电源网络

万物运行，本源太极。太极分阴阳而生动能。
对于电路来说，这个能量就是电源。阴阳就是 Power 和 Ground 。在数字逻辑中，电源本身只是提供能量，不构成逻辑，应该说更多的属于物理设计的部分。
如果只涉及一种电源，那情况应该是比较简单的。设计中的主要任务可以概括为两个问题：

1. 如何连接标准单元或者 hard macro 的 power/ground pin。
2. 如何确保提供足够的电源供应。

电源的连接

对于 standard cell 来说，如前所述，cell 被按照 site row 排成一排一排的，power/ground pin 分别在 cell 的顶部和底部。因而只要沿着 site row 的上下布好金属层(power rail)即可。这些 power rail 再连接到围在芯片四围的 power ring 上，从而实现与电源的连接。
macro cell 的 powe pin 因为是随设计不同而不同，因此从 routing 的角度考虑即可。

电源的充足

用来传导电源的金属层是有电阻的，电流通过这些金属层会产生电压降，称之为 IR Drop。这里 I 表示电流，R 表示电阻。IR Drop 的后果是可能会导致某些 cell 的电源电压供应不够。为了减少 IR Drop，主要是减少电源网络的电阻，实际设计中的的主要方法就是 Power Grid，即网格状的横的和竖的金属层(Power Strap)。这些 Power Grid 同样也同 Power Ring 相连，从而减少了整个电源网络的电阻。问题是，这个 Power Grid 的密度和 Power Strap 的宽度该如何确定。
就密度而言，自然是够用即可，从而节省布线资源。就宽度而言，考虑的主要是电流密度的影响。电流密度过大会导致金属层失效。减少电流密度的方法是加宽金属。

Power Planning

ASIC 设计中的一个重要步骤 Power Planning 主要就是设计一个电源网络以尽可能少的布线资源提供足够的电源连接。设计往往是与分析不分的。Power Analysis 就是对一个已有的电源网络分析其电气特性，主要是 IR Drop 和 EM。
Power Analysis 在数学层面主要就是电路网络分析。

多电源 （Multiple Supply/Multiple Voltage)

多电源是这几年比较流行的概念，而其实呢，应该说多电源并不陌生。通常芯片的 IO 接口部分所用的电源和主要逻辑部分所用的电源就是不同的。IO 电源的处理自有其策略。这里所说的主要还是逻辑部分本身就有多个电源。
多个电源出现的原因可以归结为下面两种情况：

- 不同的逻辑部分需要不同的电源。通常是电压不同，也可能是虽然电压相同，但电源供应彼此独立。
- 实际上是同一个电源，但某些逻辑会根据需要暂时关闭电源供应以达到节省能量消耗。

有时，电压不同（概念上必然电源不同）被称为 Multiple Voltage；电压相同，电源不同被称为 Multiple Supply。但实际上，这种区分的意义并不是特别大，两者之间的区别不过是在信号穿越两个不同的电源时是否需要进行电压变换。

Legalize 简介

Legalize

基于标准单元的数字 IC 设计（下文中简称为数字 IC 设计）中的 legalize 关心的就是 cell 是否被放置在正确的位置上。有时我们也称 Legalize 为 Detail Placement

在数字 IC 设计中，标准单元(cell)的高度通常是相等的，宽度则是某一宽度的整数倍。用数学来表示就是

- $H = H_{unit}$,
- $W = W_{unit} * N$ (N>0)

我们将宽度为 W_{unit}，高度为 H_{unit} 的矩形称之为 Site。这样我们也可以说 标准单元(cell)的大小是 Site 大小的整数倍。

一个设计(Design)中有成千上万的标准单元(cell)，如何将这么多的 Cell 简单高效的排布在一起，就成了一个重要问题。

想象一下一个大型停车场，如果没有任何规划，任由每辆车随意停放，那一定会出现很多麻烦。于是我们在现实生活中看到的是空地被划分成很多停车位，这些停车位连成一排一排的。每辆车必须要停在停车位内。

在数字 IC 设计中采用的也是类似的办法，我们把 Site 连成一排一排的，要求每个 Cell 必须放在 Site 内。这就像把车停在停车位内一样。

当然了，你可能已经想到了，一个 Cell 可能需要占据连续的好几个 Site。是的，我们就把它当作一辆加宽的汽车吧（别跟我较真说现实中没有这么宽的车）。

这里所讲的其实就是判断“得位”与否的**第一条原则：Cell 必须被放在 Site 内（车辆必须停放在车位内）**。

我们这里说得停车也好，Cell 的放置也好，其实都可以看作是一个二维平面上的几何问题。想象一个二维平面上的确定大小的几何图形，除了位置，还有那些其他因素呢。“方向”算是一个，“对称”或者说“镜像”应该也算一个。

- **方向**：再以停车场为例，比如说我们可以要求所有的车必须车头向里或向外，这样可以使得管理起来更加简单。
- **镜像**：车子可以在它的车位上左右镜像吗？那岂不是要把汽车翻过来才行。哈哈。是的，也许你的车还不行。但是假如有这么一辆车，它的上下两面都有轮子，我可以今天用这面的轮子，明天用另外一面的轮子。但是到了这个停车场，你必须用最难看的那一面。呵呵。

停车可以这样要求，放置 Cell 也可以这样要求。而且考虑到“镜像”其实也可以算作一种方向，我们可以归纳为**第二条原则：Cell 的方向必须符合要求**（车辆在停车位内的方向必须符合要求，否则罚款）

不就是停个车吗，还这么多规矩。别急，还没完呢。这两天领导来视察，那几个离你最近的车位被征用了，旁人不得使用。做人不容易，做一个 Cell 也不容易的。

在数字 IC 设计中，我们是通过 Blockage 圈定一些区域是不可以用来放置 Cell 的，这就形成了**第三条原则：Cell 不能放在 Blockage 下面**。

那么，能不能把两辆车放在同一个车位内呢？这不是废话吗（你心想）。对，是废话，但在严谨的表述中，这就不能当作废话一样不列出来了。**第四条原则：两个 Cell 不可以重叠(Overlap)**。

可以说上面的四条就构成了判断是否“得位”(legality) 的基本原则：

- 第一条原则：Cell 必须被放在 Site 内

- 第二条原则：Cell 的方向必须符合要求
- 第三条原则：Cell 不能放在 Blockage 下面
- 第四条原则：两个 Cell 不可以重叠(Overlap)

在 Synopsys 的 IC Compiler 中，命令 check_legality 基本上就是根据上面的原则进行检查的。
所谓“规矩在，方圆成”。直到了如何是正确的，自然也就知道如何把不正确的编程正确的。
数字 IC 设计中，legalize 就是指把 cell 放置到正确的位置上的过程。
IC Compiler 中，命令 legalize_placement 就是用来做这件事情的。

Multi-Height 和 Multi-Site

大千世界，变化万千。这世上之物要都是一个模子刻出来那样一致，这世界倒是简单了。你看这天下之物，大体上无不是共性之中透着特质，常规之外存有特例。
上面说了加宽的车，大家可能没见过，这加长的应该就不稀奇了。Cell 也是一样，有的时候，有些 Cell 偏偏就是要比那些普通 Cell 高出一倍甚至两倍。既然存在例外，处理方法就要考虑到。我们称这种 Cell 高度是 Site 高度一倍以上的情况为“**Multi-Height**”。
Cell 可以不一样，Site 可以不一样（终生平等嘛）。一个 Design 中也可以有多余一种 Site，这称为 “**Multi-Site**”。
但无论怎样，只要记得让各个 Cell“得其位”就好。这样你也可以去做 legalize 的活了——用你的双手把 Cell 们正确地放到他们对应的 Site 上去。

Placement 简介

Coarse Placement

夫天下之事，能一蹴而就者，鲜有闻矣。欲成大厦，必先成其大概，而后成其屋室。
Cell 的 Placement 亦如是。在进行 Legalize 之前，需要把 Cell 的大致位置先确定下来。这个确定 Cell 大致位置的过程就叫做 Coarse Placement.
Coarse Placement 可以描述为这样一个问题：在指定的平面空间（Floorplan）内，确定各个标准单元 Cell 的位置，以使时序等设计指标最优。
Coarse Placement 由 EDA 工具根据设计者的要求自动完成。设计者的任务就是给出正确的要求。说来简单吧——这做具体事情可能不大容易，但像领导一样只是动嘴指挥大概就简单许多了吧。也对也不对。许多领导动嘴指挥错了，大概不会有什么大问题，但你要是做不出符合要求的设计来，估计就要被老板骂了。

如何影响 Coarse Placement 的结果？

夫治水之道，唯疏堵二法而已。天下之理同也。
所谓设计者的“要求”，就 Placement 来说，归结起来，大的方面无非两类：

- **堵**：指定 Cell 不能放在某个区域。这个称之为 **Blockage**
- **疏**：指定 Cell 只能放在某个区域。这个称之为 **Bound**

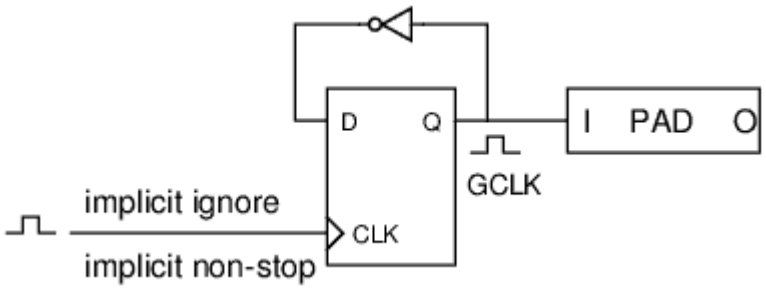
关于 Placement 的物理约束基本上都由此演变而来。

- Hard Blockage: 所有 Cell 一定不能放在指定区域内。
- Soft Blockage: 所有 Cell 只有在特定的步骤（比如 Legalize）中才能放在指定区域内。
- Bound: 指定的 Cell 必须放在指定的区域内。
- Exclusive Bound: 指定的 Cell 必须放在指定的区域内，而且其他 Cell 不许放在指定区域内。
- Group Bound: 指定 Cell 必须放在指定长度和宽度（但没有指定位置）的矩形区域内。
- 其他一些名称不同，概念类似的约束……

知以上几法，功有半矣！那另一半是什么呢？曰：知其所指。
所谓知其所指，有如见云乌而知雨或在途，有如见叶落而知秋至；知雨之将近而备伞，知秋至而加衣。做为设计者，即要知晓设计中的各种概念和因果关系，能防患于位然，能解患之已生。
其实不仅仅是 Placement，设计中的各个步骤都要求如此。至于如何达到这样的要求，无它，唯熟而能生巧。
上面提到的这些策略或方法，其实也正是数字 IC 设计中另一个话题 Floorplan 的重要内容。

既是 ignore pin 又是 non-stop pin 的 clock tree

近日使用 IC Compiler 过程中，在做 CTS 时，发现有一个 register 的 clock pin 既是 ignore pin 又是 non-stop pin。
如果是 ignore pin，则意味着 CTS 时应把这个 sink 排除在外；如果是 non-stop pin，则意味着要穿过这个 sink。而这两种情况是矛盾的。
研究的结果，发现是由于特殊的 clock 结构造成的。如下图：



其中的 register 被用做二分频，输出端 Q 通过一个 inverter 连接到了自己的输入端上；Q 端上定义了一个 Generated Clock(GCLK)。
因此工具推导出这个 register 的 CLK pin 是一个 non-stop pin。
同时，这个 GCLK 通过一个 PAD 输出到芯片外部，没有再与其他 register 相连。因些，工具推导出这是一个 implicit ignore pin。
知道了原因，解决办法就容易多了。将图中 register 的 CLK pin 设置为 ignore pin 更为合理些。

Verilog 中的 Blocking Assignment 和 Non-Blocking Assignment

Verilgo 中有两种赋值语句，用"="表示的"Blocking Assignment"和用"<="表示的"Non-Blocking Assignment"。

有意思的是，这两种赋值语句对是否"blocking"的解释并不如其字面那样易于理解。
看下面的例子，尝试分别画出两个模块对应的电路图：

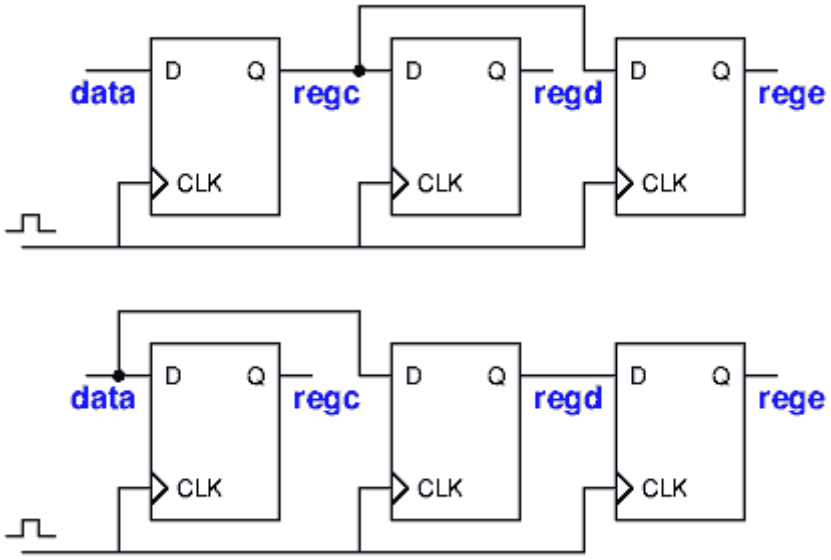
```
module rtl_a(clk, data, regc, regd, rege);
input data, clk;
output regc, regd, rege;
reg regc, regd, rege;

always @(posedge clk)
begin
   regc <= data;
   regd  = regc;
   rege <= regd;
end
endmodule


module rtl_b(clk, data, regc, regd, rege);
input data, clk;
output regc, regd, rege;
reg regc, regd, rege;

always @(posedge clk)
begin
   regc  = data;
   regd <= regc;
   rege  = regd;
end
endmodule
```

问题的关键是 regc,regd,rege 这三个变量中，哪两个是一样的。对于"rtl_a"来说，如果你认为 regd==regc 的话，那你就错了。
正确的答案如下（上面的部分对应 rtl_a）。



Blocking 到底是 Blocking 了什么

其实问题的关键就是理解所谓的"Blocking"到底是针对什么而言。"Blocking"是什么模拟器（simulator）来说的。

当看到"="，即所谓的 Blocking Assignment 时，simulator 会等到该条语句对应的动作实际执行后才去调度一下条语句。而"<="，即"Non-Blocking Assignment"则无需等待语句的实际执行就可以去安排调度下一条语句。

这里用了"调度"这个词，是因为 simulator 的任务实际上主要是安排一系列的"event"的发生，就像一个调度员。

CTS: Clock Tree Synthesis

CTS 的全称是 Clock Tree Synthesis，其目的是尽可能的使同一个时钟信号到达各个终端节点的时间相同。

CTS 的实现办法最常见的是通过在时钟信号的各个分支上插入 buffer 或者 inverter 来 balance 时钟信号的延迟。

Pin 结点的类型

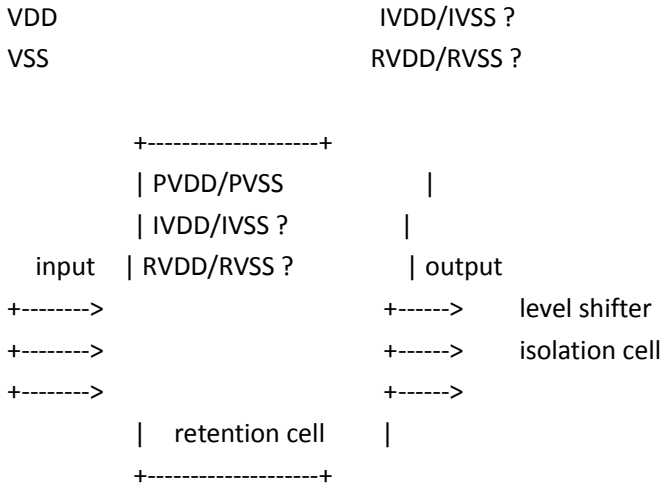
Pin 类型	别名	备注
exclude pin	ignore pin	不用平衡的节点
stop pin	sync pin	需要平衡的节点
non_stop pin		信号会穿过这个节点
float pin		最终节点”藏“在后面

TODO

数字集成电路中的多电源设计

当一个芯片中的电源数目不止一个的时候，就需要小心谨慎地描述各电源之间的关系。这种

设计类型被称为 Multi-Voltage(MV)。
一个相对独立的电源区域和周围区域的关系可以简化如下：



如果当前区域的电源电压和外部不同，则信号传输时需要电平转换。这可以通过在信号线中间插入被称为 Level Shifter 的 cell 来实现。
如果当前区域或者外部区域中的一个电源被关闭了，则相应的信号线上的被驱动端（sink）就会信号不稳定。这可以通过插入被称为 Isolation Cell 的 cell 来实现。
如果当前区域的电源被关闭了，但有些数据又需要在之后重新上电时恢复之前的值，这就需要用到被称之为 Retention Cell 的逻辑单元（cell）来实现。

Level Shifter

Level Shifter 很显然可以放在当前区域内部，也可以放在外部。
在 IC Compiler 中，是通过 insert_level_shifter 这个命令来进行操作的。
为了用于描述插入的策略（strategy），下面两个命令应运而生：

- set_level_shifter_strategy
  - 电平转换的方向：是高到低还是低到高？
  - 插入的位置：是内部还是外部？
- set_level_shifter_threshold ：多大的电压差才需要 level shifter？

Isolation Cell

Isolation Cell 的主要作用是：

- 通电的情况下表现为一个 buffer
- 断电的情况上向外部输出稳定的信号

从逻辑上看，很显然，Isolation Cell 需要一个控制信号（-isolation_signal）来决定工作模式。这个稳定的信号值（-clamp_value）无非是三种情况：0、1、或者 latch（锁存器）的值，显然这对应三种不同的 Isolation Cell 类型。
由于 Isolation Cell 需要在相关电源断电的情况下正常工作，它显然需要一个独立的电源供应。这在 IC Compiler 中被称为 Isolation Power。
用来描述 Isolation Cell 的插入策略可以用命令：

- set_isolation ：描述 Power/Ground 信号、锁存值、哪些 port 上需要插入等。
- set_isolation_control : 描述控制信号和插入位置

Retention Cell

Isolation Cell 主要用于一个电源区域的边界上。Retention Cell 则主要用于该区域内部，它可以在该区域断电的情况下保存住特定的值，并在需要时恢复。从功能上看，Retention Cell 是个稍微复杂点的寄存器（Register）。从电源供应上看，它显然也需要一个独立的电源供应。

这个电源供应在 IC Compiler 中被称为(Retention Power)。
用来描述 Retention Cell 的插入策略的命令：

- set_retention
- set_retention_control

Power Switch Cell

Multi-Voltage 类型的芯片设计中，既然电源的开和关是很常见的，自然就需要一类特殊的 cell 单元来控制电源的开和关。这种类型的 cell 某种程度上更像是一种特殊的 IO Cell，其设计和使用理念也很相像。
最直接的想法就是做成 chip 中的 chip——在芯片内部放置一个小的"IO Ring"。
别外一种想法就是把 power switch cell 像普通 cell 一样放置在 row 上。
TODO

UPF Abstract Demo

```
create_supply_set primary_sset

create_power_domain PD
set_domain_supply_net PD
     -primary_power_net    primary_sset.power \
     -primary_ground_net primary_sset.ground \

set_isolation iso_cstr -domain PD \
  -isolation_power_net    primary_sset.power \
  -isolation_ground_net    primary_sset.ground \
  -clamp_value 0 | 1 | latch                    \
  -applies_to outputs | inputs | both          \
  -source                                        \
  -sink                                          \
  -diff_supply_only true | false                \
  -elements port | pin                             \
  -no_isolation                                    \
# -isolation_supply_set

set_isolation_control iso_cstr -domain PD \
    -isolation_signal pin | port | net    \
    -isolation_sense low | high              \
    -location self | parent | fanout       \

set_retention ret_cstr
    -retention_power_net    primary_sset.power \
    -retention_ground_net primary_sset.ground

create_supply_net VDD -domain PD
create_supply_net VSS -domain PD
create_supply_set primary_sset -update \
     -function {power VDD} \
```

-function {ground VSS}

Scenario in ASIC Design

IC Compiler 中用 Scenario 来描述 design 的不同工作状态(Mode)或工作环境(Corner)。
可能的 Scenario 的数目 = Mode 的数目 x Corner 的数目。

Mode & Corner

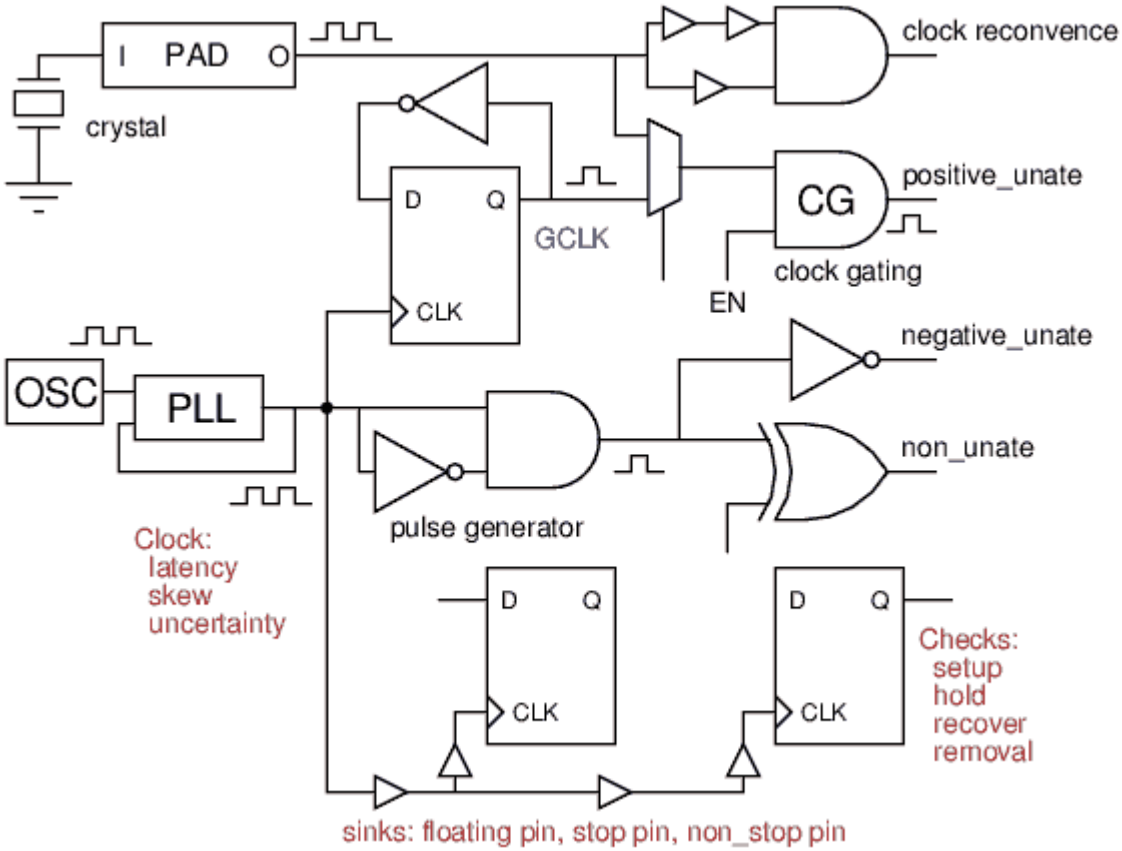
- Function Mode
- Scan Mode
  - Scan Shift Mode
  - Scan Capture Mode
  - Scan Compression Mode
- MBIST Mode
- Boundary Scan Mode
- Fast
- Slow
- Typical
- OCV(On Chip Variatio)

Scenario Diff

随着 design 复杂度的提高，一个 design 往往有数个不同的 scenario，这些 scenario 之间的差别有时并不是那么大的。
一个想法是如果能简单比较出不同 scenario 之间的区别就好了。
TODO

数字 ASIC 设计概要：时钟（Clock）信号

数字集成电路中，各逻辑单元自行其是而不顾它，整个电路却能工作得井然有序，可以说全赖时钟信号发号施令，统御全局之故。



时序约束（Timing）简介

万物有序，唯有序而变化生，变化生而世界成。序者，规则也。
数字设计中，时序是最基本，也是最重要的概念。

基本概念

我们所说的数字设计多数时候都是指的同步逻辑。所谓同步逻辑，是说所有的时序逻辑都在时钟信号的控制下完成。这很像是大合唱，有很多的人参与，大家都根据同一个节拍来控制节奏，保持整齐。时钟信号就是那个节拍。其实很多地方都需要有一个节拍来协调系统的各个部分。比如工厂里的一条流水线。

流水线的每个工人从前一个人那里拿到中间产品，装配一个零件，然后交给下一个人；每一个人面前的空间只有放置一个中间产品的空间。装配一个零件并把中间产品交到下一个那里是需要时间的。如果这个时间太短，就会发生下一个人手中的零件还没装配好，新的中间产品又来了，却没处放的问题。如果这个时间太长了，就会发生下一个人闲着没事干的情况。前一种情况会导致流水线混乱，后一种情况则导致流水线的效率下降。因此，在确定的流水线效率要求下，我们就必须要求每位工人装配一个零件并把中间产品传到下一位工人的时间不能太快也不能太慢。这话也可以反过说，只有保证每位工人所用的时间在一个确定的时间范围内，我们才能保证流水线在按照特定的效率运行。

同步逻辑数字设计也好比是设计一个满足上述要求的“流水线”，只不过这个流水线是由传递二进制数据的数字器件组成。数据在两个存储单元之间传递，“效率”由时钟的周期(T)决定。每个存储单元在时钟节拍（通常是时钟信号的上升沿）的号令下工作。如同每个工人装配零件需要时间一样，每个存储单元正确地存储数据也是有要求的。它要求数据必须在时钟

沿之前的某个时间就准备好（我们把这个时间称为 setup time，记为 t_{setup}），并且有这个数据必须保持不变一直到时钟沿之后的某个时间（我们把这个时间称为 hold time，记为 t_{hold}）才能完成数据的存储。 每个存储单元传递数据到下一个单元的时间我们记为 t_{data}，累比于上面流水线的例子，我们很容易得出，为了保证数据的正确传递，必须要求:

$t_{hold} < t_{data} < T - t_{setup}$

如果不是 $t_{hold} < t_{data}$，就会干扰下一个存储单元当前的数据存储；如果不是 $t_{data} < T - t_{setup}$，下一个存储单元就来不及存储下一个数据。实际中为了便于分析，这种约束关系也可以表示为：

$t_{slack\text{-}max} = (T - t_{setup}) - t_{data} > 0$

$t_{slack\text{-}min} = t_{data} - t_{hold} > 0$

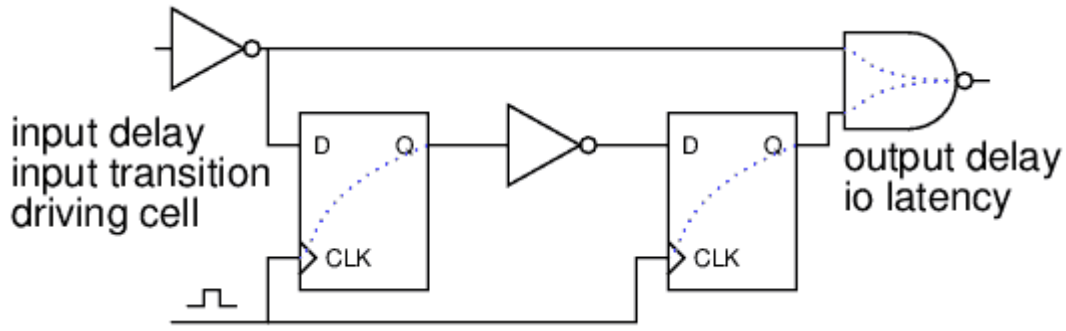
这样一来，我们只需要检查每两个存储单元之间的 $t_{slack\text{-}max}$ 和 $t_{slack\text{-}min}$ 是否为正值就可以了。我们把这两个差值称为 Slack 。Slack 为正，我们称之为 "MET"，Slack 为负，我们称之为 Violation。

事实上，两个存储单元之间通常还会有组合逻辑单元，它们也传递数据，也对数据进行“加工”（二进制运算），只不过他们不需要理会时钟信号的存在，它们只是按照自己的能力大小（速度快慢）尽职而已。仍然以时钟节拍（比如时钟信号上升沿）做为参考的话，存储单元完成内部数据存储并输出到它的输出端（比如 Q 端口)也是需要时间的（我们记为 $t_{clk\text{-}q}$）。数据通过两个存储单元之间的组合逻辑所需要的时间记为 t_{comb}，则有:

$t_{data} = t_{clk\text{-}q} + t_{comb}$

至此，我们应该已经在体上知道了所谓的时序约束是指什么。但实际应用中，为了更好地将相关问题数学模型化，我们还需要进一步的概念定义。

Timing Path



上面提到的 t_{data} 其实就是数据沿着一条数据通道传递所用的时间。而这条数据通道，我们就叫做 Timing Path。总结一个 Design 中可能的 Timing Path，有下面四种：

- 从存储单元的时钟端口到另一个存储单元的数据输入端口的数据通道
- 从存储单元的时钟端口到 design 的输出端口
- 从 design 的输入端口到存储单元的数据输入端口
- 从 design 的输入端口到 design 的输出端口

有了 Timing Path 的定义，关于时序的计算事实上就可以描述为遍历 design 中所有的 Timing Path ，计算它们的 Slack。

- False Path：所谓 False Path，简单来说就是指那些理论上存在，但实际上却不可能有数据沿它传输的 Timing Path。
- Disabled Timing Path：人们有意识地排除掉的 Timing Path 。
- Multicycle Path：通常的 Timing Path 在一个时钟周期内完成数据传输。有时，数据路径比较长，需要多于一个时钟周期才能完成数据传递，这种特殊的 Timing Path 叫

做 Multicycle Path 。

时钟信号（Clock）

在上面的讨论中，我们所说的时钟信号都是理想情况下的。现实世界常常都是不会尽如人意的。

时钟信号传递到各个节点所用的时间不会总是相同的。时钟信号两个相邻的周期的波形也不会完全相同。

- Latency：我们把时钟信号传递到存储单元的输入端所用的时间定义为 Clock Latency 。
- Uncertainty：时钟信号传递到两个不同的节点所有的时间的差值我们称为 Clock Uncertainty。而 Uncertainly 事实上又可以分为两部分。
  - Skew：同一个时钟信号由于传递路径不同导致的时间差异，我们称之为 Skew 。
  - Jitter (抖动)：两个不同时刻（主要是相邻两个周期）的时钟信号由于时钟源本身的不稳定而产生的信号边沿的时间差异，我们称之为 Jitter。

$t_{uncertainty} = t_{skew} + t_{jitter}$

对于 Jitter, 通常都相对 Skew 而言比较小，我们也只能尽可能挑选稳定性好的时钟信号源。在实际设计中，我们更关心 Skew 多一些。我们常说的 CTS (Clock Tree Synthesis) 的主要目的就是如何让 Skew 尽可能地小。

功耗分析及设计原则

低功耗很重要，因为功耗越低越省电、延长电池使用时间、减少芯片发热。

Leakage Power

Leakage Power 是一种静态电源功耗。所谓静态，是指信号状态不发生改变的情况下所产生的功耗。MOS 管即使是在稳定状态也是有

Leakage Power 的计算

一个标准单元的 Leakage Power 在不同的输入状态下可能是不一样的。为了使计算值与实际值更接近，可以对各个输入状态的值按概率进行加权。

比如一个两输入的与非门(NAND)的 Leakage Power 可以计算如下：

Power = P(AB)*Power(AB) + P(A'B')*Power(A'B') + P(AB')*Power(AB') + P(A'B)*Power(A'B)

其中的 P(AB)表示 AB 这种输入状态的发生概率。很显然各种状态的概率之和应该等于一。

在 Synopsys 的 lib 库文件中，或者显示地定义了某种状态对应的 leakage power，或者使用默认值 cell_leakage_power

```
cell(AND2) {
  cell_leakage_power : 0.00123456 ;

 leakage_power() {
     when : "!A1&!A2" ;
     value : "0.00654321" ;
  }
```

```
  leakage_power() {
    when : "!A1&A2" ;
    value : "0.00345678" ;
  }
}
```

为了知道各个输入状态的发生概率，就需要用到文件 SAIF(Switching Activity Interchange Format)文件或者 VCD(Value Change Dump)文件。

Dynamic Power

动态功耗是由于信号的跳变过程中对相关电容进行了充放电引起的。

Dynamic Power = V2 • f • C

显而易见，要想降低 Dynamic Power，或者降低电压，或者降低频率，至于负载 C 则主要由芯片工艺决定了。

SAIF & VCD

- SAIF: Switching Activity Interchange Format
  - 包含信号的 toggle count 和信号在各个状态（0/1/X/Z）的时间长度等信息
  - 通常比 VCD 文件要小.
- VCD: Value Change Dump
  - 记录仿真模拟(simulation)过程中信号发生变化的细节
  - vcd2saif 可以转换 VCD 文件为 SAIF 文件

Design Rule Constraints 的意义和使用

Design Optimization 过程中所说的 DRC(Design Rule Constraints)主要是指 max_transtion、max_fanout、max_capacitance 三种。

- max_transition / min_transition (input pin, output pin)
- max_fanout / min_fanout (output pin)
- max_capacitance / min_capacitance (output pin)

max_transtion 主要是用在 input pin 上，max_fanout 和 max_capacitance 主要是用在 ouput pin 上。

max_transtion 和 max_capacitance 背后的意义

max_transition 和 max_fanout 本质上是间接的(indirectly)的 max_capacitance。

在计算 cell 的延迟(delay)和输出 pin 上的 transition 时，基本上以 input pin 的 transition time 和 output pin 驱动的 capacitance 作为参数计算的。由于其中的关系是非线性(nonlinear)的，实际中是通过 LUT Table 查询或插值(interpolation)计算得出。当 input transition 或者 output capacitance 超出了 LUT Table 的范围时，就需要使用外插值(extrapolation)，这时会影响计算的精确度。

基本上 max_transtion 和 max_capacitance 的值应该就是 LUT Table 中相应参数最大值。

由此可知，DRC Violation 的大小，其真正意义在于警示计算结果的精确度。

DRC 的优先级

1. min_capacitance
2. max_transition
3. max_fanout
4. max_capacitance
5. cell_degradation : max_capacitance = f(input_transition)

库中的 DRC 信息

report_lib -timing 可以列出 lib_pin 的 DRC 信息。

PIN(name) : in, capacitance, fanout_load, max_transition,
rise_capacitance, fall_capacitance ;

PIN(name) : out, capacitance, max_fanout, max_transition, max_capacitance,
min_fanout, min_capacitance ;

PIN(name) : inout, capacitance, fanout_load, max_fanout, max_transition, max_capacitance,
min_fanout, min_transition, min_capacitance,
rise_capacitance, fall_capacitance ;

数字集成电路(ASIC)中的标准单元(standard cell)

数字集成电路(ASIC)中都会用到哪些基本单元(standard cell)呢？这可以从 Synopsys 工具的 gtech 库中略见一斑。

- buffer
- inverter
- AND/NAND
- OR/NOR
- XOR/XNOR
- Mux
- Adder
- D Flip-Flop
- JK Flip-Flop
- Latch
- ISO/ISO Latch
- RS Latch

```
#name    #pins #inputs #outpus #pin_function_class #pin_names

GTECH_ZERO                  1    0=>1 a0                  | Z
GTECH_ONE                   1    0=>1 a0                  | Z
GTECH_BUF                   2    1=>1 a1                  | A Z
GTECH_NOT                   2    1=>1 a1                  | A Z
```

GTECH_AND2	3	2=>1 a2	| A B Z
GTECH_AND3	4	3=>1 a3	| A B C Z
GTECH_AND4	5	4=>1 a4	| A B C D Z
GTECH_AND5	6	5=>1 a5	| A B C D E Z
GTECH_AND8	9	8=>1 a8	| A B C D E F G H
GTECH_NAND2	3	2=>1 a2	| A B Z
GTECH_NAND3	4	3=>1 a3	| A B C Z
GTECH_NAND4	5	4=>1 a4	| A B C D Z
GTECH_NAND5	6	5=>1 a5	| A B C D E Z
GTECH_NAND8	9	8=>1 a8	| A B C D E F G H
GTECH_OR2	3	2=>1 a2	| A B Z
GTECH_OR3	4	3=>1 a3	| A B C Z
GTECH_OR4	5	4=>1 a4	| A B C D Z
GTECH_OR5	6	5=>1 a5	| A B C D E Z
GTECH_OR8	9	8=>1 a8	| A B C D E F G H
GTECH_NOR2	3	2=>1 a2	| A B Z
GTECH_NOR3	4	3=>1 a3	| A B C Z
GTECH_NOR4	5	4=>1 a4	| A B C D Z
GTECH_NOR5	6	5=>1 a5	| A B C D E Z
GTECH_NOR8	9	8=>1 a8	| A B C D E F G H
GTECH_XOR2	3	2=>1 xor2	| A B Z
GTECH_XOR3	4	3=>1 xor3	| A B C Z
GTECH_XOR4	5	4=>1 xor4	| A B C D Z
GTECH_XNOR2	3	2=>1 xor2	| A B Z
GTECH_XNOR3	4	3=>1 xor3	| A B C Z
GTECH_XNOR4	5	4=>1 lxor4	| A B C D Z
GTECH_AND_NOT	3	2=>1 a2	| A B Z
GTECH_OR_NOT	3	2=>1 a2	| A B Z
GTECH_AO21	4	3=>1 a2b2	| A B C Z
GTECH_OA21	4	3=>1 a2b2	| A B C Z
GTECH_OA22	5	4=>1 a2b2b2	| A B C D Z
GTECH_AO22	5	4=>1 a2b2b2	| A B C D Z
GTECH_AOI21	4	3=>1 a2b2	| A B C Z
GTECH_AOI22	5	4=>1 a2b2b2	| A B C D Z
GTECH_AOI222	7	6=>1 a3b2b2b2	| A B C D E F Z
GTECH_AOI2N2	5	4=>1 a2b2b2	| A B C D Z
GTECH_OAI21	4	3=>1 a2b2	| A B C Z
GTECH_OAI22	5	4=>1 a2b2b2	| A B C D Z
GTECH_OAI2N2	5	4=>1 a2b2b2	| A B C D Z
GTECH_MAJ23	4	3=>1 maj23	| A B C Z
GTECH_MUX2	4	3=>1 mux21	| A B S Z
GTECH_MUXI2	4	3=>1 mux21	| A B S Z
GTECH_MUX4	7	6=>1 mux41	| D0 D1 D2 D3 A B Z
GTECH_MUX8	12	11=>1 mux81	| D0 D1 D2 D3 D4 D5 D6 D7

```
GTECH_ADD_AB              4   2=>2 a2 xor2              | A B COUT S
GTECH_ADD_ABC             5   3=>2 maj23 xor3         | A B C COUT S
GTECH_ISO0_EN1            3   2=>1 a2                 | DI EN DO
GTECH_ISO1_EN1            3   2=>1 a2                 | DI EN DO
GTECH_ISO0_EN0            3   2=>1 a2                 | DI EN DO
GTECH_ISO1_EN0            3   2=>1 a2                 | DI EN DO
GTECH_ISOLATCH_EN1        3   2=>0                     | EN DI DO
GTECH_ISOLATCH_EN0        3   2=>0                     | DI EN DO
GTECH_OUTBUF              3   2=>1 unknown             | DATA_OUT OE PAD_OUT
GTECH_INOUTBUF            4   3=>2 unknown a1(2io)     | DATA_OUT OE PAD_INOUT DATA_IN
GTECH_INBUF               2   1=>1 a1                  | PAD_IN DATA_IN
GTECH_TBUF                3   2=>1 unknown             | A E Z
GTECH_FD1                 4   2=>0                     | D CP Q QN
GTECH_FD14               13   1=>0                     | D0 CP Q0 QN0 D1 Q1 QN1 D2
GTECH_FD18               25   1=>0                     | D0 CP Q0 QN0 D1 Q1 QN1 D2
GTECH_FD1S                6   4=>0                     | D TI TE CP Q QN
GTECH_FD2                 5   3=>0                     | D CP CD Q QN
GTECH_FD24               14   2=>0                     | D0 CP CD Q0 QN0 D1 Q1 QN1
GTECH_FD28               26   2=>0                     | D0 CP CD Q0 QN0 D1 Q1 QN1
GTECH_FD2S                7   5=>0                     | D TI TE CP CD Q QN
GTECH_FD3                 6   4=>0                     | D CP CD SD Q QN
GTECH_FD34               15   3=>0                     | D0 CP CD SD Q0 QN0 D1 Q1
GTECH_FD38               27   3=>0                     | D0 CP CD SD Q0 QN0 D1 Q1
GTECH_FD3S                8   6=>0                     | D TI TE CP CD SD Q QN
GTECH_FD4                 5   3=>0                     | D CP SD Q QN
GTECH_FD44               14   2=>0                     | D0 CP SD Q0 QN0 D1 Q1 QN1
GTECH_FD48               26   2=>0                     | D0 CP SD Q0 QN0 D1 Q1 QN1
GTECH_FD4S                7   5=>0                     | D TI TE CP SD Q QN
GTECH_FJK1                5   3=>0                     | J K CP Q QN
GTECH_FJK1S               7   5=>0                     | J K TI TE CP Q QN
GTECH_FJK2                6   4=>0                     | J K CP CD Q QN
GTECH_FJK2S               8   6=>0                     | J K TI TE CP CD Q QN
GTECH_FJK3                7   5=>0                     | J K CP CD SD Q QN
GTECH_FJK3S               9   7=>0                     | J K TI TE CP CD SD Q
GTECH_LD1                 4   2=>0                     | G D Q QN
GTECH_LD2                 4   2=>0                     | D GN Q QN
GTECH_LD2_1               3   2=>0                     | D GN Q
GTECH_LD3                 5   3=>0                     | G D CD Q QN
GTECH_LD4                 5   3=>0                     | D GN CD Q QN
GTECH_LD4_1               4   3=>0                     | D GN CD Q
GTECH_LSR0                4   2=>0                     | R S Q QN
```

查找等效的 lib_cell

例如，查找两个输入端的与门(AND2)，在 IC Compiler 中可以用如下命令：

```
get_alternative_lib_cells [get_lib_cell gtech/GTECH_AND2]
```

IC Compiler 中的系统变量

列出系统变量

print_variable_group system

print_variable_group timing

print_variable_group power

print_variable_group test

report_app_var *

IC Compiler 中影响 Timing 计算的因素

Timing 计算总要涉及一个用哪个模型的问题。

1. Operating Condition 对应 PVT
2. 一个 PVT 可以对应多个 library 文件
3. 一个 library 可以有一个与之对应的 min version
4. 可以用 OCV(On Chip Variation)的方式使得 constraint 变得更为严格
5. 可以通过 set_timing_derate 进一步收紧 constraint

TODO

High Fanout Net Synthesis

High Fanout Net

- Clock Tree
- Reset Network
- Scan Enable Singal Network
- Normal High Fanout Net

查找 High Fanout Net

icc_shell> printvar high_fanout_net_threshold

high_fanout_net_threshold = "1000"

all_high_fanout -nets -through_buf_inv -threshold 1000

High Fanout Net 的处理

Ideal Network

在 design 进行 layout 之前，cell 的 location 是不确定的，这时对于 high fanout net 的 timing 估算是很不准确的。

实践中的作法是把 high fanout net 的处理推迟到后端，而在前端就把这些 net 当作理想中的连线，即 Ideal Network。
对于 ideal network，不需要进行优化，也不需要进行 DRC 检查，Timing 计算也按照理想值进行。

前端处理

set_ideal_network

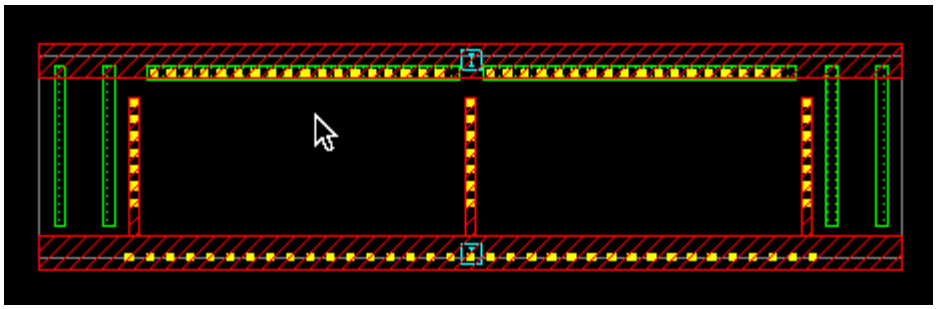
- DRC Check Free: max_capacitance, max_fanout, max_transition
- Latency and Transition Time are constant (default 0)
    - set_ideal_latency / set_ideal_transition
- size_only for cells on ideal network
- dont_touch for nets on ideal network

后端处理

remove_ideal_network

DeCap Cell

在 CMOS 电路运行过程中，信号的不断变化可能会导致电源电压的不稳定。这可以视为稳定的直流电压(DC)上加了一个交流(AC)干扰电压。
如果在电源的 Power 和 Ground 之间接入一个适当大小的 capacitor，利用 capacitor 的通交流阻直流的特性，可以平缓电源电压的波动。这个电容被称为 Decoupling Capacitor(解耦电容)。
ASIC 设计中用来实现这个目的的 cell 被称为 Decap Cell。由于它自身没有逻辑功能，属于 physical only cell 的一种。
Decap Cell 的内部实现很显然就是一个连接 Power 和 Ground 的 capacitor。
Decap Cell 通常放置在 row 的两端，或者 hard macro 的附近。
下图是一个 Decap Cell 和 Tap Cell 的组合体，通常可以用作 Endcap Cell。



上方的绿色矩形是 poly，和中间红色的 M1 组成所谓的 Diffusion Capacitor（N Well Capacitor 的一种实现）。这个 capacitor 的两端分别和 Power 和 Ground 相连。

Tap Cell

N Well 和 P-Substrate 形成了一个 PN 结，为了避免这个 PN 结被正向偏置(Forwad Bias)，最容易的办法是把 N-Well 连接到最高电压上，同时把 P-Substrate 连接到最低电压上。为了这个目的的连接被称做 Well Ties 和 Substrate Ties，有时也叫做 Well Contacts 或才 Substrate Contacts。
如果上面提到的 PN 结被导通的话，很可能会形成 Latch-Up 效应，从而损坏整个芯片。

在《集成电路版图基础》里这样形容 Well Ties 和 Substrate Ties:

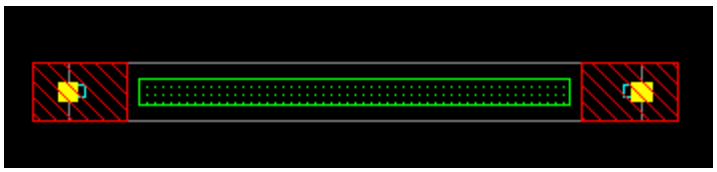
- Rule of Thumb: There is no such thing as too many tie-downs.
- Rule of Thumb: Wherever there is any spare space in an N well, put in a well tie. Wherever there is any space in the substrate, put in a substrate tie.
- Rule of Thumb: Place your well ties and substrate contacts before you do any wiring.

其中一种放置 ties 的策略就是规定每隔一定间距，就需要放置一个 well contact 和 substrate contact。所谓 tap cell 就是用来实现这一目的的。

设计上也可以直接把 well tie 集成在标准单元中，这种情况就不需要单独的 tap cell 了。

实践中，tap cell 在 floorplan 阶段就应该放置好了。在 IC Compiler 中，可以用命令 add_tap_cell_array 和 insert_tap_cells_by_rules 来实现。

下图是一个 tap cell 的 FRAM View 视图。中间绿色的部分是“冗余”的 poly。



一个用 group bound 优化 timing path 的例子

有一条 timing path 的其中一段如下所示：

```
...
buf_SMC_2/A (THHBUFFXT)        0.013 *    21.471 f      (1126.96,3018.00)
buf_SMC_2/Y (THHBUFFXT)        0.263      21.735 f      (1130.33,3018.60)
buf_SMC_N_2 (net)              0.000      21.735 f      # dont_touch,long wire
buf_SMC_1/A (THHBUFFXT)        9.619 *    31.353 f      (1279.84,-4059.60)
buf_SMC_1/Y (THHBUFFXT)        1.145      32.499 f      (1283.21,-4060.20)
buf_SMC_N_1 (net)              0.000      32.499 f      # dont_touch
buf_SMC/A (THHBUFFXT)          0.006 *    32.504 f      (1364.60,-4071.60)
buf_SMC/Y (THHBUFFXT)          0.121      32.625 f      (1367.97,-4071.00)
...
```

一个 buffer 放置在 design 的顶部，然后纵向“长途跋涉”7000nm 多，穿过一个 thin channel 来到 design 的底部，造成了 9ns 多的 delay。而且由于 net 被标注为 dont_touch，不能进行优化。

好的布局(placement)结果应该是把 3 个 buffer 放在一起。

这固然是因为工具没有足够“聪明”，我们也可以通过更好的 constraints 来引导工具做得更好。

```
create_bounds -name buf_SMC -effort medium    [get_cell {buf_SMC_2 buf_SMC_1 buf_SMC}]
```

Buffer/Inverter 的选择对 QoR 的影响

一个实际例子,除了 AHFS(High Fanout Synthesis)时选用的 buffer/inverter 范围有所不同，其他条件相同。得到的结果却差异很大——主要是 Routing DRC 的数目。

```
set bufs { TM7BUFXC TM7BUFXH TM7BUFXJ TM7BUFXL
           TM7BUFXM TM7BUFXN TM7BUFXP TM7BUFXQ }
set invs { TM7INVXC TM7INVXH TM7INVXJ TM7INVXL
           TM7INVXM TM7INVXN TM7INVXP TM7INVXQ }

set_ahfs_options -default_reference [concat $bufs ]                ;# Flow A
set_ahfs_options -default_reference [concat $bufs TM7INVXP ]   ;# Flow B
set_ahfs_options -default_reference [concat $bufs TM7INVXC ]   ;# Flow C
set_ahfs_options -default_reference [concat $bufs $invs ]        ;# Flow D
```

下面是做完 post-route hold fixing 后的结果（timing 结果相当）：

Flow	A	B	C	D
Buf/inv cell count	272534	285452	299507	267999
Route drc violations	694	4327	115	91

从 routing 的角度来看，Flow D 的结果最好，buffer/inverter 的数目有较大的减少。

"Flow B"结果不好，是因为 TM7INVXP 的 size 太大了。换成小一些的之后，"Flow C"给出了还不错的结果。

Timing Related Script

列出存在 Timing Path 的 clock

```
proc list_clock_group {} {
    set all_clocks [get_clock *]
    echo "# clock = [sizeof_collection $all_clocks]"
    foreach_in_collection clk_from $all_clocks {
        set name_from [get_object_name $clk_from]
        foreach_in_collection clk_to $all_clocks {
            set name_to [get_object_name $clk_to]
            # Skip same clock
            if {$name_from == $name_to} continue
            set timing_path [get_timing_path -from $clk_from -to $clk_to]
            set n_path [sizeof_collection $timing_path]
            if {$n_path]>0} {
                echo "TIMING PATH: $name_from -> $name_to : $n_path"
```

```
            }
        }
    }
}
```