



# 芯动力——硬件加速设计方法

## 第四章 逻辑综合(5)

邸志雄@西南交通大学

zxdi@home.swjtu.edu.cn



DC进行优化的目的是权衡timing和area约束，以满足用户对功能，速度和面积的要求。

- 优化过程是基于用户为design所加载的约束。

design rule constraint

transition

fanout

capacitance

optimization constraint

delay

area

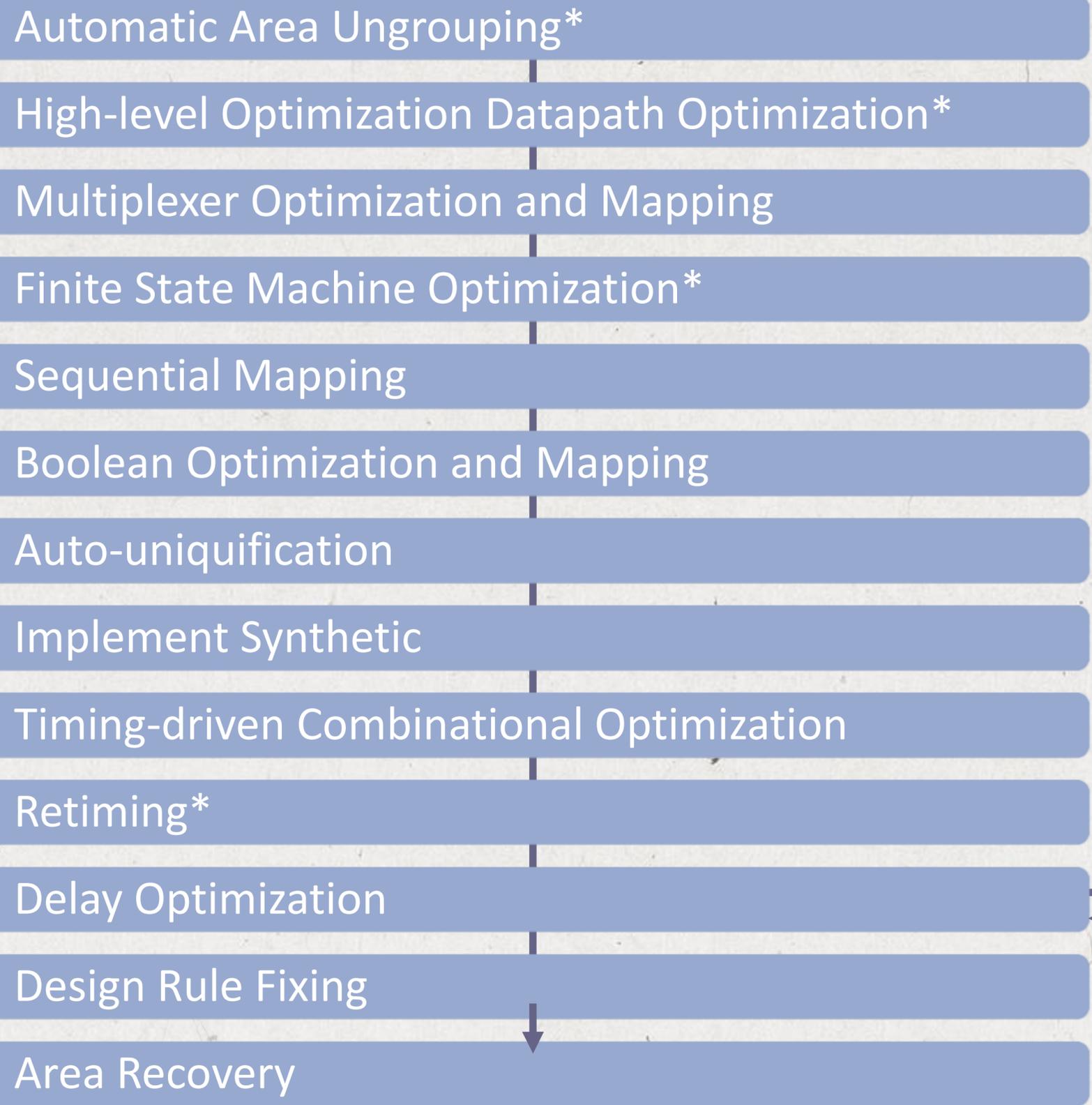
- DC默认DRC约束有较高优先权，必须先于optimization 约束满足。



## DC有很多的优化策略:

- 对数据通道的优化
- 对状态机的优化
- 对布尔逻辑的优化

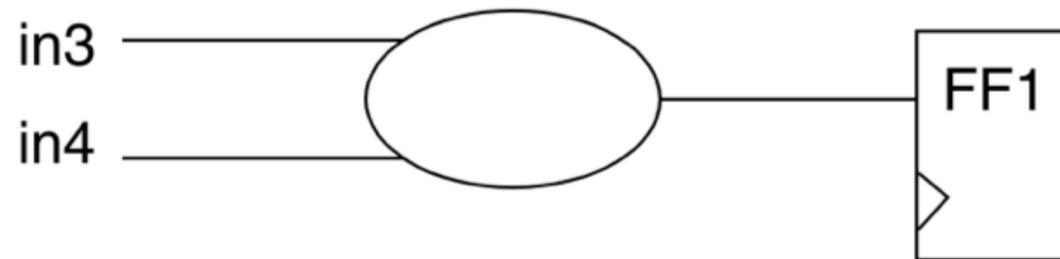




## Creating Path Groups

- 默认情况下，DC根据不同的时钟划分path group。但是如果设计存在复杂的时钟，复杂的时序要求或者复杂的约束，用户可以将所关心的几条关键路径划分为一个path group，指定DC专注于该组路径的优化。
- 也可以对不同的组设置不同的权重，权重的值范围为0.0—100.0

### *Path Group Example*



如果从in3到FF1的路径要设置为最高优先权，可以使用以下指令：  
`命令:shell> group_path -name group3 -from in3 -to FF1/D -weight 2.5`

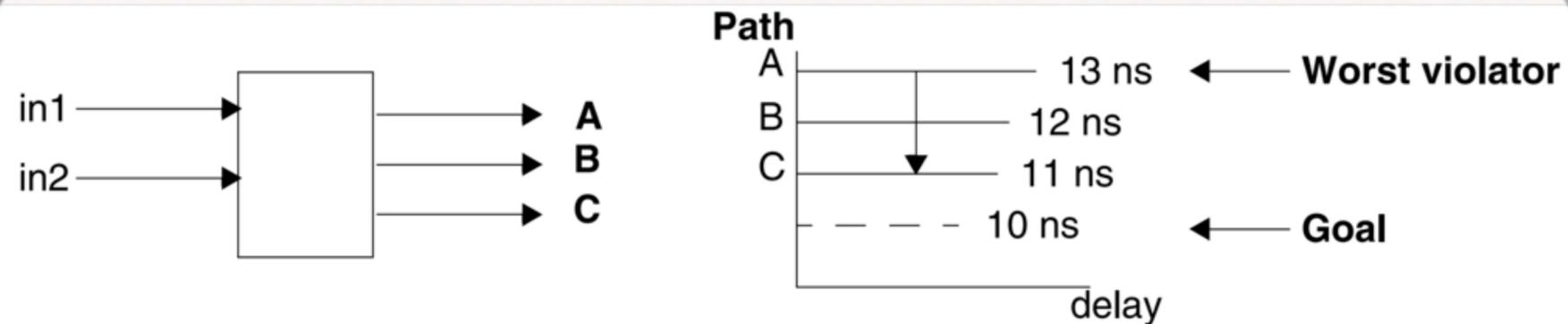


## Optimizing Near-Critical Paths

- 默认情况下，DC **只优化关键路径**，即负slack最差的路径。如果在关键路径附近指定一个范围，那么DC就会优化指定范围之内内的所有路径。若指定范围较大，会增大DC运行时间，因此一般情况该范围设定为时钟周期的10%。

使用以下指令设定关键路径的范围：

- Use the `-critical_range` option of the `group_path` command
  - Use the `set_critical_range` command



## Optimizing Near-Critical Paths

例：假定时钟周期为20ns，每条路径的最大延迟为10ns。默认DC只优化最差路径A。

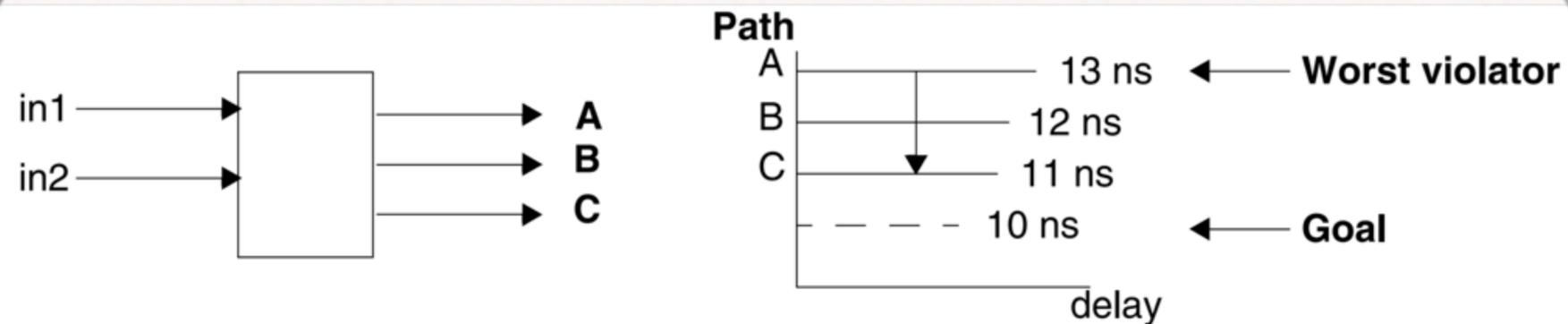
- 为了优化所有的路径，可设定关键路径范围为3.0ns。

```
create_clock -period 20 clk
```

```
set_max_delay 10 {A B C}
```

```
set_critical_range 3.0 $current_design
```

```
group_path -name group1 -to {A B C}
```



## Performing High-Effort Compile

- **High-effort compile** 能够让DC更加努力地达到所约束的目标，该措施在关键路径上进行重新综合，同时对关键路径周围的逻辑进行了 **restructure** 和 **remap**。

### High-effort指令有两种：

#### compile\_ultra command

- 附带两个option，这两个option分别包含一些脚本，提供额外的时序和面积优化，option为 **-area\_high\_effort\_script option** 和 **-timing\_high\_effort\_script**。

#### compile command

- 附带一个option， **map\_effort -high option**。



## Performing High-Effort Compile

High-effort对关键路径的重新优化包括：逻辑复制和映射为大扇入的门单元（Logic Duplication and Mapping to Wide-Fanin Gates）。

- 如果用一些工艺库中结构复杂、扇入较大的门单元代替关键路径上的一些标准单元，能够更好地电路的时序和面积，DC会在关键路径上做相应的优化。
- 此外，对于一些高扇出或者高负载的线，DC会通过逻辑复制和重新构造（restructuring）这部分电路来进行优化。这些逻辑复制和重新构造通常会导致设计在面积上有显著的增加。



## Performing a High-Effort Incremental Compile

- 通常，使用incremental可以提高电路优化的性能。如果电路在compile之后不满足约束，通过incremental也许能够达到想要的结果。
- Incremental只进行门级 (gate-level) 的优化，而不是逻辑功能级 (logic-level) 的优化。它的优化结果可能是电路的性能和之前一样，或者更好。

Incremental会导致大量的计算时间，但是对于将最差的负slack减为0，这是最有效的办法。



## Performing a High-Effort Incremental Compile

Incremental会导致大量的计算时间，但是对于将最差的负slack减为0，这是最有效的办法。

- 为了减少DC运算时间，可将那些已经满足时序要求的模块设置为dont\_touch属性。

```
dc_shell> dont_touch noncritical_blocks
```

```
dc_shell> compile -map_effort high -incremental_mapping
```

- 对于那些有很多违例逻辑模块的设计，通常Incremental最有效。



## Gate-Level Optimizations

门级优化主要通过选择库中的合适的标准单元来对电路进行优化，分为三个阶段执行：

delay optimization

DC对电路进行局部的调整

- DC已经开始考虑DRC，同样条件下，会选择DRC代价最小的方案。

design rule fixing

DC主要通过插入buffer，调整单元的大小等措施来满足各种DRC的约束。

- 一般不会影响时序和面积结果，但是会引起optimization constraints违例。

area recovery

不会引起DRC和delay的违例，一般只是对非关键路径进行优化。

- 如果没有设置面积约束，那么优化的幅度会很小。



## Automatic Ungrouping

- Ungrouping取消设计中的层次，移除层次的边界，并且允许DC Ultra通过减少逻辑级数改进时序，以及通过共享资源减小面积。

- 实现该功能

compile\_ultra

compile -auto\_ungroup

delay-based auto-ungrouping

area-based auto-ungrouping

需要单独的license  
才可以使用



## delay-based auto-ungrouping

- 默认情况下, `compile_ultra` 执行该种策略, 它主要是围绕关键路径进行一些优化。
- 在该情况下, `DesignWare` 中的结构是不会被优化的, 因为DC认为他们已经被优化的非常好, 在时序和面积上都没有可以再优化的可能。

## area-based auto-ungrouping



## mapping

DC会估算unmapped的层次, 然后移除一些小的子设计

### 目的

- 降低面积和提升时序性能

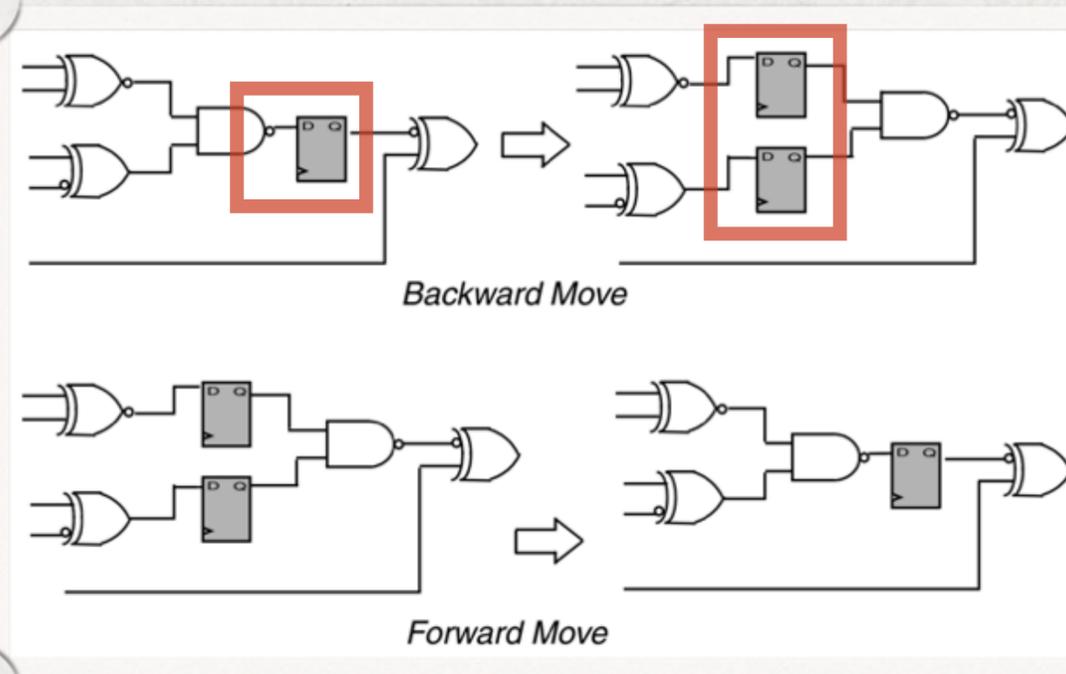
```
compile -autoungroup area
```



## Adaptive Retiming

### DC使用retiming策略

- 可在时序路径上前后移动寄存器，以提高电路的时序性能。



### retime在优化过程中

- 如果有违例的路径，则调整寄存器的位置。
- 如果没有违例的路径，则可用来减少寄存器的数量。



## Adaptive Retiming

DC在移动寄存器的优化中，只能对有相同时序约束的寄存器进行调整，如果两个寄存器约束不同，则不能一起移动。

- 移动后的寄存器在网表中，名字通常带有一个R的前缀，和一个系列号

R\_xxx

- 且retime策略不能和compile\_ultra的以下option一起使用：

-incremental

-top

-only\_design\_rule

除此三个option之外的，其他option均可以同时使用。

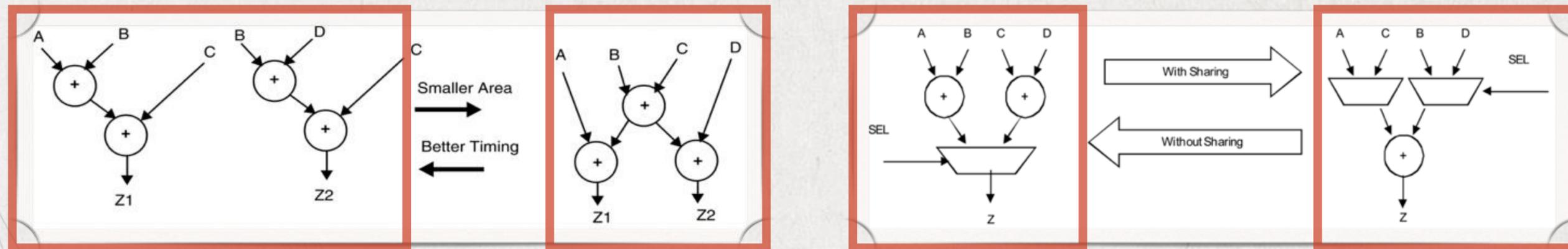


# High-Level Optimization and Datapath Optimization

DC对数据通路的优化，通常主要通过以下手段：

- 采用树形结构的运算逻辑，比如： $a+b+c+d$ ，优化为 $(a+b)+(c+d)$ ;
- 逻辑上的简化，比如： $(a*3*5)$ 简化为 $(a*15)$
- **资源共享**

先选后加，还是先加后选。要注意其中的差别。

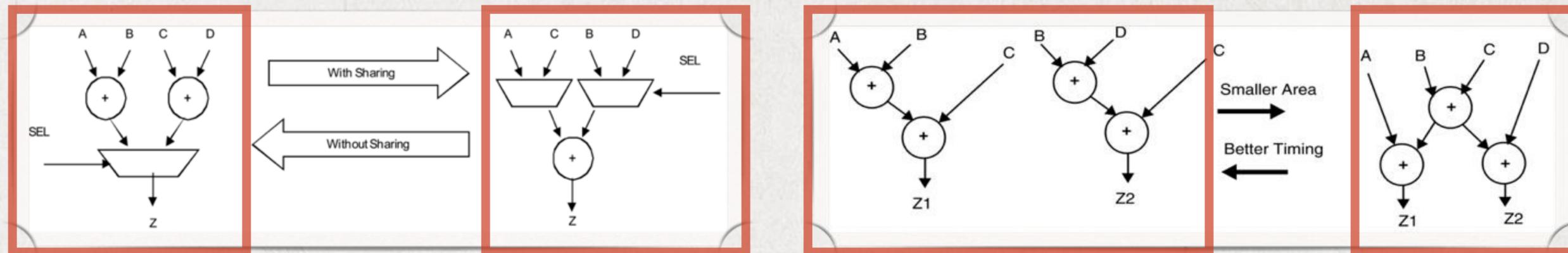


# High-Level Optimization and Datapath Optimization

DC对数据通路的优化，通常主要通过以下手段：

- 采用树形结构的运算逻辑，比如： $a+b+c+d$ ，优化为 $(a+b)+(c+d)$ ;
- 逻辑上的简化，比如： $(a*3*5)$ 简化为 $(a*15)$
- **资源共享**

先选后加，还是先加后选。要注意其中的差别。



## High-Level Optimization and Datapath Optimization

### DC Ultra对数据的优化主要通过以下手段：

- 使用design ware库。
- 数据路径提取：使用多个树形阵列的CSA的加法器代替数据通路中的加法运算，可大大提高电路的运算速度。但是，这只适合多个运算单元之间没有任何逻辑。同时，design ware中的单元不能被提取。
- 对加乘进行重新分配，比如： $(A * C + B * C)$ 优化为 $(A + B) * C$ 。
- 比较器共享，比如 $A > B, A < B, A \leq B$ 会调用同一个减法器。
- 优化并行的常数相乘。
- 操作数重排。



## Verifying Functional Equivalence

以下优化均会引起网表和RTL design不一致，因此需要使用formality工具进行一致性检查，确认不一致的地方是否由DC优化造成：

- 由ungroup, group, uniquify, rename\_design等造成部分寄存器，端口名字改变。
- 等效和相反的寄存器被优化，常量寄存器被优化。
- Retiming策略引起的寄存器，电路结构不一致。
- 数据通路优化引起的不一致。
- 状态机的优化。

因此，DC在综合过程中必须生成formality的setup文件（默认为default.svf）



## 设计划分 Partitioning for Synthesis

### What Is Partitioning?

- 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。

## 设计划分 Partitioning for Synthesis

### What Is Partitioning?

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。

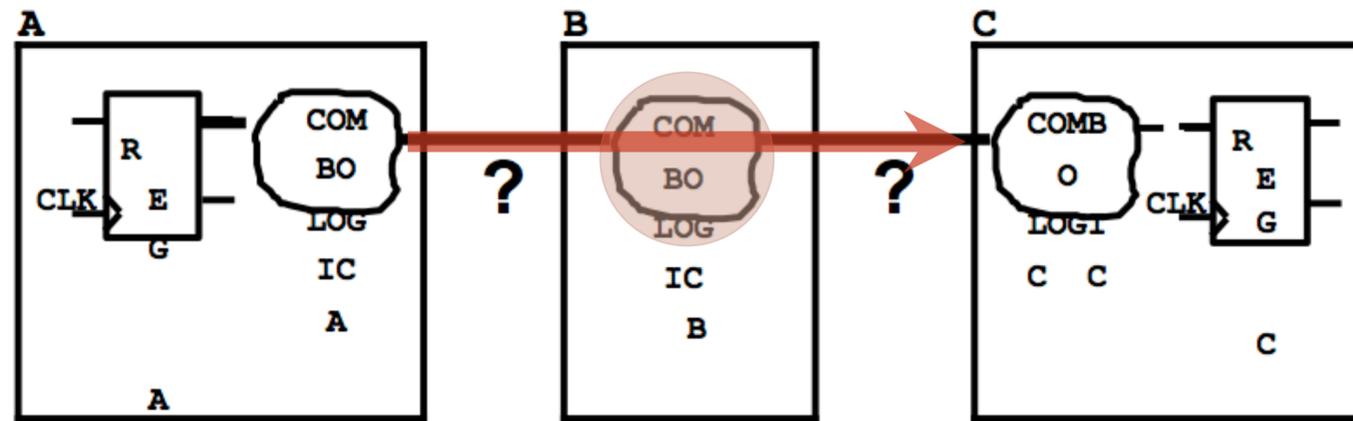
比如在优化的过程中

- 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。



## 设计划分 Partitioning for Synthesis

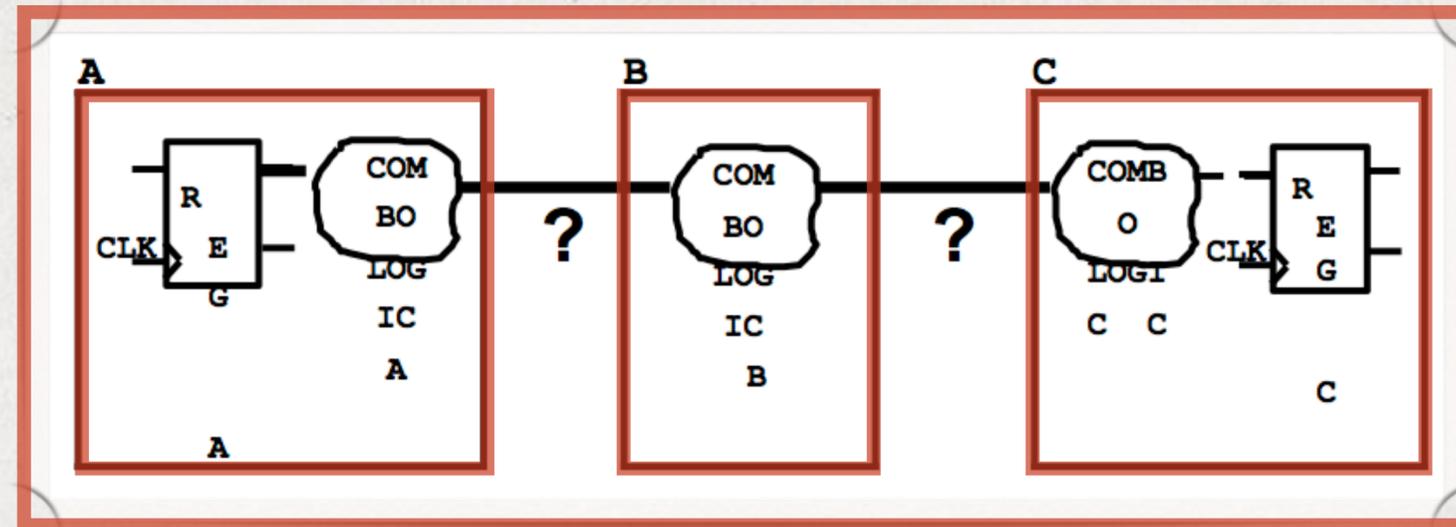
设计划分原则一：不要让一个组合电路穿越过多的模块



- 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。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。

## 设计划分 Partitioning for Synthesis

设计划分原则一：不要让一个组合电路穿越过多的模块

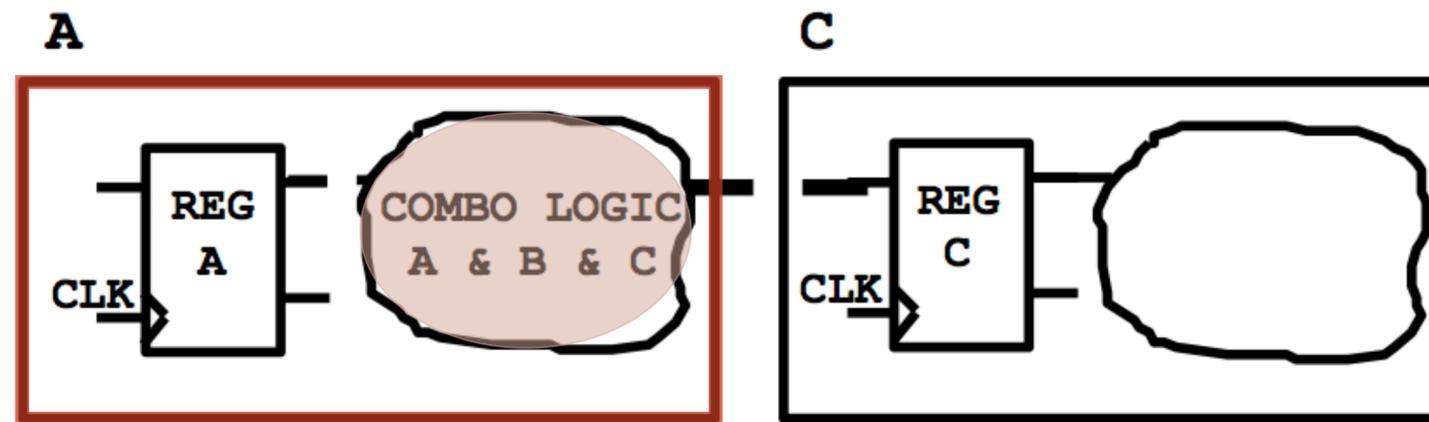


- 这样的话，**DC 在作组合电路的优化**的时候就会分别针对 A、B、C 三块电路进行，这样势必会影响到 DC 的优化能力，不必要的增加了这条路径的延时和面积。因此，可以考虑将三块分散的组合逻辑划分到一个模块中。

## 设计划分 Partitioning for Synthesis

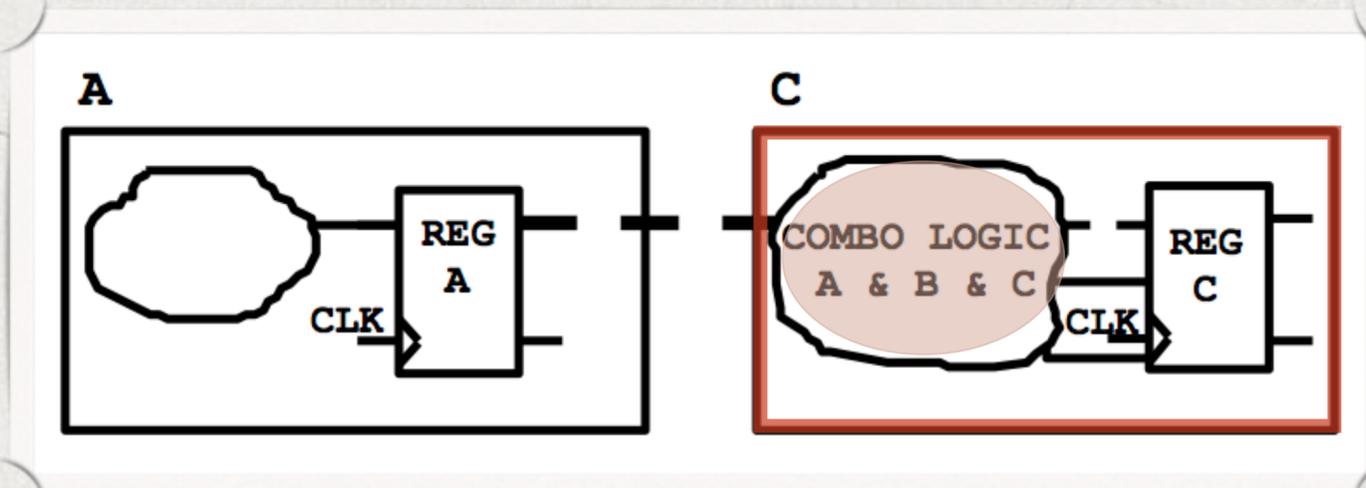
设计划分原则一：不要让一个组合电路穿越过多的模块

- 考虑将三块分散的组合逻辑划分到一个模块中。



Better Partitioning





Best Partitioning

组合的最佳优化

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。

D 触发器



JK 触发器

T 触发器

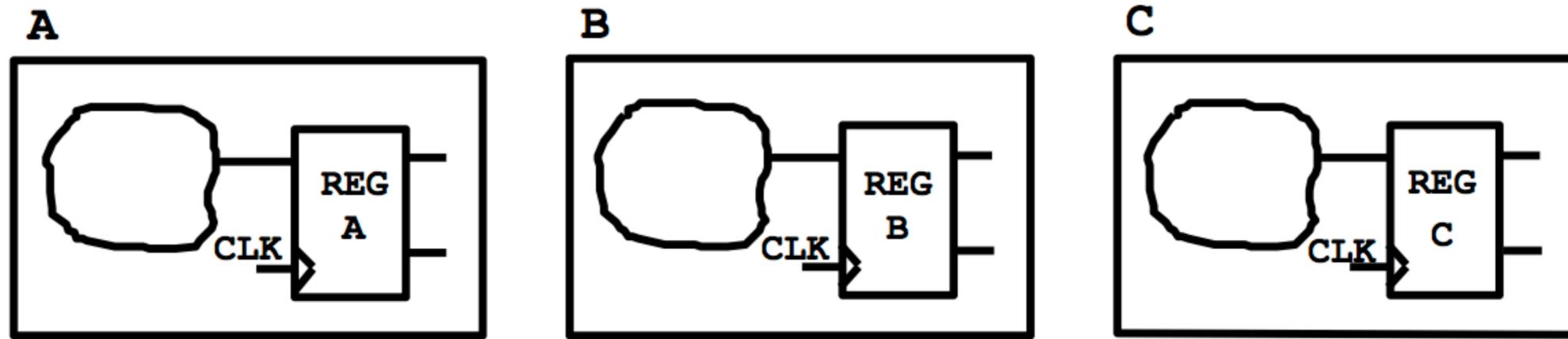
带时钟使能端的触发器

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。



# 设计划分 Partitioning for Synthesis

## 设计划分原则二：寄存模块的输出



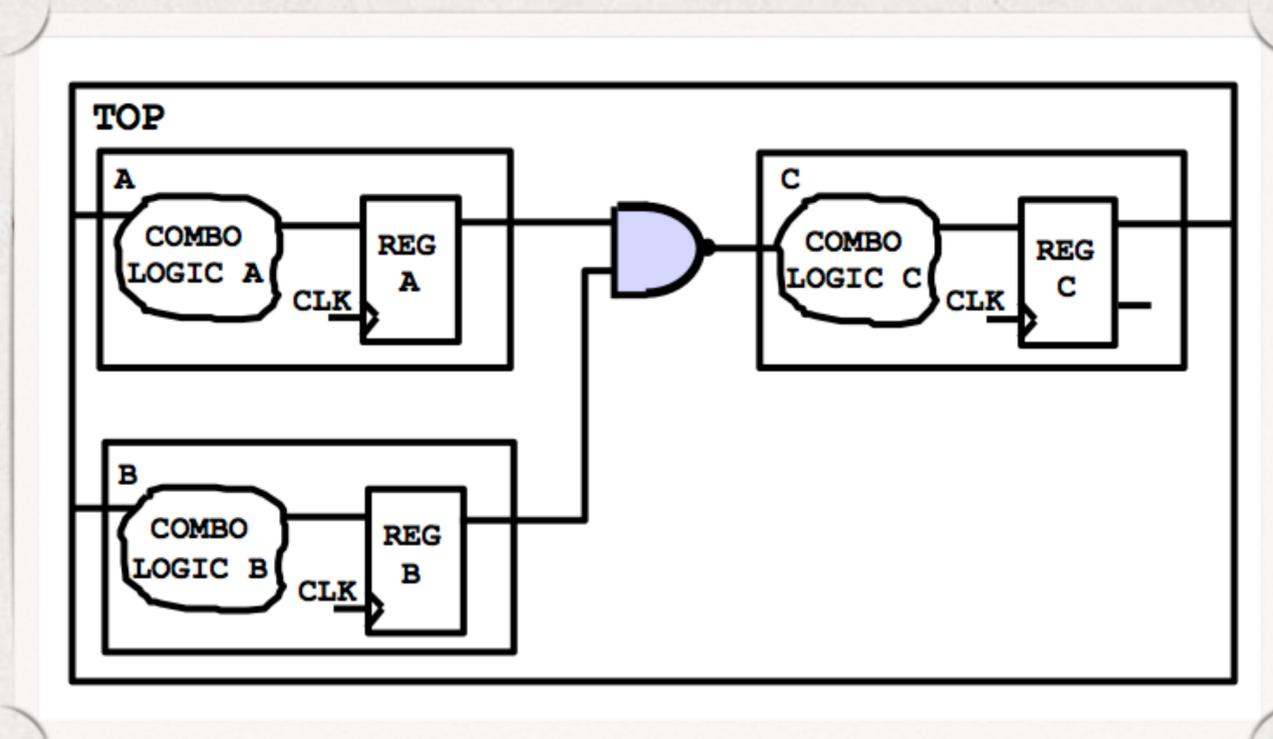
- 代码或者综合的过程中，将所有的输出寄存起来。其实这样不但是最佳的优化结构，也可以简化时序约束（使得所有模块的输入延时相等）



# 设计划分 Partitioning for Synthesis

## 设计划分原则二：寄存模块的输出

- 就算遵循了输出寄存的原则，我们还是可能犯下面的错误。

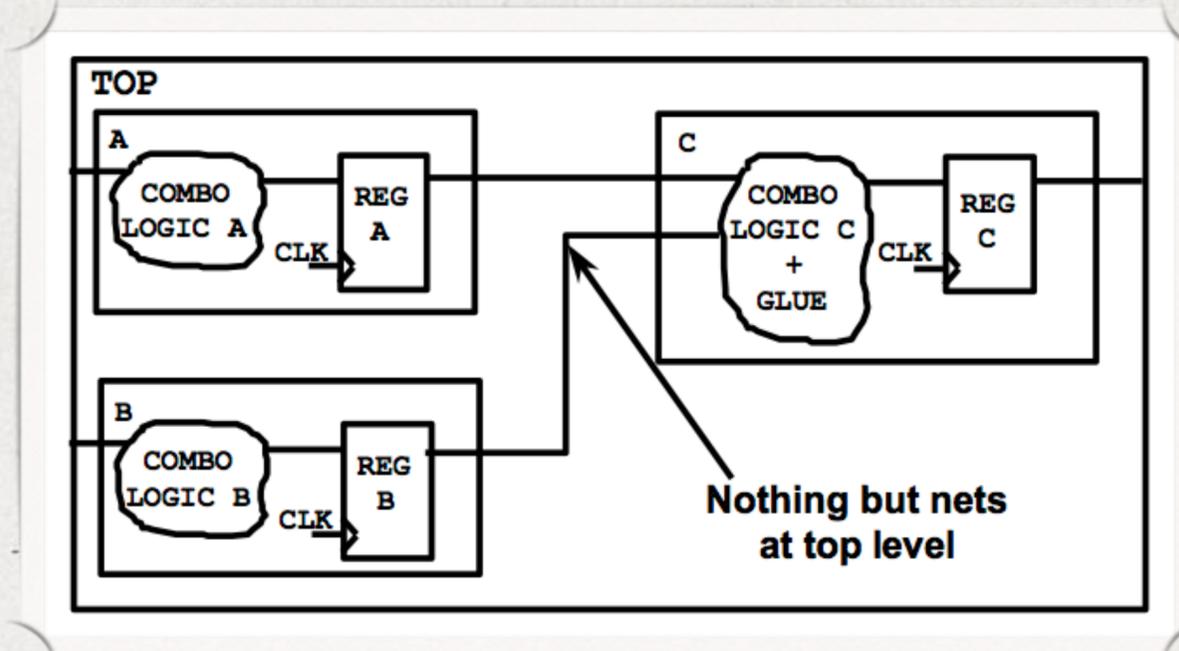


Poor Partitioning



# 设计划分 Partitioning for Synthesis

## 设计划分原则二：寄存模块的输出

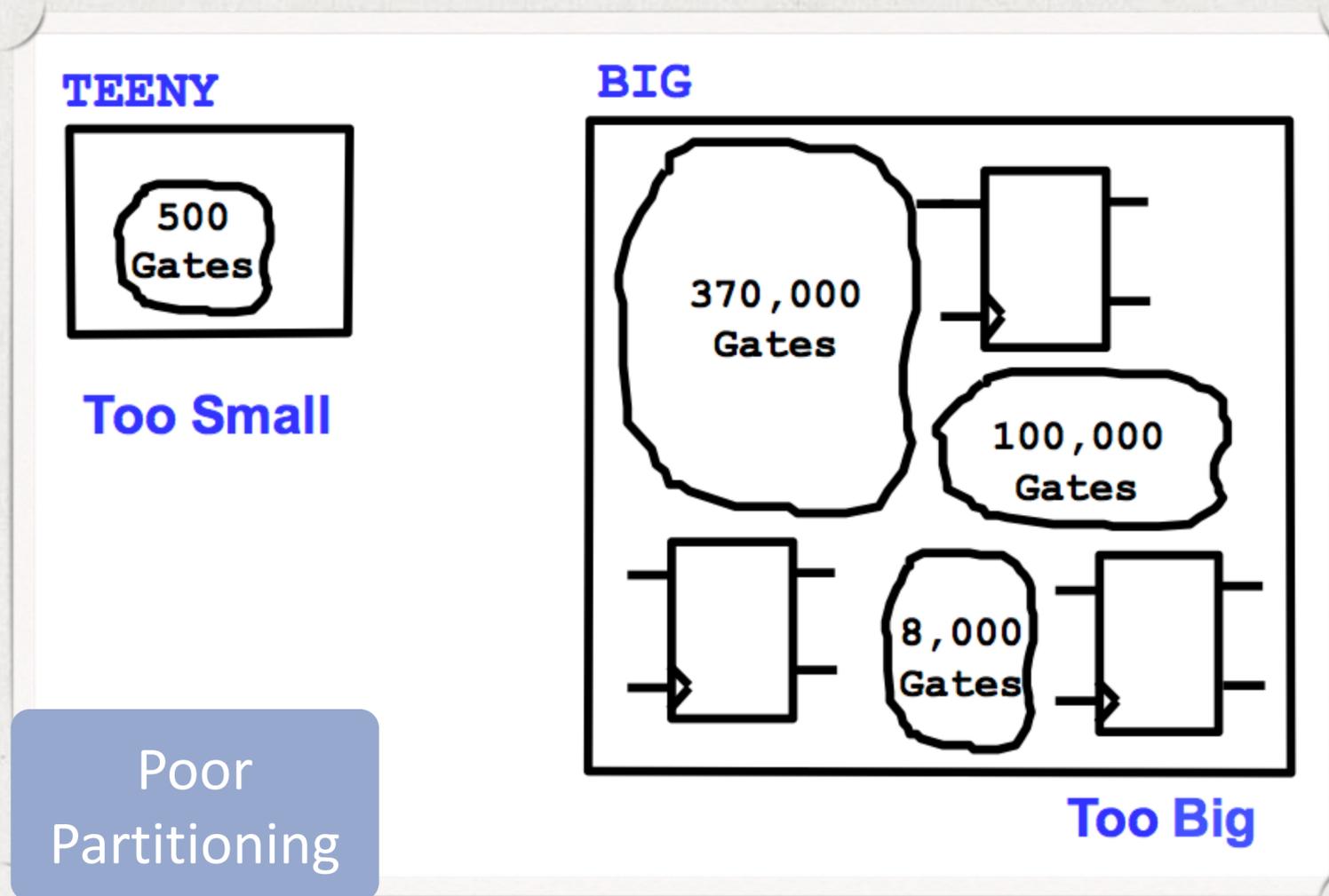


Good Partitioning

- 把与非门吸收到 C 中的组合逻辑的方法消除粘滞逻辑，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。

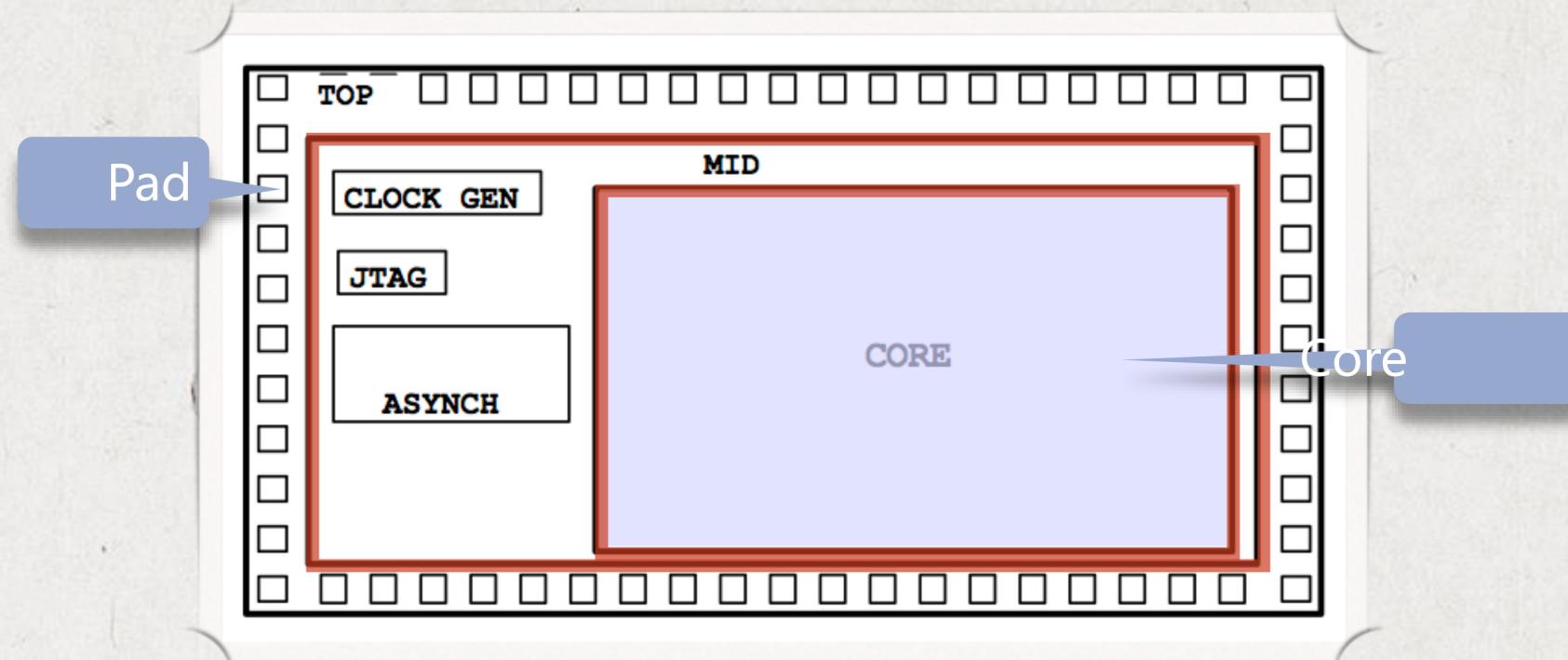
# 设计划分 Partitioning for Synthesis

设计划分原则三: 根据综合时间长短控制模块大小



## 设计划分 Partitioning for Synthesis

原则四：将同步逻辑部分与其他部分分离

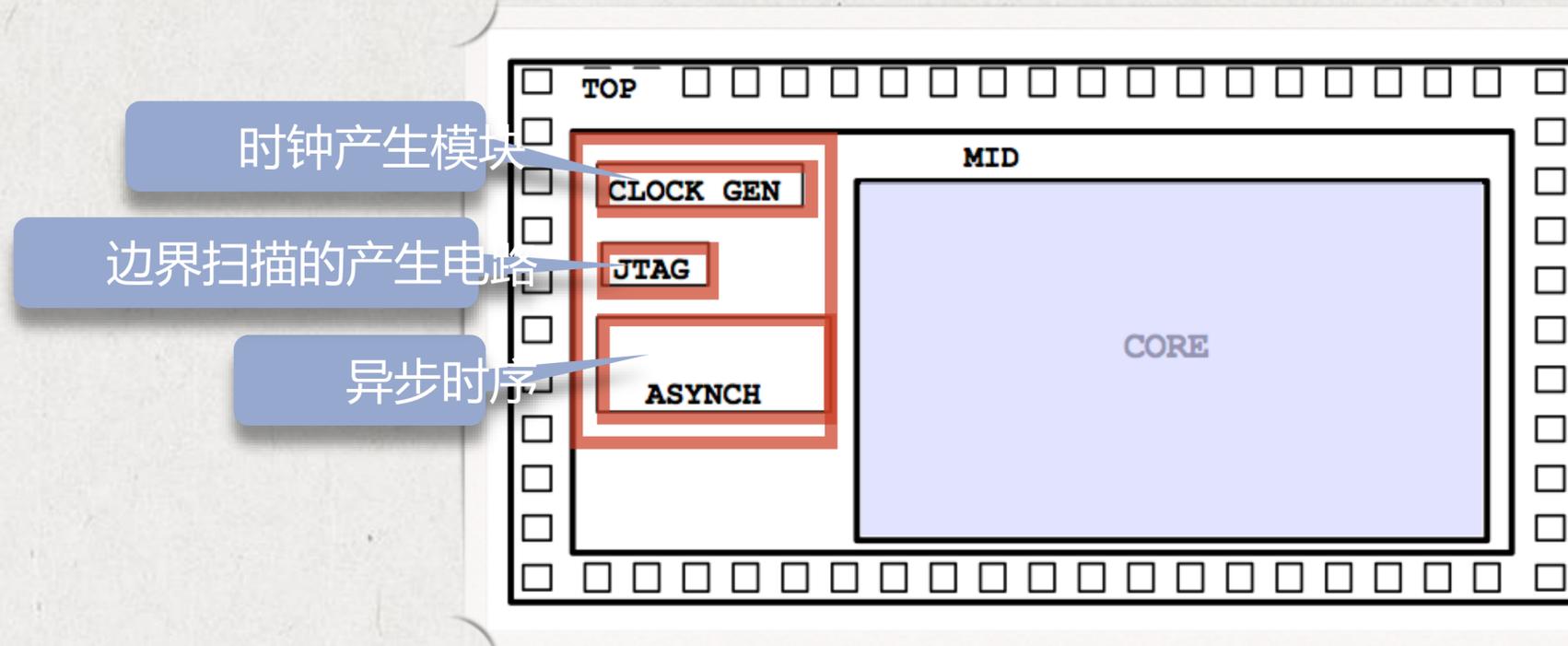


- 最外边是芯片的 Pad，Pad 是综合工具中没有的，也不是工具能生成的，它由 **Foundry 提供**，并由设计者根据芯片外围的环境手工选择；中间一层被分成四个部分，其中最里面那个称为 Core，也就是 DC 可以综合的**全同步逻辑电路**。



## 设计划分 Partitioning for Synthesis

原则四：将同步逻辑部分与其他部分分离



- 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。