DSD复习笔记

by: 吴小茜

Chap2 Verilog 基本概念

- 模块的结构

```
module 模块名(端口列表/端口声明列表);
    端口声明；
    参数声明；
    wire, reg ... 变量声明；
    assign [#delay] LEFT=RIGHT；
    initial, always ... 行为语句；
    低层模块实例语句；
    任务和函数；
endmodule
```

- 利用层次命名引用对象
- 模块内并发执行的语句：
  - assign, 每条连续赋值语句的执行顺序依赖于发生在变量a和b上的事件，**assign不要忘记写！** 多个可以共用assign用逗号连接不过最好分开写
  - always，initial只能用于建模和仿真，**只有reg类型可以在initial/always中被赋值而wire不可以，所有initial/always语句在0时刻并发执行，顺序过程的延迟时间是累加的**
  - 结构方式：门原语，用户定义原语UDP，模块实例
- 时间：
  - #2指两个时间单位
  - `timescale 1ns/100ps，定义延时单位为1ns，精度为100ps(1ps=10^(-12)s), #2.24表示2.2ns
- 模块内input和模块外output，只能用net类型，inout在模块内外都只能用net

- [ ] 端口位宽匹配, ppt-P26，更高位补零？是否有补1的情况？
- [ ] 4位行波进位计数器，用到了T触发器，D触发器，ppt-P30

Chap3 Verilog语言基础

- 标识符：第一个字母必须是字母或者下划线，区分大小写，不能是关键字
- 基本值：x,z不区分大小写，0x1z和0X1Z相同，**wire默认=z，reg默认=x**
- 常量：整数，实数，字符串
- 整数表示法：十进制数格式，或基数格式：**自己写结果时候基数格式要写全**
  - [size]'[s/S][base][value],
  - d/D-10, b/B-2, o/O-8, h/H-16，缺省为10进制，
  - 值x/z以及十六进制中a-f不区分大小写，
  - 数值部分不能为负，4'd-2(x)
  - 8'h2A，8'和h之间不能有空格
  - 位宽不能使用表达式‘=
  - 补最高位：4'bX0=4'bxxx0, 4'hZ=4'bzzzz

    - 整数中？可以替代z
    - 下划线，不能作为首字符，不能出现在位宽和进制处，8'b_0101_1100(x)
    - 默认位宽和机器有关，10, 'bx
- 实数
    - 6.(x), 小数点两侧必须有一位数字
    - 科学计数法
- 字符串
    - 双引号
    - 不能多行书写
    - 每个字母用8位ASCII值表示'=
- 参数parameter表示常量，可以重载,表达式右边必须是常数表达式，只能是数字和已经定义过的参数，用于定义延迟时间和变量宽度

```
comb #(.N(8)) m_comb(...);
```

- 局部参数localparam，值不能重载修改，状态机的状态编码
- tri，线网类型，多个驱动源

```
tri y;
assign y=a&b;
assign y=a^b;
// a&b=01x, a^b=11z,->y=x1x
//0-0=0,1-1=1,0-z=0,1-z=1,z-z=z,ow=x
```

- reg
    - 变量的值被解释为无符号数，负数以二进制补码保存

```
reg [3:0] tag;
tag=-2; //保存的是4'd14=4'b1110, 0010->1101->1110
```

    - 声明为有符号类型变量

```
reg [4:0] x;
reg [4:0] y;
initial begin
 x=5;
 $display("x=%5b",x); //x=00101
 y=-x;
 $display("y=%5b",y) //y=11011, 补码，00101->11010->11011
 $display("y=%d",y) //y=27, 默认解释为无符号整数
end

reg [4:0] x;
reg signed [4:0] y; //signed 的位置在wire/reg 和位宽之间
```

```
initial begin
 x=5;
 $display("x=%5b",x); //x=00101
 y=-x;
 $display("y=%5b",y) //y=11011, 补码，00101->11010->11011
 $display("y=%d",y) //y=-5
end
```

- 部分位选择，高低顺序一致

```
reg [255:0] data1;
reg [0:255] data2;
reg [7:0] byte;
byte = data1[31-:8]; //31:24
byte = data1[24+:8]; //31:24
byte = data2[31-:8]; //24:31
byte = data2[24+:8]; //24:31
```

- 寄存器类型：reg,integer,real,time,realtime
- 存储器类型memory，寄存器数组，可以用索引进行部分位选择

```
parameter WSIZE=64, MSIZE=64;
reg [WSIZE-1:0] mem[MSIZE-1:0];
reg [7:0] mem[0:63]; //64x8为存储器
mem = 0; //x
mem[1] = 0;
```

- 数组，声明各种类型的数组：reg,integer,time,real,realtime,wire，区分位宽和数组维度，不能对整个数组赋值，不能对数组一整行赋值
- 表达式：操作数和操作符
- 算术运算符，+-*/%
  - / 整数除法，截断小数部分
  - % 取模，两个操作数均为整数，*求出与第一个操作符号相同的余数，两个绝对值做运算*
  - **如果算术运算符中的任意操作数是x或z，则整个结果为x**

```
5'b0_10x1+5'b0_1111=5'bx_xxxx;
5'b1_z010+5'b0_0010=5'bx_xxxx;
```

  - **独立表达式中，运算结果的长度由最长的操作数决定；赋值语句中，由左边目标的长度决定**

```
wire[4:1] a,b;
wire[5:1] c;
wire[6:1] d;
wire[8:1] F;
```

```
if((a+c)+(b+d)) //(a+c)结果为6位，因为看整个表达式
F=(a+c)+(b+d); //中间结果(a+c)结果为8位
```

  - 无符号数和有符号数
    - 无符号数：wire,reg；有符号数：integer, reg signed, wire signed
    - 表达式有符号/无符号只依赖于操作数，不依赖于左边被赋值变量
    - 十进制数是有符号数
    - 带符号基数格式，4'sd12
    - 位选择和部分位选择是无符号数

```
reg [5:0] bar;
bar = -4'd12; // 001100->110011->110100->52

integer tab;
tab = -4'd12; // 32bit，同样存储补码
```

- 位运算符
  - **~ 按位取反**
  - & 按位与
  - | 按位或
  - **^ 按位异或**
  - **^~ 按位同或**
  - **若操作数长度不同，长度较小的操作数在最左边添0**
  - **位运算规则**
    - ~x=x, ~z=x
    - & 两个操作数中有一个是0，结果为0(包括出现x/z)，1&1=1，其余都为x
    - | 两个操作数中有一个是1，结果为1(包括出现x/z)，0|0=0，其余都为x
    - ^, ^~, 两个操作数中出现x/z，则结果为x
- 逻辑运算符
  - **用于条件判断，总是相当于1位二进制位运算**
  - **&&，||，！**
  - **对于向量的逻辑运算，非0向量作为1处理**

```
reg flag;
reg [3:0] val_a, val_b;
reg [3:0] a,b,c,d,e,f;

inital begin
  val_a = 4'b0110; //比如在initial/always语句中赋值
  val_b = 4'b0000;
  c = !val_a; //c=0000
  d = !val_b; //d=0001, 高位填充0
  flag=1'bx;
  e=!flag; //e=000x
  f=a+e; //f=xxxx
end
```

- 关系运算符
  - **如果操作数中有一位为x或z，那么结果为x**
- 相等关系运算符
  - ==, !=, ===, !==
  - **如果操作数中有一位为x或z，那么结果为x**
  - **如果操作数长度不等，长度较小的操作数左边添0补位**

```
data='b11x0;
addr='b11x0;
data==addr //x
data===addr //1
```

- 移位运算符
  - 如果右侧操作数的值为x/z，移位操作结果为x
  - 用0填补移出的空位
- 拼接/连接运算符
  - 可以用作左值

```
assign bus0={bus[3:0],bus0[7:4]};
{bus0,5} //x, 不允许使用非定长数
```

- 重复复制

```
abus = {3{4'b1011}};
```

- 缩减/归约运算符
  - &, ~&, |, ~|, ^, ~^, 这里带~的运算符得到相反结果
  - 如果某一位为x/z，则结果为x，可以用^确定向量中是否有位不确定

```
`timescale 1ns/1ns
module logic_eval();
  reg [3:0] a;
  reg [3:0] data;
  initial begin
    a = 4'b0;         //d=4'b0000
    #5 a = 4'b1;      //d=4'b0001
    #5 a = 4'bz0x1;   //d=4'b0001
    #5 a = 4'b1z00;   //d=4'b0001
    #5 a = 4'b0zx0;   //d=4'b000x
    #5 a = 4'b0xx0;   //d=4'b000x
    #5 a = 4'bx;      //d=4'b000x
    #5 a = 4'bz;      //d=4'b000x
  end
```

```
  always @(*) data = !(!a);
  //逻辑运算符！先对向量做按位或|缩减?
  initial
  $monitor("At %2t time, a=%4b, data=%4b", $time, a, data);
endmodule
```

- 条件运算符

```
data_out = (a) ? 4'b110x : 4'b1000;
//如果 a 为真，则 data_out = 4’b110x
//如果 a 为假，则 data_out = 4’b1000
//如果 a 为 x，则 data_out = 4’b1x0x
//如果 a 为 z，则 data_out = 4’b1x0x, 求同存异
```

- 运算优先级

Chap4 门级建模

- 当任意一个输入端口的值发生变化是，立即重新计算输出端口值
- 原语：and, nand, or, nor, xor异或, xnor同或
- 引用门原语实例时可以没有实例名
- 输入端口超过两个时，先计算两个，得到的结果在和第三个输入值进行运算
- **真值表**
- 缓冲器buf，非门not，多个输出，所有输出端的值相同；一个输入，端口列表的最后一个, x/z->x
- 三态门：带控制端的缓冲器/非门，bufif1,bufif0,notif1,notif0,**若控制信号无效则输出z，真值表**
- 门实例数组

```
wire [7:0] OUT, IN1, IN2;// 门实例数组引用
nand n_gate[7:0](OUT, IN1, IN2);// 与下面 8 条实例引用语句相同
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
...
```

- 门延迟
  - 上升延迟: 0/x/z->1, 下降延迟: 1/x/z->0, 关断延迟: 0/1/x->z
  - 最大、最小和典型延迟

```
// 3种延时（上升、下降和关断）都等于 delay_time 表示的延时时间
and #(delay_time) a1(out, i1, i2);
// 说明了上升延时和下降延时时间，关断延时时间 = min(rise_val, fall_val)
and #(rise_val, fall_val) a2(out, i1, i2);
// 说明了上升延时、下降延时和关断延时时间
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

```
// One delay
// 如果选择使用 mindelays, delay= 4
// 如果选择使用 typdelays, delay= 5
// 如果选择使用 maxdelays, delay= 6
and #(4:5:6) a1(out, i1, i2);

// Two delays
// 如果选择使用 mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// 如果选择使用 typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// 如果选择使用 maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);

// Three delays
// 如果选择使用 mindelays, rise= 2 fall= 3 turn-off = 4
// 如果选择使用 typdelays, rise= 3 fall= 4 turn-off = 5
// 如果选择使用 maxdelays, rise= 4 fall= 5 turn-off = 6
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);

and #4 a1(e,a,b), u1(out,e,c);//两个门的延迟都是4
```

```
//判断仿真结果
module gate_delay(out,a,b,c);
  output out;
  input a,b,c;
  wire e;

  and #5 a1(e,a,b);
  or #4 u1(out,e,c);
endmodule

`include "gate_delay.v"
module tb_gate_delay;
  reg A,B,C;
  wire OUT;

  gate_delay m1(OUT,A,B,C);

  initial begin
  A=1'b0;B=1'b0;C=1'b0;
  #10 A=1'b1;B=1'b1;C=1'b1;
  #10 A=1'b1;B=1'b0;C=1'b0;
  #20 $finish;
  end
endmodule
```

```
//四位全加器
// Instantiate four 1-bit full adders.
fulladd fa0(c1, sum[0], a[0], b[0], cin);
```

```
fulladd fa1(c2, sum[1], a[1], b[1], c1 );
fulladd fa2(c3, sum[2], a[2], b[2], c2 );
fulladd fa3(cout, sum[3], a[3], b[3], c3 );

// Instantiate four 1-bit full adders. 注意顺序，从最高位到最低位对应
fulladd fa0[0:3] ( {cout,c3,c2,c1}, sum, a, b, {c3,c2,c1,cin});
```

Chap5 数据流建模

- 连续赋值语句，**要写关键字assign**，只要右边表达式中操作数上有事件发生，表达式立即计算，新结果赋值给左边的线网

```
assign mux = ( s == 0 ) ? A : 'bz, // 逗号(,)结尾
      mux = ( s == 1 ) ? B : 'bz, // 逗号(,)结尾
      mux = ( s == 2 ) ? C : 'bz, // 逗号(,)结尾
      mux = ( s == 3 ) ? D : 'bz; // 最后是分号(;)

// 连续赋值，out 必须是线网类型变量，
// i1 和 i2 可以是 wire 类型， 也可以是 reg 类型
assign out = i1 & i2

//标量型线网向量
wire scalared [63:0] bus; // 可以位选择/部分位选择
//向量型线网向量
wire vectored [31:0] data; // 不许以位选择/部分位选择

wire cout, cin;
wire [3 : 0] sum, a, b;
assign {cout, sum} = a + b + cin;

//在线网声明的同时对其赋值，隐式连续赋值
// 普通的连续赋值
wire out;
assign out = in1 & in2;
// 隐式连续赋值，实现与上面两条语句相同的功能
wire out = in1 & in2;
wire clear = 1'b1;

//隐式线网声明
// 连续赋值, out 是一个线网变量.
wire i1, i2;
assign out = i1 & i2; // 虽然没有声明 out 为线网类型
```

- 延时

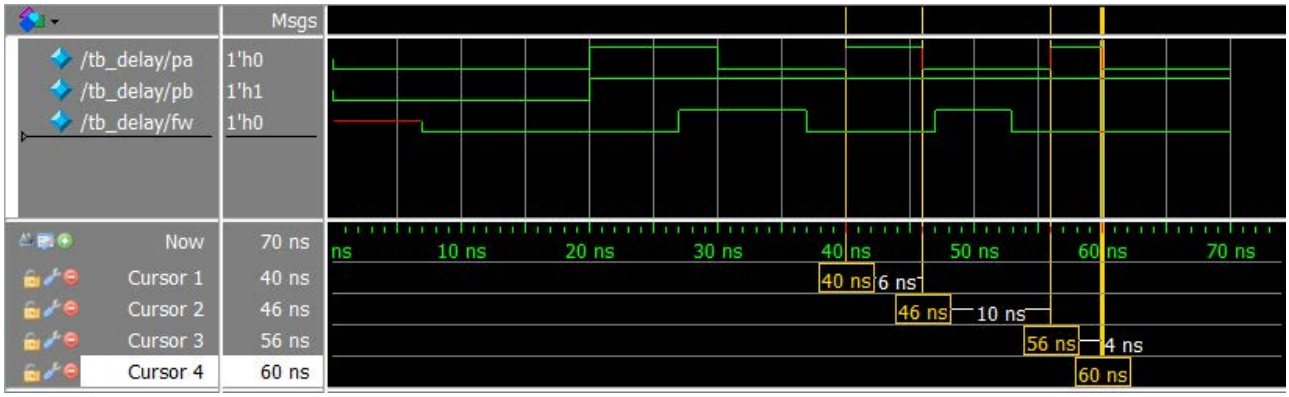
```
//普通赋值延时
assign #10 out = in1 & in2;
//惯性延时——小于延时的脉冲将被取消，脉冲宽度小于赋值的延时的输入变化不会对输出产生影响
```

```
//信号保持的持续时间必须大于延时宽度

// 隐式连续赋值延时
wire #10 out = in1 & in2;
// 等效于
wire out;
assign #10 out = in1 & in2;

wire #5 rd; //驱动源的值改变与线网 rd 本身之间的延时
assign #2 rd = a & b; //延时为7

//既有线网延时又有赋值延时，信号保持的持续时间必须大于二个延时的最大值
// 线网延时
wire #10 out;
assign out = in1 & in2;
// 等效语句
wire out;
assign #10 out = in1 & in2;
```



- 延迟

```
assign #(rise, fall, turn-off) target = expression; //?
```

```
//mux4_to_1
// 产生输出的逻辑方程
assign out = (~s1 & ~s0 & i0)|
(~s1 & s0 & i1)|
( s1 & ~s0 & i2)|
( s1 & s0 & i3) ;
// 使用条件操作符
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0);

//三态门
assign y1=(ctrl)?in:1'bz; //bufif1

//3-8译码器
module decoder3x8 ( output [7:0] fout, input [2:0] din );
  assign fout = (din == 3'b000 ) ? 8'b0000_0001 :
                (din == 3'b001 ) ? 8'b0000_0010 :
```

```
                (din == 3'b010 ) ? 8'b0000_0100 :
                (din == 3'b011 ) ? 8'b0000_1000 :
                (din == 3'b100 ) ? 8'b0001_0000 :
                (din == 3'b101 ) ? 8'b0010_0000 :
                (din == 3'b110 ) ? 8'b0100_0000 :
                (din == 3'b111 ) ? 8'b1000_0000 : 8'h00;
endmodule
```

Chap6 行为建模

- initial 和 always 语句不能嵌套使用
- 多个 initial 或 always 语句并发执行，执行顺序与其在模块的顺序无关
- 仿真时，所有的 initial 和 always 语句都在 0 时刻开始执行
- fork ... join 将多条语句组合成一个并行语句块
- 顺序语句块中可以包含并行语句块，并行语句块中可以包含顺序语句块
- initial 和 always 过程语句只能对寄存器类型变量进行赋值

```
module ex01;

// 定义时钟变量
reg clock;

// 设置时钟变量的初值为 0
initial clock = 0;

// 在时钟变量声明时，将其初始化
// 只适用模块一级的变量声明
reg clock = 0;

always @(*) begin: BLOCK1
reg temp; // 不可初始化变量 temp
temp = a;
end
endmodule
```

```
//使用 always 语句设计一个时钟发生器
module clock_gen (output reg clock);
// 在 0 时刻，初始化变量 clock
initial
clock = 1'b0;
// 每半个周期翻转一次 clock, (time period = 20)
always
#10 clock = ~clock;
initial
#1000 $finish;
endmodule
```

- initial 语句块用于测试模块的描述 —— 不可用于设计电路, always建议尽量避免用于测试模块中，如：testbench， top模块中
- 时序控制
  - 延时控制
    - 常规延时控制：遇到该语句与开始执行该语句的时间间隔
    - 延时不必是常量，可以是表达式
    - 如果延时表达式的值为 x 或 z，与 0 延时等效
    - 如果延时表达式的值为负时，其二进制补码作为延时

```
module delay_neg;
reg clock;
reg [2:0] delay;
initial
clock = 1'b0;

initial
delay=-3;

initial
#(delay) clock = ~clock; //3'b101,5

endmodule
```

    - 内嵌赋值延时：遇到该语句后，首先立即计算右侧表达式的值，推迟指定的时间之后，再将这个值赋给左边的变量
    - 带零延时控制的语句在执行时刻相同的多条语句中最后执行
  - 事件控制
    - 边沿触发事件控制

```
@ (posedge clock)
  state0 = state1;

posedge （正沿） —— 正沿是下面变化的一种
  □0 → x
  □0 → z
  □0 → 1
  □x → 1
  □z → 1

negedge（负沿） —— 负沿是下面变化的一种
  □1 → x
  □1 → z
  □1 → 0
  □x → 0
  □z → 0

@(posedge clock) q = d; // 当 clock 正边沿到来时执行
@(negedge clock) q = d; // 当 clock 负边沿到来时执行
```

```
q = @(posedge clock) d; // 立即计算 d 的值， // 当 clock 正边沿到来时赋给 q
```

- 电平敏感事件控制

```
@(clock) q = d; // 当 clock 的值发生变化时执行
```

```
@ flag y = a;
或写成：@flag;
y = a; // 当 flag 上发生事件，就执行赋值语句，如：原来是低电平（0），现在变成高电平（1），反之亦然
```

  - 敏感变量列表, 由 or 连接的多个信号变量, or 只是事件控制表达式中说明有不同事件关键字，并非逻辑或
  - 在敏感列表中用逗号 , 代替 or
  - 使用 @* 和 @(*), 表示对其后语句块中所有输入变量的变化都是敏感的
- 敏感列表事件控制, **wait等待电平敏感条件为真**

```
always
  wait (count_enable) #20 count = count + 1
□执行过程：
□如果 count_enable 为 0，不执行后面的语句 ——— 只能为真
□仿真程序会停到下来，直到其为真，如：1 （高电平）
```

- 命名事件控制

```
// 定义一个事件：
received_dataevent received_data;
// 在 clock 正边沿到来时刻进行检测
always @(posedge clock) begin
  if(last_data_packet) // 如果是最后一个数据帧
    -> received_data; // 触发事件 received_data
  end
always @(received_data) // 等待事件 received_data 发生
 // 当事件发生时，存储数据
data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

- 如果有标识符, 在语句块中可以声明寄存器变量,语句块可以用其标识符引用，如：使用禁止语句停止语句块的执行

```
initial
begin : break_block
```

```
i = 0;
forever begin
if (i==a) disable break_block;
#1 i = i + 1;
end
end
```

- 顺序语句块:**语句的延时总是相对于前面的语句的完成时间**, 语句块可以有自己的名字 —— 称之为命名块
- 并行语句块:**各语句的延时均以语句块的开始时间为基准**, 各语句中的延时和事件控制决定语句的执行顺序

```
begin
pm_enable = 1'b0;
#1 pm_enable = 1'b1;
end
initial
begin : seq_a
#4 pm_write = 6'd5;
fork : par_a // 语句块的开始时间?
#6 pm_select = 4'd7;
begin : seq_b
wdog_rst = pm_enable;
#2 wdog_intr = wdog_rst;
end
# 2 frc_sel = 4'd3;
# 4 pm_iter = 4'd2;
# 8 itop = 4'd4;
join
# 8 pm_lock = 1'b1; //20
# 2 pcell_id = 6'd52; //22
# 6 $stop; //28
end // 语句块的结束时间? 28
```

- 在过程语句结构的顺序语句块中，阻塞赋值语句按顺序执行,下一条语句必须等待上面的阻塞赋值完成后才能执行
- 非阻塞赋值语句执行后计算右侧表达式的值，并不立即赋值,继续执行下一条语句,在当前时间步的其它计算完成后，再对左边目标赋值,同一时间步,执行下一条语句时，无需上一条赋值完成

```
module non_block;
reg [3:0] state;
initial begin
state=4'b0000; //0
#2 state=4'b1111; //2
state<=#5 4'b1010; //7
#3state=4'b0101; //5
end
endmodule

begin
a <= 4'd1;
```

```
a <= #10 4'd0;
a <= #5 4'd4;
end

fork
a = 4'd1;
a = #10 4'd0;
a = #5 4'd4;
join

`timescale 1ns/1ns
module nonblocking_d;
reg [3:0] a;
initial
begin
a <= 4'd1;
#10 a <= 4'd0; //10
#5 a <= 4'd4; //15
end
initial
$monitor("At time t=%4t, a=%4d", $time, a);
endmodule


`timescale 1ns/1ns
module blocking_d;
reg [3:0] a;
initial
fork
a = 4'd1;
#10 a = 4'd0; //10
#5 a = 4'd4; //5
join
initial
$monitor("At time t=%4t, a=%4d", $time, a);
endmodule
```

- 条件语句
  - 逻辑表达式必须使用括号括起来
  - 如果逻辑表达式的值为 0，x，z，则按“假”处理

```
if ( 条件1)
□ 过程语句 1;
□if ( 条件1)
□ 过程语句 1;
□else
□ 过程语句 2;
□if ( 条件1)
□ 过程语句 1;
□else if (条件2)
□ 过程语句 2;
```

```
else if (条件3)
  过程语句 3;
```

- 多路分支语句,case
  - 在 case 语句中，x 和 z 作为值 x 和 z 解释
  - 在 casez 语句中，表达式中的 z 是无关位 —— 忽略，不进行比较
  - 在 casex 语句中，表达式中的 x 和 z 都是无关位

```
reg a;
casez (a)
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase //输入z，不做比较，直接执行语句1

reg [4:0] a;
case ( 3'b101 << 2)
  3'b100 : a = 3'd1;
  4'b1000 : a = 4'd4;
  5'b1_0100 : a = 5'd16;
  default: a = 5'dx;
endcase
//先扩展位宽，再进行计算，然后进行比较
```

- 禁止语句 disable
  - disable 任务标识符；
  - disable 语句块标识符；
  - 禁止语句是过程性语句，只能出现在 always 和 initial 的语句块中
- 循环语句:forever repeat while for

```
forever
  procedual_statement

repeat (循环计数表达式)
  过程语句；
// 如果计数表达式的值不确定，即为 x 和 z，则循环次数按 0 处理

while (条件表达式)
  过程语句；
// 如果条件表达式的值为 x 和 z，则按 0（假） 处理

initial begin
  for (index=0; index < 10; index = index + 2)
  mem[index] = index;
end
```

- **if, case, forever, repeat, for, while, 在initial/always语句块中**
- 生成块
  - 不允许出现的声明 □参数、局部参数 □输入、输出和输入/输出声明
  - 三种生成语句：循环生成，条件生成，case 生成

```
//循环生成
module bitwise_xor (out, i0, i1);
parameter N = 32; // 32-bit bus by default
output [N-1:0] out;
input [N-1:0] i0, i1;

// 声明用于循环生成块的循环控制变量，只用于生成快
genvar j;
// 生成按位异或
generate for (j=0; j<N; j=j+1)
begin : xor_loop
  xor g1 (out[j], i0[j], i1[j]);
end
endgenerate // 生成块结束

// 另一种设计方式，使用 always 过程语句块
// reg [N-1:0] out;
// generate for (j=0; j<N; j=j+1) begin: bit
// Always @(*) out[j] = i0[j] ^ i1[j];
// end
// endgenerate

endmodule
```

- **时序逻辑电路，非阻塞赋值；组合逻辑电路，阻塞赋值**

```
//二进制编码转换成格雷码
module bin2gray1 #(parameter N=8)
( output reg [N-1:0] gray_val, input [N-1:0] bin_val );
always @(*) gray_val = { bin_val[N-1], bin_val[N-1:1]^bin_val[N-2:0]};
endmodule

module bin2gray2 #(parameter N=8) ( output [N-1:0] gray_val,
input [N-1:0] bin_val );
genvar k;
generate for (k=0; k<N; k=k+1 ) begin:loop
assign gray_val[k] = ( k==N-1) ? bin_val[k] : bin_val[k]^bin_val[k+1];
end
endgenerate
endmodule

module bin2gray3 #(parameter N=8) ( output reg [N-1:0] gray_val,
input [N-1:0] bin_val );
genvar k;
generate for (k=0; k<N; k=k+1 ) begin:loop
```

```
always @(*) gray_val[k] = ( k==N-1) ? bin_val[k] : bin_val[k]^bin_val[k+1];
end
endgenerate
endmodule
```

```
//连用
and #4 a1(e,a,b), u1(out,e,c);//两个门的延迟都是4
output reg f,g //f,g都是output reg
```

Chap7 任务和函数

- 函数，**能调用另外一个函数，但不能调用另外一个任务；总是在仿真时刻0就开始执行；不能包含任何延时、事件控制或时序控制声明语句；至少有一个输入变量；不能有输出（output）和双向（inout）变量；函数只能返回一个值**
- 任务和函数必须在模块内进行定义，其作用范围仅局限于定义它们的模块
- 任务,输出参数的值直到任务退出时，才传给调用参数,**任务调用语句是过程语句，在 always 和 initial 中使用,任务调用中的输出/双向(输入输出)参数必须是寄存器类型（任务外）**, 与定义中的输入/输出参数顺序相匹配
- 如果一个在模块中的两个地方被同时调用，则两次调用任务将对同一块地址空间进行操作;通过使用关键字 automatic 声明自动任务,每次调用都动态分配存储空间，每个任务调用都对各自独立的地址空间进行操作

```
task 任务名标识符; // 要有一个任务标识符
□ parameter_declaration;
□ input_declaration;
□ output_declaration;
□ inout_declaration;
□ register_declaration;
□ event_declaration;
□ statement;
endtask

module top;
reg [15:0] cd_xor, ef_xor;
reg [15:0] c, d, e, f;
...
task automatic bitwise_xor;
output [15:0] ab_xor;
input [15:0] a, b;
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
...
always @(posedge clk) bitwise_xor(ef_xor, e, f);
...
```

```
  always @(posedge clk2) bitwise_xor(cd_xor, c, d);
  ...
  endmodule

  // 使用 ANSI C 风格的变量声明进行任务定义
  task bitwise_op(output [15:0] ab_and, ab_or, ab_xor, input [15:0] a, b);
  begin
  #delay ab_and = a & b;
  ab_or = a | b;
  ab_xor = a ^ b;
  ende
  ndtask
```

- 函数：不能包含时序控制语句，将函数作为表达式中的操作数使用，至少要有一个 input 变量，不能有 output 或 inout 变量
- 函数中局部变量是静态分配的, 每次调用都对对同一块地址空间进行操作
- 通过使用关键字 automatic 声明自动函数,每次调用都动态分配属于这次调用的存储空间

```
function [15:0] negation;
  input [15:0] a;
  negation = ~a;
endfunction

function real multiply;
  input a, b;
  real a, b;
  multiply = ((1.2 * a) * (b * 0.17)) * 5.1;
endfunction

// 定义计算奇偶校验函数
function calc_parity;
input [31:0] address;
// 使用隐含的内部寄存器 calc_parity.
calc_parity = ^address; // 返回所有位的异或值（异或规约运算）
endfunction

// 采用 ANSI C 风格定义计算奇偶校验函数
function calc_parity (input [31:0] address);
// 使用隐含的内部寄存器 calc_parity.
calc_parity = ^address; // 返回所有位的异或值（异或规约运算）
endfunction

module decoderlogN #(parameter N=8) (
output reg [N-1:0] y,
input [clog2(N)-1:0] a ); // 2^n = N --> log2(N)==n
function integer clog2(input integer n);
begin
clog2 = 0;
n--;
while ( n>0 ) begin
clog2 = clog2 + 1;
```

```
n = n>>1;
end
end
endfunction
always @*
y = 1'b1 << a;
endmodule
```

- 文件输出任务

```
integer 文件指针 = $fopen( file_name );
$fclose (文件指针);
$fdisplay(文件指针，... );
$write(文件指针，... );
$monitor(文件指针，... );
```

- 文件输入任务

```
$readmemb(“数据文件名”, memory_name, [起始地址, [结束地址]]
$readmemh(“数据文件名”, memory_name, [起始地址, [结束地址]]
```

- $random [(seed)],返回一个32位有符号随机数
- `timescale <时间单位> / <时间精度>
    - 时间单位定义延迟的时间单位
    - 时间精度定义仿真时间的精确程度
    - 时间精度不能大于时间单位
    - 编译指令后面不加分号（；）
    - $printtimescale [(层次路径名)] ;
- 宏定义, 用一个指定的宏名（标识符）代表一个字符串（宏内容）, 引用宏名时必须在宏名之前加上符号“`”

```
module macro_test;
 reg a, b, c;
 wire y_out;
`define ab a & b
`define abc `ab & c
 assign y_out = `abc;
endmodule
```

```
`define WORDSIZE 16
module m_name;
 reg [ `WORDSIZE – 1 : 0 ] data;
 ...
endmodule
```

- 条件编译命令

```
□`ifdef 宏名
□ 程序段 1
□[`else
□ 程序段 2 ]
□`endif
```

- `include “文件名”

Chap8 用户定义原语UDP

- UDP不能综合，只可用于仿真
- 只有一个1 bit 输出端，端口列表中的第一个,如果定义的是表示时序逻辑的原语，输出端口必须声明为 reg 类型
- 不支持 inout 端口
- 状态表中可包含的值为 0、1和 x，不能有 z
- UDP与模块同级
- 无关项可用符号“？”表示，？—— 自动展开为 0、1或 x

```
primitive udp_and(out, a, b);
output out; // 组合逻辑的输出端不能声明成 reg 类型
input a, b; // 输入端口声明
// 定义状态表
table
// a b : out;
0 0 : 0;
0 1 : 0;
1 0 : 0;
1 1 : 1;
endtable
endprimitive

xor (s1, a, b); // 使用 Verilog 内置原语
udp_and (c1, a, b); // 使用 UDP
```

- 带清零端的电平敏感锁存器,若 clear 为 1，输出 q 恒为 0,若 clear 为 0 □如果 clock 为 1， q = d □如果 clock 为 0，保持 q

```
primitive latch(q, d, clock, clear);
output q;
reg q; // 声明 q 为 reg 类型保存内部数据

input d, clock, clear;// 初始化时序 UDP，只允许有一条 initial 语句
initial q = 0; // 初始化输出为 0
```

```
//state table
table
// 当前状态 下一状态
// d clock clear : q : q+
? ? 1 : ? : 0 ; // 清零
1 1 0 : ? : 1 ; // 将 d 的值锁存在 q = 1
0 1 0 : ? : 0 ; // 将 d 的值锁存在 q = 0
? 0 0 : ? : - ; // - 表示 q 保持原状态不变
endtable
endprimitive
```

- 带清零端的下降沿触发的D触发器,若 clear = 1，则 q 的输出恒为 0,若 clear = 0 □当 clock 从 1 跳变到 0 时，则 q = d，否则，q 保持不变 □当 clock 保持稳定时，而 d 改变值，q 不变

Chap10 组合逻辑设计

Chap11 时序电路设计

- 锁存器（latch）,电平敏感存储器

```
//一定是非阻塞赋值
module SRLatch(output reg q, qbar,
input s, r );
always @(*) begin
case ({s,r})
2'b01: {q, qbar} <= 2'b01;
2'b10: {q, qbar} <= 2'b10;
2'b11: {q, qbar} <= 2'bx;
default ; //有分号，00时候保持不变
endcase
end
endmodule

// D Latch
module DLatch(output reg Q, output QN, input D, input C );
always @(*)
if (C) Q <= D; //非阻塞赋值
assign QN = ~Q;
endmodule
```

- 触发器,由时钟跳变沿触发的存储器，在控制时钟上升沿到来的时刻，采样D输入信号，并据此改变Q和QN的输出

```
module dff (output reg q, output qn,input d, input clk);
always @(posedge clk)
  q <= d; //非阻塞赋值
assign qn = ~q;
endmodule
```

```
module dff_ne( output reg q, output qn,
input d, input clk);
always @(negedge clk) q <= d;
assign qn = ~q;
endmodule

module dff_en(output reg q, output qn, input d, input en, input clk);
always @(posedge clk)
if ( en ) q <= d;
assign qn = ~q;
endmodule

module JKFF( output reg q, qn,
input clk, j, k);
always @ (posedge clk)
case ({j,k})
2'b01: {q, qn} <= 2'b01;
2'b10: {q, qn} <= 2'b10;
2'b11: {q, qn} <= {qn, q};
default: ;
endcase
endmodule

module JK_FF ( output reg Q, output Q_b, input Clk, J, K );
assign Q_b = ~ Q ;
always @( posedge Clk)
case ({J,K})
2'b00: Q <= Q;
2'b01: Q <= 1'b0;
2'b10: Q <= 1'b1;
2'b11: Q <= ~Q;
endcase
endmodule

module TFF(output reg q, qn, input clk, rst_n, t);
always @(posedge clk) begin
if (~rst_n)
{q, qn} <= 2'b01;
else
if (t)
{q, qn} <= {qn, q};
end
endmodule
```

- 同步置位和复位，只有在时钟的有效跳变沿的时刻，置位和复位信号脉冲才能使触发器置位和复位

```
always @(posedge CLK )
□ if (SET)
□ Q <= 1’b1;
□ else Q <= D;
```

```
//复位优先于置位,置位优先于输入
module DFF_sr( output reg q, qn, input d, clk, reset, set );
always @(posedge clk)
if (reset) {q, qn} <= 2'b01;
else if (set) {q, qn} <= 2'b10;
else begin {q, qn} <= {d, ~d}; end
endmodule
```

- 异步——操作信号不受时钟脉冲（边沿）控制, 当置位与复位脉冲到来时，立即将触发器的输出端置 1 或 0, 将置位和复位信号列入 always 语句的事件控制列表中

```
always @(posedge CLK or posedge SET)
 if (SET)
 Q <= 1’b1;
 else Q <= D;

module DFF_asr( output reg q, qn,input d, clk, reset, set );
always @(posedge clk, posedge reset, posedge set )
if (reset) {q, qn} <= 2'b01;
else if (set) {q, qn} <= 2'b10;
else {q, qn} <= {d, ~d};
endmodule
```

- 移位寄存器（shift register）

```
//串行输入/串行输出
module shiftreg4b( output reg dout, input clk, reset, din);
reg [3:0] r;
always @( posedge clk or posedge reset )
if (reset) r <= 4'b0000;
else
begin r <= {r[2:0], din};
dout <= r[3];
end
endmodule

//串行输入/并行输出
module shiftreg4b( output reg [3:0] dout, input clk, reset, din);
always @(posedge clk or posedge reset)
if (reset) dout <= 4'h0;
else dout <= {dout[2:0], din};
endmodule

//左移、右移、加载
module UShiftReg #(parameter N=8) ( output reg [N-1:0] q,input [N-1:0] d, input
[1:0] s,input Lin, Rin,input clk, rst_n );always @ (posedge clk)
if (~rst_n) q <= 0;
else
case (s)
```

```
2'b11:q <= d;
2'b10:q <= {q[N-2:0], Lin};
2'b01:q <= {Rin, q[N-1:1]};
default: ;
endcase
endmodule

module filter(output reg y, input clk, rst_n, din);
reg [3:0] q;
always @(posedge clk)
begin
if (!rst_n)
  begin q <= 4'b0; y <= 1'b0; end
else
begin
if ( &q[3:1]) y <= 1'b1;
else if (~|q[3:1]) y <= 1'b0;

q <= {q[2:0], din};
end
end
endmodule
```

- 线性反馈移位寄存器（LFSR）,**电路图**

```
module LFSR( output reg [0:7] q, // 8 bit data output.
input clk, // Clock input.
input rst_n, // Synchronous reset input.
input load, // Synchronous load input.
input [0:7] din // 8 bit parallel data input.
);
always @( posedge clk ) begin
if ( ~rst_n )
q <= 8'b0;
else begin
if (load)
q <= (|din) ? din : 8'b0000_0001;
else begin
if ( q == 8'b0 )
q <= 8'b0000_0001;
else begin
q[7] <= q[6];
q[6] <= q[5] ^ q[7];
q[5] <= q[4] ^ q[7];
{q[4], q[3], q[2]} <= {q[3], q[2], q[1]};
q[1] <= q[0] ^ q[7];
q[0] <= q[7];
end
end
end
end
endmodule
```

- 具有异步复位的 4 位计数器

```
module counter #(parameter N=4) ( output reg [N-1:0] count, input clk, rst_n);
always @(posedge clk or posedge rst_n) //negedge rst_n?
if ( ~rst_n )count <= 0;
else count <= count + 1;
endmodule

//可逆
module decimal_counter #(parameter N = 4)  //parameter · 格式
( output reg [N-1:0] count, output reg sup, inf, input clk, rst_n, load, dir,
input [N-1 : 0] data );
always @ (posedge clk)
begin
if (!rst_n)
begin
count <= 4'd0; {inf, sup} <= 2'b0;
end
else if (load) count <= data;
else
if (!dir)
if ( count < 4'd9)
begin count <= count + 4'd1; {inf, sup} <= 2'b0; end //{inf, sup} <= 2'b0; 两个都
赋值
else begin count <= 4'd0; {inf, sup} <= 2'b01; end
else if ( count > 4'd0) begin count <= count - 4'd1; {inf, sup} <= 2'b0; end
else begin count <= 4'd9; {inf, sup} <= 2'b10;
end
end
endmodule

//对应的仿真
initial begin
p_clk = 0;
forever #5 p_clk = ~p_clk;
end

initial begin
p_rst = 0;
#20 p_rst = 1'b1;
end

initial begin
p_load = 0;
p_dir = 0;
p_data = 4'bx;
#120 p_data = 4'd9;
#18 p_load = 1'b1;
#18 p_load = 1'b0;
#5 p_data = 4'dz;
```

```
#12 p_dir = 1'b1;
end
```

- 存储器：跳过了一些内容

```
module ROM(output [7:0] data, input [3:0] address, input en);
assign data = (en) ? ROM_LOC(address) : 8'bz;
function [7:0] ROM_LOC( input [3:0] a );
case (a)
4'h0: ROM_LOC = 8'b1010_1001;
4'h1: ROM_LOC = 8'b1111_1101;
4'h2: ROM_LOC = 8'b1110_1001;
4'h3: ROM_LOC = 8'b1101_1100;
4'h4: ROM_LOC = 8'b1011_1001;
4'h5: ROM_LOC = 8'b1100_0010;
4'h6: ROM_LOC = 8'b1100_0101;
4'h7: ROM_LOC = 8'b0000_0100;
4'h8: ROM_LOC = 8'b1110_1100;
4'h9: ROM_LOC = 8'b1000_1010;
4'hA: ROM_LOC = 8'b1100_1111;
4'hB: ROM_LOC = 8'b0011_0100;
4'hC: ROM_LOC = 8'b1100_0001;
4'hD: ROM_LOC = 8'b1001_1111;
4'hE: ROM_LOC = 8'b1010_0101;
4'hF: ROM_LOC = 8'b0101_1100;
default:ROM_LOC = 8'bx;
endcase
endfunction
endmodule

module ROM ( output reg [7:0] data, // Address input
             input [3:0] address, // Data output
             input en, // Read Enable
             input ce // Chip Enable
             );
always @(*) begin
if ( en && ce )
  case (address)
    0 : data = 8'ha;
    1 : data = 8'h37;
    2 : data = 8'hf4;
    3 : data = 8'h0;
    4 : data = 8'h9;
    5 : data = 8'hff;
    6 : data = 8'h11;
    7 : data = 8'h1;
    8 : data = 8'h10;
    9 : data = 8'h15;
    10 : data = 8'h1d;
    11 : data = 8'h25;
    12 : data = 8'h60;
    13 : data = 8'h90;
```

```
    14 : data = 8'h70;
    15 : data = 8'h90;
    default: data = 8'hz;
  endcase
else data = 8'hz;
end
endmodule
```

- 堆栈（LIFO）
- 有限状态机（FSM，finite state machine）,设计时采用同步复位
  - Mealy 机:输出同时取决于状态和输入
  - Moore机:输出只由状态决定

```
module mealy( output reg yout, input clk, rst, xin );
  parameter A=2'b01, B=2'b10;
  reg [1:0] state;
  always @(posedge clk) begin
  if (!rst) begin state <= A; yout <= 1'b0; end
  else
  case(state)
    A: if (xin) begin yout <= 1'b1; state <= B; end
    else begin yout <= 1'b0; state <= A; end
    B: if (xin) begin yout <= 1'b0; state <= A; end
    else begin yout <= 1'b1; state <= B; end
    default:begin yout <= 1'b0; state <= A; end
  endcase
  end
endmodule
```

```
//JK触发器的Moore 电路表示
module Moore_JK( output reg q, qn, input clk, rst, j, k );
parameter A=2'b01, B=2'b10;
reg [1:0] state;
always @(posedge clk) begin
if (!rst) begin {q,qn} <= 2'b01; state <= A; end
else begin
case(state)
A: begin
{q,qn} <= 2'b01;
state <= (j==1'b1) ? B : A;
end
B: begin
{q,qn} <= 2'b10;
state <= (k==1'b1) ? A : B;
end
default: ;
endcase
end
```

```
end
endmodule
```

Chap12 使用Verilog_HDL进行综合

```
//仿真模块

initial begin
p_s = 0;
forever begin
#8 p_s = 1'b1; #2 p_s = 1'b0;
end
end
```