

模拟与数字电路

Analog and Digital Circuits



课程主页 扫一扫

第十一讲：**Verilog 介绍**

Lecture 11 : **Verilog Introduction**

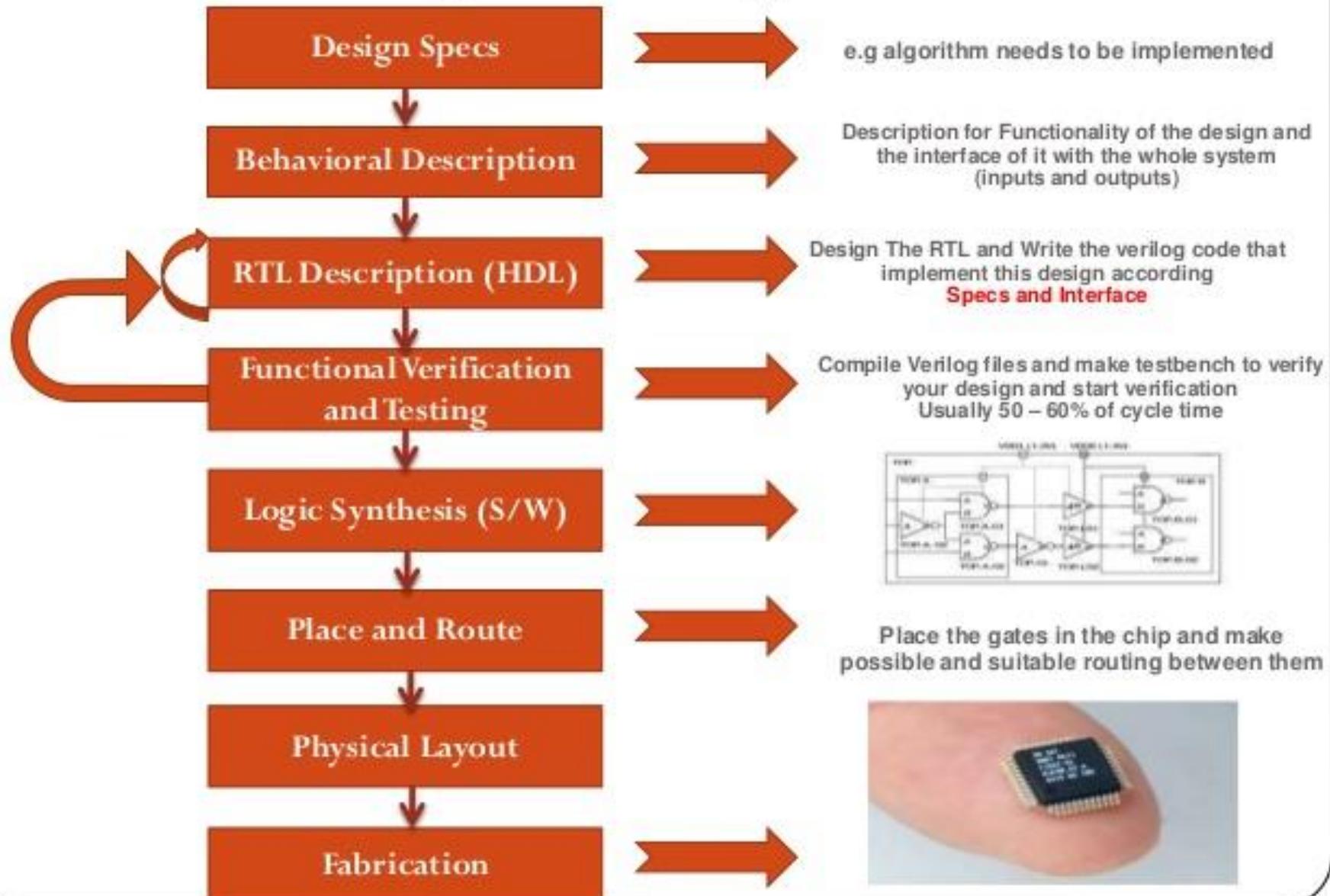
主讲：陈迟晓

Instructor : Chixiao Chen

提纲

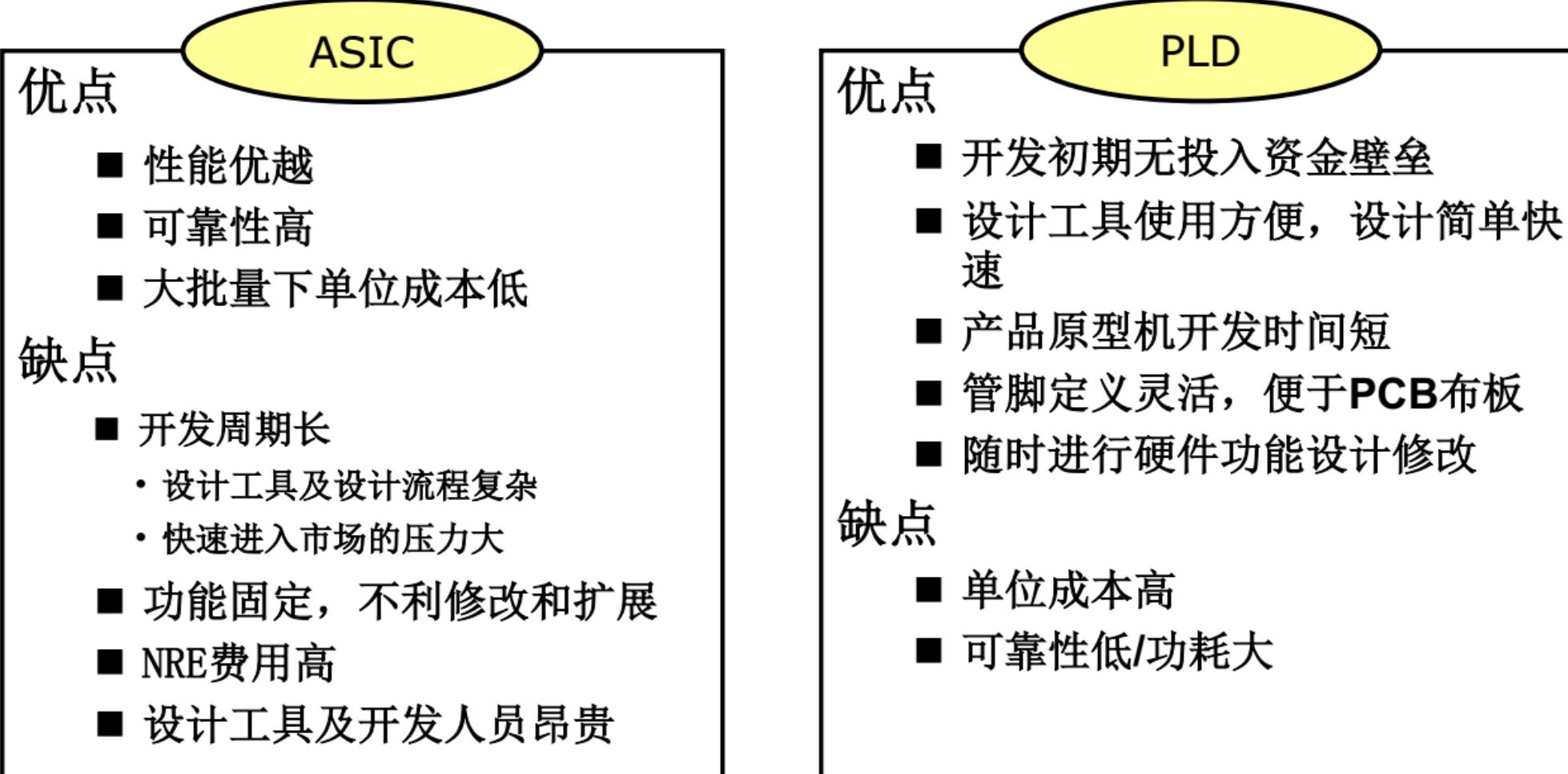
- 复习
 - 布尔逻辑的化简方法
- 数字流程
- Verilog always 块
- Verilog中的条件判断与循环

Digital Design Flow



- 数字设计流程
- 逻辑综合
- 布局布线

PLD vs. ASIC



Verilog HDL基本要素

```
module module_name (port_list);
    port declarations
    data type declarations
    circuit functionality
    timing specifications
endmodule
```

注意点：

- 大小写敏感
- 所有关键词须小写
- 空格用于增加可读性
- 分号是语句终结符
- 单行注释： //
- 多行注释： /* */
- 时间规范用于仿真

模块（module）和端口（port）

► 模块声明

```
module <模块名> (<模块端口列表>);  
<模块内容>  
endmodule
```

► 端口列表

- 列出所有端口名称

► 端口声明

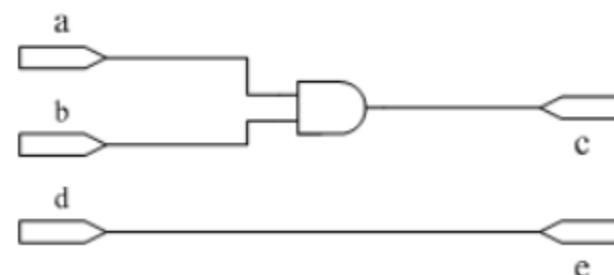
```
<端口类型> <端口名>;
```

► 端口类型

- **input** → 输入端口
- **output** → 输出端口
- **inout** → 双向端口

例：

```
module hello_world(a,b,c,d,e);  
    input a, b, d;  
    output c, e;  
  
    assign c = a & b;  
    assign e = d;  
  
endmodule
```



➤常量

- 参数 (parameter)

➤变量

- 线网型 (nets type)

- wire型最常用

- 寄存器型 (register type)

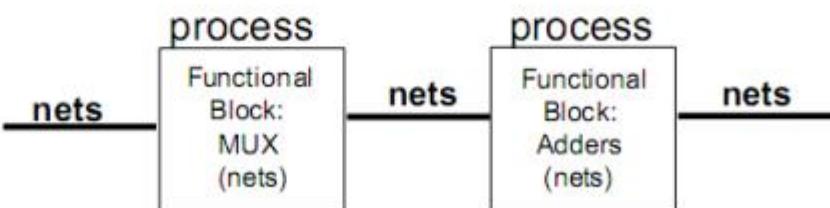
- 标量
 - 向量
 - 数组

➤线网型

- 用关键词**wire**等声明
- 相当于硬件电路里的物理连接，特点是输出值紧跟输入值变化

例 : **wire[7:0] in, out;**

assign **out=in;**



常量

➤ **parameter** 可用来定义常量

例： **parameter size=8;**

reg [size-1:0] a, b;

*常量的合理正确使用在高级编程和大型程序设计中很重要！

数字 (Numbers)

► sized, unsized: <位宽>'<进制><数字>

- sized: 3'b010 // 3位二进制数字, 值为010
- unsized:
 - <进制>默认为十进制;
 - <位宽>默认为32-bit.

► 进制

- 十进制 ('d 或 'D)
- 十六进制 ('h 或 'H)
- 二进制('b 或 'B)
- 八进制 ('o 或 'O)

► 负数——在<位宽>前加负号

- 例: -8'd3

► 特殊数符

- ‘_’ (下划线): 增加可读性
- ‘x’或’ X’ (未知数)
- ‘z’ 或’ Z’ (高阻)

► 若定义的位宽比实际位数长

- 如果高位是0, x, z, 高位分别补0, x, z;
- 如果高位是1, 左边补0.

运算符 (Operators)

►1. 算术运算符 (Arithmetic)

| | |
|---|----|
| + | 加 |
| - | 减 |
| * | 乘 |
| | 除 |
| % | 取模 |

►2. 逻辑运算符 (Logical)

| | |
|----|-----|
| && | 逻辑与 |
| | 逻辑或 |
| ! | 逻辑非 |

►3. 位运算符 (Bitwise)

| | |
|--------|------|
| ~ | 按位取反 |
| & | 按位与 |
| | 按位或 |
| ^ | 按位异或 |
| ^~, ~^ | 按位同或 |

►4. 关系运算 (Relational)

| | |
|----|-------|
| < | 小于 |
| <= | 小于或等于 |

►注: <=也用于表示一种赋值操作

运算符 (Operators)

►5. 等价运算(Equality)

`==` 等于

`!=` 不等于

`==` 全等

`!==` 不全等

►7. 移位(Shift)

`>>` 左移

`<<` 右移

►9. 位拼接 (Concatenation)

{ }

►8. 条件(Conditional)

`?:`

3.1 组合电路中的always块

- Verilog HDL使用一些顺序执行的过程语句来进行行为描述。这些语句封装在一个always块或initial块中，但只有always块能够进行综合，生成能够执行逻辑运算或控制的电路模块；
- always块可以看成一个包含内部过程描述语句的黑盒子，过程语句包含多种结构，但是很多都没有对应的硬件，编码不佳的always块通常会导致不必要的复杂实施或者根本无法综合。

3.1.1 基本语法格式

□ 敏感信号列表的always块的简化使用格式如下：

```
always @([敏感信号列表])  
  begin [可选的模块名]  
    [可选的本地变量声明];  
    [顺序执行语句];  
    [顺序执行语句];  
    ...  
  end
```

□ 敏感信号列表是always块响应的信号和事件列表，对于组合电路，应该包含所有的输入信号。

3.1.1 基本语法格式

□ 敏感信号列表的always块的简化使用格式如下：

```
always @([敏感信号列表])  
  begin [可选的模块名]  
    [可选的本地变量声明];  
    [顺序执行语句];  
    [顺序执行语句];  
    ...  
  end
```

□ 敏感信号列表是always块响应的信号和事件列表，对于组合电路，应该包含所有的输入信号。

3.1.1 基本语法格式

- @[敏感信号列表]项实际上是一个时序控制结构，它是可综合always块中的唯一时序控制结构。模块体包含任意数目的过程语句，当模块体只有一条语句时，定界符begin和end可以省略；
- 敏感信号可分为两种类型：一种是电平敏感型，一种是边缘敏感型。对于组合电路，一般采用电平触发；对于时序电路，一般由时钟边沿触发，Verilog HDL提供了posedge和negedge两个关键词来分别描述上升沿和下降沿。

3.1.2 过程赋值

□ 过程赋值只能用在**always**块或**initial**块中，有两种赋值方式：阻塞赋值和非阻塞赋值，其基本语法格式如下：

阻塞赋值：[变量名] = [表达式]； 非阻塞赋值：[变量名] <= [表达式]；

- 在阻塞赋值中，在执行下一条语句前，一个表达式只能赋给一个数据类型的值，可以理解为赋值阻断了其他语句的执行，与C语言中的正常变量赋值行为相似。
- 在非阻塞赋值中，表达式的值在**always**块结束时进行赋值，这种情况下赋值没有阻断其他语句的执行。

3.1.2 过程赋值

□在**Verilog**的初学者经常会混淆阻塞赋值和非阻塞赋值，不能正确理解它们的区别可能会导致意外的行为或竞争条件，它们的基本使用原则是：

- (1) 组合电路建议使用阻塞赋值；
- (2) 时序电路建议使用非阻塞赋值。

□因为本章我们关注的是组合电路，因此只使用阻塞赋值语句。

3.1.4 简单实例----1位比较器

- 1位比较器
- 代码如例3.1所示：
- 因为**eq**, **p0**和**p1**信号在**always**块内赋值，它们都声明为**reg**型数据类型，敏感列表包括**i0**和**i1**，并由逗号隔开，当其中一个发生变化时，**always**块则被激活。

```
module eq1_always
(
    input i0, i1,
    output reg eq // eq 声明为reg类型
);
reg p0, p1; // p0 和 p1 声明为reg类型
always @(i0, i1) // i0 和 i1 必须在敏感信号列表中
begin
    // 语句的顺序是很重要的
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0 | p1 ;
end
endmodule
```

例3.1

3.1.4 简单实例----1位比较器

- 三条阻塞赋值语句顺序执行， 和C程序中的语句非常类似， 语句的顺序十分重要， 在使用前p0和p1必须赋值。
- 要正确建立所需的行为模型， 组合电路的敏感列表**必须包含所有的输入信号**， 忽略一个信号就可能导致综合和仿真结果不一致，在Verilog HDL 2001中， 我们可以用下面的符号： **always @*** 来隐式地包含所有输入信号，在本书中， 我们在组合电路描述中使用这种结构。

3.1.4 简单实例----过程赋值语句和持续赋值语句

例3.2中的代码，使用的是过程赋值语句，它完成的功能是执行a，b和c的逻辑与运算（即 $a \& b \& c$ ），综合的电路如图3.1(a)所示。

例 3.2 使用一个变量的与电路

```
module and_block_assign( input  a , b , c,  
    output reg  y );  
    always @ *  
    begin  
        y = a ;  
        y=y&b;  
        y=y&c;  
    end  
endmodule
```

图3.1(a) 综合的电路



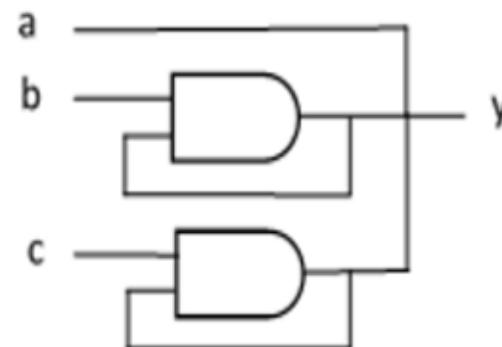
3.1.4 简单实例----过程赋值语句和持续赋值语句

- 如果我们采用类似如例3.3中的持续赋值语句，则描述结果不正确。
- 在例3.3的代码中，每条程序赋值综合一部分电路，左端3次出现的y表明三个输出捆绑在一起，对应的电路如图3.1(b)所示，这显然不是我们想要的设计结果。

例 3.3 与电路的不正确代码

```
module and_cont_assign( input a,b,c,  
    output y  
);  
    assign y = a;  
    assign y = y & b;  
    assign y = y & c ;  
endmodule
```

图3.1 (b) 不正确的电路图



3.2 条件语句

- 条件语句有**if-else**语句和**case**语句两种。
- 它们都是顺序语句，应该放在**always**块内使用。
- 以下分别结合实例对这两种语句进行介绍

3.2.1 if-else语句----语法

□ If-else语句的简化语法如下：

```
if [表达式]
begin
    [顺序执行语句];
    [顺序执行语句];
end
else
begin
    [顺序执行语句];
    [顺序执行语句];
end
```

- [表达式]项一般为逻辑表达式或者关系表达式，也可以是一位的变量。语句先对表达式判断，如果表达式为真，则执行下面分支的语句，否则执行else分支的语句。
- else分支具有选择性，可以省略。如果分支里只有一条语句，则定界符begin和end可以省略。

3.2.1 if-else语句----语法

□多条if语句可以“级联”，以执行多布尔条件并建立优先级，如下所示：

```
if [表达式1]
    ...
else if [表达式2]
else if [表达式3]
    ...
else
```

3.2.1 if-else语句----实例

□4位优先编码器：

□4位优先编码器有四个优先级：r[3]、
r[2]、r[1]和r[0]作为一组4位输入信号
，**r[3]**优先级最高，输出是最高优先级的二进制代码，优先编码器的功能表如表3.1所示，HDL代码如例3.4所示：

表3.1 四输入优先编码器功能表

| 输入 (r[3] r[2] r[1] r[0]) | 输出 (y[2] y[1] y[0]) |
|--------------------------|---------------------|
| 1 x x x | 1 0 0 |
| 0 1 x x | 0 1 1 |
| 0 0 1 x | 0 1 0 |
| 0 0 0 1 | 0 0 1 |
| 0 0 0 0 | 0 0 0 |

3.2.1 if-else语句----实例

□代码首先检查信号r[3]请求，如果为1则将100赋给y，如果r[3]为0，则继续检查信号r[2]请求并重复以上过程，直至所有请求信号检查完毕。

□注意当r[3]为1时布尔表达式（r[3]==1'b1）为真，由于在Verilog中真值也可表示为1'b1，因此表达式也可以写成（r[3]）。

例 3.4 使用if语句的优先编码器

```
module prio_encoder_if(input [3:0] r,  
    output reg [2:0] y);  
    always @*  
        if (r[3]==1'b1)      // 可以写成 (r[3])  
            y = 3'b100;  
        else if (r[2]==1'b1) // 可以写成 (r[2])  
            y = 3'b011;  
        else if (r[1]==1'b1) // 可以写成 (r[1])  
            y = 3'b010;  
        else if (r[0]==1'b1) // 可以写成 (r[0])  
            y = 3'b001;  
        else  
            y = 3'b000;  
    endmodule
```

3.2.1 if-else语句----实例

□2-4译码器

- 根据n位输入的信号，一个n- 2^n 二进制译码器将 2^n 位输出中的1位置位，2-4译码器的真值表如表3.2所示。
- 该电路包括控制信号端en，使能译码功能，HDL代码如例3.5所示。

表3.2 带使能的2-4译码器的真值表

| en | a[1] | a[0] | y |
|----|------|------|------|
| 0 | x | x | 0000 |
| 1 | 0 | 0 | 0001 |
| 1 | 0 | 1 | 0010 |
| 1 | 1 | 0 | 0100 |
| 1 | 1 | 1 | 1000 |

3.2.1 if-else语句----实例

□4位优先编码器代码首先检查使能位en是否是1，如果条件为假（即en为0），则设置输出y为无效状态；如果条件为真，则检测4个二进制数的组合顺序，设置输出y的状态。

□注意布尔表达式（en==1'b0）也可以写成（~en）。

例3.5 使用if语句实现的二进制译码器

```
module decoder_2_4_if(input [1:0] a,input en,
                      output reg [3:0] y);
  always @*
    if (en==1'b0)
      y = 4'b0000;
    else if (a==2'b00)
      y = 4'b0001;
    else if (a==2'b01)
      y = 4'b0010;
    else if (a==2'b10)
      y = 4'b0100;
    else
      y = 4'b1000;
endmodule
```

3.2.2 case语句

- 相对于if语句只有两个分支而言，csae语句是一种多分支语句，故case语句可用于描述**多条件分支电路**，如译码器、数据选择器、状态机及微处理器指令译码等。
- case语句有**case**、**casez**和**casex**三种表示方式，以下分别进行介绍。

3.2.2.1 case语句语法

□ case语句的简化语法如右所示：

□ case语句是一条多路决策语句，其将**case[表达式]**和多个表达式**[分支项]**进行比较，程序跳入与当前**[表达式]**相等的**[分支项]**对应的分支执行。

□ 最后一条分支为可选的，关键词是**default**，包含**[表达式]**未指定的所有值。如果一条分支中只有一条语句，则定界符begin和end可以省略。

```
case [表达式]
[分支项1]:
begin
[顺序执行语句];
...
end
[分支项2]:
begin
[顺序执行语句];
...
end
[分支项3]:
begin
[顺序执行语句];
...
end
...
default:
begin
[顺序执行语句];
end
endcase
```

3.2.2.2 case语句实例

□我们采用和if-else语句例子中相同的优先编码器和译码器来说明case语句的用法.

□2-4译码器使用case语句的HDL代码如例3.6所示

例3.6 使用case语句实现的二进制译码器

```
module decoder_2_4_case
  (input [1:0] a,
   input en,
   output reg [3:0] y);
  always @*
    case ({en, a})
      3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
      3'b100: y = 4'b0001;
      3'b101: y = 4'b0010;
      3'b110: y = 4'b0100;
      3'b111: y = 4'b1000; //也可以使用default语句
    endcase
  endmodule
```

3.3 循环语句

□ 在Verilog HDL中共有4种类型的循环语句，用来控制语句的执行次数。它们分别是：

- **forever**：连续执行语句；多用在“**initial**”块中，用来生成时钟等周期性波形。
- **repeat**：连续执行一条语句n次。
- **while**：执行一条语句直到某个条件不满足，如果一开始条件就不满足，则语句一次也不执行。
- **for**：有条件的循环语句。

3.3.1 for语句

□ for语句的一般形式为：

for (表达式1; 表达式2; 表达式3)

begin

[顺序执行语句];

[顺序执行语句];

...

end

□ 它的执行过程是：

(1) 先求解表达式1;

(2) 再求解表达式2，如果其值为真（非0），
则执行下面的第(3)步；如果为假（0），则
结束循环，转到第(5)步；

(3) 执行**begin/end**块中的顺序执行语句，然后
求解表达式3；

(4) 转回第(2)步继续执行；

(5) 执行**for**语句下面的语句。

□ 当顺序执行语句只有一条时，定界符**begin**和**end**可以省略。

3.3.1.2 for语句实例

□for循环实现两个8位二进制数的乘法操作

例 3.9：2个8位二进制数相乘的for语句描述

```
module mult_for (input [8:1] op0, input [8:1] op1, output reg [16:1] result );
    integer i;
    always @*
    begin
        result = 0;
        for(i=1;i<=8;i=i+1)
            if(op1[i])
                result = result + (op0 << (i-1));
    end
endmodule
```

3.3.2 repeat语句-语法

□ repeat语句的使用格式如下：

```
repeat (表达式)
```

```
begin
```

```
    [顺序执行语句];
```

```
    [顺序执行语句];
```

```
    . . .
```

```
end
```

□当顺序执行语句只有一条时，定界符begin和end可以省略。

□在repeat语句中，其表达式通常为常量表达式，用来指定循环执行的次数。

3.3.2 repeat语句-实例

□下面的例子用repeat循环实现两个8位数的二进制数的乘法操作，实现代码参见例3.10所示。

```
module mult_repeat  
(  
    input [8:1] op0,  
    input [8:1] op1,  
    output reg [16:1] result  
);  
    reg [16:1] tempa;  
    reg [8:1] tempb;  
    always @*  
    begin  
        result = 0;
```

例3.10

```
tempa = op0;  
tempb = op1;  
repeat(8)  
begin  
    if(tempb[1])  
        result = result + tempa;  
    tempa = tempa << 1;  
    tempb = tempb >> 1;  
end  
end  
endmodule
```

3.3.3 while语句---语法

□while语句的使用格式如右所示：

□当顺序执行语句只有一条时，定界符
begin和end可以省略。

□while语句在执行时，首先判断表达式是否为真，如果为真，则执行后面的语句或语句块，然后再回头判断表达式是否为真，为真的时候，再执行一遍后面的语句，如此不断，直到表达式不为真。因此，在执行语句中，必须有一条改变表达式值的语句

```
while ( 表达式 )  
begin  
    [顺序执行语句] ;  
    [顺序执行语句] ;  
    ...  
end
```

3.3.3 while语句---实例

□下例用while循环实现两个8位二进制数的乘法操作，实现代码见例3.11所示。

例 3.11：2个8位二进制数相乘的while语句描述

```
module mult_while ( input [8:1] op0,input [8:1] op1, output reg [16:1] result );
    integer i=1;
    always @*
    begin
        result = 0;
        while (i<=8)
            if(op1[i])
                result = result + (op0 << (i - 1));
            i = i + 1;
    end
endmodule
```

3.4 always块的一般编码原则

- Verilog用于建模和综合，当编写可综合代码时，我们需要知道不同的语言结构如何与硬件匹配，尤其是always块，因为变量和过程语句可以在模块内使用；
- 我们要谨记编写代码的目的是**综合为硬件电路**而不是用C语言描述顺序算法，做不到这些会经常导致无法综合的代码，不必要的复杂实施，或者在仿真和综合之间产生差异。
- 在本部分，我们讨论一些常见错误并提出一些编码原则。

3.4.1组合电路代码中的常见错误

□组合电路代码中的常见错误主要包括：

- ① 变量在多个always块中赋值
- ② 不完整的敏感信号列表及不完整分支
- ③ 输出赋值

□下边分别讨论这些错误

3.4.1组合电路代码中的常见错误

□ 变量在多个always块中赋值

□ 在Verilog中，变量（出现在左端部分）在多个always块中赋值，例如以下代码段中两个always块都含y变量：

```
reg y;  
reg a, b, clear;  
...  
always @*  
  if (clear) y = 1'b0;  
always @*  
  y = a&b;
```

也可以进行仿真，但是无法综合。

3.4.1组合电路代码中的常见错误

口不完整的敏感列表

口对于组合电路，要求**所有的输入信号**都应该包含在敏感信号列表中

口例如2输入与门可以写成：

```
always @ (a, b) // a和b都在敏感列表中  
    y = a&b;
```

口如果**不包含b**，代码变成：

```
always @ (a) // b不在敏感列表中  
    y = a&b;
```

口由于always块对b“不敏感”，当b发生变化时，y仍保持之前的值不变。

口导致**仿真与综合**的结果不一致。

3.4.1组合电路代码中的常见错误

口不完整分支和不完整输出赋值

- 口组合电路always模块的变量如果没有赋值则保持原来的值。
- 口在综合时，这将导致内部状态（通过闭合反馈回路）或存储元件（锁存器）的产生
- 口为了防止always块中意外的存储器，所有输出信号在任何时候都应该赋予恰当值

3.4.1组合电路代码中的常见错误

口有两种方法来解决错误

口加上else分支并明确给所有输出变量赋值，代码如下所示：

```
always @*
  if (a > b)
    begin
      gt = 1'b1;
      eq = 1'b0;
    end
  else if (a == b)
    begin
      gt = 1'b0;
      eq = 1'b1;
    end
  end
```

口在always块的起始部分，给每个变量赋默认值，以包含未指定的分支和未赋值变量，代码如下所示，如果gt和eq之后未赋值，则认为是0。

```
always @*
  begin
    gt = 1'b0; // gt的默认赋值
    eq = 1'b0; // eq的默认赋值
    if (a > b)
      gt = 1'b1;
    else if (a == b)
      eq = 1'b1;
  end
```

3.4.1组合电路代码中的常见错误

- 必须保证任何时候y都被赋值
- 采用关键字default以包含所有未指定值，如（1）。
- 也可以用以下代码替代最后一条分支项表达式，如（2）。
- 在always块的开始部分赋给一个默认值，如（3）。

```
case (s)
  2'b00: y = 1'b1;
  2'b10: y = 1'b0;
  default: y = 1'b1;
endcase
```

(1)

```
case (s)
  2'b00: y = 1'b1;
  2'b10: y = 1'b0;
  2'b11: y = 1'b1;
  default: y = 1'bx;
endcase
```

(2)

```
y = 1'b0;
case (s)
  2'b00: y = 1'b1;
  2'b10: y = 1'b0;
  2'b11: y = 1'b1;
endcase
```

(3)

