

综合与 Design Compiler

综合是前端模块设计中的重要步骤之一，综合的过程是将行为描述的电路、RTL 级的电路转换到门级的过程；Design Compiler 是 Synopsys 公司用于做电路综合的核心工具，它可以方便地将 HDL 语言描述的电路转换到基于工艺库的门级网表。本章将初步介绍综合的原理以及使用 Design Compiler 做电路综合的全过程。

§1. 综合综述

1.1 什么是综合？

综合是使用软件的方法来设计硬件，然后将门级电路实现与优化的工作留给综合工具的一种设计方法。它是根据一个系统逻辑功能与性能的要求，在一个包含众多结构、功能、性能均已知的逻辑元件的单元库的支持下，寻找出一个逻辑网络结构的最佳实现方案。即实现在满足设计电路的功能、速度及面积等限制条件下，将行为级描述转化为指定的技术库中单元电路的连接。

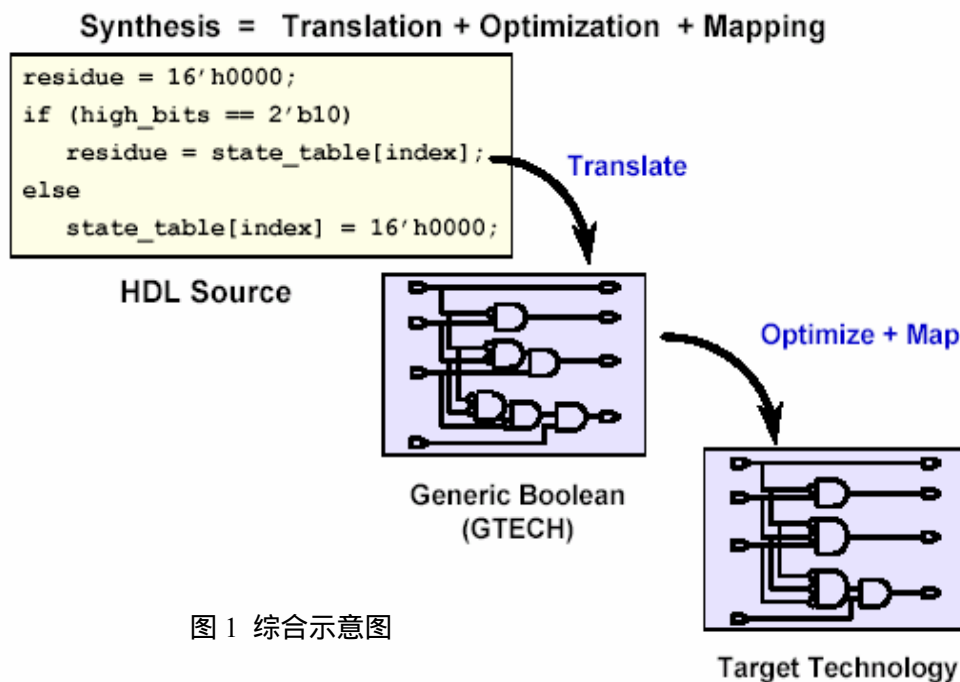


图 1 综合示意图

如图 4 - 1 所示，综合主要包括三个阶段：转换(translation)、映射(mapping)与优化(optimization)。综合工具首先将 HDL 的描述转换成一个与工艺独立(technology-independent)的 RTL 级网表(网表中 RTL 模块通过连线互联)，然后根据具体指定的工艺库，将 RTL 级网表映射到工艺库上，成为一个门级网表，最后再根据设计者施加的诸如延时、面积方面的约

束条件，对门级网表进行优化。

1.2 综合的不同层次

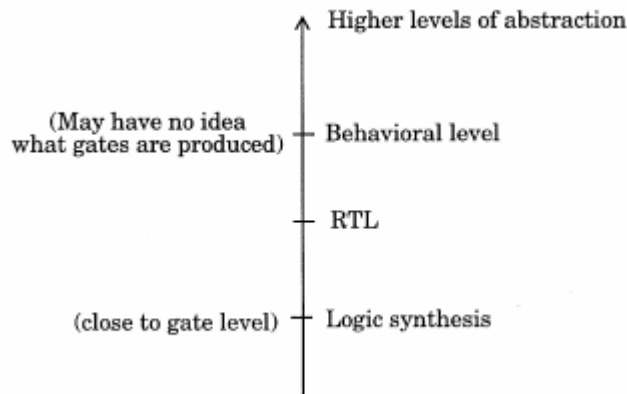


图 2 抽象层次

设计的不同的抽象层次如图 2 所示，随着抽象层次的升高，设计者对于最终硬件（门和触发器）的控制能力越来越小。设计者可以在上述的三个层次用 HDL 语言描述他的设计，根据 HDL 语言描述的层次的高低，综合也相应的可以分为逻辑级综合，RTL 级综合以及行为级综合。

1.2.1 逻辑级综合

在逻辑级综合中，设计被描述成布尔等式的形式，触发器、锁存器这样的基本单元采用元件例化(instantiate)的方式表达出来。下面是一个加法器的逻辑级描述，输出寄存——

```
module INCREMENT (A, CLOCK, Z);
    input [0:2] A;
    input CLOCK;
    output [0:2] Z;

    wire A1BAR, S429, DZ0, DZ1, DZ2;

    assign DZ1 = ! ((A[1] || DZ2) && (A[2] || A1BAR));
    assign DZ2 = ! A[2];
    assign DZ0 = ! ((A[0] || DZ2) && (A1BAR || S429));
    assign A1BAR = ! A[1];
    assign S429 = ! ((DZ2 || A1BAR) && A[0]);

    FD1S3AX    S0 (DZ2, CLOCK, Z[2]),
               S1 (DZ1, CLOCK, Z[1]),
               S2 (DZ0, CLOCK, Z[0]);
endmodule
```

它综合以后的电路网表如下图 3 所示,对比一下不难看出,逻辑级描述实际上已经暗示了综合以后的网表。

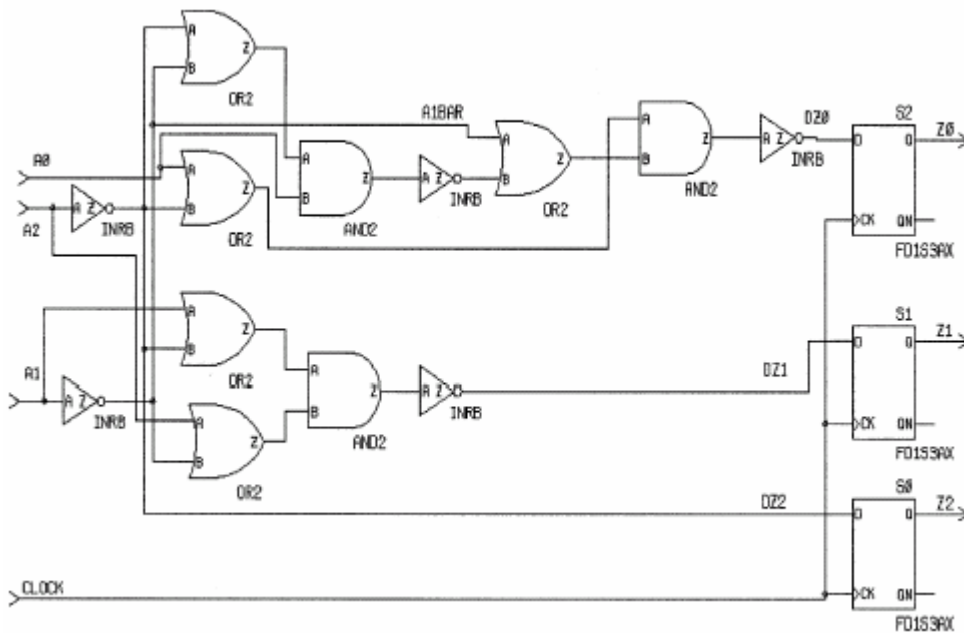


图 3 综合后的网表

1.2.2 RTL 级综合

与逻辑级综合不同,在 RTL 级综合中,电路的数学运算和行为功能分别通过 HDL 语言特定的运算符和行为结构描述出来。对于时序电路,我们可以明确的描述它在每个时钟边沿的行为。下面同样是一个加法器的描述,它综合以后的网表如图 3 所示——

```
module INCREMENT (A, CLOCK, Z);
  input [0:2] A;
  input CLOCK;
  output [0:2] Z;
  reg [0:2] Z;

  always @(posedge CLOCK)
    Z <= A + 1;
endmodule
```

注意到,图 3 中的三个触发器不是例化而是通过 HDL 的特定结构推断出来的。这种推断是根据一些推断法则(Inference rule)进行的,例如在这个例子中,当一个信号(变量)在时钟的边沿进行赋值(always 语句),那么这个信号(变量)可以推断为一个触发器。

1.2.3 行为级综合

行为级综合比 RTL 级综合层次更高,同时它描述电路也越抽象,在 RTL 级中,电路在每个时钟边沿的行为必须确切的描述出来,而行为级描述却不是这样,这里没有明确规定电



路的时钟周期，推断法则也不是用来推断寄存器。电路的行为可以描述成一个时序程序 (sequential program)，综合工具的任务就是根据指定的设计约束，找出哪些运算可以在哪个时钟周期内完成，需要在多个周期内用到的变量值需要通过寄存器寄存起来。

请看一个简单的行为综合的例子——

```
module SIMPLE_TREE (InData, OutData);
  input [0:3] InData;
  output [0:3] OutData;
  reg [0:3] OutData;
  reg [0:3] Tree;

  always @(InData)
  begin
    Tree = InData + 1;
    Tree = Tree + 1;
    OutData <= Tree + 1;
  end
endmodule
```

上面这个例子没有任何时钟的信息，现在假设一次加法操作(加法器)需要 5ns 的延时并且假设系统的时钟是 6ns，那么可以看出执行完上述操作需要 3 个周期的时间。另外，所有的三个加法语句可以通过重用一個加法器来实现，而且只需要一个叫做 Tree 的寄存器保存中间变量的值(不同时钟周期的变量值)。这种假设下的电路结构图如图 4 所示，控制器的时序关系如图 5 所示。

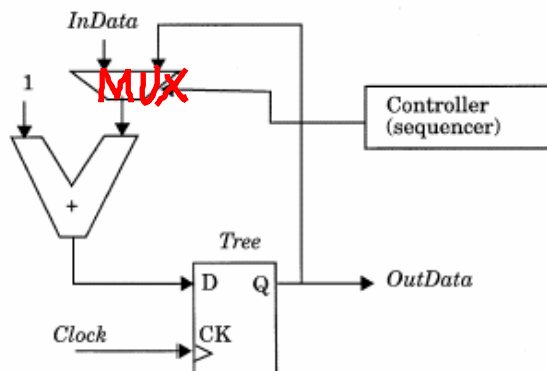


图 4 6ns 周期下的电路结构

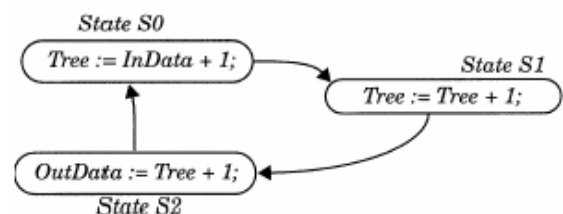


图 5 控制时序图

如果改变约束条件，假设时钟周期是 11ns，那么完成全部操作仅需 2 个周期，同时需要 2 个加法器，图 6 和图 7 分别是此时的电路结构图和控制器时序图。

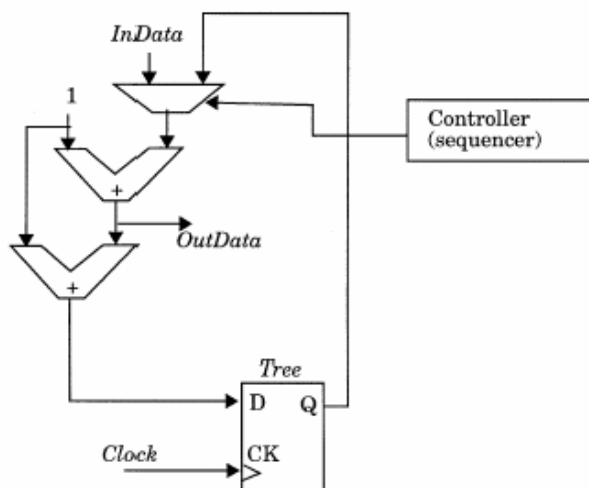


图 6 11ns 周期下的电路结构

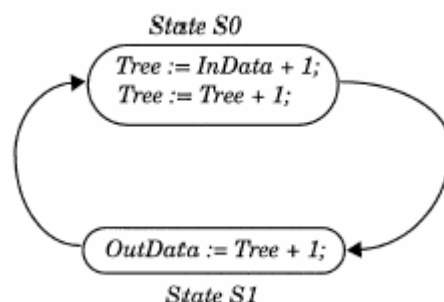


图 7 控制时序

1.2.4 Design Compiler 所处的位置

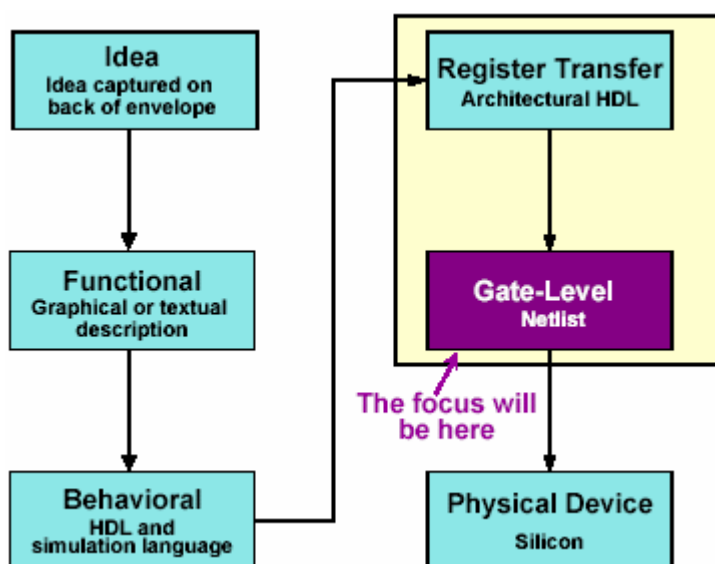


图 8 Design Compiler 所处位置

图 8 向我们展示了一个设计从最初的最抽象的概念阶段到最终的芯片阶段的转化过程，在这个过程中，Design Compiler 主要完成将设计的 RTL 级描述转化到门级网表的过程，比 RTL 更高的行为级的综合，将由 Synopsys 的另外一个工具——Behavior Compiler 来完成。在以下的章节中，我们主要围绕怎样将一个 RTL 级描述的设计转化为门级网表来进行讨论。

1.3 使用 Design Compiler 做综合的流程示意图

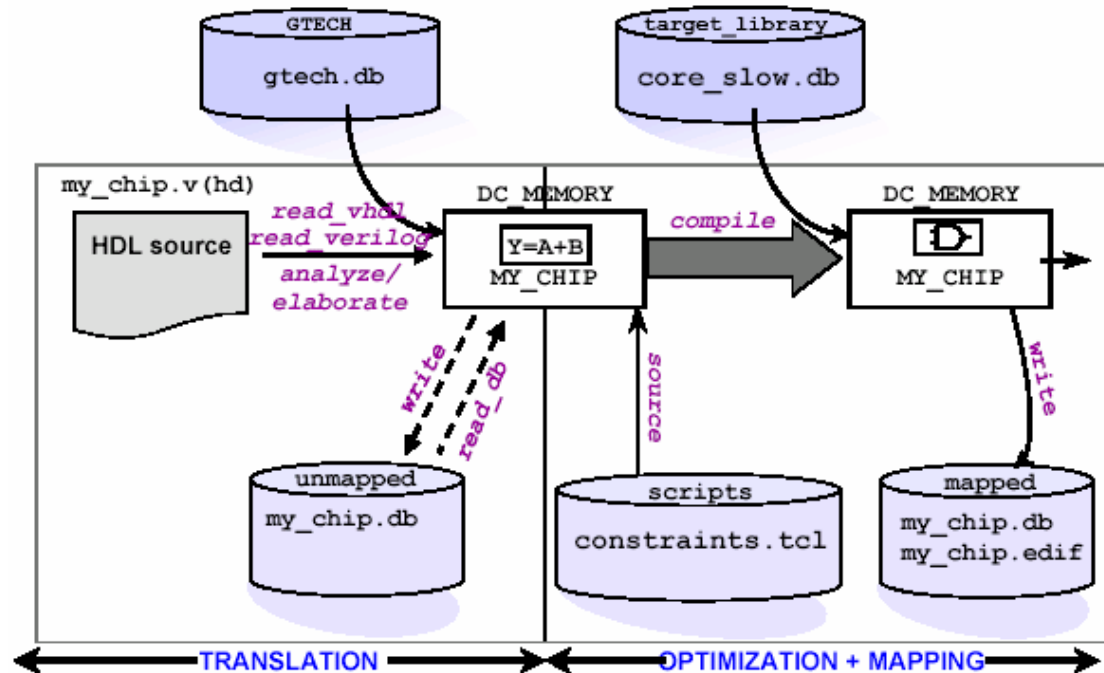


图 9 DC 综合示意图

与一般的综合过程相同，使用 DC 做综合也包含转换、优化和映射三个阶段。转换阶段综合工具将 HDL 语言描述的电路或未映射的电路用工艺独立的 RTL 级的逻辑来实现，对于 Synopsys 的综合工具 DC 来说，就是使用 gtech.db 库中的 RTL 级单元来组成一个中间的网表。优化与映射是综合工具对已有的中间网表进行分析，去掉其中的冗余单元，并对不满足限制条件(如 constraints.tcl)的路径进行优化，然后将优化之后的电路映射到由制造商提供的工艺库上(如 core_slow.db)。

1.4 超深亚微米给综合工具带来的挑战

当半导体工艺的最小特征尺寸小于 1um 时，称之为亚微米设计技术，当最小特征尺寸小于 0.5um 时称为深亚微米设计技术(DSM:Deep Sub Micrometer)，而当进一步小于 0.25um 时，则称为超深亚微米设计技术(VDSM:Very Deep Sub Micrometer)。当进入超深亚微米设计后，原有的综合工具受到了很大的挑战，其中一个主要表现是：连线的延时迅速上升。

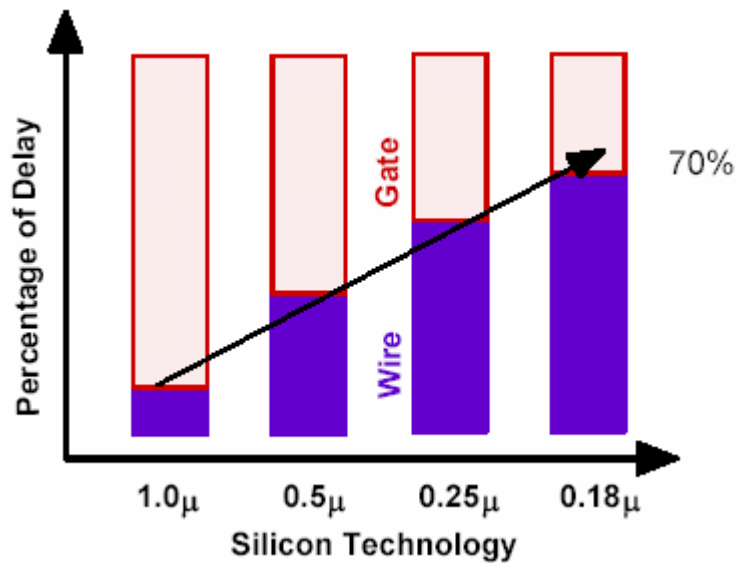


图 10 VDSM 对电路延时的影响

当特征尺寸大于 $0.5\mu\text{m}$ 时,电路的延时主要集中在门级单元的延时上,如果门级单元延时占系统延时的 70%以上,则前端综合后的电路延时与后端进行布局布线以后反标 (back-annotate)回来的电路延时相差不大。从图 10 可以看出,当特征尺寸进一步缩小时,单位连线上的延时以及互连线的总长度迅速上升,这两方面的因素都使得连线延时在系统总延时上所占的比重越来越大。通常在 $0.35\mu\text{m}$ 设计时,连线延时已经达到了总延时的 50%以上,于是版图发标的延时与综合出来得到延时相差会比较大,单靠一次综合已经不能准确估计电路的延时情况,此时需要经过前端后端工具不断叠代来达到比较真实的结果(见图 11)。在 $0.18\mu\text{m}$ 的时候,连线延时已经达到了 70%,这时就算增加叠代的次数也不一定能得到满意的结果,因而必须引入新的综合手段,保证优化叠代过程的收敛,这就是物理综合方法 (Physical Synthesis Flow)。

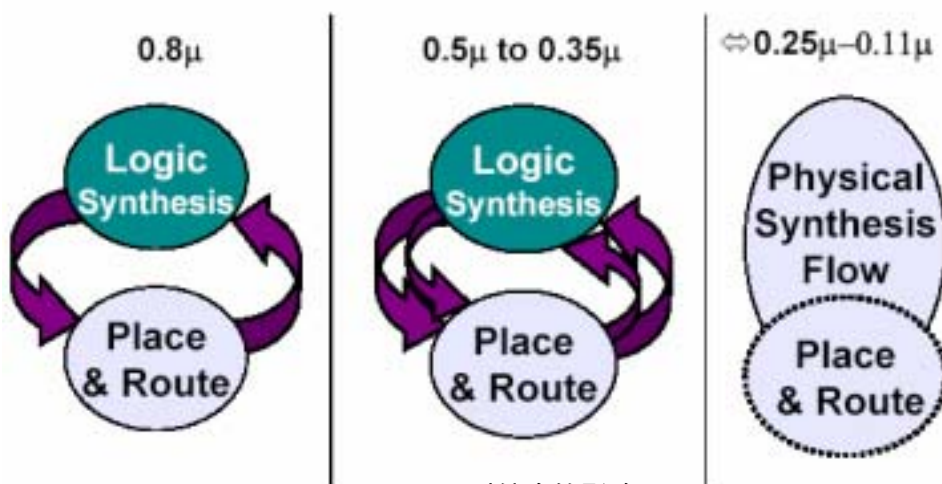


图 11 VDSM 对综合的影响

Physical Compiler 是 Synopsys 推出的新一代综合工具,它逐渐打破了前后端设计分离的设计方法而将它们结合在一起。它与 Design Compiler 的最大区别是综合出的门级网表除了反映电路的连接关系之外还有它在版图中具体位置的信息,有了单元的位置信息,综合工具就可以比较准确的估算出互连线的延时,从而减少叠代次数,提高设计效率。

§2. Verilog 语言结构到门级的映射

Verilog 编码效率的高低是综合后电路性能高低的决定性因素，两种不同风格的编码，即使它们所表达的逻辑功能一样，也会产生出大不一样的综合结果。就算综合工具运用的再好，也不能完全依赖它把一段编码很差的代码综合出一个像样的电路来。本节将通过大量的实例介绍综合时 Verilog 的各种语言结构(always、if、case、loop 等等)到门级的映射。这些语言结构是在编写 Verilog 代码的时候经常用到的一些基本结构，希望通过分析他们与门级网表之间的对应关系，大家能够对什么样的语句能生成什么样的具体电路有个初步的认识。

编写用于综合的 HDL 代码的**三个原则**——

- 编写代码的时候**注意代码综合后大概的硬件结构，不写不可综合的语句**
- 编写代码的时候**注意多用同步逻辑，并将异步和同步逻辑分开处理**
- 编写代码的时候**注意代码的抽象层次，多用 RTL 级的描述**

2.1 always 语句的综合

always 语句用来描述电路的过程行为(procedural behavior)，表示当事件列表中的状态发生变化时，执行语句体中的语句。下面是一个包含过程赋值的 always 语句的例子。

```
module EvenParity (A, B, C, D, Z);  
  input A, B, C, D;  
  output Z;  
  reg Z, Temp1, Temp2;  
  
  always @(A or B or C or D)  
  begin  
    Temp1 = A ^ B;  
    Temp2 = C ^ D;  
    Z = Temp1 ^ Temp2;  
    // Note that the temporaries are really not  
    // required. They are used here to illustrate the  
    // sequential behavior of the statements within  
    // the sequential block.  
  end  
  
endmodule
```

电路综合后的网表如下图所示——

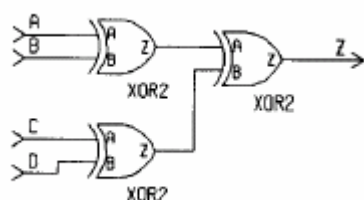


图 12 always 语句的综合结果

使用 always 语句描述组合电路要注意的是：在该语句中读入的所有变量都需要出现在事件列表中(对 Verilog 语言而言是指”@”符号之后的信号)，否则可能会得不到用户期望的结果。

2.2 If 语句的综合

if 语句用于描述受条件控制的电路，下面是一个例子——

```
module SimpleALU (Ctrl, A, B, Z);  
  input Ctrl;  
  input [0:1] A, B;  
  output [0:1] Z;  
  reg [0:1] Z;  
  
  always @ (Ctrl or A or B)  
    if (Ctrl)  
      Z = A & B;  
    else  
      Z = A | B;  
endmodule
```

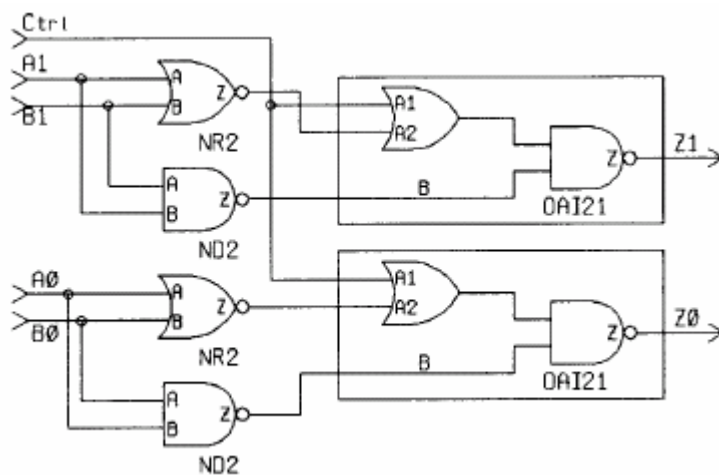


图 13 if 语句的条件选择操作

如果在使用 if 语句时，没有指出条件判断的所有可能情况，则会在电路中引入锁存器 (Latch)，如下例所示——

```
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

```

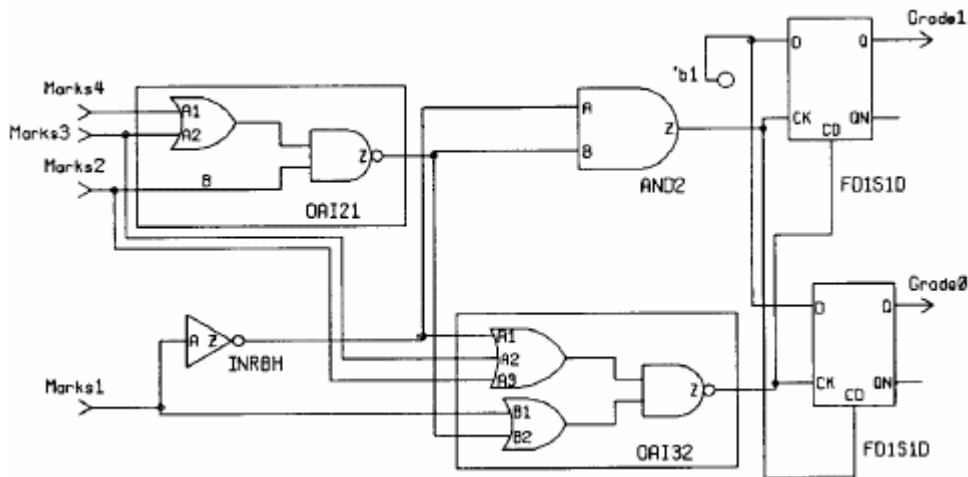


图 14 if 语句条件未全引入 Latch

上面的电路中，当条件 Marks 的值为小于 5 时，电路的输出 Grade 的值为 FAIL，如果 Marks 的值为 5 到 10 之间，那么 Grade 的值为 PASS，由于没有指定 Marks 的值大于 10 的时候 Grade 的值，于是产生的电路中引入了锁存器保存原来 Grade 的值。

由于锁存器和触发器两种时序单元共存的电路会增大测试的难度，因此，在综合的时候尽量只选用一种时序单元，为了不在电路中引入锁存器，可以在使用该语句时设置缺省的状态，即在判断条件之前先对输出赋值，或者使用 if...else if...else 的语句结构。

上面的例子经过改动后可以得到下面的例子——

```

module ComputeNoLatch (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;
    else
      Grade = EXCELLENT;
endmodule

```

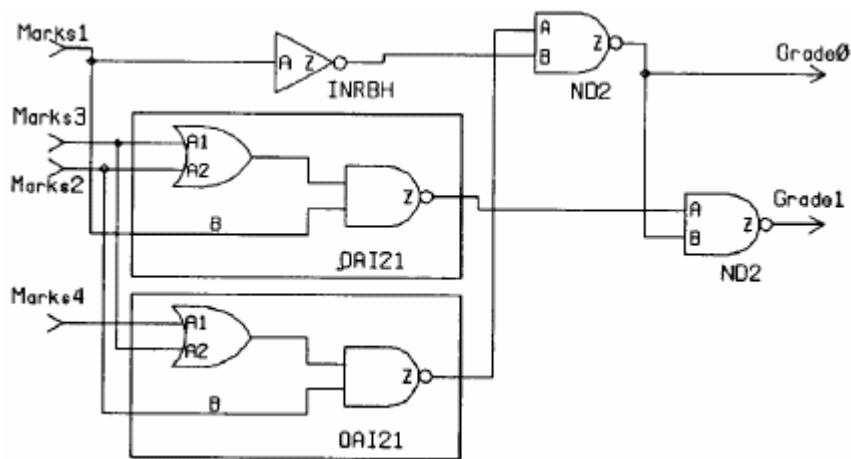


图 15 if 语句条件补全消除 Latch

2.3 case 语句的综合

2.3.1 条件互斥的 case 语句

对于条件互斥(mutually exclusive)的 case 语句来说，它不存在优先级的概念。

先看一个 case 语句的简单实例——

```

module ALU (Op, A, B, Z);
  input [1:2] Op;
  input [0:1] A, B;
  output [0:1] Z;
  reg [0:1] Z;

  parameter ADD = 'b00, SUB = 'b01, MUL = 'b10,
             DIV = 'b11;

  always @(Op or A or B)
    case (Op)
      ADD : Z = A + B;
      SUB : Z = A - B;
      MUL : Z = A * B;
      DIV : Z = A / B; // The A/B operation may not be
                       // supported by some synthesis tools.
    endcase
endmodule

```

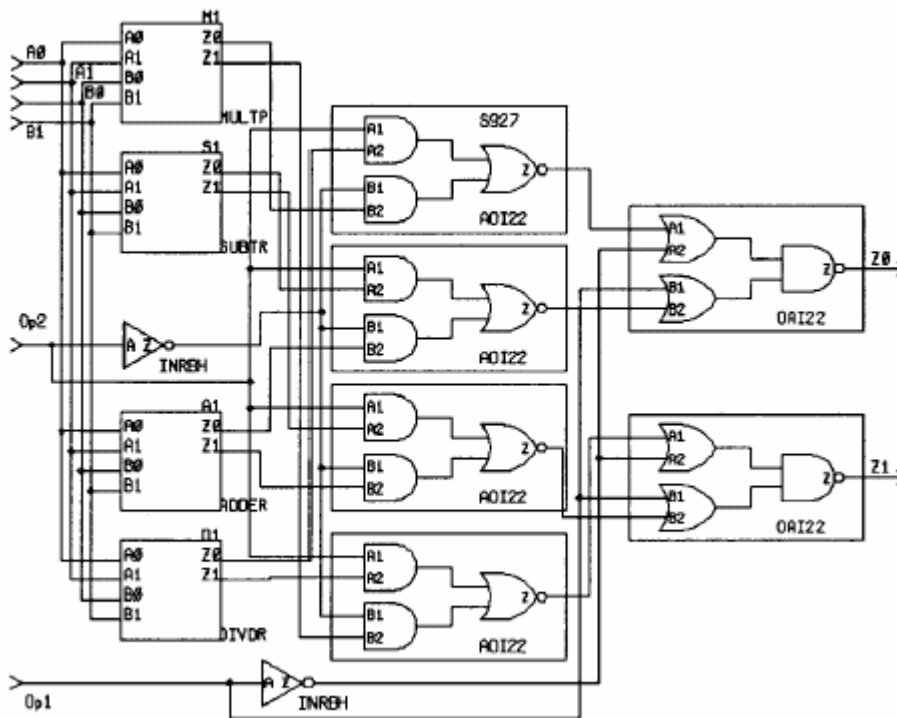


图 16 一个 2-bit 的 ALU

从条件互斥的 case 的行为来看，它有点像 if...else if...else 这种结构的 if 语句，也就是说 case 表达式(Op)先同 case 的第一个 case 项(ADD) 如果不匹配的话再同第二个(SUB)比较，依次类推。与上例等价的 if 语句如下所示——

```

if (Op == ADD)
    Z = A + B;
else if (Op == SUB)
    Z = A - B;
else if (Op == MUL)
    Z = A * B;
else if (Op == DIV)
    Z = A / B;

```

2.3.2 Casex 语句

在 casex 语句中，case 项中的 x 或者 z 的值表示不关心(don't-care)。从综合的角度看，这些值不能作为 case 表达式的一部分。

下面是一个用 casex 语句描述优先级编码器的例子——

```

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

    always @ (Toggle)
        case (Toggle)
            3'bxx1 : NextToggle = 3'b010;    001,011,101,111
            3'bx1x : NextToggle = 3'b110;    010,110,
            3'b1xx : NextToggle = 3'b001;    100
            default : NextToggle = 3'b000;    000
        endcase
    endmodule

```

图 17 用 casex 描述的优先级编码电路

```
if (Toggle[0] == 'b1')
    NextToggle = 3'b010;
else if (Toggle[1] == 'b1')
    NextToggle = 3'b110;
else if (Toggle[2] == 'b1')
    NextToggle = 3'b001;
else
    NextToggle = 3'b000;
```

可见 casex 所描述的电路是具有优先级的。如果我们将上例中的 casex 用 case 语句代替 (同时将 case 项中的 x 用 0 代替), 则综合出来的电路将不具有优先级。

2.3.3 隐含 Latch 的 case 语句

与 if 语句类似，如果没有指出 case 项的所有可能情况，则会在电路中引入锁存器 (Latch)，下面是一个例子——

```

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

endmodule

```

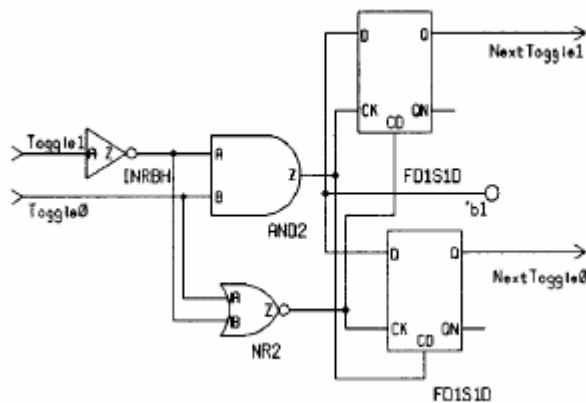


图 18 case 语句条件不全引入 Latch

上例中的 Zip 信号没有在所有可能情况下赋值，故在 Toggle=0 或 3 时，NextToggle 保持原有的值，从而产生 Latch。避免产生 latch 的一个方法是给被赋值信号(NextToggle)赋初值。一种方法是加在 case 的最后加入一种 default 的情况（如下）

```

always @ (Toggle)
  case (Toggle)
    2'b01 : NextToggle = 2'b10;
    2'b10 : NextToggle = 2'b01;
    default : NextToggle = 2'b01; // Dummy assignment.
  endcase

```

或者在 case 语句之前就对被赋值信号加上一个初值（如下）

```

always @ (Toggle)
  begin
    NextToggle = 2'b01; // Default assignment.
    case (Toggle)
      2'b01 : NextToggle = 2'b10;
      2'b10 : NextToggle = 2'b01;
    endcase
  end

```

2.3.4 Full_case

从 2.3.3 的例子我们知道，如果条件不全，case 综合之后会产生 latch。如果设计者知道除了列出的一些 case 项之外不会再有其他的条件出现，也就是说 Toggle 的值除了 2'b01 和 2'b10 之外不会有其他的值，而且又不想让电路产生 latch，那么他就需要把这种情况告诉综合工具，这里就可以通过一条综合指令(synthesis directive)——`synopsys full_case` 来传达。综合指令是 HDL 语言中的一类特别的代码，它负责向综合工具传递额外的信息；由于综合指令是以注释的形式存在于 HDL 代码中的，它对 Verilog 语言本身没有其他影响。

加入综合指令之后，电路如下所示——

```
module NextStateLogicFullCase (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  
endmodule
```

`synopsys full_case`

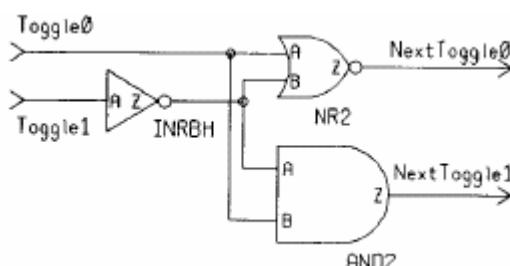


图 19 加入 full_case 后的综合结果

可见在加入 full_case 综合指令后，综合的结果不存在 Latch。但是要注意的是：

1. 加入综合指令会使代码的结果依赖于所用的综合工具，从而降低代码的可移植性。
2. 加入综合指令后产生的电路网表会和当初的 Verilog 建模有出入，导致验证的复杂

2.3.5 Parallel_case

从 2.3.3 节可以看出，case 语句是具有优先级的，与 if...else if...else 语句相当。那么假如我们知道 case 项是互斥的该怎么办呢？(在互斥的情况下，case 将平行的检查 case 项中所有可能的情况，而不是先检查第一个然后第二个.....)。这时，我们需要将互斥的信息传达给综合工具，这就是 parallel_case 的综合指令。当加上这条指令后，Design Compiler 能够

理解 case 项是互斥的，这样就不会产生带优先级的电路，而是平行的译码结构。

加入综合指令后的电路如下所示——

```
module ParallelCase (NextToggle, Toggle);  
  input [2:0] Toggle;  
  output [2:0] NextToggle;  
  reg [2:0] NextToggle;  
  
  always @ (Toggle)  
    casex (Toggle) // synopsys parallel_case  
      3'bxx1 : NextToggle = 3'b010;  
      3'bx1x : NextToggle = 3'b110;  
      3'b1xx : NextToggle = 3'b001;  
      default : NextToggle = 3'b000;  
    endcase  
endmodule
```

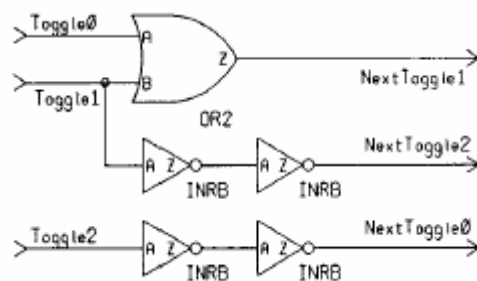


图 20 加入 parallel_case 后的综合结果

与上面的 casex 语句对应的 if 语句如下——

```
if (Toggle[0] == 'b1)  
  NextToggle = 3'b010;  
  
if (Toggle[1] == 'b1)  
  NextToggle = 3'b110;  
  
if (Toggle[2] == 'b1)  
  NextToggle = 3'b001;  
  
if ((Toggle[0] != 'b1) &&  
    (Toggle[1] != 'b1) &&  
    (Toggle[2] != 'b1))  
  NextToggle = 3'b000;
```

由于 parallel_case 同 full_case 一样都是综合指令语句，在应用 parallel_case 的时候也要考虑到可移植性和提高验证复杂度的问题。

2.3.6 case 项不是常数的 case 语句

前面讨论的 case 项都是常数，实际上我们也会遇到 case 项不是常数的情况，如下例——

```
module PriorityEncoder (Pbus, Address);  
  input [0:3] Pbus;  
  output [0:1] Address;  
  reg [0:1] Address;  
  
  always @ (Pbus)  
  case (1'b1) / synopsys full_case  
    Pbus[0] : Address = 2'b00;  
    Pbus[1] : Address = 2'b01;  
    Pbus[2] : Address = 2'b10;  
    Pbus[3] : Address = 2'b11;  
  endcase  
endmodule
```

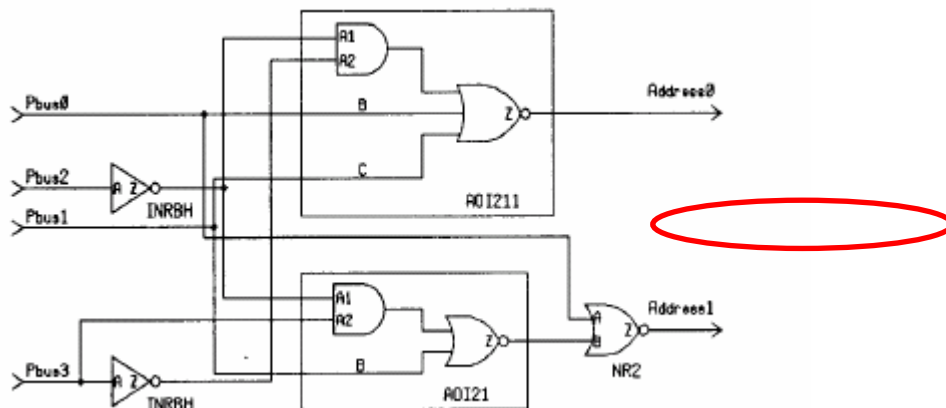


图 21 case 项不是常数的优先级译码

这里加入 full_case 指令是相当必要的，否则综合后会引入 latch(Pbus 全不为 1 的情况没有考虑)。另外，也可以通过赋初值的办法避免 latch 的产生。

值得注意的是，case 项不是常数与 case 项是常数不同，它综合后的电路是带优先级的。

2.4 loop 语句的综合

在 Verilog 语法中，一共有四种 loop 语句——while-loop, for-loop, forever-loop 以及 repeat-loop。其中 for-loop 使用的最多，也是一种典型的可以被综合的 loop 语句。For-loop 语句综合的基本原则就是将里面的所有语句进行展开，下面举一个例子——

```
module DeMultiplexer (Address, Line);  
  input [1:0] Address;
```

```

output [3:0] Line;
reg [3:0] Line;

integer J;

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

```

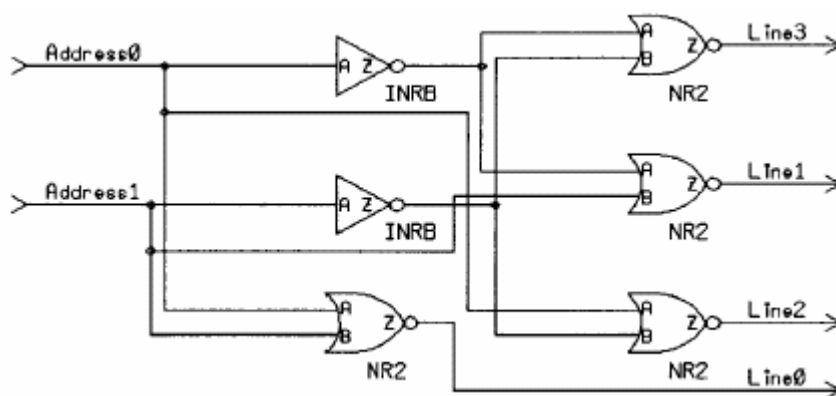


图 22 loop 语句的综合

将 loop 语句展开之后，可以得到下面的 if 语句——

```

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;

```

使用 loop 语句值得注意的地方是：不要在循环体内加入一些延时或面积较大的单元（例如加法器）。由于 loop 语句综合时需要展开循环体，所以相当于把循环体内的单元复制出来，循环多少次则复制多少次，这样势必会造成综合后的网表面积和延时很大，影响性能。

2.5 触发器的综合

触发器是组成时序电路的一个基本元件，也是 Design Compiler 作静态时序分析的要素之一，当一个信号（变量）在通过 always 语句在时钟边沿（上升沿或下降沿）赋值时，触发器就可以推断出来。

先看下面的一个例子——

```

module Incrementor (ClockA, Counter);
    parameter COUNTER_SIZE = 2;
    input ClockA;
    output [COUNTER_SIZE-1:0] Counter;
    reg [COUNTER_SIZE-1:0] Counter;

    always @ (posedge ClockA)
        Counter <= Counter + 1;
endmodule

```

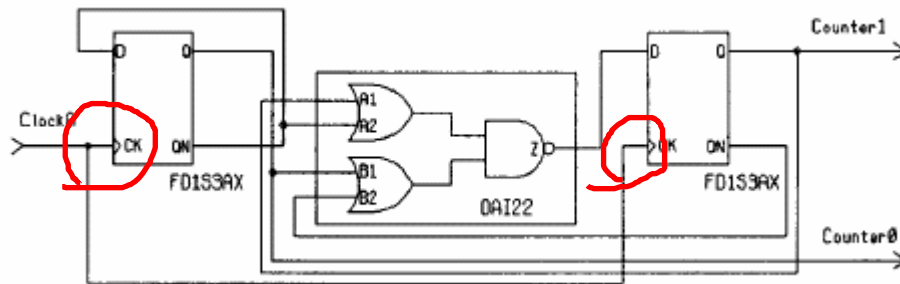


图 23 2-bit 加法器的综合

上面的 always 语句说明，每当 ClockA 出现一次上升延跳变，信号 Counter 就加 1。由于 Counter 是受时钟上升沿控制的，那么它综合以后会出现上升沿的触发器。

描述时序电路要注意一点：如果一个信号既要在 always 内部赋值也要在 always 语句之外赋值，那在 always 内部赋值时需要用非阻塞赋值语句，如上例中的 Counter<=Counter + 1。这样可以准确的反映时序电路的行为。

在读入 Verilog 文件的时候，Design Compiler 能够分析出代码中的时序元件(触发器和锁存器)，并将结果报告出来。

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-Flop	1	-	-	Y	N	N	N	N

图 24 DC 报告时序元件

如图 21 所示的 Q_reg，Design Compiler 推断出是一个一位宽的异步复位 (AS) 的触发器。

2.6 算术电路的综合

DC 在综合遇到运算符的时候，会在 DesignWare 中选取合适的逻辑电路来实现该运算符。DesignWare 是集成在 DC 综合环境中的可重用电路的集合，主要包括 ‘+’ ‘-’ ‘x’ 等算术运算符和 ‘<’ ‘>’，‘<=’ ‘>=’ 等逻辑运算符。针对同一种运算符，DesignWare 可能提供不同的算法，具体选择那一种是由给定的限制条件决定的。

DesignWare 分为 DesignWare Basic 与 DesignWare Foundation，DesignWare Basic 提供基本的电路，DesignWare Foundation 提供性能较高的电路结构。

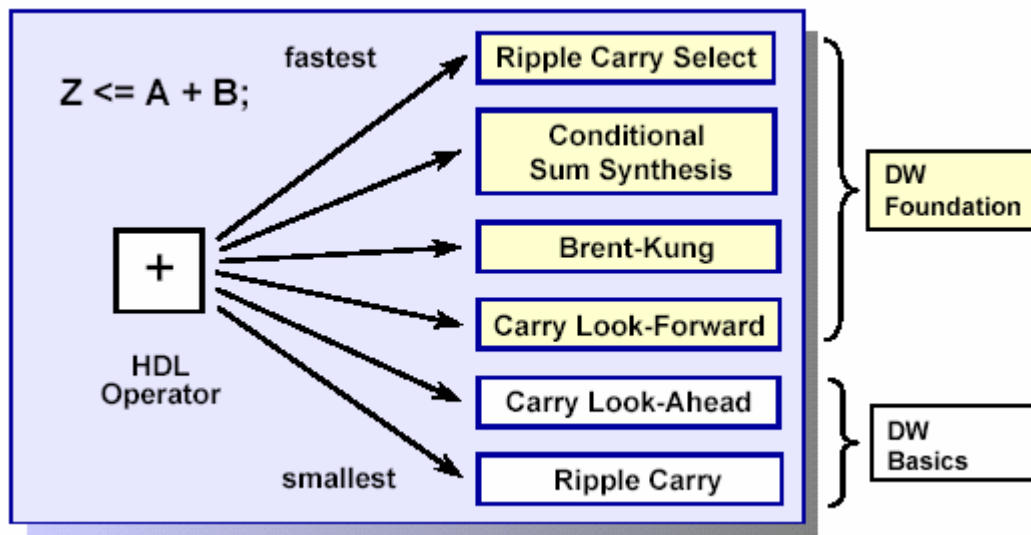


图 25 DW Foundation 与 DW Basic

从上图的加法器可以看出，DW Foundation 除了具有 Carry Look-Ahead 和 Ripple Carry 两种结构的加法器外，还有另外四种结构，并且速度更快。

如果要 Foundation 的 DesignWare，除了需要有 DesignWare Foundation 的 Licesen 之外还需要在综合的时候设置 synthetic_library。

```
set synthetic_library {dw_foundation.sldb}
```

```
lappend link_library $synthetic_library
```

在 verilog 语言中，一个 `reg` 类型的数据是被解释成无符号数，`integer` 类型的数据是被解释成二进制补码的有符号数，而且最右边是有符号数的最低位。

进行算术运算时，它也有各个运算符的优先级，运算按照由左至右进行，如下面的式子

```
SUM <= A*B + C*D + E + F + G
```

它综合出来会是下面的样子——

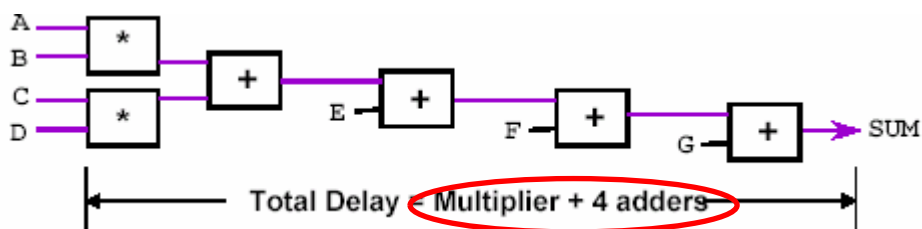


图 26 不好的运算电路结构

显然这种结构的延时是很大的，我们可以通过交换运算次序和加入括号形成优化的结构

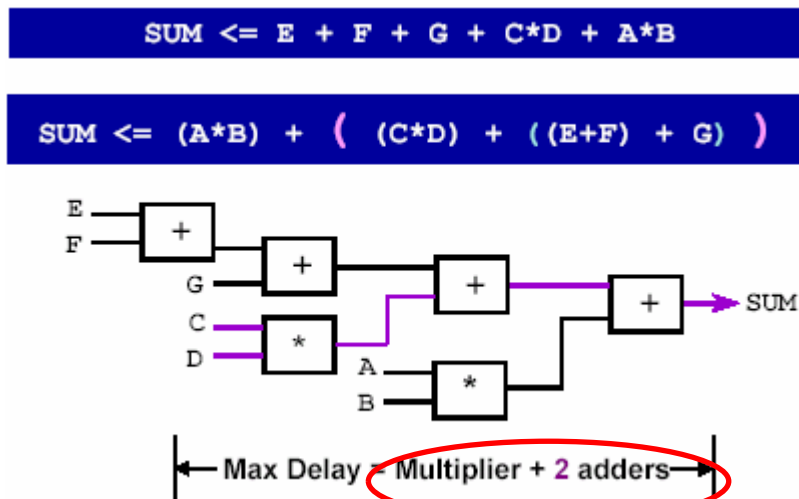


图 27 优化后的电路结构

本节详细介绍了 Verilog 的各种语句和运算符的综合电路，本节综合没有施加任何的限制条件，在下一节中，将会详细给出 Design Compiler 工具的详细介绍和具体的命令。

§3. 使用 Design Compiler 进行综合

在接下来的几章里，我们将具体介绍使用 Synopsys 公司的 Design Compiler 作综合的过程，整个过程大致可以分为下面几个部分——

- 预综合过程(Pre-synthesis Processes)
- 施加设计约束(Constraining the Design)
- 设计综合(Synthesizing the Design)
- 后综合过程(Post-synthesis Process)

下面我们这个步骤分别对各个过程作一个详细的介绍。

3.1 预综合过程

预综合过程是指在综合过程之前的一些为综合作准备的步骤，包括 Design Compiler 的启动、设置各种库文件、创建启动脚本文件、读入设计文件、DC 中的设计对象、各种模块的划分以及 Verilog 的编码等等。其中 Verilog 编码在第 2 章中已经讨论过。

3.1.1 Design Compiler 的启动

对于 2000.11 版的 Design Compiler，用户可以通过四种方式启动 Design Compiler，他们是——dc_shell 命令行方式、dc_shell-t 命令行方式、design_analyzer 图形方式和 design_vision 图形方式。其中后面两种图形方式是分别建立在前面两种命令行方式的基础上的。

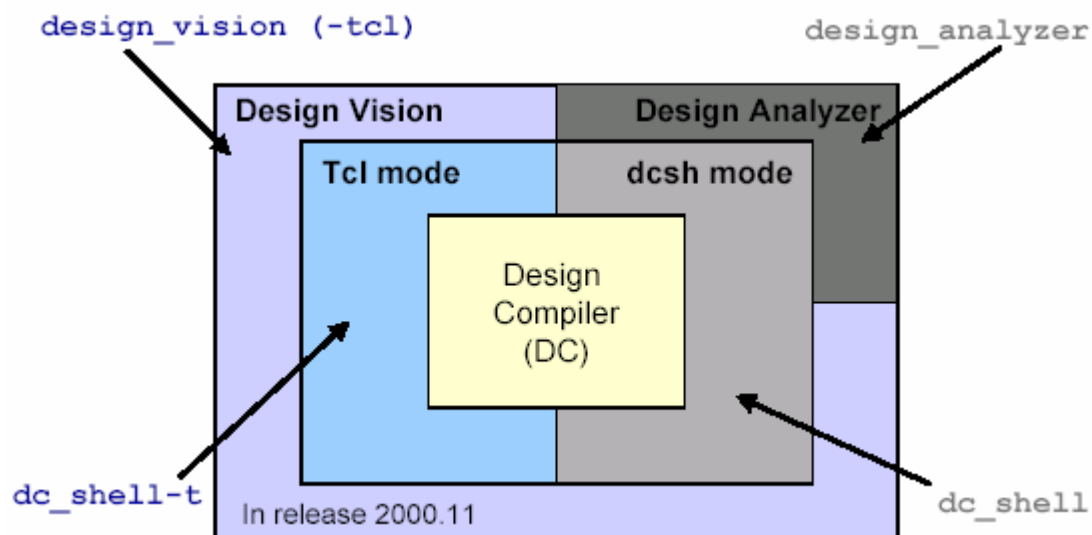


图 28 DC 的四种界面

1. dc_shell 命令行方式

该方式以文本界面运行 Design Compiler。在 shell 提示符下直接输入“dc_shell”就可以进入这种方式。也可以在启动 dc_shell 的时候直接调用 dcsh 的脚本来执行(dc_shell -f script)。目前这种方式用的已经不是很普遍。

2. dc_shell-t 命令行方式

该方式是以 TCL (Tool Command Language 后面章节将有介绍) 为基础的，在该脚本语言上扩展了实现 Design Compiler 的命令。用户可以在 shell 提示符下输入“dc_shell-t”来运行该方式。该方式的运行环境也是文本界面。也可以在启动 dc_shell-t 的时候直接调用 tcl 的脚本来执行(dc_shell-t -f script)。TCL 命令行方式是现在推荐使用的命令行方式，她相对 shell 方式功能更强大，并且在 Synopsys 的其他工具中也得到普遍使用。

3. design_analyzer 图形界面方式

Design Analyzer 使用图形界面，如菜单、对话框等来实现 Design Compiler 的功能，并提供图形方式的显示电路。用户可以在 shell 提示符下打“design_analyzer”来运行该方式。Design_analyzer 图形方式是今后要经常用到的图形界面方式。由于它所对应的是 dc_shell 的命令行方式，所以我们不能在 design_analyzer 里运行 tcl 命令。另外需要注意的是：Design analyzer 的工作模式不是用于编辑电路图的，它只能用于显示 HDL 语言描述电路的电路图。

4. design_vision 图形界面方式

Design_vision 是与 tcl 对应的图形方式，用户可以在 shell 提示符下打“design_vision”来运行该方式。由于它是在 Windows NT 下开发的，在工作站环境下不太普及，今后的课程中将不会用到。

不论 dcsh 模式还是 tcl 模式都提供了类似于 unix 的 shell 脚本的功能，包括变量赋值、控制流命令、条件判断等等，但是 dcsh 模式的语法规则不同于 tcl 的语法规则，因此，使用 dcsh 书写的脚本不能直接用于 TCL 工作模式；使用 TCL 书写的脚本也不能直接用于 dcsh

工作模式。

Design Analyzer 在启动时自动在启动目录下面创建两个日志文件：command.log 和 view_command.log，用于记录用户在使用 Design Compiler 时所执行的命令以及设置的参数，在运行过程中同时还产生 filenames.log 的文件，用于记录 design compiler 访问过的目录，包括库、源文件等，filenames.log 文件在退出 design compiler 时会被自动删除。启动 dc_shell 时则只产生 command.log 的日志文件。

3.1.2 库文件的设置

在 Design Compiler 的运行过程中需要用到几种库文件，他们是工艺库、链接库、符号库以及综合库，下面对他们一一说明。

1. 工艺库(target_library)

工艺库是综合后电路网表要最终映射到的库，读入的 HDL 代码首先由 synopsys 自带的 GTECH 库转换成 Design Compiler 内部交换的格式，然后经过映射到工艺库和优化生成门级网表。工艺库他是由 Foundry 提供的，一般是 db 的格式。这种格式是 DC 认识的一种内部文件格式，不能由文本方式打开。db 格式可以由文本格式的 lib 转化过来，他们包含的信息是一致的。下面是一个 lib 的工艺库例子——

Example of a cell description in .lib Format

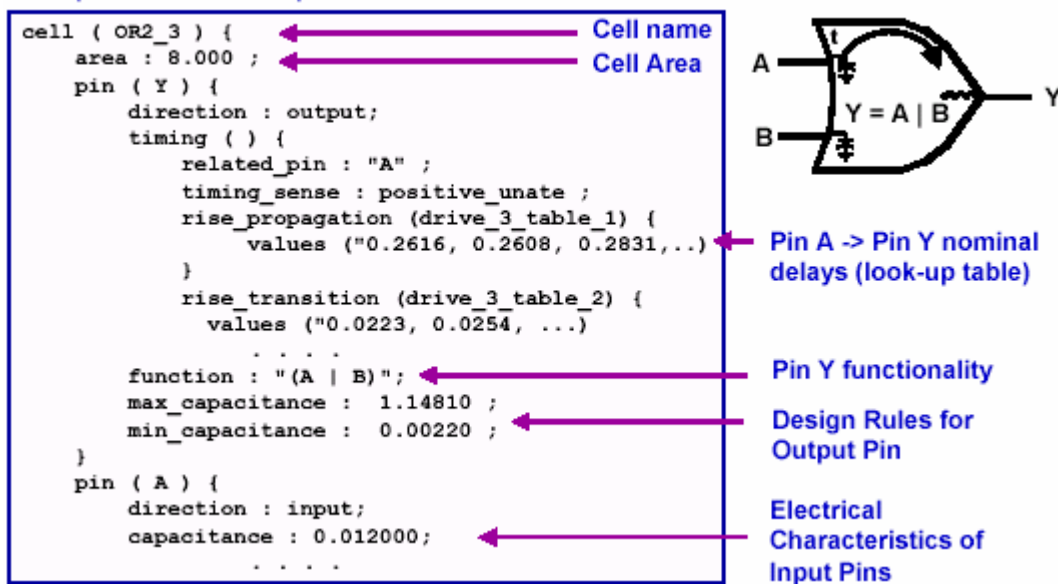


图 29 工艺库文件(.lib)

从图中可以看出，工艺库中包含了各个门级单元的行为、引脚、面积以及时序信息（有的工艺库还有功耗方面的参数），DC 在综合时就是根据 target_library 中给出的单元电路的延迟信息来计算路径的延迟。并根据各个单元延时、面积和驱动能力的不同选择合适的单元来优化电路。

在 tcl 模式下，我们可以根据下面的命令指定工艺库——

```
set target_library my_tech.db
```

2. 链接库(link_library)

link_library 设置模块或者单元电路的引用，对于所有 DC 可能用到的库，我们都需要在 link_library 中指定，其中也包括要用到的 IP。

值得注意的一点是：在 link_library 的设置中必须包含 '*'，表示 DC 在引用实例化模块或者单元电路时首先搜索已经调进 DC memory 的模块和单元电路，如果在 link library 中不包含 '*'，DC 就不会使用 DC memory 中已有的模块，因此，会出现无法匹配的模块或单元电路的警告信息(unresolved design reference)。

另外，设置 link_library 的时候要注意设置 search_path，请看下面这个例子——

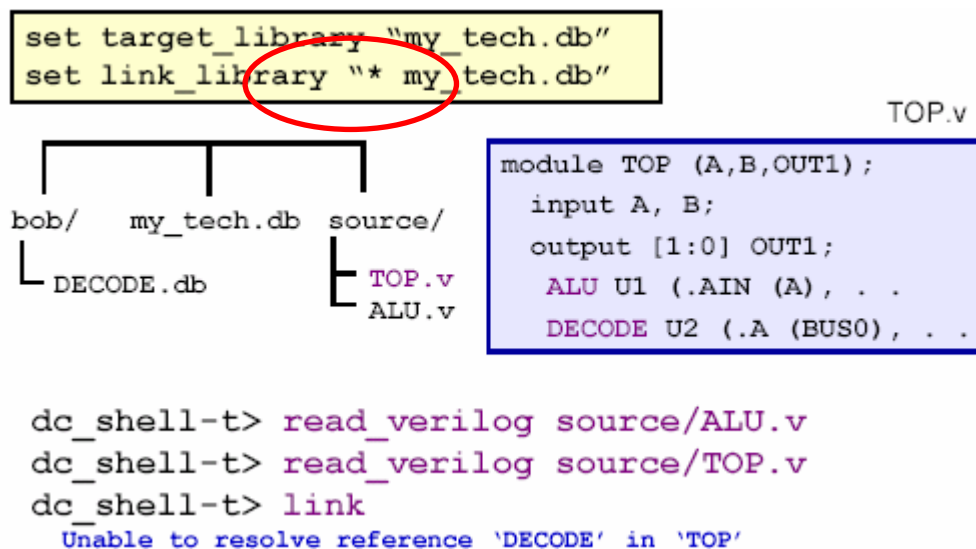


图 30 link_library 的使用(1)

图中设置了 link_library，但是 DC 在 link 的时候却报错，找不到 TOP 中引用的 DECODE 模块，这说明 link_library 默认是在运行 DC 的目录下寻找相关引用。要使上例的 DECODE 能被找到，需要设置 search_path，如下图所示——

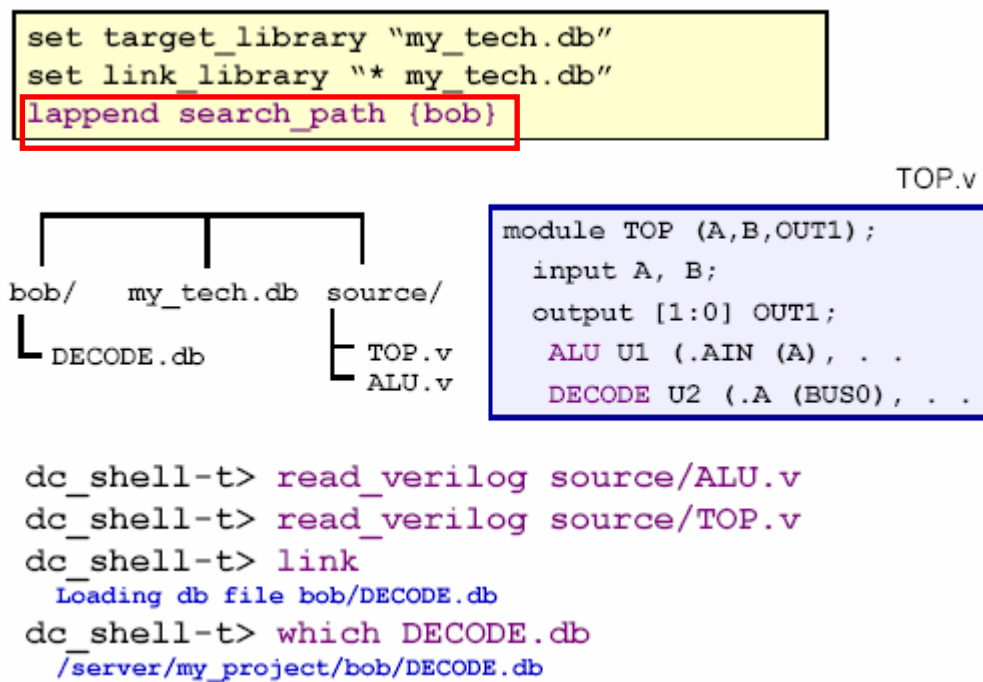


图 31 link_library 的使用(2)

3. 符号库 (symbol_library)

symbol_library 是定义了单元电路显示的 Schematic 的库。用户如果想启动 design_analyzer 或 design_vision 来查看、分析电路时需要设置 symbol_library。符号库的后缀是.sdb，加入没有设置，DC 会用默认的符号库取代。

设置符号库的命令是：set symbol_library

4. 综合库(synthetic_library)

在初始化 DC 的时候，不需要设置标准的 DesignWare 库 standard.sldb 用于实现 Verilog 描述的运算符，对于扩展的 DesignWare，需要在 synthetic_library 中设置，同时需要在 link_library 中设置相应的库以使得在链接的时候 DC 可以搜索到相应运算符的实现。

3.1.3 设置启动文件.synopsys_dc.setup

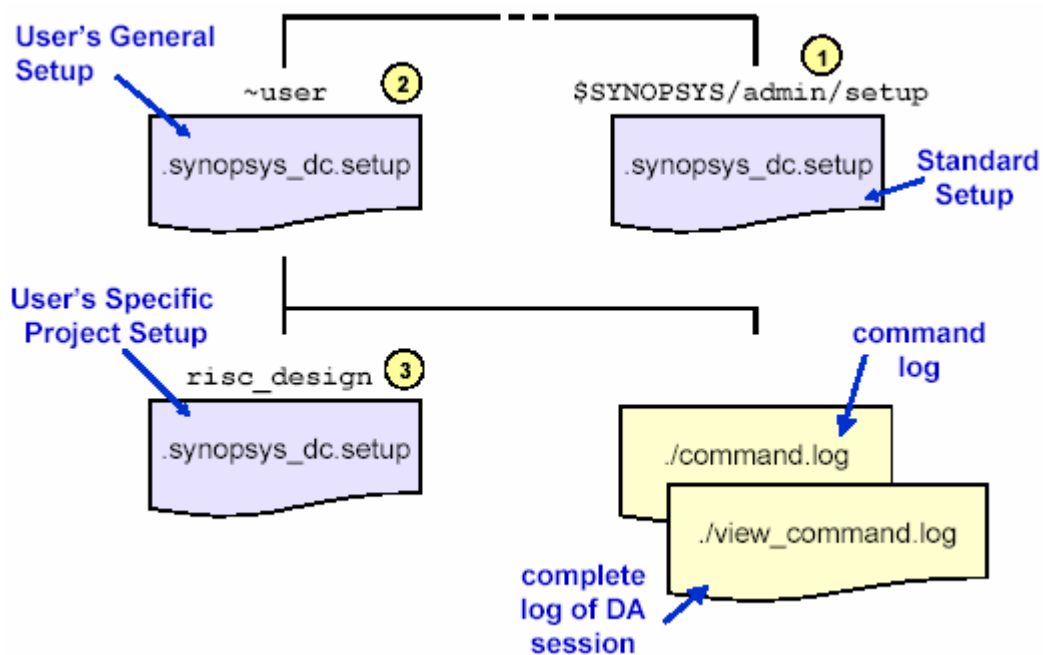


图 32 .synopsys_dc.setup

启动文件顾名思义，就是 DC 在启动的时候首先读入的文件，DC 在启动的时候，会自动在三个目录下搜索该文件（如上图所示），对 DC 的工作环境进行初始化：

1. `$SYNOPSYS/admin/setup` 目录下，DC 安装的标准初始化文件。
2. 当前用户的 `$HOME` 目录下，一般用于设置一些用户本人使用的变量以及一些个性化设置。
3. DC 启动所在的目录下，一般用于与所在设计相关的设置。

其中后面的 `setup` 文件可以覆盖前面文件中的设置。该文件主要包括库的设置、工作路径的设置以及一些常用命令别名的设置等等。

由于 `dcshell` 的启动脚本和 `tcl` 的脚本语法不一致，所以如果只有一种方式的启动脚本，那么运行另一种方式的时候会报错。因此，DC 的启动脚本有一种兼容两种方式的格式。下面是这种脚本的举例——

```
#
set target_library {core_slow.db}
set link_library {* core_slow.db}

set symbol_library {core.sdb}
set search_path "$search_path ./unmapped"

alias h history
alias rc "report_constraint -all_violators"
```

它区别与其他启动脚本的特征是**第一行有一个“#”**，说明它是 dcshell 的一个子集，同时兼容两种方式。文件的第一段设置工艺库和链接库，第二段设置符号库和搜索路径，第三段设置 DC 命令的别名，这一点与 Shell 相似。

3.1.4 读入设计文件

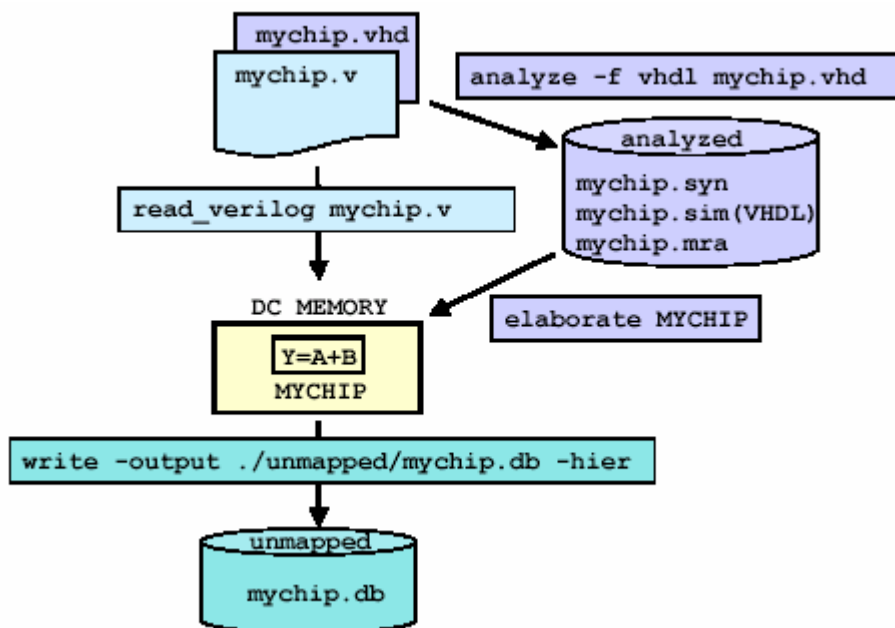


图 33 读入设计文件

Design Compiler 支持多种硬件描述的格式，.db、.v、.vhd、等等，对于 dcsh 工作模式来说，读取不同的文件格式只需要带上不同的参数，对于 TCL 的工作模式来说，读取不同的文件格式需要使用不同的命令。两种工作模式的读取命令的基本格式如下：

read -format verilog[db、vhdl etc.] file	//dcsh 的工作模式
read_db file.db	//TCL 工作模式读取 DB 格式
read_verilog file.v	//TCL 工作模式读取 verilog 格式

read_vhdl file.vhd

//TCL 工作模式读取 VHDL 格式

Design Compiler 可以读取设计流程中任何一种数据格式，如行为级的描述、RTL 级的描述、门级网表等等，不过由于不同的数据格式使得 Design Compiler 综合的起点不同，即使实现相同的功能，也可能产生不同的结果。

读取源程序的另外一种方式是配合使用 analyze 命令和 elaborate 命令：analyze 是分析 HDL 的源程序并将分析产生的中间文件存于 work(用户也可以自己指定)的目录下；elaborate 则在产生的中间文件中生成 verilog 的模块或者 VHDL 的实体，缺省情况下，elaborate 读取的是 work 目录中的文件。

当读取完所要综合的模块之后，需要使用 link 命令将读到 Design Compiler 存储区中的模块或实体连接起来，如果在使用 link 命令之后，出现 unresolved design reference 的警告信息，需要重新读取该模块，或者在 synopsys_dc.setup 文件中添加 link_library，告诉 DC 到库中去找这些模块，同时还要注意 search_path 中的路径是否指向该模块或单元电路所在的目录。

3.1.5 设计对象

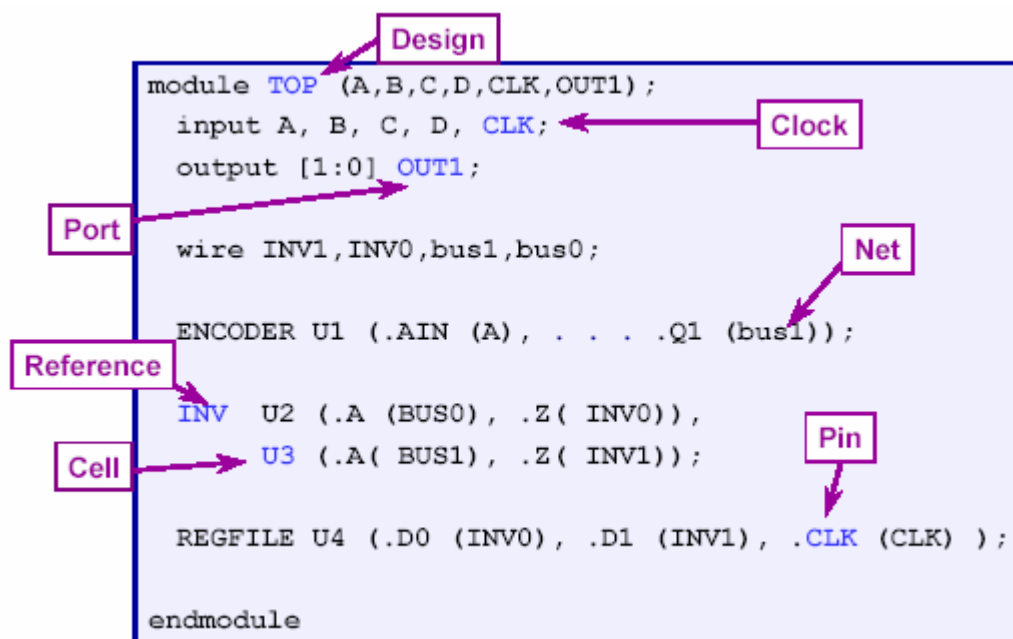
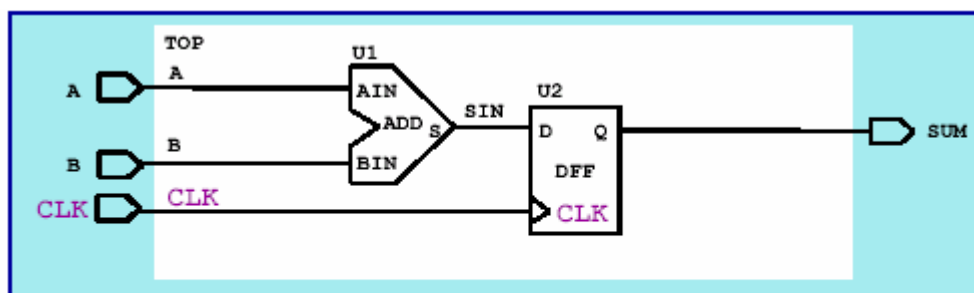


图 34 DC 中的设计对象

上图是一个 Verilog 描述的设计实例，里面包含了我们所要讨论的几种设计对象。这些对象也是今后 DC 命令的操作对象。Verilog 描述的各个模块可以称之为设计(Design)，里面包含时钟(Clock)，他的输入输出称为端口(Port)，模块中的互连线是线网(Net)，内部引用的元件称为引用(Reference)，引用的实例称为单元(Cell)，引用单元的内部端口是管脚(Pin)。元件和实例的区别

其中值得注意的是 DC 识别 Clock 不是通过 HDL 的书面表达，而是要通过设计者施加一定的约束来区分的，具体内容后续章节会讨论。

如果各个设计对象互相重名怎么办？



```
set_load 5 CLK
```

图 35 对象的重名

在上图中，一个设计的端口，连线以及内部一个管脚都有一个相同的名字，假如要对名叫“CLK”的线网设置一个为 5 的负载，那应该怎样表示呢？这里，我们需要借助 DCTCL 的一个特殊的数据类型集合（collection）。

```
dc_shell-t> set_load 5 [get_nets CLK]
```

其中的方括号里面表示在所有的线网中搜索名叫 CLK 的线网，将它的负载值设置为 5。get 命令返回对象的集合，如果这个对象没有找到，则返回为空集合。

集合是 DCTCL 的特殊数据类型，它不同于 TCL 语言中的一种标准数据类型列表(list)。列表类似与 C 语言中的字符串。我们可以通过下面的方式定义一个列表——

```
dc_shell-t> set mylist {el1 el2 el3}
Information: Defining new variable 'mylist'
el1 el2 el3
```

集合相当于 C 语言的指针，这个指针指向一个数据结构。例如上面指向的是一个名为 CLK 的线网数据类型，这个数据类型中除了包含负载这个属性之外，还有电阻值、扇出(fanout)等属性，而这些是在列表中不存在的。

除了前面的 get_nets 外，还有下面的一些命令可以搜索设计对象，这里列出了 TCL 和 dcshell 两种语法，便于对比——

可以在 dc_shell-t 环境下输入“help get*”列出所有以 get 打头的 DC 命令

Tcl mode	dcsh mode
get_cells *U*	find(cell, *U*)
get_nets *	find(net, "")
get_ports CLK	find(port, CLK)
get_clocks CLK	find(clock, CLK)
all_inputs	all_inputs()
all_outputs	all_outputs()

图 36 搜索对象

3.1.6 设计划分

把一个复杂的设计分割成几个相对简单的部分，称为设计划分(Design Partition)。这种方法，也可以称为“分而治之”(Divide and conquer)的方法，在平常的电路设计中这是一种普遍使用的方法，一般我们在编写 HDL 代码之前都需要对所描述的系统作一个系统划分，根据功能或者其他的原则将一个系统层次化的分成若干个子模块，这些子模块下面再进一步细分。这是一种设计划分，模块(module)就是一个划分的单位。在运用 DC 作综合的过程中，默认的情况下各个模块的层次关系是保留着的，保留着的层次关系会对 DC 综合造成一定的影响，比如在优化的过程中，各个子模块的管脚必须保留，这势必影响到子模块边界的优化效果。下面我们将详细介绍在作设计划分的过程中要注意的几点原则，并根据每个原则举一个实例说明。

原则一. 不要让一个组合电路穿越过多的模块

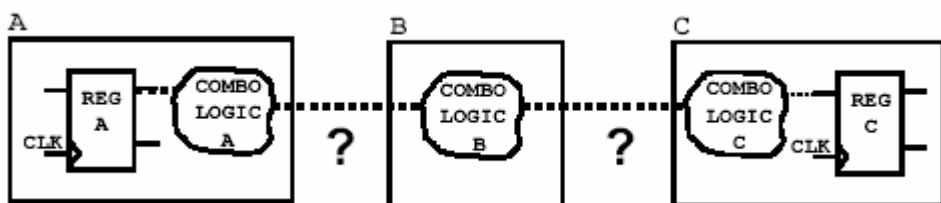


图 37 设计划分实例

上图中的组合逻辑电路存在于寄存器 A 与寄存器 C 之间，它同时穿过了模块 A、模块 B 以及模块 C。前面提到了，如果直接将这样的划分交给 DC 综合，那么综合后的电路将仍旧保持上面的层次关系，即端口定义不会改变。这样的话，DC 在作组合电路的优化的时候就会分别针对 A、B、C 三块电路进行，这样势必会影响到 DC 的优化能力，不必要的增加了这条路径的延时和面积。因此，可以考虑将三块分散的组合逻辑划分到一个模块中，如下图所示——

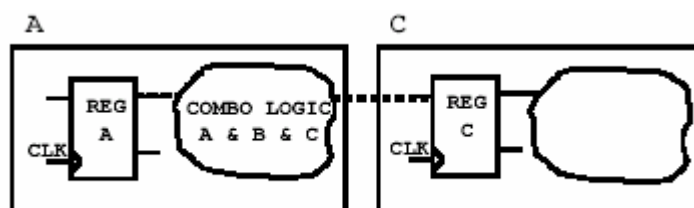


图 38 较好的划分

这张图说明了组合电路划分到一个模块之后的电路情况，这样 DC 就可以充分的施展它的优化技巧，综合出比较满意的电路来。为什么说它只是一个较好的划分呢？因为它只是考虑到组合电路的最优划分而没有想到时序电路部分。先看下面一张图——

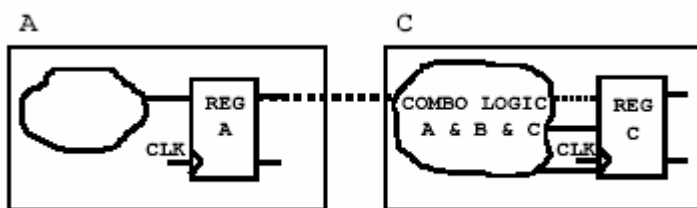


图 39 最好的划分

在这张图里，组合逻辑被划到了 C 模块中，它不仅能保证组合的最佳优化还能保证时序的最佳优化，因为里面的寄存器在优化的过程中可以吸收前面的组合逻辑，从而形成其他形式的时序元件，如由原先的 D 触发器变成 JK 触发器、T 触发器、带时钟使能端的触发器等等。这样工艺库中的大量的时序单元都可以得到充分的利用了。

原则二. 寄存模块的输出

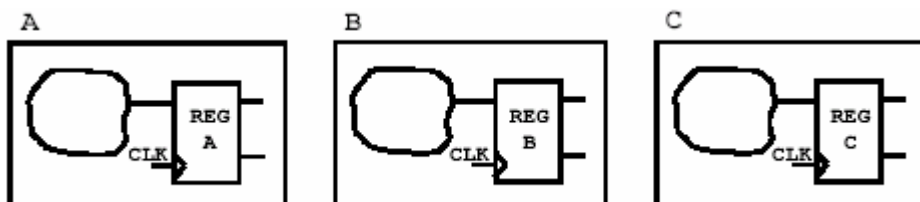


图 40 寄存模块输出

通过前面的讨论，可以知道：在编写代码或者综合的过程中，我们可以把模块尽量写成如图 37 那样的逻辑结构：将所有的输出寄存起来。其实这样不但是最佳的优化结构，也可以简化时序约束（使得所有模块的输入延时相等）。

就算遵循了输出寄存的原则，我们还是可能犯下面的错误——

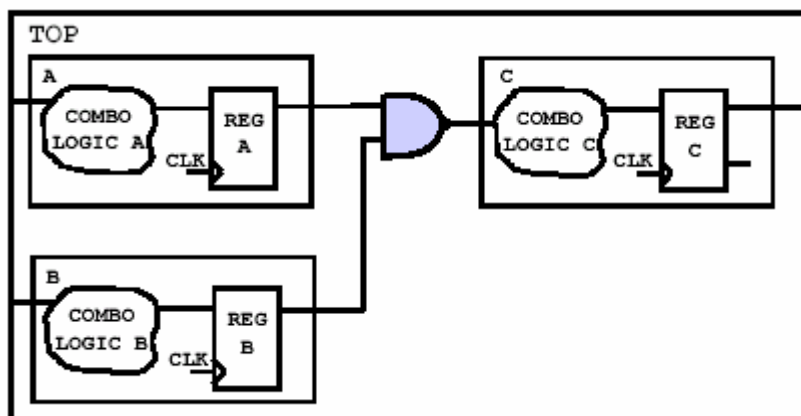


图 38 粘滞逻辑(Glue Logic)

图中可以看到，一个与非门连接了 A、B、C 三个模块，同样的不难看出，它也会影响到 C 的组合逻辑的优化。一般这种情况只会在至下而上的综合策略中出现。可以通过把与非门吸收到 C 中的组合逻辑的方法消除粘滞逻辑（如下图），从而使得电路的顶层模块仅仅是将子模块拼接在一起，而没有独立的电路结构，这样的另一个好处是可以使得在至下而上的设计策略中不需要编译顶层模块。

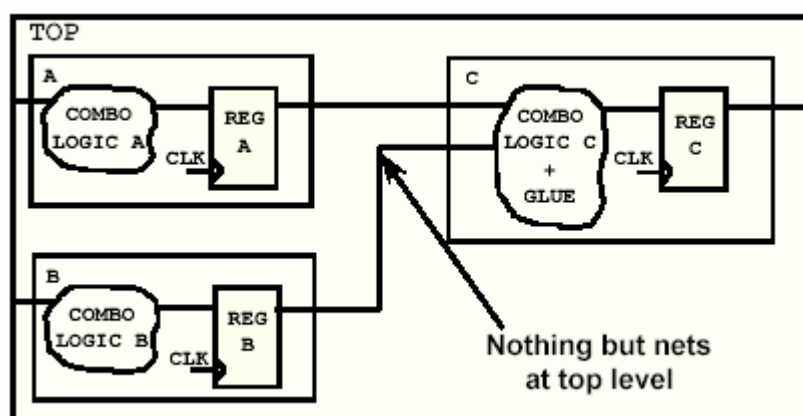


图 41 消除粘滞逻辑

原则三. 根据综合时间长短控制模块大小

综合时间主要受到硬件条件(主频、内存)的制约，对于早期的工作站而言，硬件水平不高，跑一个大型的设计可能会一次花上几天时间才会有结果，这样对调试和缩短工期是不利的，所以需要根据工作站的能力选择合适的模块大小。请看下面一个例子——

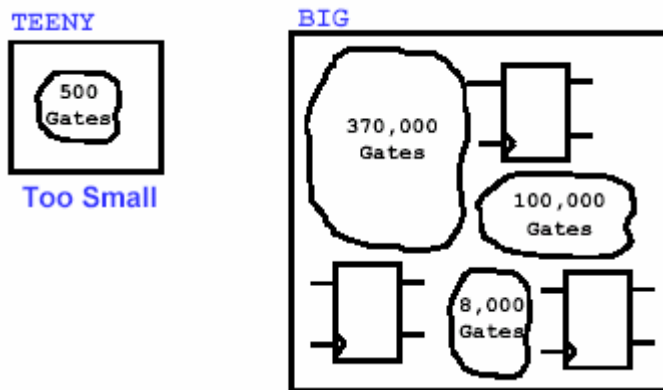


图 42 针对模块大小不好的划分

这个例子的模块大小从 500 门到 37 万门不等，假设工作站的硬件条件限制最多只能跑 30 万门的设计，那么上面的这种划分就有一些弊病。首先，TEENY 模块太小，不适合优化出最好的结果，可以考虑将它合并到其他模块中。其次，另一个组合模块逻辑太大，这势必使得综合的时间变得不能承受。因此，改进的划分可以如下图所示——

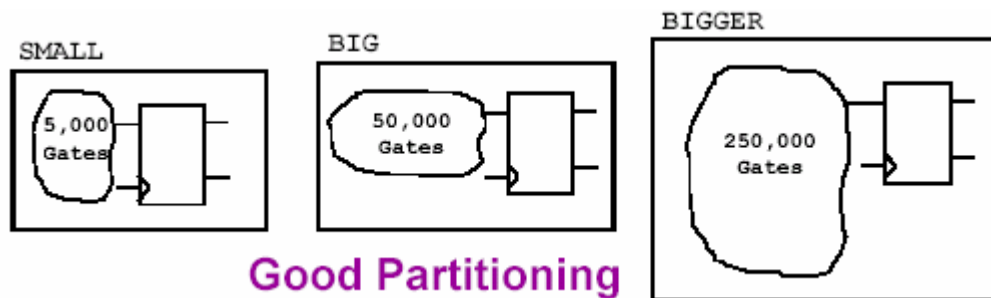


图 43 针对模块大小较好的划分

可以看到，虽然各个模块也是大小不一，但却可以取得较前面的划分更好的结果。值得注意的是——Design Compiler 软件本身没有模块大小的限制，它完全根据工作站的环境决定，在具体作设计的时候，我们可以在硬件条件允许的条件下编译较大的模块，假如硬件条件的确有限，只能选择小的模块来综合了⑥ **模块大小该如何划分？**

原则四. 将同步逻辑部分与其他部分分离

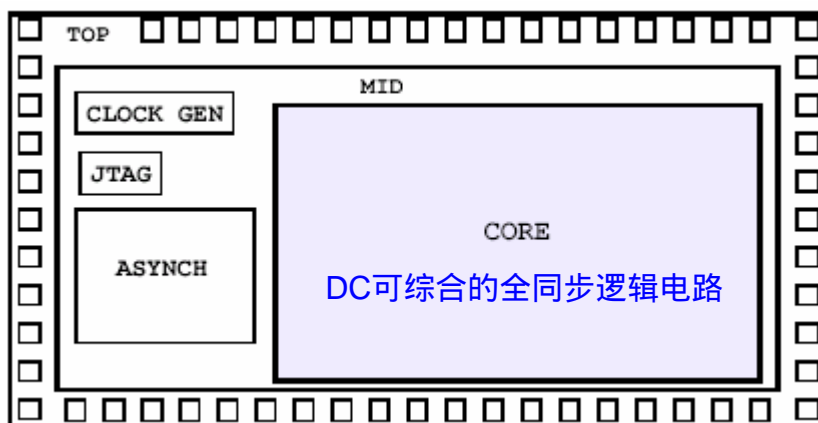


图 44 顶层设计的划分

上图是一个芯片的顶层设计，可以看到它被分层了三个层次——最外边是芯片的 Pad，Pad 是综合工具中没有的，也不是工具能生成的，它由 Foundry 提供，并由设计者根据芯片外围的环境手工选择；中间一层被分成四个部分，其中最里面那个称为 Core，也就是 DC 可以综合的全同步逻辑电路，另外的三个部分 DC 不能综合，需要其他的办法来解决：ASYNCH 是异步时序部分，不属于 DC 的范畴；CLOCK GEN 是时钟产生模块（可能用到 PLL），尽管有一部分同步电路，但也不符合综合的条件；JTAG 是边界扫描的产生电路，这一部分可以由 Synopsys 的另外一个工具 BSD Compiler 自动生成。

同步用DC综合，异步和时钟怎么办啊？

上面我们介绍了四个划分的原则，当然这些原则并不是我们在编写 HDL 代码的时候就必须遵守的，它只是说明什么样的设计划分对于 DC 来说是最理想的，能得到最优化的结果。事实上除了通过 HDL 中的模块体现划分，我们还可以运用 DC 的两个命令（Group 及 Ungroup）来调整设计划分。

先看一个例子，调整“原则一”里提到的那个划分不好的模块——

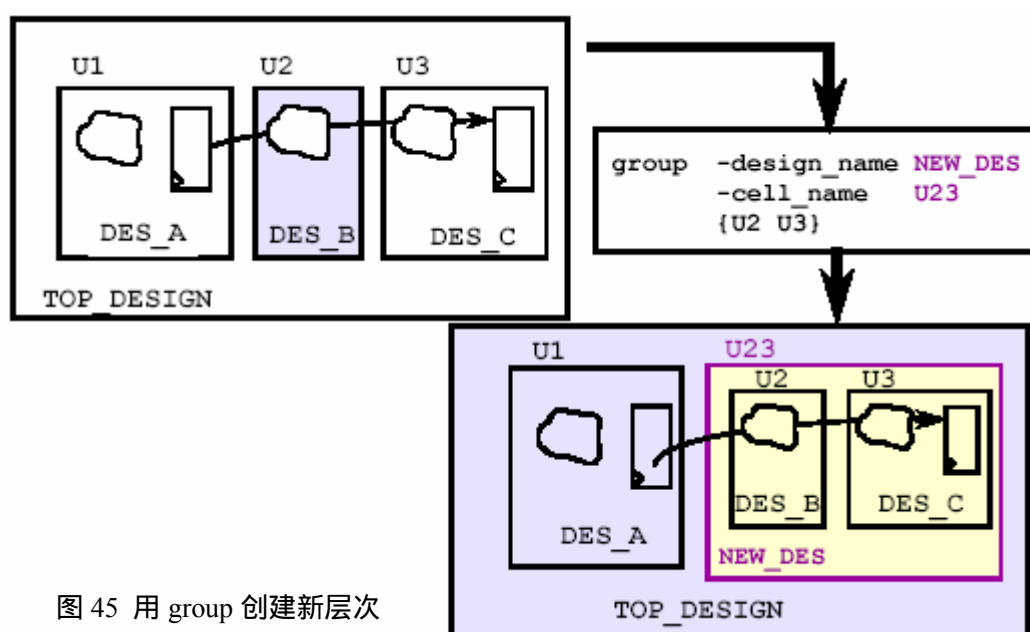


图 45 用 group 创建新层次

第一步是使用 group 命令，创建一个新的模块 NEW_DES(设计名)，单元名为 U23，包含了 U2 和 U3，这个命令很直观，很容易看懂。

第二步则是使用 ungroup 命令，将 U23 中的 U2 和 U3 的边界去掉，使之称为一个整体，如下图所示——

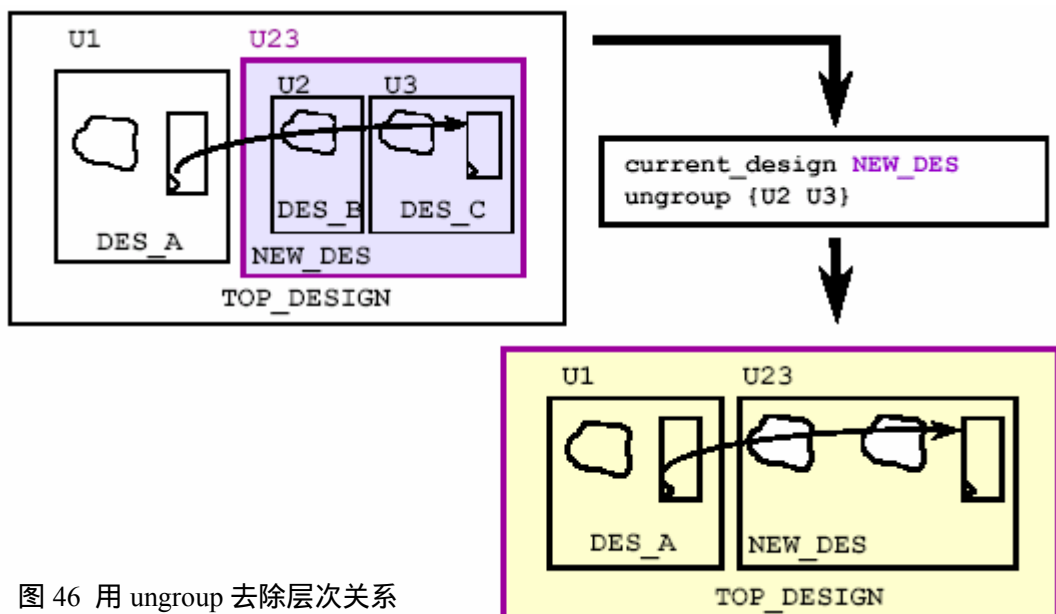


图 46 用 ungroup 去除层次关系

3.2 施加设计约束

Design Compiler 是一个约束驱动(constrain-driven)的综合工具，它的结果是与设计者施加的约束条件密切相关的。在这一章里，我们主要讨论怎样给电路施加约束条件，这些约束主要包括——时序和面积约束、电路的环境属性、时序和负载在不同模块之间的分配以及时序分析，在本章的最后一节还将讨论 DC Tcl 语言的一些基本语句。

3.2.1 时序和面积

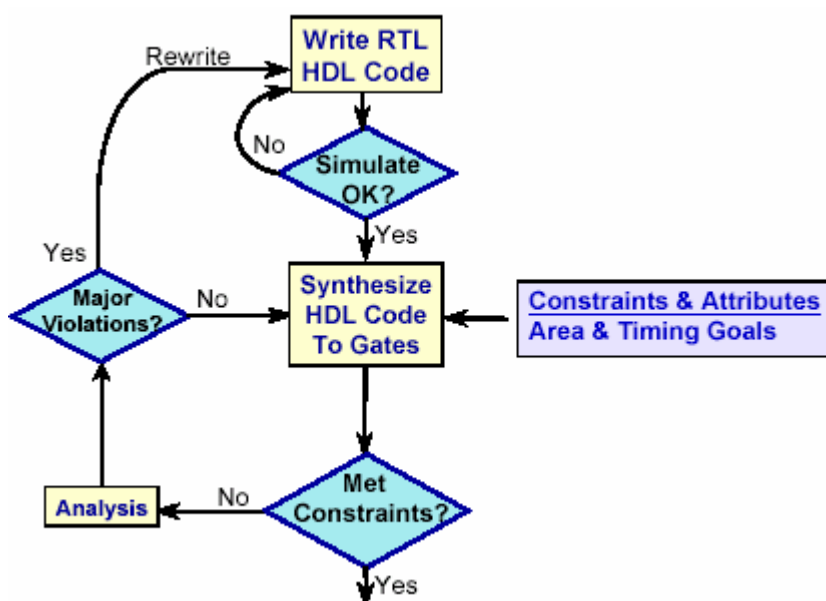


图 43 RTL 模块综合示意图

上图是 RTL 模块的综合示意图，可以看出在 RTL 代码仿真通过以后，就开始将它进行综合，综合时需要对他加入约束和设计属性的信息，DC 根据这些约束将 RTL 模块综合成门级网表，然后分析综合出的网表是否满足约束条件，如果不满足就要修改约束条件，甚至重写 RTL 代码。值得注意的是，上面提到的仅仅是 RTL 模块的综合过程，而不是整个芯片的综合，整个芯片是由很多这样的模块组成的，它的综合过程与上图描述的过程有一定的区别，具体我们将在后面的章节中进行讨论。

3.2.1.1 定义面积约束

因为芯片面积直接关系到芯片的成本，面积越大，成本越高，因此，集成电路的设计总是希望面积尽量小，以减小芯片成本。定义面积约束是通过 `set_max_area` 命令来完成的，比如——

```
dc_shell-t> current_design PRGRM_CNT_TOP
dc_shell-t> set_max_area 100
```

上面的例子给 PRGRM_CNT_TOP 的设计施加了一个最大面积 100 单位的约束。100 的具体单位是由 Foundry 规定的，定义这个单位有三种可能的标准：一种是将一个二输入与非门的大小作为单位 1；第二种是以晶体管的数目规定单位；第三种则是根据实际的面积(平方微米等等)。至于设计者具体用的是哪种单位，可以通过下面的一个小技巧得到——即先综合一个二输入与非门，用 `report_area` 看他的面积是多少，如果是 1，则是按照第一种标准定义的；如果是 4，则是第二种标准；如果是其他的值，则为第三种标准。

3.2.1.2 同步设计的时序特点和目标

同步时序电路是 DC 综合的前提，因此这里有必要先讨论一下同步时序电路的特点及目标。这里所讨论的同步时序电路的特点是——电路中的信号从一个受时钟控制的寄存器触发，到达另一个受时钟控制的寄存器。而我们要达到的目标是——约束电路中所有的时序路径，这些时序路径可以分为三类：输入到寄存器的路径、寄存器到寄存器之间的路径以及寄存器到输出的路径。他们分别对应与下图所示的标号为 N、X 和 S 的电路。

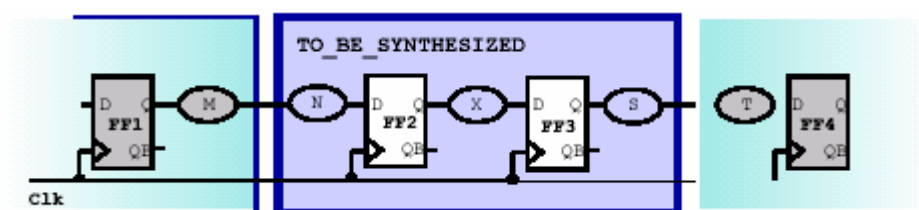


图 44 同步设计的时序特点

假设在上面的电路中，我们要控制触发器 FF2 到 FF3 之间的时序，即 X 电路的延时，那要通过什么方式让 DC 知道呢？显然一个直观的办法就是定义系统的时钟 Clk，如果我们定义好了 Clk 的周期，那么 DC 会自动的尽量保证从 FF2 触发的信号能在一个周期内到达

FF3 寄存器。假如周期是 10ns，FF3 触发器的建立时间(setup time)是 1ns，那么留给 X 电路的延时最大只能有 $10-1=9\text{ns}$ 。

3.2.1.3 定义时钟

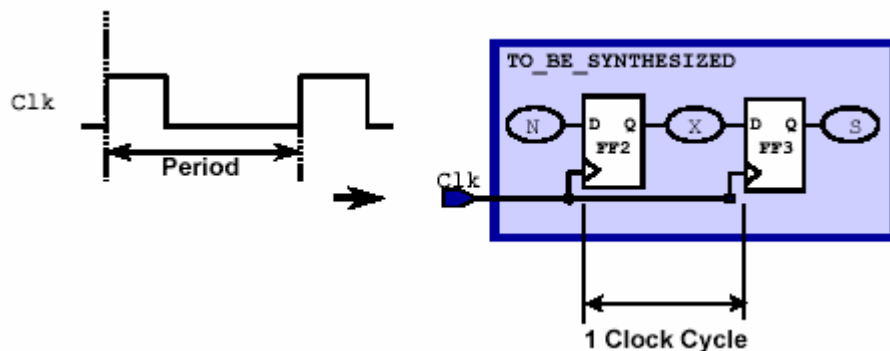


图 45 定义时钟

在电路综合的过程中，所有时序电路以及组合电路的优化都是以时钟为基准来计算路径延迟的，因此，一般都要在综合的时候指定时钟，作为估计路径延迟的基准。定义时钟的时候我们必须定义它的时钟源(Clock source)，时钟源可以是端口也可以是管脚；另外还必须定义时钟的周期。另外有一些可选项，比如占空比(Duty Cycle)、时钟偏差(Clock Skew)和时钟名字(Clock Name)。定义时钟采用一个语句 create_clock 完成——

```
dc_shell-t> create_clock -period 10 [get_ports Clk]
dc_shell-t> set_dont_touch_network [get_clocks Clk]
```

第一句定义了一个周期为 10ns 的时钟，它的时钟源是一个称为 Clk 的端口。

第二句对所有定义的时钟网络设置为 don't_touch，即综合的时候不对 Clk 信号优化。如果不加这句，DC 会根据 Clk 的负载自动对他产生 Buffer，而在实际的电路设计中，时钟树(Clock Tree)的综合有自己特别的方法，它需要考虑到实际布线后的物理信息，所以 DC 不需要在这里对它进行处理，就算处理了也不会符合要求。

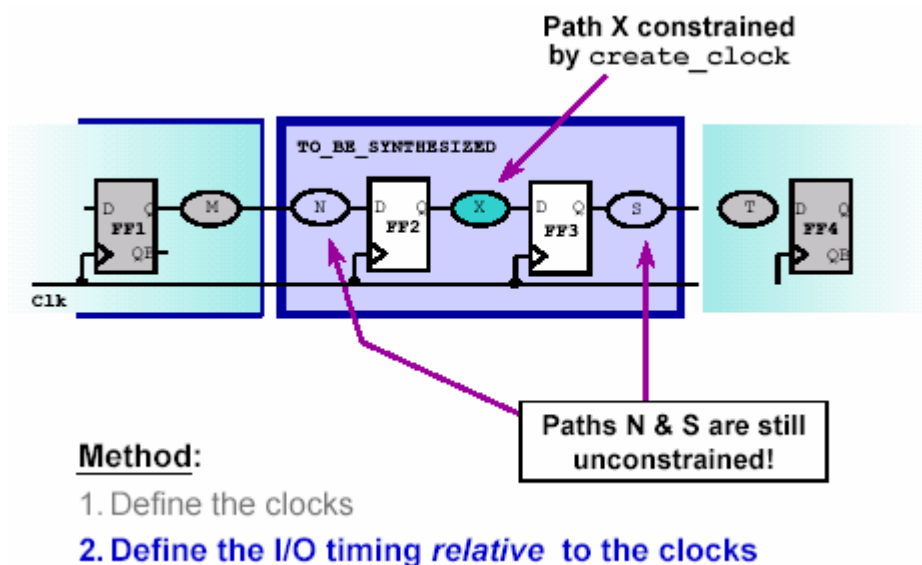


图 46 定义时钟后的电路

可以看到，定义了系统时钟后，图 44 中的 X 电路已经被约束起来了，但是电路的输入输出两块还没有施加约束，这可以通过 DC 的另外两个命令来完成——

3.2.1.4 约束输入路径

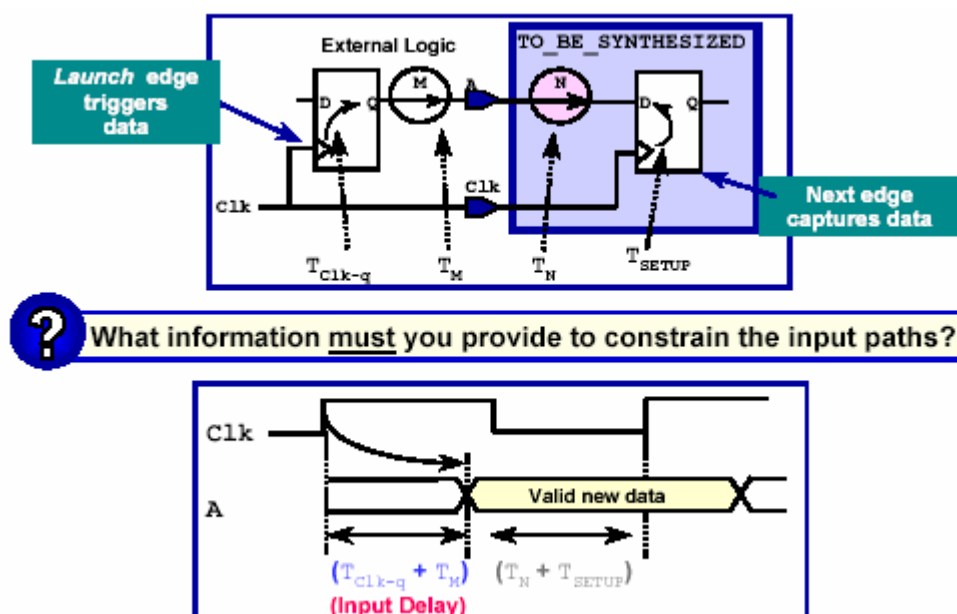


图 47 定义输入路径延时

从上图可以看出，我们所要定义的输入延时是指被综合模块外的寄存器触发的信号在到达被综合模块之前经过的延时，在上图中就是外围触发器的 clk-q 的延时加上 M 电路的延时。当确定这段延时之后，被综合模块内部的电路延时的范围也可以确定下来了。加入时钟周期是 20ns，输入延时是 4ns，内部触发器的建立时间为 1.0ns，那么就可以推断出要使电路正常工作，N 电路的延时最大不能超过 $20 - 4 - 1.0 = 15.0\text{ns}$ 。

设置输入延时是通过 DC 的 set_input_delay 命令完成的——

```
dc_shell-t> set_input_delay -max 4 -clock Clk [get_ports A]
```

如上面的语句指出了被综合模块的端口 A 的最大输入延时为 4ns。-max 选项是指目前设置的是输入的最大延迟，为了满足时序单元建立时间（setup time）的要求。另外还有一个选项是-min，它是针对保持时间的约束使用的，后面章节有详细介绍。-clk 是指出这个端口受哪个时钟周期的约束。

定义了输入延时之后，相对应的还要设置电路的输出延时。

3.2.1.5 约束输出路径

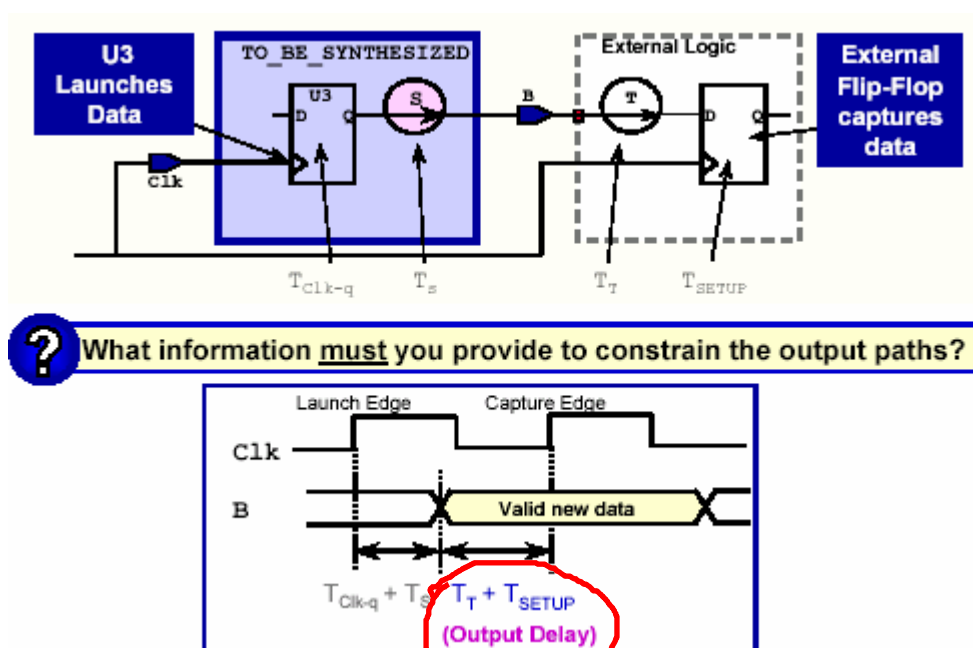


图 48 定义输出路径延时

上图中，信号在被综合模块的触发器 U3 里触发，被外围的一个触发器接收。对外围电路而言，它有一个 T 电路延时和外围触发器的建立时间。当确定了他们的延时之后，被综合模块内部的输出路径延时范围也就确定下来了。假如，时钟周期 20ns，输出延时 5.4ns，U3 触发器的 clk-q 延时为 1.0ns，那么输入路径 S 的最大延时就是 $20 - 5.4 - 1.0 = 13.6\text{ns}$ 。

设置输入延时是通过 DC 的 set_output_delay 命令完成的——

```
dc_shell-t> set_output_delay -max 5.4 -clock Clk [get_ports B]
```

上面的语句指出了被综合模块的输出端口 B 的最大输出延时为 5.4ns。-max 选项是指目前设置的是输入的最大延迟；-clk 是指出这个端口受哪个时钟周期的约束。

至此，模块的面积、时钟、输入输出延时都施加了相应的约束。在施加了这些约束之后，可以使用下面的几个命令检查约束是否施加成功——

- report_port -verbose

报告在当前设计中所有的输入输出端口属性和施加的约束值

- report_clock

报告当前设计中定义的时钟及其属性情况

- reset_design

删除当前设计中所有的属性值和约束(通常用在约束脚本的第一句)

- list_libs

列出内存中所有可用的库

3.2.2 环境属性

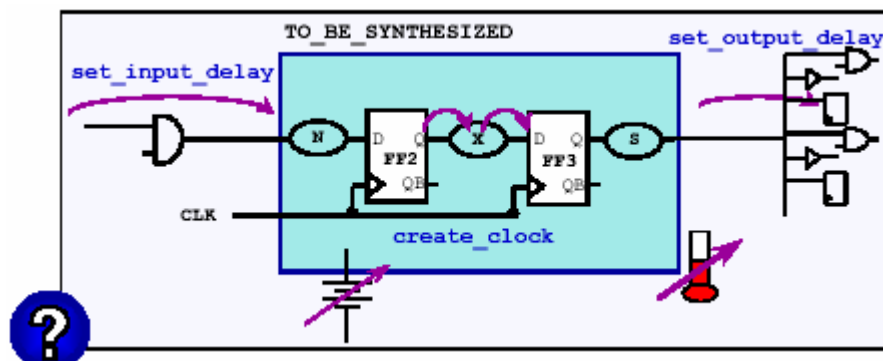


图 49 加入时序约束后的电路图

在 3.2.1 节中，我们主要讨论了怎样电路中加入时序约束，如设置 clock 周期、设置输入输出延时等，但是仅仅靠这些约束还是不够的。因为还要考虑到被综合模块周围环境的变化，举个例子说，如果当外界的温度变化，或者电路的供电电压发生变化时，延时会相应的改变，所以这些方面也是必须考虑到的。类似的上一节仅仅约束了输入输出的延时，而没有考虑到他们的电平转化时间(transition time)，这些是有输入输出的外围电路的驱动能力和负载大小决定的。另外，电路内部的互连线的延时也没有估计在内。这一节我们主要讨论怎样给电路施加这些环境属性。

设置环境属性的命令如下图所示——

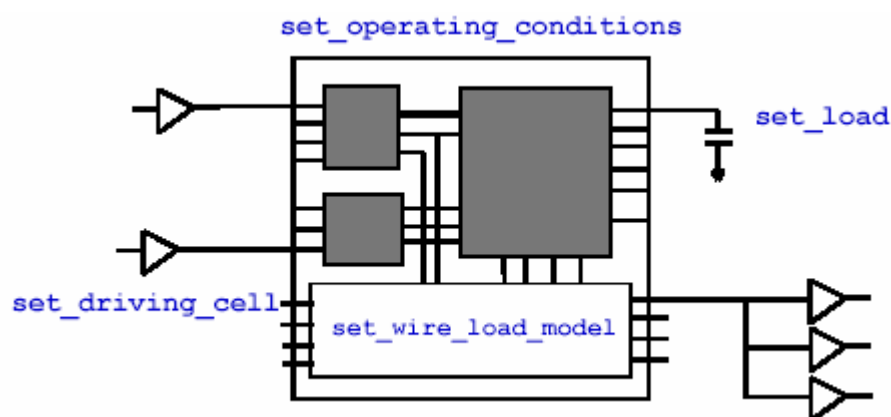


图 50 设置环境属性

3.2.2.1 设置输出负载

为了更加准确的估计模块输出的时序,除了知道它的输出延时之外还要知道输出所接电路的负载情况,如果输出负载过大会加大电路的 transition time,影响时序特性。另外,由于 DC 默认输出负载为 0,即相当于不接负载的情况,这样综合出来的电路时序显然过于乐观,不能反映实际工作情况。

设置输出负载是通过 DC 的 set_load 命令完成的——

该命令有两种用法,一种是直接给端口赋一个具体的值,另外则结合另一个命令 load_of 指出它的负载相当于工艺库中的哪个单元的负载值。

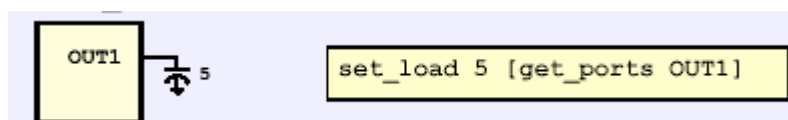


图 51 设置输出负载(1)

例如上图,给 OUT1 端口设了一个负载为 5 的值。这里的单位也是由 Foundry 提供,具体的单位,可以通过 report_lib 命令查看,一般而言是 pf。

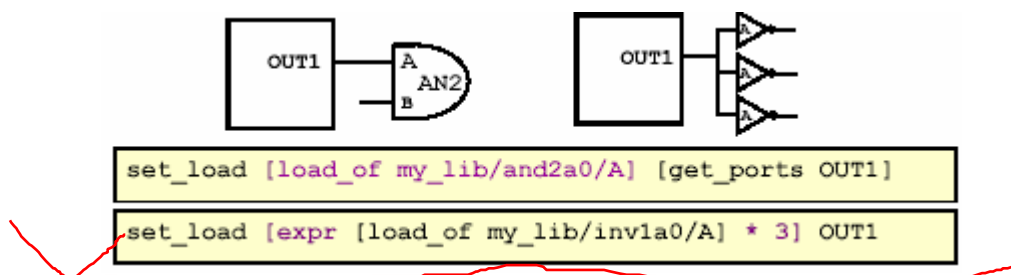


图 52 设置输出负载(2)

图 52 中设置负载采用了第二种方法,从图中可以看出,第一条语句说明 OUT1 端口接的负载值是 my_lib 中 and2a0 单元的 A 管脚的负载值。第二条语句则多用了 TCL 语言的表达式的语法,它说的是,OUT1 相当于接了三个 inv1a0 单元的 A 管脚的负载值。一般后面的这种方法用的多些。

3.2.2.2 设置输入驱动

与设置输出负载类似,为了更加准确的估计模块输入的时序,我们同样需要知道输入端口所接单元的驱动能力。在默认的情况下,DC 认为驱动输入的单元的驱动能力为无穷大,也就是说,transition time 为 0。

设置输入驱动是通过 DC 的 set_driving_cell 命令完成的。set_driving_cell 是指定使用库中的某一个单元来驱动输入端口。该命令是在输入端口之前假想一个驱动单元,然后按照该单元的 output resistance 来计算 transition time,从而计算输入端口到门单元电路的延迟——

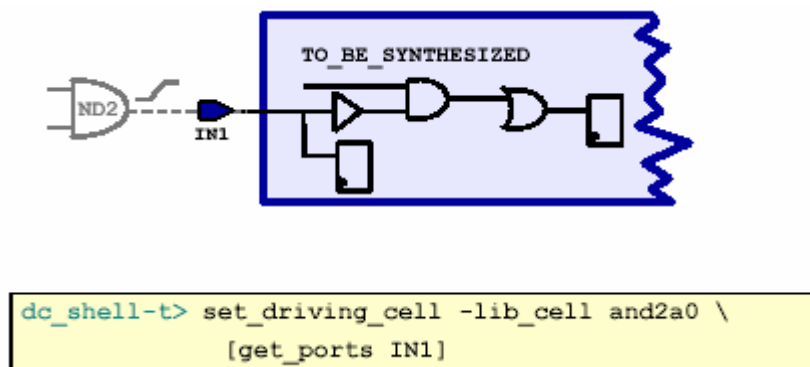


图 53 设置输入驱动单元

如上图所示，它设置了模块输入端口 IN1 的驱动单元是工艺库中的 and2a0。

3.2.2.3 设置工作条件

工作条件包括三方面的内容——温度、电压以及工艺。在 Foundry 提供的工艺库里，它的各个单元的延时是在一个“标准”(nominal)条件下得到的，比如说温度 25.0 度、工艺参数 1.0 和工作电压 1.8V。一旦工作条件发生了改变，电路的时序特性也必将收到影响，以上三方面的因素对电路时序的影响如下所示——

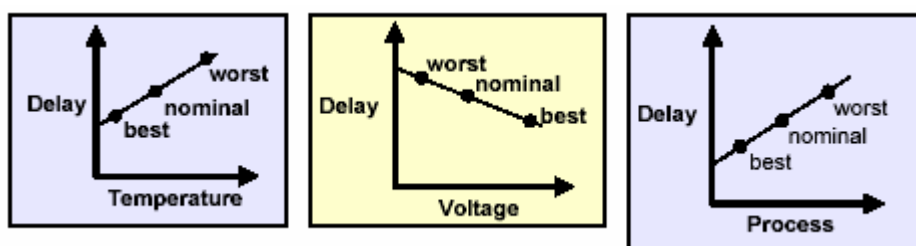


图 54 工作条件对时序的影响

从图中可以看出，单元的延时会随着温度的上升而增加；随着电压的上升而减小；随着工艺尺寸的增大而增大。以上的这些工作条件的变化，Foundry 在建库的时候已经考虑到了，因此它在工艺库中提供了几种工作条件的模型(operating condition model)以供设计者选择。这些工作条件一般分为三种：最好情况(best case)、典型情况(typical case)以及最差情况(worst case)。我们为了以后能使电路正常的工作在上面的三种情况下，在综合的时候就必需要将他们考虑进来。一般综合只要考虑到最差和最好两种情况，最差情况用于作基于建立时间(setup time)的时序分析，最好情况用于作基于保持时间(hold time)的时序分析。

在默认情况下，Design Compiler 不会自动指定工作条件，我们可以先通过 report_lib 命令来列出在当前的工艺库里提供了哪几种工作条件——

Operating Conditions:				
Name	Library	Process	Temp	Volt
typ_25_1.80	my_lib	1.00	25.00	1.80
slow_125_1.62	my_lib	1.00	125.00	1.62
fast_0_1.98	my_lib	1.00	0.00	1.98

图 55 列出可选的工作条件

然后指定需要用到工作条件,在做建立时间分析的时候需要用到最差情况的条件——

```
dc_shell-t> set_operating_conditions -max "slow_125_1.62"
```

如果我们既要分析建立时间,又要分析保持时间那么就要同时指定最差和最好情况——

```
dc_shell-t> set_min_library core_slow.db \
               -min_version core_fast.db
dc_shell-t> set_operating_conditions -max slow_125_1.62 \
               -min fast_0_1.98
```

其中 core_slow.db 和 core_fast.db 分别是最差和最好条件下的工艺库文件,第一句话先用 set_min_library 设定作保持时间检查的库,第二句话则分别对应了两种时间检查需要用到工作条件。

3.2.2.4 设置连线负载模型

在 DC 综合的过程中,连线延时是通过设置连线负载模型(wire load model)确定的。连线负载模型基于连线的扇出,估计它的电阻电容等寄生参数,它也是由 Foundry 提供的。Foundry 根据其他用这个工艺流片的芯片的连线延时进行统计,从而得到这个值。

下面是一个负载模型的例子——

Name	:	160KGATES	
Location	:	ssc_core_slow	
Resistance	:	0.000271	R per unit length
Capacitance	:	0.00017	C per unit length
Area	:	0	
Slope	:	50.3104	Extrapolation slope
Fanout	Length		
1	31.44		
2	81.75		
3	132.07		
4	182.38		
5	232.68		

Time Unit	:	1ns
Capacitive Load Unit	:	1.000000pf
Pulling Resistance Unit	:	1kilo-ohm

图 56 连线负载模型 (WLM)

这个例子可以通过命令 report_lib 得到,它是 ssc_core_slow 这个工作条件下的一个名为

160KGATES 的负载模型。其中时间单位为 1ns，电容负载单位为 1pf，电阻单位为 1kΩ。从图中可以看出单位长度的电阻以及电容值，DC 在估算连线延时，会先算出连线的扇出，然后根据扇出查表，得出长度，再在长度的基础上计算出它的电阻和电容的大小。若扇出值超出表中的值（假设为 7），那么 DC 就要根据扇出和长度的斜率(Slop)推算出此时的连线长度来。

事实上，在每一种工作条件下都会有很多种负载模型，各种负载模型对应不同大小的模块的连线，如上图的模型近似认为是 160K 门大小的模块适用的。可以认为，模块越小，它的单位长度的电阻及电容值也越小，负载模型对应的参数也越小。

设置输入驱动是通过 DC 的 set_wire_load_model 命令完成的。

```
dc_shell-t> set current_design addtwo
dc_shell-t> set_wire_load_model -name 160KGATES
```

如上面的语句，则设置了 addtwo 这个模块的连线负载模型为 160KGATES。

另外我们也可以让 DC 自动根据综合出来的模块的大小选择负载模型，这个选项在默认下是打开的。如下图所示，当综合出的电路的面积小于 43478.00 时，使用 5KGATES 的模型，属于 43478.00 和 86956.00 之间时，使用 10KGATES 的模型。

dc_shell-t> report_lib ssc_core_slow

Selection		Wire load name
min area	max area	

0.00	43478.00	5KGATES
43478.00	86956.00	10KGATES
86956.00	173913.00	20KGATES
173913.00	347826.00	40KGATES
347826.00	695652.00	80KGATES

图 57 自动设置连线负载模型（WLM）

以上讨论的情况是一个模块内部连线的负载模型的估计。如果连线连接的是不同的模块，那么它的负载模型又将怎么估计呢？这就要用到连线负载模式(set_wire_load_mode)这个命令了。

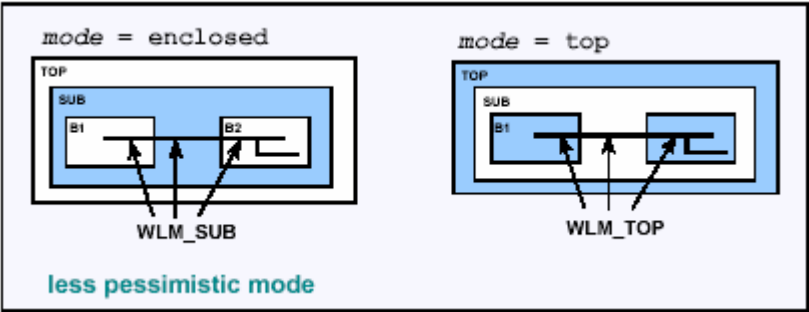


图 58 连线负载模式

连线负载模式一共有 3 种，围绕(enclosed)、顶层(top)以及分段 (segmented)。如上图所示，一根连线连接了 B1 和 B2 两个模块，这两个模块都位于 TOP 下的 SUB 这个子模块中，围绕模式是指连接 B1 和 B2 的连线的负载模型用围绕它们的模块的负载模型代替，即用 SUB

的负载模型；顶层模式是指用顶层模块的负载模型代替；分段模式顾名思义，分别根据穿过的三段的模型相加得到。如果要设置成围绕模式，可以使用如下命令——

```
dc_shell-t> set_wire_load_mode enclosed
```

在定义完环境属性之后，我们可以使用下面的几个命令检查约束是否施加成功——

- check_timing

检查设计是否有路径没有加入约束

- check_design

检查设计中是否有悬空管脚或者输出短接的情况

- write_script

将施加的约束和属性写出到一个文件中，可以检查这个文件看看是否正确

3.2.3 时序和负载预算

前面的两节里我们讨论了怎样给一个被综合的电路模块施加时序约束以及设置环境属性，大家学习完之后可能有这样一个疑问：模块的输入输出延时、负载和驱动单元的具体的值是怎样确定下来的呢？这一节里，我们就着重来探讨这个问题，也就是说怎样在综合模块之前先给它的时序和负载作一个预算，一般而言，这个工作是由项目的体系设计者(Achitecture Designer)完成，当他先确定好各个模块外围的时序和负载预算之后，再由具体的模块设计者(Module Designer)完成模块的综合。

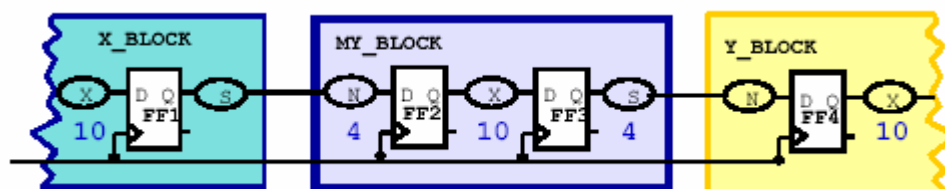


图 59 MY_BLOCK 的时序预算

3.2.3.1 时序预算

先看时序预算，假设电路中有三个模块 X_BLOCK、MY_BLOCK 和 Y_BLOCK，时钟周期是 10ns。如果要先综合 MY_BLOCK 模块，我们可以看到它的输入部分的 S 电路必须和 X_BLOCK 中的 N 电路共享一个周期的延时，同样输出部分的 S 电路也必须和 Y_BLOCK 的 N 电路共享一个周期的延时。我们可以作这样一个估计，认为处于两个模块交界部分的 S 和 N 电路只能分别占用 40% 的周期延时（如下图所示），也就是说限定所有的 S 和 N 电路

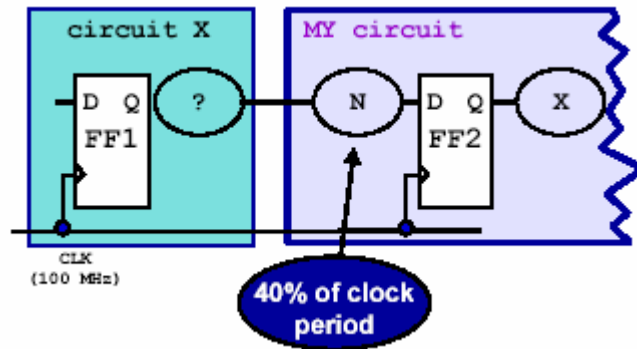


图 60 保守的时序预算

的延时为最大为 4ns，这样可以看出 S+N 的延时总共只有 8ns，小于 10ns，因此这是一种保守的预算方法，预留的 2ns 的延时可以在电路不满足时序的时候再加上。经过预算之后的 MY_BLOCK 模块的约束如下——

```
# A generic Time Budgeting script file
# for MY_BLOCK
create_clock -period 10 [get_ports CLK]
set_dont_touch_network [get_clocks CLK]

set_input_delay -max 6 -clock CLK [all_inputs]
remove_input_delay [get_ports CLK]
set_output_delay -max 6 -clock CLK [all_outputs]
```

同样的方法，我们也可以得出另外两个模块的时序约束——

```
# Time Budgeting script file
# for X_BLOCK and Y_BLOCK
create_clock -period 10 [get_ports CLK]
set_dont_touch_network [get_clocks CLK]

set_input_delay -max 6 -clock CLK [all_inputs]
remove_input_delay [get_ports CLK]
set_output_delay -max 6 -clock CLK [all_outputs]
```

3.2.3.2 负载预算

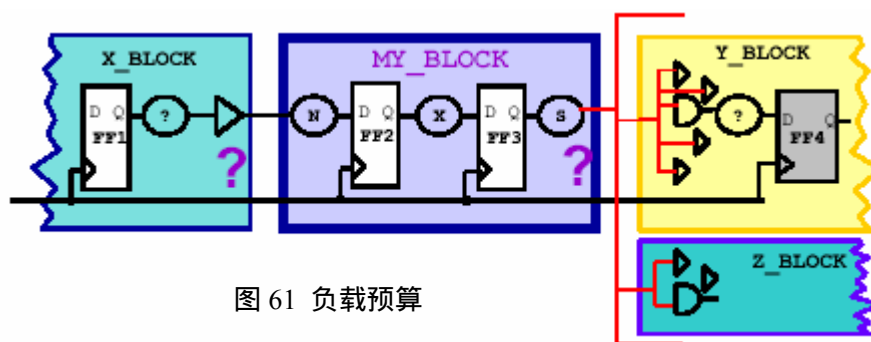


图 61 负载预算

负载预算也是在实际综合编译之前体系设计者根据模块将来可能的工作情况认为估计出来的，一般它的估计有几个原则：

- 保守起见，假设驱动模块的单元的驱动能力较弱
- 限制每一个输入端内部的负载电容
- 估计每一个输出端口最多可以驱动几个相同的模块

请看下面这个负载运算的例子——

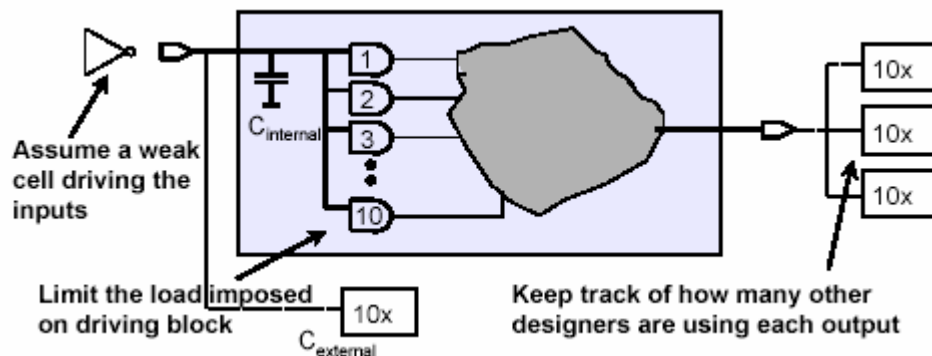


图 62 保守的负载预算实例

```
current_design myblock
link
source timing_budget.tcl

# Assume a weak driving buffer on the inputs
set_driving_cell -lib_cell inv1a0 [all_inputs]
remove_driving_cell [get_ports Clk]

# Limit the input load
set MAX_INPUT_LOAD [expr [load_of tech_lib/and2a0/A] * 10]

set_max_capacitance $MAX_INPUT_LOAD [all_inputs]
remove_attribute [get_ports Clk] max_capacitance

# Model the max possible load on the outputs, assuming
# outputs will only be tied to 3 subsequent blocks
set_load [expr $MAX_INPUT_LOAD * 3] [all_outputs]
```

在这个例子中，我们假设输入的驱动单元是一个 inv1a0 的反相器，然后限制了最大的输入负载，即每个输入端口的负载最大不得超过 10 个二输入与非门的负载大小，同时也规定了一个模块最多能同时驱动三个同样大小的其他模块。

3.2.4 时序分析

Design Compiler 是一个约束驱动(Constraint-driven)的综合工具，约束中最重要的就是时序约束，前面我们已经讨论了怎样在设计中施加时序约束，但是综合出来的电路能否满足这

些约束条件却是另一个重要的问题。在这一章里，我们主要将着重介绍 Design Compiler 内嵌的一个时序分析引擎 Design Time，并分析它是怎样根据工艺库中的单元延时信息和连线负载模型分析电路的静态时序的，并介绍调用 Design Time 的命令 report_timing。

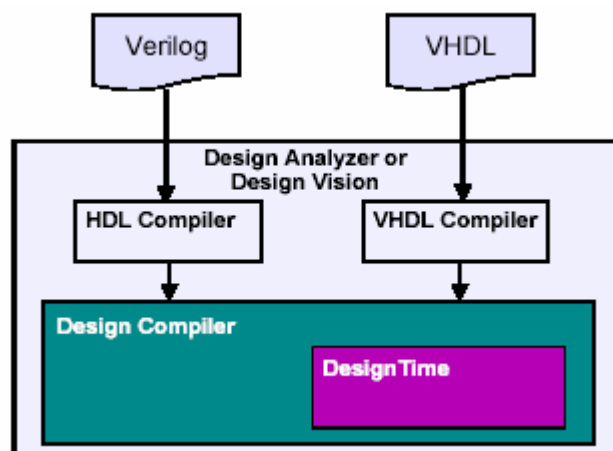


图 63 Design Time 在 DC 中的位置

图 63 中告诉了我们 Design Time 是 DC 的一个内嵌的静态时序分析引擎，DC 就是依靠它来计算电路的延时情况。DesignTime 和 PrimeTime 都是静态时序分析的工具，但是两者并不完全相同，PrimeTime 是在 DesignTime 的基础上发展起来的独立的专业的时序工具，而且效率和应用范围更高。

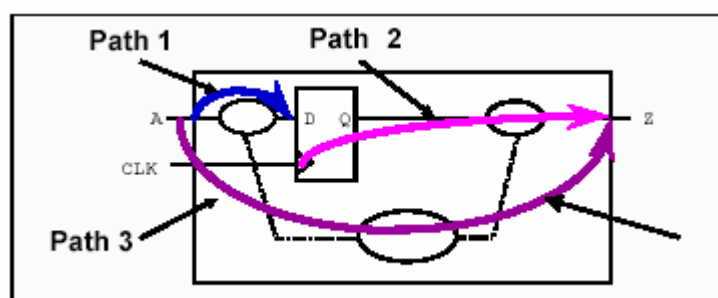


图 64 静态时序分析(STA)

静态时序分析(STA)是进行电路时序分析的一种方法，它的主要特点是分析不需要通过动态仿真，并且对电路的覆盖率更高。动态仿真(比如 VCS)需要给电路施加一个激励，并检查输出信号，与理想信号比较，这种办法速度较慢，而且不一定能覆盖到所有的逻辑。

静态时序分析分为三个步骤——

1. 将电路分解成不同的时序路径 (timing paths)
2. 计算每段路径的延时
3. 检查所有路径的延时，看是否能满足时序要求

下面我们就按照这三个步骤详细介绍 DC 是怎样分析电路的时序的。

3.2.4.1 分解时序路径

DesignTime 对时序路径的分解是根据时序路径的起点和终点的位置来决定的。每一条时序路径都有一条起点和终点，起点是输入端口或者触发器的时钟输入端；终点是输出端口或者触发器的数据输入端，如图 64 就有三条时序路径。另外根据终点所在的触发器的时钟不同还可以对这些时序路径进行分组(Path Group)，如下图电路中存在 4 条时序路径，3 个路径组，CLK1 和 CLK2 组分别表示他们的终点是受 CLK1 和 CLK2 控制的，DEFAULT 组则说明他们的终点不受任何一个时钟控制。

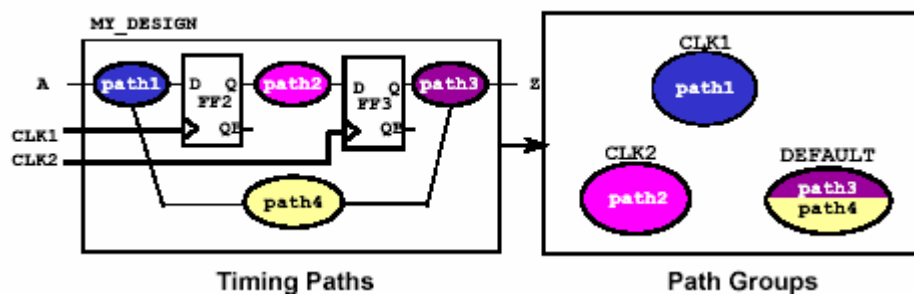


图 65 时序路径和路径组

再例如下面的一个电路，一共有 12 条时序路径和 3 条路径组。

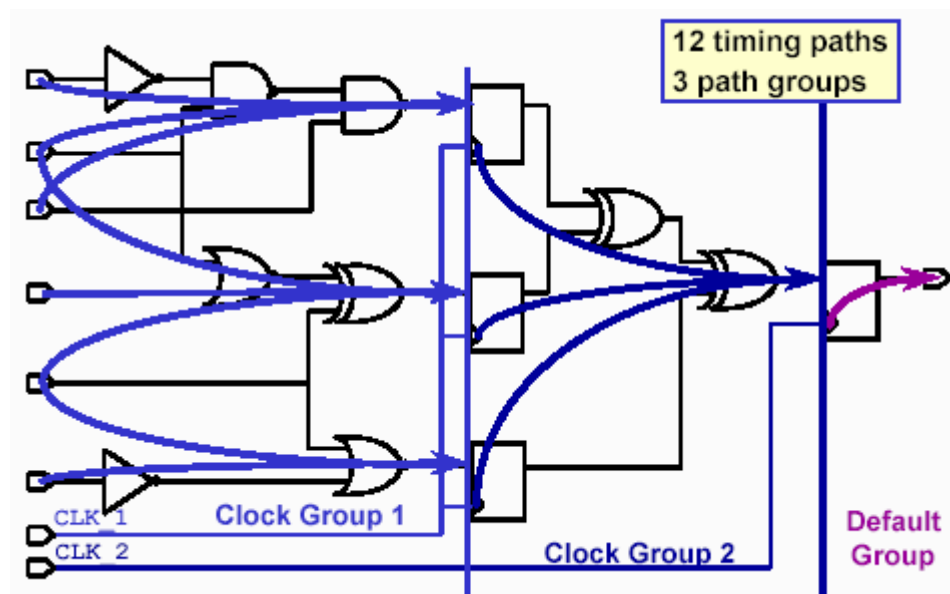


图 66 时序路径和路径组举例

3.2.4.2 计算单个路径延时

时序路径的延时包含单元延时和连线延时，为了便于观察，我们将上图的时序路径进一步划分，得到如下图所示——

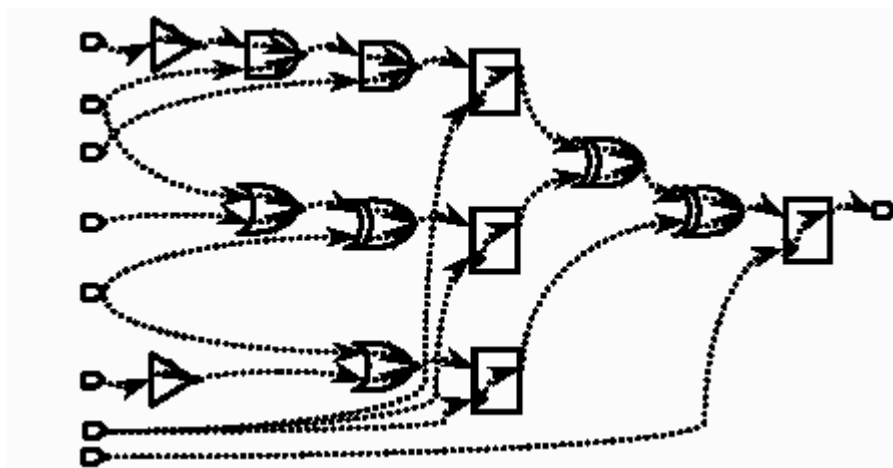


图 67 时序弧

- 可以看出，每条时序路径被分割成了一段段的时序弧(timing arc)，每条时序弧代表的不是一个单元延时就是一段连线的延时。下一步的工作就是计算出它们的值。

单元延时的计算

单元延时的计算是根据单元延时模型进行的，这里介绍两种单元延时模型，线性延时模型(Linear Delay Model)和非线性延时模型(Nonlinear Delay Model)，他们的计算方法如下图所示——

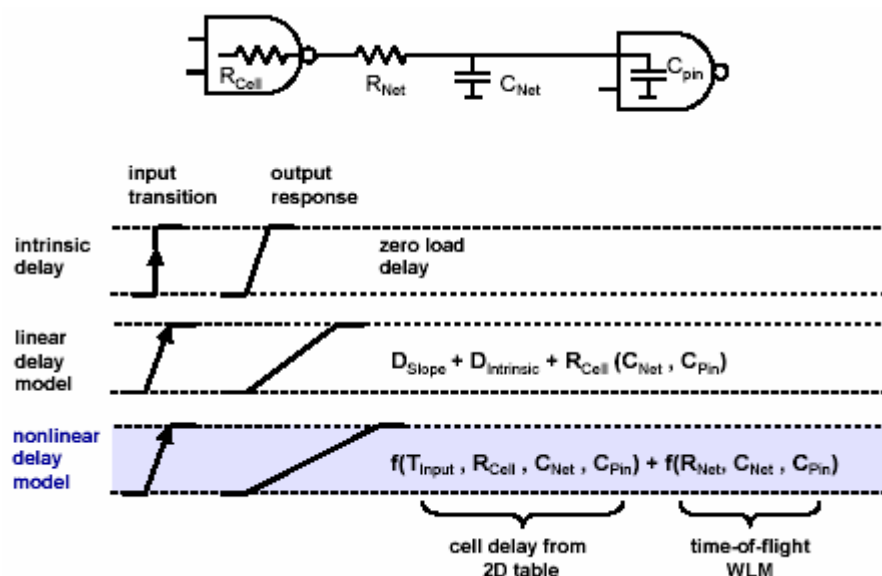


图 68 延时模型(delay model)

线性模型由三部分组成： D_{Slope} 表示单元输入信号的延时、 $D_{intrinsic}$ 表示单元的固有延时、 $R_{Cell}(C_{Net}, C_{Pin})$ 表示输出的管脚电容和连线电容对单元的附加延时。

非线性模型是 DC 计算单元延时的主要模型。它分为两部分：单元的输入延时(transition time)和输出负载的函数。与线性模型不同的是，它是通过查找表的方式得到的，请看下面

一个例子——

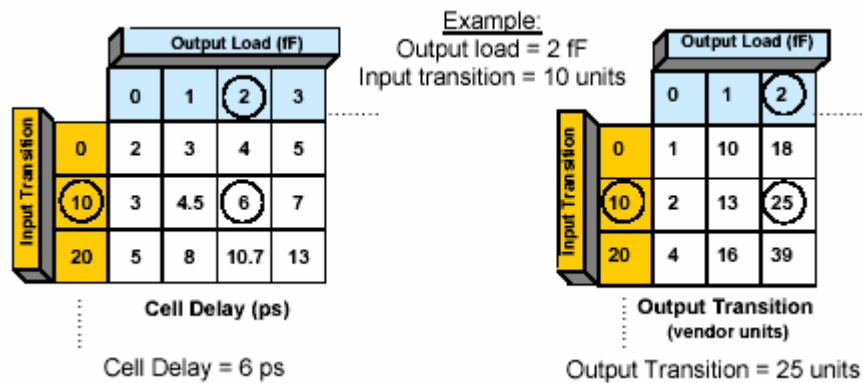


图 69 非线性延时模型的计算

对于一个单元来说，它有两张查找表，一张用于计算单元延时，另一张用于计算输出延时，并作为下一级单元的输入。在这个例子中，通过查表可以得到，单元延时为 6ps，输出的 transition time 为 25 个单位，这个单位又作为下一级的输入。

不难看出，非线性的模型的计算类似波浪的传播，从上一个单元的延时得出下一个单元的延时，并以此类推，对于一个较大规模的电路来说，计算量是比较大的，但也能得到比较准确的结果。

连线延时计算及其拓扑

在设置环境属性这一节中我们讲到了连线负载模型，通过连线负载模型我们可以得到一条连线上的电阻和电容的值，但是仅仅有 RC 值并不能得到连线的延时，还需要知道这些 RC 的分布，RC 分布有三种情况——

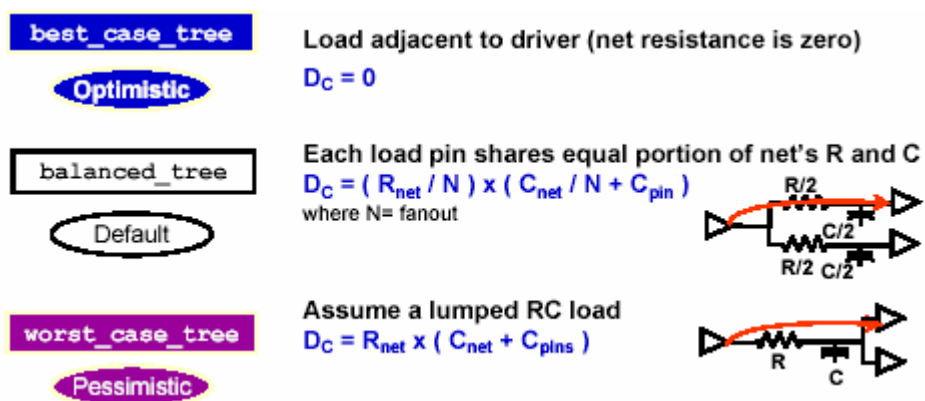


图 70 连线拓扑(RC-树)

- best_case_tree 是一种理想情况，它假设连线的电阻为零，平常很少使用
- balanced_tree 认为连线的 RC 均匀的分布在各条负载支路上
- worst_case_tree 假设 RC 值全部集中在负载共有的连线上，因此它的延时是最大的

连线的不同拓扑结构是通过工作条件的不同体现出来的，工作条件不但影响连线的延时，还通过温度、电压和工艺的变化影响单元延时的计算。

Operating Conditions:					
Name	Library	Process	Temp	Volt	Interconnect
slow_125_1.62	ssc_core_slow	1.00	125.00	1.62	balanced_tree
slow_125_1.62_WCT	ssc_core_slow	1.00	125.00	1.62	worst_case_tree

图 71 工作条件对连线延时的影响

3.2.4.3 计算整条路径的延时

DesignTime 计算完所有的路径延时之后的下一步工作就是根据这些延时，找出电路中延时最大或者最小的路径来。对于设计者而言，他们或许不关心每个单元的延时而更加关注到底电路是不是满足了设定的时序约束的要求。例如下图，有两个触发器 FF1 和 FF2，它们之间是一个很多单元组成的组合逻辑云，当作建立时间检查的时候，设计者就需要知道这个逻辑云的最大延时是否满足建立时间，即最大延时加上 FF2 的建立时间是否小于一个周期的时钟周期。

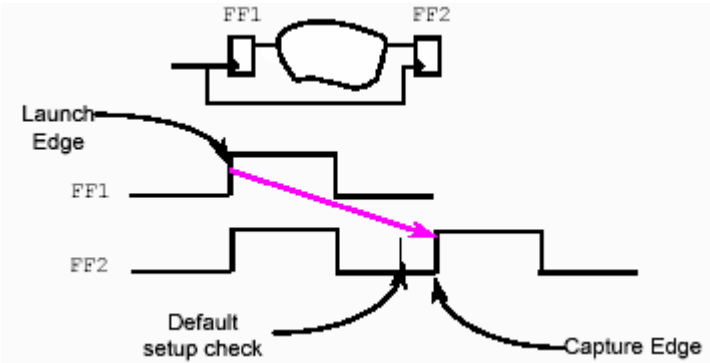


图 72 建立时间检查

那么怎样计算整条路径的最大延时呢？是不是把这条路径上的所有单元和连线的最大延时简单相加得到的呢？答案是否定的，因为这里涉及到一个时钟边沿敏感性(Edge Sensitivity)的问题。

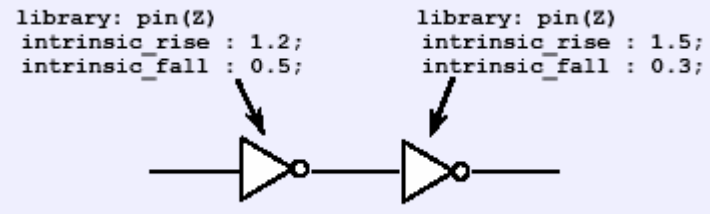


图 73 时钟边沿敏感性

如上图，一条路径上有两个不同的反相器，他们的固有上升时间和下降时间都不同，要计算他们的最大最小延时，需要弄清楚他们的工作过程——当第一个反相器的输入在时钟上

升沿时，它的延时是 1.2，同时第二个反相器处于时钟的下降沿，延时为 0.3；反之，当第一个反相器输入为时钟下降沿时，它们的延时分别为 0.5 和 1.5。因此，最大路径延时不是简单的 1.2+1.5，而是分别检查 0.5+1.5，在输入为时钟下降沿得到。实际上，DesignTime 在计算每条路径的时候，都会考虑边沿敏感性，即分别根据上升下降沿计算两次。

3.2.4.4 用 report_timing 检查时序

上面几节讲的是 DesignTime 作静态时序分析的基本原理和步骤，这节介绍引入 DesignTime 的命令——report_timing。

report_timing 命令的具体参数如下，具体信息请查看 DC 的 man 页。

```
report_timing
[ -delay max/min ]
[ -to name_list ]
[ -from name_list ]
[ -through name_list ]
[ -input_pins ]
[ -max_paths path_count ]
[ -nets ]
[ -capacitance ]
[ -path full_clock ]
...

```

默认的时候 report_timing 报告每一条路径组中的最长路径。报告一共分为 4 个部分——

```
*****
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : TT
Version: 2000.05
Date   : Tue Aug 29 18:22:38 2000
*****

Operating Conditions: slow_125_1.62   Library: ssc_core_slow
Wire Load Model Mode: enclosed

Startpoint: data1 (input port clocked by clk)
Endpoint:   u4 (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type:  max

Des/Clust/Port      Wire Load Model      Library
-----
TT                  5KGATES                ssc_core_slow

```

图 74 report_timing(1)

第一部分显示了路径的基本信息——工作状态是 slow_125_1.62，工艺库名称为 ssc_core_slow，连线负载模式是 enclosed。接下来指出这条最长路径的起点是 data1（输入端口），终点是 u4（上升沿触发的触发器），属于 clk 路径组，做的检查是建立时间检查(max)。这一部分的最后还报告了电路的连线负载模型。

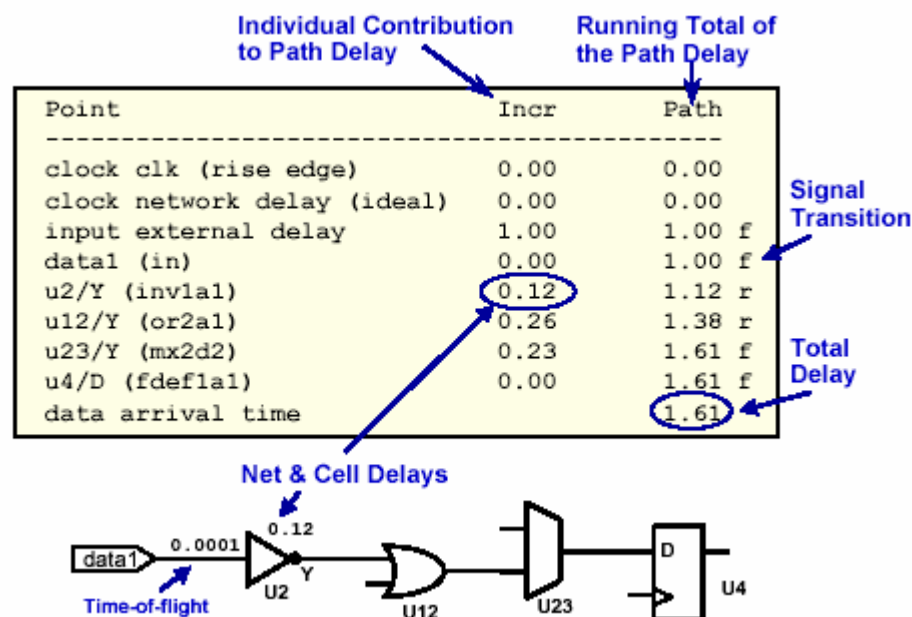


图 75 report_timing(2)

第二部分列出了这条最长路径所经过的各个单元的延时情况，分成三列：第一列说明的是各个节点名称，第二列说明各个节点的延时，第三列说明路径的总延时，后面所接的 f 或者 r 则暗示了这个延时是单元的哪个时钟边沿。例如图中的路径经过了一个反相器，一个二输入与非门，一个二输入 MUX，最后到达 D 触发器。其中反相器的延时为 0.12ns，路径总延时为 1.61ns。

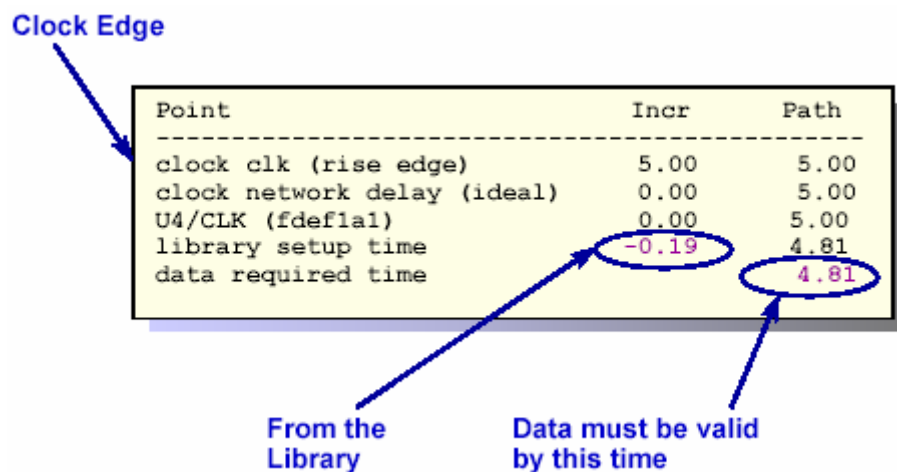


图 76 report_timing(3)

第三部分说明了这条路径所要求的延时，它是设计者通过时序约束施加的。例如时钟周期为 5ns，触发器的建立时间为 0.19（从工艺库中得到），要满足建立时间的要求，组合路径延时必须在 4.81ns 之内。

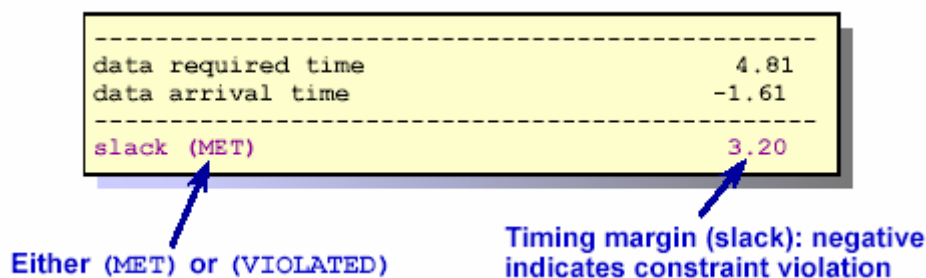


图 77 report_timing(4)

第四部分为时序报告的结论,它把允许的最大时间减去实际的到达时间,得到一个差值,这个差值称为时序裕量(Timing margin),如果为正数,则说明实际电路满足时序要求,为负数则说明有时序违反。上图的裕量为 3.20,说明最长路径满足建立时间的要求,且有 3.20ns 的裕量。暗含的意思是所有这个 clk 组的路径都满足建立时间的要求,并且裕量大于 3.20ns。

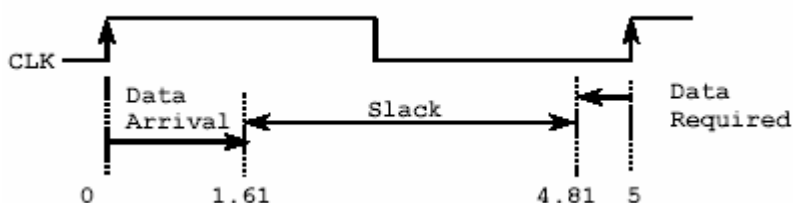


图 78 时序裕量

report_timing 除了报告电路综合后的时序之外,还可以帮助我们诊断综合电路中存在的时序问题,下面是一个例子——

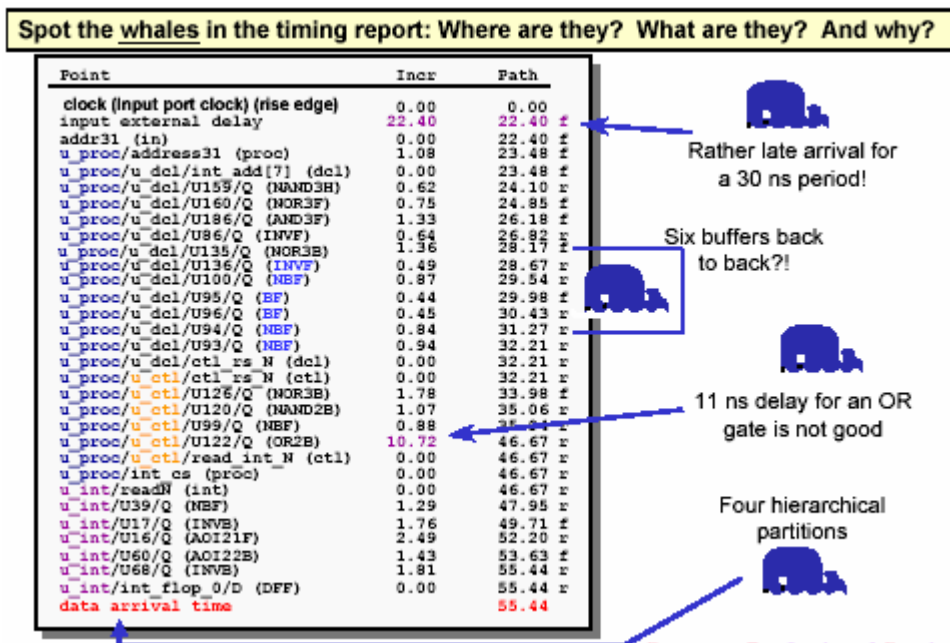


图 80 诊断综合结果

这个例子仅仅列出了报告的第二部分,报告的右边有四头鲸,它们分别指出了电路中存在的四个问题:

第一个问题出现在 input external delay 一栏,这一栏对应的就是时序约束中施加的

input_delay, 它的值为 22.4ns, 假设时钟周期为 30ns, 那么可以看出这个 input_delay 就已经占据了整个周期的 70%多, 因此留给下面单元的裕量就很少了。需要考虑是不是需要设置这么大的 input_delay。

第二个问题出现在路径中的 6 个串连的 buffer 上, 它们对应的单元分别是 INVF、NBF、BF、BF、NBF 以及 NBF。就逻辑功能来看, 6 个显得多余, 而且产生了不必要的延时。

第三个问题是由一个延时为 10.72 的或门造成的。其他的单元延时一般不超过 2ns, 一个 10.72 的单元是不是因为负载过重引起的呢? 值得仔细审查。

最后一个问题反映在这条路径穿过的层次上。从设计分层那一节可以知道, 要使延时最小, 应该把组合路径全部放在一个模块内。而这条组合逻辑同时穿过了 u_proc、u_proc/u_dcl、u_proc/u_ctl 以及 u_init 四个层次, 可以通过 group/ungroup 命令把层次重新组织一下。

3.2.5 DC Tcl 初步

TCL 的全称是 Tool Command Language, 它是由 UCA berkeley 开发的一种开放的工业标准语言。和 dc_shell 相比, Tcl 功能更加强大, 使用范围也更加广泛, 除了 Design Compiler 外, synopsys 的其他工具如 Formality、PrimeTime、Physical Compiler 等等都支持 Tcl。另外一些 EDA 厂商的工具也大多使用 Tcl 界面。

DC 除了 DC-Tcl 之外, 还支持较老的命令行方式 dc_shell, 我们可以在 Unix Shell 下通过使用 dc-transcript 命令将 dc_shell 的脚本转化为 DC-Tcl——

```
UNIX% dc-transcript my_script.scr my_script.tcl
```

3.2.5.1 执行 DC-Tcl 脚本

Tcl 脚本类似于 windows 环境下的 .bat 批处理文件, 就是将要执行的 Tcl 命令写在一个脚本中, 让后调用 dc_shell-t 执行之, 这样可以提高综合的效率。

执行 Tcl 脚本有两种方式: 一种是在 dc_shell-t 中调用——

```
dc_shell-t> source my.tcl
```

另一种是直接在 Unix Shell 中调用——

```
UNIX% dc_shell-t -f my.tcl
```

```
UNIX% dc_shell-t -f my.tcl > my.log
```

```
UNIX% dc_shell-t -f my.tcl | tee my.log
```

上面的 >my.log 表示将 DC 输出定向到 my.log 文件中, tee my.log 表示除了定向到 my.log 文件外还要在屏幕中输出。

3.2.5.2 在 DC-Tcl 中获得帮助

在 Tcl 环境中最基本的获得帮助的命令就是 help——

```
dc_shell-t> help
Procedures:

Builtins:
  after, alias, append, apropos, array, break, catch,
  cd, clock, close, concat, continue,
  create_command_group, define_proc_attributes, echo,
  eof, error, error_info, eval, exec, exit, expr,
  fblocked, fconfigure,
  . . .
Default Command Group:
  add_module, add_to_collection, all_clocks,
  all_cluster_cells, all_clusters, all_connected,
  all_critical_cells, all_critical_pins, all_designs,
  all_inputs, all_outputs, all_registers, analyze,
  balance_buffer, balance_registers, bc_check_design,
```

直接键入 help 后将列出 DC-Tcl 中的所有命令。另外我们也可以使用通配符(“*”)找到需要查找的命令，例如要找到含有 clock 的命令——

```
dc_shell-t> help *clock
clock                # Builtin
create_clock          # create_clock
create_test_clock     # create_test_clock
remove_clock          # remove_clock
remove_propagated_clock # remove_propagated_clock
report_clock          # report_clock
set_propagated_clock  # set_propagated_clock
```

如果要进步找到某一个命令的参数，就可以使用 help -v 选项，下面这个例子就列出了所有的 set_input_delay 的开关——

```
dc_shell-t> help -verbose set_input_delay
set_input_delay      # set_input_delay
[-clock clock_name] (relative clock)
[-clock_fall]        (delay is relative to falling edge of clock)
[-level_sensitive]   (delay is from level-sensitive latch)
[-rise]              (specifies rising delay)
[-fall]              (specifies falling delay)
[-max]               (specifies maximum delay)
[-min]               (specifies minimum delay)
[-add_delay]         (don't remove existing input delay)
delay_value          (path delay)
port_pin_list        (list of ports and/or pins)
```

注意：假设要看这个命令的更详细的帮助，就只能借助 man 命令。

3.2.5.3 DC-Tcl 中的注释

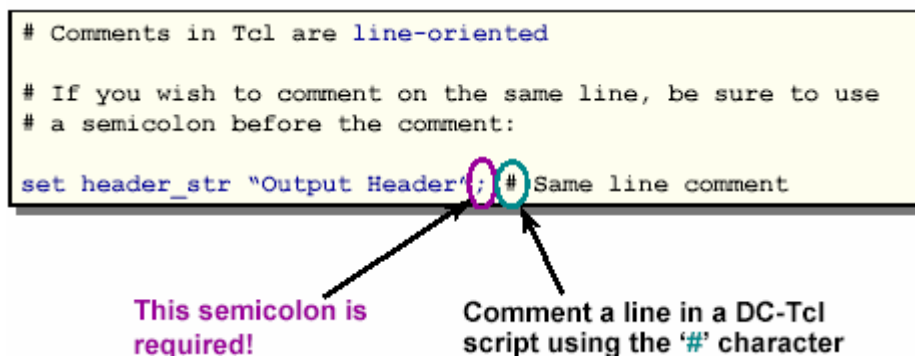


图 81 Tcl 中注释

DC-Tcl 的注释有两种：行注释和行内注释。行注释和 Cshell 类似，只要在开头加井号即可("#")；行内注释除了在注释前加井号外还要在井号前加分号(";")。

3.2.5.4 DC-Tcl 中的变量

1. Tcl 变量可以通过 set 来创建和赋值，如

```
dc_shell-t> set my_var 10
```

设置变量 my_var 的值为 10。

2. 引用变量的值，需要在变量名之前加入"\$"符号，如

```
dc_shell-t> set my_New_Var $my_var
```

将 my_var 的值赋给了新的变量 my_New_Var。

3. 打印变量的值，使用 echo 语句，如

```
dc_shell-t> echo $my_var
```

4. 删除变量，使用 unset 语句，如

```
dc_shell-t> unset my_var
```

5. 变量的弱引用，使用双引号，如

```
dc_shell-t> set a 5
dc_shell-t> set s "temp = data[$a]"
temp = data[5]
```

上面的 data[\$a] 中变量 a 的值为 5

6. 变量的强引用，使用大括号，如

```
dc_shell-t> set s {temp = data[$a]}
temp = data[$a]
```

上面的 data[\$a] 中的 a 将不再作为一个变量使用。

7. 变量中的表达式，使用 expr 语句，如

```
dc_shell-t> set period 10.0
10.0
dc_shell-t> set freq [expr (1 / $period)]
0.1
dc_shell-t> echo "Freq =" [expr $freq * 1000] "MHz"
Freq = 100.0 MHz
dc_shell-t> set_load [expr [load_of cba_core/and2a0/A] * 5] \
[all_outputs]
```

注意 expr 两边的空格。

3.2.5.5 DC-Tcl 中的列表

列表是一般 Tcl 语言的组成部分，它是一串元素的集合，中间以空格分开。列表相当于 C 语言中的字符串。如下面的语句定义了一个 L1 列表，包含 3 个元素，echo \$L1 语句返回所有的元素值，llength 用于返回这个列表的长度。

```
dc_shell-t> set L1 {el1 el2 el3}
el1 el2 el3
dc_shell-t> echo $L1
el1 el2 el3
dc_shell-t> set Num_of_List_Elements [llength $L1]
3
```

另外 lappend 语句用于给列表附加新的元素——

```
dc_shell-t> set link_library {*}
*
dc_shell-t> lappend link_library tc6a.db opcon.db
* tc6a.db opcon.db
dc_shell-t> echo $link_library
* tc6a.db opcon.db
```

3.2.5.6 DC-Tcl 中的集合

集合(Collection)是 DC-Tcl 特有的数据类型，它主要用于描述有多个属性的元素的集合，它是为了描述硬件而设计的。例如电路中的端口(ports)，具有输入/输出、驱动单元、最大电容等属性，电路中的设计(designs)，具有面积、工作条件、最大面积等属性。要描述这些具有属性的元素的集合，列表是不能胜任的。因此引入了集合的概念，它相当于 C 语言的指针。每一个集合都有一个集合柄(collection handle)，相当于指针指向的地址。

我们可以通过”get_”语句和”all_”语句创建一个集合，DC-Tcl 用到的创建集合的命令有如下一些(部分)——


```

get_cells      # Create a collection of cells
get_clocks     # Create a collection of clocks
get_designs    # Create a collection of designs
get_libs       # Create a collection of libraries
get_nets       # Create a collection of nets
get_pins       # Create a collection of pins
get_ports      # Create a collection of ports

all_clocks     # Create a collection of all_clocks
all_designs    # Create a collection of all_designs
all_inputs     # Create a collection of all_inputs
all_outputs    # Create a collection of all_outputs
all_registers  # Create a collection of all_registers


```

为了便于和 list 对比，下面举了一个例子——

```

dc_shell-t> set mylist {a b hello world}
a b hello world
dc_shell-t> llength $mylist
4
dc_shell-t> set foo [all_inputs]
{"Clk", "Reset", "Crnt_Instrn[31]", ... "Crnt_Instrn[0]"}
dc_shell-t> llength $foo
1
dc_shell-t> echo $foo
_sel5
dc_shell-t> sizeof_collection $foo
34
dc_shell-t> query_objects $foo
{"Clk", "Reset", "Crnt_Instrn[31]", ... "Crnt_Instrn[0]"}

```



可以看到，这里先创建了一个 mylist 的列表和 foo 的集合，llength 返回的 foo 的长度为 1，echo \$foo 返回的是集合柄(_sel5)，这些是与列表不一样的。我们可以用 sizeof_collection 类似 llength，用 query_objects 类似 echo。

在 DC-Tcl 中查找所有包含有 collection 的命令，可以得到——

```

dc_shell-t> help *collection*

add_to_collection      # Add object(s)
compare_collections    # compares two collections
copy_collection        # Make a copy of a collection
filter_collection      # Filter a collection, resulting
                        # in a new collection
foreach_in_collection  # Iterate over a collection
index_collection       # Extract object from collection
remove_from_collection # Remove object(s) from a
                        # collection
sizeof_collection      # Number of objects in a
                        # collection
sort_collection        # Create a sorted copy of a
                        # collection

```

其中 remove_from_collection 前面章节已经接触过，它表示从集合中删去元素；相反的，在集合中增加元素的命令是 add_to_collection，下面是一个例子——


```

# Constrain a design for timing, using a time budget

set CLK_PER          10.0; # clock period (ns)
set time_budget      40.0; # percentage of clock period
                        # allowed for input/output logic

# calculate intermediate variables
set IO_DELAY [expr ( (1-$time_budget/100.0) * $CLK_PER)]

set all_except_clk [remove_from_collection \
                    [all_inputs] [get_ports CLK] ]

# constrain the design for timing
create_clock -period $CLK_PER -name MY_CLOCK [get_ports Clk]

set_input_delay $IO_DELAY -max -clock MY_CLOCK \
                $all_except_clk

set_output_delay $IO_DELAY -max -clock MY_CLOCK \
                [all_outputs]

```

这个例子灵活的使用了变量定义了时钟周期和时序预算，这样的好处是参数化，便于以后修改。另外还定义了 IO_DELAY 变量和 all_except_clk 集合。请大家仔细观察上面的约束脚本的写法，并争取能在自己的脚本中引入上述方法。

另外一个比较重要的命令是 filter_collection，它相当于一个过滤器，找出符合要求的元素形成一个新的集合。例如——

```
filter_collection [get_cells *] "ref_name == AN2"
```

从所有的 cell 的集合中过滤出引用名为 AN2 的一个，返回值为一个新的集合

```
filter_collection [get_cells *] "is_mapped == true"
```

从所有的 cell 集合中过滤出映射过的 cell，返回值为一个新的集合。

除了 filter_collection 之外，-filter 选项也能完成相同的任务——

```
get_cells -filter "@dont_touch == true"
```

创建一个设置了 don't_touch 属性的 cell 的集合

```
set fastclks [get_clocks -filter "@period < 10"]
```

创建名为 fastclks 的 clock 的集合，它的周期属性小于 10 (ns)

3.2.6 高级时钟约束

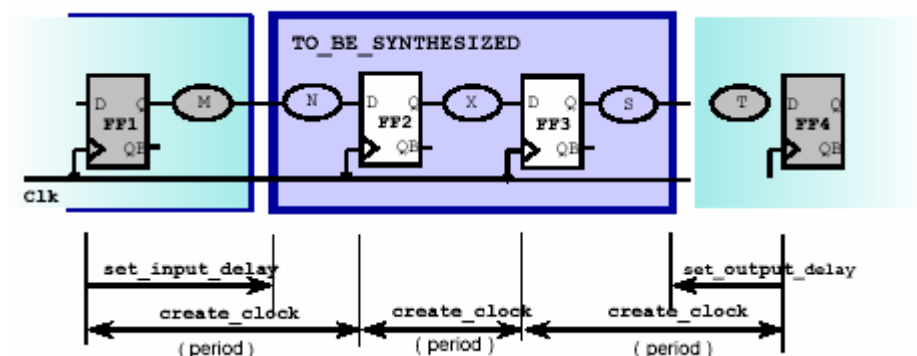


图 82 时序约束回顾

在 3.2.1 时序和面积一节中，我们讨论了一些简单的时序约束，如定义时钟，设置模块的输入输出延时等等（如上图），但是这些时序约束都是简单的约束，离较大规模芯片的实际工作条件还有一定的差距，比如时钟信号只有一个，并且周期严格遵守给定的值。在这一节里，我们着重讨论时钟约束的下面几个问题——

- 非理想的单时钟网络
- 同步多时钟网络
- 异步多时钟网络
- 多周期路径

3.2.6.1 非理想的单时钟系统

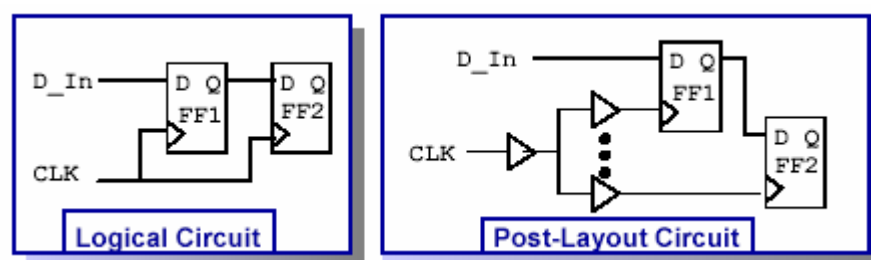


图 83 实际的时钟网络（时钟树）

前面提到过，在定义时钟之后，都要给该时钟设置一个 `dont_touch`，这是告诉 DC 不要给时钟网络作综合，因为综合时钟网络需要考虑单元的实际物理位置，这是前端的逻辑综合目前不能完成的工作。如上图所示，实际的时钟网络也称为时钟树，它在时钟的各条路径上产生大小不一的 buffer，目的是为了保证时钟到达每个触发器的延时尽量相等。

虽然 DC 无法最终综合时钟树，但是我们可以加入一些约束让此时的时钟更加接近实际的工作情况。例如，实际的时钟达到各个触发器的时间不是一样的，它们之间有一个偏差，称为时钟偏差(Clock Skew)，为了反映这个偏差，我们在综合的时候可以用一个命令来模拟

它，即 `set_clock_uncertainty`，下面是一个例子——

Example:

```
create_clock -period 10 [get_ports CLK]
set_clock_uncertainty 0.5 [get_clocks CLK]
```

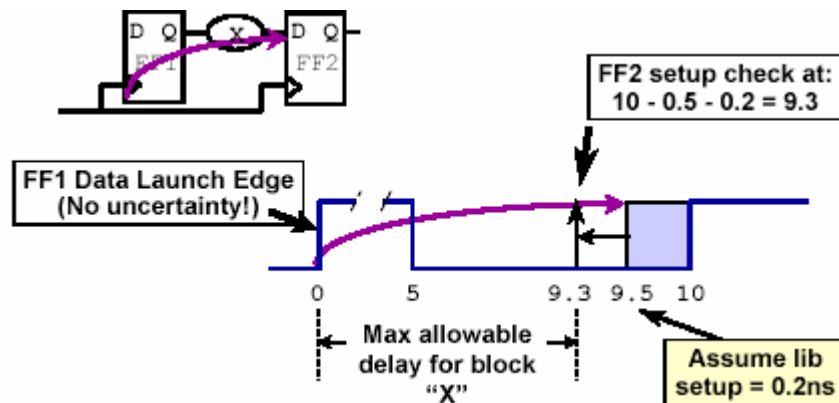


图 84 模拟时钟偏差

假设时钟周期是 10ns，FF2 的建立时间为 0.2ns，预先估计时钟偏差为 0.5ns，从 FF1 触发的数据必须在一个周期之内到达 FF2，当引入时钟偏差以后，所谓的一个周期就不再是 10ns，而可能最短为 $10 - 0.5 = 9.5\text{ns}$ ，再减去 0.2 的建立时间，实际留给 X 路径的延时最大只能有 9.3ns。

除了时钟偏差(Clock Skew)之外，还有两个命令值得注意，这就是 `set_clock_latency`，和 `set_propagated_clock`，如下图所示——

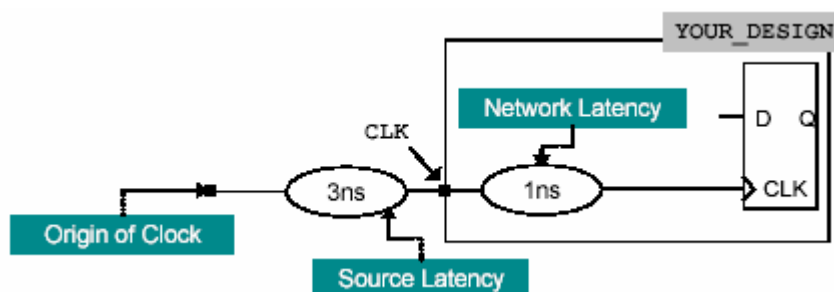


图 85 时钟延时

一般而言，时钟都是由一个专门的模块(Clock_gen)来生成的，这里称为时钟源(Clock Source)，时钟产生之后，必定要经过一段网络延时才能到达被综合的模块，这段延时称为时钟源延时(Source Latency)，到达模块的端口后，要到达内部的触发器，也要经过一定的延时，这个延时称为网络延时(Network Latency)。这分别通过 `set_clock_latency -source` 和 `set_clock_latency` 来描述。另外，`set_propagated_clock` 主要用在布局之后(post-layout)的综合上，意思是说，此时的网络延时已经可以由时钟树上的 buffer 确切的推断出来。布局之前和布局之后描述时钟的命令对比如下图——

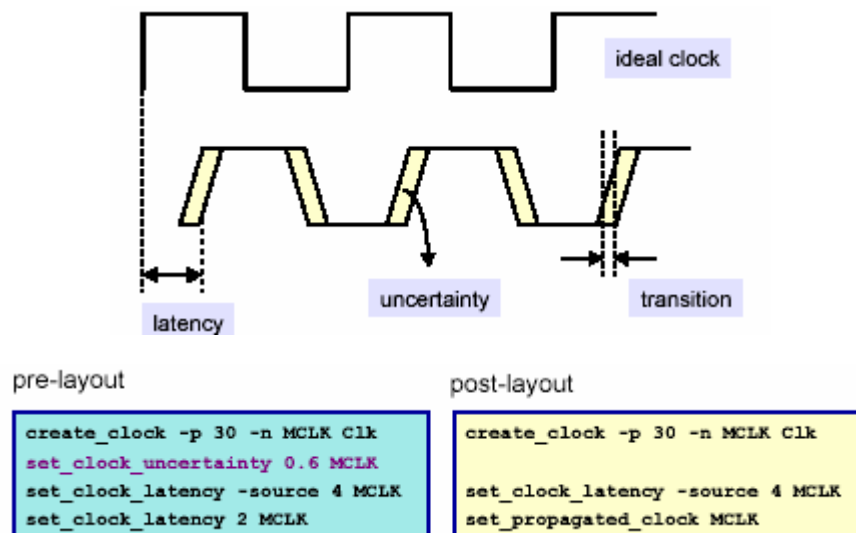


图 85 布局前后时钟描述对比

在布局前正因为没有 buffer，才需要用网络延时和时钟偏移来模拟布局后的情况。

3.2.6.2 同步多时钟网络

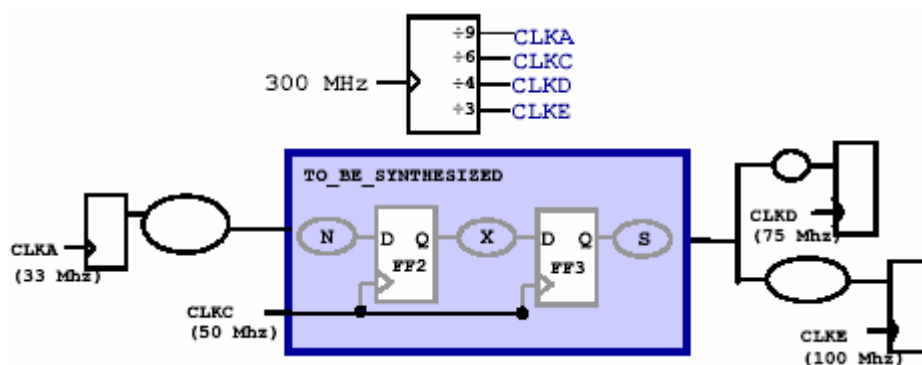


图 86 同步多时钟网络

上图是一个同步多时钟网络，中间的模块是我们综合的模块，内部只有一个 CLKC，但是输入和输出都是由不同周期的时钟控制的，也就是说，它们属于不同的路径组(Path Group)，但是这些时钟 CLKA-CLKE 都是从一个时钟分频得到的，因此称为同步多时钟。

观察被综合模块的输入端口，它同时受两个时钟的约束，由于它们周期不同，所以 CLKA 触发的信号到达 FF2 的时间也不是固定的。

对于这样的时钟网络，我们需要用到虚拟时钟的概念(Virtual Clock)。对上图要综合的模块而言，除了 CLKC 之外的其他时钟都可以称为虚拟时钟，它们有如下要求——

- 在顶层模块之内的其他模块内定义的时钟
- 在当前的被综合模块(current_design)内不包含虚拟时钟驱动的触发器
- 作为当前模块的输入输出延时参考

定义虚拟时钟和定义时钟的命令差不多，只是不要指定虚拟时钟的端口或者管脚，另外必须指定时钟的名字——

```
create_clock -name vTEMP_CLK -period 20
```

Must be named No source pin or port!

如上述语句指定了一个周期为 20ns 的虚拟时钟 vTEMP_CLK。

定义了虚拟时钟之后，我们就可以描述图 86 的电路时序了，先设定输入延时 (set_input_delay)，从图中看出，CLKA 和 CLKC 的周期分别为 30ns 和 20ns，假设 IN1 端口的输入延时为 5.5ns，可以设定如下——

```
create_clock -period 30 -name CLKA
create_clock -period 20 [get_ports CLKC]
set_dont_touch_network [get_clocks CLKC]

set_input_delay 5.5 -clock CLKA -max [get_ports IN1]
```

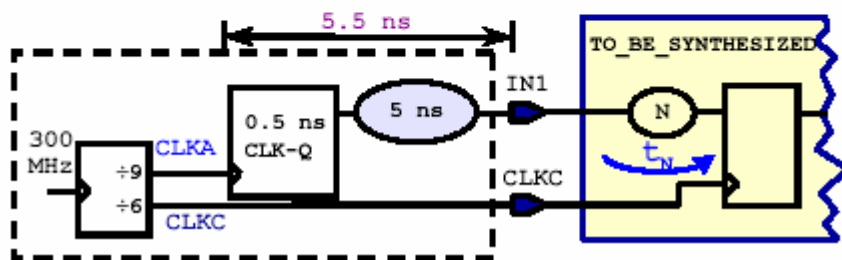


图 87 同步多时钟的输入延时

值得注意的就是先定义虚拟时钟 CLKA，然后在 set_input_delay 的驱动时钟开关中选择 CLKA。在设定完输入延时之后，Design Compiler 就会在 CLKA 和 CLKC 的所有情况中找到其中最短的周期，作为对这段路径的约束——

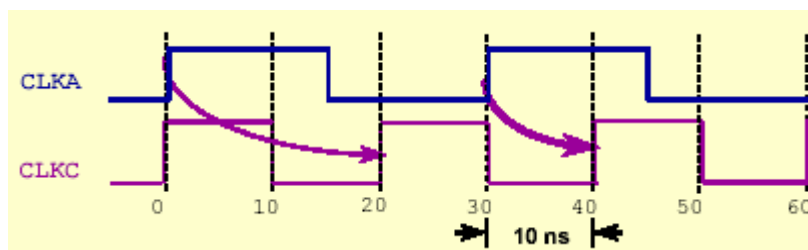


图 88 寻找输入最小周期

寻找最小周期的过程如下：先计算 CLKA 和 CLKC 的最小公约数(30 和 20 的公约数)为 60ns，即两个 CLKA 的周期，然后分别以这两个上升沿为触发沿，计算此时的最短捕捉(被 CLKC 接收)的时间，最后对比这两个时间，取其中最小的一个。如下图计算出的最短捕捉时间为 10ns，因此留给路径 N 的延时为——

$$t_N < 10 - 5.5 - t_{\text{setup}}$$

按照上面同样的方法，我们也可以得出图 86 的输出延时(set_output_delay)，如下图所示

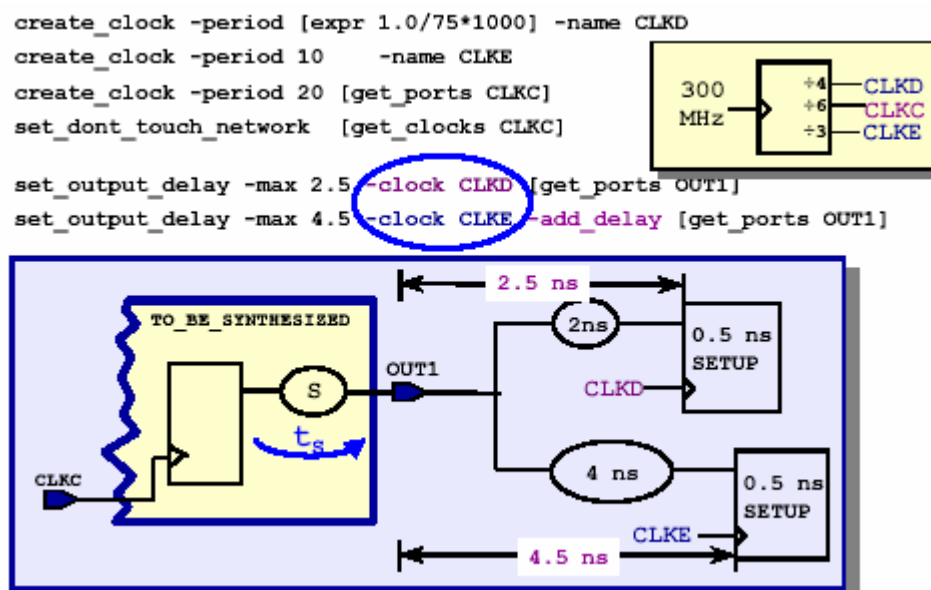


图 89 同步多时钟的输出延时

值得注意的是：这里的输出同时驱动两个虚拟时钟电路 CLKD 和 CLKE，因此描述第二个延时的时候需要加入-add_delay 的开关,否则将覆盖前一条路径的约束。

这里的各时钟波形关系如下所示——

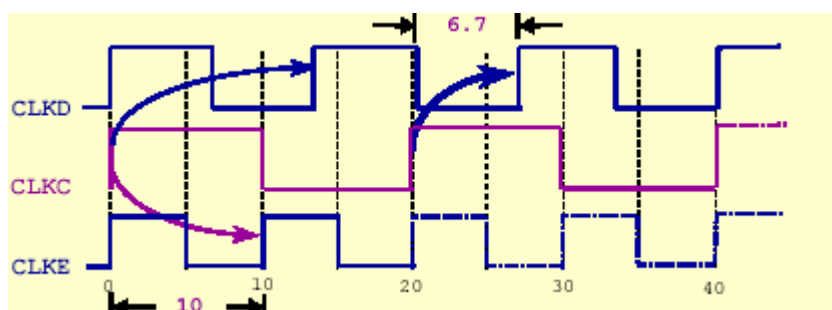


图 90 寻找输出最小周期

可见，CLKC 到 CLKD 的最小捕捉时间为 6.7ns，到 CLKE 的最小捕捉周期为 10ns。因此输出最小周期为 6.7ns 和 10ns，对应的输出电路的延时必须满足下面的条件

$$t_s < 10 - 4.5 \quad \text{AND} \quad t_s < 6.7 - 2.5$$

从前面的推导过程可以看出，DC 是在各个相关时钟周期的最小公约数的基础上计算最小输入/输出时间的，因此我们在定义时钟的时候尽量选用整数，不要加上小数点(比如 20ns 和 30.1ns)，避免不必要的麻烦。

3.2.6.3 异步多时钟网络

异步多时钟网络和同步多时钟网络的结构类似，只是它的各个时钟 CLKA-CLKE 不是从同一个时钟源中分频产生的，而可能是不同的两个晶振，如下图所示——

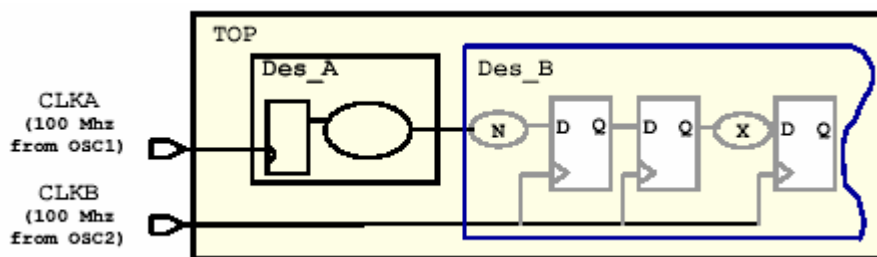


图 91 异步多时钟网络

由于是不同的晶振产生的时钟，它们之间的就不存在最小公约数的关系，但是在默认情况下，DC 并不知道，它会认为它们是同步的时钟网络而尽量去找两个时钟之间的最小捕捉时间，不但浪费了时间而且会产生出不符合要求的电路。在这种情况下，我们需要告诉 DC 不要管两个时钟之间路径的时序，这里需要用到一个命令——set_false_path。

False Path（伪路径）是指电路中的一些不需要考虑时序约束的路径，它一般出现在异步逻辑之中。上面的例子设置伪路径的语句如下——

```
current_design TOP

/* Make sure register-register paths meet timing */
create_clock -period 10 [get_ports CLKA]
create_clock -period 10 [get_ports CLKB]

/* Don't optimize logic crossing clock domains */
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
set_false_path -from [get_clocks CLKB] -to [get_clocks CLKA]

compile
```

这样，所有的从 CLKA 到 CLKB 和 CLKB 到 CLKA 的路径都被认为是伪路径。

3.2.6.4 多周期路径

在前面的讨论中，我们默认所有组合路径的延时都是一个周期(如图 91 中的 N 和 X 路径)，然而实际电路中也可能存在超过一个周期的路径，如下图所示——

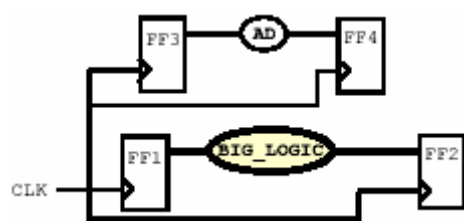


图 92 多周期路径

在 FF1 和 FF2 之前，存在一个 BIG_LOGIC，假设我们允许它的延时在两个周期之内，那么因该怎样把这个信息告诉 Design Compiler 呢？这里就需要用到 DC 的一个设置多周期路径的命令——set_multicycle_path

```
set_multicycle_path 2 -setup -from [get_cells FF1] -to [get_cells FF2]
```

```
set_multicycle_path 1 -hold -from [get_cells FF1] -to [get_cells FF2]
```

第一个语句说明建立时间是在 FF1 触发后的第二个周期后检查，第二个语句说明保持时间在 FF1 触发后的第一个周期检查。可得此时的波形图如下——

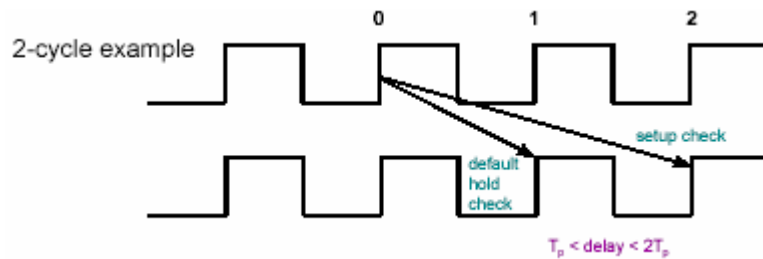


图 93 多周期路径波形

3.3 设计综合过程

在前面一章介绍完施加约束之后，接下来要做的工作就是将设计进行综合编译 (compile)，这一章我们将主要讨论综合编译的过程。这一章主要分为这样几个部分——

- 优化的三个阶段及其特点
- 编译的策略
- 编译层次化的设计

3.3.1 优化的三个阶段

这一节我们介绍 Design Compiler 进行优化的三个阶段：结构级、逻辑级以及门级，在不同的阶段，DC 运用的方法和优化余地是不一样的，通过这一节的学习，你将对这几个阶段的特点和优化方法有一个比较清楚的了解，其中有一些方法我们在前面的章节中已经提及过，这里一起归纳一下，希望能加深大家的认识。

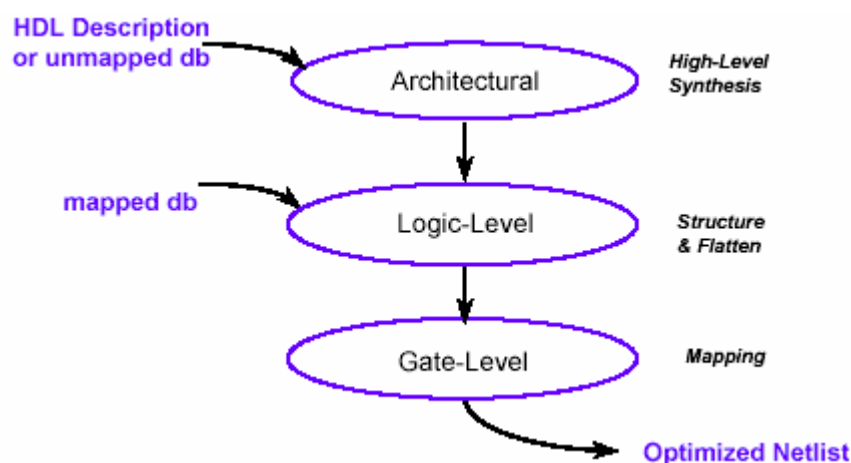


图 94 DC 优化的三个阶段

上图是这三个阶段的关系图，可以看到，结构级属于最高的抽象层次，当读入 Verilog 代码或者没有经过映射的 db 文件后，DC 的优化从这个阶段层次开始，可以说，结构级是优化的最高层次，所以对 DC 来说，这个层次的综合可以称为高层次综合(High-Level Synthesis)。结构层的下一个优化阶段是逻辑级阶段，也是读入映射过的 db 文件的 DC 的初始优化阶段。再往下一个阶段是门级阶段，也是优化的最后阶段，这里所要作的工作主要就是 GTECH 到工艺库的映射。

3.3.1.1 结构级优化

因为结构级是优化过程中层次最高的一级，因此它也是 DC 可以采用的优化方法最多的

一级，它的主要优化方法如下图所示——

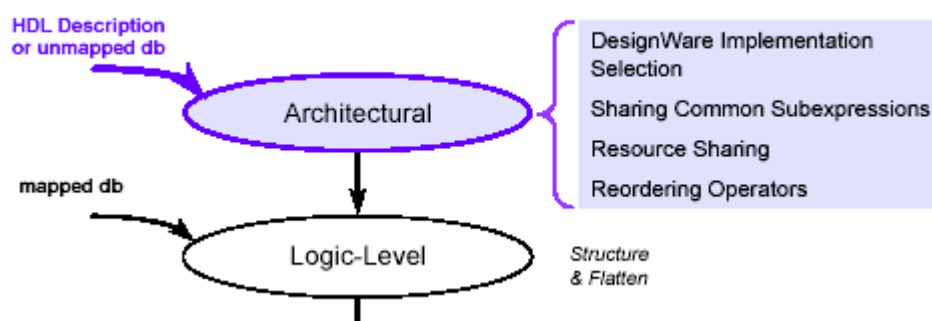


图 95 结构级优化

1. DesignWare 选择

DW 选择是结构级优化的一个很主要的特点，在这个阶段 DC 能够根据设计者施加的时序或者面积的约束在 DW 的不同实现方式中找到它认为最佳的实现方案。比如加法器的实现方式一共有如下几种——

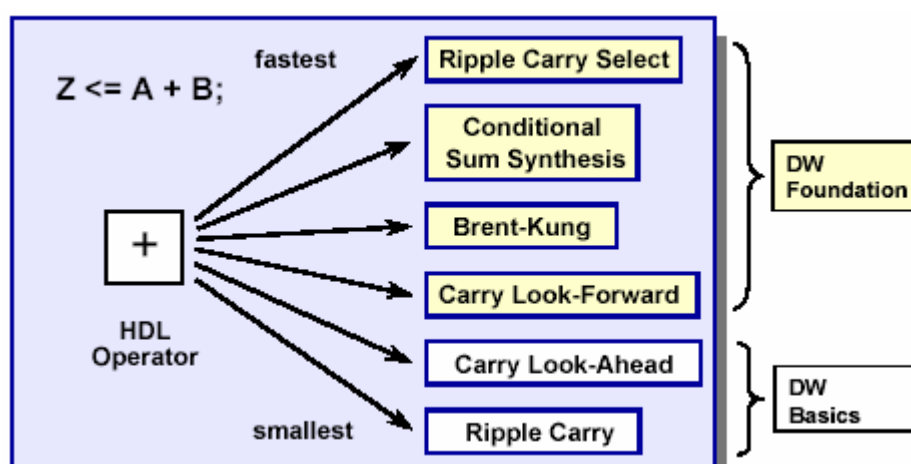


图 96 DW 选择

其中 DW Foundation 需要有专门的 license，而且使用之前还要设置综合库(synthetic library)。

2. 共享子表达式(Sub-Expressions)

这里的子表达式主要是指数学表达式，比如下面这个例子，如果按照原来的语句综合，会包含 6 各加法器，但是如果表达式之间的公共项提取出来，便可以大大的减小面积，如下图 ——

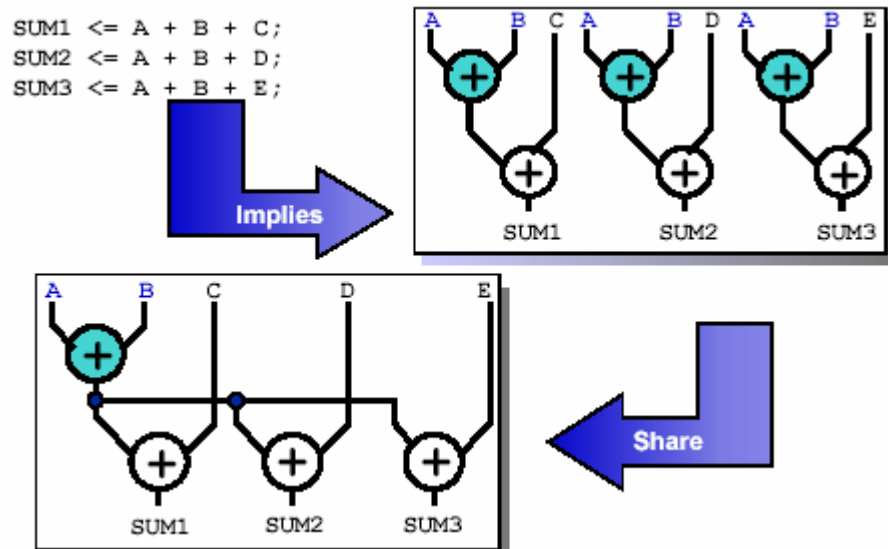


图 97 共享子表达式

如果要直接综合出共享后的电路 ,可以在编写 RTL 代码的时候强制指定共享项——

```
temp := A + B;
SUM1 <= temp + C;
SUM2 <= temp + D;
SUM3 <= temp + E;
```

3. 资源共享(Resource Sharing)

资源共享的原理与共享子表达式类似，只不过这里指的所谓资源是一些 HDL 的运算符和表达式，比如加(+)、减(-)、乘(*)、除(/)以及大于(>)、大于等于(>=)、小于(<)、小于等于(<=)。这里举的例子，在前面的章节里也见过，比如给定一个语句——

```
if (SEL = '1') then
    SUM <= A + B;
else
    SUM <= C + D;
end if;
```

它可能有下面两种电路实现——

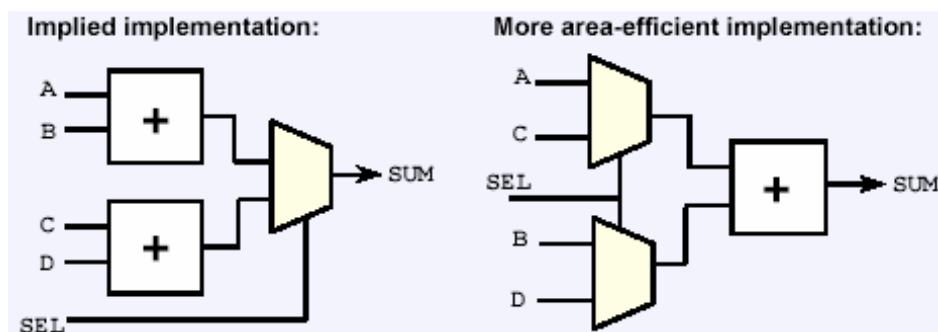


图 98 资源共享

DC 会根据具体的约束条件综合出最符合要求的结构来。

4. 运算符排序(Operator Reordering)

对于下面这个表达式 $Z \leq A + B + C + D$ (输出 Z 是施加了一定时序约束), DC 最初是按照从左至右的顺序计算的, 也就是说它最初的排序如下——

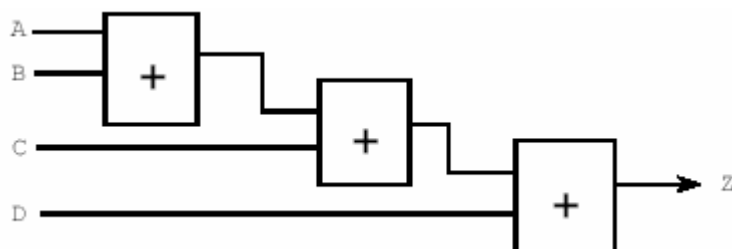


图 99 运算符排序(1)

如果几个输入信号到达的时间相同, DC 会通过运算符排序优化成下图的平衡的结构, 减小延时——

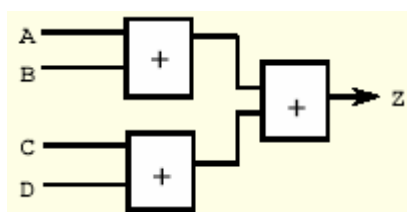


图 100 运算符排序(2)

如果 A 信号较迟到达, 则综合出的电路结构会如下——

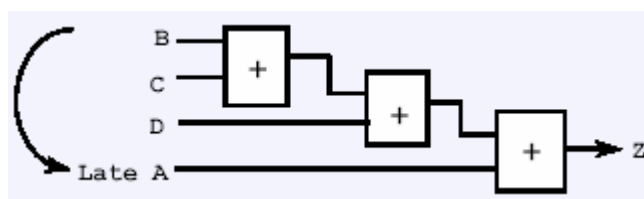


图 101 运算符排序(2)

3.3.1.2 逻辑级优化

在经过结构级优化之后, 电路被转化成了工艺无关的 GTECH 库的形式, 这级也称为逻辑级, 对于逻辑级优化来说, 只有一个方法, 那就是——结构化 (structure) 或者扁平化 (flatten)。

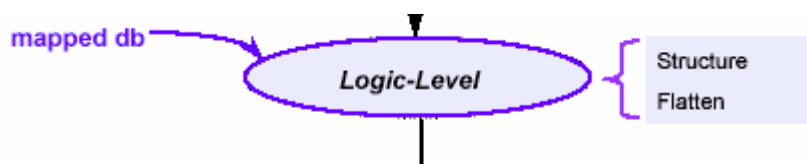


图 102 逻辑级优化

1. 结构化(structure)

结构化是 DC 在逻辑级的默认的优化方法，它是指：使用电路的一些中间项构成一个多级的电路结构。如下图的电路一共有三级逻辑，下一级的输入是上一级的输出，使用这种优化方法一般情况下能综合出兼顾时序和面积的电路来。

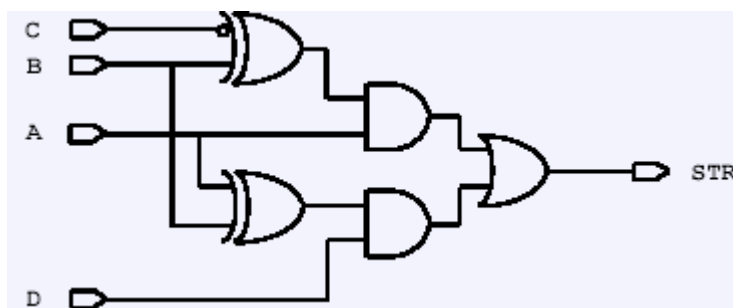


图 103 结构化电路

结构化电路的典型是奇偶校验电路。

2. 扁平化(flatten)

扁平化是将所有的组合逻辑打平成乘积项和(SOP)的两级结构，类似与可编程阵列逻辑(PAL)。使用这种结构的特点由于没有利用中间项，综合后电路面积将会变得很大，但是却不一定能取得较好的时序。

扁平化结构的电路和设置扁平化的 DC 命令如下所示——

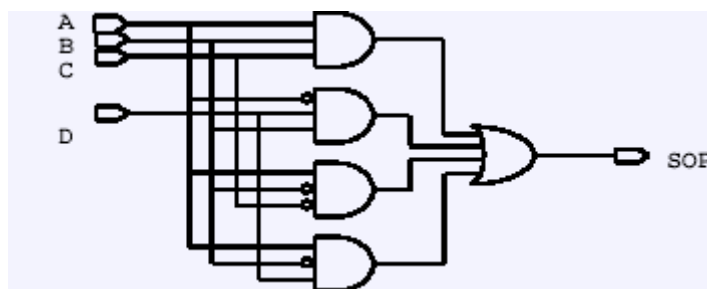


图 104 扁平化电路

```
set_flatten true -effort low | medium | high
```

综合结构化和扁平化的特点，可以归纳如下——

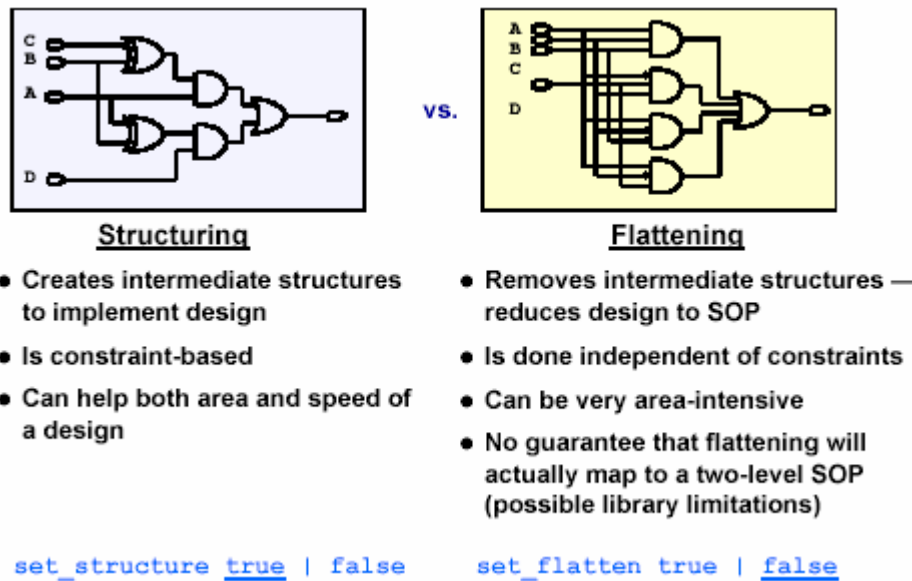


图 105 结构化 Vs 扁平化

Good candidates for SOP flattening	Bad candidates for SOP flattening
State machines	Arithmetic components
Random logic	XOR structures (parity trees)
Blocks that use don't-cares	Blocks with many muxes

由于 DC 默认是用结构化的方式综合逻辑级电路，而且这种方式可以得到兼顾时序和面积的结果，因此我们可以先用这种方式优化。在优化后的电路中找到关键路径，看看关键路径上有没有符合使用 SOP 电路的模块，再将这些方便使用 SOP 的模块 `set_flatten`，以便取得最佳的效果。

3.3.1.3 门级优化

门级优化是优化的最后阶段，它所要完成的任务就是将 GTECH 的电路映射到最终的工艺库中，并且保证映射后的电路不违反设计规则(Design Rule)。

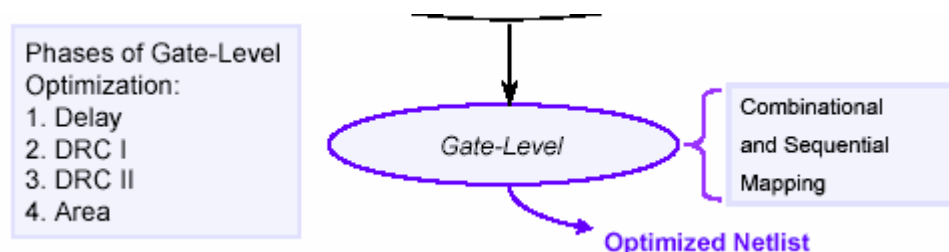


图 106 门级优化

1. 工艺映射

工艺映射包括组合逻辑映射和时序逻辑映射。组合逻辑映射是指 DC 使用工艺库中

的各种门替换 GTECH 单元，并选择能实现相同逻辑的符合时序及面积要求的单元——

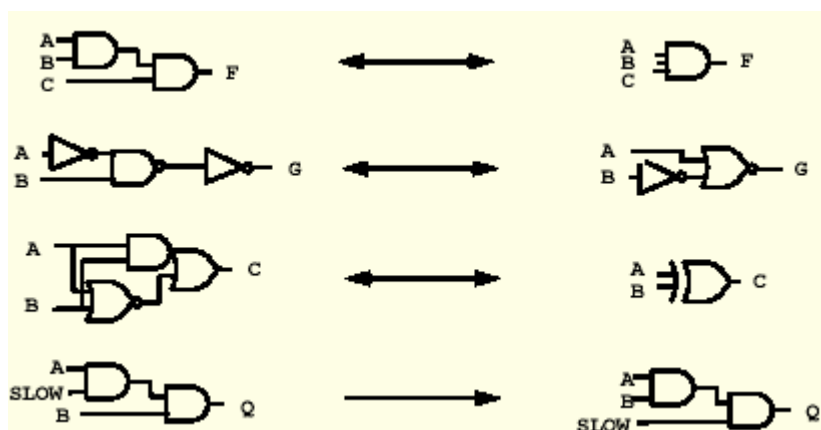


图 107 组合逻辑映射

时序逻辑映射的方法和组合逻辑相类似，也是出于速度和面积的考虑，**尽量使用复杂的时序单元吸收一部分组合逻辑。**

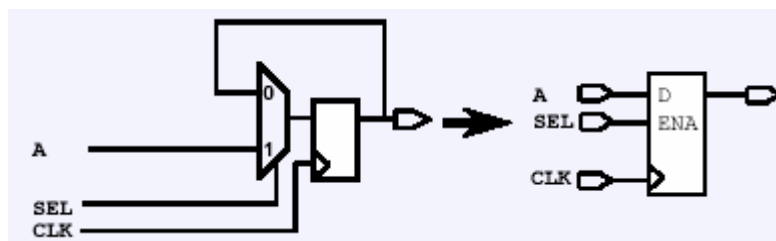


图 108 时序逻辑映射

2. 设计规则检查(DRC)

对于工艺库的单元而言，Foundry 都指定了每个单元的工作条件的限制，比如最大电容(max_capacitance)等等，这些限制也可以称为设计规则(Design Rule)，在设计规则限定的范围内，Foundry 提供的参数才有实际的意义。比如一个单元允许的最大电容为 5pf，而实际工作电路中出现的电容值为 10pf，那么这时，便违反了设计规则，单元的参数也就不能确保是准确的了。

因此，DC 在作门级优化的时候，在映射的过程中也会检查电路的设计规则，一般的做法是在单元中插入 buffer 增加驱动能力，或者将小驱动的单元替换为大单元。设计规则检查分为两个过程——DRC I 和 DRC II。(如图 106 所示)

DRC I 是指 Design Compiler 在不影响电路的时序和面积的前提下修正违反规则的一些单元，如果在这个前提下不能完全修正，则要进行下一步的检查，即 DRC II，这一步的修正必然是以牺牲一部分时序和面积为代价的。

3.3.2 编译策略

编译过程是指设计经过三个阶段的优化，最终形成门级网表的过程，在这一节里，我们主要就编译的策略，它包含如下几方面的内容——

- 中断编译的方法
- 从报告中检查时序，调整策略
- 修正保持时间违反(Hold time violations)

3.3.2.1 中断编译的方法

在 DC-Tcl 的界面下，当我们键入 compile 命令时，DC 就开始了编译，也就是优化的过程。优化是在设计规则的条件下，运用不同的算法，综合最终出满足时序和面积的电路。优化首先是时序驱动(timing-driven)的一个过程，其次再是面积。如果找到了一个满足时序和面积等约束的电路，编译将会停止；如果通过种种编译仍不能满足时序，编译也会停止下来；另外，我们也可以人为的中断编译。

人为中断编译的方法是键入 Ctrl-C，经过一段时间的等待后（有可能时间会很长），优化过程暂停，并弹出如下菜单——

```
Please type in one of the following option:
 1 to Write out the current status of the design
 2 to Abort optimization
 3 to Kill the process
 4 to Continue optimization
Please enter a number:
```

这里有四个选项，设计者可以根据情况作出选择。

DC 在编译的过程中，会自动打印出一个报表，报告编译的总时间，设计的面积，关键路径的时序违反和总共时序违反情况，我们可以根据需要更改打印的列项目——

Beginning Delay Optimization Phase

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:10:04	2761.7	1.38	3.20	18.1	Zro_Flag_reg/D
0:10:05	2761.7	1.38	3.20	18.1	Zro_Flag_reg/D
0:10:08	2761.7	1.28	3.10	18.1	Zro_Flag_reg/D
0:10:12	2761.7	1.26	3.06	18.1	Zro_Flag_reg/D

Critical Path timing violations (points to WORST NEG SLACK)

Sum of all timing violations (points to TOTAL NEG SLACK)

图 109 编译报表

3.3.2.2 分析报告，调整策略

一般情况下，我们先作一个默认的编译，这样一般可以取得既快又准确的结果，然后在编译完成后使用一些报告时序的命令，并分析它们的输出结果，使用的命令主要有——

1. `report_constraint -all_violators`

报告电路中所有没有满足的约束条件，包括设计规则、建立时间、保持时间以及面积。通常这应该是最先执行的命令。

2. `report_timing -delay max`

报告基于建立时间检查的关键路径，每一个路径组的关键路径都被报告出来。

3. `report_timing -delay min`

报告基于保持时间检查的关键路径，每一个路径组的关键路径都被报告出来。

从这些报告中，我们可以看到电路中是否有违反的约束，如果有，那么它是什么类型，还有电路中的最大负裕量(worst negative slack)是多少，等等。下面我们就几个常见的约束违反情况，谈谈纠正它的综合策略——

较大的时序违反

请看下面这个例子——

```
dc_shell> report_constraint -all
Information: Updating design information... (UID=85)

*****
Report : constraint
        -all_violators
Design : RISC_CORE
Version: 1999.05
Date   : Thu Nov 11 09:38:42 1999
*****
max_delay/setup ('Clk' group)
```

Endpoint	Required Path Delay	Actual Path Delay	Slack	
RESULT_DATA[1]	1.20	2.84 r	-1.64	(VIOLATED)
RESULT_DATA[2]	1.20	2.84 r	-1.64	(VIOLATED)
RESULT_DATA[8]	1.20	2.84 r	-1.64	(VIOLATED)
RESULT_DATA[14]	1.20	2.84 r	-1.64	(VIOLATED)
RESULT_DATA[5]	1.20	2.84 r	-1.64	(VIOLATED)
RESULT_DATA[11]	1.20	2.84 r	-1.64	(VIOLATED)

A rather big violation

图 110 较大时序违反

从 `report_constraint -all` 这个命令的报告可以看出，需要到达的时间是 1.20ns，而实际到达为 2.84ns，违反了 1.64ns。之所以判断它是一个较大时序违反的情况，并不是因为 1.64 这个绝对值很大，而是相比较需要时间而言，1.64 是一个较大的值。一般而言，如果电路中的最大负裕量(简称 WNS)所占时钟周期的 15% 以上的话，可以认为电路存在较大的时序违反。

确认存在较大时序违反之后，下一步就是找出原因，消除违反情况。可供参考的步骤有下面几种——

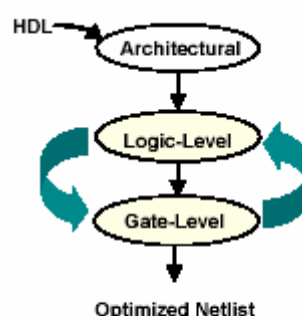
1. 检查约束条件，看是否有疏漏或错误
2. 检查模块划分，看组合逻辑是否穿过多个模块
3. 重新编译优化后的网表
4. 修改 RTL 代码

下面详细讨论后面的三种情况——

- 重新编译(Re-Compile)

当重新读入映射后的网表进行重新编译时，DC 会自动将门级的网表重新返回到 GTECH 的结构，相当于逻辑级。然后分别进行逻辑级和门级的优化，但是同时也可以进行 DesignWare 的替换。

如果设计者仅仅将映射后的网表拿来再做一次 compile，编译后的结果并不会不一定会比原来的好，无非把以前做过的优化再跑一遍。因此，重新编译之前会改变一些参数，如——改变设计约束、改变 `set_structure` 和 `set_flatten` 参数以及改变编译的 `map_effort`。



```
dc_shell-t> compile
```

图 111 重新编译

- 改变 `map_effort` 重新编译

对设计进行编译的时候，有三种编译级别可以选择，它们分别是低级、中级和高级。

```
compile -map_effort (low | medium | high)
```

不同的级别编译要求的编译时间和编译结果都各不相同，`compile -map_effort low` 编译时间最短，但是结果不一定好，它一般用于设计预估(Design Exploration)，不用在重新编译环节。

`compile -map_effort medium` 是 DC 默认的编译级别，大多能在一定的时间内得到较为满意的结果。这也是我们推荐的初始编译级别。

`compile -map_effort high` 编译的过程中会使用前面的级别中没有的算法，因此它所要求的时间是最多的，结果也是相对最好的。这种级别一般用在重新编译的阶段。

- 修改 RTL 代码

修改源代码所能取得的效果是最直接的，同时也是代价最高的。修改代码后，DC 会从最上层的结构级开始优化，前面也讨论过，越上层次的优化方法越多。所以通常这样得到的结果也越满意。但是，修改代码也不一定放之四海皆准的方法，因此并不是所有的设计我们都能获得源代码，同时也不是可以随便修改的。

较小的时序违反

请看下面这个例子——

```
dc_shell> report_constraint -all
Information: Updating design information... (UID-85)
```

```
*****
Report : constraint
        -all_violators
Design : RISC_CORE
Version: 1999.05
Date   : Thu Nov 11 09:38:42 1999
*****
max_delay/setup ('Clk' group)
```



Assuming your constraints and partitions are correct, what should you do?

Endpoint	Required Path Delay	Actual Path Delay	Slack
RESULT_DATA[1]	1.20	1.30 r	-0.10 (VIOLATED)
RESULT_DATA[2]	1.20	1.26 r	-0.06 (VIOLATED)
RESULT_DATA[8]	1.20	1.26 r	-0.06 (VIOLATED)
RESULT_DATA[14]	1.20	1.22 r	-0.02 (VIOLATED)
RESULT_DATA[5]	1.20	1.22 r	-0.02 (VIOLATED)
RESULT_DATA[11]	1.20	1.22 r	-0.02 (VIOLATED)

图 112 较小时序违反

从上图看出,相比较 1.20 的允许路径延时,0.10 的负最大裕量(WNS)是比较小的(小于 15%),而且已经认定了约束和模块划分都是正确的,那么应该怎样修复这个错误呢?

这里主要讲一下 Incremental Mapping——

- **compile -incremental_mapping**

这个开关告诉 DC,在重新编译的时候不需要把网表返回到 GTECH 结构,因此也不需要作逻辑级优化,速度也较一般的编译更省时间。这里 DC 所要作的是进行门级单元的替换,即在不违反设计规则的情况下用延时小的单元替换延时较大的单元。另外,如果读入的是 db 格式的网表,在这个阶段也可以进行 DesignWare 的替换。

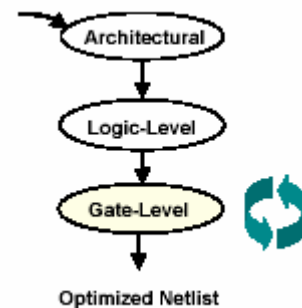


图 113 重新编译(-inc)

- **compile -inc -map high**

这里多加了一个-map high 的开关,-map high 开关前面已经提到过,它是让 DC 使用更多的优化算法优化电路,与上面的优化不同,这里需要把层次提高到结构级,如右图所示。

需要注意的一点是:这里所指的优化仅仅优化电路中的关键路径(critical path),也就是说,如果电路中的一个路径组中有多条路径违反,优化后也不可能全部满足时序。

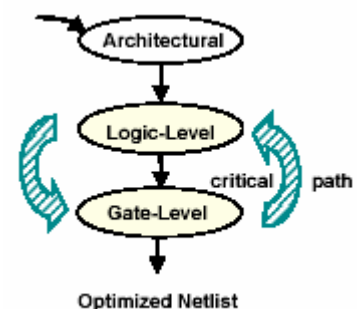


图 114 重新编译(-inc -map high)

如果要同时优化多条路径,需要使用另外一个命令——set_critical_range

这个命令设置的 `critical_range` 是以 WNS 的值为基准的，优化的是和这个值的绝对值差设置值的那些路径。因此，如果设置值为 0，那么就仅仅优化一条关键路径。

例如，假设电路中的 WNS 为 -3.4ns，如果设置了

```
set_critical_range 2 [current_design]
```

那么，当前设计中的所有负裕量的绝对值大于 1.4 的路径都将被优化掉。

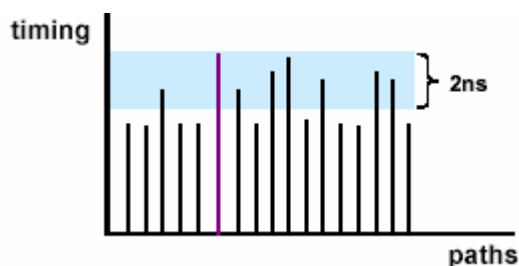


图 115 `set_critical_range`

设计规则违反

有些时候的时序违反是由于设计规则违反引起的，比如说一个单元的扇出（fanout）过大，导致它的 transition time 的时间迅速增加。对于这种情况，我们可以通过

```
report_net -connections -verbose
```

```
report_timing -net (for fanout)
```

两个命令审查连线的连接和负载情况。

要修正设计规则的错误，可以使用一个编译的开关

```
compile -only_design_rule
```

如下面这个例子，为了满足最大电容的规则，在 A 端口的内部加上了一个 buffer，用于缓冲 N 路径对 A 的负载。

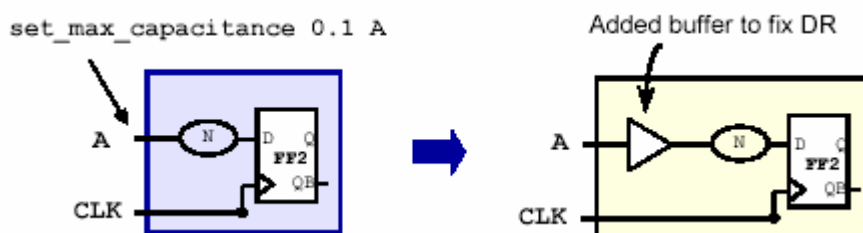


图 116 修正设计规则违反

3.3.2.3 修正保持时间违反

一个时序电路要想正常工作，除了必须满足建立时间要求之外，也需要满足保持时间要求。

保持时间的概念和设置

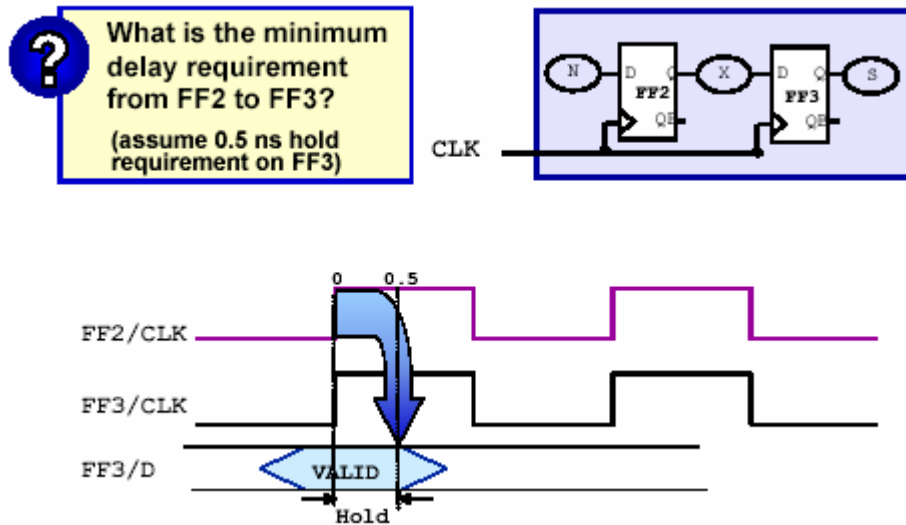


图 117 保持时间(1)

从前面的课程可以知道，为了满足建立时间，X 路径的延时加上 FF3 的建立时间必须小于 CLK 一个周期的时间。这样做可以保证从 FF2 触发的数据能在一个周期后被 FF3 捕捉到。

同样，触发器还有一个保持时间，它是指在时钟边沿过后的一段时间触发器输入必须稳定，否则就会出现数据异常。为了满足这个条件，就必须使得触发器在一个时钟边沿触发数据后必须等待一段保持时间才能接收新的数据。上图中假设 FF3 的保持时间为 0.5ns，可见，在图示的上升沿，FF2 触发新的数据，FF3 捕捉了 FF2 在前一个周期触发的数据，如果 X 路径延时足够小，那么有可能在 0.5ns 之内 FF2 触发的新数据也到达了 FF3 的输入端，这样就引起了保持时间违反。

从上面的分析不难看出，保持时间容易出现在组合路径延时较小的路径中。下面我们分析时钟偏移(Clock Skew)对保持时间的影响。

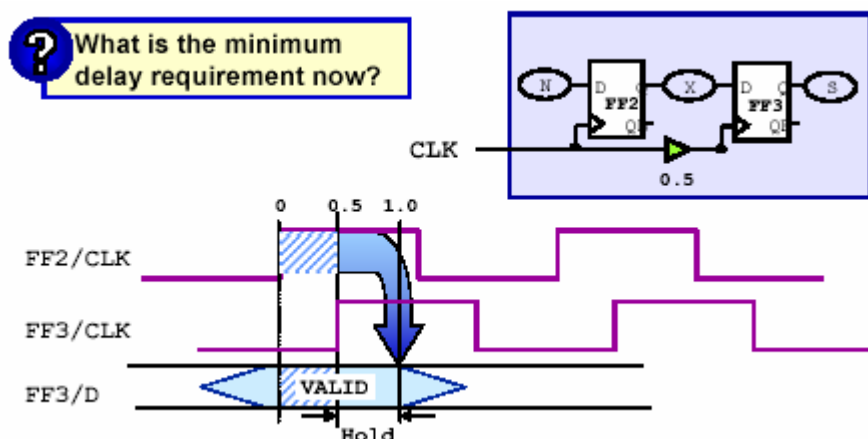


图 118 保持时间(2)

假设 CLK 到达 FF3 比 FF2 晚 0.5ns，那么从上图可以看出，FF2 的新数据到达 FF3 的可能性会比没有 Skew 增加。它现在的保持时间要求也从 0.5ns 提高到 1.0ns。

除了时钟偏移之外，工作条件也会对保持时间产生一定的影响。工作条件的变化直接影

★

响到的是各个单元(时序和组合)的延时。前面讨论过，最差情况(Worst case)下组合电路的延时最大，所以检查建立时间时用的都是最差情况。相反的，在最佳情况(Best case)情况下，组合电路的延时会变小，产生保持时间违反的可能性也增加了。

虽然保持时间检查和建立时间检查是同样重要的，但是我们在实际综合的过程中却不是把它们同时考虑，而是更多的把保持时间的检查放到布局后。这是因为——

- 时钟偏移必须要到布局完成后才能得到准确值
- 修正保持时间的通常做法是插入 buffer，而这可能会增加建立时间违反的可能性，并且增大了组合电路的面积
- 保持时间检查用的一般都是电路工作的最佳条件，而在这个条件下，连线延时往往是被忽略的，连线延时也是必须在布局后才准确

如果确定要同时作建立和保持时间检查，那么在施加电路约束的时候要加入相应的开关，比如——

```
set_clock_uncertainty -hold
set_input_delay -min
set_output_delay -min
set_operating_conditions -min -max
```

以及设置各自的工艺库——

```
set_min_library max_library -min_version min_library
```

下面详细谈一下设置保持时间的输入/输出延时——

- set_input_delay -min

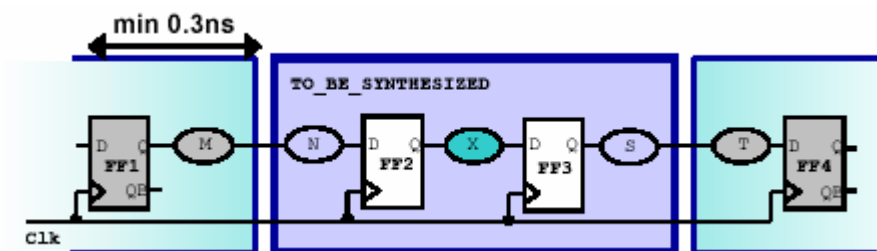


图 119 保持时间的输入延时

上面描述的是基于保持时间的输入延时，需要设置外围输入电路最快到达被综合模块输入端口的时间，假设 CLK 周期 10ns，FF2 的保持时间为 1ns，输入最小延时 0.3ns，那么可以写成——

```
create_clock -period 10 [get_ports Clk]
set_input_delay -min 0.3 -clock Clk $all_in_ex_clk
```

可以推断出，此时 N 路径必须满足的最小延时为 $1\text{ns} - 0.3\text{ns} = 0.7\text{ns}$ 。

- set_output_delay -min

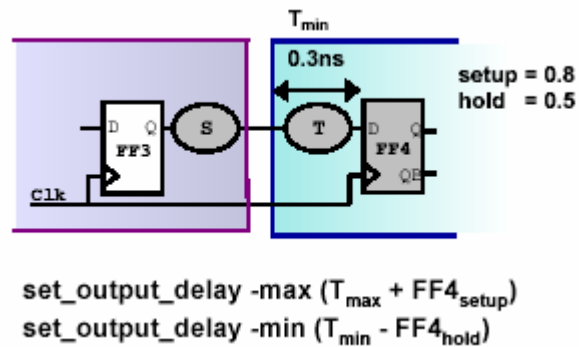


图 120 保持时间的输出延时

图中所示，FF4 是输出电路外围的一个触发器，它的保持时间是 0.5ns，T 路径的最小延时是 0.3ns，那么得到 FF3 所在电路的输出最小延时(set_output_delay -min)就要比较小心，它的计算公式不是 $0.5 - 0.3 = 0.2\text{ns}$ ，而是 $0.3 - 0.5 = -0.2\text{ns}$ 。这一点需要引起大家的注意，此时留给 S 路径的最小延时为 0.2ns。

设置最小输出延时的命令如下——

```
create_clock -period 5 [get_ports Clk]
set_output_delay -min [expr 0.3-0.5] -clock Clk \
    [all_outputs]
```

-0.2

修正保持时间违反

默认情况下，DC 不修正保持时间的违反。如果确定要作修正，需要先设置一个变量再作检查——

```
set_fix_hold [all_clocks]
compile -only_design_rule
```

加上 only_design_rule 的开关后，编译过程中仅仅更换单元大小，并增加 buffer，以便修正设计规则违反和保持时间违反

下面是一个设置保持时间约束和修正保持时间违反的脚本——


```

read_db Top_meetsSetup.db
source TimingConstraints_max.tcl

set_operating_conditions -max WORST -min BEST
set ALL_IN_EX_CLOCK [remove_from_collection \
    [all_inputs] [get_ports Clk]]
set_input_delay -min 0.2 -clock Clk $ALL_IN_EX_CLOCK
set_output_delay -min -0.1 -clock Clk [all_outputs]
set_clock_uncertainty -hold 0.5 [get_clocks Clk]

report_timing -delay min

# Fix min timing violations
set_fix_hold [all_clocks]
compile -only_design_rule

redirect top.rpt {report_constraint -all_violators}

```

3.3.3 层次化设计的编译

3.3.3.1 层次化设计的编译过程

一个层次化设计的编译过程包含两个阶段——

1. 将所有的子模块映射到门级

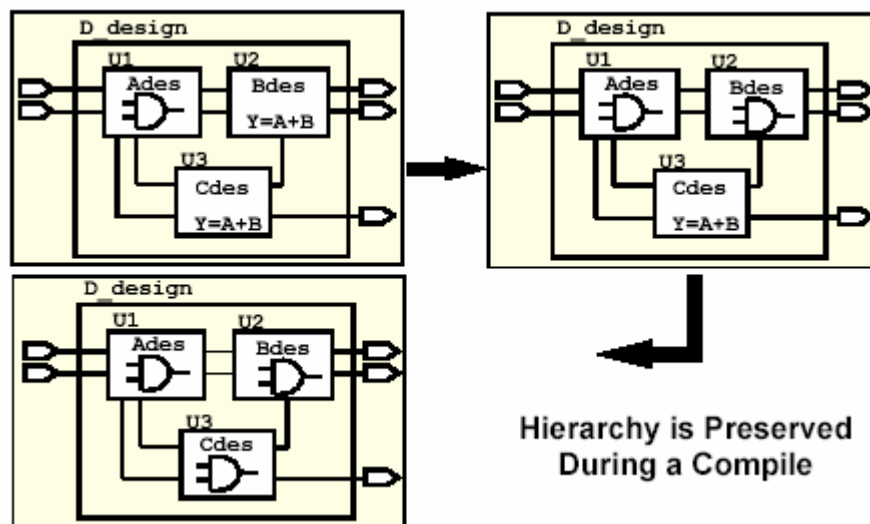


图 121 层次化设计的编译(1)

从上图可以看出，D_design 含有三个不同的模块，在编译的这个阶段，U1、U2、U3 分别由 RTL 级映射到门级，并且各个模块之间的层次关系保持不变。在映射的过程中，设计约束都暂时没有考虑。

2. 优化

在这个阶段, Design Compiler 根据各个子模块的时序和面积约束对它们分别进行优化, 并且在优化的过程中要考虑到不同子模块周围的环境, 修正产生的错误。

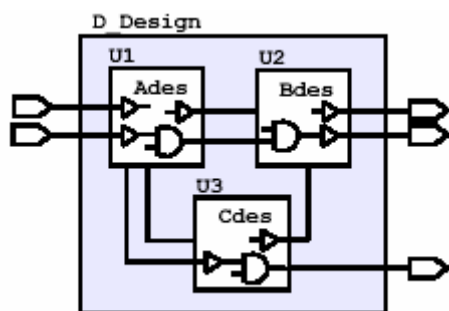


图 122 层次化设计的编译(2)

3.3.3.2 多次例化模块的编译

在一个层次化的设计中, 我们可能会遇到下面这种情况——

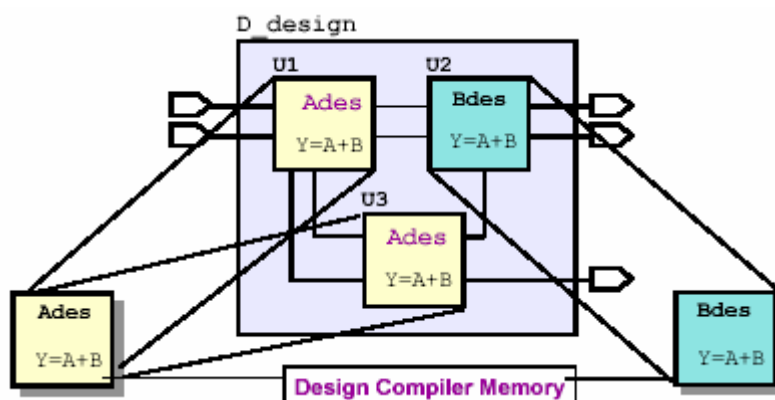


图 123 多次例化的单元

上图中, 被综合的模块中 D_design 中含有三个子模块 U1、U2 和 U3, 其中 U1 和 U3 都是由模块 Ades 例化而来, 这里的 Ades 称为多次例化的模块。对于这样一个设计, 在 compile 之前使用 check_design 作检查的时候会报一个 warning, 即设计中存在多次例化的模块 (multiple instantiations), 如果在这种情况下, 我们不考虑多次例化的模块 (Ades), 那么在继续的 compile 时候程序会终止退出。因此, 必须对它进行处理, 这一节里我们介绍两种方法——uniquify 和 compile + dont_touch。

- 方法一: uniquify

使用 uniquify, DC 会对每个例化的模块作一份拷贝, 然后对它们分别取一个名字, 即把不同的例化模块当作不同的两个模块处理——

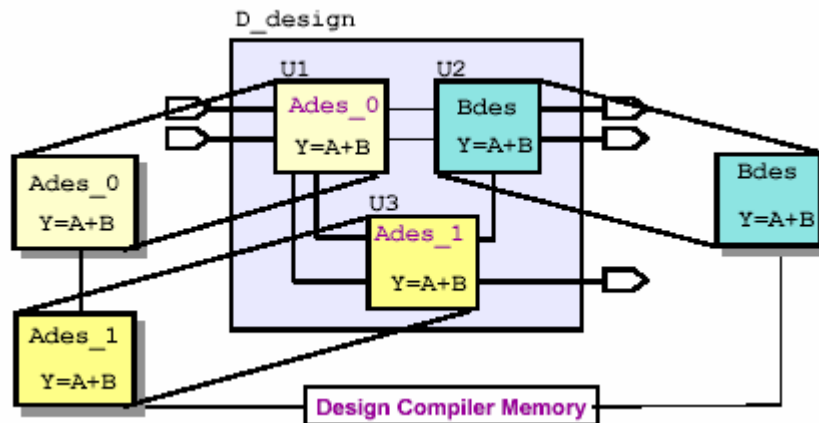


图 124 uniquify

注意看上图，U1 和 U3 两个模块的设计名分别由原来的 Ades 变成了 Ades_0 和 Ades_1，因此在编译时，DC 会将它们当作两个不同的模块，这样就可以根据它们不同的周围环境作优化。

使用 uniquify 的具体实现方法如下——

```
current_design D_design
source D_constraints.tcl
uniquify
compile
```

这段脚本与以前的脚本只有一处不同，即在 compile 之前加上 uniquify 这一行。

- 方法二：compile + dont_touch

这种方法先将多次例化的模块作单独的约束和编译，然后在整合到上一级模块的过程中将它的属性设置为 dont_touch，再编译。

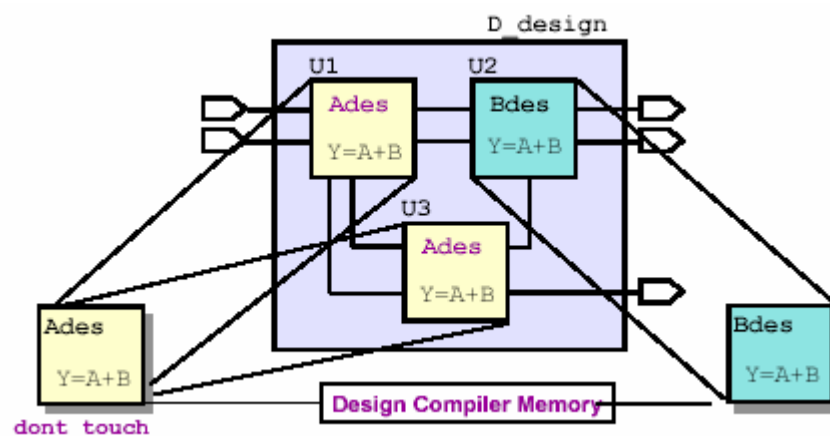


图 125 compile+don't_touch

上图中，U1 和 U3 两个模块的设计名都没有变化，只是在编译 D_design 之前先将 Ades 编译一次。这样 U1 和 U3 实际上是一模一样的模块。

compile+don't_touch 的实现方法如下——

```
read_db unmapped/Ades.db
current_design Ades
link
source Aconstraints.tcl
compile

read_db unmapped/D_design.db
current_design D_design
link
set_dont_touch [get_designs Ades]
source Dconstraints.tcl
compile
```

这里的约束文件有两个，一个是 Ades 的 Aconstraints.tcl，另一个是 D_design 的 Dconstraints.tcl，并且在 source 后一个约束文件之前要对编译过的 Ades 设置成 dont_touch。

在设置了 dont_touch 属性之后，编译 D_design 的时候就会忽略 Ades，这样有好处也有坏处，好处是可以保护模块不被修改，但是这样同时也限制了 DC 对 U1 和 U3 的进一步的优化。

- **uniquify Vs compile + dont_touch**

通过对上述两种方法的介绍，我们不难看出它们各自的优缺点——

compile+dont_touch 由于只需要对多次例化的模块编译一次，因此可以减少整个设计的编译时间，也可以减少内存的使用量。在多次例化的模块很复杂并且工作站的硬件条件有限的情况下，使用这种方法的优越性的比较明显的。还有，如果这个 Ades 是一个第三方提供的硬核(hard-core)，那么我们也只能使用这种方法。

使用这种方法的缺陷也是显而易见的：由于顶层模块在编译的时候 Ades 设置了 dont_touch，这就妨碍了 DC 针对 Ades 的各个实例周围环境的不同的进一步优化，从而使得结果不能真实的反映各个实例周围的环境变化。

uniquify 由于把各个多例化模块作为独立的模块来看，因此 DC 可以分别针对它们作出更好的优化，从而得到的结果也是比较理想的。缺点就是编译的时间稍微较长，但是对于一些不大的模块来说，这些是可以忽略的。

正因为 uniquify 可以综合出更好的结果，所以如果一般推荐使用 uniquify 解决多例化模块的综合问题。

3.3.4 DC-Tcl 控制流及函数

在前面的 DC-Tcl 初步中，我们介绍了 DC-Tcl 的基本数据类型——变量、列表和集合，在这一节里将进一步介绍 DC-Tcl 的控制流语句——条件转移语句和循环语句，以及 DC-Tcl

函数。

3.3.4.1 DC-Tcl 的条件转移语句

DC-Tcl 的条件转移语句有两种——if 语句和 switch 语句。前者相当于 verilog 的 if 语句，后者相当与 case 语句，下面是它们的例子——

- if 语句

```
if [file exists My_Design.db] {  
    read_db My_Design.db  
} else {  
    echo Could not read My_Design.db  
}
```

上面这个例子说明，如果存在 My_Design.db 这个文件，则将该文件读入 DC 的内存中，否则打印“Could not read My_Design.db”。

使用 if 语句的时候要注意 else 必须和前一个条件的“}”在同一行。

- switch 语句

```
set FTYPE [file type My_Design.db]  
switch $FTYPE {  
    file      {read_db My_Design.db}  
    link      {echo db file is a symbolic link}  
    default {echo File is not a valid type for reading}  
}
```

这段语句先将 My_Design.db 的文件类型返回到 FTYPE 变量中，再根据不同的文件类型分别进行不同的操作。

3.3.4.2 DC-Tcl 的循环语句

循环语句介绍三种——foreach 语句、while 语句以及 foreach_in_collection 语句

- foreach 语句

```
# foreach loop example - iterates over  
# elements of a list:  
set MYlist {Hello World}  
foreach list_element $MYlist {  
    echo $list_element  
}
```

这个例子先创建一个列表 Mylist，然后将 Mylist 中的元素依次赋值给 list_element 变量，并将其打印出来。这里 echo \$list_element 要执行两次。

- while 语句

```
# while loop example
set idx 0
set clk_per 10.0

# Create divided clocks on ports CLK0 - CLK9
while {$idx < 10} {
    create_clock -period $clk_per [get_ports CLK$idx]
    incr idx
    set clk_per [expr (2 * $clk_per)]
}
```

这个 while 语句很像 C 语言中的 while 语句。它所要完成的任务是将 CLK0 至 CLK9 这 10 个时钟分别赋值为 10、20、40...

- **foreach_in_collection 语句**

这条语句顾名思义，是用在集合的循环中的——

```
read_db mapped/PRGRM_CNT_TOP.db
set CellColl [get_cells *]
set Count 1

# Print a list of all cells in the Design
foreach_in_collection SingleCell $CellColl {
    set CellName [get_object_name $SingleCell]
    echo Cell $Count is $CellName
    incr Count
}
```

这段语句先创建一个 CellColl 的集合，包含设计中的所有单元，然后针对每一个单元，分别将它们的名字赋值给 CellName 变量并打印出来。

3.3.4.3 DC-Tcl 的子函数

DC-Tcl 支持内部的子函数调用，子函数的引入使得设计者可以定义他们自己的 DC 命令，并且可以修改这些命令的参数以及初始值。下面是一个很简单的实例——

```
myproc.tcl
proc CALC_PERIOD {Clock_Freq} {
    # Convert clock frequency (Mhz) to period (ns)
    return [expr ( (1.0 / $Clock_Freq) * 1000)]
}
```

necessary whitespace same line
↓ ↓

这里创建了一个 myproc.tcl 的文件，里面定义了一个名为 CALC_PERIOD 的子函数，子函数的功能是根据输入的时钟的频率值返回它的周期。

调用这个子函数的方法如下——

定义子函数的时候要注意“必须和 proc 语句在同一行，并且后面要留有空格

```
dc_shell-t> source myproc.tcl
dc_shell-t> CALC_PERIOD 125.0
8.0

dc_shell-t> create_clock \
    -period [CALC_PERIOD 125.0] \
    [get_ports Clk]
```

一般调用之前先要 source 函数所在的 tcl 文件，然后输入函数名和参数，便会自动返回所要求的值。

- **全局变量(Global Variable)**

在谈到子函数的时候，不能不提到子函数内部的变量。一般来说，一个子函数内部定义的变量，它的有效范围只在子函数的内部。任何在子函数外部定义的变量称为全局变量(Global Variable)，全局变量对于子函数来说是不可见的，要想在子函数内部使用全局变量，就必须在子函数中首先声明，如下面的例子——

```
dc_shell-t> proc SP {} {
    global search_path
    set search_path "$search_path ./scripts"
}
dc_shell-t> SP
dc_shell-t> echo $search_path
{... slow_core.db ./scripts}
```

在这里，子函数 SP 需要用到全局变量 search_path，因此在使用之前必须先用 global 语句声明。只有这样，修改后的 search_path 才能在子函数外部有效。

- **查看子函数**

要查看 DC 内存中存在哪些子函数，可以在 dc_shell-t 中使用 info proc 命令。

要查看 DC 内存中特定子函数的内容，可以在 dc_shell-t 中使用 info body 命令——

```
dc_shell-t> info body CALC_PERIOD
# Convert clock frequency (Mhz) to period (ns)
return [expr ( (1.0 / $Clock_Freq) * 1000)]
dc_shell-t>
```

这个例子显示出了子函数 CALC_PERIOD 的具体内容。

- **检查子函数错误**

在编写完一个子函数之后，我们需要先对它进行调试，检查中间是否有错误，查错比较常用的命令是 check_error 命令。

如果子函数编写正确，check_error -v 的输出将为 0——

```
dc_shell-t> check_error -reset
dc_shell-t> source my_script.tcl
dc_shell-t> check_error -v
0
dc_shell-t>
```

如果有错误，check_error -v 的输出为错误代码，可以用 error_info 显示错误具体信息——

```
dc_shell-t> check_error -reset
dc_shell-t> source my_script.tcl
dc_shell-t> check_error -v
{CMD-010}
dc_shell-t> error_info
```

- 子函数实例

下面是一个比较实用的子函数实例，这里定义了一个 TimeBudget 的子函数，输入参数为时钟频率以及时序预算的百分比，然后子函数根据输入的值自动设置设计的时钟、输入输出延时。子函数内容如下(proc.tcl)——

```
procs.tcl
proc TimeBudget {clock_freq time_budget} {

    # Constrain a design for timing, using a time budget

    # clock_freq  <real>    clock frequency in Mhz
    # time_budget <real>    percentage of clock period allowed
    #                                for delay of input/output logic
    #                                in design being constrained

    # calculate intermediate variables
    set CLK_PER [expr ((1/$clock_freq) * 1000)]
    set MY_IO_CONSTRAINT [expr ($CLK_PER*($time_budget/100.0)) ]
    set IO_DELAY [expr ($CLK_PER - $MY_IO_CONSTRAINT)]

    set all_except_clk [remove_from_collection \
                        [all_inputs] [get_ports Clk*] ]

    # constrain the design for timing

    # create clock on clock port
    create_clock -period $CLK_PER -name MY_CLOCK \
                [get_ports Clk*]

    # constrain the inputs
    set_input_delay $IO_DELAY -max -clock MY_CLOCK \
                    $all_except_clk

    # constrain the outputs
    set_output_delay $IO_DELAY -max -clock MY_CLOCK \
                    [all_outputs]

}; # end of TimeBudget
```

下面是调用这个子函数的主程序代码——

```
source procs.tcl

read_db PRGRM_CNT_TOP.db

current_design PRGRM_CNT_TOP

# constrain design for Timing
# using a clock period of 100 Mhz
# and 40% of the clock period for IO timing

TimeBudget 100.0 40.0

# constrain design for environmental attributes
```

3.4 后综合过程

在这一章里，我们着重讨论使用 Design Compiler 综合大型设计时要注意的一些问题，比如怎样调整综合方法，出现约束违反后怎样修正，怎样给不同的子模块作时序和负载预算，以及给整个设计在具体综合之前先作一个预估(Design Exploration)等等。

3.4.1 编译一个大型设计

对于一个大型设计而言，由于模块规模的扩大，编译时间也相应的变长，要长达几个小时甚至超过一天，这样的时间对于讲究“Time to market”的设计者是比较重要的。因此就更加注重编译的技巧，本节我们主要讨论下面三个方面的技巧——

- 编译层次化设计的技巧
- 第二次(Second-pass)编译技巧
- characterize

3.4.1.1 层次化编译

对一个大型设计来讲，有两种层次化编译技巧——自上而下(Top-down)以及自下而上(Bottom-up)。自上而下的方法是指将整个设计一次性读入，施加顶层约束后直接进行编译；自下而上的方法则先一个个编译比较底层的子模块，给它们加入时序和负载预算，然后在顶层将各个子模块整合起来。

- 自上而下(Top-down)

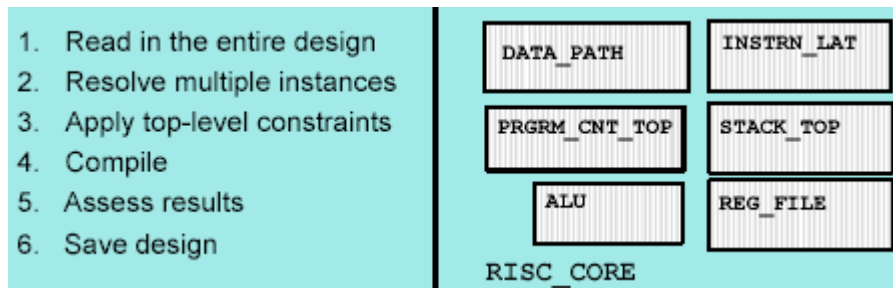


图 126 Top-down 编译步骤

上图是自上而下编译方法的具体步骤，可以看出假如顶层设计是 RISC_CORE 这个模块，则先直接将它读入，然后处理多次例化的模块，施加顶层约束后就直接编译。它的代码基本上如下所示——

```
top-down.tcl
analyze -format vhd1 {alu.vhd reg_file.vhd ... risc_core.vhd }
elaborate RISC_CORE
uniquify
source scripts/top_level.tcl
compile
report_constraint -all
write -format db -hierarchy -output mapped/RISC.db
quit
```

自上而下的编译方法有一个明显的优点，即它使得设计者无需考虑各个子模块之间的依赖关系，也就不需要制定子模块之间的时序和负载预算，这一切都由 Design Compiler 自动考虑。另外，使用这种方法也使得设计者编写脚本变得简单，维护起来也比较方便。

在介绍自上而下的编译方法的时候，我们还要顺便提及 DC 编译的一种模式——Simple Compile Mode(简单编译模式)

这种模式在设计没有严格的约束的情况下能取得较快的编译速度，另外多例化模块的处理也自动进行。下面是 RISC_CORE 的 Simple Compile Mode 脚本——

```
analyze -format vhd1 {alu.vhd... risc_core.vhd}
elaborate RISC_CORE

source scripts/top_level.tcl

# Do NOT execute uniquify
set_simple_compile_mode true
compile
set_simple_compile_mode false
```

可见，使用这种模式省去了 uniquify 这句，同时编译之前要先设置一个变量 set_simple_comile_mod。

● 自下而上(Bottom-Up)

自下而上的编译方法其步骤如下图所示——

1. Constrain and compile subblocks independently
2. Make sure all subblocks meet their initial constraints
3. Read in the entire compiled design and apply top-level constraints
4. Check constraint report: if your design passes, you're are done!

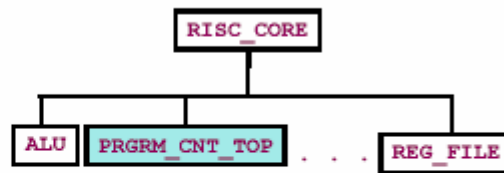


图 127 Bottom-Up 编译步骤

和前一种方法不同，自下而上的编译方法需要先单独编译各个子模块，在编译子模块的同时要考虑到与其它模块之间的关系，看是否满足约束，然后再读入顶层文件，施加顶层约束，顶层编译完成之后还必须看顶层约束是否满足。下面是单个模块编译的脚本——

```

analyze -format vhd1 {PRGRM_CNT.vhd ... PRGRM_CNT_TOP.vhd}
elaborate PRGRM_CNT_TOP

source constraints.tcl
compile
redirect ./reports/PRGRM_CNT_TOP.rpt {report_constraint -all}
# MAKE SURE timing has been met! If not, recode or recompile
write -format db -hier -output mapped/PRGRM_CNT_TOP.db
  
```

下面是顶层模块编译脚本——

```

read_vhdl source/RISC_CORE.vhd

# Bring in compiled .db files
link

# SYSTEM-LEVEL Constraints
source Top_level.tcl

# Check for timing violations
redirect ./reports/RISC_CORE.rpt {report_constraint -all}
write -format db -hier -output ./mapped/RISC_CORE.db
  
```

从上面的过程不难看出 Bottom-Up 方法的一些特点——

优点是利用了“分而治之”的策略，这对于大型的不可能一次编译的设计是十分有用的；另外它也摆脱了 Top-down 方法的对工作站硬件条件的限制，使得大型设计也能在一般的机器上编译完成。

缺点是实现步骤比较多，尤其对各个模块之间的时序和负载预算要求很高，如果不注意会很容易造成违反。

综合上述两种方法，我们可以做一个小结：对于规模不算太大的设计，我们推荐使用 Top-down 的编译方法，这样可以在不长的时间内得到满意的结果。

对于其他需要 Bottom-Up 的设计，我们必须确认时序负载预算能很好的反映实际的工作情况。

3.4.1.2 第二阶段编译

第二阶段(Second-Pass)编译是指当第一阶段(First-Pass)编译出现违反之后，分析违反原因从而重新编译的过程，对应的还有第零(Zero-Pass)阶段编译。

关于第二阶段编译，前面的编译策略一章中有比较详细的介绍，前一章介绍的 Top-down 的第二阶段编译的步骤主要有——

- 检查模块划分
- 检查约束脚本
- 用更高的 map_effort 编译——`compile -inc -map_effort high`

这一节中，我们主要讨论用 Bottom-Up 方法编译后出现违反的情况——

- **重新编译顶层模块**

这种方法是在 Bottom-Up 出现顶层模块时序违反的情况下采用的，具体的命令如下

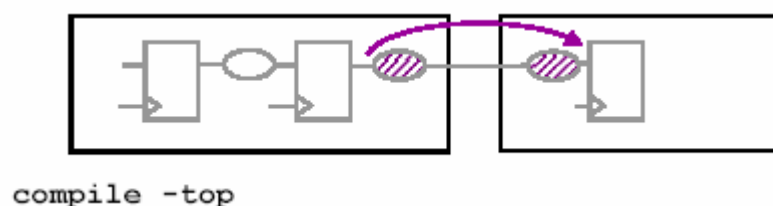


图 128 编译顶层模块

这个命令仅仅修正顶层子模块之间的路径，因此速度会比 `compile -inc` 更快。

- **修正设计预算(Design Budget)**

设计预算对于 Bottom-Up 的方法来说是至关重要的，在预算的时候，我们都尽量能收紧(Tighten)每一个子模块时序、负载和驱动的预算，例如我们在最初介绍设计预算的时候，举的例子是给本模块留整条路径的 40%，因此模块之间能够空出 20%的裕量。在编译子模块的时候能尽量做到满足预算的要求。这样最后整合顶层设计的时候就不会出现大的问题。

如果出现问题了，一个方法就是调整预算脚本。看看施加的约束是否与综合后的电路相吻合。下一节，我们将介绍调整设计预算的一个很有用的命令——`characterize`。

3.4.1.3 characterize

Characterize 这个命令用于映射到门级的子模块，作用是计算出该子模块周围的环境(延时、负载和驱动)，并将得到的实际值作为该子模块的新的约束。如下图一个例子——

- 使用 characterize

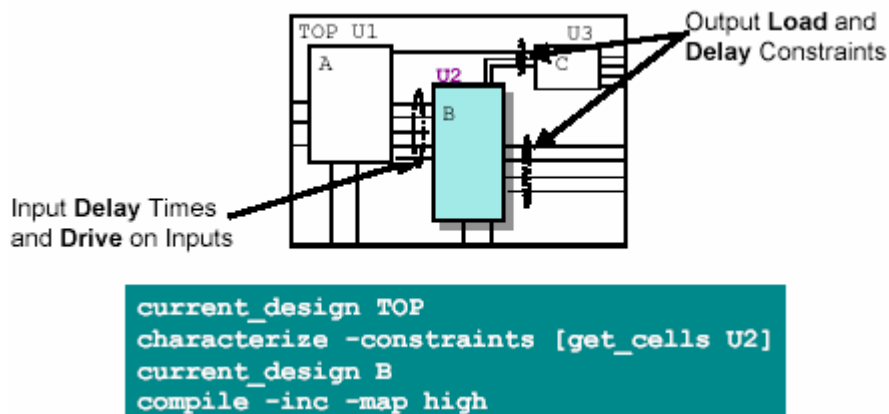


图 129 characterize

由于整个设计已经映射到了门级，因此这个例子可以计算出子模块 U2 周围的输入输出延时、输入驱动和输出负载的实际值。然后将这些实际值施加在 U2 模块中，作为 U2 的新的约束。这种方法有点类似于给 U2 的周围照了一张照片。U2 施加了新的约束之后，就可以在这个基础上做一次高级别的编译。

通过 write_script 命令，我们可以看到 characterize 之后到底照下了哪些信息——

```
/******
Created by write_script() on Mon Oct 12 18:44:54 1998
*****/

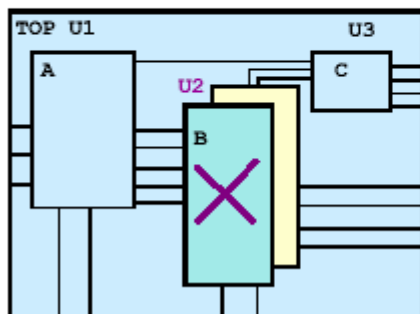
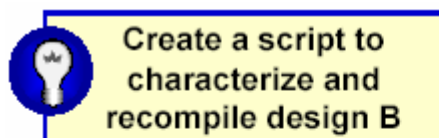
/* Set the current_design */
current_design ALU

create_clock -name "my_clock" -period 10 -waveform {0 5} find(port,"Clk")
set_dont_touch_network find(clock, "my_clock")
set_input_delay 2.2439 -max -rise -clock "my_clock" find(port,"Latch_Flags")
set_input_delay 2.28963 -max -fall -clock "my_clock" find(port,"Latch_Flags")

set_output_delay 5.77132 -max -rise -clock "my_clock" find(port,"Carry_Flag")
set_output_delay 5.80142 -max -fall -clock "my_clock" find(port,"Carry_Flag")

set_load -pin_load 0.343 find(port, "Reset")
set_wire_load "tc6a120m2" -library "cba_core" -port_list find(port, "Reset")
set_driving_cell -lib_cell buf1a4 -pin "Y" -no_design_rule find(port,"Reset")
set_max_capacitance 2.4 find(port, "Reset")
```

characterize 给我们提供了一种比较好的第二次编译的方法，加入一个子模块所占的延时很重，就可以在保持其他模块不动的情况下将这个子模块重新编译一次，当然重新编译可以从 HDL 代码开始，下面是一个例子——



```

❶ current_design TOP
❷ characterize -cons U2
❸ current_design B
❹ write_script > B_w.tcl
❺ remove_design -hier B

❻ read_verilog B.v
❼ current_design B
❽ source B_w.tcl
❾ compile
❿ write -hier -o B.db

⓫ current_design TOP
⓬ report_constraints

```

图 129 用 characterize 作第二次编译

这个例子和前一个例子的不同在于，它没用 `compile -inc high`，而是直接将它从内存中删除，读入它的源文件重新编译，这样可以取得较上一种方法更好的结果。

● characterize 的局限性

`characterize` 无疑向我们提供一种较好的子模块二次编译方法，但是同时它也有一定的局限性，在使用的时候务必要注意——

首先，它要求所有的模块必须映射到门级，这是使用 `characterize` 的一个前提。

其次，`characterize` 只能一次对一个子模块使用，即给 `U2` 作 `characterize` 的时候 `U1` 和 `U3` 模块必须保持不变，否则 `U2` 得到的环境就不是确定的值。

再次，`characterize` 将外界环境直接作为它的约束，这使得它和其他的子模块之间不存在任何裕量(margin)，这些裕量全部被该子模块吸收。

3.4.1.4 附录：Design Budgeter

Design Budgeter 是一个专门给层次化的设计作设计预算的工具，不过在 `dc_shell-t` 中不能使用该工具，有兴趣的同学可以参照这一节的简单介绍自己做一些小练习。

使用 Design Budgeter 有两种方式：一种是在 PrimeTime 中输入 `allocate_budgets`——

```
pt_shell> allocate_budgets
```

另一种是直接调用 Design Budget Shell——

```
budget_shell> allocate_budgets
```

Design Budgeter 可以根据模块周围的环境和它在组合逻辑中的比重自动调整它们的预算大小，与 characterize 不同的是，它可以同时对多个模块作 Budget。这种预算对下面这几种设计非常有用——

- 层次化的设计
- 大型的设计
- 子模块没有寄存输出的设计

它的大致的设计流程如下图所示——

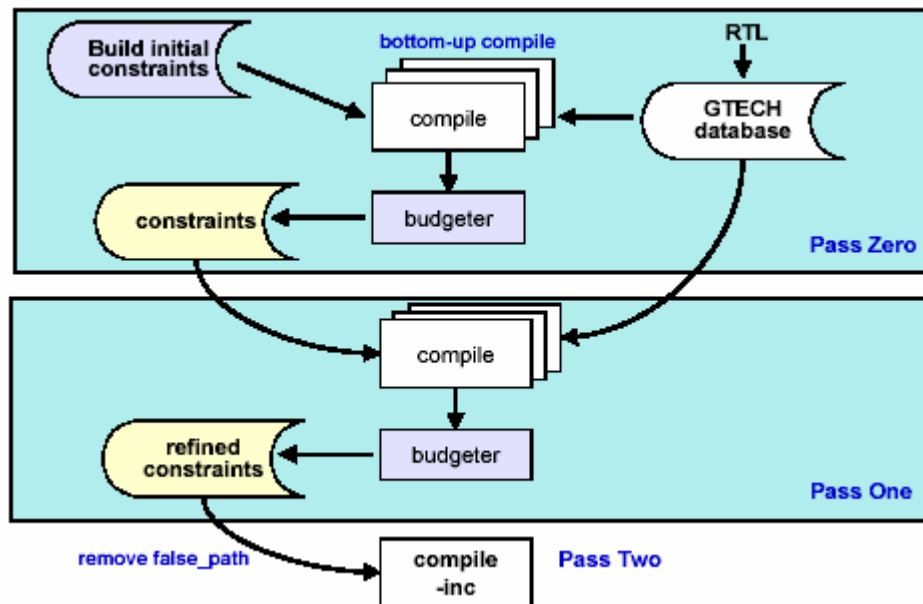


图 130 Design Budgeter 设计流程

图中大家可以看到编译的三个阶段——第一阶段由设计者提供初步的设计预算信息 (initial constraints)，读入 RTL 代码进行编译，得到的网表交由 budgeter，提出新的预算信息 (constraints)；新的预算信息再和 RTL 代码综合得到新的网表，这个网表再交给 budgeter，生成一个最终的预算信息(refined constraints)，这个约束文件就可以提供给 DC 作第二阶段的编译了。

各个阶段的执行命令如下——

■ Pass Zero

```
budget_shell> read_db mapped/TOP.db
budget_shell> allocate_budgets -check_only
budget_shell> allocate_budgets -write_context {U1 U4 U5}
budget_shell> sh ls
U1.ptsh U4.ptsh U5.ptsh
```

■ Pass One

```
budget_shell> read_db unmapped/U1.db
budget_shell> source U1.ptsh
budget_shell> compile
budget_shell> allocate_budgets -write_context U1
budget_shell> write -f db -hier -out mapped/U1.db
```

critical design

■ Pass Two

```
budget_shell> current_design U1
budget_shell> source U1.ptsh
budget_shell> compile -inc
budget_shell> write -f db -hier -out mapped/U1.db
```

这里 Pass Zero 中的第一句读入的 TOP.db 是在施加 initial constraints 以后编译的结果。

从上面的步骤不难看出, Design Budgeter 的流程是一个不断叠代的过程, 它根据最初的人为编写的预算信息, 一步步编译最终得到与实际情况比较接近的信息。

Design Budgeter 除了可以对门级网表进行操作之外也可以对时序模型(Timing Model)进行操作。下面是一个例子——

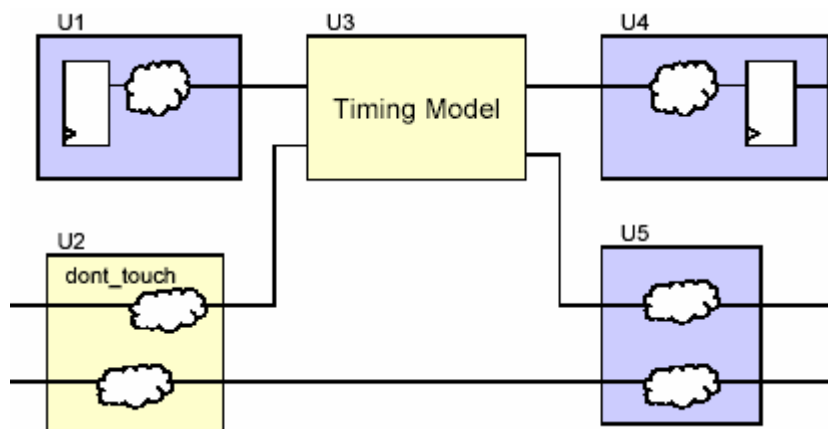


图 131 对 Timing Model 作 Budget

对它作 Budget 的命令如下——

```
allocate_budget -write_context {U1 U4 U5}
```


3.4.2 设计预估

设计预估(Design Exploration)是指在整个设计的RTL代码尚处在验证的阶段就对设计进行预先综合的过程,在这一节里,我们将讨论相比瀑布式的设计流程,设计预估的优点以及设计预估的流程,这些内容可能更多的不是介绍 Design Compiler 的使用,而是设计方法学问题。

3.4.2.1 为什么要设计预估

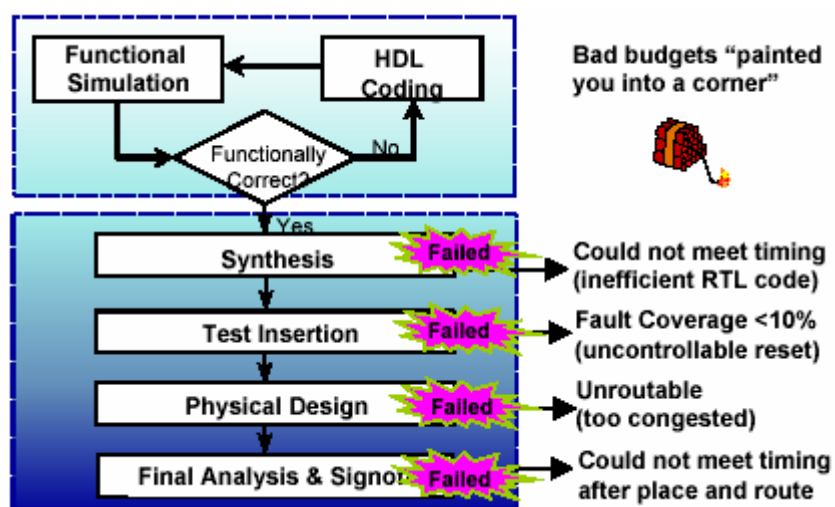


图 132 传统的设计流程

上图左边是一个传统的设计流程,这个流程先作 HDL 代码的编写,在代码仿真通过之后作设计综合,然后插入扫描单元,通过以后作后端的物理设计,直到最后交给 Foundry 流片。这种流程之所以称为瀑布式的是因为后面的步骤都必须等前一个步骤完成之后才能进行。

稍加分析,不难发现这种流程有很大的危险性,从上图的右半部分可以看到,从综合到后端设计,每一个步骤都不可能不产生错误,如综合的时候碰到 RTL 代码引起的时序违反,加入扫描单元后发现测试覆盖率很低,后端布线由于过于拥挤而布不通,等等。这些错误如果可以通过工具的技巧修复还好,如果不能修复,就必须返回到前面的步骤直至 RTL 代码的修改,这样做的代价是随着从前到后迅速增加的,如下图所示——



图 133 不同步骤的自由度 Vs 修改代价

上图是各个不同的步骤的自由度与相对应的修改代价的对比关系,可以看出,越是前面的设计步骤(比如体系结构规划、算法的确定)自由度越大,但是相应所需要付出的代价(时间、人力等)则越小,越到后端则修改的自由度越小,而代价却急遽上升。

因此,要提高设计的效率,就必须在很早的情况下就发现问题并解决它,越是拖到后面,则越不划算。最理想的情况就是在编写 RTL 代码的时候就万无一失,并且在这个时候就能估算出最后的芯片面积和运行速度,以及内部长线的延时。这种思想就要进行设计预估的初衷。

下面是引入设计预估之后的设计流程图——

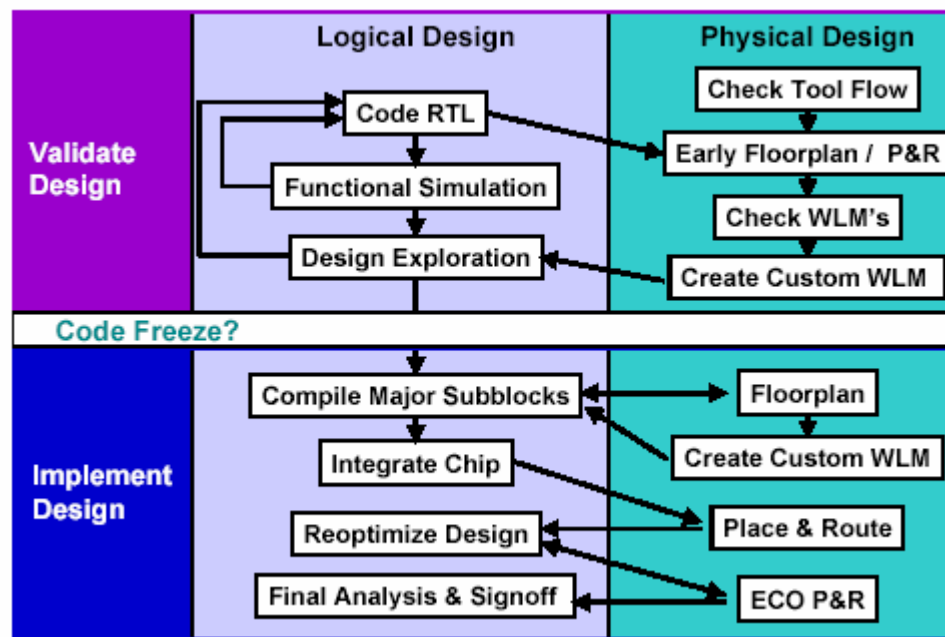


图 134 含有设计预估的设计流程

在这个流程中,前端逻辑设计与后端物理设计同时进行,在编写 code 的同时,对芯片的初步布局和布线先作一个规划,然后检查工艺库中提供的连线负载模型(WLM),根据实际需要创建新的负载模型。而 code 在仿真的同时也可以作一个设计规划,在设计规划的时候调用新的 WLM,以便得出更加真实的结果。

在设计预估做完,code 仿真完成之后,就可以正式编译设计的主要模块,并将它们整合到顶层模块中去。同时对这些模块进行布局重新创建更加真实的 WLM,直到最后的分析直至流片。

可以看到这个流程是一个前端与后端同时进行的过程,也是一个不断融合的过程。前端的设计者总是想尽早知道模块在实际的芯片中的位置,以便得到准确的 WLM。这样才能尽量把错误提前找出来,避免留到后面的高成本。

3.4.2.2 设计预估的目标

在上面的这个流程中,设计预估是与 code 的功能验证平行进行的,而不是等到功能验证完成后再作。每当 code 的改动足以影响系统性能的时候,我们都需要进行一次设计预估。

一般说来，设计预估有下面几个目标——

- **验证代码的可综合性**

这是设计预估的一个基本功能，不能综合的代码对后面的一切步骤都只能是空谈。

- **控制代码的时序违反在一定的范围(10-15%)**

预估后的网表的时序违反如果超过这个范围就可以即使更改代码，保证在正式综合的时候能通过调整综合参数达到时序要求，而不再该代码。

- **验证施加的约束的真实性和充分性**

设计约束不是设计者凭空想象出来的，他需要通过多次的设计预估，并对得到的网表进行分析，从而对施加的约束不断修正，让他能足够反映真实情况，并且不漏掉容易忽视的约束条件。

- **鉴别时序的特殊情况**

大型设计中难免会有一些 DC 不能综合或者无需综合的路径，设计预估需要把这些路径找到。

- **鉴别模块划分问题**

模块划分可以在编写 code 的时候进行，也可以在 DC 的约束脚本中用 group/ungroup 完成，设计预估需要得到一个较好的设计划分。

- **鉴别测试问题**

测试问题我们再 DFT 课程中详细介绍。

- **保证 WLM 的合理**

WLM 是 DC 处理连线延时的依据之一，合理的 WLM 可以保证时序的准确性。

3.4.2.3 设计预估的流程

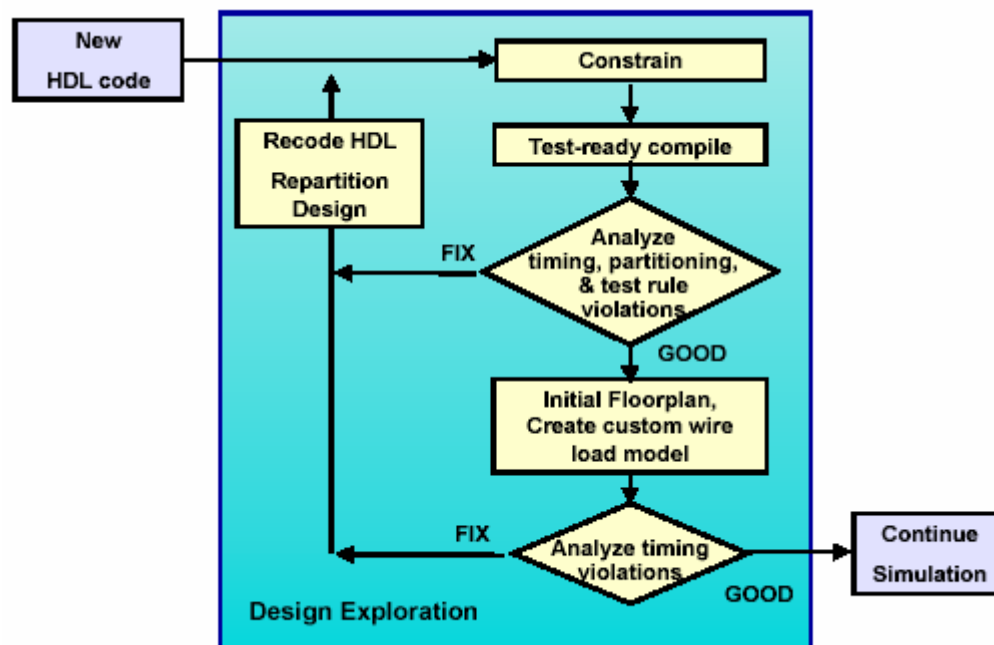


图 135 设计预估的流程

上面的设计预估流程可以看出设计预估包括——读入 HDL 代码，施加约束，加入扫描链，分析综合结果，根据初始化布局和创建的新 WLM 分析时序等等。每当其中一个步骤出现大的错误，都需要该代码。

下面我们主要针对施加设计约束和编译的一些技巧作展开——

设计约束

```
# Define clock
create_clock -period 5 [get_ports CLK]
set_dont_touch_network [all_clocks]

# Delay and drive strength on input ports
set all_inputs_but_clk [remove_from_collection [all_inputs] CLK]
set_input_delay $clk_to_q -clock CLK $all_inputs_but_clk
set_driving_cell -lib_cell $my_register $all_inputs_but_clk

# Delay and load on output ports
set_output_delay [expr 5 - $clk_to_q] -clock CLK [all_outputs]
set_load [expr $pessimistic_load * 3] [all_outputs]

# Describe environment
set_operating_conditions WCCOM
set_wire_load_model -name 100k_WLM -mode top
```



这是一个比较典型的约束脚本，里面包含了定义时钟、输入延时和驱动、输出延时和负

载、运行环境几个方面。里面的命令我们在前面的章节都介绍过，下面着重讨论几个关键的约束——

- 负载约束

```
set pessimistic_load [load_of TECH_LIB/invl1a1/A]
# Account for Pin Load and Wire Load on Outputs
set_load [expr $pessimistic_load * 3] [all_outputs];      # pins
set_port_fanout_number 3 [all_outputs];                  # wires

# Account for Pin Load and Wire Load on Inputs too!
set_load $pessimistic_load $all_inputs_but_clk;          # pins
set_port_fanout_number 1 [all_inputs];                    # wires
```

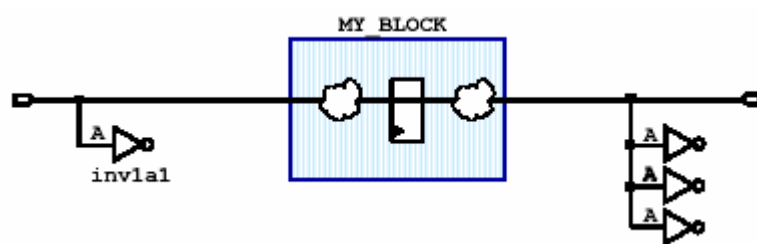


图 136 负载约束

上面的负载约束设置比较保守的输入输出的负载值，它的单位负载值是 invl1a1 的 A 管脚的负载。允许输出端口带 3 个这样的负载并且扇出数目是 3；允许输入端口带 1 个这样的负载并且输出数目是 1。

- 针对端口的 WLM

```
# Describe different WLMs for internal nets vs. global nets
current_design TOP_BLOCK
set_wire_load_mode top
set_wire_load_model -name 100k_WLM [current_design]
set_wire_load_model -name GLOBAL_NET_WLM [get_ports *]
```

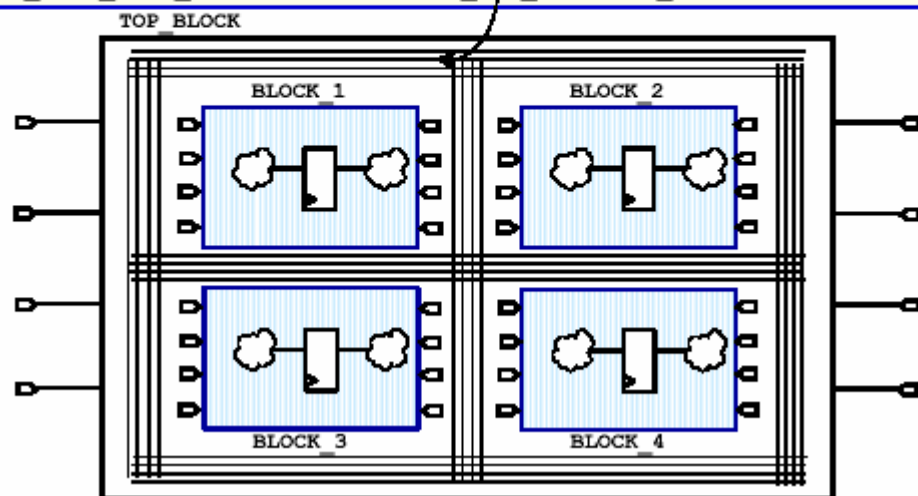


图 137 长线的 WLM

这段脚本对顶层模块内部的连线设置了 100k_WLM 的连线负载模型，对所有连接到端口的连线则设置了 GLOBAL_NET_WLM 的模型，这样能比较真实的反映连线情况。

● 设计规则约束

在所有的设计约束中，设计规则约束的优先级是最高的，也就是说综合的过程中，DC 首先必须满足设计规则。因此，我们可以对整个设计先制定一个设计规则，这样就可以避免单元的规则到达允许的极限值而使延时显著的增加。

设计规则包含——最大电容(max_capacitance)、最大电平转化时间(max_transition)以及最大扇出(max_fanout)。

1. set_max_capacitance

```
# Find the max capacitive load allowed on your expected driver
set DRIVE_PIN TECH_LIB/inv1a27/Y
set MAX_CAP [get_attribute $DRIVE_PIN max_capacitance]; # 3.60
# Add some margin so DC won't fully load the driver
set CONSERVATIVE_MAX_CAP [expr $MAX_CAP / 2.0]; # 1.80
set_load 1.2 [get_ports IN1]
set_max_capacitance $CONSERVATIVE_MAX_CAP [get_ports IN1]
# max internal load DC can put on IN1 is [1.8 - 1.2 = 0.6pf]
```

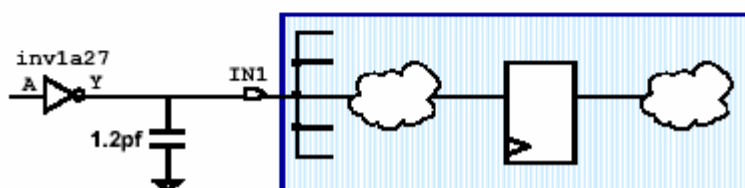


图 138 set_max_capacitance

设置最大电容的命令是 set_max_capacitance。最大电容这个规则一些工艺库中会有定义，如果没有则需要设计者自己找出电容最大的那个单元(本例中用的是 inv1a27)。一般为了保险起见，都会原来的设计规则的基础上加上一个裕量形成新规则。这里就在原来的基础上减少了一半，并将这个值加在 IN1 这个端口上。这样，如果 IN1 端口有一个 1.2pf 的负载，那么留给 IN1 端口内部的电容值就只剩下 0.6pf 了。

2. set_max_transition

```
# Find the max transition allowed on your expected driver
set DRIVE_PIN TECH_LIB/invla27/Y
set MAX_TRANS [get_attribute $DRIVE_PIN max_transition]
0.400
# Add some margin so DC won't fully load the driver
set CONSERVATIVE_MAX_TRANS [expr $MAX_TRANS / 2.0]
0.200
set_max_transition $CONSERVATIVE_MAX_TRANS [get_ports IN1]
# DC accounts for the driving_cell type and external load on it,
# limits internal loads placed on IN1 to meet your design rule
```

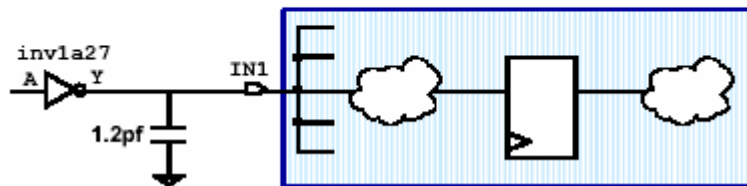
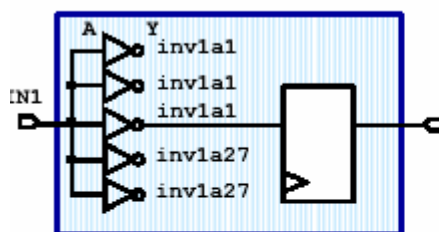


图 139 set_max_transition

设置最大电平转化时间的命令是 set_max_transition。这个值，也是从 invla27 单元中用 get_attribute 得到的。同样留下的裕量是一半，使得 DC 在编译的时候不会接近实际的规则值。这样，如果在 IN1 端口外部有一个负载，DC 会根据这个规则调整内部单元的大小，使得 IN1 满足 max_transition 不超过 0.200 的要求。

3. set_max_fanout

```
set_max_fanout 6 [get_ports IN1]
```



Is the max_fanout design rule on port IN1 met?

How many cells might port IN1 have to drive?

Does it matter what the cell type is?

```
get_attribute TECH_LIB/invla1/A fanout_load
0.25
# DC might load port IN1 with 6 / 0.25 = 24 invla1 cells!
get_attribute TECH_LIB/invla27/A fanout_load
3.00
# DC can only load port IN1 with 6 / 3.00 = 2 invla27 cells!
```

图 140 set_max_fanout

上面的例子给输入端口 IN1 设置了最大扇出为 6。值得注意的是，这里的 6 并不是扇出的数目为 6 个负载，而是扇出值，即负载的大小。例如，如果 IN1 接的的负载是 invla1 的单元，该单元的扇出负载是 0.25，那么 IN1 可以带最多 24 个 invla1；如果负载单元是负载值 3.00 的 inv127，那么只能带 2 个这样的单元。

set_max_fanout 命令是按照单元的 fanout_load 属性来计算负载的，也就是说所有的

负载值相加不能超过 max_fanout。fanout_load 这个属性，有的工艺库单元并不存在，这样的话，DC 会自动去找该工艺库有没有默认的 fanout_load，若有会使用该值代替，如果还是没有则会将 fanout_load 设成 0，即输入端口可以带任意个负载，这显然是不符合实际的，因此，这时就有必要手动设置一个 default_fanout_load。

下面是设置 fanout_load 的一些技巧——

a) 如果想让所有的输入端口只带一个负载单元，可以简单设置 max_fanout 为 1，也可以根据单元中最小的 fanout_load 来设置——

```
# Easiest case
set_max_fanout 1 [all_inputs]
# Trickier case
set SMALL_CELL TECH_LIB/bufla1/A
set SMALL_FOL get_attribute $SMALL_CELL fanout_load
0.5000
set_max_fanout $SMALL_FOL [all_inputs]
```

b) 查看工艺库中是否存在默认 fanout_load——

```
get_attribute TECH_LIB default_fanout_load
0.0000
# Uh-oh!
```

c) 如果没有，手动设置 fanout_load——

```
set_attribute TECH_LIB default_fanout_load 1.0 \
    -type float
1.0000
```

设计编译

```
# Read in source files and build initial Gtech design
if (You_are_using_VHDL) {
    analyze -format vhd1 {file1.vhd file2.vhd file3.vhd TOP.vhd}
    elaborate TOP
} else {
    read_verilog {file1.v file2.v file3.v TOP.v}
}
# Constrain the Gtech design
source my_block_constraints.tcl
source my_dr_cons.tcl
# Optimize and map the design
compile
# Save the design and exit
write -format db -hierarchy -output my_block.db
quit
```

上面是一个典型的设计编译的脚本，和前面章节讨论的编译没有什么太大差别，这里 source 的两个约束文件是设计约束脚本以及设计规则脚本。事实上，对于设计预估而言，为

了尽量减少编译时间，引入可用的 DW_foundation 库可以将上面的代码写成如下所示——

```
current_design MY_BLOCK
reset_design
source time_and_load_budget_constraints.tcl
remove_attribute MY_BLOCK "max_area"
if (You_Have_DesignWareFoundation_and_Plan_To_Use_It) {
    set synthetic_library dw_foundation.sldb
    append link_library " $synthetic_library"
}
set_simple_compile_mode true
set_compile_dw_simple_mode true
set_scan_configuration -style multiplexed_flip_flop
compile -area_effort none -scan
```

上面是一个比较典型的用于设计预估的编译脚本，remove_attribute MY_BLOCK “max_area”以及 compile -area_effort none -scan 两句都说明了编译过程不考虑面积因素，因而可以节约编译时间，另外一个提高编译速度的措施是使用了 simple_compile_mode，这个命令和紧接的 dw_simple_mode 都是告诉 DC，在编译的时候采用较少的优化算法，并对多例化模块只作一次处理。

下面再介绍两个提高 DC 速度的方法，当然他们并非仅仅用于设计预估中——

- `dc_shell-t> set hdlin_enable_presto true`

这个命令告诉 DC，在 HDL 编译的时候采用“Presto”的代码编译器。“Presto”代码编译器是在 2000.10 之后的版本中推出的，它相对前期的 HDL compiler 编译器而言，编译速度提高了 6 倍，但是内存消耗量减少 35%，而且支持更多的 Verilog 语言结构。对于大规模的设计，这种方法的收效是很明显的。默认这个开关是打开的。

- `dc_shell-t> set enable_verilog_netlist_reader true`
● `dc_shell-t> read_verilog -netlist mapped.v *****`

这个命令用在读入门级 verilog 网表中，它相对于前期的网表读入，能平均提高 3 倍的速度且少用 3 倍的内存。默认这个开关是打开的。

3.4.2.4 设计预估实例

前面介绍了设计预估中的一些步骤和特点，这一节我们将讨论顶层设计中的一个子模块的设计预估实例——

我们要做预估的模块是 SUBDESIGN_A，这个模块是位于顶层模块(MAJOR_BLOCK_1)中的一个子模块(位置关系如下图所示)。它的规模在 40K 左右，不算太大，顶层设计 140K，准备采用 Top-down 的编译方法。现在它的 RTL 源代码已经编写完成了，其他几个子模块的代码还在编写中，由于该模块在整个电路中是一个关键的模块，它的性能对整体设计有较大的影响，因此很有必要在对顶层模块综合之前先单独对它做一个预估。

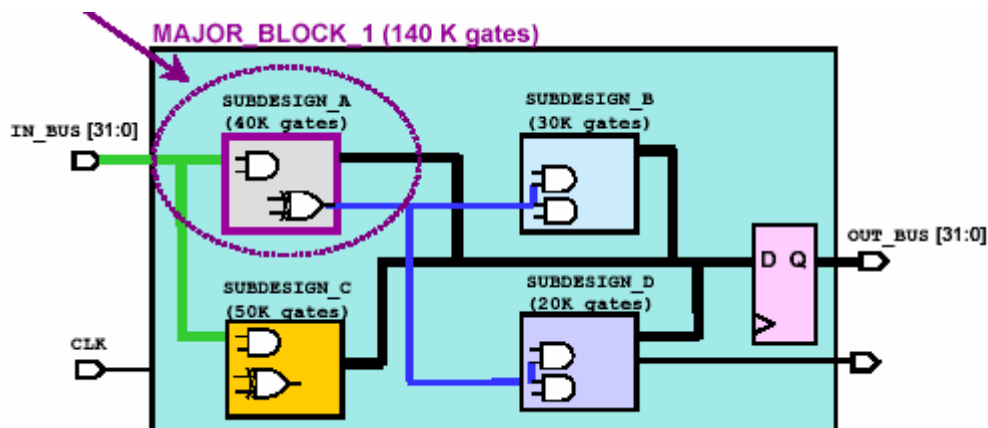


图 141 预估模块在顶层模块中的位置

在结构上，SUBDESIGN_A 并不是全部寄存输出的，并且和它相连的其他模块也可能不是寄存输出的。那么我们应该怎样对它进行约束呢？

这时，我们知道的仅仅有时钟周期和电路工作环境，其他的诸如时序预算、输入驱动、输出负载、WLM 等等都需要自己估计。估计的驱动和负载模型如下——

```
create_clock -period 10 [get_ports CLK]
set_dont_touch_network [get_clocks CLK]
set_operating_conditions SLOW_COMMERCIAL
Set_wire_load_mode top
set_wire_load_model -name 140Kgates [current_design]
set_driving_cell -lib_cell NAND2 -pin Y $all_in_ex_clk
set_load [expr [load_of TECH_LIB/NAND2/A] * 4] $all_in_ex_clk
set_load [expr [load_of TECH_LIB/NAND2/A] * 6] [all_outputs]
```

这里假设驱动单元为 NAND2，WLM 为 140Kgates，每个输入端接有 4 个 NAND2，输出接 6 个 NAND2。

下面设置输入输出延时 如果 SUBDESIGN_A 连接的子模块都是寄存输出的话(如下图)，可以设置延时为——

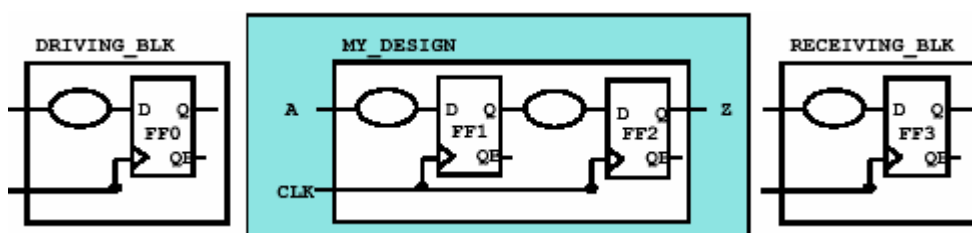


图 142 子模块输出寄存

```
# Assume every block has registered outputs, same 10ns clock
set_input_delay -max $clk_to_q -clock CLK $all_in_ex_clk
set_output_delay -max [expr 10 - $clk_to_q] -clock CLK [all_outputs]
```

或者

```
# Assume every block has registered outputs, same 10ns clock
set_input_delay -max [expr $CLK_PER * 0.1] -clock CLK $all_in_ex_clk
set_output_delay -max [expr $CLK_PER * 0.9] -clock CLK [all_outputs]
```

如果子模块并不是输出寄存(如下图所示),那么可以假设每个子模块平分 50%的延时——

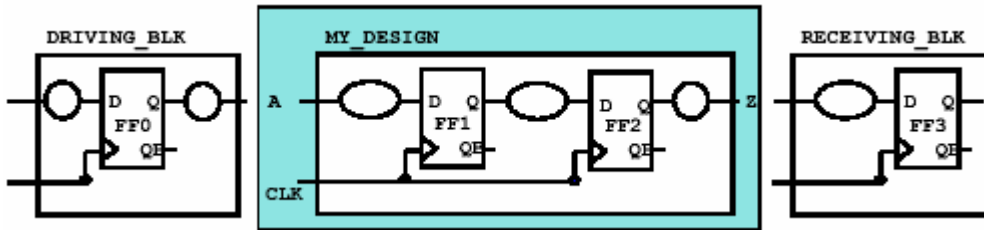


图 143 子模块非输出寄存

```
# Assume blocks do not have registered outputs; split delay equally among both sides
# This is the typical case when exploring smaller, lower-level blocks
set_input_delay -max [expr $CLK_PER * 0.5] -clock CLK $all_in_but_clk
set_output_delay -max [expr $CLK_PER * 0.5] -clock CLK [all_outputs]
```

本例中,子模块 SUBDESIGN_A 的输出不是寄存,所以使用上面的延时。可以看到,各自占用 50%的延时之后,剩下的裕量就不存在了,这样会造成一定的危险性,比如如果该模块存在下图所示的电路——

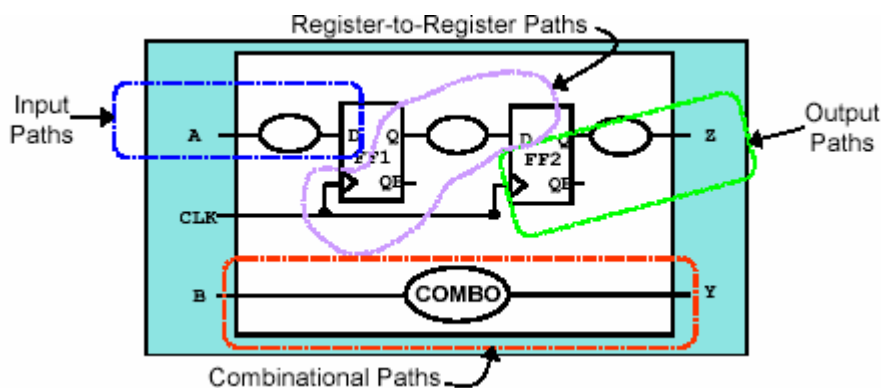


图 143 子模块非输出寄存

这个模块中,除了输出不寄存之外还有一个特点,即中间存在纯的组合逻辑电路(COMBO),试想如果使用 50%的预算方案,那么对于 COMBO 电路而言,输入输出都占了 50%,那么留给自己的就只有 0%的延时,这也许并不是设计者的本意,但是 DC 并没有意识到这一点,它会尽量去优化 COMBO 路径,尽管结果都是时序违反。

碰到这种情况,我们该怎么处理呢?这里介绍一种用于编译的命令 group_path,用户自定义路径组。

默认情况下,上述电路有两个路径组——CLK 和 DEFAULT,其中 COMBO 和输出电路属于 DEFAULT 路径组。在优化的时候,DC 会根据不同的路径组优化电路,并且报告每个路径组的最大时序负裕量(WNS)。如果新增了用户路径组,DC 会把这些新的路径组也作为一个优化的对象,从而便于用户干预优化的过程。

自定义用户组的命令如下——

```
group_path -name INPUTS -from [all_inputs]

group_path -name OUTPUTS -to [all_outputs]

group_path -name COMBO -from [all_inputs] -to [all_outputs]
```

这三个用户组在图中表示为 Input Paths、Output Paths 以及 Combinational Paths。这几个用户组和默认的 CLK 组一起构成 DC 优化的对象。

```
# Avoid getting stuck on one path in the reg-reg group
group_path -name INPUTS -from [all_inputs]
group_path -name OUTPUTS -to [all_outputs]
group_path -name COMBO -from [all_inputs] -to [all_outputs]
group_path -name clk -critical_range 0.3
```

上面这段代码除了设置 3 个用户组之外还对 clk 组设置了 0.3 的 critical_range，这说明和 clk 组内的 WNS 差值在 0.3 内的所有时序违反路径都将被优化，而其余的 2 个路径组则只优化最差的路径。

path_group 和 critical_range 都是用户介入 DC 优化的两个手段，具体使用哪个要依情况而定。path_group 每增加一个，report_timing 的时候就会增加一个 WNS 路径。critical_range 主要用于优化一个路径组中的一些违反路径。

3.5 总结

综合与 Design Compiler 的使用在这里基本上告一段落了，这一节我们将前面的内容做一个简单的总结，大家在使用 Design Compiler 的同时一定会遇到不少问题，这一节也将介绍一些获取帮助的手段。

3.5.1 课程回顾

在本课程的第二章里，我们花了比较大的篇幅讲述各个 Verilog 的语法结构对应的综合网表，事实上，我们反复强调 RTL 代码对 Design Compiler 的综合效率影响是最大的，我们不能指望工具帮助我们一段垃圾代码综合出像样的电路来，即所谓的 Garbage In->Garbage Out。如果综合出来之后的关键路径仅仅是简单的一个门，但是还是违反时序，那这时我们可以断定不是编码的问题了。编码的好坏就相当于一个起点的好坏——



图 144 编码对于综合结果的影响

不好的编码只能使我们的工作适得其反。

更进一步说，决定编码的因素则是决定系统的算法(algorithm)和体系结构(architecture)。例如乘法器是采用流水线在多个周期内完成，还是不用流水线在一个周期内完成；采用的是并行还是串行的结构等等。这些已经脱离了综合的范畴，这里不再展开。

在介绍 Design Compiler 工具时，我们首先介绍了设置库文件及其启动脚本，然后讲述怎样给设计施加约束，接下来是编译和优化的方法和策略，最后介绍了设计预估。基本上是按照一个从前到后，由简入繁的思路讲述的。

关于施加约束，值得一提的是：没有一个放置四海而皆准的黄金脚本可以使用。任何脚本都是根据具体的设计的环境编写的，要得到一个非常满意的约束，需要经过反复的估计。

关于编译，编译的方法很多，compile 的开关也很多，但是要注意的是，时序违反并不是由于 compile 的开关设置不当引起的，原因来自于编码或者约束文件的编写不当。compile 的开关只是提供了修正一些小的时序违反的方法。有关编译策略的流程图如下所示——

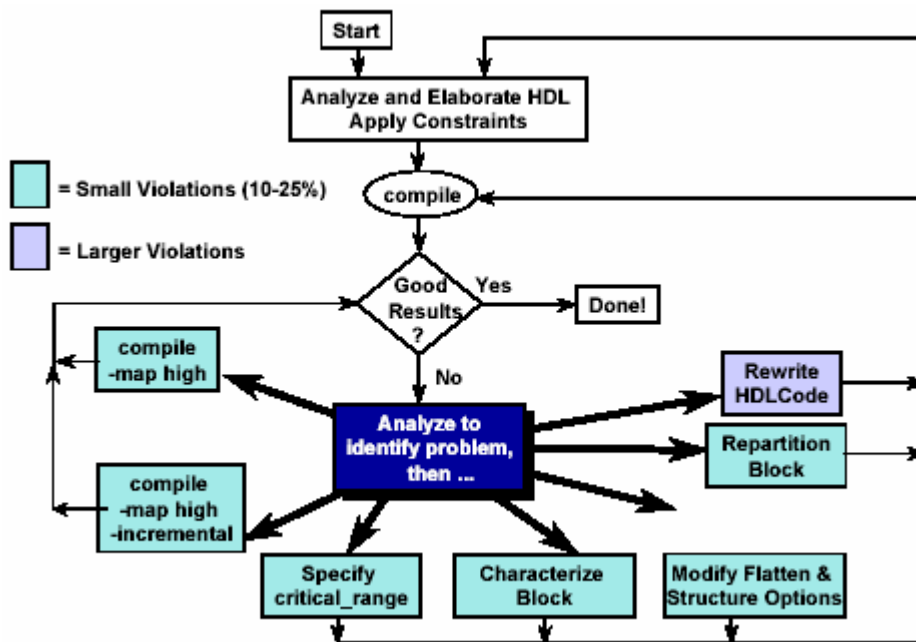


图 145 编译策略流程图

3.5.2 更多其他课程

Design Compiler 不是一个完全独立的工具，事实上它是和其他的工具结合起来使用的，前面的章节我们曾经提到过用于静态时序分析的工具(PrimeTime)，以及设计预估中提到的

DFT (Design For Test), 这两个课程将要在后续课程中深入学习。另外还有一个课程,是本课程的延续,称为高级芯片综合(Advanced Chip Synthesis),主要讲述怎样通过 Floorplaner 创建新的 WLM, 这个课程所要讲述的内容如下所示——

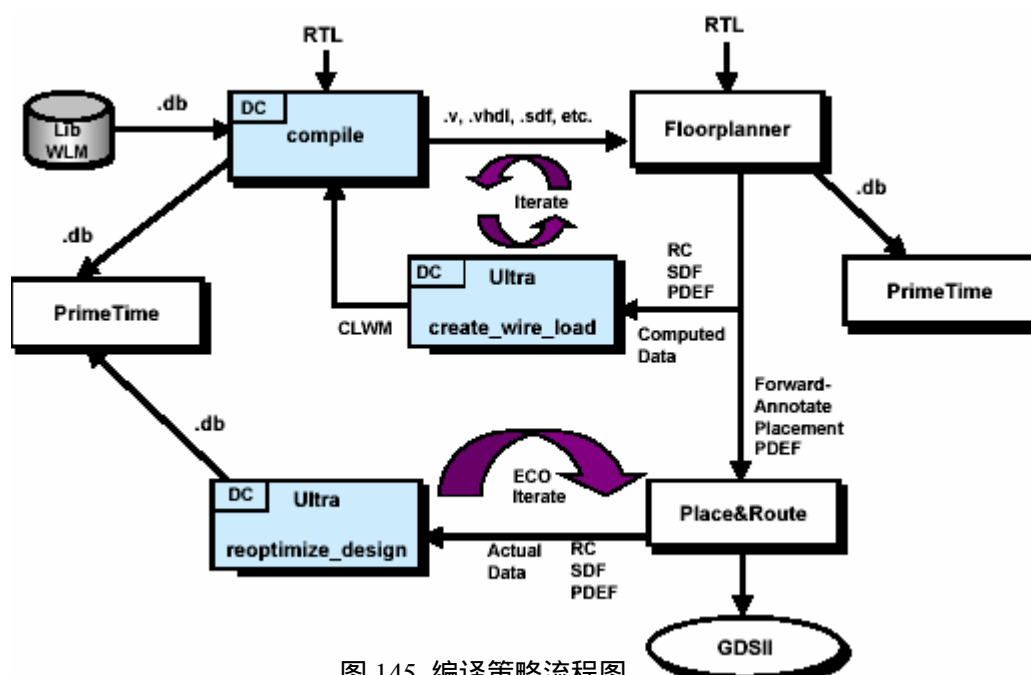


图 145 编译策略流程图

3.5.3 更多帮助

关于帮助,这里介绍三个——SOLD、SolvNet 以及 ESUNG。

- Synopsys Online Documentation(SOLD)

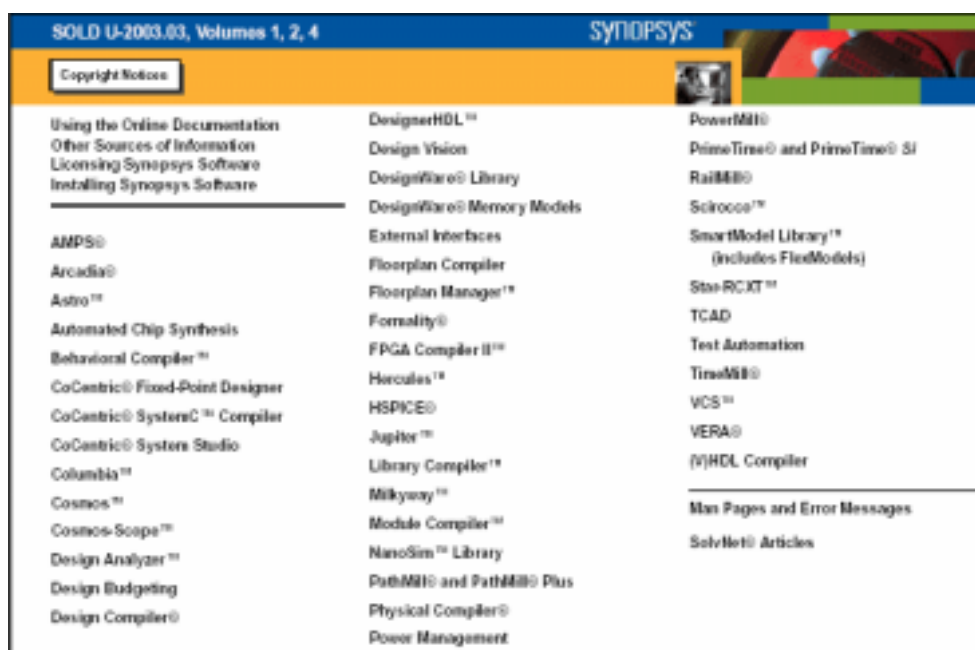


图 146 SOLD

上图是 Synopsys 最新的一期 SOLD，SOLD 里包含了 Synopsys 所有的工具的在线文档，有关 Design Compiler 的其他的命令的解释以及使用说明，这里都有非常详细的介绍。

- **SolvNet**

SolvNet 是一个 Synopsys 官方的用户在线论坛，任何 Synopsys 的合法用户都可以通过 www.synopsys.com 注册称为 SolvNet 的成员。一般说来，通过搜索 SolvNet 上的文档，一些比较常见的问题在这里都能找到答案。

除了在线寻找之外，SOLD 的光盘中也会自带一些以前保留下来的 SolvNet 中的资料。对于设计者来说，可以先从光盘中查找，如果没有相关信息，可以登陆到网站上查找。

下面是 SOLD 中的 SolvNet 主页面——

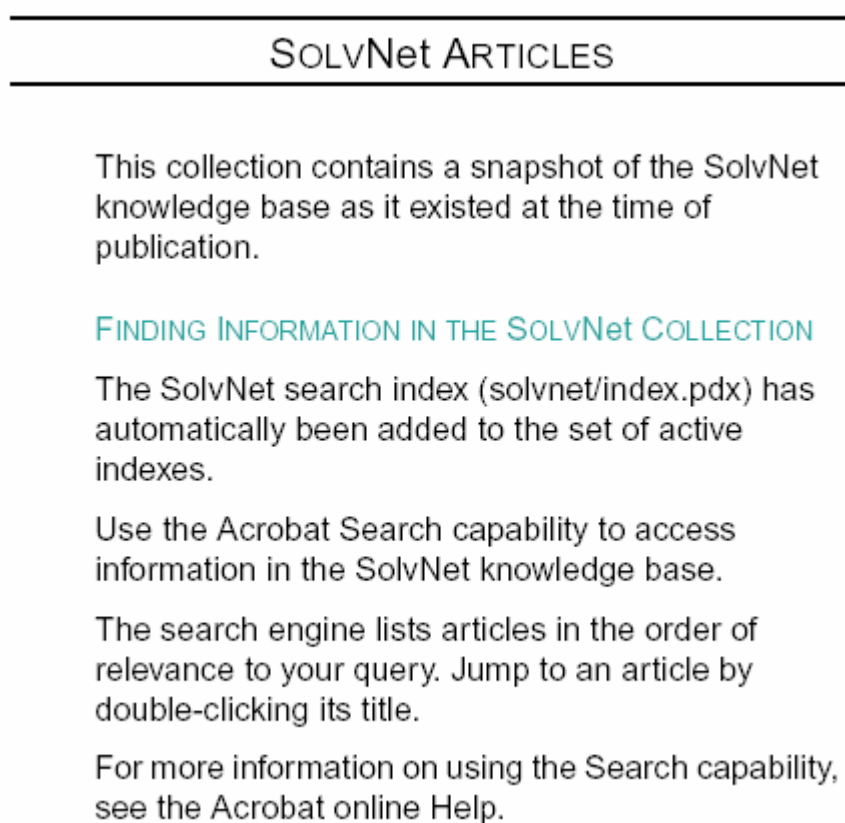


图 147 SolvNet

- **SNUG**

SNUG 是一个非正式的 Synopsys 用户论坛，这里也云集了一大批 Synopsys 用户，它除了提供一些使用技巧和热点疑难解答之外，还会有一些将 Synopsys 工具与其他 EDA 厂商工具的对比的文章。任何注册的 synopsys 用户都能在 SNUG 上下载文章，下面是 SNUG 的网站(www.snug-universal.org)——

地址(D) http://www.snug-universal.org/

North America Europe Asia Israel SEARCH PAPERS

Synopsys Users Group
 Our Mission
 To facilitate an open forum that provides Synopsys users the opportunity to exchange ideas, discuss problems and explore solutions.

Calendar Papers Author's Kit Contact Us

Synopsys Education/Training SolvNet

SNUG Highlights

Featured SNUG Paper ▶ [How To Successfully Use Gated Clocking in an ASIC Design](#)

Europe
 March 6-7, 2003
 Munich, Germany

San Jose
 March 17-19, 2003
 San Jose, CA

Ottawa
 May 8, 2003
 Kanata, ON Canada

Asia
 Tokyo & Osaka - March 2003
 Singapore, India, Korea & Taiwan - May 2003

Boston
 September 8-9, 2003
 Newton, MA

SNUG WORLDWIDE

The Synopsys Users Group welcomes Avant! users.

- [SNUG San Jose 2003 - VIEW PAPERS](#)
Register on-site March 17!
- [SNUG Europe 2003](#)
Papers and Presentations posted
- [SNUGs in Asia](#)
Call for Papers!
- [JSNUG](#)
Tokyo and Osaka

SUG SABER
 • Detroit
 • Munich

图 148 ESNUG