浅析 Verilog HDL 硬件语义

ONIONI

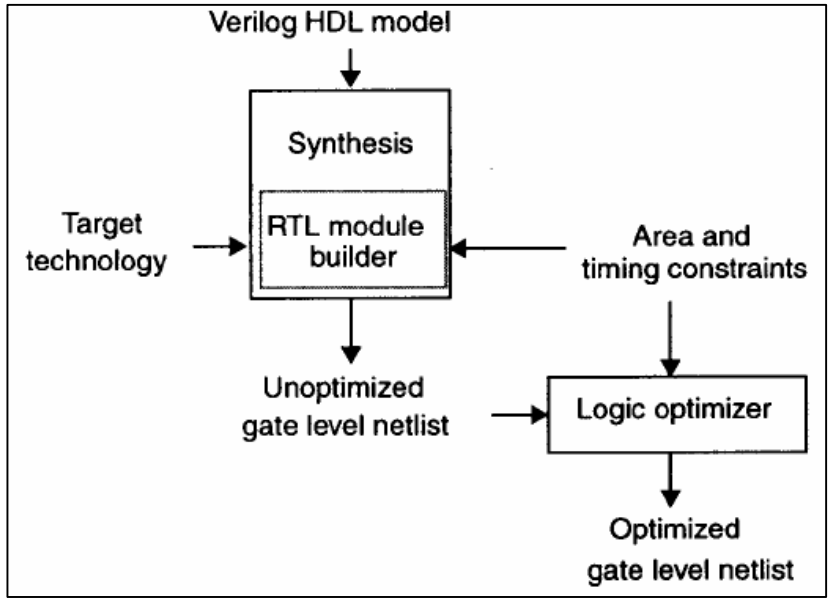
2004.8.14

内容简介

- 写本文的初衷是为了使已经对Verilog HDL 有过初步了解的读者，能够更进一步的了解Verilog HDL 与综合后的硬件之间的映射关系，从而把握Verilog HDL的应用规则，改善代码风格，写出高效，可综合的代码。
- 全文共分为3个部分：
  - Verilog HDL 的基本知识
  - Verilog HDL 从结构语句到门级映射
  - 模型的优化
- 参考文献：
  - Verilog HDL Synthesis A Practical Primer. J.Bhasker
  - A Guide To Digital Design And Synthesis. Samir Palnitkar
  - Verilog HDL Reference Manual. Synopsys

什么是综合？

- **综合**是从采用Verilog HDL描述的寄存器传输级（RTL）电路模型构造出门级网表的过程。
- 综合可能有个中间步骤，生成的网表是由一些RTL级功能块连接组成。这时就需要**RTL模块构造器**来针对用户指定的目标工艺从预定义库中构造或获取每一个必须的RTL功能块的门级网表。



- 产生门级网表之后，**逻辑优化器**读入网表并以用户指定的**面积和定时约束**为目标优化网表。这些**面积和定时约束**也可以用来指导**RTL模块构造器**适当的选取或生成RTL级功能块。
- 这里我们研究Verilog的硬件语义是为了分析以下几个问题：
  - 1）数据类型如何变成硬件？2）常量如何映射成逻辑值？3）语句如何转变成硬件？

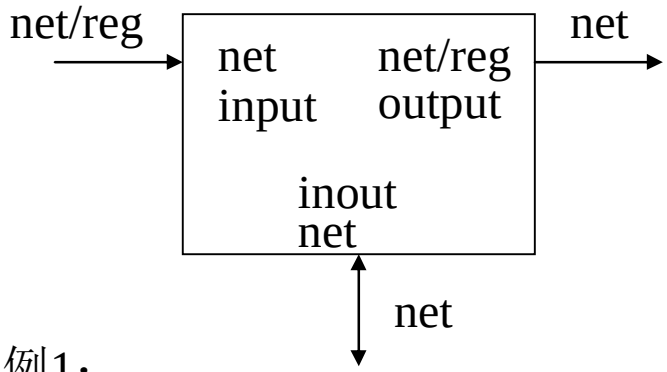
逻辑值体系

- 硬件建模中常用的值有：
  - 逻辑 0
  - 逻辑 1
  - 高阻抗
  - 无关值
  - 不定值
- Verilog HDL 对于无关值之外的其它值都作了明确的定义，当值 X 被赋给某个变量时，系统通常把此值视为无关值。
- Verilog HDL 中的值与硬件建模中的值之间的对应关系如下：
  - **0** <--> 逻辑 0
  - **1** <--> 逻辑 1
  - **z** <--> 高阻抗
  - **z** <--> 无关值（**casez**和 **casex** 语句中）
  - **x** <--> 无关值
  - **x** <--> 不定值
  - 注：没有激励时，**net** 类型变量通常被系统初始为 **z**，**reg** 类型变量则被初始为 **x**。

数据类型

- Verilog HDL 中的变量隶属于以下两种（不可综合的类型略去）
  - **网线数据类型（net）**
    - **wire** 仅有一个驱动源的节点。
    - **tri** 有多个驱动源的节点。
    - **wor，wand** 由多个驱动源产生“**或**”或者“**与**”逻辑输出的节点。
    - **supply0，supply1** 始终由**逻辑 0** 或 **逻辑 1** 驱动的节点。
  - **寄存器数据类型（reg）**
    - **reg** 任意宽度的二进制变量。（无符号，可指定宽度）
    - **integer** 默认为32位二进制补码的变量。（有符号，系统根据该变量的赋值语句优化宽度）
    - 注：**reg** 类型变量并不一定就被综合成寄存器。

- 在模块定义和例化模块时，数据类型的设定应按照以下规则：



- 例1：

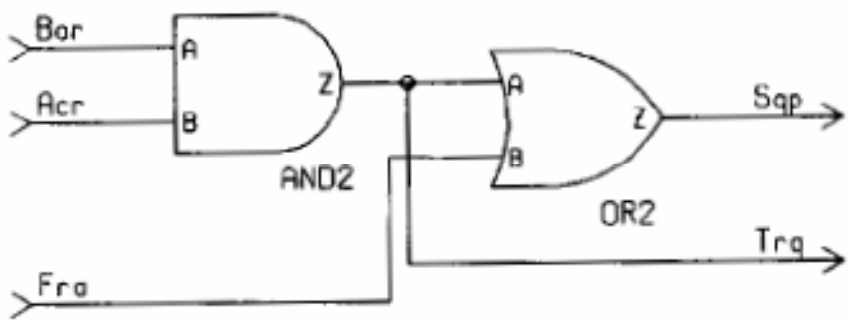
```
module ex1(in1,in2,out1,out2)
    input in1,in2 ;
    output out1,out2 ;
    wire in1,out1 ;
    reg in2,out2 ; // in2 应为net
endmodule
```

例化时，in 可为net/reg，out 应为 net

数据类型（续）

- **持续赋值**语句中左端应为net。
  assign a = b ;
  reg a ; // 非法
- **过程赋值**语句中左端应为reg。
  always a = b ;
  always c <= d ;
  wire a,c ; // 非法
- reg 类型变量如果在一个块语句中先被赋值，再被引用，不生成寄存器。
- 在右边的例2中，变量 **Trq** ，先被赋值，后被引用，其间没有必要存储该变量的值，因而未生成器存器。
  例2：

```
wire Acr, Bar, Fra;     // A wire is a net type.
reg Trq, Sqp;           // A reg is a register type.
. . .
always @ (Bar or Acr or Fra)
begin[1]
  Trq = Bar & Acr;
  Sqp = Trq | Fra;
end
```

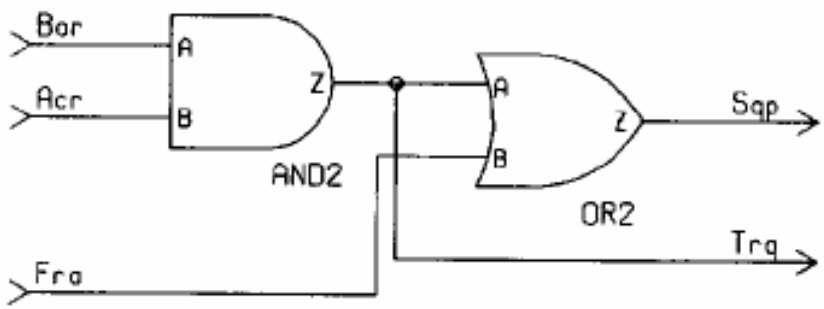


1: begin …. end 是一个块语句，内部的语句按顺序执行。此例中用的是阻塞赋值，没有生成寄存器，非阻塞赋值时则可能生成。以后将详细说明。

值保持器的硬件建模

- 硬件中有3种基本的值保持器：
  - **连线**
  - **触发器**（边沿触发）
  - **锁存器**（电平触发）
- net型变量被映射成硬件**连线**。
- reg型变量则根据上下文被映射成存储元件。（**触发器**或**锁存器**）
- 例3：（改自例2）

```
wire Acr, Bar, Fra;
reg Trq, Sqp;
. . .
always @ (Bar or Acr or Fra)
begin
  Sqp = Trq | Fra;
  Trq = Bar & Acr;
end
```



- 从上图中我们发现虽然变量 **Trq**在赋值前就被引用，但同样没有映射出存储元件，因为它没有在任何条件的控制下被赋值。
- 如果 **Trq**是在某个电平控制下被赋值，则被映射成**锁存器**。
- 如果 **Trq**是在某个边沿控制下被赋值，则被映射成**触发器**。
- 本文后续部分还将继续说明不同的结构语句对于值保持器硬件建模的影响。

常量与参数

- Verilog HDL 中有3种常量：**整形**、**实型**和**字符串型**。后两种的常量不能用来综合。
- 整形常量有两种形式：
  - 简单的十进制（有符号数）
  - 基数格式（无符号数）
- 只有赋值语句中指定了整形常量的位宽，综合时才使用指定的位宽，否则位宽为32。
- 请看以下示例：

30	32位有符号数
-2	二进制补码形式的32位有符号数
2'b10	位宽为2的无符号数
-6'b4	6位无符号数（-4的补码）

- 参数是命名常量。由于不能指定位宽，因此其位宽与其对应的常量位宽相同。

```
parameter RED = -1 ;
parameter READY = 2'b01 ;
```

- RED 为有符号常量参数，READY为位宽为2的参数。
- 注意：

defparam在某些综合器中无效。应尽量采用重载模块参数的方法。如：

```
Mem mem_inst
        # (32,1024)
       (clr,rstN,wrN,rdN,din,
        dout) ;
```

运算符体系

- **算术运算符：**
  - **+，-，*，/，%**（取模）
  - 可直接对加法器硬件建模，但要注意位宽的统一性，否则会丢失进位。乘法器则应调用综合器的**RTL功能块。**除法器在某些综合器中无效。
- **位运算符：**
  - **&，|，~，^，^~，~^**
  - 可直接用来逻辑门硬件建模，但要注意对于矢量，是按照位生成逻辑门的，多用于编码/解码器硬件建模。
- **逻辑运算符：**
  - **!，&&，||**
  - 多用于优先级电路的硬件建模。
- **关系运算符：**
  - **<，>，>=，<=**
  - 可直接对比较器硬件建模，也可用于生成优先级电路。
- **相等性运算符：**
  - **==，!=**（**===，!==** 不能综合）
  - 多用于编码/解码器硬件建模。
- **移位运算符：**
  - **>>，<<**
  - 多用于移位寄存器硬件建模。特别是对于位宽不定的移位寄存器建模。
- **条件运算符：**
  - **?:**
  - 多用于多路选择器硬件建模。

运算符体系（续）

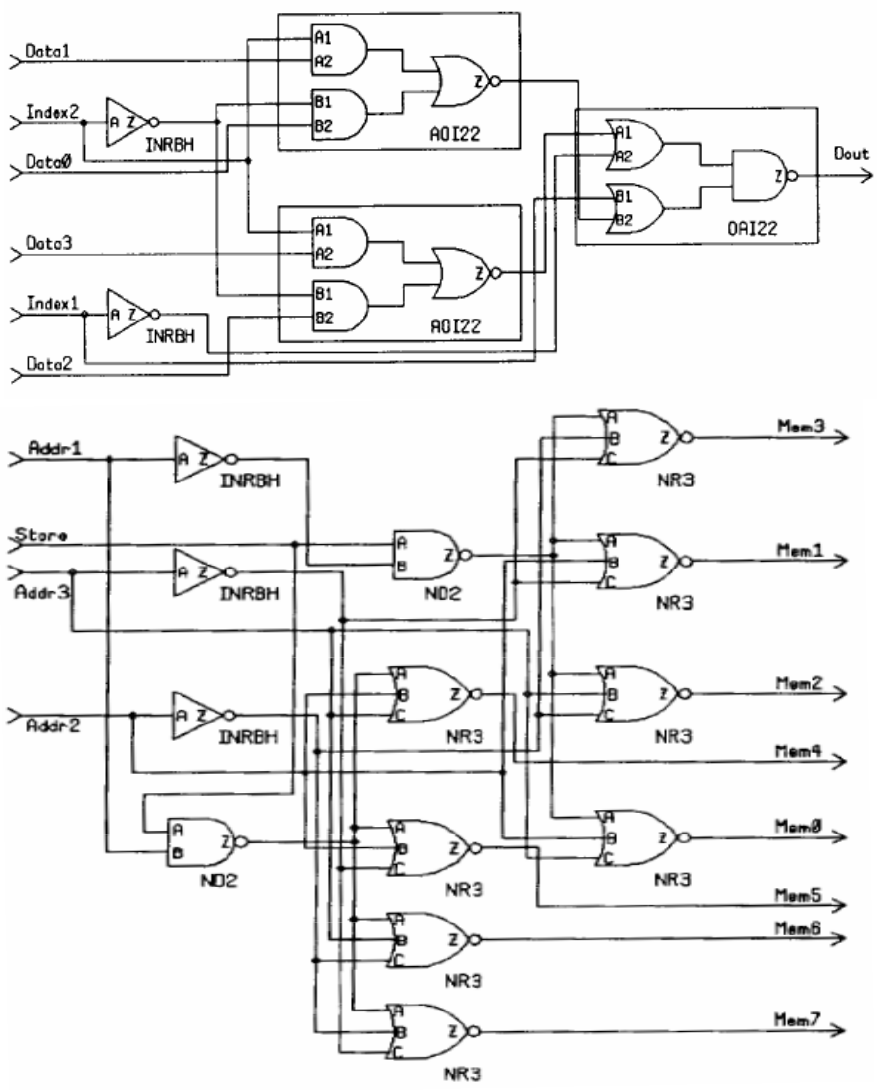
- 在Verilog HDL 的运算符体系中，必须注意以下三个问题：
  - 运算表达式中的变量如果为“**x**”，或“**z**”，则表达式的值为未知。应尽量避免这种情况的发生。
  - 运算表达式中变量的位宽应尽量统一，否则将无法顺利通过综合。
  - 应重视非常量下标的应用，可产生译码，编码，多路选择，等电路。

```
module NonComputeRight (Data, Index, Dout);
  input [0:3] Data;
  input [1:2] Index;
  output Dout;

  assign Dout = Data [Index];
endmodule
```

一个多路选择器

```
module NonComputeLeft (Mem, Store, Addr);
  output [7:0] Mem;
  input Store;
  input [1:3] Addr;

  assign Mem [Addr] = Store;
endmodule
```

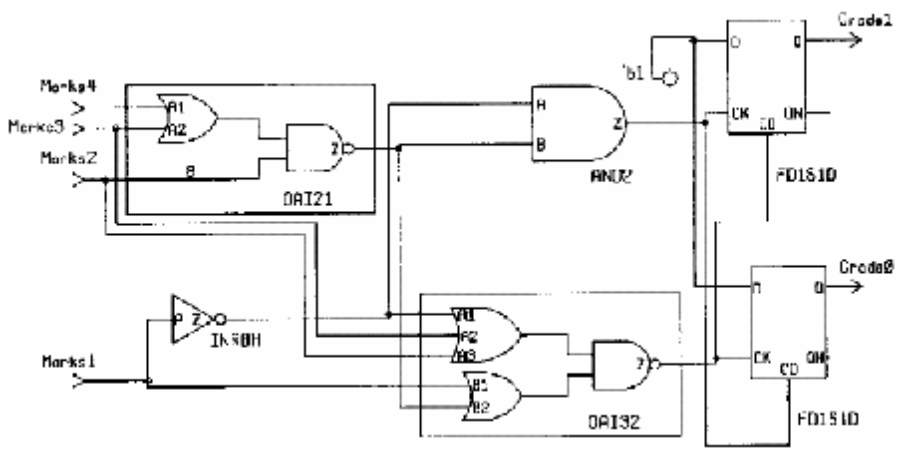
一个译码器



锁存器建模 ——If 语句

- **If** 语句中，应注意变量是否在所有分支中都被赋值。如果不是，则会产生锁存器。

```
module Compute (Marks, Grade);
  input [1:4] Marks;
  output [0:1] Grade;
  reg [0:1] Grade;
  parameter FAIL = 1, PASS = 2, EXCELLENT = 3;

  always @ (Marks)
    if (Marks < 5)
      Grade = FAIL;
    else if ((Marks >= 5) & (Marks < 10))
      Grade = PASS;
endmodule
```

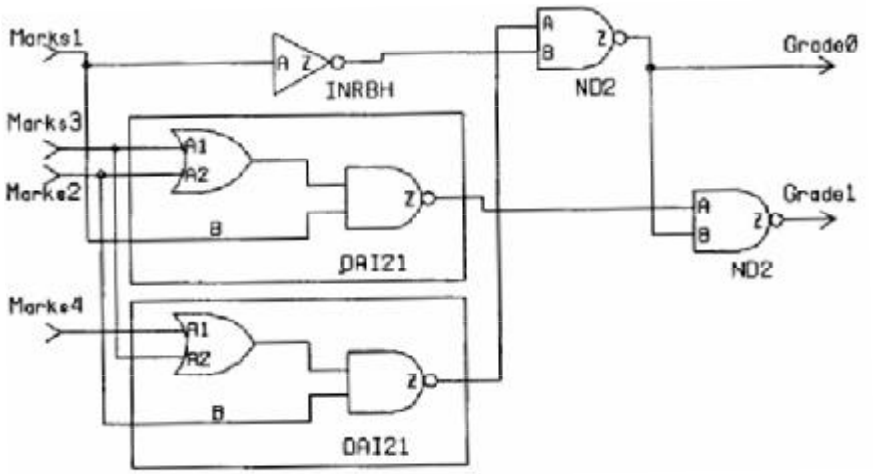


- 上例生成锁存器，如果在分支条件中再加上

  else

  Grade ＝ EXCELLENT ；

  则不会产生锁存器。



锁存器建模 ——Case 语句

- 与**If**语句一样，如果变量没有在所有分支中被赋值，也会生成锁存器。
- 通常为了避免生成**Case**语句生成锁存器，有以下几种方法。
  - 默认分支
  - 默认赋值
  - 综合指令 **full_case**。

```
module NextStateLogic (NextToggle, Toggle);
  input [1:0] Toggle;
  output [1:0] NextToggle;
  reg [1:0] NextToggle;
  always @ (Toggle)
    case (Toggle)
      2'b01 : NextToggle = 2'b10;
      2'b10 : NextToggle = 2'b01;
    endcase
```

会生成锁存器

```
always @ (Toggle)
  case (Toggle)                    // synthesis full_case
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
  endcase
```

```
always @ (Toggle)
  case (Toggle)
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
    default : NextToggle = 2'b01;
  endcase
```

默认分支

```
always @ (Toggle)
begin
  NextToggle = 2'b01;

  case (Toggle)
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
  endcase
end
```

默认赋值

- 注：**full_case** 指令可能导致设计模型和综合后的网表功能不一致。要慎用!

锁存器建模——小结

- 根据前面的几个例子，我们可以对锁存器建模的规则作出如下总结：
  - 变量在条件语句中被赋值。
  - 变量未在所有分之中被赋值。
  - 在**always**语句中多次调用需要保存变量值。
  - 必须同时满足以上3个条件，变量才能生成锁存器。
- 以下再通过几个例子来说明。

```
wire Control, Jrequest;
reg DebugX;
. . .
always @ (Control or Jrequest)
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = DebugX;
```
（1）

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;
  Bdy = DebugX;
end
```
（2）

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;
  if (Jrequest)
    Bdy = DebugX;
end
```
（3）

```
always @ (Control)
begin
  if (Control)
    DebugX = Jrequest;
  else
    DebugX = Bdy;

  if (Jrequest)
    Bdy = DebugX;
  else
    Bdy = 'b1;
end
```
（4）

- 上面4段程序中只有程序（3）中的**Bdy**会被综合成锁存器。
- 其他的几段程序都违反了锁存器建模的规则，综合时会出现警告，并提示可能发生功能不一致。

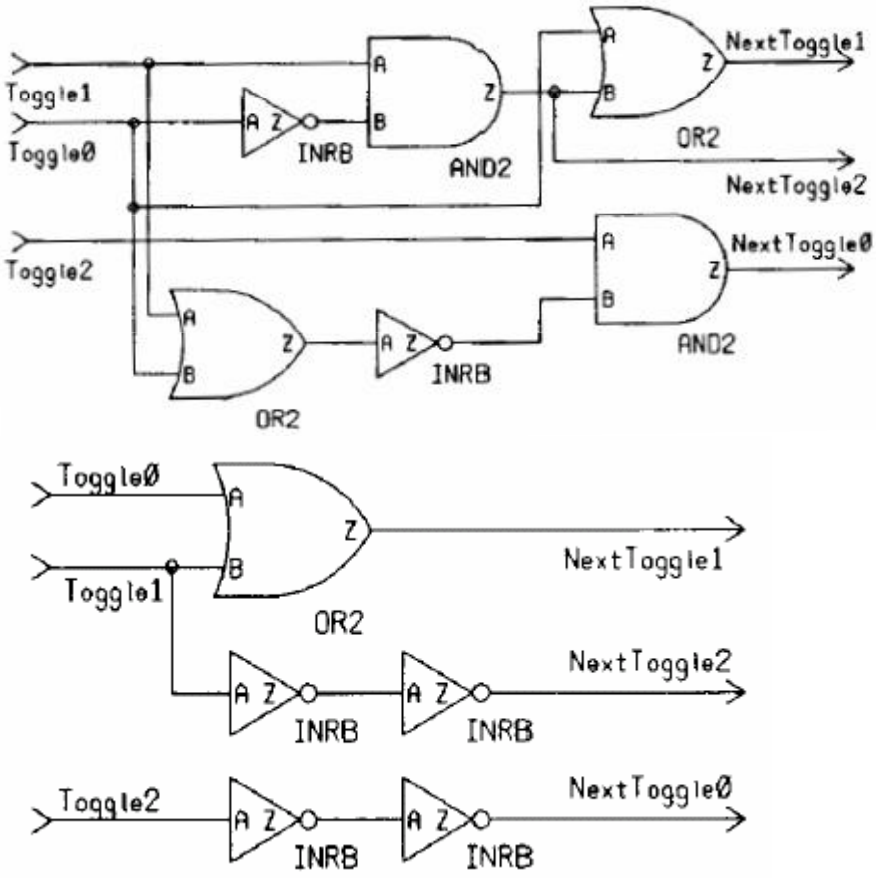
优先级编码器/译码器建模——Case语句

- **Case**语句暗示了检查分支项的优先级顺序。极易综合出优先级逻辑（if…，else if…，else…），如果分支项很多，则会导致嵌套得很深。
- 如果设计者知道所有的分支项相互排斥，可是用综合指令**parallel_case**来生成译码电路，取代优先级电路。

```
module ParallelCase (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  always @ (Toggle)
    casex (Toggle)
      3'bxx1 : NextToggle = 3'b010;
      3'bx1x : NextToggle = 3'b110;
      3'b1xx : NextToggle = 3'b001;
      default : NextToggle = 3'b000;
    endcase
endmodule
```

注意

```
// synthesis parallel_case
```



- 第一个图为优先级电路
- 第二个图为译码电路
- 显然，第二个图的电路更简洁，高效。

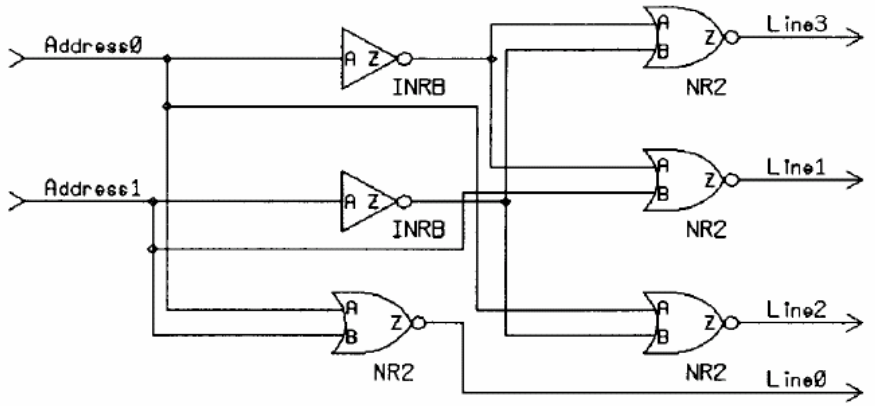
循环语句——For语句

- Verilog HDL 中有4种不同类型的循环语句：
  - **While** 循环
  - **For** 循环
  - **Forever** 循环
  - **Repeat** 循环
- 其中**for**循环语句是典型的能用于综合的循环语句。其综合是通过展开循环来实现的，这就要求对**for**循环的边界加以限制，要保证循环边界都是常量。

```
always @ (Address)
  for (J = 3; J >= 0; J = J - 1)
    if (Address == J)
      Line[J] = 1;
    else
      Line[J] = 0;
```

上述代码等价于：

```
if (Address == 3) Line[3] = 1; else Line[3] = 0;
if (Address == 2) Line[2] = 1; else Line[2] = 0;
if (Address == 1) Line[1] = 1; else Line[1] = 0;
if (Address == 0) Line[0] = 1; else Line[0] = 0;
```



- 当**for**循环次数很多时，应注意加上综合指令，

  // **synthesis loop_limit 200**

  否则综合器会报出警告，应尽量避免**for**语句多次使用。

触发器建模

- 触发器建模的规则如下：
  - 如果变量在时钟沿的控制下被赋值，则生成触发器。但有一个例外，若对变量的赋值和引用仅出现在同一条**always**语句中。则该变量会被视作中间变量，而不会生成触发器。
- **always** @ （<时钟事件>）

  <语句>
- 在触发器建模中应尽量使用非阻塞赋值，这样可以防止设计模型与综合出的网表之间存在的任何功能上的不一致。
- 对于变量的赋值，则不能受多个时钟控制，也不能受两种不同的时钟条件的控制。
- 同步预置位和清零：

  仅在时钟控制的**always**语句中描述同步预置位和清零逻辑就行了。
- 异步预置位和清零：

  使用特定形式的 **if** 语句，以下是一个简单的模板。

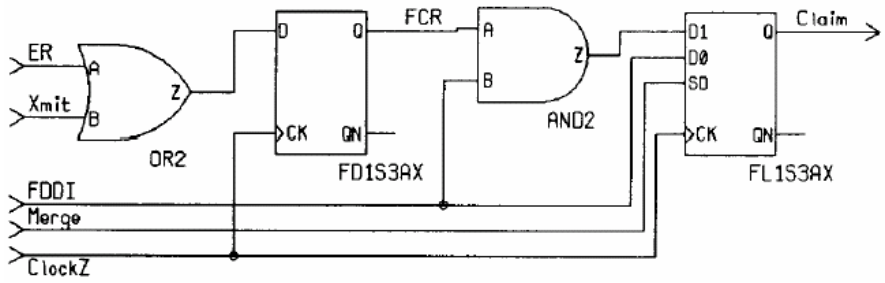
```
always @ (posedge A or negedge B or negedge C . . .
          or posedge Clock)
  if (A)                    // posedge A.
    <statement>             // Asynchronous logic.
  else if ( ! B)            // negedge B.
    <statement>             // Asynchronous logic.
  else if ( ! C)            // negedge C.
    <statement>             // Asynchronous logic.
  . . .                     // Any number of else if's.
  else                      // posedge Clock implied.
    <statement>             // Synchronous logic.
```

阻塞式与非阻塞式赋值

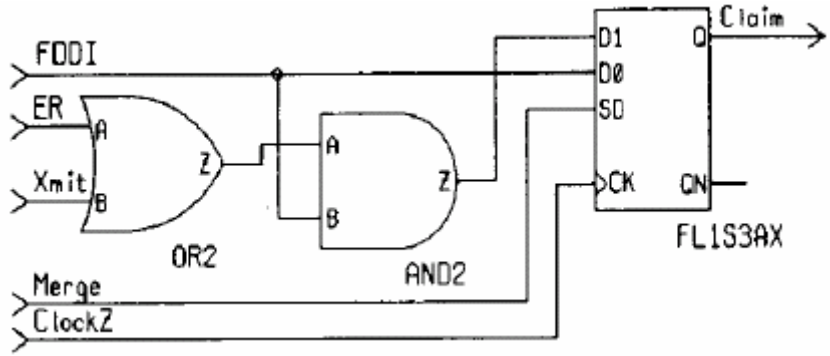
- 阻塞式与非阻塞式赋值的语义差异如下：
  - 阻塞赋值，在执行下一条语句之前就完成了对左端对象的赋值。
  - 非阻塞赋值：对左端对象的赋值安排在该仿真周期（一个时钟）结束时。（即赋值并不立即生效）
- 我们可以通过下面的例子清楚的看到它们之间的差异：

```
always @ (posedge ClockZ)
begin
  FCR <= ER | Xmit;              // Assignment 1.
    if (Merge)
      Claim <= FCR & FDDI;       // Assignment 2.
    else
      Claim <= FDDI;
  end
endmodule
always @ (posedge ClockZ)
begin
  FCR = ER | Xmit;               // Assignment 1.
```

```
    if (Merge)
      Claim = FCR & FDDI;        // Assignment 2.
    else
      Claim = FDDI;
end
```



非阻塞赋值



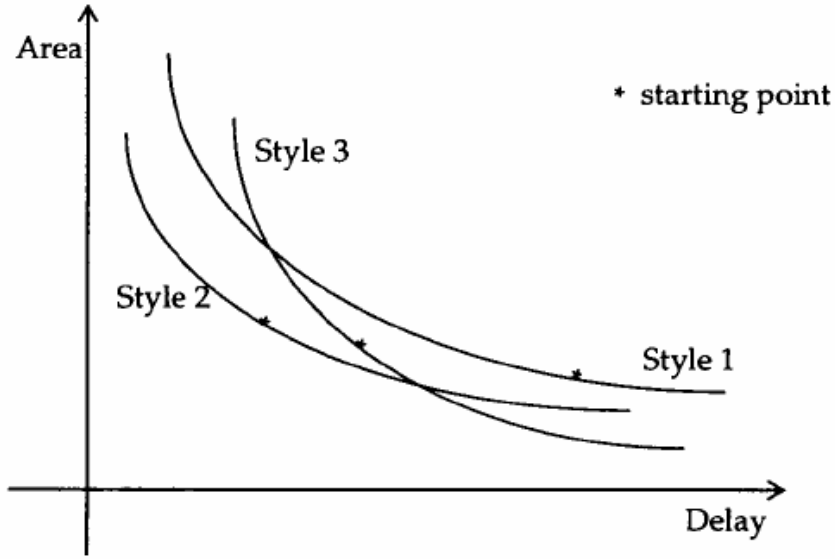
阻塞赋值

阻塞式与非阻塞式赋值（续）

- 从之前的例子，我们可以发现，阻塞赋值更容易生成组合逻辑（赋值立刻生效），而非阻塞赋值更容易生成时序逻辑（赋值不立刻生效）。我们在选择这两种赋值方式的时候，应慎重考虑时序所要受到的影响。
- 对于组合逻辑建模应注意：
  - 非阻塞赋值不反映逻辑流。
  - 需要将所有赋值对象都列入事件表中。
- 对于时序逻辑建模应注意：
  - 禁止使用阻塞赋值在多个块语句内对同一变量多次赋值，将有可能导致毛刺的生成。
  - 如果在某**always**语句内对变量赋值而在该**always**语句外读取变量，那么应采用非阻塞赋值。
  - 如果仅在一条**always**语句内，要求变量既被赋值，又被引用，则应改用阻塞赋值，这样引用的实时性较好。
- 通常我们建议多使用非阻塞赋值，因为阻塞赋值常常会引发设计与综合功能不一致的弊端。
- 对于同一变量，其赋值方式必须统一，否则综合器将无法完成该变量的综合。
- 如果设计者对于这两种赋值方式的语义差别有了较深的理解，则可以根据设计要求灵活使用。

模型的优化

- 前面我们对于Verilog HDL 的硬件建模方法作出了一定的归纳，但我们仍然无法保证建立的硬件模型具有较好的电路性能，因此，本节我们将继续讨论模型优化的相关问题。
- 综合所生成的逻辑易受到模型描述方式的影响。 把语句从一个位置移到另一个位置，或者拆分表达式都会对所生成的逻辑产生重大影响，这可能会造成综合出的逻辑门数有所增减，也可能改变其定时特性。
- 如右图所示，由Verilog HDL 代码综合出的网表所提供的优化起点不同，决定了逻辑优化器所能达到的终点（最佳面积/速度）。



- 本节将探讨几种优化思想，设计者通过改写Verilog HDL 综合模型的相应代码，从而得到更高品质的设计，并相应的缩短综合过程的运行时间。

资源分配

- 所谓资源分配指的是互斥条件下共享算术逻辑运算单元（ALU）的过程。
- 通常利用多路选择器的引入，来减少ALU的个数，从而节省资源。
- 如下示例：

```
if (! ShReg)
  DataOut = AddrLoad + ChipSelectN;
else if (ReadWrite)
  DataOut = ReadN + WriteN;
else
  DataOut = AddrLoad + ReadN;
```

改进后：

```
if (! ShReg)
begin
  Temp1 = AddrLoad;
  Temp2 = ChipSelectN;
end
else if (ReadWrite)
begin
  Temp1 = ReadN;
  Temp2 = WriteN;
end
else
begin
  Temp1 = AddrLoad;
  Temp2 = ReadN;
end

DataOut = Temp1 + Temp2;
```

- 虽然改进后的代码较长，但只用了1个加法器；而原来的代码却用了3个加法器。
- 显然，改进后，多产生了一个多路选择器。如果选择器十分复杂，则改进可能增加了复杂度。因此，在资源分配时应注意复杂度的权衡。

公共子表达式

- 在编写软件时，有一个思想贯穿始终，那就是**复用**。同样，我们在编写HDL代码时也应该坚持这种思想。
- 也就是说，我们要在代码中尽可能找出公共子表达式并重用以计算过的值。
- 公共子表达式的提取应注意：
  - 交换律和结合律的使用。

    lam = a – (b + c) ;

    ram = a + (b – c) ;

    应提取：temp = a – c ;
  - **for** 循环中的子式提到循环外。

    for (…..) **begin**

    ……； tip = car – 6 ;

    end

    应改为：

    temp = car – 6 ;

    for (…..) **begin**

    ……； tip = temp ;

    end
  - **if / case** 语句的互斥分支中的子式提到条件语句外。

    if (test)

    Ax = A & (B + C);

    else

    By = (B + C) | T;

    应改为：

    temp = B + C ;

    if (test)

    Ax = A & temp;

    else

    By = temp | T;

触发器/锁存器的优化

- 前面我们已经讲到了触发器和锁存器建模的规则，如果我们没有理解这些规则，综合出的网表就可能还有远远多于实际需要的触发器。请看下面的例子：

```
reg PresentState;
reg [0:3] Zout;
wire ClockA;
. . .
always @ (posedge ClockA)
  case (PresentState)
    0 :
      begin
        PresentState <= 1;
        Zout <= 4'b0100;
      end
    1 :
      begin
        PresentState <= 0;
        Zout <= 4'b0001;
      end
  endcase
```

改进后：

```
always @ (posedge ClockA)
  case (PresentState)
    0 : PresentState <= 1;
    1 : PresentState <= 0;
  endcase

always @ (PresentState)
  case (PresentState)
    0 : Zout = 4'b0100;
    1 : Zout = 4'b0001;
  endcase
```

- 前者的 Zout 也生成了4个触发器而实际上只需PresentState 一个，后者的改进显然更加高效。
- 可见，我们应注意适当的引入组合逻辑，以减少冗余的触发器。
- 对于锁存器，此方法同样适用。

设计规模

- **小型设计综合起来更快**

  通常逻辑电路规模在2000至5000门时，逻辑优化器的优化效果最佳。这意味着Verilog HDL 模型中的**always** 语句不能过长。对设计的描述应组织成多个**always**语句或者多个模块。

- **层次**

  根据always语句来保持Verilog HDL模型的层次关系是很有必要的。这使得综合工具能产生各子电路的层次关系，以便逻辑综合器能有效的对此进行处理。

- **宏结构**

  综合不是用来建立ROM或RAM之类的存储器的正确机制。通常在工艺库中就可以获得预定义的RAM。若需要RAM之类的模块，最好象对待元件那样，在模型中直接对其进行实例化，然后再综合此实例模型。综合工具只是为RAM建立一个黑盒，稍后设计者再把RAM模块连接至该黑盒中。

- 如果设计者遇到以下形式的语句，并期望综合器实现一个高效的乘法器，则有必要采取类似的措施。

  此时，设计者最好采用直接实例化乘法器的方法，而不是采用乘法运算符，因为乘法运算符通过综合未必能得到高效的乘法器。

Verilog HDL 设计——小结

- 前面我们对于Verilog HDL建模的一些代码细节，技巧作出了较为详细的说明，这里我们再次从设计模型的角度，探讨一下设计思想，同时对前面的部分作一个小结。
- 在开始设计之前，我们要先弄清楚所要设计的功能（如要实现的协议），并在头脑中整理模型中各个数据的行为流程，进一步也要清楚这些行为背后隐藏的逻辑流程。通过分析逻辑流程，确定模块的划分，以及模块之间通信功能，这有点类似软件中的进程间通信，是整个设计的核心。
- 对于复杂的设计，可能还需要专门用一个模块作为模块间通信信号的处理机，通常使用状态机来描述。
- 而对于设计的优化，实际上就是对逻辑流程进行**提取**，**拆分**，**移位**，**复用**的过程。这时需要考虑的是：
  - 复杂度与效率之间的权衡
  - 健壮度与冗余度之间的权衡
  - 面积与速度之间的权衡
- 基于设计日趋复杂的现状，我们应时常注意设计思想的总结和创新，交流，让思想的空间能够在更大的范围内不断的膨胀。