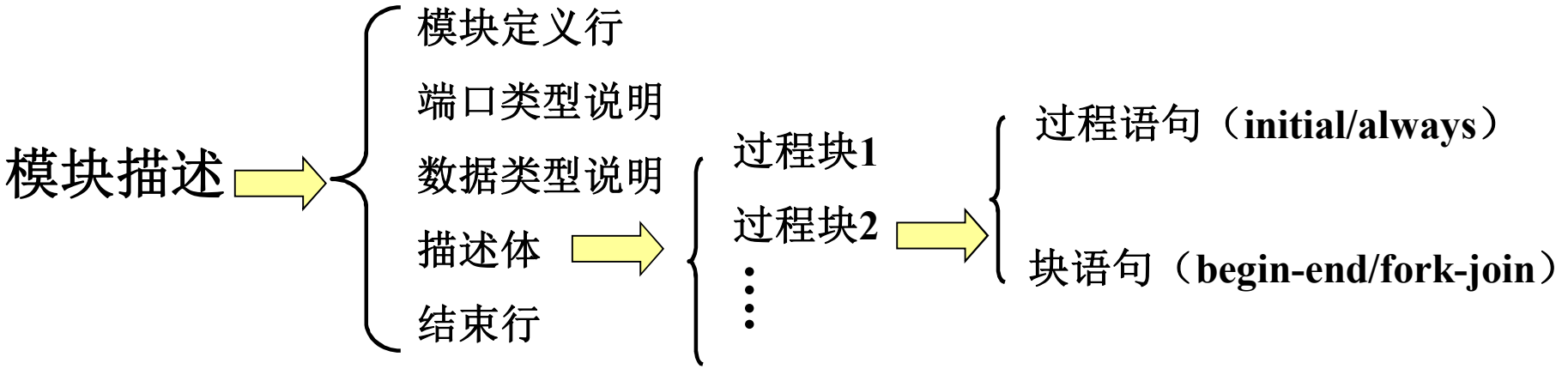
目 录

内 容

- 第一节 硬件描述语言
- 第二节 Verilog HDL设计入门
- 第三节 Verilog基础知识
- 第四节 行为描述
- 第五节 数据流描述
- 第六节 结构描述
- 第七节 用户自定义元件
- 第八节 其他论题
- 第九节 仿真
- 第十节 实例分析

Module的基本结构（复习）

Verilog HDL描述中模块的构成框架：



Verilog HDL对模块的行为描述以过程块为基本组成单位，一个模块的行为描述由一个或多个并行运行的过程块组成。

第四节 行为描述

第四节 行为描述

initial语句

initial语句的表示形式：

initial

时序控制 进程语句;

- initial语句在仿真的0时刻开始执行,其各个进程语句只执行一次。
- 顺序过程(begin.....end)最常出现在进程语句中。
- 时序控制可以是时延控制，或事件控制。
- 如果进程语句中出现时间控制，initial语句在以后的某个时间完成执行 。

always语句

always语句的表示形式：

always

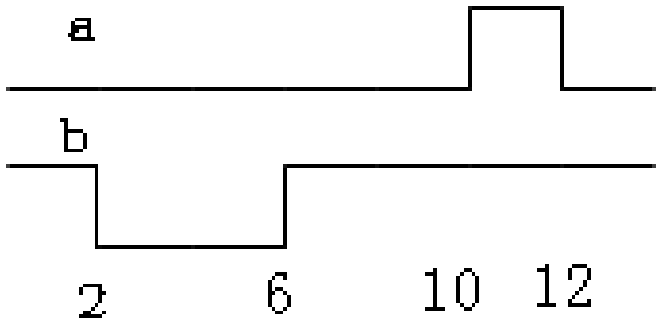
时序控制 进程语句;

- always语句不同于initial语句，它的各个进程语句按顺序循环地执行。
- 时序控制可以是时延控制，或事件控制。
- 一个模块的行为描述中，可以有多个initial和always语句，它们之间相互独立，并行执行。

第四节 行为描述

顺序块begin-end

```
begin
    a=0；  b=1；
    #2 b=0；
    #4 b=1；
    #4 a=1；
    #2 a=0；
end
```



并行块fork-join

fork

a=0； b=1；

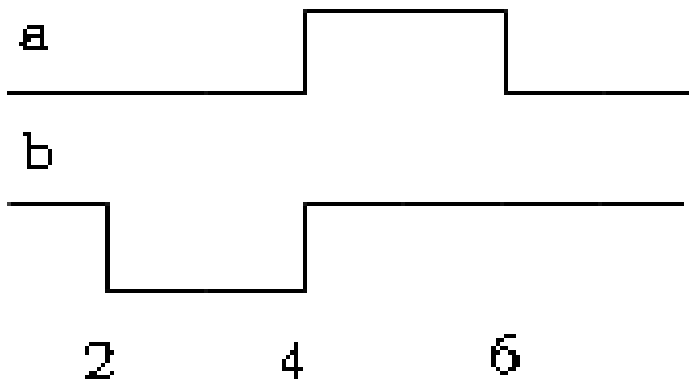
#2 b=0；

#4 b=1；

#4 a=1；

#6 a=0；

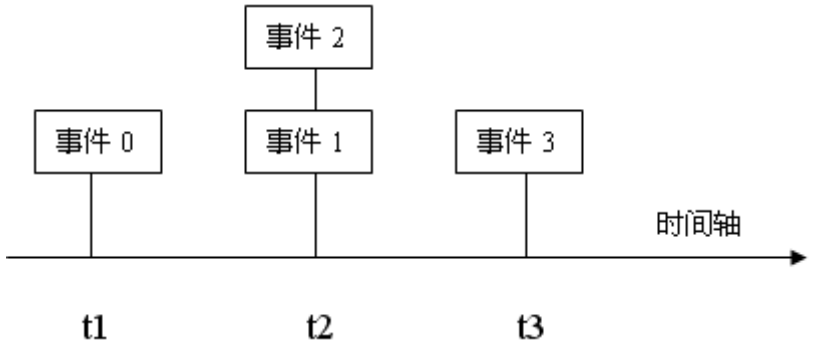
join



第四节 行为描述

时间（Time）和事件（Event）

与其他计算机高级语言一样，Verilog也采用文本形式作为人机界面，但是HDL源码（source code）与计算机程序（program）在执行的方式上却有本质的区别。计算机程序以语句出现的先后次序为依据执行任务，它只能按时间串行，称为时间驱动（time-driven）。然而在硬件电路中，所有的电路元器件都是同时并行工作的，因此在HDL源码中，除了时间轴以外还有事件轴，在任何一个时刻都可以有多个事件同时发生，称为事件驱动（event-driven）。



时延控制和事件控制

Verilog提供以下三种时延控制和事件控制的方式：

- #（表达式）

描述延时（delay）功能，“表达式”是一个时间长度的计算式。当表达式规定的延时结束时，某个进程（process）开始执行，是典型的时间控制。

- @（事件）

典型的事件控制。当规定的事件发生时，进程启动。一个事件被执行过一次以后是否在将来的某个时刻仍然有效，需要看其他条件，例如，当@(事件)与always联用时，该事件将永远有效，相反，在initial中，事件只有效一次。

- wait（表达式）

通常用于几个功能块之间的协调。当另一个进程使wait后面的表达式成立时，wait所在的进程启动。

举 例

```
initial #70 $stop;
initial
  begin
        #10 cell = 6;
        #20 cell = 6;
        #30 cell = 6;
  end
initial
  begin
        # 5   cell = 9;
        # 20 cell = 9;
        # 30 cell = 9;
  end
always @ cell
  begin
        $display ("cell = %d at time %d", cell, $time);
  end
```

执行的结果是：

```
cell = 9 at time 5
cell = 6 at time 10
cell = 9 at time 25
cell = 6 at time 30
cell = 9 at time 55
cell = 6 at time 60
```

事件变量

事件变量是变量的一种，在变量说明中定义。Verilog使用“-> 事件”的格式触发一个事件变量。

```
event flag;
    begin
        count = count + 1;
        -> flag;
    end
```

每当计数器count计一次数，事件变量flag被触发一次。

事件控制

- 事件控制形式如下：

@ 事件 进程语句;

- 具体地说，它可分为以下四种：

@ (信号名)

例：@ (rst) clock = 2；

说明：仅当信号rst发生变化时(上升沿或下降沿)，clock被赋值为2。

***@ (posedge** 信号名)*

例：@ (**posedge** rst) clock = 2；

说明：仅当信号rst出现上升沿时，clock被赋值为2。

@ (negedge 信号名)

例：@ (**negedge** rst) clock = 2；

说明：仅当信号rst出现下降沿时，clock被赋值为2。

@ (事件1 or 事件2 or ……)

例：@ (**posedge** rst or **negedge** load) clock =2；

说明：当信号rst出现上升沿，或信号load出现下降沿时，clock被赋值为2 。

第四节 行为描述

过程赋值语句

```
always
@ (A or B or C or D)
begin: AOI
  reg  Temp1 ，Temp2;
  Temp1 = A & B;
  Temp2 = C & D;
  Temp1 = Temp1 | T e m p 2 ;
  Z = ~T e m p 1;
end
```

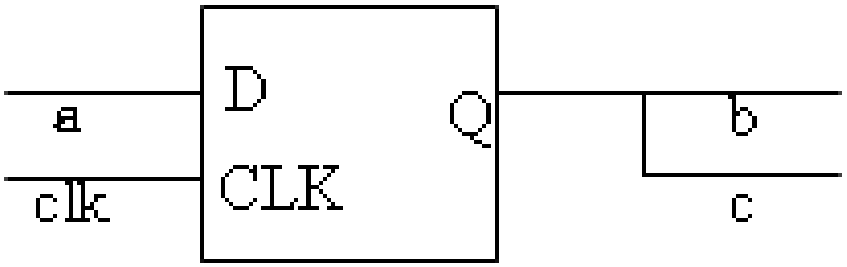
- 过程赋值语句是在always语句或者initial语句中对寄存器数据类型的变量赋值。
- 从例中还可以看出，对过程赋值语句而言，在always或者initial模块内部，每条语句和其周围的语句是按照顺序依次执行的。

阻塞性过程赋值

阻塞型过程赋值的形式的简单实例如：b=a;其特点是：

- 赋值语句执行完后，块才结束；
- b的值在赋值语句执行完后就立即发生改变(或者立即经过所定义的延时后发生改变);
- 常用来给复杂组合逻辑电路建模，当用来给时序逻辑电路建模时常带来意料之外的结果。

```
always @(posedge clk)
        begin
          b=a;
          c=b;
        end
```



- 例中使用阻塞型过程赋值，在上升clk信号的上升沿到来的时候，信号赋值开始作用：先将a的值赋给b，之后马上又将b的值赋给c，其效果就是c的值也和a的值相等。

- 很显然，只使用一个触发器，用两个寄存器来保存a的值，使得c寄存器的使用失去了它的价值，显然不是设计者的初衷。所以我们努力避免这种阻塞型过程赋值方式。

阻塞过程赋值语句

阻塞赋值语句有以下三种格式：

1. 变量名 = 表达式

表达式的任何变化都会立即反应到赋值号 “=”（等号）的左面；

2. 变量名 = #延时 表达式

表达式变化要等到规定的延时结束后才会向赋值号左面赋值；

3. 变量名 = @事件 表达式

只有当触发条件中的事件发生时表达式的新内容才向赋值号左面赋值。

所有以上三种格式的赋值语句都必须按语句排列的先后顺序执行，因此称为阻塞赋值语句。换句话说，每一条语句的执行都受到前面语句的阻挡。

非阻塞性过程赋值

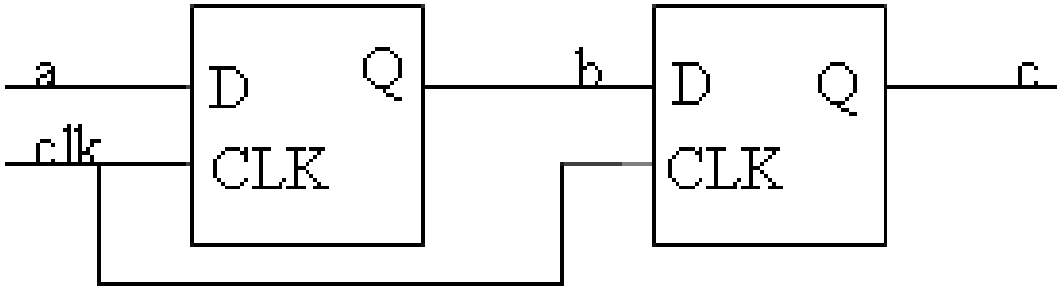
- 赋值形式：以符号“<=”进行赋值，如b<=a。
- 其特点是：

➢在块结束后才完成值操作；

➢b的值并不是立刻就改变的；

➢常用于时序逻辑电路的建模。

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



- 在always块结束之后，clk的上升沿到来之时，将b的值赋给c，将a的值赋给b。其特点是很明显的，相当于使用了两个触发器，既保持了b的值，又赋予b一个新值。
- 可以明显的看出使用非阻塞性过程赋值能够有效地克服阻塞性过程赋值所带来的缺点。因此这种赋值方法广泛应用于时序逻辑电路的建模。

非阻塞过程赋值语句

非阻塞赋值语句与阻塞赋值语句的结构十分相象，不同的只是用赋值号 "<=" 替代语句中的 "=" 。其语句结构也有以下三种格式：

变量名 <= 表达式

变量名 <= #延时 表达式

变量名 <= @事件 表达式

在非阻塞赋值中，各条语句的赋值号右面的表达式会被逐条执行，但不会向左面赋值。只有当一个描述块中全部非阻塞赋值语句的赋值号右面都执行完毕，所有语句的赋值过程才按各自的条件同时开始执行。

非阻塞过程赋值语句（Cont'd）

在下面的移位寄存器描述中，两条阻塞赋值语句的顺序显然不可以颠倒，否则执行的结果会完全不一样。

```
step1 = step2;
step2 = step3;
```

而非阻塞赋值会给电路描述带来很大的方便。同样是上面的移位寄存器用非阻塞赋值描述就不在意顺序的先后，因为仿真器首先把所有右面的状态全部记录下来，然后才向左面赋值，因此不会出现语句顺序问题。下面的两种描述方式是完全等效的：

```
step1 <= step2;          step2 <= step3;
step2 <= step3;          step1 <= step2;
```

阻塞和非阻塞赋值语句

在可综合设计中，在使用阻塞和非阻塞赋值语句方面，应注意以下原则：

- 原则1：时序电路建模时，用非阻塞赋值；
- 原则2：锁存器电路建模时，用非阻塞赋值；
- 原则3：用always块描述组合逻辑时，采用阻塞赋值；
- 原则4：在同一个always块中同时建立时序和组合逻辑电路时，用非阻塞赋值；
- 原则5：在同一个always块中不要同时使用非阻塞赋值和阻塞赋值；
- 原则6：不要在多个always块中为同一个变量赋值；
- 原则7：用$strobe系统任务来显示用非阻塞赋值的变量值；
- 原则8：在赋值时不要使用＃0延时。

遵循以上原则，有助于正确编写可综合电路，并且可以消除90％以上在仿真时可能产生的竞争冒险现象。

连续赋值语句

- assign连续赋值语句专门针对连线进行赋值，考虑了连线驱动强度与延时时间后，完整的连续赋值语句形式为：

  assign *（strength0，sthength1）#（delay）* **赋值表达式**

  如：assign c=a&b;

- assign连续赋值语句是把一个由基本逻辑门构成的组合逻辑功能电路用布尔表达式的形式予以表示。当赋值语句右端表达式中的量发生变化时，将随时反映到赋值语句左端的连线上。

示 例

```
module ander (out, ina, inb);
  input [7:0] ina, inb;
  output [7:0] out;
  wire [7:0] out;
  assign out = ina & inb;
endmodule
```

连续赋值与过程赋值

- 赋值对象不同
  连续赋值语句用于对线型变量赋值；过程赋值语句完成对寄存器变量赋值。
- 赋值过程实现方式不同
  线型变量一旦被连续赋值语句赋值后，赋值语句右端表达式中的信号有任何变化，都将随时反映到左端的线型变量中；过程赋值语句只有在语句执行到时，赋值过程才进行一次，且赋值过程的具体执行时刻还受到时延控制和事件控制等多方面的影响。
- 语句出现的位置不同
  连续赋值语句不能出现在任何一个过程块中；过程赋值语句只能出现在过程块中。
- 语句结构不同
  连续赋值语句以关键词assign为先导，语句中的赋值算符只有阻塞型一种形式；过程赋值语句不需要相应的先导关键词，语句中的赋值算符分阻塞型和非阻塞型两种。
- 冲突处理方式不同
  一条连线可被多条连续赋值语句同时驱动，最后的结果依据连线类型的不同有相应的冲突处理方式；寄存器变量在同一时刻只允许一条过程赋值语句对其进行赋值。

过程连续赋值语句

- 定义：

  过程连续赋值语句就是将连续赋值语句应用到过程块中对寄存器变量进行赋值。有两组过程连续赋值语句，即assign/deassign与force/release。

- 作用：

  用来实现异步功能，具有强制的、更高的优先级。

- 注意：

  (1)过程性连续赋值本质上属于过程性赋值，它不能够在**always**语句或**initial**语句外出现。

  (2)当用一条assign语句对寄存器变量赋值后，原有对该寄存器变量进行过程赋值的语句不再起作用，直到执行一条deassign语句将强制过程释放，正常的过程赋值语句才重新起作用。因此，出现过程连续赋值语句之处，总会存在一条deassign释放语句；

  (3)assign过程连续赋值语句不允许对寄存器变量的某一位或其中的某几位进行赋值。

assign/deassign示例

例：用过程连续赋值语句实现具有异步清零功能（低电平有效）的上升沿D触发器

```
module dff_asyn(q,d,clear,clk);
   output q;
   input d, clear, clk;
   reg q;
//asynchronous clear procedure block
   always @(clear)
         if(!clear)
            assign q=0;
         else
            deassign q;
//normal D flip-flop procedure block
   always @(posedge clk)
         q=d;
endmodule
```

过程连续赋值语句（续）

force/release的定义和作用同assign/deassign一样，差别在于：

- assign/deassign的赋值对象只能是寄存器变量；force/release的赋值对象既可以是寄存器变量也可以是线型变量；
- 对寄存器变量而言， force/release赋值语句比assign/deassign有更高优先级；对线型变量而言，它同样具有最高优先级，可以覆盖其它所有对这条线的驱动。

Note：

force/release语句的作用是为设计者在模拟过程中进行快速调试提供一种介入手段，并不是为对硬件系统的描述而设置的。

force/release示例

```
wire Prt;
. . .
or #1 (Prt ， Std， Dzx);
initial
begin
force Prt=Dzx&Std;
#5;
release Prt;
end
```

第四节 行为描述

4.1 过程结构
- 4.1.1 initial语句
- 4.1.2 always语句

4.2 语句块
- 4.2.1 顺序语句块(begin……end)
- 4.2.2 并行语句块(fork……join)

4.3 时序控制
- 4.3.1 时延控制
- 4.3.2 事件控制

4.4 赋值语句
- 4.4.1 过程赋值语句
- 4.4.2 阻塞型过程赋值语句和非阻塞性过程赋值语句
- 4.4.3 连续赋值与过程赋值的比较
- 4.4.4 过程性连续赋值

4.5 控制语句

控制语句

- 控制语句（control flow）
  - -- 条件语句（if-else，if-else-if）
  - -- case语句（case，casex，casez）
  - -- 循环语句（for，while，repeat，forever ）

条件语句

条件语句（if-else）：

与一般高级语言相同，Verilog中的if-else语句提供一个条件转移的机会。当if后面的表达式是一个非0的确定数值时，第一条语句将被执行；否则，当表达式是0、x或z时，else后面的语句将被执行。

if (*condition*_1)
procedural_statement_1
{else if (*condition*_2)
*procedural_statement*_2**}**
……
{else
*procedural_statement*_n**}**

- 如果*condition_1*值为1，那么*procedural_statement_1*被执行；如果*condition_1*值为0、**x**或**z**，那么*procedural_statement_1*不执行。

条件语句（Cont'd）

由于else是选项，因此if和else并不总是成对地出现，于是就有else与那一个if配合的问题。Verilog规定，else与它前面最近的一个if是一对。如果需要，可以使用begin-end标志成对的if和else。

例：

```
if (index > 0)
        if (rega > regb)
                result = rega;
        else
                result = regb;
```

else与第二个if是一对；

```
if (index > 0)
        begin
        if (rega > regb)
                result = rega;
        end
        else
                result = regb;
```

else与第一个if是一对

条件语句（Cont’d）

条件语句（if-else-if）：

if-else-if结构通常用于多于两个选择的情况。在一串if-else-if中，if语句将顺序被评估是否有效，第一个有效if后面的语句将被执行，并终止整个if-else-if串，跳到后面的语句。最后一个else用于处理上面所讨论的if条件一个也不满足的情况，用于“软着陆”。通常这里应该给一个缺省（default）出路。如果设计工程师十分确定不需要给出任何缺省，最后的else也可以不要。

建模应用示例（1）

- 多路选择器

```
always @(enable or data_a or
  data_b)
if (enable)
out=data_a;
else
out=data_b;
```

- 推断锁存器

```
always @(enable or data)
if (enable)
out=data;
```

建模应用示例（2）

• 具有同步清零功能的D触发器

```
module dff_sync(q,d,clear,clk);
output q;
input d,clear,clk;
reg q;
always @(posedge clk)
if(!clear)
q=0;
else
q=d;
endmodule
```

具有异步清零功能的D触发器

```
module dff_asyn2(q,d,clear,clk);
output q;
input d,clear,clk;
reg q;
always @(clear or posedge clk)
if(!clear)
q=0;
else
q=d;
endmodule
```

case语句

case (*case_expr*)

case_item_expr{ *,case_item_expr*} *:procedural_statement*

. . .

. . .

[default:*procedural_statement*]

- casez、casex语句

```
parameter
MON=0,TUE=1,WED=2,
THU=3,FRI=4,
SAT = 5， SUN=6;
reg [0:2] Day;
integer Pocket_Money;

case (Day)
TUE : Pocket_Money=6; //分支1
MON,
WED : Pocket_Money=2; //分支2
FRI,
SAT,
SUN:Pocket_Money=7; //分支3
default:Pocket_Money=0; //分支4
endcase
```

- **case**语句首先对条件表达式*case_expr*求值，然后依次对各分支项求值并进行比较，第一个与条件表达式值相匹配的分支中的语句被执行。可以在一个分支中定义多个分支项；这些值不需要互斥。缺省分支覆盖所有没有被分支表达式覆盖的其他分支。

case语句（Cont'd）

case与if-else-if有以下两点重要差别：

(1)if-else-if语句中的条件表达式的结构简单、明了，通用性较强，但在同一个层次上只有“是”与“否”两种选择，因此较为粗糙。case语句则在一个层次上提供多个选择，因此条件表达式需要一系列的比较才能完成其功能，相对来说较为复杂，但也精细很多；

(2)case条件表达式中的x和z与数值0和1一样参与比较，不影响给出一个明确结果，而if-else-if条件表达式中的x或z只能笼统地给出false的结果。

casez和casex

- casez和casex与case的功能基本相同，不同的是：casez把选择条件中的z处理为任意态（don‘t care），而casex把z和x都处理为任意态。
- case（casex，casez）中的条件表达式也可以是一个常数。

case建模应用

- 与if语句相比，当多路选择的控制条件集中在某个变量的变化上时，用case语句表达显得更为方便。
- case语句适宜于对CPU的译码等部件的描述以及对有限状态机的描述。

case语句示例

例：使用case语句实现解码器。

```
case (control)
        2'd0:  decode = 4'b1000;
        2'd1:  decode = 4'b0100;
        2'd2:  decode = 4'b0010;
        2'd3:  decode = 4'b0001;
        default decode = 'bx;
endcase
```

举 例

```
casex (control)
  8'b001100xx:  decode = 4'b1000;
  8'b1100xx00:  decode = 4'b0100;
  8'b00xx0011:  decode = 4'b0010;
  8'bxx010100:  decode = 4'b0001;
  default decode = 'bx;
endcase
```

当control = 8'b11001100时，第二条语句将被执行。

```
case (1)
  control[3]：decode = 4'b1000;
  control[2]：decode = 4'b0100;
  control[1]：decode = 4'b0010;
  control[0]：decode = 4'b0001;
  default：decode = 'bx;
endcase
```

条件表达式是常数1，每个序号表达式是控制变量control[3:0]中的一位，第一个等于1的序号将使它所对应的语句被执行。

循环语句

Verilog支持for，while，repeat和forever四种循环语句：

- forever

  持续无条件执行一条语句。

- repeat

  执行一条语句n次，其中n是一个确定的常数。如果n的值未知（x）或是高阻（z），n将被认为是0，语句将不会被执行。

- while

  持续执行一条语句直到某个设定的条件不再满足为止。

- for

  执行一条（或几条）语句，同时按照一定的条件操作一个计数器，当计数器的值降到0时，循环停止。

forever循环

语法：

forever

procedural_statement

注：不可综合

例：

initial

begin

Clock=0;

#5 **forever**

#10 *Clock* =~*Cloc*k;

end

注：但一般习惯于用always来描述时钟。

举 例

```
initial forever
  begin @ (a or b or c)
        if (a & b = = c)
        begin
        $dispaly ("a(%d) & b(%d) = c (%d)", a, b, c);
        $stop;
        end
  end
```

用forever语句永远监视仿真过程中的一个现象：（a & b = = c），并把它及时报告出来，在查错（debug）工作中十分有用。

repeat循环

- 语法：

repeat (*loop_coun*t)

procedural_statement

注：可综合

例：

repeat *(Coun*t)

@ ***(posedge*** *Clk) Sum=Sum*+10;

举 例

```
repeat (n)
    begin
        if (control[2])
        shift_register1 = shift_register1 << 1;
        shift_register2 = shift_register2 >> 1;
    end
```

在 control[2] 的控制下 n 次操作移位寄存器 shift_register1 和 shift_register2，其中 n 是正整数，可以用任何合法的方式赋值。

while循环

语法：

while (*condition*)

procedural_statement

注：不可综合

例：

```
while (BY>0)
begin
Acc=Acc<<1;
By=By-1;
end
```

举 例

```
while (control[1] & control[2])
  begin
        shift_register1 = shift_register1 << 1;
        count = count + 1;
   end
```

只要条件（control[1] & control[2]）成立，移位寄存器shift_register1被操作一次，并在计数器count上记一次数。

for循环

for(*initial_assignment;condition;step_assign ment*)

procedural_statement

注：可综合

例：

```
for ( i = 0; i < 9; i = i + 1)
    begin
        $display ("i = %d", i);
    end
```

在屏幕上打出从0到9的阿拉伯数字。

举 例

```
integer K;
for (K=0 ; K < MAX_RANGE ;
    K = K + 1)
begin
if(Abu s[K] == 0)
Abu s[K] = 1;
else if(Abu s[k] == 1)
Abu s[K] = 0;
else
$display ( "Abu s[K] is an x or a
    z");
end
end
```

小结

- Verilog中的控制语句和C语言很类似，比较容易理解。但是应该注意到在Verilog语言中这些语句表示的不是一个直接的计算过程，它们表示的是逻辑电路硬件的行为。
- 语句细微的差别其含义有很大的不同，通过综合生成的对应的硬件也有很大的变化。必须认真理解这些细节才能够设计出符合要求的逻辑。
- 需要注意：
  - 条件语句中是否存在无关的位，if else语句的else是不是设计中想要的行为；
  - 在case语句中如果条件都不符合究竟如何处理；
  - for循环变量的增加与C不同，不能用简化的写法。