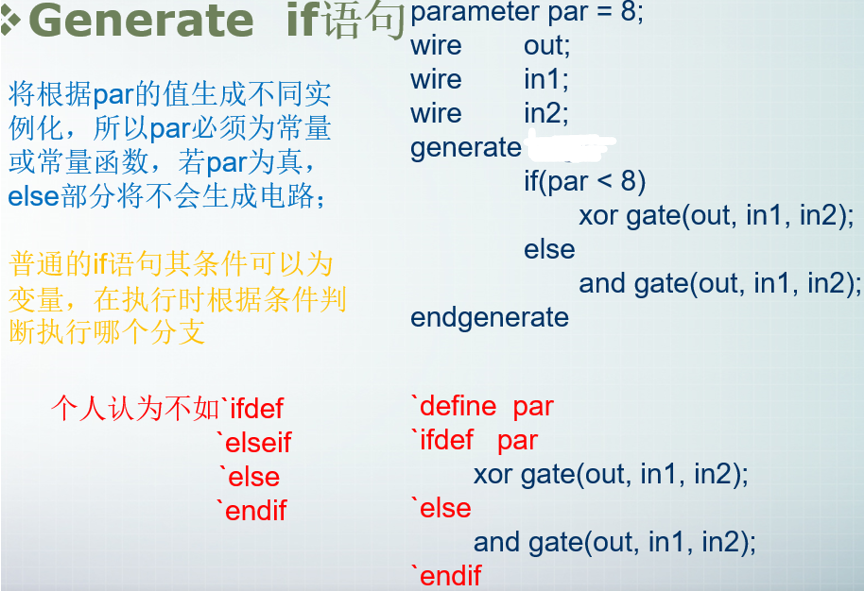
虽然verilog-2001允许多维数组，但电路不易理解，应避免

1. 生成块（verilog-2001支持）

允许for根据索引生成、if或case根据条件生成；

generate语句都只在编译时有效；





Case与if类似

1. 位宽声明的索引可以从小到大，也可以从大到小

大端模式：reg [0:7] a;

小端模式：reg [7:0] a;

1. &、|、^
   1. 双目运算符：同C语言
   2. 单目运算符：作为缩位运算符，又称归约运算符

运算过程如下，首位和第二位运算，运算结果再跟第三位，新的结果再跟第四位运算，依次类推，直到最后一位，最终得到一位运算结果

* 1. 复合运算符，&、|、^可以和~构成符合运算符~&、~|、~^表示与非、或非、同或
  2. 同或除了可以写成~^也可以写成^~
  3. 注意奇校验时计算奇偶校验位，应为parity=~(^data)而不是parity=~^data

1. 8’b1、8’bz、8’b1z、8’bz1z

8’b1是0b00000001；

8’bz是0bzzzzzzzz；

8’b1z是0b0000001z；

8’bz1z是0bzzzzzz1z；

1. 0、x、z都是逻辑假，只有1才是逻辑真
2. 示例数组构造模块实例

|  |
| --- |
| module reggac (Q, D, clock, clear);  output [7:0] Q;  input [7:0] D;  input clock, clear;  // 分别引用Q、D线，共用clear、clock线  dff r[7:0] (Q,D,clear,clock);  endmodule |

将被展开为

|  |
| --- |
| module reggac (Q, D, clock, clear);  output [7:0] Q;  input [7:0] D;  input clock, clear;  dff r[7] (Q[7],D[7],clear,clock);  dff r[6] (Q[6],D[6],clear,clock);  dff r[5] (Q[5],D[5],clear,clock);  dff r[4] (Q[4],D[4],clear,clock);  dff r[3] (Q[3],D[3],clear,clock);  dff r[2] (Q[2],D[2],clear,clock);  dff r[1] (Q[1],D[1],clear,clock);  dff r[0] (Q[0],D[0],clear,clock);  endmodule |

而

|  |
| --- |
| module reggac (Q, D, clock, clear);  output [7:0] Q;  input [7:0] D;  input clock, clear;  // 分别引用Q、D线，共用clear、clock线  dff r[3:0] (Q,D,clear,clock);  endmodule |

将被展开为

|  |
| --- |
| module reggac (Q, D, clock, clear);  output [7:0] Q;  input [7:0] D;  input clock, clear;  dff r[3] (Q[7:6],D[7:6],clear,clock);  dff r[2] (Q[5:4],D[5:4],clear,clock);  dff r[1] (Q[3:2],D[3:2],clear,clock);  dff r[0] (Q[1:0],D[1:0],clear,clock);  endmodule |