

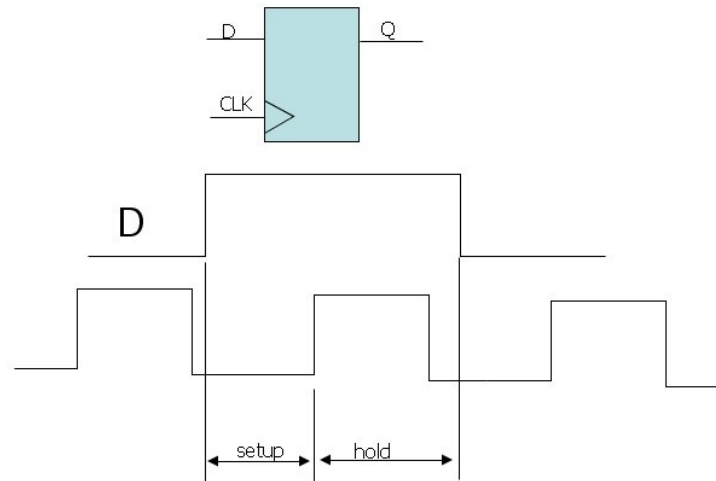
DC 概论全总结

DC 概论之一 setup time 与 hold time	1
DC概论二之fanout与skew	5
DC概论三之setup time 与 hold time 之二	14
DC概论四之setup time 与hold time 之三	20
DC概论五之high fanout.....	37
DC 概论六之multicycle_path	59
DC概论七之gated clock.....	78
DC概论之IO约束.....	90
DC优化约束	99

DC 概论之一 setup time 与 hold time

2009-03-13 10:49:56 来源：网络转载作者：佚名共有评论(0)条浏览次数:521

ic 代码的综合过程可以说就是时序分析过程，dc 会将设计打散成一个个路径，这些路径上有 cell 延迟和 net 延迟，然后 dc 会根据你加的约束，来映射库中符合这种延迟以及驱动的器件。从而达到综合的目的。dc 的所有时序约束基础差不多就是 setup time 和 hold time。 可以用下面的图片说明：



所谓 setup time 即建立时间，也就是说数据在时钟到来之前保持稳定所需要的时间，

hold time 即保持时间，也就是说在时钟到来之后数据需要保持稳定的时间。

在深入建立时间和保持时间之前。先了解下 dc 中的路经以及 start point , end point。

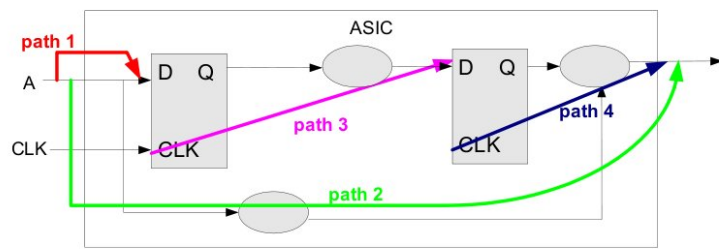
所谓 start point 就是：

1. input port (顶层设计的输入端口)
2. clock pin of sequential cell (触发器的 clock pin)

所谓的 end point 就是：

- 1 output port (顶层设计的输出端口)
3. data pin of sequential cell (触发器的 data pin)

了解 start point 和 end point，就可以方便的了解 dc 是如何将设计打散成路经，一个设计中基本的路经分为 4 种，如下图：



path1: input port to data pin of sequential cell

path2: input port to output port

path3: clock pin to data pin of next sequential cell

path4: clock pin to output port

所有的设计也就这四种类型的路径。

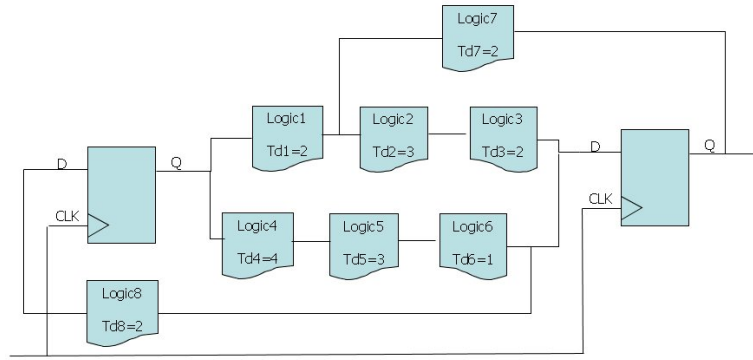
另外一种就是带反馈的，也就是本身的 clock pin to data pin 。

有了路径的概念之后，我们可以分析更复杂的 setup time 和 hold time 。

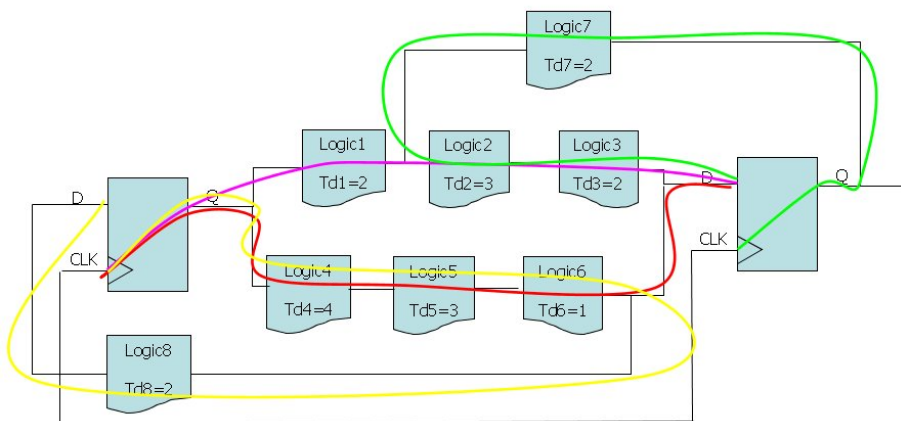
dc 中对于建立时间的分析是基于路径的最大延迟；而对于保持时间的分析是基于路径的最小延迟。

看下面一个例子：

这个是给定 setup time 和 hold time 的案例，要求算出最小时钟周期。同理也可以给你一个周期和 setup time 和 hold time，计算时间裕度。 我们假设时钟周期是 20, 每个触发器的 cell 延迟是 1, 触发器的建立时间是 1, 保持时间是 0.5, 分析下列图中的建立时间和保持时间的 slack。



看到设计，首先要分析路径，找出最长和最短路径，因为 dc 的综合都是根据约束而得到最短和最长路径来进行器件选择的。所以接下来将图中的所有路径标出。因为没有前级 (input_delay) 和后级电路 (output_delay)，我们只分析图中给出的 路径，如下图：



对于红色路径： $T_d = T_{cell} + T_{d4} + T_{d5} + T_{d6} = 1 + 4 + 3 + 1 = 9$

对于黄色路径： $T_d = T_{cell} + T_{d4} + T_{d5} + T_{d6} + T_{d8} = 1 + 4 + 3 + 1 + 2 = 11$

对于粉色路径： $T_d = T_{cell} + T_{d1} + T_{d2} + T_{d3} = 1 + 2 + 3 + 2 = 8$

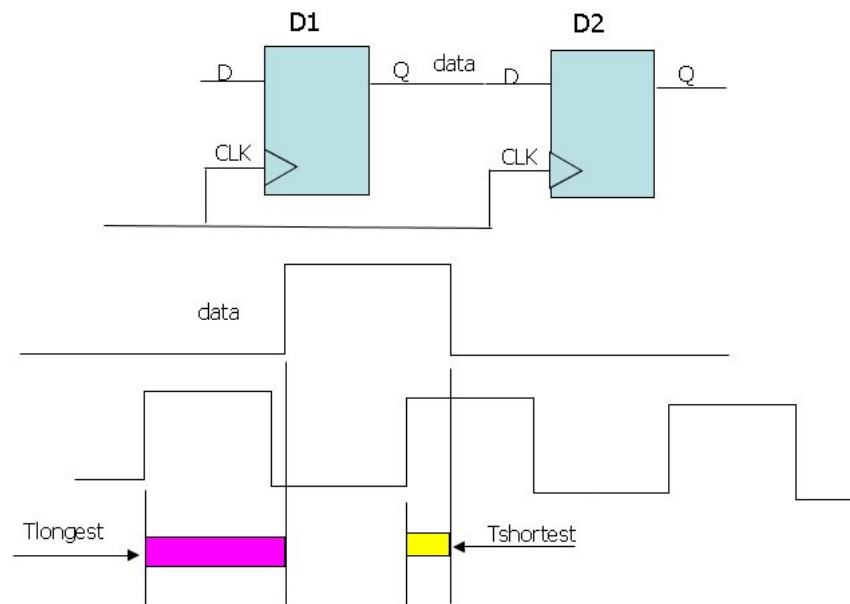
对于蓝色路径： $T_d = T_{cell} + T_{d7} + T_{d2} + T_{d3} = 1 + 2 + 3 + 2 = 8$

所以 $T_{longest} = 11, T_{shortest} = 8$

对于 setup time 的 slack: $T_{clk} - T_{longest} - T_{setup} = 30 - 11 - 1 = 18$

对于 hold time 的 slack : $T_{shortest} - T_{hold} = 8 - 0.5 = 7.5$

对于 setup time 和 hold time 的 slack 的计算，可以体会下面的示意：



对照第一副示意图与此比较，建立时间看 D2，保持时间看 D1, 因为同时把 T1 和 Ts 放在一个图例中，看起来可能有些误解：)

有空会继续讨论 setup time 和 hold time，下次讨论将包括 clock skew 和 input delay, output delay 在其中。

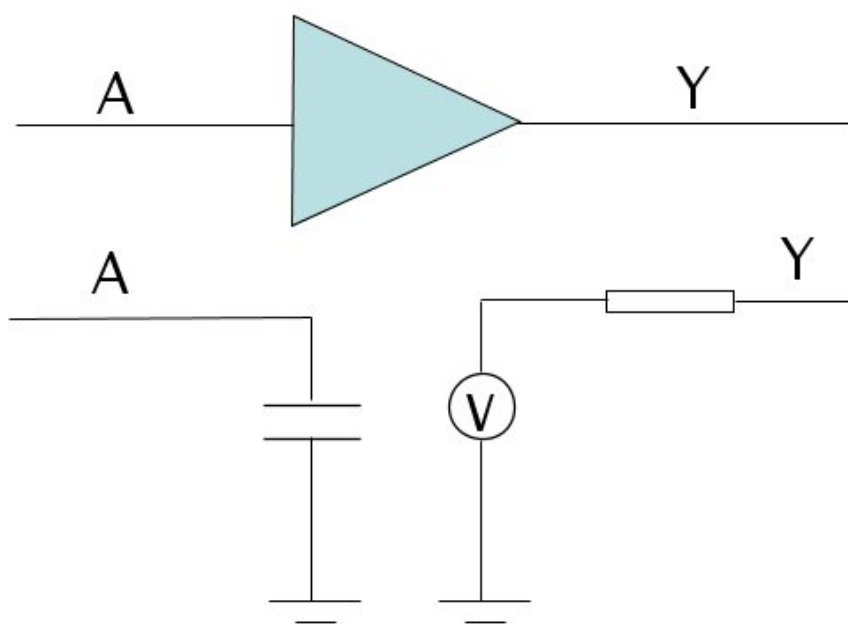
DC 概论二之 fanout 与 skew

SOC debug 中 Dc 综合是基于路径，每个路径上都有 Cell 和 net，所以基于路径的综合就是计算路径上的 delay 和 rc（dc 是使用互连线模型进行估算）。

在了解 delay 和 rc 的计算时，我们要先了解下一个 cell 对于 drive（前级）和 driven（后级）所用到的模型是什么。如下图，一

个 buffer，从前级看过来是一个 load (capacitance, 想获得这个 load, 可以通过 load_of buffer/a 获得), 从后级看来是一个 drive (resistance)。电路的驱动能力是上一级的 $1/R$, 即电阻的倒数, 驱动能力大, 说明看过去的电阻小, 也说明这个器件比较大 (大器件有较大的驱动能力)。电路的负载能力是下一级的 load (即电容) 总和, 负载能力大, 说明能驱动下级的期间就很多。

大器件是大电容, 小电阻, 而小器件是小电容, 大电阻。理解这些, 对于 dc 综合以及后端 apr 版图都有很好的操作。



对于 cell 的延迟, dc 是根据 input_transition 和 out_load 对应的查找表来计算的。

对于 net 的延迟, dc 是根据 wire_load_model 中的 fanout_length 和 resistance, capacitance, area 的查找表计算的。

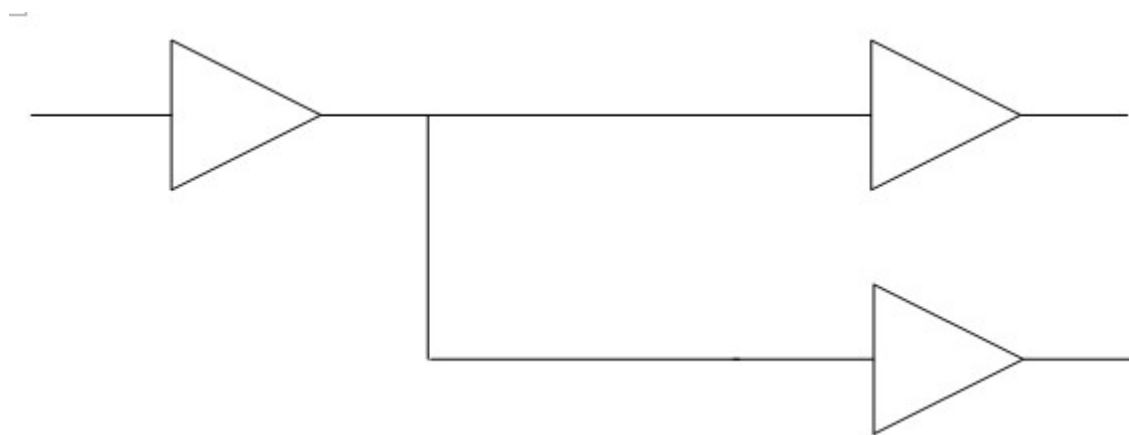
例如:

```

Wire_load(small) {
Resistance : 0.2;
Capacitance : 1.0;
Area :0;
Slop :1.0;
Fanout_length(1, 0.022);
Fanout_length(2, 0.046);
Fanout_length(3, 0.070);
Fanout_length(4, 0.095);
}

```

比如现在扇出是 2，



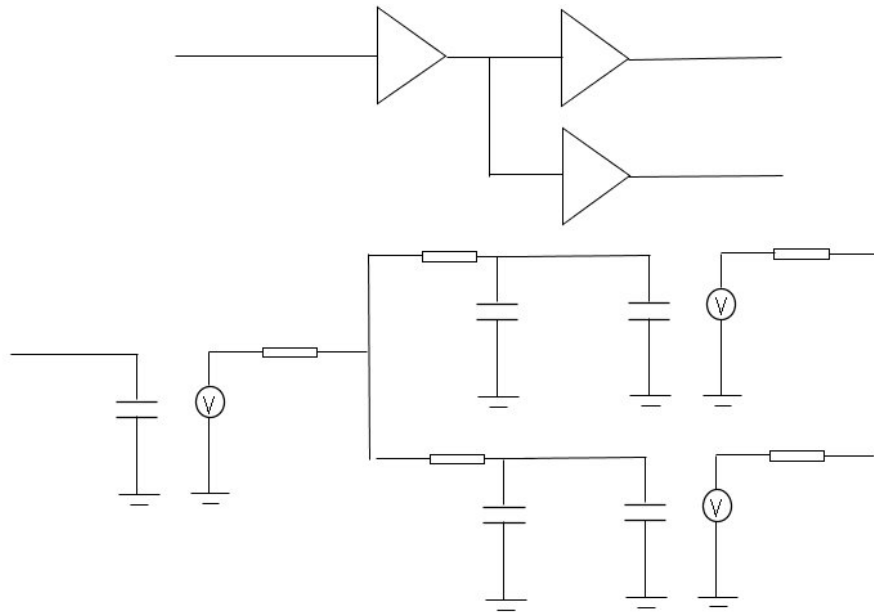
根据 fanout_length (2, 0.046) 可以知道这个互连线的长度是 0.046，然后再根据 capacitance, resistance 可以得出这个互连线的电容为：0.046x1.0, 互连线电阻为：0.046x0.02。

如果扇出是 5，在查找表中没有找到 fanout_length 为 5 的项，互连线长度将会是

$$= \text{fanout_length}(4, 0.095) + (5-4) * \text{slop} = 0.095 + 1 * 1.0 = 1.095$$

得出了 rc 就可以计算出信号的 transition 时间=2.2RC。

实际的互联线如下：



扇出线上的转换时间根据在版图之后提取的 rc 参数信息求得：

$$2.2RC = (R_{\text{net}} + R_{\text{out}}) \times (C_{\text{net}} + C_{\text{in}})$$

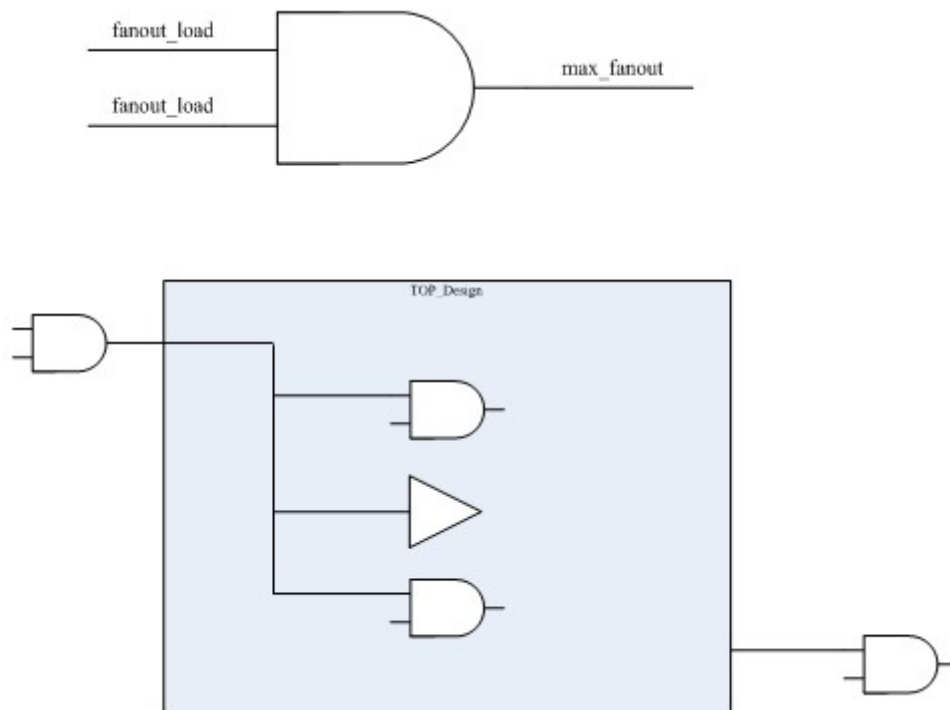
无论如何，要记住的就是 dc 是基于路径分析的，每个路径上有 cell 延迟和 net 延迟，而 cell 延迟是根据 input_transition 和 out_load 得出的，net 延迟是根据 fanout_length, resistance, capacitance 得出的。驱动和电阻成反比，负载和电容成正比。

由上面可以知道 fanout 影响到 load (capacitance)，transition, delay。了解了上面，我们来理解下 dc 中对 design 建模，所用到的一些跟 fanout 有关的参数。

Dc 中的约束，其实就是给 chip 设计一个环境，比如驱动这个 chip

输入端口的 cell，或者这个 chip 输出端口驱动了那些单元或者端口接入了哪些负载，以及这个芯片的工艺，电压，温度，等等。

对于一个 cell 来说，输出端口具有 max_fanout 属性，输入端口有 fanout_load 属性。



例如将一个 AND2 作为 design 的驱动 cell(set_driving_cell)，这样就把 AND2 的 max_fanout 属性加在了输入端口上。如果一个 AND2 的输出端口 max_fanout 是 5，输入端口 fanout_load 是 2。一个 buffer 输入端口的 fanout_load 是 3。那么这个 AND2 的输出端可以接 2 个 AND2，或者可以接一个 buffer，或者可以接一个 buffer 和一个 AND2。如上图所示，则会引起 DC 产生 DRC 错误。因为输入端口的 $\text{fanout_load} = 2 \times \text{AND2} + \text{buffer} = 7$ ，超过了 AND2 的 max_fanout2. 如果

使用了系统提供的 `set_max_fanout 5 [all_inputs]`, 将会忽略 `set_driving_cell` 中 `cell` 的 `max_fanout` 属性, 而使用 `set_max_fanout` 属性。

如果将一个 AND2 作为 design 的负载, 那么这个输出端口上的 `fanout_load` 属性将会为 2. dc 中一般的做法是 `set_fanout_load [expr [get_attribute slow/and2/a fanout_load] *xxx] [all_outputs]`, 来设置输出端口的 `fanout_load` 属性。

这样 dc 就可以根据这些设置, 选择优化端口处的器件, 以及时序。

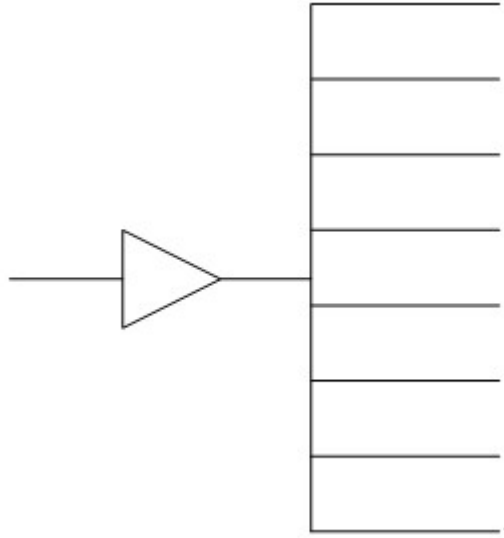
说道 fanout, 所以顺带说明一下容易混淆的 `max_fanout`, `max_capacitance`。如上图所示,

输入端口的 `fanout_load`=2 个与门的 `fanout_load` 和一个 buffer 的 `fanout_load`。

输入端口的 `load (capacitance)` =2 个与门的 `load_of` 和一个 buffer 的 `load_of`。(如果通过 `set_load` 设置了输入端口, 另外还要加上 `set_load` 的值)。

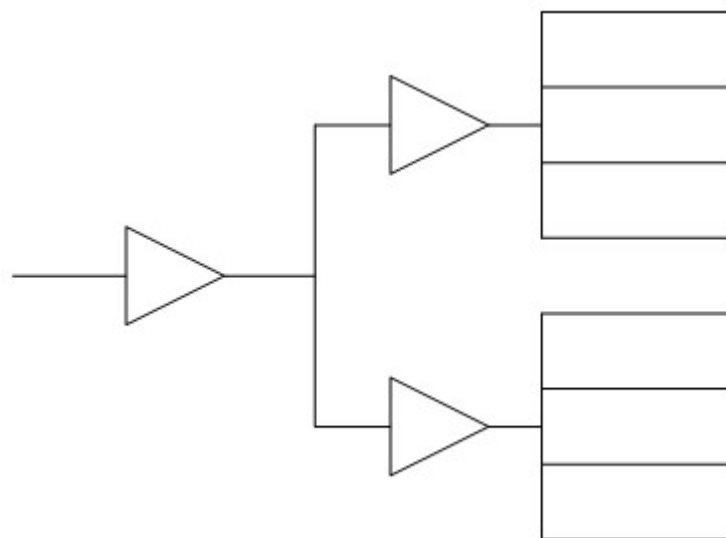
`Max_fanout` 检查的是输入端口的 `fanout_load` 最大值, `Max_capacitance` 检查的是输入端口 `load` 值。两者概念不同。

下面讲下 fanout 与 delay, 看如下一个例子:



到 buffer 的 net 延迟是 2, buffer 延迟是 1, fanout 为 1 时 net 延迟为 3, 每增加一个扇出, net 延迟增加 2. 如果一个信号经过这个扇出网络后, 那么延迟为: $2+1+(3+(8-1)\times 2)=20$;

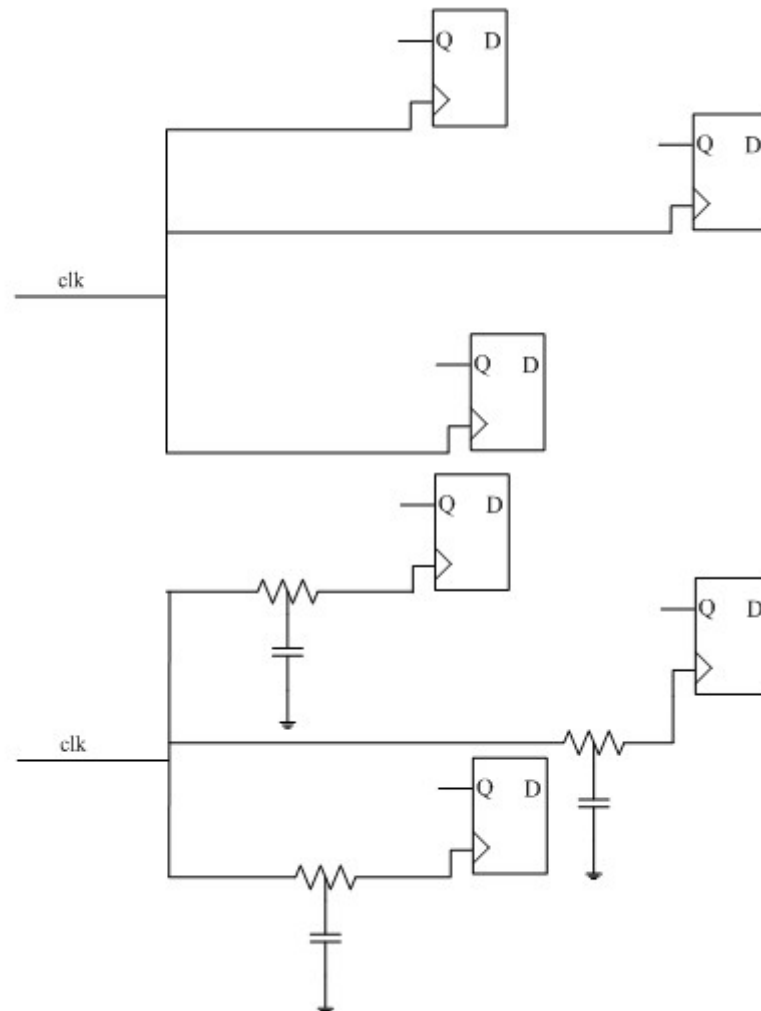
如果把扇出结构优化成如下形式:



那么信号经过这个网络后, 延迟为: $2+1+2+1+(3+(4-1)\times 2)=15$.

那么延迟减少了 5。

接下来讲一下 skew，既然知道了 fanout 对于 delay 的影响，下面看一个例子：



由于时钟到每个触发器的互连线长短不一样，造成信号到达 clock pin 的时间也不一样，触发器也不会同时翻转。Skew 的定义就是最长路径减去最短路径的值。

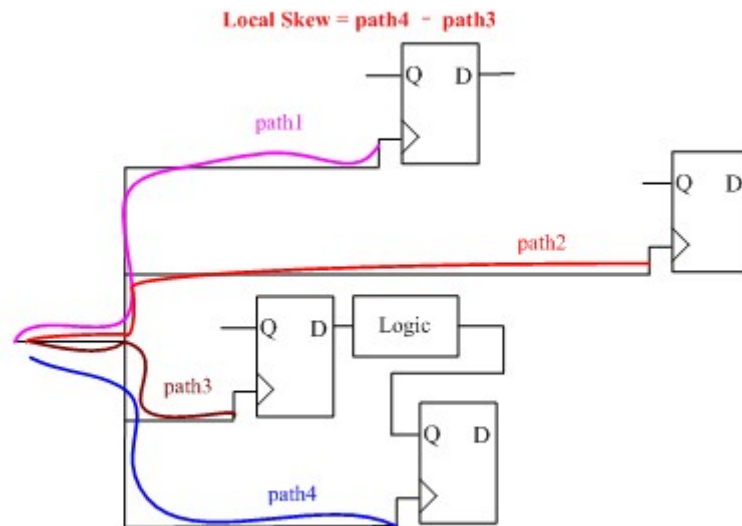
根据时钟域以及路径关系，skew 可以分为 global skew, local skew, interclock skew。

Global skew 是指，同一时钟域，任意路径的最大 skew。

Local skew 是指，同一时钟域，任意 2 个有逻辑关联关系的路

径最大 skew。

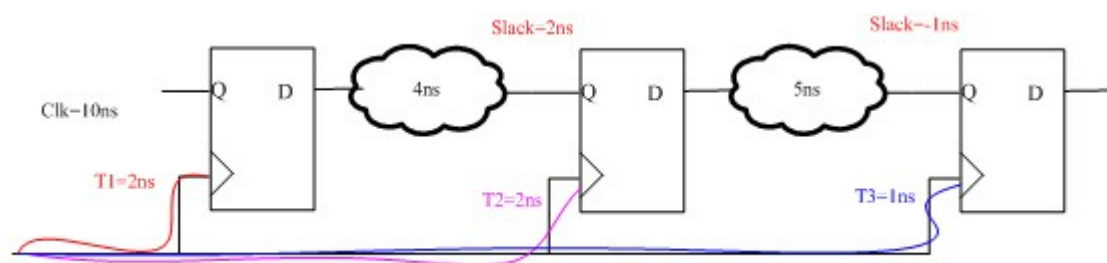
interClock skew 是指，不同时钟域之间路径的最大 skew



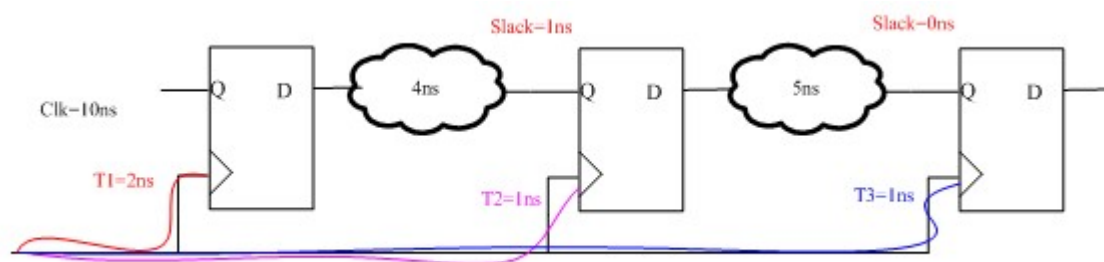
|

另外还有一个 useful skew。本来打算在 setup time 和 hold time 中讲解。这里先大概说下

如下图：时钟周期为 10ns，各时钟路径延迟如下：可以看到有一条路径的 slack 为 -1，说明这条路径违规。可以看到与这条路径相关的 skew 是 $T3 - T2 = -1\text{ns}$ 。



下面我们利用 useful skew 向前面一个 slack 比较充裕的路径 (slack=2ns) 借点 time, 来修正现在这条路径。如下图:

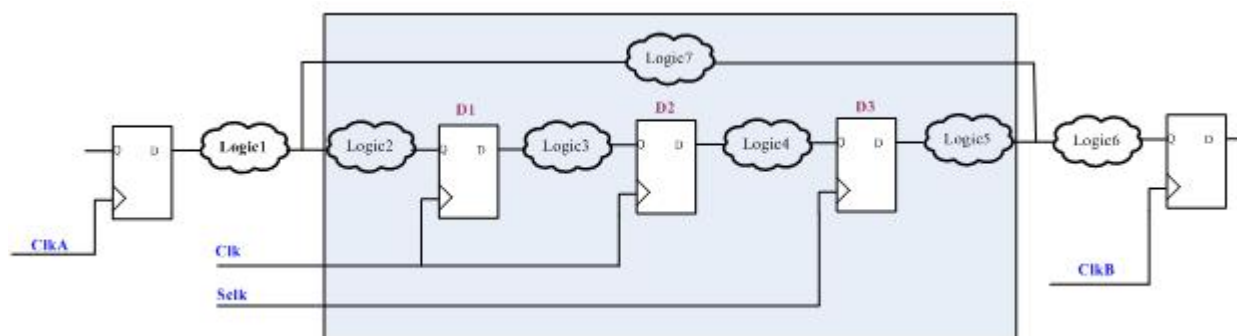


经过 useful skew, 修正了原来的 violator。

这就是 useful skew 的作用, 可以向前, 或者向后接 time 来修正 violator

DC 概论三之 setup time 与 hold time 之二

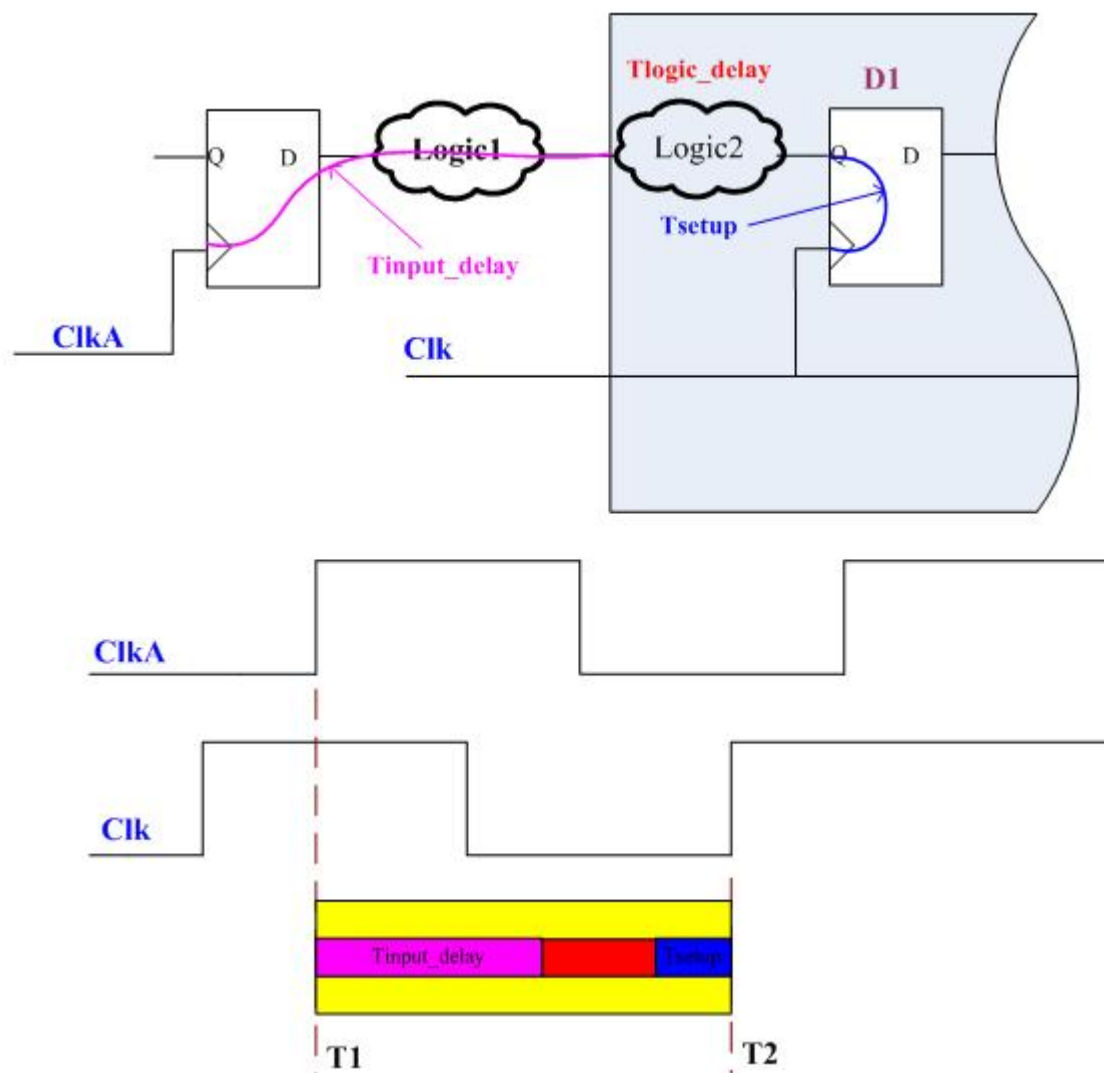
前面一篇讲了基本的建立时间和保持时间以及时序路径划分。在这篇开始之前, 先介绍一下很经典而且会时常用到的用来讲解的一个电路图, 如下。这篇文章的讲解也会给予这个电路图, 讲解的时候我把电路图分割成需要的部分:)



既然我们知道了建立时间和保持时间的含义，这篇主要是根据工厂提供的标准单元库中时序器件的建立时间和保持时间来预估我们的约束对设计的影响，是否满足时序要求，简单点就是说，时序有没有 violator。

set_input_delay :

input_delay 是设置外部信号到达输入端口的时间，dc 会用它来计算留内内部逻辑的空余时间是多少。



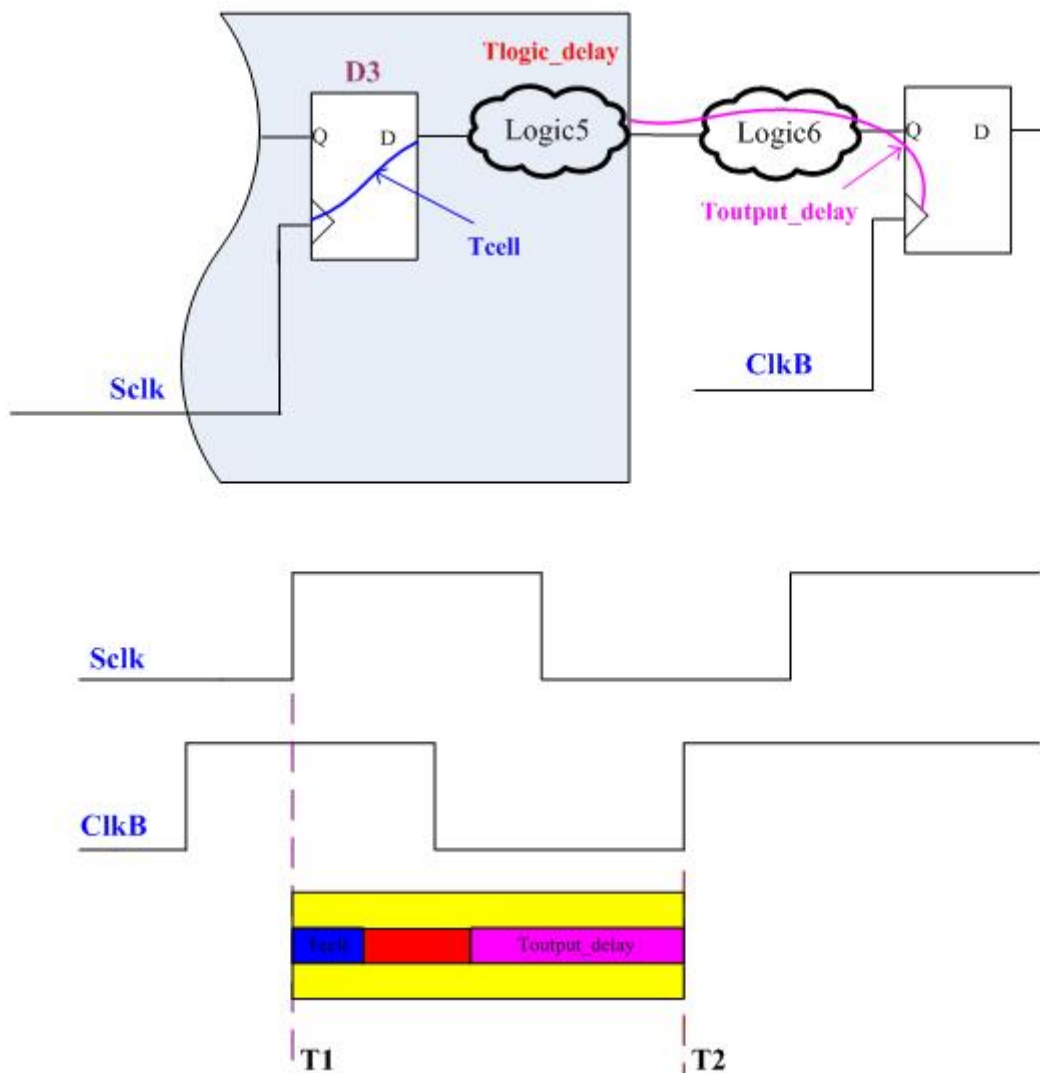
之所以让 ClkA 和 Clk 的时钟周期不一致，主要是用来说明上图中黄色部分的。黄色部分的确认是很重要的。这是 DC 用来确定时间

余量 (slack) 的关键。如上图所示，黄色部分已经确实是最小相位差。那么根据 input_delay 时间以及库中触发器的 setup 建立时间，可以知道留给内部逻辑的延迟时间是红色部分

$T_{\min} - T_{\text{input_delay}} - T_{\text{setup}}$ 。综合过程中，dc 会优化 Logic2 的时序，以使他达到时序要求。

同样 set_output_delay 是设置输出端口到数据采集处的延迟。如下图：

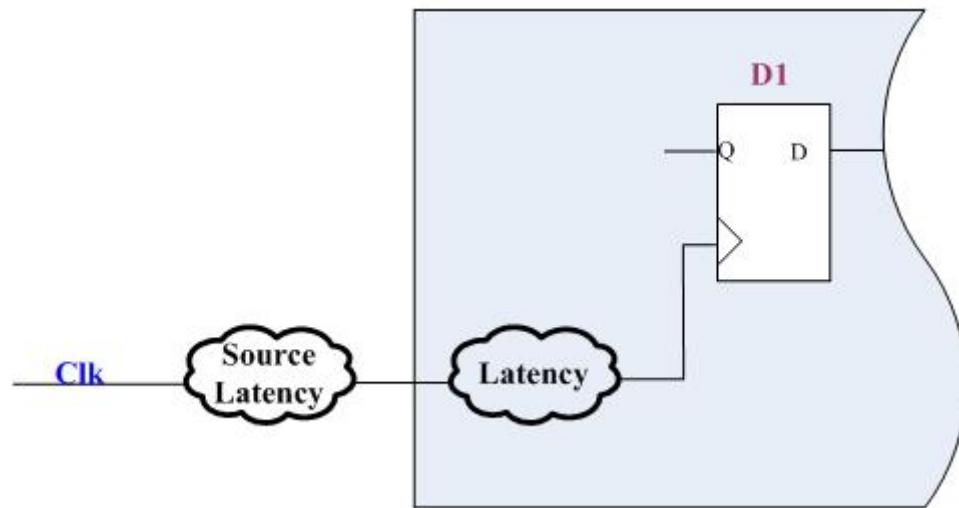
DC 会根他来计算留给内部逻辑的时间。



这里还是要强调一下不同频率的问题：)

介绍完了 input_delay 和 output_delay，以及 skew 在分析时间余量之前再介绍下时钟的 latency。

Latency 分为 source latency 和一般的 latency。

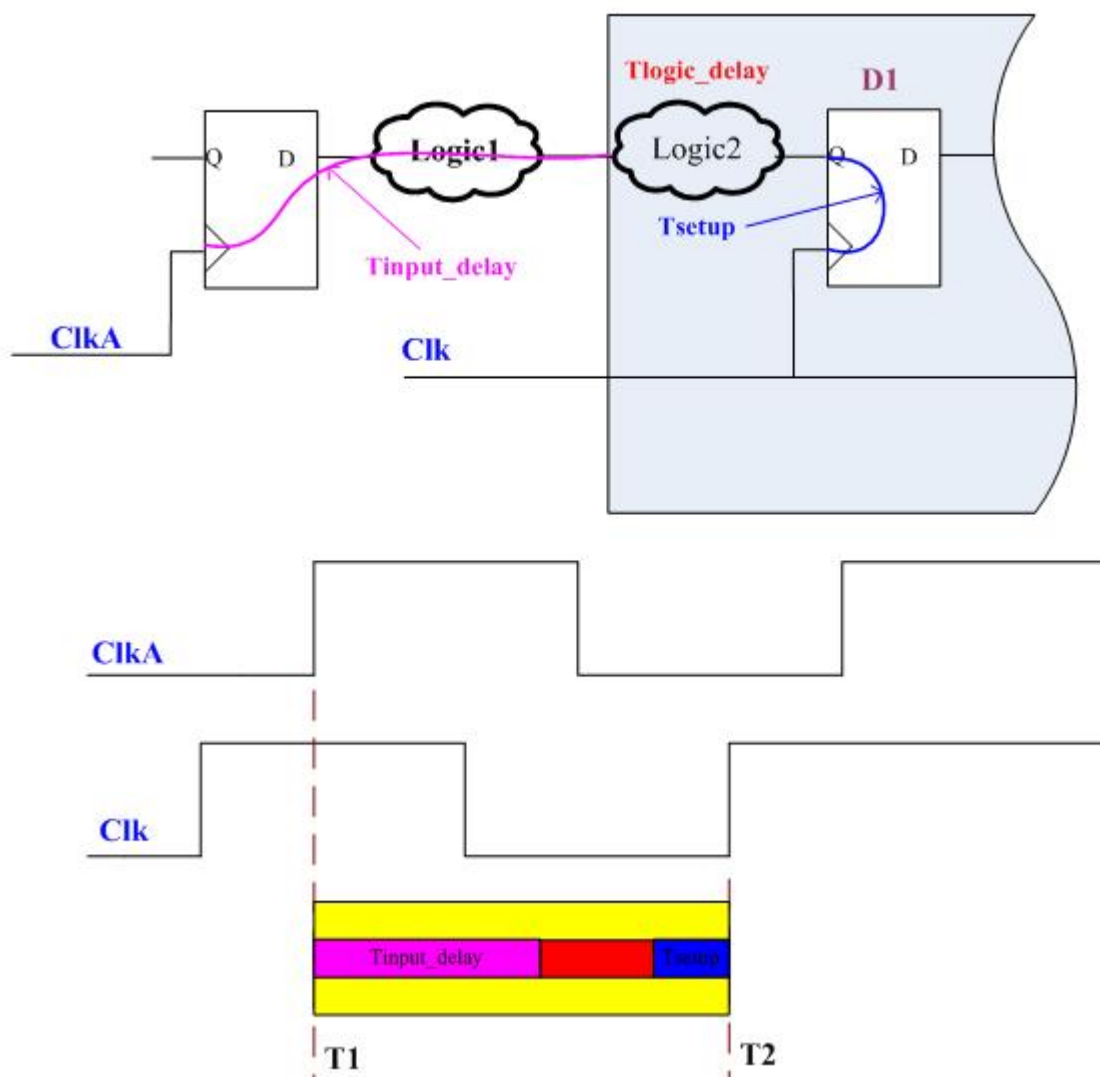


Source latency 指的是时钟源到时钟端口的延迟。

Latency 指的是时钟端口到内部时序器件的时钟管脚的延迟。

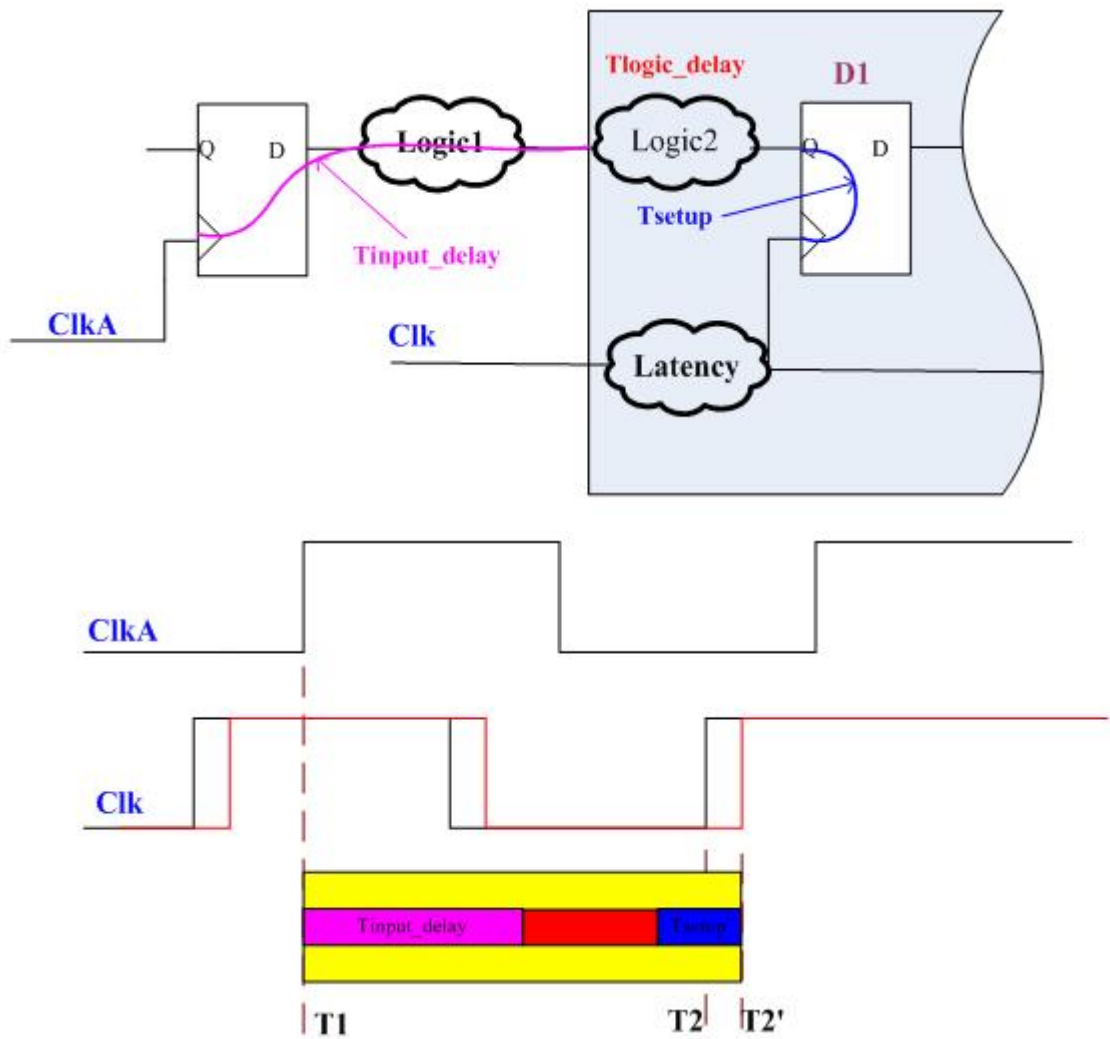
一般只考虑 latency 即可。

再看下 latency 对于内部逻辑的影响. 下图是不考虑 latency 的情况：



内部逻辑延迟的限度为 $T1 - T2 - T_{input_delay} - T_{setup}$ 。

当考虑了 latency 的时候。D1 上的 clock 会变成时序图中的红色部分

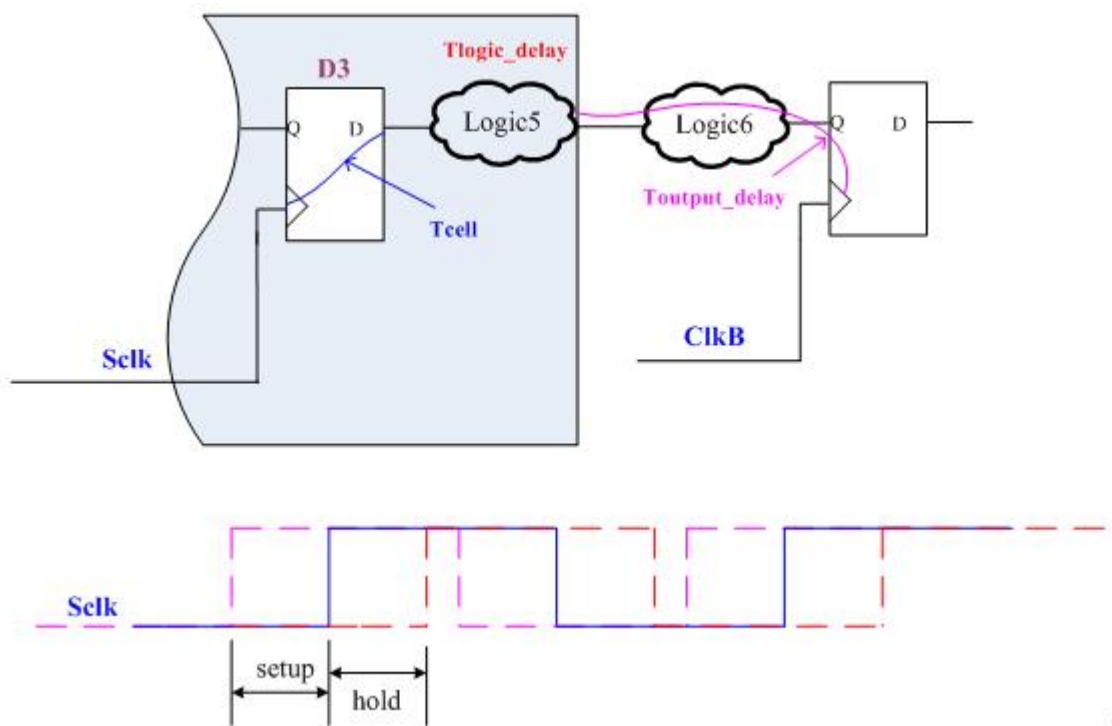


那么留给内部逻辑的最大延迟为：T2' (T2+Tlatency)

-T1-Tinput_delay-Tsetup。会发现留给内部逻辑的延迟会变大，那么会给 DC 更大的空间来综合。

同样如果 ClakA 上面如果也设置了 Latency 在分析 slack 的时候也要算进去。

顺便介绍一下时钟的 jitter，如下图



正常的时钟到来时蓝色的时序图。但是由于无法预知的因素，时钟可能提前（粉色）或者延迟（红色）到来，这就是 jitter。提前到来充为 setup jitter，延迟到来充为 hold jitter。

DC 中把 skew 和 jitter 合成一个 uncertainty。用 set_clock_uncertainty 来设置。

下文中提到的 skew 指的是合体，即 uncertainty。

Uncertainty 分为 setup 和 hold，顾名思义，如上图理解即可。

DC 概论四之 setup time 与 hold time 之三

下面开始正式分析时间余量 slack。其实有了上面的知识，只要稍微说明下大家都会很明白。

在介绍 slack 之前，我们要先了解一下要求时间（required time）和达到时间（arrive time）的概念以及计算方法。

如果没有特殊说明，黑色 clock 代表没有影响因素的理想时钟，红色（粉色）clock 代表收到 latency 影响的时钟。蓝色 clock 代表同时受到 latency 和 skew（uncertainty）影响的时钟。

对于建立时间（setup time）的到达时间和要求时间。

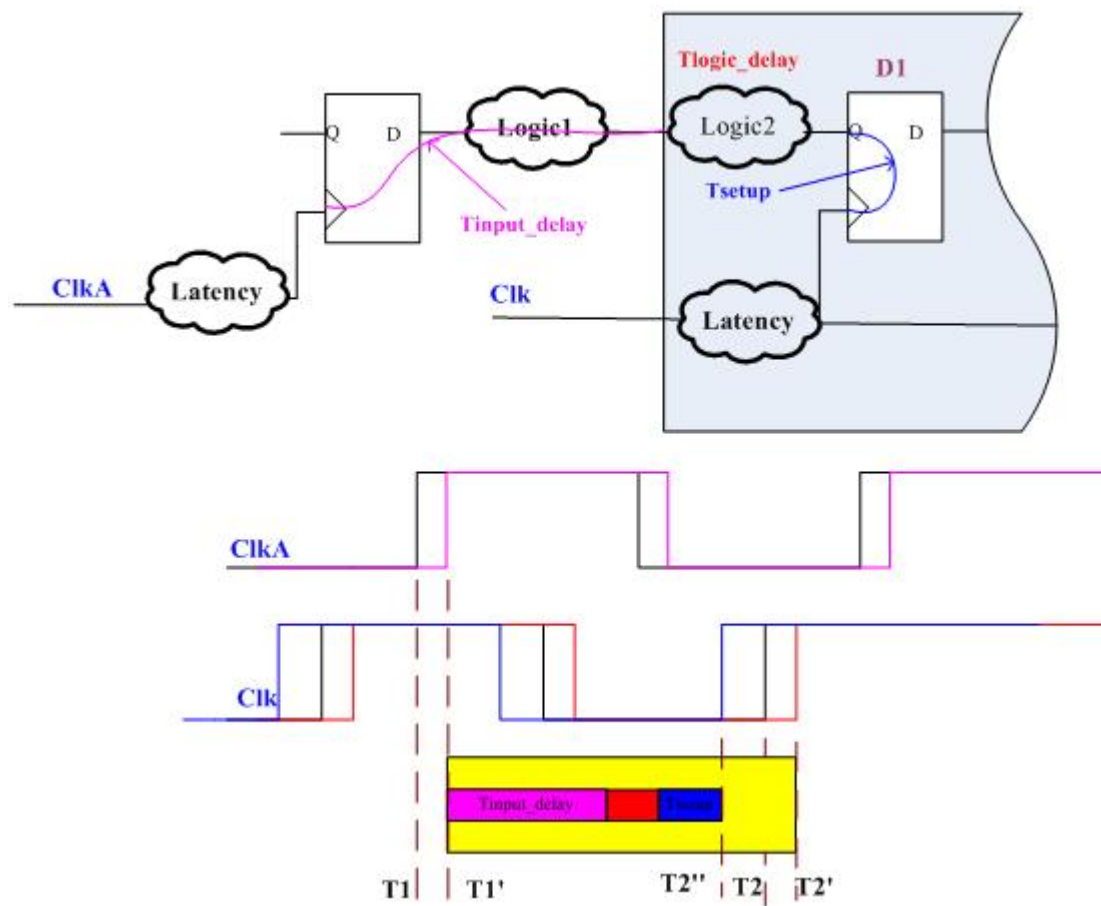
在分析前，记住，建立时间是分析路径中的考虑到各种最不利因素的最大延迟，打个比方：



前面一级用尽最大程度向后推（最大的延迟），本级就近打力气向前顶（最大不确定因素）。

然后看中间有没有漏气（slack 为负，时序违规）。

<!--[if !supportLists]-->1, <!--[endif]-->输入端口到时序器件的数据端口。



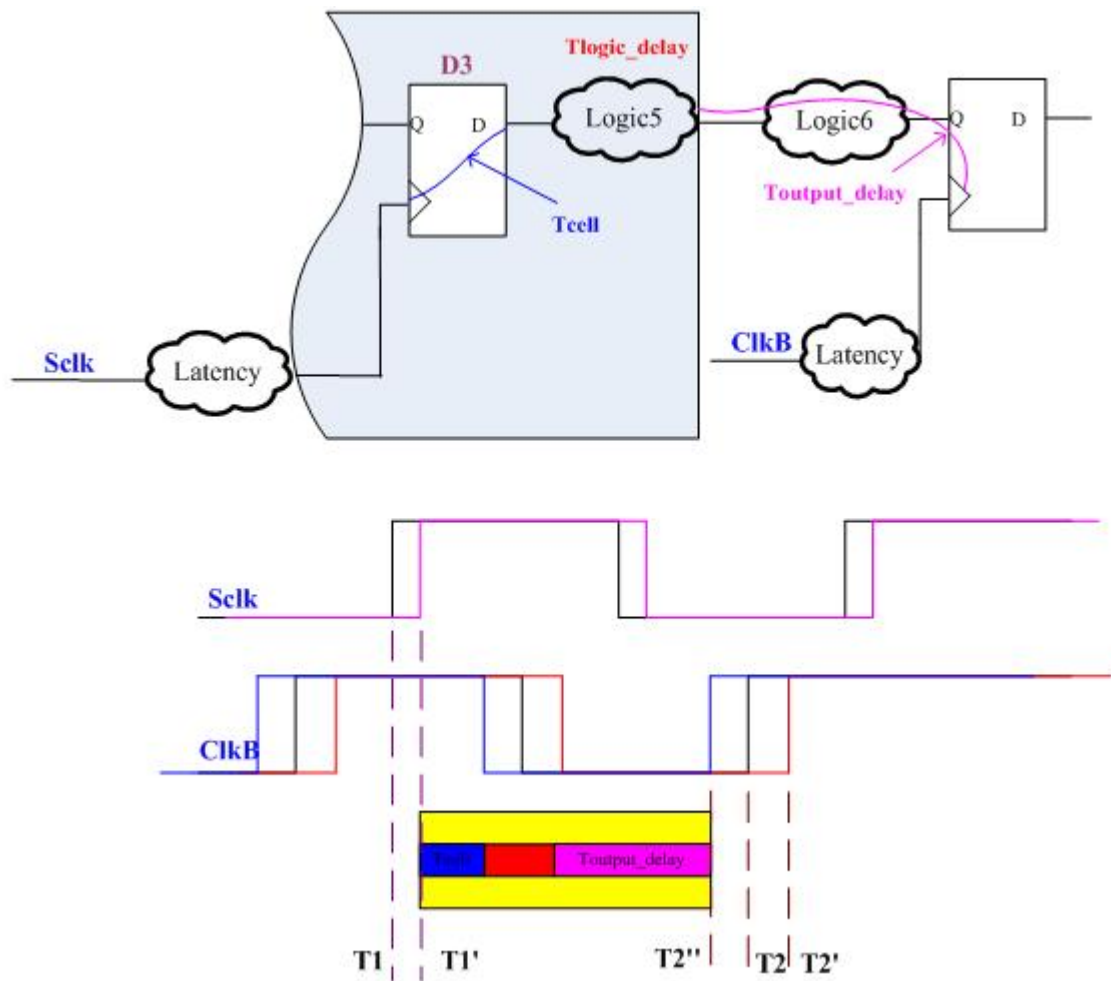
上图中

要求时间= $T_2 + T_{latency} - T_{uncertainty_setup} - T_{setup}$

到达时间= $T_1 + T_{latency} + T_{input_delay} + T_{logic2}$

<!--[if !supportLists]-->2, <!--[endif]-->时

序器件的输出管脚到输出端口

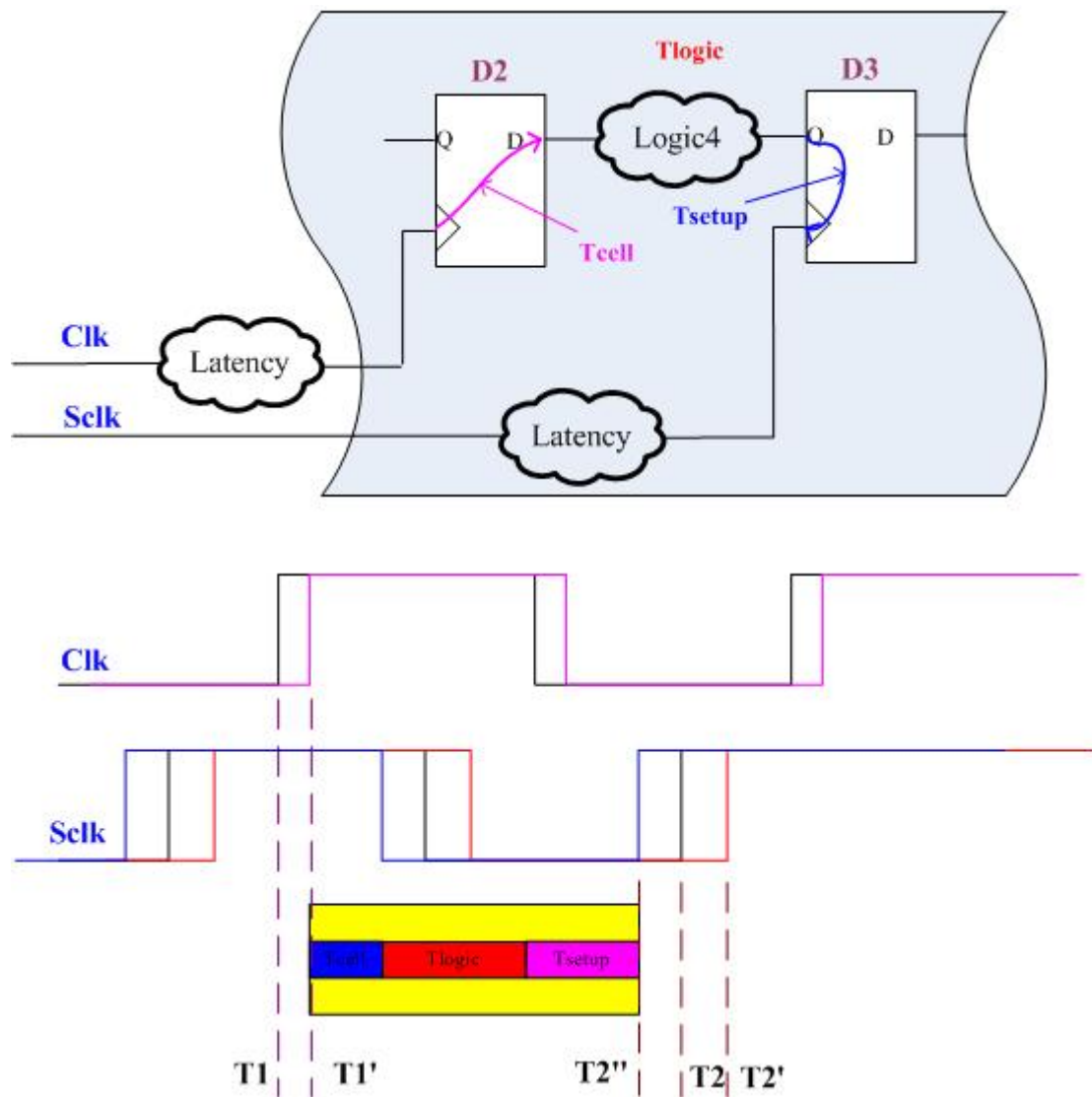


上图中：

要求时间= $T_2 + T_{latency} - T_{output_delay} - T_{uncertainty_setup}$

到达时间= $T_1 + T_{latency} + T_{cell} + T_{logic5}$

3. 时序器件到时序器件

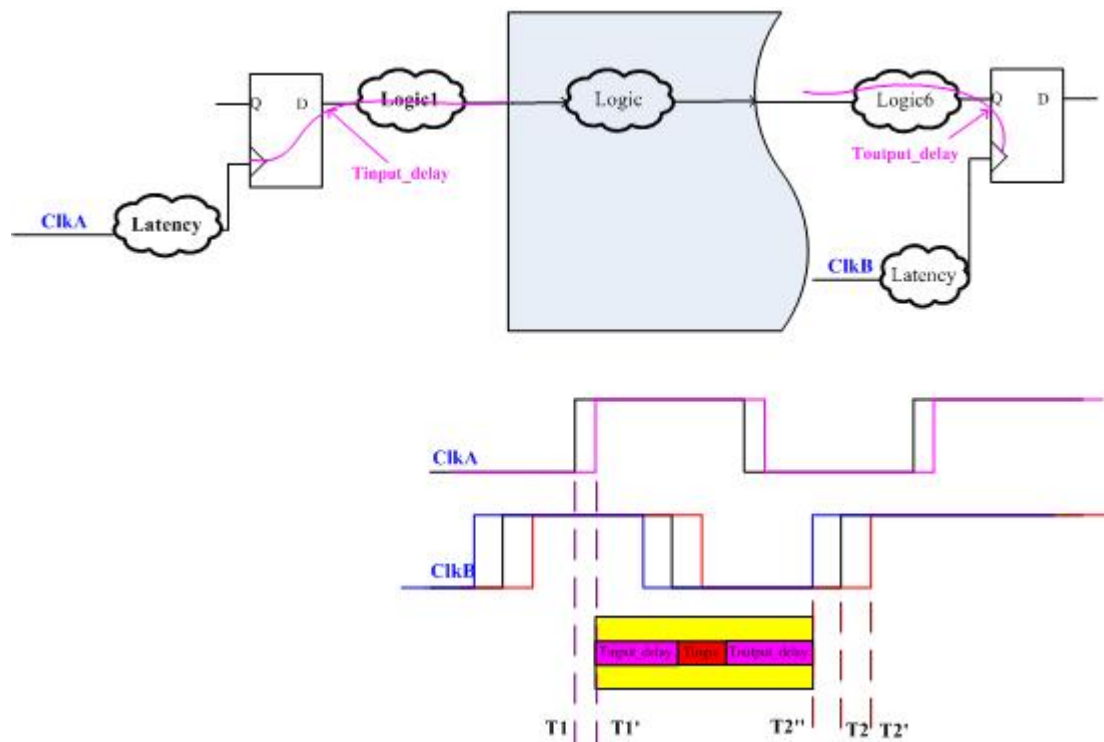


上图：

要求时间= $T_2 + T_{latency} - T_{uncertainty_setup} - T_{setup}$

到达时间= $T_1 + T_{latency} + T_{cell} + T_{logic}$

4, 输入端口到输出端口



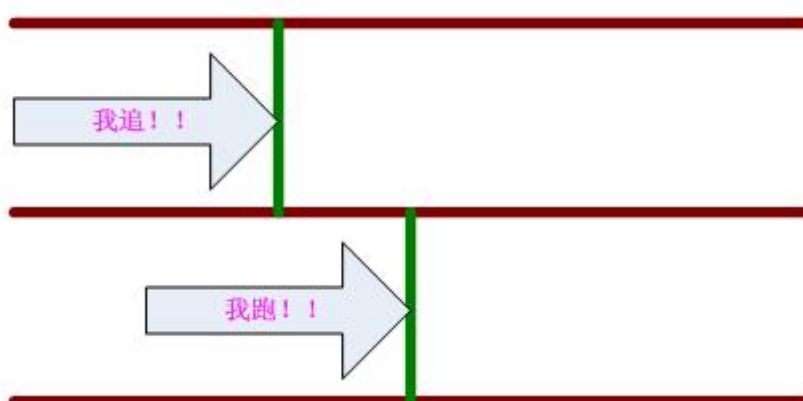
如上图：

要求时间= $T_2 + T_{\text{latency}} - T_{\text{uncertainty_setup}} - T_{\text{output_delay}}$

到达时间= $T_1 + T_{\text{latency}} + T_{\text{input_delay}} + T_{\text{logic}}$

我们再来看下保持时间，保持时间的到达时间和建立时间的到达时间是一样的。只是保持时间的要求时间不一样而已。

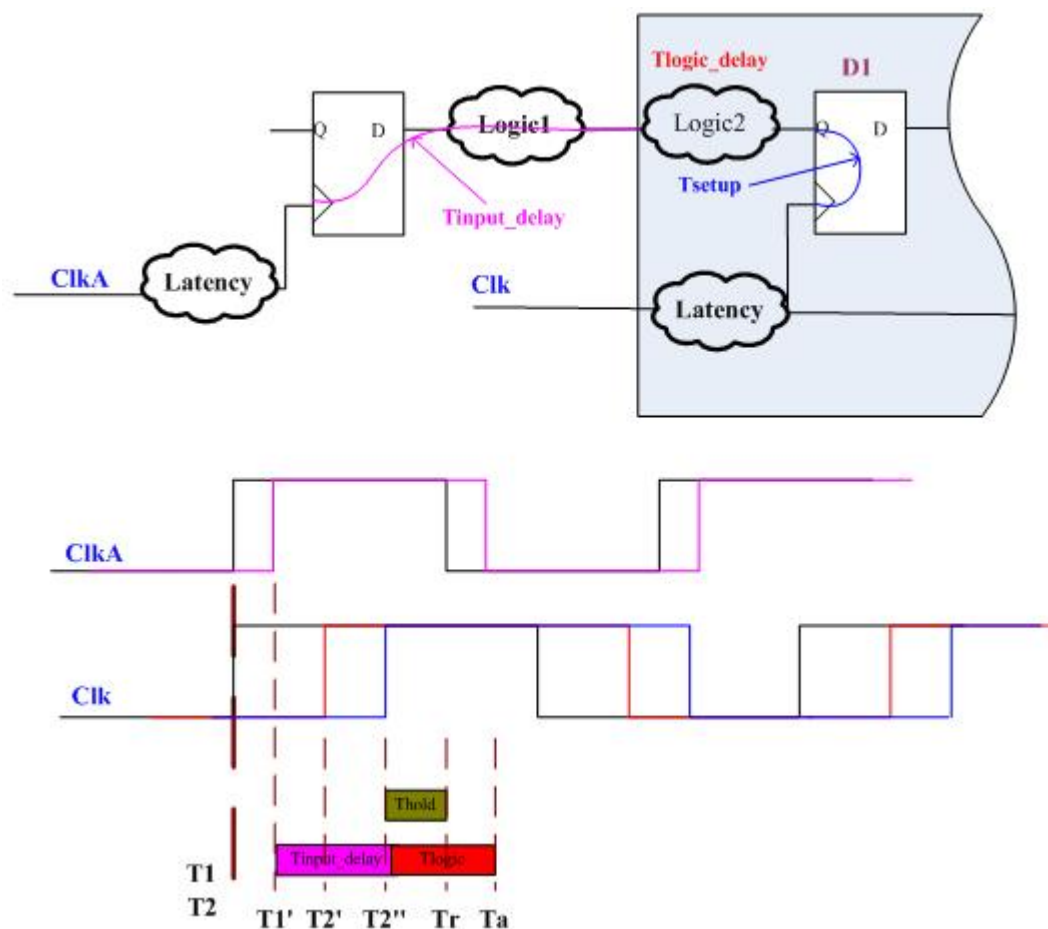
保持时间类似于龟兔赛跑：



系统要求时间就是尽可能的向后，而线上信号就是追那个要确定的时间，追上了，就没问题，追不上问题就有了。

说明：红色和粉色表示受 latency 影响，蓝色表示受 uncertainty 影响。

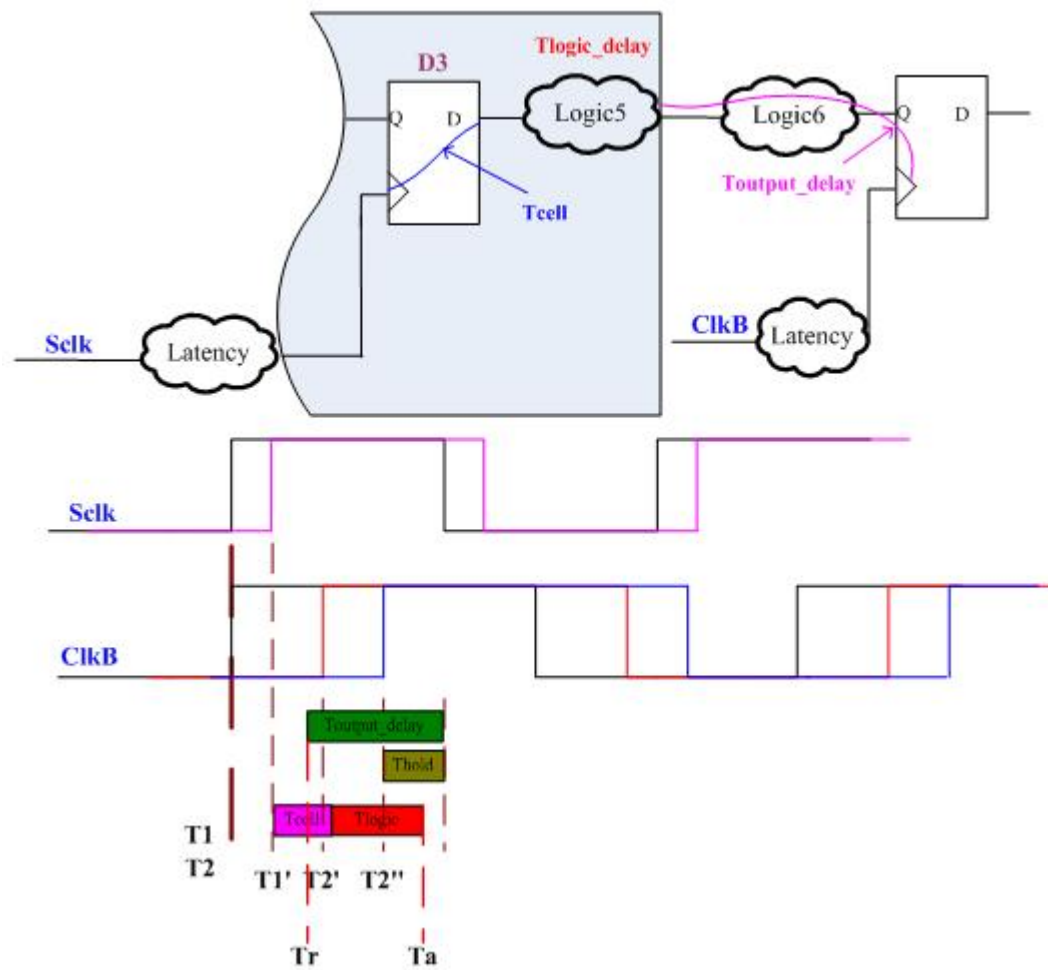
1. 输入端口到时序器件的数据端口。



到达时间: $T_{arrive} = T1 + T_{latency} + T_{input_delay} + T_{logic}$

要求时间: $T_{require} = T2 + T_{latency} + T_{uncertainty_hold} + T_{hold}$

2. 时序器件的输出管脚到输出端口

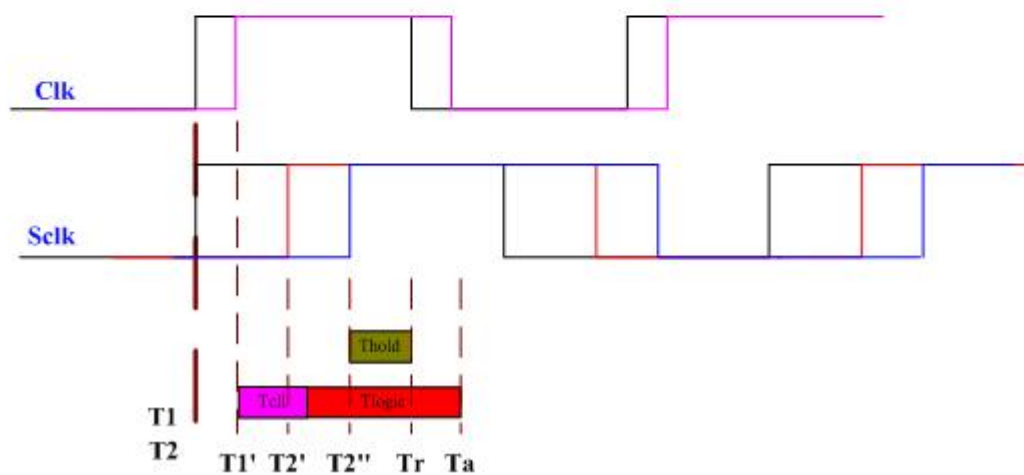
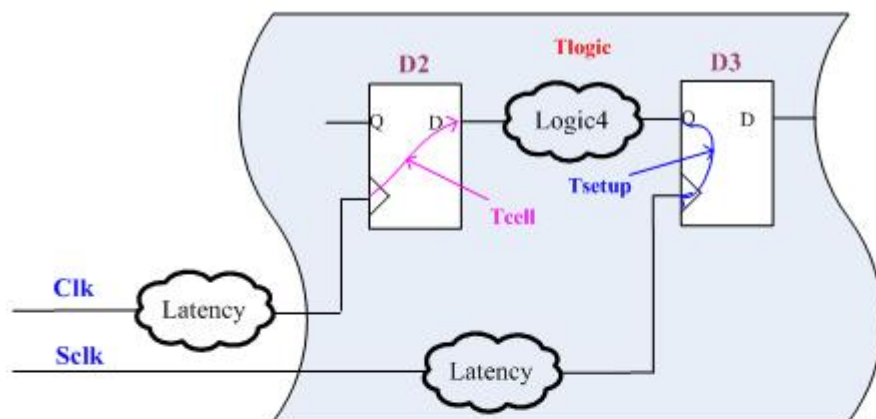


到达时间： $T_a = T_1 + T_{latency} + T_{cell} + T_{logic}$

要求时间：

$T_r = T_2 + T_{latency} + T_{uncertainty_hold} + T_{hold} - T_{output_delay}$

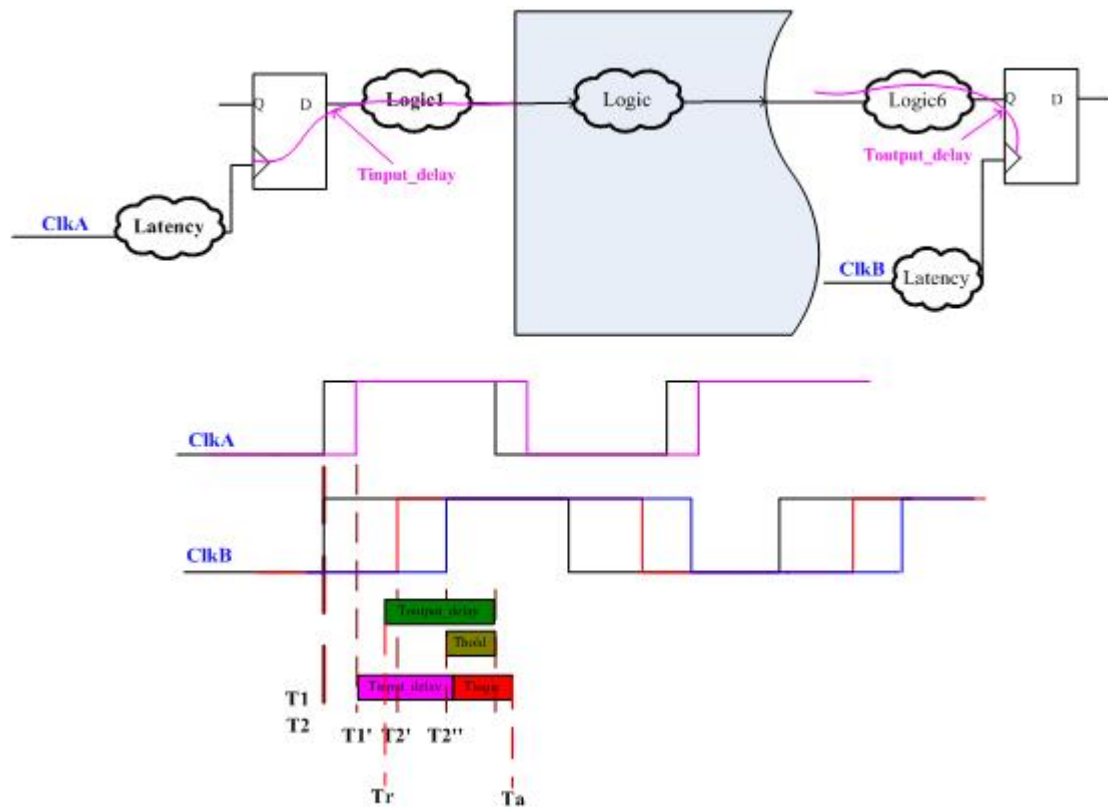
3 时序器件到时序器件



到达时间: $T_a = T_1 + T_{latency} + T_{cell} + T_{logic}$

要求时间: $T_r = T_2 + T_{latency} + T_{uncertainty_hold} + T_{hold}$

4, 输入端口到输出端口



到达时间： $T_a = T_1 + T_{latency} + T_{input_delay} + T_{logic}$

要求时间：

$T_r = T_2 + T_{latency} + T_{uncertainty_hold} + T_{hold} - T_{output_delay}$

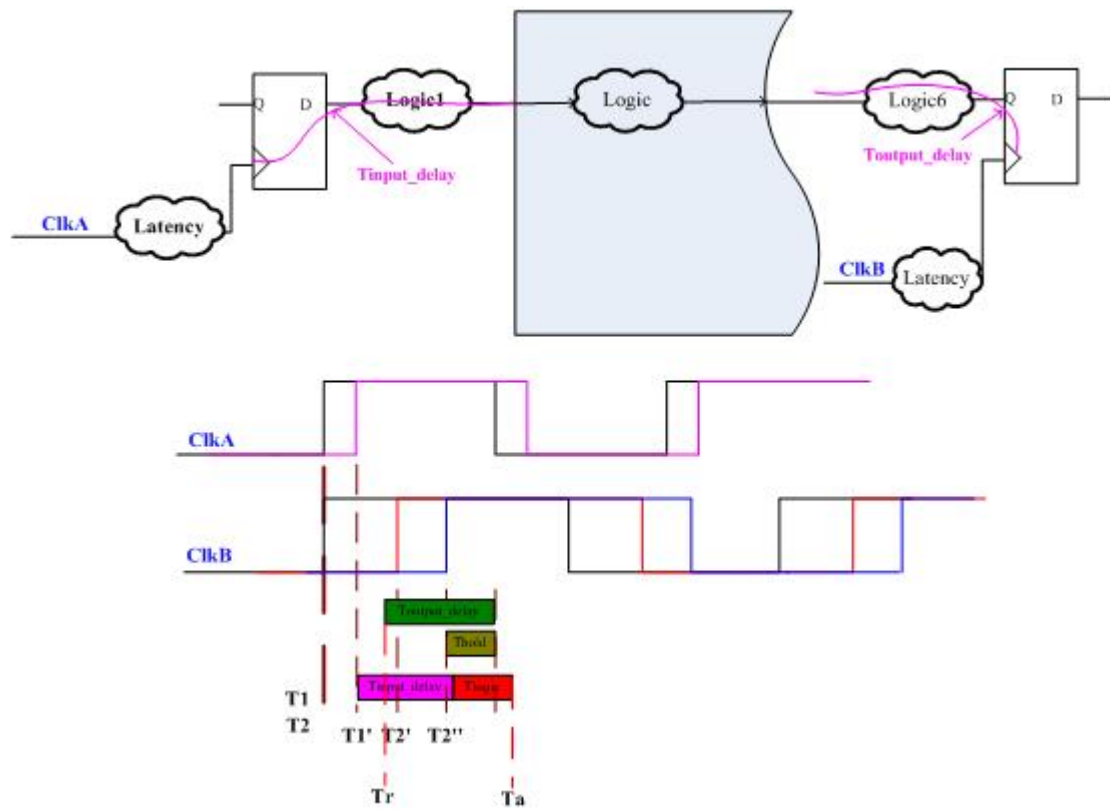
Slack 计算：

对于建立时间：

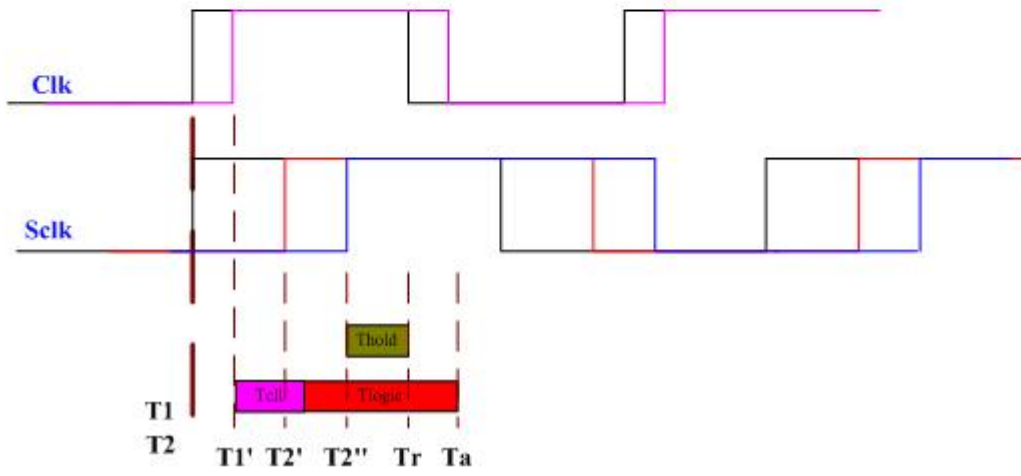
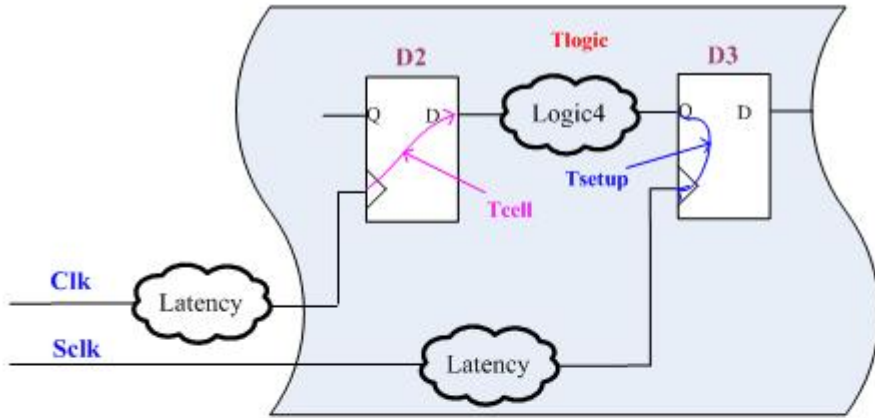
$Slack = \text{要求时间} - \text{到达时间}$

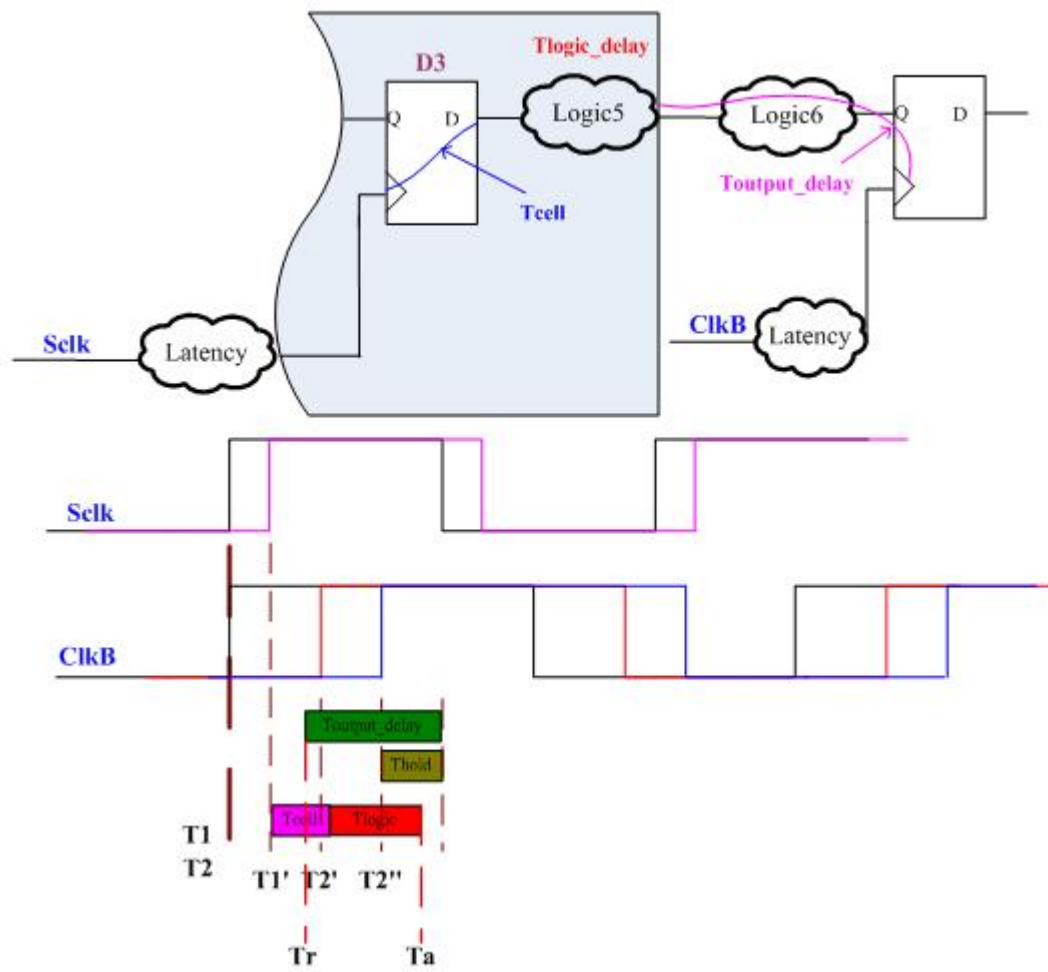
对于保持时间

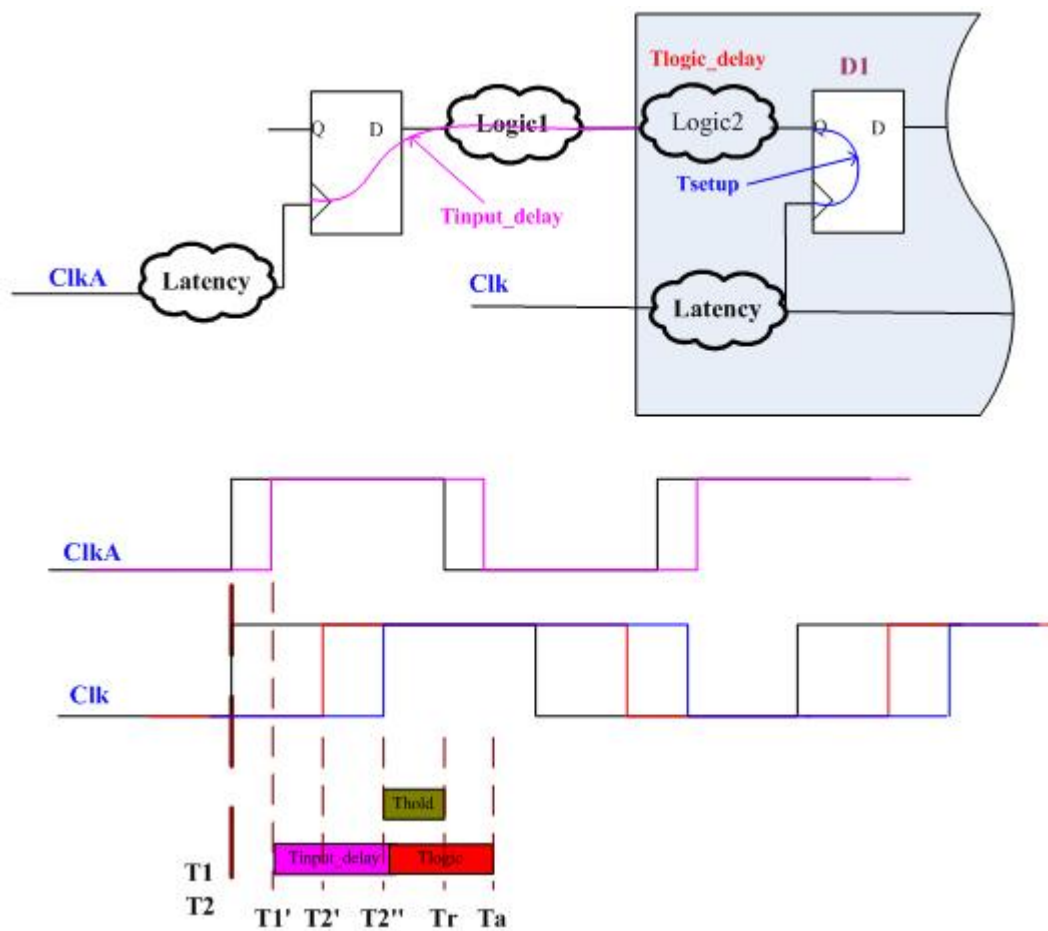
$Slack = \text{到达时间} - \text{要求时间}$



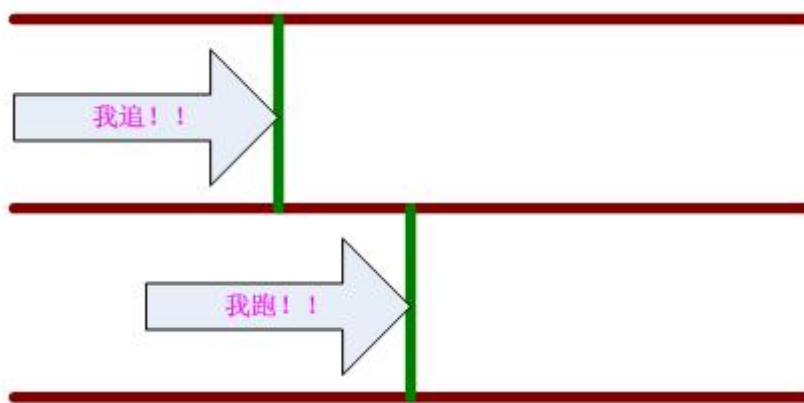
a



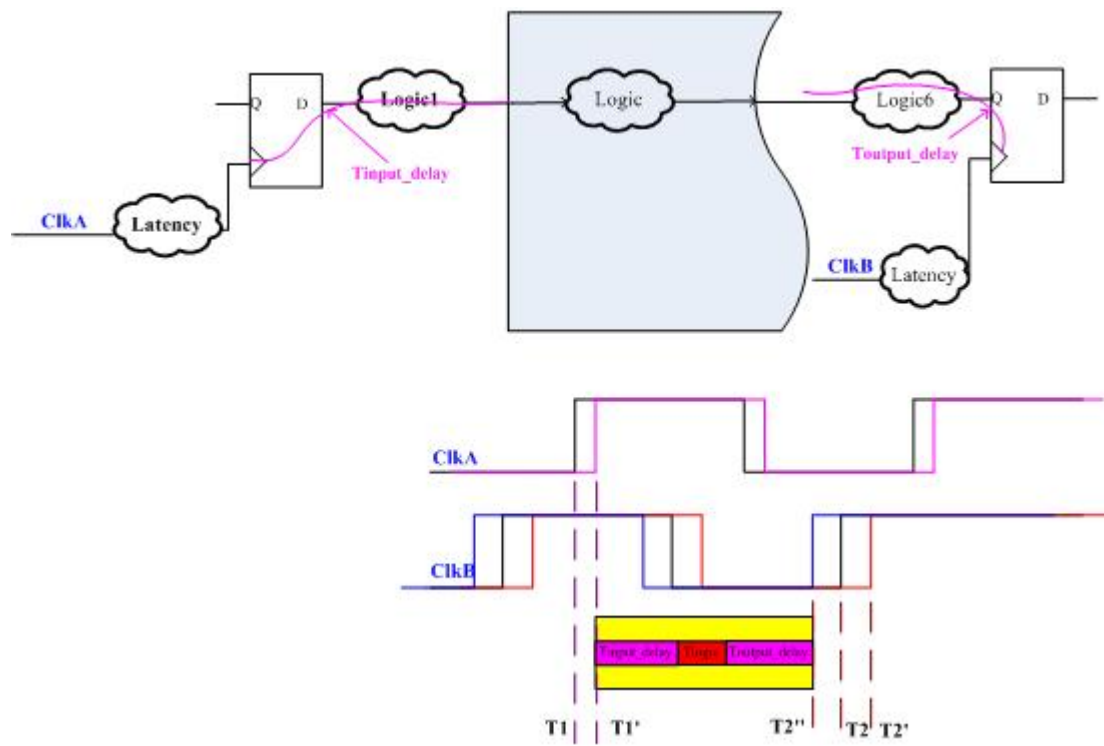


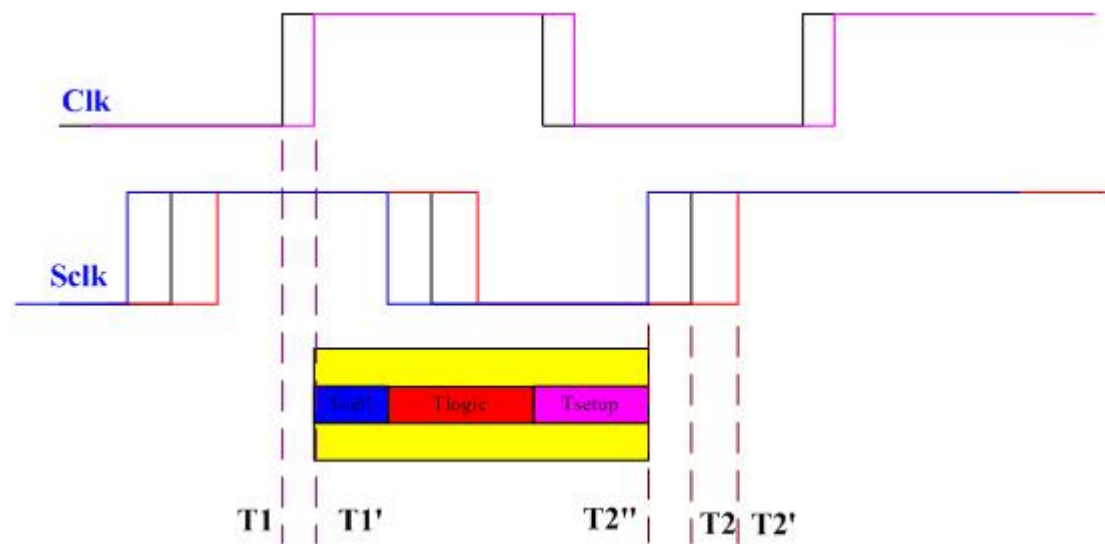
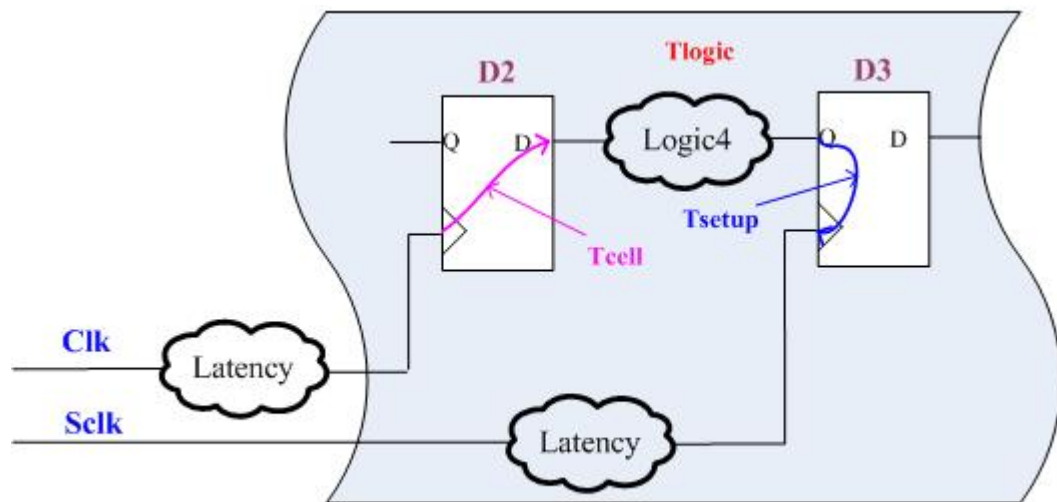


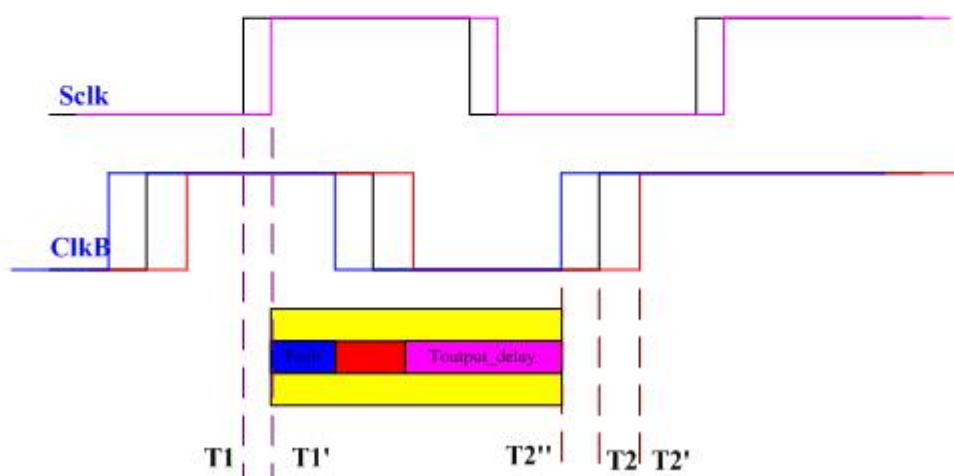
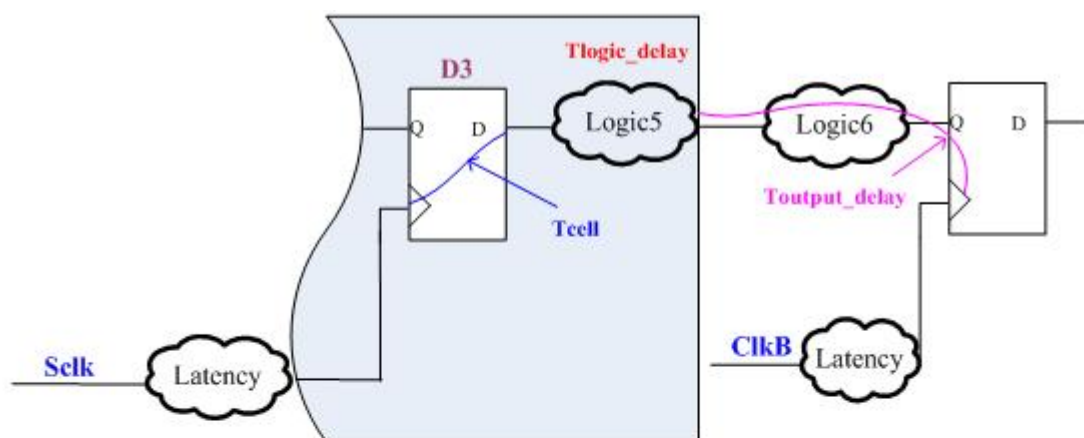
7

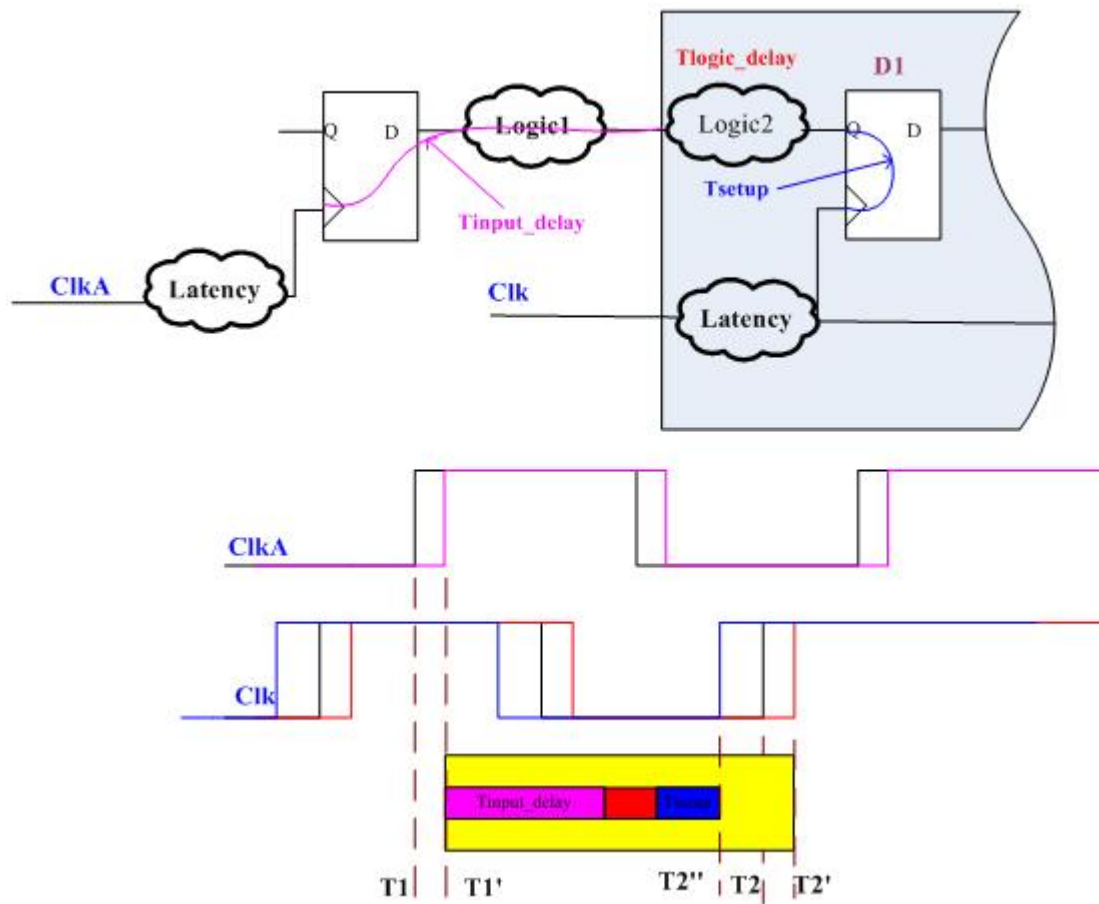


6









2



DC 概论五之 high fanout

dc 在综合高扇出的网络的时候，约束优先级是：

- 1.功能正确
- 2.DRC (max_transition ,max_fanout,max_capacitance)
- 3.Setup time (max_delay)

4. Hold time (min_delay)

5. Other...

为了符合 drc 要求，dc 通常花费很多时间来编译和修正这些 DRC violators。

路径上的 cell 延迟由 input_transition 和 output_load(包括扇出 pin 上的 load) 决定，这个由查抄表可以得到。

而 net 延迟是由 net 上的 R, C 决定的。在没有布局布线之前，我们不知道实际的 R, C 是多少，dc 根据互联线模型(set_wire_load_model) 来计算出 R, C。然后根据得到的 R, C 计算出 net 上的延迟：

$$\text{Net_delay} = R * C * OC$$

其中系数 OC 是根据操作环境 (set_operating_conditions) 中设置的 rc 树模型得到。

一般的工艺库的操作环境有三种，WORST, TYPICAL, BEST, 分别是最差，典型，最好。

在 ic 中出现 high fanout 的情况基本有三种：

1. 时钟 clock

2. 复位 reset

3. 一般信号

```
<!--[if !supportLineBreakNewLine]-->
```

```
<!--[endif]-->
```

dc 中对于高扇出 net 的处理，基本就是加 buffer（前提，如果没有对这条 net 设置一些约束，比如 set_ideal_network,set_dont_touch，后面会讲到），以此来减少 cell 输出端的负载，从而减少 transition time 和 delay time，以及 max_capacitance。而事实上我们是不希望 dc 这么处理的，我们希望的是可以在后端版图的时候让后端工具自己加 buffer，因为我们不知道真实的 high fanout net 上的 RC，所以不知道应该加怎么样的 buffer，dc 只是根据互联线模型来计算 RC，接着加入 buffer，不是真实的，只有布线以后我们才可以得到几乎真实的 rc。

所以在 dc 综合过程中我们要阻止 dc 最 high fanout net 进行 insert buffer 处理。因此这些没被处理的高扇出 net 就会引起一些 drc 或者 timing 错误，在 dc 中，dc 用价值函数（cost function）来判断这些约束对设计的影响。价值函数=DRC violator 和+ timing violator 和。一般的，dc 会根据所有 drc 和 timing 错误，通过使价值函数趋近等于 0 来修正这些违规。为了达到效果，dc 会每次修正一个路径，然后重新计算价值函数，如果价值函数变小，说明设计被改进了。

在介绍如何处理 high fanout net 之前，先介绍 3 个命令。

Set_ideal_net （已经被 set_ideal_network -no_propagate 代替） 忽略 port, pin, net 上的时序优化(timing optimization)，以及 drc 修正(drc

fixxing)。network 具有传输型。

Set_dont_touch （已经被 set_dont_touch_network -no_propagate 代替）

忽略，port，cell，design，pin 上的优化(timing optimization)，但是不会忽略 DRC。network 具有传输型

这样我们在综合的时候就要对 high fanout net 做一定的约束，让 dc 不对这些 net 做优化以及加入 buffer。下面分三种情况来说明。

1.Clock，对于 clock，当我们用 create_clock,or creat_generated_clock 创建 clock 的时候，这些 clock 已经有了 ideal_network 的属性。Dc 已经不会在 clock tree 上加入 buffer，同时也不会计算 drc violation，但是 delay timing 仍然会被计算。不计算 drc 不是说没有负载。

2.Reset，对于复位高扇出信号，因为没有那些属性，所以要手动设置,set_ideal_network

3.一般信号。同样需要手动 set_ideal_network

下面看例子：

下面看一个高扇出实例，有时钟，有复位还有一般信号

```
module test(clk,clk_G, d_in ,s_r1, s_r2, rst_N1, rst_N2,dout);
```

```
parameter size =1100;
```



```
input d_in, rst_N1,rst_N2, s_r1, s_r2,clk_G,clk;
```

```
output dout;
```

```
reg dout;
```

```
reg [size-1:1] tmp;
```

```
wire G_clk, rst_N, s_r;
```

```
integer i;
```

```
assign G_clk = clk & clk_G;
```

```
assign rst_N = rst_N1 & rst_N2;
```

```
assign s_r = s_r1 & s_r2;
```

```
always@(posedge G_clk or negedge rst_N) begin
```

```
    if(!rst_N) begin
```

```
        dout <= 0;
```

```
        tmp  <= 0;
```

```
    end
```

```
    else begin
```

```
        dout <= tmp[size-1] | s_r;
```

```
        for(i=size-1 ; i>1; i=i-1)
```

```
        tmp[i] <= tmp[i-1] | s_r;
```

```
    tmp[1] <= d_in | s_r;
```

```
end
```

```
end
```

```
endmodule
```

综合脚本:

```
set lib $env(DC_LIB)
```

```
set target_library "slow.db fast.db"
```

```
set link_library "* $target_library"
```

```
set search_path ". ../src ../scripts $lib"
```

```
set hdlin_while_loop_iterations 5000
```

```
analyze -format verilog test.v
```

```
elaborate test
```

```
uniquify
```

```
link
```

```
check_design
```

```
create_clock -period 100 [get_ports clk]
```

```
set_operating_conditions -max slow -min fast
```

```
set_wire_load_mode top
```

```
set_min_library slow.db -min_version fast.db
```

```
set input_exp_clk [remove_from_collection [all_inputs] [get_ports clk]]
```


3. 较大的输出转换时间 output_transition，尤其是 U1340 的 output_transition 作为下一级的 input_transition，经过下一级的 cell 时候会造成更大的延迟。

时序分析

<!--[if !supportLists]-->1. <!--[endif]-->clock

```
*****
Report : timing
        -path full
        -delay max
        -nets
        -max_paths 1
Design : test
Version: Z-2007.03-SP1
Date   : Mon May 12 11:47:09 2008
*****

# A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

Startpoint: clk (clock source 'clk')
Endpoint: dout_reg/CK (internal pin)
Path Group: (none)
Path Type: max

Point                Fanout    Incr    Path
-----
clk (in)              0.00     0.00  0.00 r
clk (net)              1       0.00  0.00 r
U1106/Y (AND2X4)      1592.53 # 1592.53 r
G_clk (net)           1100     0.00  1592.53 r
dout_reg/CK (DFFRH0X1) 0.00 # 1592.53 r
data arrival time                    1592.53
-----
(Path is unconstrained)
```

可以看到 clock tree 上没有插入 buffer，但是 cell 的延迟却很大

2.reset

Startpoint: rst_N1 (input port clocked by clk)
 Endpoint: tmp_reg[500]/RN (internal pin)
 Path Group: (none)
 Path Type: max

Point	Fanout	Incr	Path

input external delay		60.00	60.00 r
rst_N1 (in)		0.00	60.00 r
rst_N1 (net)	1	0.00	60.00 r
U2441/Y (AND2X2)		0.16	60.16 r
n3 (net)	3	0.00	60.16 r
U1314/Y (INVX1)		0.09	60.25 f
n229 (net)	4	0.00	60.25 f
U1337/Y (INVX1)		0.15	60.40 r
n126 (net)	4	0.00	60.40 r
U1325/Y (INVX1)		0.11	60.51 f
n218 (net)	4	0.00	60.51 f
U1334/Y (INVX1)		0.15	60.66 r
n129 (net)	4	0.00	60.66 r
U1308/Y (INVX1)		0.13	60.78 f
n235 (net)	5	0.00	60.78 f
U1335/Y (INVX1)		0.16	60.94 r
n128 (net)	4	0.00	60.94 r
U1318/Y (INVX1)		0.11	61.05 f
n225 (net)	4	0.00	61.05 f
U1242/Y (INVX1)		0.41	61.46 r
n176 (net)	13	0.00	61.46 r
tmp_reg[500]/RN (DFFRHQX1) <-		0.00	61.46 r
data arrival time			61.46

Dc 自动插入了 buffer。

3 一般信号

Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00 f
s_r1 (in)		0.00	60.00 f
s_r1 (net)	1	0.00	60.00 f
U1340/Y (AND2X2)		0.16	60.16 f
n2 (net)	3	0.00	60.16 f
U1338/Y (INVTX1)		0.14	60.30 r
n9 (net)	4	0.00	60.30 r
U1329/Y (INVTX1)		0.13	60.43 f
n6 (net)	5	0.00	60.43 f
U1286/Y (INVTX1)		0.16	60.58 r
n110 (net)	4	0.00	60.58 r
U1151/Y (INVTX1)		0.22	60.80 f
n64 (net)	12	0.00	60.80 f
U1949/Y (OR2X2)		0.24	61.04 f
N601 (net)	1	0.00	61.04 f
tmp_reg[500]/D (DFFRHQX1)		0.00	61.04 f
data arrival time			61.04

clock clk (rise edge)		100.00	100.00
clock network delay (ideal)		0.00	100.00
tmp_reg[500]/CK (DFFRHQX1)		0.00	100.00 r
library setup time		-0.36	99.64
data required time			99.64

data required time			99.64
data arrival time			-61.04

slack (MET)			38.60

Dc 同样自动插入了 buffer。

下面我们修改一下脚本，如下；

```
set lib $env(DC_LIB)
```

```
set target_library "slow.db fast.db"
```

```
set link_library "* $target_library"
```

```
set search_path ". ../src ../scripts $lib"
```

```
set hdlin_while_loop_iterations 5000
```

```
analyze -format verilog test.v
```

elaborate test

uniquify

link

check_design

create_clock -period 100 [get_ports clk]

set input_exp_clk [remove_from_collection [all_inputs] [get_ports clk]]

set_input_delay 60 -clock [get_clocks clk] \$input_exp_clk

set_output_delay 30 -clock [get_clocks clk] [all_outputs]

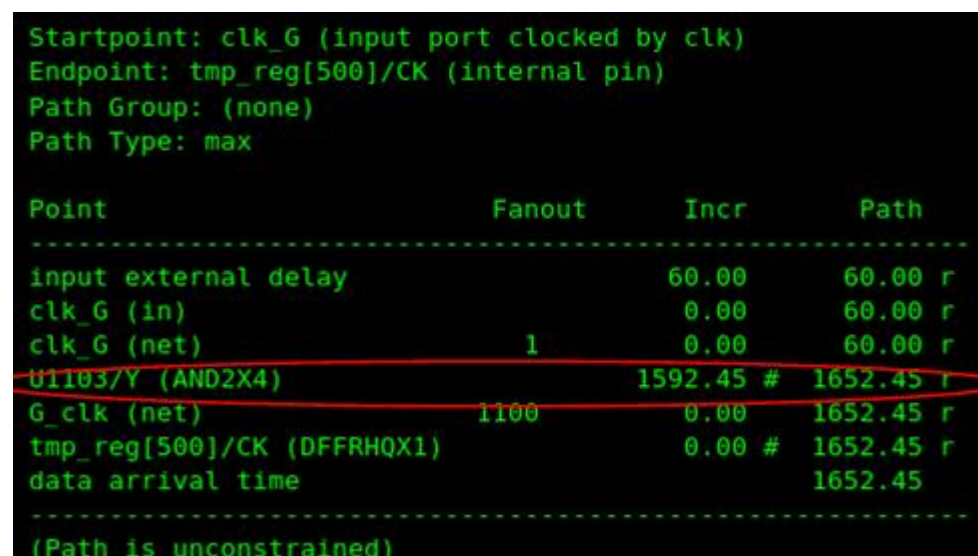
set_ideal_network -no_propagate [get_nets s_r]

set_ideal_network -no_propagate [get_nets rst_N]

compile

重新分析 timing

<!--[if !supportLists]-->1. <!--[endif]-->clock



```
Startpoint: clk_G (input port clocked by clk)
Endpoint: tmp_reg[500]/CK (internal pin)
Path Group: (none)
Path Type: max
```

Point	Fanout	Incr	Path

input external delay		60.00	60.00 r
clk_G (in)		0.00	60.00 r
clk_G (net)	1	0.00	60.00 r
U1103/Y (AND2X4)		1592.45 #	1652.45 r
G_clk (net)	1100	0.00	1652.45 r
tmp_reg[500]/CK (DFFRHQX1)		0.00 #	1652.45 r
data arrival time			1652.45

(Path is unconstrained)			

基本和原来一样，cell 上仍然有很大延迟。

<!--[if !supportLists]-->1. <!--[endif]-->reset


```

Startpoint: rst_N2 (input port clocked by clk)
Endpoint: tmp_reg[500]/RN (internal pin)
Path Group: (none)
Path Type: max

```

Point	Fanout	Incr	Path

input external delay		60.00	60.00 f
rst_N2 (in)		0.00	60.00 f
rst_N2 (net)	1	0.00	60.00 f
C5514/Y (AND2X4)		0.15 #	60.15 f
rst_N (net)	1100	0.00	60.15 f
tmp_reg[500]/RN (DFFRHQX1) <-		0.00 #	60.15 f
data arrival time			60.15

(Path is unconstrained)			

原来插入的 buffer 现在没有了。不过令我不明白的是 cell 上竟然没有大延迟。等知道的朋友解答。

<!--[if !supportLists]-->1. <!--[endif]-->一般信号

Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00 f
s_r2 (in)		0.00	60.00 f
s_r2 (net)	1	0.00	60.00 f
C5515/Y (AND2X4)		0.15 #	60.15 f
s_r (net)	1100	0.00	60.15 f
U1704/Y (OR2X2)		0.17 #	60.32 f
N601 (net)	1	0.00	60.32 f
tmp_reg[500]/D (DFFRHQX1)		0.00	60.32 f
data arrival time			60.32

clock clk (rise edge)		100.00	100.00
clock network delay (ideal)		0.00	100.00
tmp_reg[500]/CK (DFFRHQX1)		0.00	100.00 r
library setup time		-0.36	99.64
data required time			99.64

data required time			99.64
data arrival time			-60.32

slack (MET)			39.32

同样 dc 也没有插入延迟，和希望的一样。不过也没有出现大延迟，不是很明白

问题:

虽然设置了 `set_ideal_net(network)`, `set_dont_touch(network)` 但是 clock 上仍然有大延迟。

为了解决这个问题, 我们还要继续设置高扇出的选项。

`<!--[if !supportLists]-->1. <!--[endif]-->high_fanout_net_threshold,`
这个变量是用来指出, 如果 net 的扇出个数超过指定值, 那么他就是高扇出, 同时 drc 检查, 还有延迟计算都是这个数值计算, 但是时间上 net 上的扇出是没有变的。

`<!--[if !supportLists]-->2. <!--[endif]-->high_fanout_net_pin_capacitance,` 结合 `high_fanout_net_threshold` 使用的, 当 net 的扇出超过 threshold, 那么 net 上的负载等于这 2 个数值的乘积。

进一步: 修改脚本:

```
set lib $env(DC_LIB)

set target_library "slow.db fast.db"

set link_library "*" $target_library"

set search_path ". ../src ../scripts $lib"

set hdlin_while_loop_iterations 5000

analyze -format verilog test.v

elaborate test

uniquify

link
```

check_design

create_clock -period 100 [get_ports clk]

set_input_exp_clk [remove_from_collection [all_inputs] [get_ports clk]]

set_input_delay 60 -clock [get_clocks clk] \$input_exp_clk

set_output_delay 30 -clock [get_clocks clk] [all_outputs]

set_ideal_network -no_propagate [get_nets s_r]

set_ideal_network -no_propagate [get_nets rst_N]

set_high_fanout_net_threshold 60

set_high_fanout_net_pin_capacitance 0.01

compile

时序分析:

<!--[if !supportLists]-->1. <!--[endif]-->clock

```

*****
Report : timing
        -path full
        -delay max
        -nets
        -max_paths 1
Design : test
Version: Z-2007.03-SP1
Date   : Mon May 12 15:53:44 2008
*****

# A fanout number of 20 was used for high fanout net computations

Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

Startpoint: clk_G (input port clocked by clk)
Endpoint: tmp_reg[500]/CK (internal pin)
Path Group: (none)
Path Type: max

Point                Fanout    Incr    Path
-----
input external delay                60.00    60.00 r
clk_G (in)                      0.00    60.00 r
clk_G (net)                       1      0.00    60.00 r
U1103/Y (AND2X4)                  0.44 #    60.44 r
G_clk (net)                   1100    0.00    60.44 r
tmp_reg[500]/CK (DFFRHQX1)        0.00 #    60.44 r
data arrival time                                60.44
-----

```

可以发现 cell 的延迟已经很合理。

```
<!--[if !supportLists]-->1.    <!--[endif]-->reset
```

```

*****
Report : timing
        -path full
        -delay max
        -nets
        -max_paths 1
Design : test
Version: Z-2007.03-SP1
Date   : Tue May 13 10:42:41 2008
*****

# A fanout number of 60 was used for high fanout net computation

Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

Startpoint: rst_N2 (input port clocked by clk)
Endpoint: tmp_reg[500]/RN (internal pin)
Path Group: (none)
Path Type: max

Point                Fanout    Incr    Path
-----
input external delay                60.00    60.00 f
rst_N2 (in)                      0.00    60.00 f
rst_N2 (net)                      0.00    60.00 f
C5514/Y (AND2X4)                  0.15 #   60.15 f
rst_N (net)                      0.00    60.15 f
tmp_reg[500]/RN (DFFRHQX1) <-    0.00 #   60.15 f
data arrival time                    60.15

```

Cell 上的延迟和原来一样

<!--[if !supportLists]-->1. <!--[endif]-->一般信号

```

Startpoint: s_r2 (input port clocked by clk)
Endpoint: tmp_reg[500]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

```

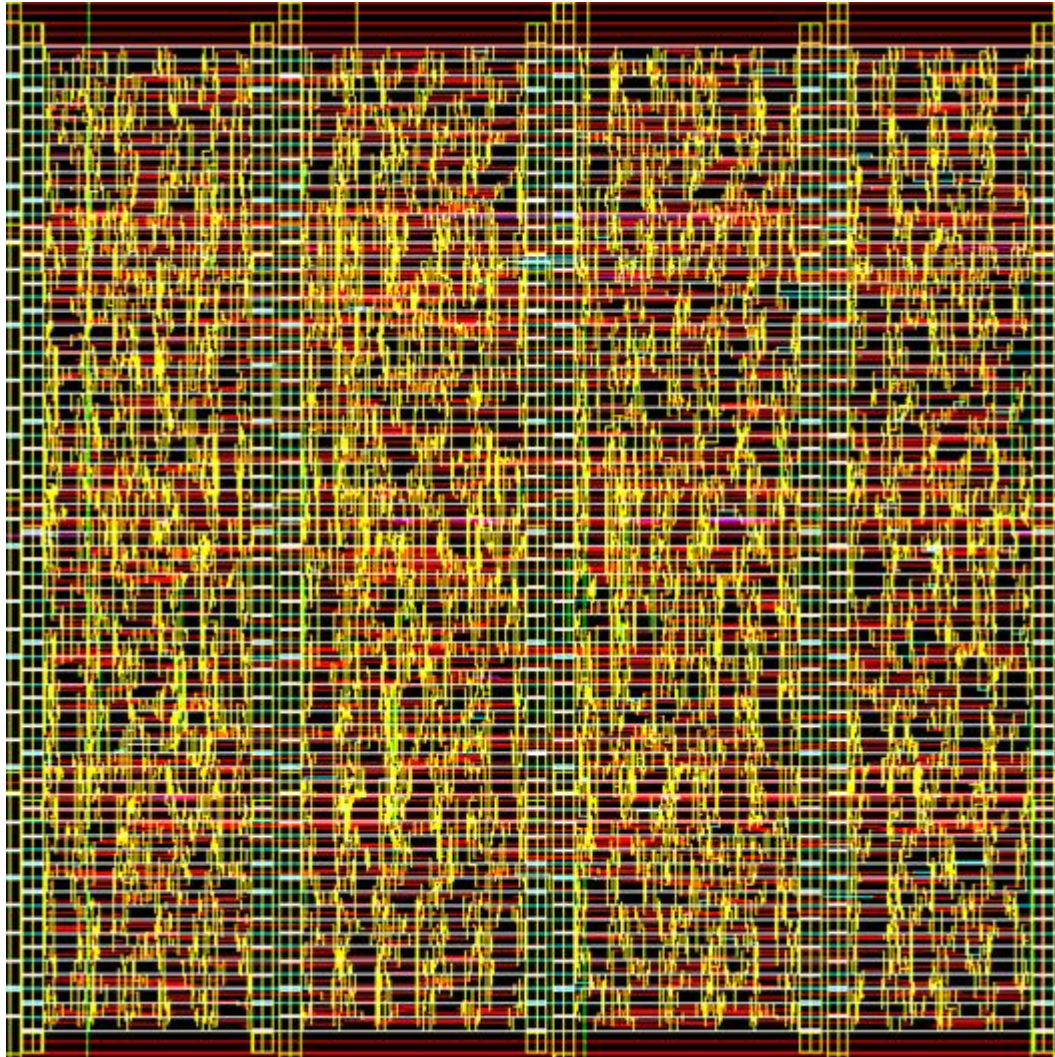
Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00 f
s_r2 (in)		0.00	60.00 f
s_r2 (net)	1	0.00	60.00 f
C5515/Y (AND2X4)		0.15 #	60.15 f
s_r (net)	1100	0.00	60.15 f
U1704/Y (OR2X2)		0.17 #	60.32 f
N601 (net)	1	0.00	60.32 f
tmp_reg[500]/D (DFFRHQX1)		0.00	60.32 f
data arrival time			60.32
clock clk (rise edge)		100.00	100.00
clock network delay (ideal)		0.00	100.00
tmp_reg[500]/CK (DFFRHQX1)		0.00	100.00 r
library setup time		-0.36	99.64
data required time			99.64

data required time			99.64
data arrival time			-60.32

发现和原来一样。

Apr 之后



导出网标，修改约束文件成初始状态，继续分析时序：

Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00
s_r1 (in)		0.00	60.00
s_r1 (net)	1	0.00	60.00
U1338/Y (AND2X2)		0.16	60.16
n1 (net)	3	0.00	60.16
U1337/Y (INVX1)		0.14	60.30
n8 (net)	4	0.00	60.30
U1328/Y (INVX1)		0.13	60.43
n5 (net)	5	0.00	60.43
U1284/Y (INVX1)		0.16	60.58
n109 (net)	4	0.00	60.58
U1149/Y (INVX1)		0.22	60.80
n63 (net)	12	0.00	60.80
U1944/Y (OR2X2)		0.24	61.04
N601 (net)	1	0.00	61.04
tmp_reg[500]/D (DFFRHQX1)		0.00	61.04
data arrival time			61.04

clock clk (rise edge)		100.00	100.00
clock network delay (ideal)		0.00	100.00
tmp_reg[500]/CK (DFFRHQX1)		0.00	100.00
library setup time		-0.36	99.64
data required time			99.64

data required time			99.64
data arrival time			-61.04

```

Wire Load Model Mode: top

Startpoint: rst_N1 (input port clocked by clk)
Endpoint: tmp_reg[500]/RN (internal pin)
Path Group: (none)
Path Type: max

Point                Fanout    Incr    Path
-----
input external delay                60.00    60.00 r
rst_N1 (in)                      0.00    60.00 r
rst_N1 (net)                      1      0.00    60.00 r
U1104/Y (NAND2X1)                  0.10    60.10 f
n2 (net)                          4      0.00    60.10 f
U1331/Y (INVSX1)                   0.18    60.28 r
n125 (net)                        5      0.00    60.28 r
U1308/Y (INVSX1)                   0.11    60.40 f
n219 (net)                        4      0.00    60.40 f
U1334/Y (INVSX1)                   0.18    60.57 r
n128 (net)                        5      0.00    60.57 r
U1317/Y (INVSX1)                   0.11    60.69 f
n228 (net)                        4      0.00    60.69 f
U1232/Y (INVSX1)                   0.41    61.10 r
n167 (net)                       13      0.00    61.10 r
tmp_reg[500]/RN (DFFRHQX1) <-    0.00    61.10 r
data arrival time                  61.10
-----
(Path is unconstrained)

```

可以看出 apr 工具自动加入了 buffer，同时优化了 net 上的 fanout

总结：

为了让 dc 在综合高扇出的 net 时候不插入 buffer tree 和 buffer chain，需要使用 set_idea_network 使这些搞扇出 net 避免时序优化（timing optimization），时序更新（timing update），drc 修正（drc fixxing）。但是这样设置之后，net 上的高负载并没有消除，我们需要额外的参数进行设置。

high_fanout_net_threshold high_fanout_net_pin_capacitance .以减少 dc 综合时间，以及减少 timing violators report

常用的技巧

五一节了，手头也没有什么急事，正好总结一下这段时间 debug SOC 所采用的一些技巧。SOC 大都带有一个 MCU，SOC 中其它的模块都在 MCU 的控制之下工作。在前期开发的过程中，因为考虑到 FW 开发的周期和开发速度的要求，验证都习惯使用一个 debug FW 来进行。只有当项目进展到一定程度的时候，才会使用比较完整的 FW 版本。在系统开发的过程中，遇到 bug，首先要定位问题到底来自于软件还是硬件。一般的做法是，在 waveform 中 trace MCU 的 PC 指针，追踪当前 bug 发生之前 FW 做的主要控制：是否正常的进入了系统中断，程序跳转是否正常，FW 是否给出了正确地控制信号？若上述都没有问题，则 bug 很有可能是由硬件造成的，再 trace 到 RTL 中去定位问题就可以了。

有时候，软硬件都没有问题，而问题发生在软硬件的配合之中，我就曾经遇到过一次。当系统频率较低的时候，没有问题；提高频率跑 debug FW 也没有发现问题；只有当告诉跑 real FW 的时候问题才会出现。后来才发现，因为中断服务程序的执行需要时间，在高速的情况下，debug FW 比较短，控制信号能够及时送出，但用 real FW 的时候控制信号却送晚了，这才造成了操作失败。解决的办法只有将原来的信号改由硬件控制。

SOC 的 debug 由于 FW 的参与而变得比较麻烦，即使 FW 不是自己写的，一般来说也要看一下主要的操作步骤。这样才能更快的定位并解决问题。

这段日子在看《Digital Intergrated Circuits》的 Timing 一章，正好把数字电路设计中关于时钟和 Timing 的问题总结一下。下面是我所知的一些待解决的问题：

- 1) Clock skew and jitter;
- 2) Clock generation and clock tree;
- 3) Asnyc & Sync circuit; Synchronization;
- 4) Clock MUX;
- 5) 分频问题;
- 6) 门控时钟问题;
- 7) 关键路径的优化;

VCS 对 verilog 模型进行仿真包括两个步骤:

1. 编译 verilog 文件成为一个可执行的二进制文件命令为:

```
$> vcs source_files
```

2. 运行该可执行文件

```
$> ./simv
```

类似于 NC，也有单命令行的方式:

`$> vcs source_files -R`

`-R` 命令表示, 编译后立即执行.

下面讲述常用的命令选项:

`-cm line|cond|fsm|tgl|obc|path` 设定 coverage 的方式

`+define+macro=value+` 预编译宏定义

`-f filename` RTL 文件列表

`+incdir+directory+` 添加 include 文件夹

`-l` 进入交互界面

`-l logfile` 文件名

`-P pli.tab` 定义 PLI 的列表(Tab)文件

`+v2k` 使用推荐的标准

`-y` 定义 verilog 的库

`-notice` 显示详尽的诊断信息

`-o` 指定输出的可执行文件的名字,缺省是 `sim.v`

DC 概论六之 `multicycle_path`

这几天看书看到这个概念, 多周期路径。书上有点解释: 指的是两个寄存器之间数据要经过多个时钟才能稳定的路径, 一般出现于组合逻辑较大的那些路径。

而且有指出, 在实际工程中, 除了乘除法器等少数比较特殊的电路,

一般应该尽量避免采用多周期路径电路。即使有所使用，也应该通过约束在综合工具中指出该路径，使得综合工具在计算Fmax的时候忽略这条路径，避免用大量的时间对该路径进行优化。

对多周期路径可加一下约束：set_multicycle_path -from D_reg -to S_reg

sta 工具会分析所有的 path,除了定义为 false path 的路径.所有的 path 都在最近的沿分析 setup,在最近的沿的前一个沿分析 hold

设置为 multicycle 的 path 会改变分析分析方法，比如 multicycle 2 就是说会在最近的沿的下一个沿分析 setup,分析 setup 的沿的前一个沿分析 hold.

在做 pre layout sta 分析时,delay 的最大值还是最小值要看你调用的 WLMS 是 fast 还是 slow 的

在做 post layout sta 分析时,delay 的最大值还是最小值要看你调用的真实 RC 参数(如 spef)是 worst 还是 best 的

data path 的计算方法都一样,不存在最大最小,只存在调用的 timing 分析参数/模型不一样

multicycle 不是以延时多少 ns 来计算的，是按照多少个周期计算。因为你路径是两个寄存器之间的路径，

所以，即使你得 multicycle 的结果稳定下来，也需要下一个时钟周期，寄存器才到，才算结束。

当你的 code 模块是 multicycle 时，你需要在约束时进行设置

在讲多周期路径之前，先看下单频率路径的建立关系和保持关系

『Design Compiler calculates the default setup and hold relations and derives single-cycle timing, based on active edges.』

- 1.对于 startpoint, active edge 是寄存器的 open edge。
- 2.对于 endpoint, active edge 是寄存器的 close edge。
- 3.对于上升沿触发的寄存器，上升沿既是 open edge 也是 close edge。
- 4.对于高电平出发的锁存器，上升沿是 open edge，下降沿是 close edge。

Figure 2-6 Single-Cycle Timing Active Edges and Setup for Rising-Edge-Triggered Flip-Flop

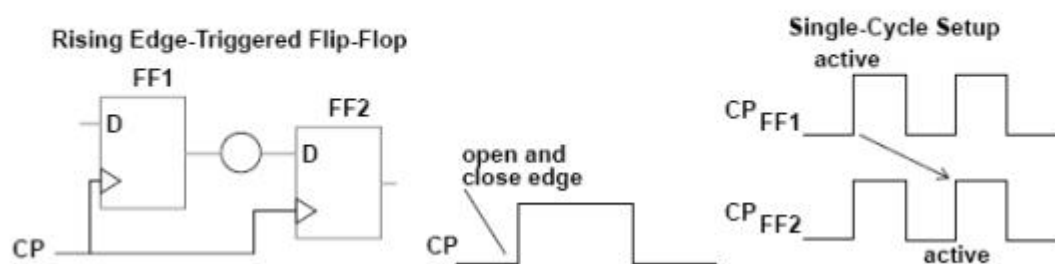
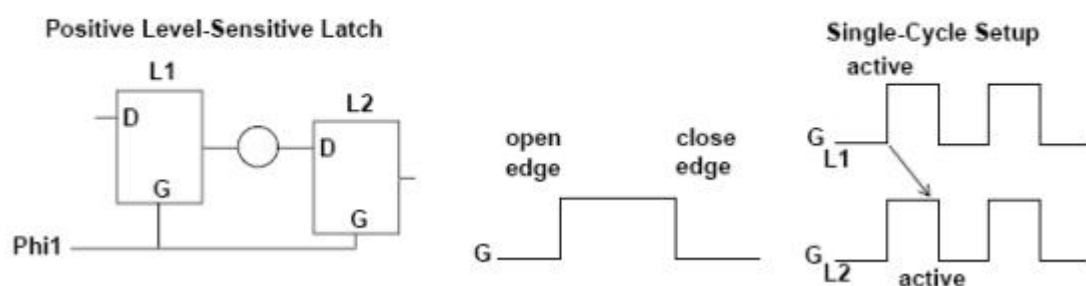
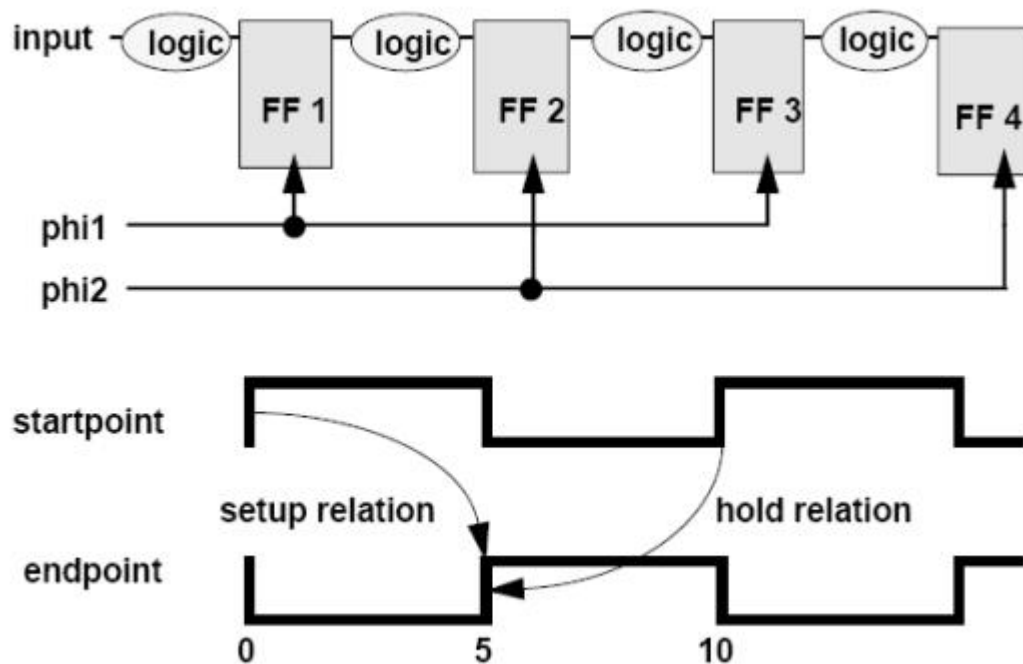


Figure 2-7 Single-Cycle Timing Active Edges and Setup for Positive Level-Sensitive Latch



看一个例子：



下面进入多频率路径建立和保持

建立关系检查：

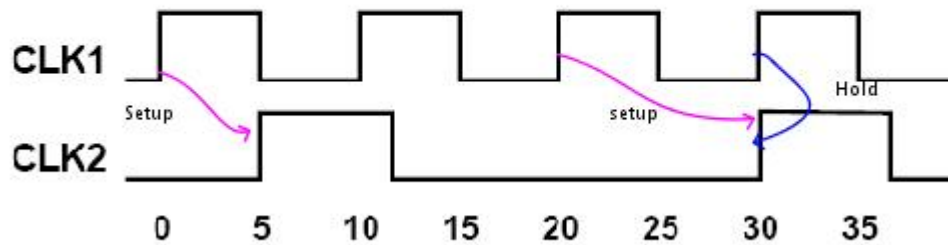
对于多频率设计，在两个时钟之间可能存在多个建立关系，对于目的时钟的每一个锁存边沿，找到捕获边沿最近的发送边沿。发送边沿和捕获边沿的最小关系决定了这个路径上的最大延迟。

保持关系检查：

简单说就是，数据从 **startpoint** 发出之后，在被 **endpoint** 的 active edge 锁存之前，不能被捕获。这个时序的最大值决定了，这个路径的最小延迟。

如下图所示：

```
create_clock -period 10 -waveform {0 5} CLK1
create_clock -period 25 -waveform {5 12.5} CLK2
```

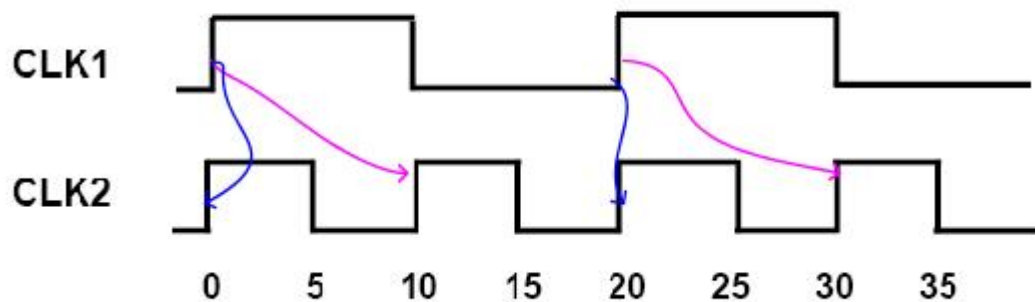


Setup 要求: $5-0=5$

Hold 要求: $30-30=0$

Exp2:

```
create_clock -period 20 -waveform {0 10} CLK1
create_clock -period 10 -waveform {0 5} CLK2
```

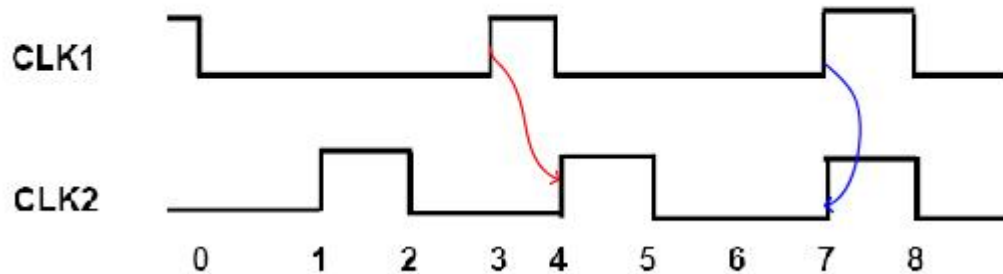


Setup 要求= $10-0=10$

Hold 要求= $0-0=0$

Exp3

```
create_clock -period 4 -waveform {3 4} CLK1  
create_clock -period 3 -waveform {1 2} CLK2
```



Setup 要求: $4-3=1$

Hold 要求: $7-7=0$

下面进入单频率的多周期:

先介绍命令 `set_multicycle_path`

常用格式:

`set_multicycle_path`

`path_multiplier`

`[-rise | -fall]`

`[-setup | -hold]`

`[-start | -end]`

`[-from from_list]`

`[-to to_list]`

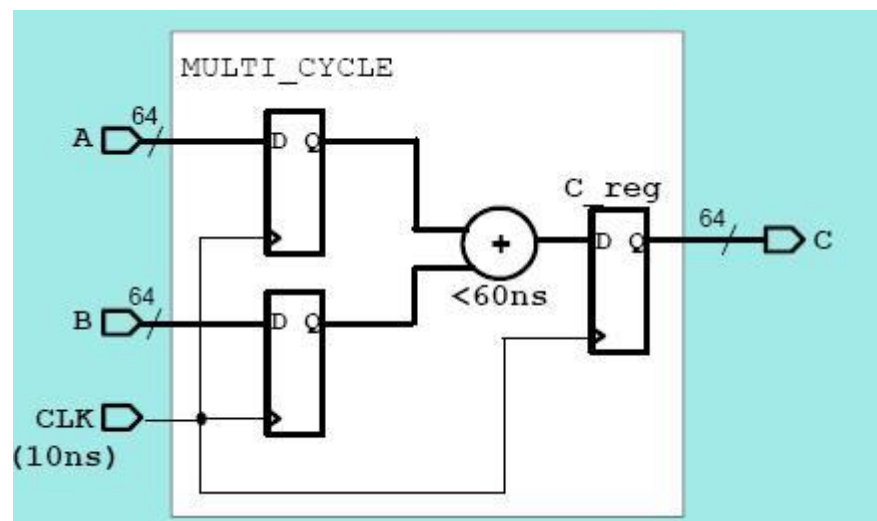
`[-through through_list]`

Rise 和 fall 用来说明多周期路径是用在上升沿还是下降沿

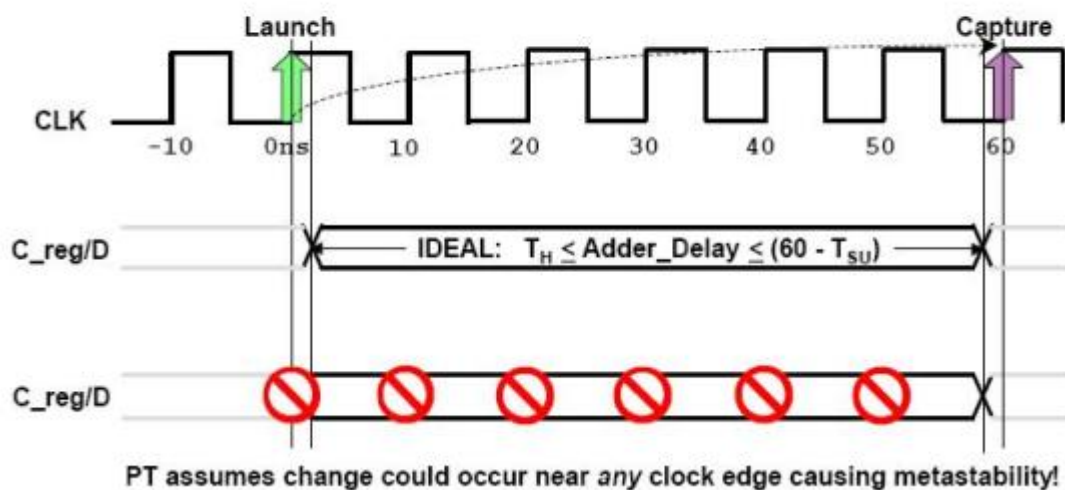
Setup 和 hold 说明多周期路径是用在建立时间检查还是用在保持时间检查。

Start 和 end 说明多周期路径依赖于 start clock 还是依赖于 end clock

例子：



```
create_clock -period 10 [get_ports CLK]
set_multicycle_path -setup 6 -to [get_pins "C_reg[*]/D"]
```

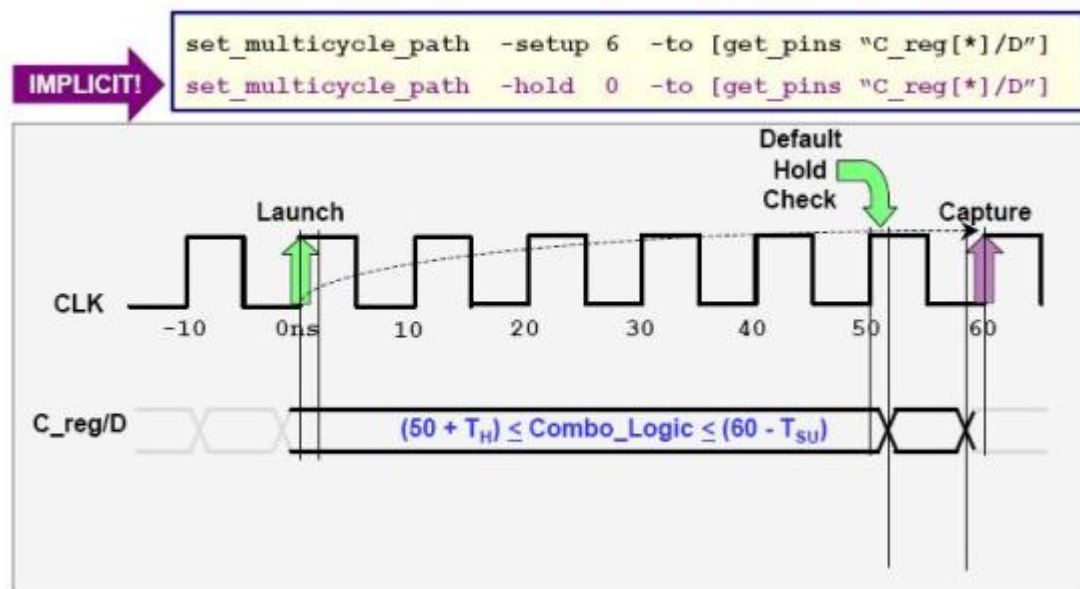


Set_multicycle_path -setup 60 -to [get_pins "C_reg[*]/D"]

用来说明多周期路径的建立时间检查。

换句话说就是数据从 launch 到 capture 的时间是 6 个周期的时间。

那么缺省的保持时间检查是什么呢？看下面

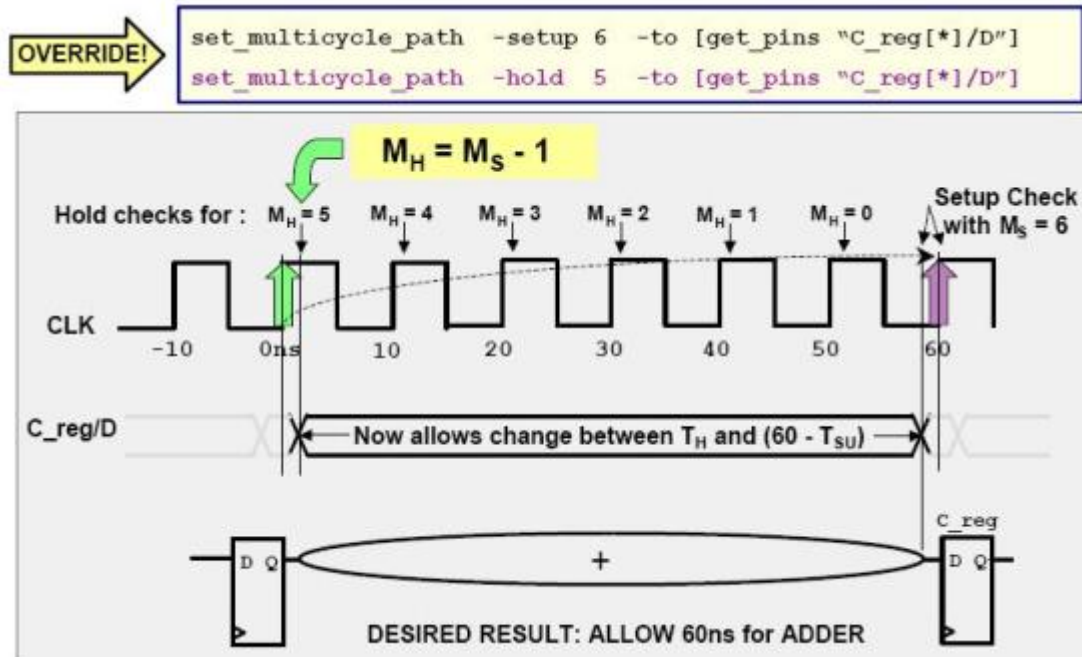


根据 dc 综合原理：

对 hold 的检查之一就是：在 capture active edge 之前不能被捕获。

所以缺省的 hold 检查实在第 6 个周期。而我们希望的是在 0 时刻进行 hold check。

所以要如下设置



但是这种写法很晦涩。我们可以通过 `set_min_dealy` 来设置 hold check:

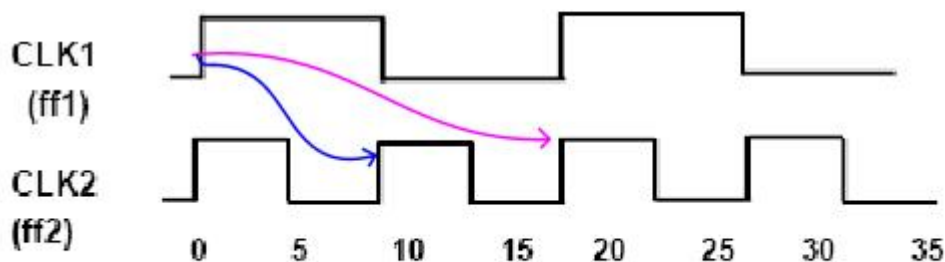
`set_min_dealy -to C_reg[0]/D`

下面进入多频率的多周期:

```

create_clock -period 20 -waveform {0 10} CLK1
create_clock -period 10 -waveform {0 5} CLK2
set_multicycle_path 2 -setup -from ff1/CP -to ff2/D

```



这个例子和上面例子差不多，只是频率不同。

例子：以下仅为例子，实际设计中不会存在

```
module m(clka, clkb, ain, bin ,cout, dout);
```

```
input clka,clkb;
```

```
input [3:0] ain,bin;
```

```
output [7:0] cout,dout;
```

```
reg [7:0] cout, dout;
```

```
reg [3:0] ain_reg ,bin_reg;
```

```
wire [7:0] mul;
```

```
assign mul = ain_reg * bin_reg;
```

```
always@(posedge clka)
```

```
begin
```

```
    ain_reg <= ain;
```

```
    cout <= mul;
```

```
end
```

```
always@(posedge clkb)
```

begin

bin_reg <= bin;

dout <= mul;

end

endmodule

dc脚本

set lib \$env(DC_LIB)

set target_library "slow.db fast.db"

set link_library "* \$target_library"

set search_path ". ../src ../scripts \$lib"

analyze -format [verilog](#) m.v

elaborate m

compile

create_clock -period 10 [get_ports clka]

create_clock -period 30 [get_ports clk_b]

set_multicycle_path 2 -to cout_reg[*]/D

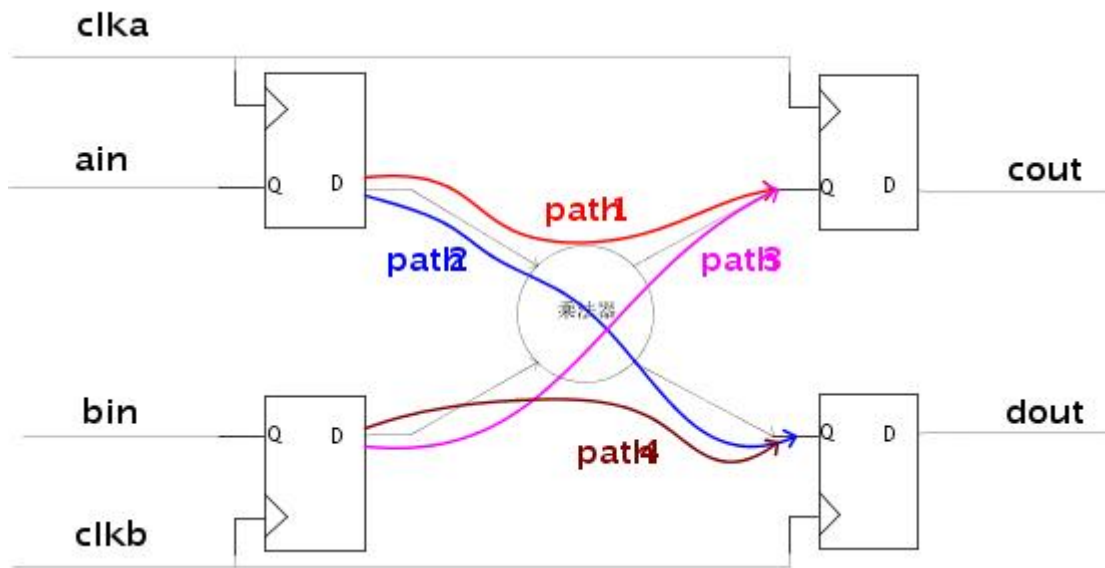
set_multicycle_path 3 -to dout_reg[*]/D

set_min_delay 0 -to dout_reg[3]/D

set_min_delay 0 -to cout_reg[3]/D

compile

结构图



原来以为路径 path4 和 path1, path2, path3 上的建立时间和保持时间检查的分析方法一项。所以上篇的 timing report 仅仅分析了一下 path1 和 path2。后来觉得有些疑问，然后分析了下 path4（慢时钟采集快数据），发现 dc 一个很微妙的分析方法，后来在 dc 的文档中发现的。后来想了下，这种建立关系和保持关系的检查其实在 dc 的文档中还是说明了的。先看分析，再总结：

综合脚本：

```
set lib $env(DC_LIB)

set target_library "slow.db fast.db"

set link_library "* $target_library"

set search_path ". ../src ../scripts $lib"

analyze -format verilog m.v
```

elaborate m

compile

create_clock -period 5 [get_ports clka]

create_clock -period 30 [get_ports clk_b]

set_multicycle_path 3 -to cout_reg[*]/D

set_multicycle_path 4 -to dout_reg[*]/D

set_min_delay 0 -to dout_reg[3]/D

set_min_delay 0 -to cout_reg[3]/D

compile

path4 分析:

Setup:

report_timing -from ain_reg_reg[*]/CK -to dout_reg[4]/D

```

(rising edge-triggered flip-flop clocked by clk
Endpoint: dout_reg[4]
(rising edge-triggered flip-flop clocked by clkb)
Path Group: clkb
Path Type: max

```

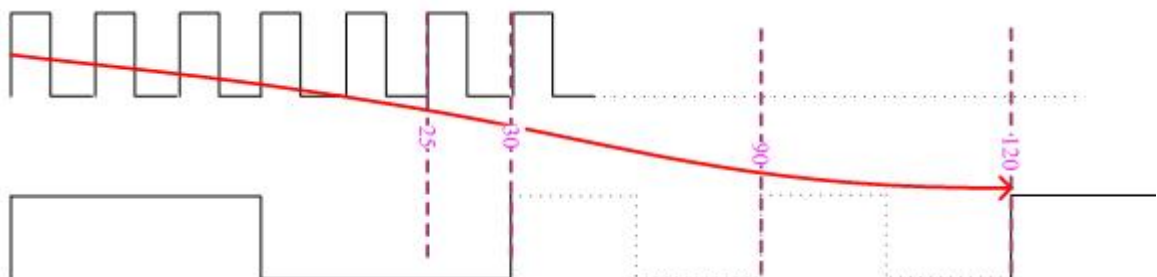
Point	Incr	Path

clock clka (rise edge)	25.00	25.00
clock network delay (ideal)	0.00	25.00
ain_reg_reg[0]/CK (DFFHQX1)	0.00	25.00 r
ain_reg_reg[0]/Q (DFFHQX1)	0.30	25.30 r
mult_11/A[0] (m_DW02_mult_0)	0.00	25.30 r
mult_11/U46/Y (INVX1)	0.10	25.40 f
mult_11/U21/Y (NOR2X1)	0.19	25.59 r
mult_11/U31/Y (AND2X2)	0.18	25.77 r
mult_11/S2_2_1/CO (ADDFX2)	0.57	26.34 r
mult_11/S4_1/S (ADDFX2)	0.60	26.94 f
mult_11/U19/Y (XOR2X1)	0.27	27.21 f
mult_11/PRODUCT[4] (m_DW02_mult_0)	0.00	27.21 f
dout_reg[4]/D (DFFHQX1)	0.00	27.21 f
data arrival time		27.21

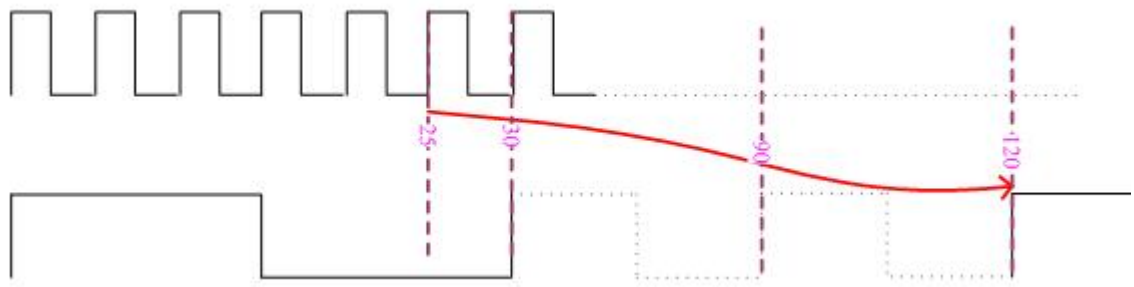
clock clkb (rise edge)	120.00	120.00
clock network delay (ideal)	0.00	120.00
dout_reg[4]/CK (DFFHQX1)	0.00	120.00 r
library setup time	-0.34	119.66
data required time		119.66

可以看出来慢时钟采集快数据的多周期路径分析，不是像快时钟采集慢数据的多周期分析一样（下图）：

希望的慢周期采集快数据的多周期分析



根据 timing report 可以知道实际的分析是：



再看下保持时间的分析

```
report_timing -from ain_reg_reg[*]/CK -to dout_reg[4]/D
-delay min
```

```
Startpoint: ain_reg_reg[3]
           (rising edge-triggered flip-flop clocked by clka)
Endpoint:  dout_reg[4]
           (rising edge-triggered flip-flop clocked by clkb)
Path Group: clkb
Path Type: min
```

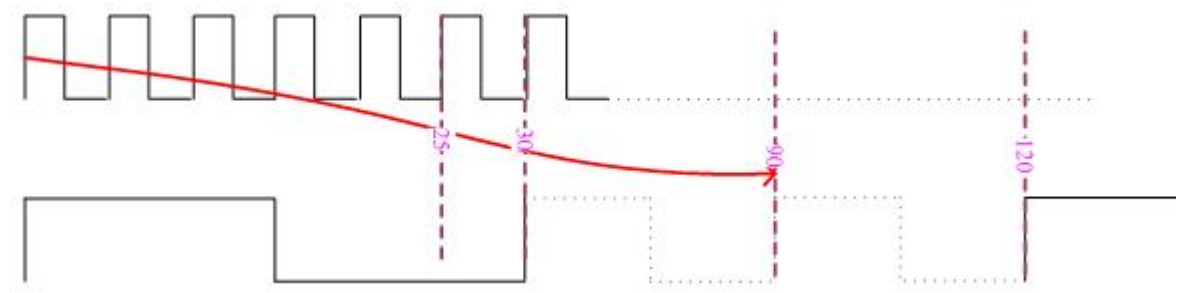
Point	Incr	Path

clock clka (rise edge)	30.00	30.00
clock network delay (ideal)	0.00	30.00
ain_reg_reg[3]/CK (DFFHQX1)	0.00	30.00 r
ain_reg_reg[3]/Q (DFFHQX1)	0.25	30.25 f
mult_11/A[3] (m_DW02_mult_0)	0.00	30.25 f
mult_11/U41/Y (INVX1)	0.16	30.42 r
mult_11/U26/Y (NOR2X1)	0.09	30.50 f
mult_11/S4_1/S (ADDFX2)	0.42	30.93 r
mult_11/U19/Y (XOR2X1)	0.17	31.10 r
mult_11/PRODUCT[4] (m_DW02_mult_0)	0.00	31.10 r
dout_reg[4]/D (DFFHQX1)	0.00	31.10 r
data arrival time		31.10

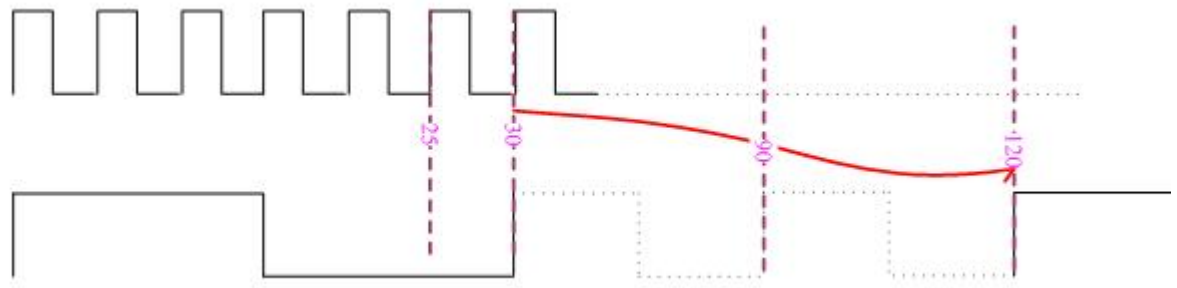
clock clkb (rise edge)	120.00	120.00
clock network delay (ideal)	0.00	120.00
dout_reg[4]/CK (DFFHQX1)	0.00	120.00 r
library hold time	-0.08	119.92
data required time		119.92

data required time		119.92

快时钟采慢数据的缺省 hold chek 如下，也是我们希望：



而实际上，dc 的分析师如下的：



所以对于慢时钟采集快数据的 setup, hold check 的分析，可以由上面的分析总结为：

对于 setup 分析：

对于每一个 capture active edge 找到最靠近它的 launch active edge, 然后这个 launch active edge 到多周期的 capture active edge 的路径就是 setup check 的路径。

对于 hold 分析：

对于每一个 capture active edge 找到它的 launch active edge 的下一个 launch active edge, 然后这个 launch active edge 到多周期的 capture active edge 的路径就是 hold check 路径。

更简单的说，慢时钟采集快数据，是先分析慢时钟的单周期 setup relation 和 hold relation, 然后再扩展到多周期路径上。

其实就是将单周期的 capture active edge, 换成多周期后的 capture active edge (多周期的相当于周期很大)

对于 hold 分析的理解, 可以将综合脚本改成下面的, 然后再分析下, 就会很明白了。

```
set lib $env(DC_LIB)

set target_library "slow.db fast.db"

set link_library "* $target_library"

set search_path ". ../src ../scripts $lib"

analyze -format verilog m.v

elaborate m

compile

create_clock -period 5 [get_ports clka]

create_clock -period 31 [get_ports clkb]

set_multicycle_path 3 -to cout_reg[*]/D

set_multicycle_path 4 -to dout_reg[*]/D

set_min_delay 0 -to dout_reg[3]/D

set_min_delay 0 -to cout_reg[3]/D
```

compile

path4:

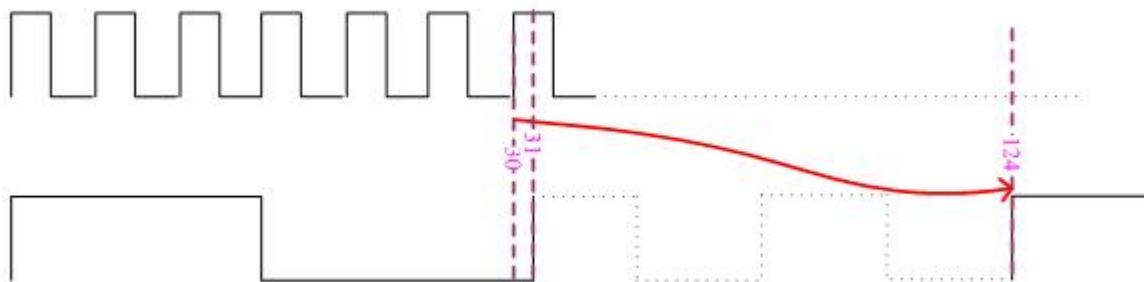
setup 分析:

report_timing -from ain_reg_reg[*]/CK -to dout_reg[4]/D

```
Startpoint: ain_reg_reg[0]
(rising edge-triggered flip-flop clocked by clka)
Endpoint: dout_reg[4]
(rising edge-triggered flip-flop clocked by clkb)
Path Group: clkb
Path Type: max
```

Point	Incr	Path
clock clka (rise edge)	30.00	30.00
clock network delay (ideal)	0.00	30.00
ain_reg_reg[0]/CK (DFFHQX1)	0.00	30.00 r
ain_reg_reg[0]/Q (DFFHQX1)	0.30	30.30 r
mult_11/A[0] (m_DW02_mult_0)	0.00	30.30 r
mult_11/U46/Y (INVX1)	0.10	30.40 f
mult_11/U21/Y (NOR2X1)	0.19	30.59 r
mult_11/U35/Y (AND2X2)	0.18	30.77 r
mult_11/S2_2_1/CO (ADDFX2)	0.57	31.34 r
mult_11/S4_1/S (ADDFX2)	0.60	31.94 f
mult_11/U19/Y (XOR2X1)	0.27	32.21 f
mult_11/PRODUCT[4] (m_DW02_mult_0)	0.00	32.21 f
dout_reg[4]/D (DFFHQX1)	0.00	32.21 f
data arrival time		32.21
clock clkb (rise edge)	124.00	124.00
clock network delay (ideal)	0.00	124.00
dout_reg[4]/CK (DFFHQX1)	0.00	124.00 r
library setup time	-0.34	123.66

示意图如下:



Hold 分析:

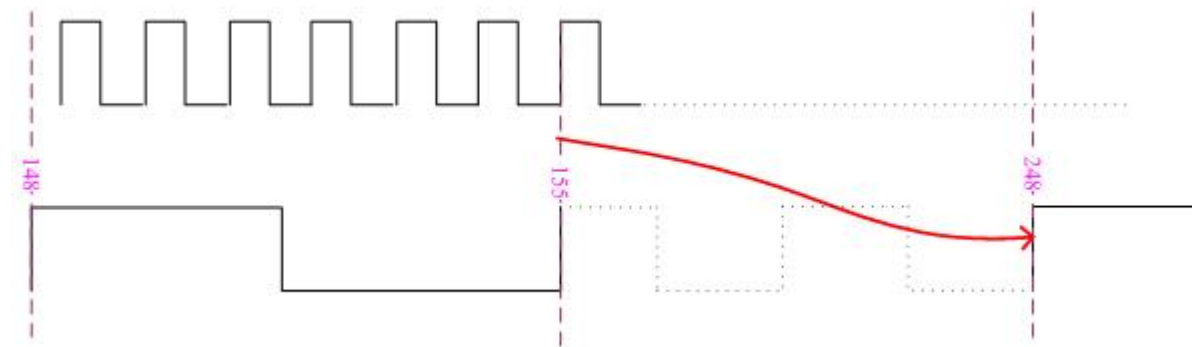
```
report_timing -from ain_reg_reg[*]/CK -to dout_reg[4]/D
```

```
-delay min
```

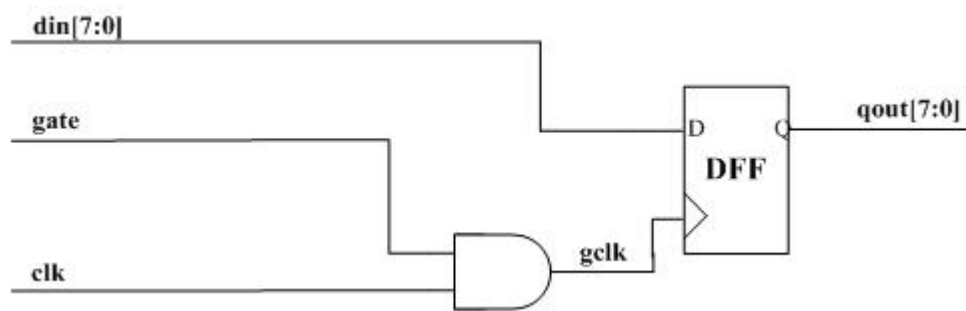
```
Startpoint: ain_reg_reg[3]
(rising edge-triggered flip-flop clocked by clka)
Endpoint: dout_reg[4]
(rising edge-triggered flip-flop clocked by clk b)
Path Group: clk b
Path Type: min
```

Point	Incr	Path
clock clka (rise edge)	155.00	155.00
clock network delay (ideal)	0.00	155.00
ain_reg_reg[3]/CK (DFFHQX1)	0.00	155.00 r
ain_reg_reg[3]/Q (DFFHQX1)	0.25	155.25 f
mult_11/A[3] (m_DW02_mult_0)	0.00	155.25 f
mult_11/U45/Y (INVX1)	0.16	155.42 r
mult_11/U26/Y (NOR2X1)	0.09	155.50 f
mult_11/S4_1/S (ADDFX2)	0.42	155.93 r
mult_11/U19/Y (XOR2X1)	0.17	156.10 r
mult_11/PRODUCT[4] (m_DW02_mult_0)	0.00	156.10 r
dout_reg[4]/D (DFFHQX1)	0.00	156.10 r
data arrival time		156.10
clock clk b (rise edge)	248.00	248.00
clock network delay (ideal)	0.00	248.00
dout_reg[4]/CK (DFFHQX1)	0.00	248.00 r
library hold time	-0.08	247.92
data required time		247.92
data required time		247.92
data arrival time		156.10

示意图：

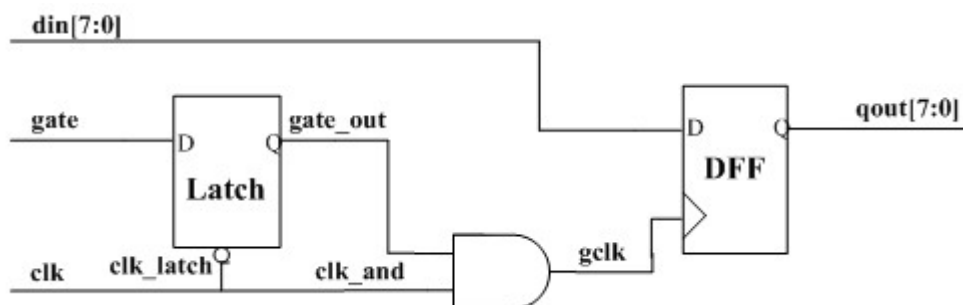


DC 概论七之 gated clock



上图是一个典型的门控时钟电路。但是这种门控时钟无法避免的要受到 `gate` 的影响，容易产生毛刺，除非你严格限制 `gate` 的输入。

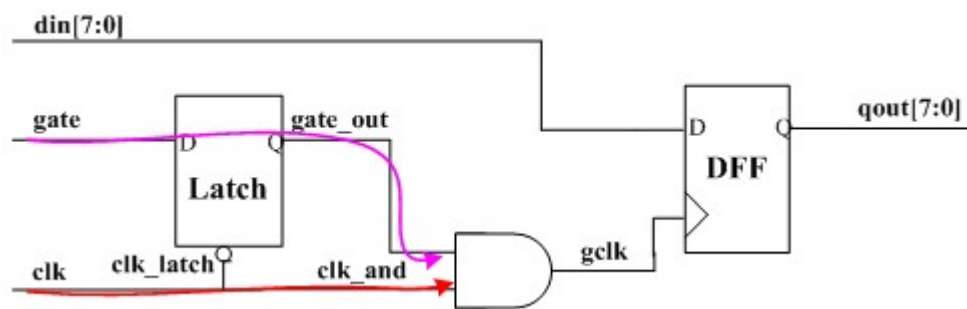
一般常用的门控时钟是下面这种形式



为了得到稳定的 gclk 高电平, 我们使用 clk 的高电平来锁存 gate 的值, 如果 gate 是低电平, 那么 gclk 保持低电平, 如果 gate 为高电平, 锁存后, gate_out 为高电平, 与 clk 运算后, 可以得到稳定的 gclk 高电平。

考虑上图中的各种延迟, 分析门控时钟的建立时间检查和保持时间检查。

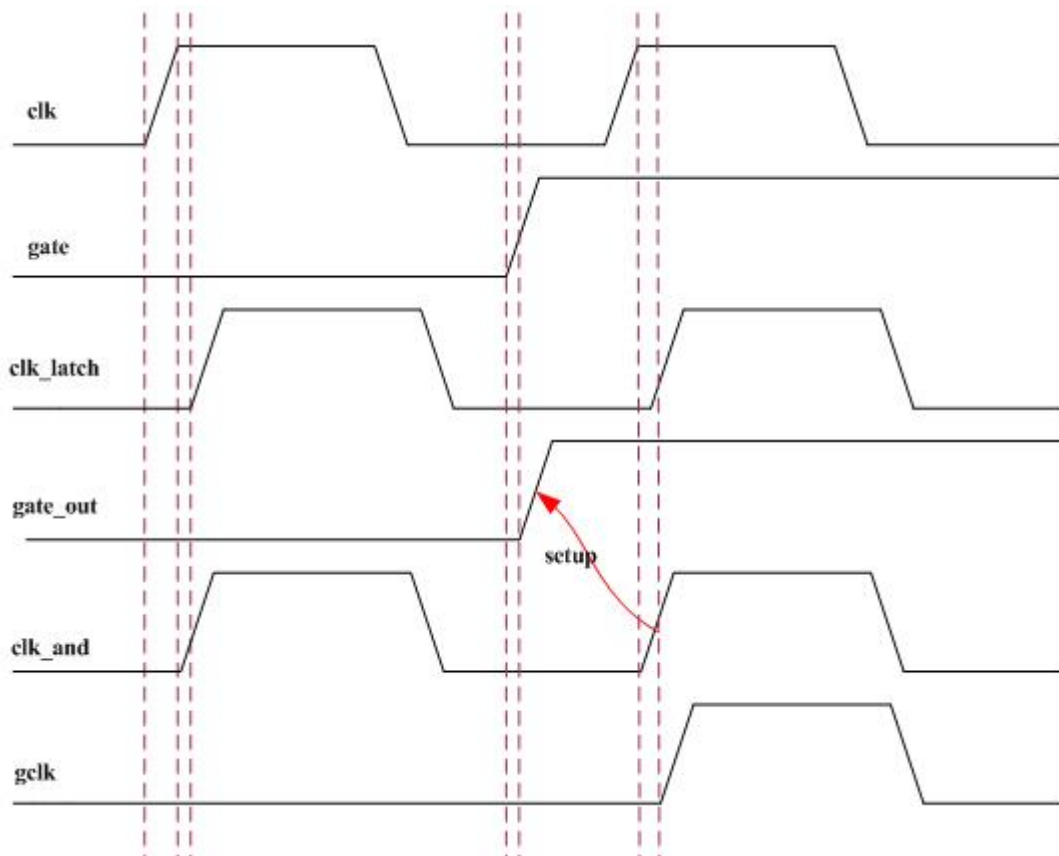
对于门控单元建立时间的检查路径:



门控的建立时间检查的路径中, 因为 `clk_latch` 为低电平, 所以理想化的可以认为, `gate` 直接传输到 `gate_out`, 但是实际要考虑点点延迟。

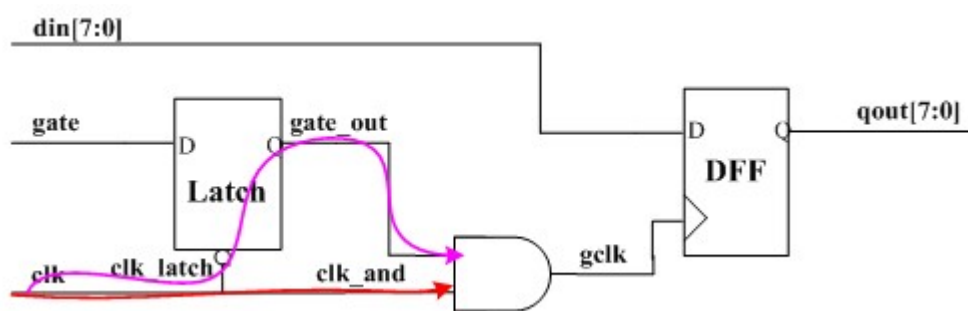
可以简化成图一

考虑到延迟的时序图:

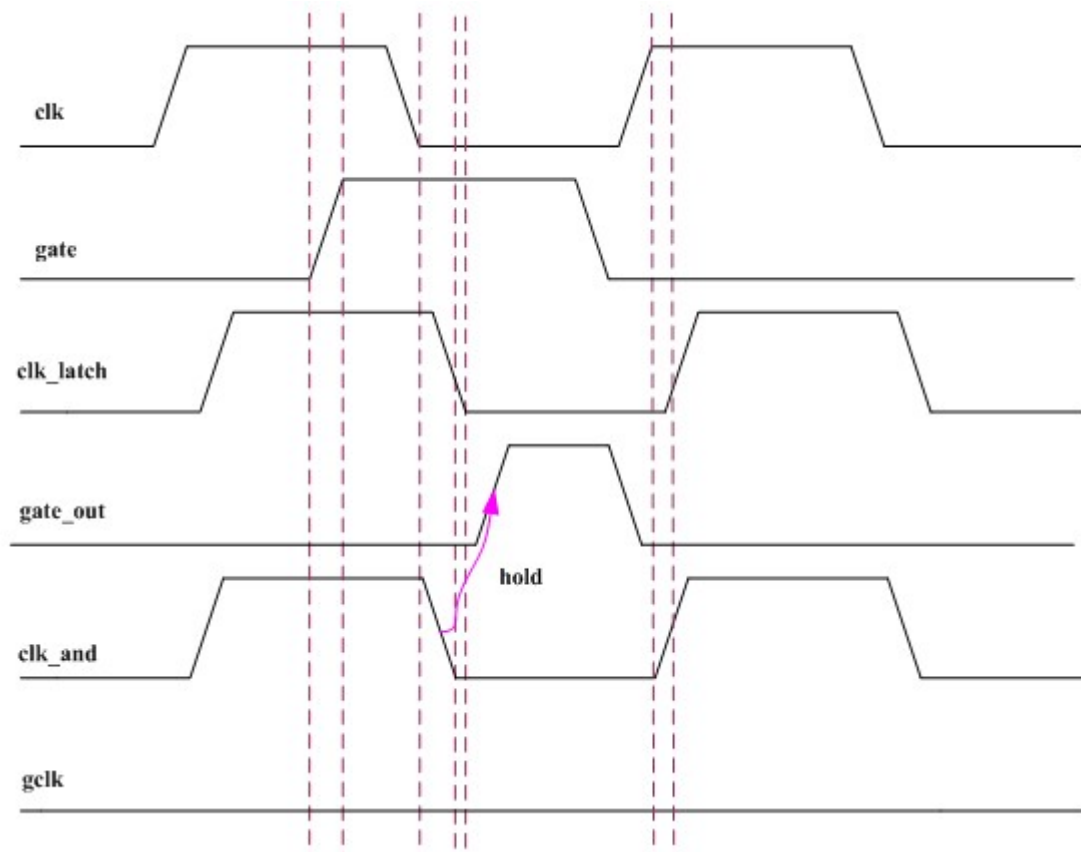


由时序图可以知道 **gate** 的上升沿或者下降沿到 **clk_and** 的时间应该长些比较好。

对于门控单元保持时间的检查路径：



考虑到延迟的保持时间检查：



由上图可以知道 clk_latch 的延迟如果比 clk_and 延迟大点，那么电路会更稳定些。

门控时钟综合：

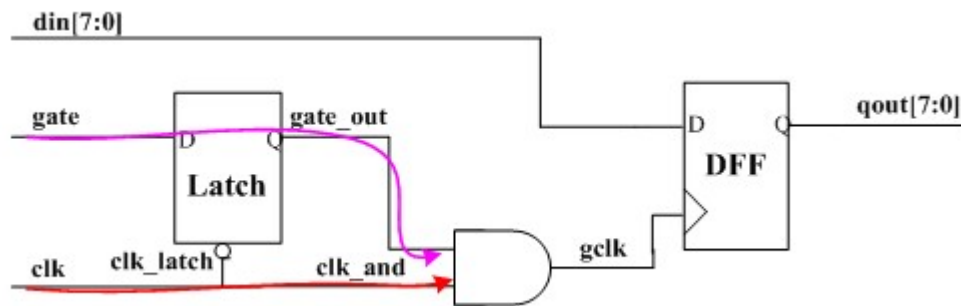
综合工具内置时钟故障分析，但是有时候会产生不正确的分析。虽然如此，由于优化算法比较强大，仍然可以得到理想的结果。

<!--[if !supportLists]-->1. <!--[endif]-->首先要开启时钟故障分析

```
set_clock_gating_check -setup $setup_time -hold $hold_time
```

\$setup_time 和\$hold_time 如上图的时序图所示

<!--[if !supportLists]-->2. <!--[endif]-->建立时间分析



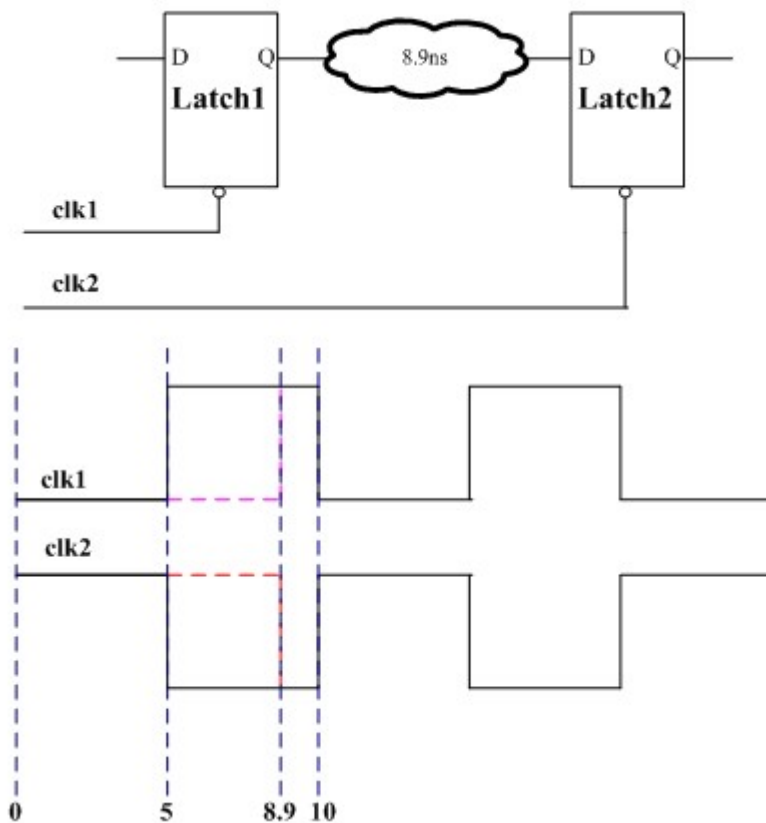
如上图所示，综合工具会认出门控时钟里的锁存器，同时标识这条路径为锁存路径，分析的时候会将这条路径分为 2 部分。

<!--[if !supportLists]-->第一条， <!--[endif]-->上级输出到锁存器的输入端 d

<!--[if !supportLists]-->第二条， <!--[endif]-->锁存器输出端 q 到门控逻辑的输入单

综合工具会实现分析第一条路径，因为这个锁存器的缘故。所以会发生前级向锁存器 borrow time 的情况，出现这种情况的时候，即使第一条路径满足时序，可能的 violations 还是会产生在第二条 path 上。

这里介绍下 latch-base 设计中的 borrow time



上图是基于 latch 的设计，黑色时钟线代表没有 borrow time 的时序图，粉红和红色部分代表 borrow time。

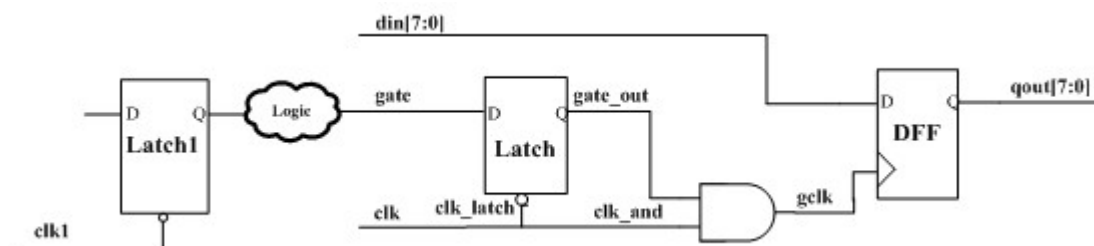
对于 latch2 进行建立关系检查。由于 logic 的 8.9ns 的延迟，latch1 输出至少要经过 8.9ns 的时间才能到达 latch2。但是根据时序图可以知道 clk1，和 clk2 的关系，当 latch2 capture 数据的时候，latch1 的数据还没有到。这样就会产生逻辑故障。

为了修正这个逻辑故障，综合工具会 borrow time，以满足建立时间关系，如上图中的红色和粉红色部分。综合工具会使 clk1 的低电平持续的时间够长，那么就要使 clk2 的低电平够端，只要满足 setup relation 就可以。

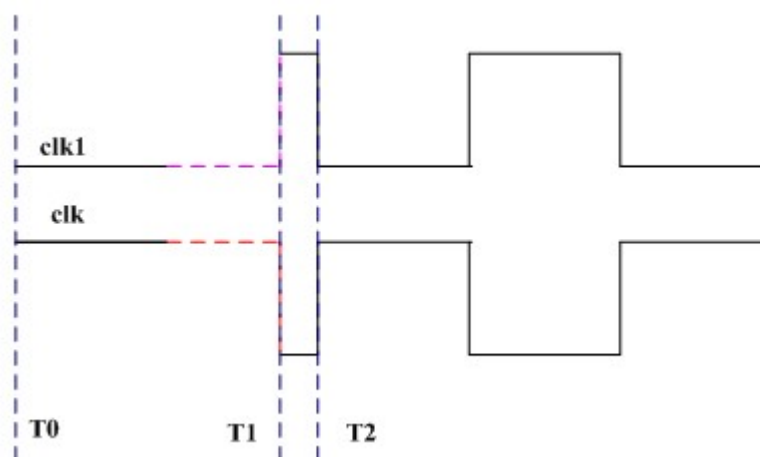
如果 latch2 还有下级，那么我们可以知道，由于上级的 borrow time，下级有可能会产生一些 time violations。

这个时候我们就要用 set_max_time_borrow 限制综合工具借入更多的时序，为了满足 setup relation，综合工具会最大优化 logic 部分。

如果将上图放入到门控电路中，如下：



结合 borrow time 的时序图，我们分析下门控时钟里面可以设置的最大 borrow time:

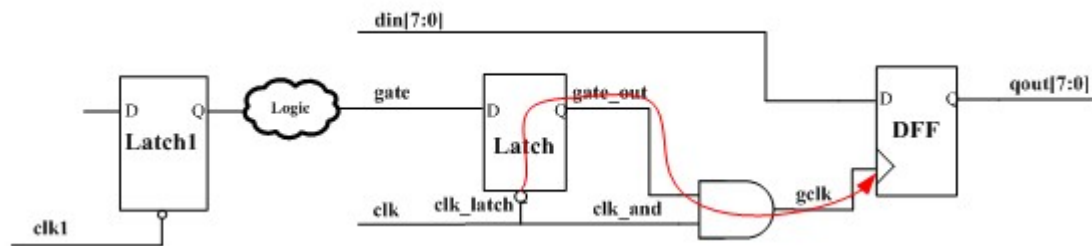


上图是 borrow time 之后时序图，由时序图我们可以知道：

必须满足的时序关系：

$$T2 - T1 > T_{cq_latch} + T_{cell_delay} + clock_uncertainty$$

如下图红色路径所示



所以 $\text{borrowTime} < T_{\text{phase}} - T_{\text{cq_latch}} - T_{\text{cell_delay}} - T_{\text{uncertainty}}$

其中 T_{phase} 为 **clk** 的低电平时间

如果综合过程中 **gclk** 上 insert buffer 或者 insert delay，这些延迟也要考虑进去，以及后端过程中可能引入的 buffer delay。

看一例：

```
module gate(clk, gate, din, qout);
```

```
    input clk, gate;
```

```
    input [7:0] din;
```

```
    output [7:0] qout;
```

```
    reg [7:0] qout;
```

```
    wire gclk;
```

```
//assign gclk = clk & gate;

always@(posedge gclk)

    qout <= din;

reg gate_out;

always@(clk)

    if(!clk)

        gate_out <= gate;

assign gclk = clk & gate_out;

endmodule
```

DC 综合脚本:

```
set lib $env(DC_LIB)

set target_library "slow.db"

set link_library "* $target_library"
```

```
set search_path ". ../src ../scripts $lib"

analyze -format verilog gate.v

elaborate gate

uniquify

link

check_design

create_clock -period 100 [get_ports clk]

set input_exp_clk [remove_from_collection [all_inputs]
[get_ports clk]]

set_input_delay 60 -clock [get_clocks clk] $input_exp_clk

set_output_delay 30 -clock [get_clocks clk] [all_outputs ]
```

```
set_clock_gating_check -setup 3 -hold 0
```

```
compile
```

综合结束后，setup 时序分析：

```
<!--[if !supportLists]-->1.    <!--[endif]-->
```

门控逻辑 AND2 的 setup 分析：

```
report_timing -delay max -nets -from gate_out
```

```

Startpoint: gate_out_reg
              (negative level-sensitive latch clocked by clk)
Endpoint: U3 (gating element for clock clk)
Path Group: clk
Path Type: max

```

Point	Fanout	Incr	Path

clock clk (fall edge)		50.00	50.00
clock network delay (ideal)		0.00	50.00
time given to startpoint		10.00	60.00
gate_out_reg/D (TLATNXL)		0.00	60.00 f
gate_out_reg/Q (TLATNXL) <-		0.24	60.24 f
gate_out (net)	1	0.00	60.24 f
U3/A (AND2X2)		0.00	60.24 f
data arrival time			60.24

clock clk (rise edge)		100.00	100.00
clock network delay (ideal)		0.00	100.00
U3/B (AND2X2)		0.00	100.00 r
clock gating setup time		-3.00	97.00
data required time			97.00

data required time			97.00
data arrival time			-60.24

slack (MET)			36.76

上图中可以看到到达时间是从 clk 的下降沿计算的，

以及要求时间是从上升沿开始，同事考虑到 clock_gating_check 中的 setup 设置

```
<!--[if !supportLists]-->1.    <!--[endif]-->s
```

etup 的 borrow time 分析

```
report_timing -delay max -nets -to gate_out_reg/D
```


Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00 r
gate (in)		0.00	60.00 r
gate (net)	1	0.00	60.00 r
gate_out_reg/D (TLATNXL)		0.00	60.00 r
data arrival time			60.00

clock clk (fall edge)		50.00	50.00
clock network delay (ideal)		0.00	50.00
gate_out_reg/GN (TLATNXL)		0.00	50.00 f
time borrowed from endpoint		10.00	60.00
data required time			60.00

data required time			60.00
data arrival time			-60.00

slack (MET)			0.00

Time Borrowing Information			

clk pulse width		50.00	
library setup time		-0.09	

max time borrow		49.91	
actual time borrow		10.00	

有 dc 综合脚本的设置，以及上图 report：

到达时间是 60，由于 clk 脉冲宽度是 50，为了保证输入端建立时间满足，需要向 endpoint borrow time，上图中可以看出，time borrowed from endpoint 为 10，使要求时间变成 60，从而 slack 为 0，满足时序要求。

现在综合脚本中加入

```
set_max_time_borrow 7 [get_clock clk]
```

从新查看 setup 的 borrow time 分析：

```
report_timing -delay max -nets -to gate_out_reg/D
```

Point	Fanout	Incr	Path

clock clk (rise edge)		0.00	0.00
clock network delay (ideal)		0.00	0.00
input external delay		60.00	60.00 r
gate (in)		0.00	60.00 r
gate (net)	1	0.00	60.00 r
gate_out_reg/D (TLATNXL)		0.00	60.00 r
data arrival time			60.00

clock clk (fall edge)		50.00	50.00
clock network delay (ideal)		0.00	50.00
gate_out_reg/GN (TLATNXL)		0.00	50.00 f
time borrowed from endpoint		7.00	57.00
data required time			57.00

data required time			57.00
data arrival time			-60.00

slack (VIOLATED)			-3.00

Time Borrowing Information			

user max_time_borrow		7.00	

max time borrow		7.00	
actual time borrow		7.00	

DC概论之IO约束

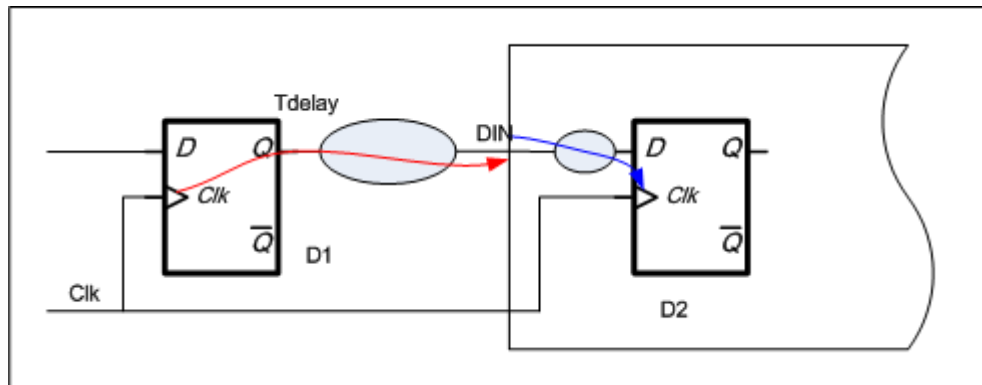
用 dc 综合过程中需要对输入输出端口进行约束。

一般的约束出发点有两种，一种是 dc，一种是 ac。

所谓的 dc 是指已知输入延迟，或者输出延迟。这个要从板机考虑。

所谓的 ac 是指输入输出延迟未知，但是知道器件输入需要的保持，建立时间的关系。

下面我们看下对于输入，根据 dc，ac 如何约束。



1.DC

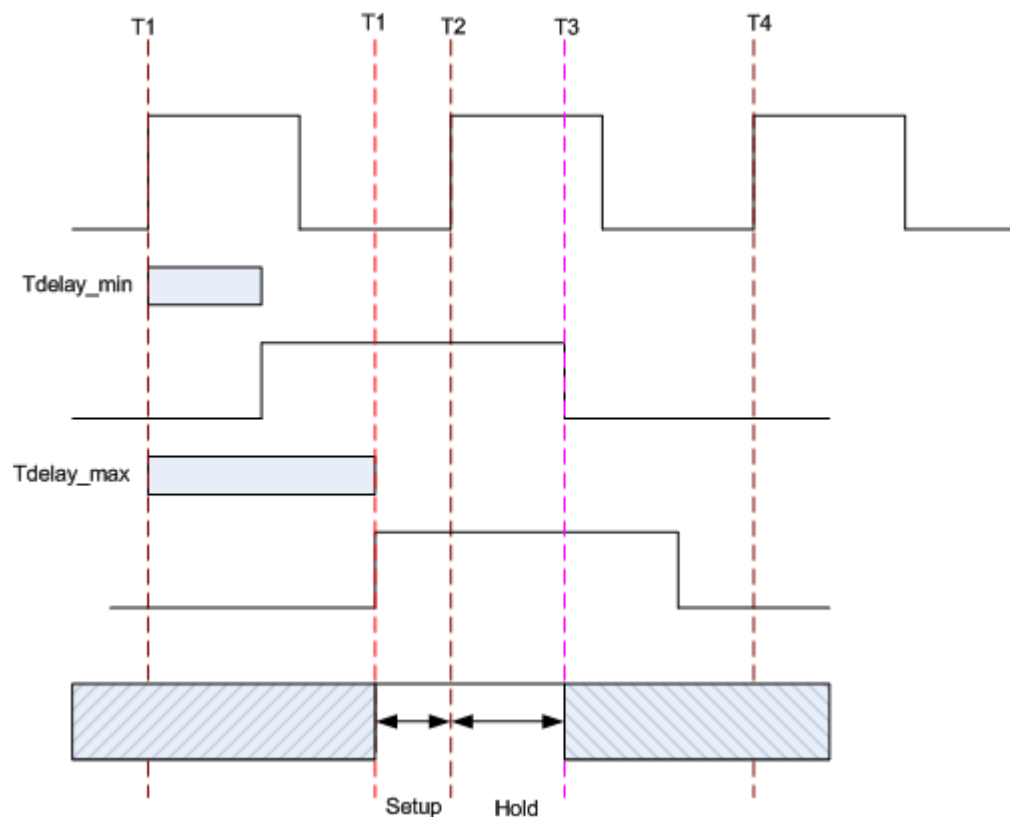
如果我们知道 Tdelay 的 max, min 那么很容易的写出约束

set_input_delay -max Tmax -clock Clk DIN

set_input_delay -min Tmin -clock Clk DIN

但是如果我们不知道输入延迟的最大最小值。那么如何约束呢。

先看下最大最小延迟对于器件的建立，保持检查起到的影响



如上图所示：Tdelay_min 对信号的影响可用从 T3 出看出。

Tdelay_max 对信号的影响可用从 T1 处看出。

所以信号的变化只能在斜线的阴影区。

2.AC

基于上图的认识，如果我们对于设计的芯片的输入信号的建立，保持有所了解。

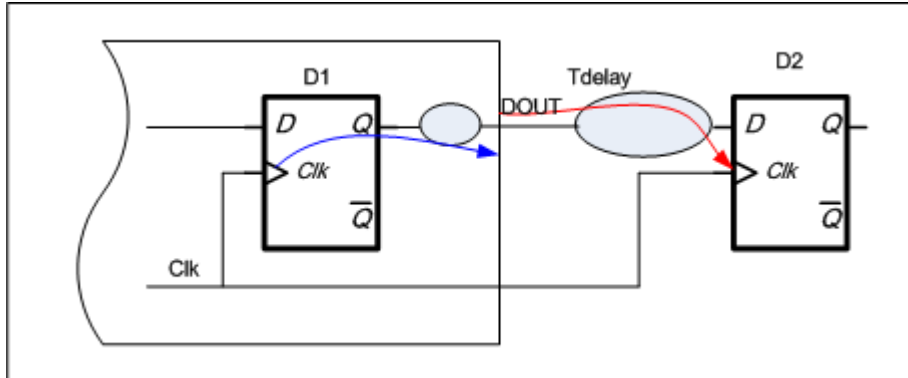
可用使用 AC 来约束,如上图，我们知道对输入信号的要求是建立要在 Setup 之前，

保持要在 Hold 之后。那么约束可用写成:

```
set_input_delay -max Tperiod-Setup -clock Clk DIN
```

```
set_input_delay -min Hold -clock Clk DIN
```

再看下对于输出，根据 dc，ac 如何约束。



1.DC

如果我们知道 pcb 板上信号延迟的最大，最小，也很容易写出约束。

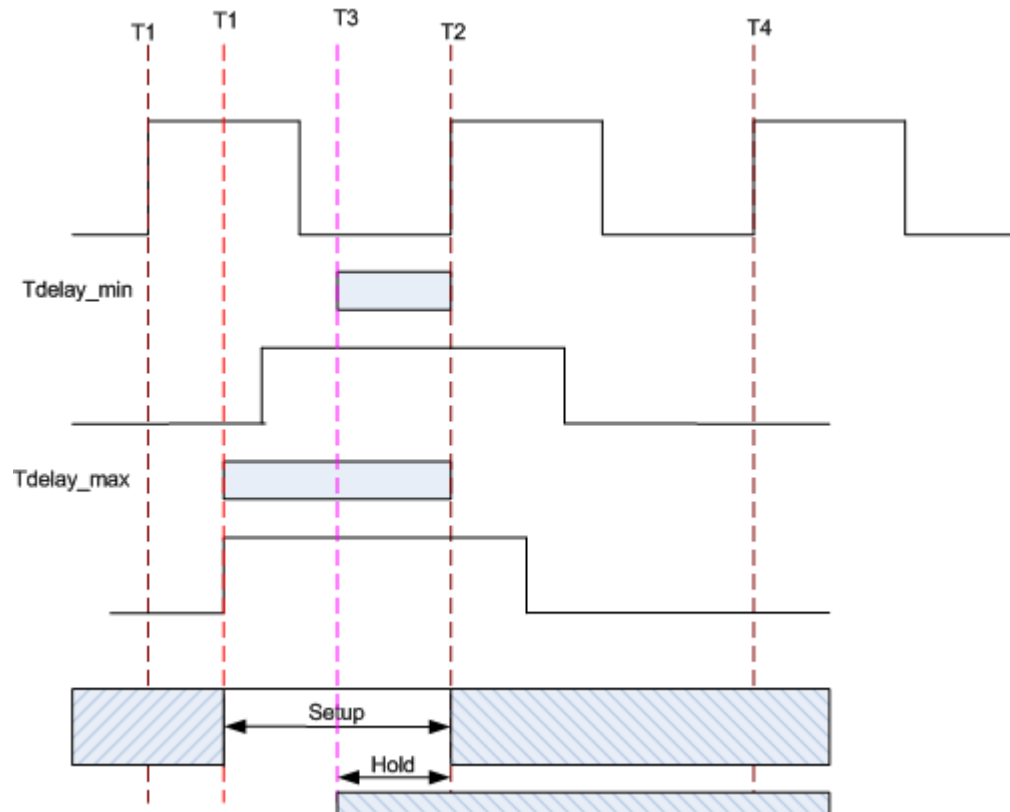
（这里的最大最小延迟仅指路径上的信号延迟，不包含器件如上图
中 D2 本身的建立时间，保持时间要求，

所以设置约束的时候要主意下）

```
set_output_delay -max Tdelay_max + Tsetup -clock Clk DOUT
```

set_output_delay -min Tdelay_min - Thold -clock Clk DOUT

下面再看下输出最大最小延迟对于建立，保持检查起到的影响



如上图所示，最大延迟影响如 T1 所示，最小延迟影响如 T3 所示，

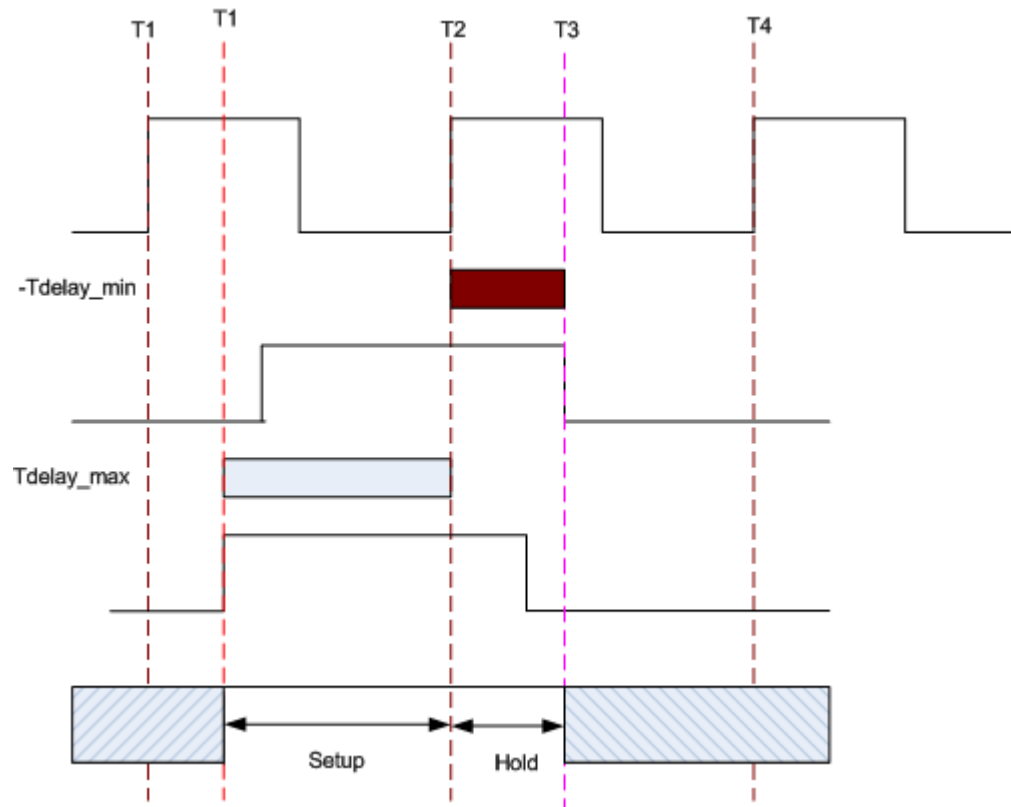
这里假设器件的 setup, hold 均为 0,可用容易理解。

这里要注意，最小延迟的影响，因为任何延迟都会满足要求。最小延迟影响，要求信号在细条阴影区变化，但是考虑到 launch 边沿在 T2, 所以信号的允许变化范围如阴影区显示(大条)。

2.AC

但是通常设计的 chip 外接可能还是 chip 或者其他器件，这些器件都有建立,保持要求。

所以如果上图的时序改成下面这种基于外围器件的建立，保持要求的。则更为理解。



约束也就相应的改成：

```
set_output_delay -max Setup -clock Clk DOUT
```

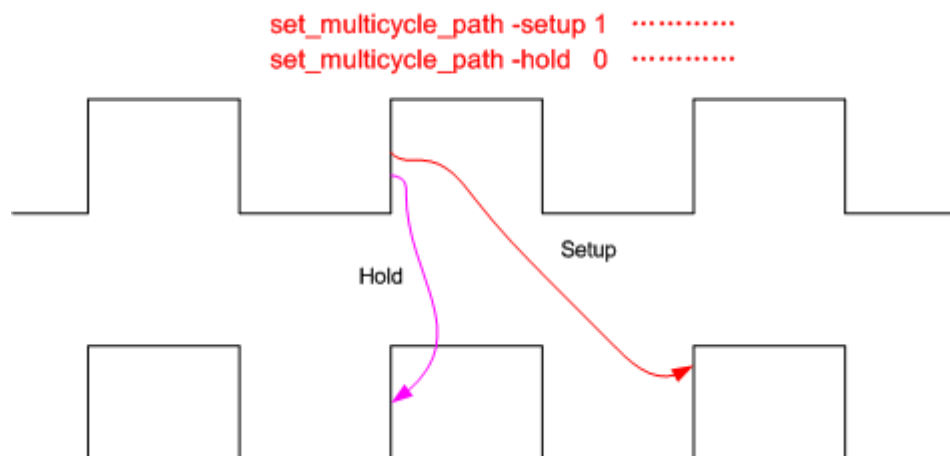
```
set_output_delay -min -Hold -clock Clk DOUT
```

如果注意观察就会发现上如对于信号的要求都是要求信号在相关时钟沿附件保持稳定。

有的时候我们要求信号在相关时钟沿附近是需要变化的。

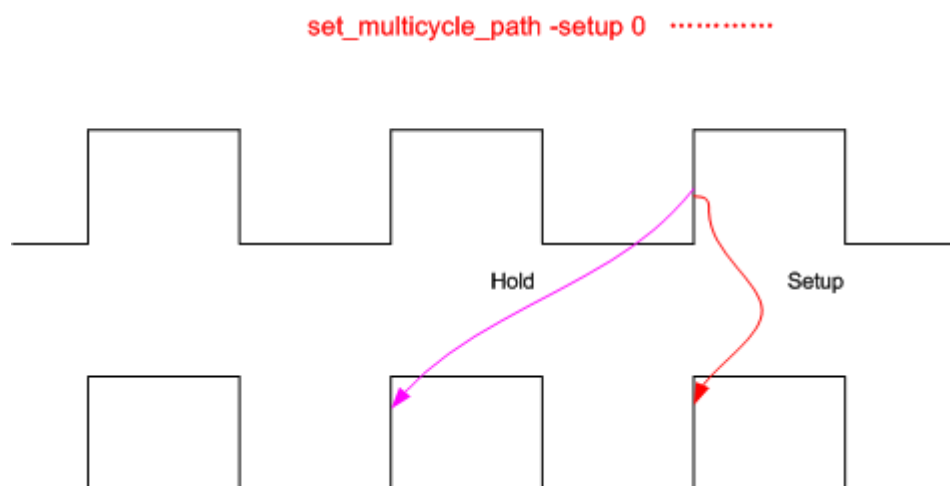
这里要用到零多周期路径。

下面看下对于一般的单周期路径，如何写成多周期路径

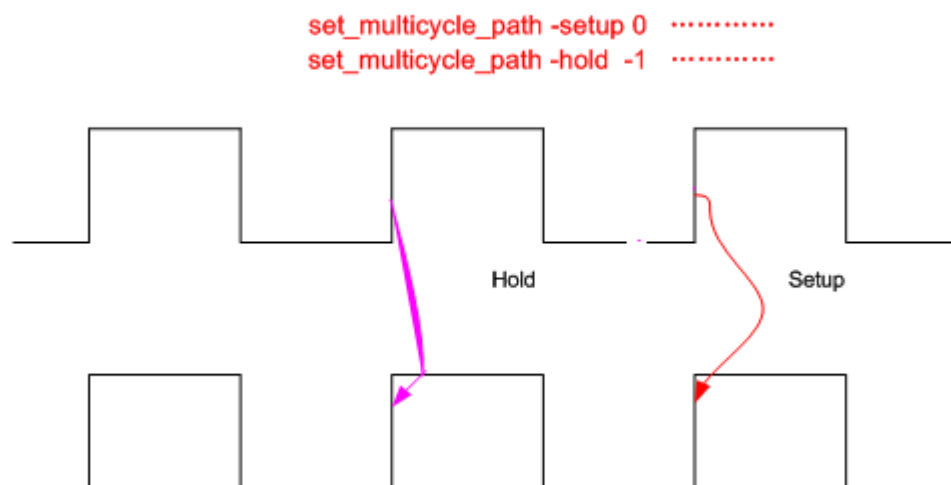


那么如果对于零多周期路径，建立，保持如何检查：

当设置 `set_multicycle_path -setup 0` 的检查如下：

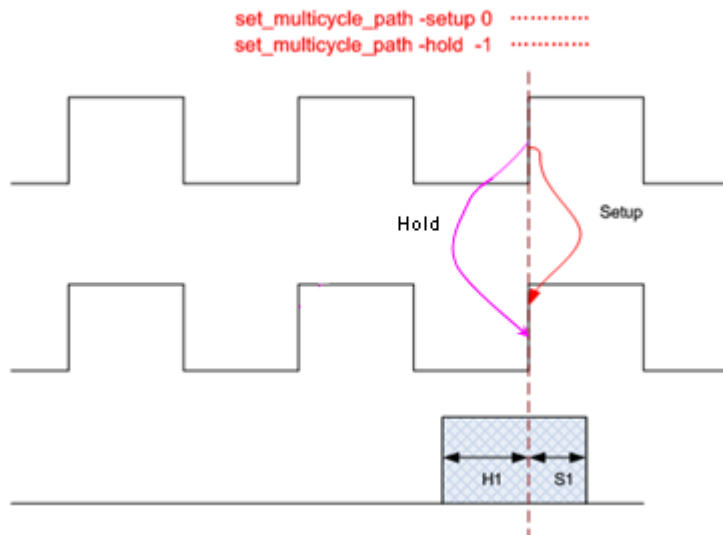


当再次设置 `set_multicycle_path -hold -1` ..时序检查如下：



再如下图，我们要求信号在 S1,H1 区域内变化。

如何约束呢。



首先是设置 0 多周期路径。

```
set_multicycle_path -setup 0 .....
```

```
set_multicycle_path -hold -1 .....
```

然后根据要求设置最大最小延迟。

有了上面的认识，我们大约可用了解：

如果设置了最大延迟，那么信号只能在最大延迟前变化，

如果设置了最小延迟，那么信号只能在最小延迟后变化。

所以我们设置成：

```
set_output_delay -max -S1 .....
```

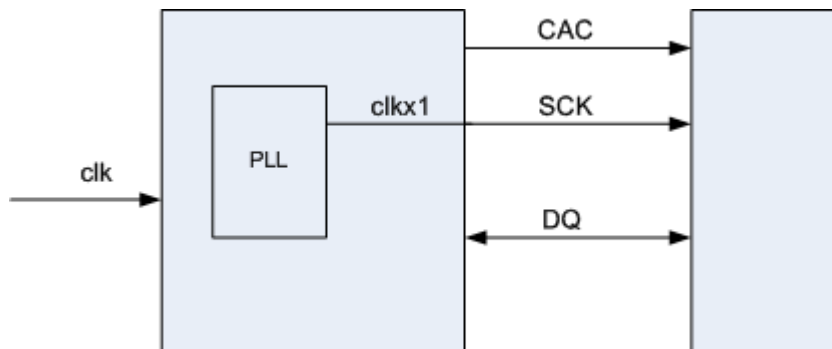
```
set_output_delay -min H1 .....
```

注意符号：)

接着我们来考虑下源同步接口，所谓源同步接口就是输入输出带有
时钟线。

同时数据的变化和时钟沿对齐（或上升沿，或下降沿，或双沿）。

如下图，单边沿对齐的源同步接口。数据 DQ 和时钟 SCK 对齐,CAC
地址控制信号



当然我们不知道 pcb 走线延迟，但是我们知道链接的器件的建立，
保持要求分别为 T_{h_ext} , T_{s_ext}

我们自己 chip 的建立保持要求为 T_h, T_s

我们可用约束如下：

```
create_generated_clock -name clkx1 -multiply_by 1 -source clk
PLL/clkx1
```

```
create_generated_clock -name SCK -divide_by 1 -source PLL/clkx1
SCK
```

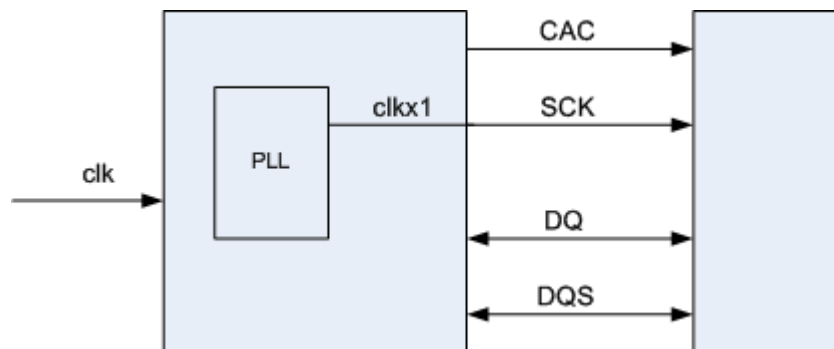
```
set_input_delay -max T-Ts -clock clkx1 DQ
```

```
set_input_delay -min Th -clock clkx1 DQ
```

```
set_output_delay -max Ts_ext -clock SCK {DQ CAC}
```

```
set_output_delay -min -Th_ext -clock SCK {DQ CAC}
```

如果数据和时钟是双边沿对齐的,建立，保持条件如上，同时下降沿和上升沿要求一样。



```
create_generated_clock -name clkx1 -multiply_by 1 -source clk
PLL/clkx1
```

```
create_generated_clock -name SCK -divide_by 1 -source PLL/clkx1
SCK
```

```
set_output_delay -max Ts_ext -clock SCK CAC
```

```
set_output_delay -min -Th_ext -clock SCK CAC
```

```
set_input_delay -max T-Ts -clock clkx1 {DQ DQS}
```

```
set_input_delay -max T-Ts -add_delay -clock clkx1 -clock_fall clkx1
{DQ DQS}
```

```
set_input_delay -min Th -clock clkx1 {DQ DQS}
```

```
set_input_delay -min Th -add_delay -clock clkx1 -clock_fall clkx1 {DQ
DQS}
```

```
set_output_delay -max Ts_ext -clock SCK {DQ DQS}
```

```
set_output_delay -max Ts_ext -add_delay -clock SCK -clock_fall {DQ
DQS}
```

```
set_output_delay -min -Th_ext -clock SCK {DQ DQS}
```

```
set_output_delay -min -Th_ext -add_delay -clock SCK -clock_fall {DQ
DQS}
```

DC 优化约束

时间约束和面积约束、功耗约束

```
set_max_area
```

```
set_input_delay
```

```
set_output_delay
```

```
create_clock
```

```
set_clock_latency
```

```
set_propagated_clock
```

```
set_clock_uncertainty
```

1.定义时钟，周期和波形

```
create_clock [ - period period_value] [ - name clock_name] [-
waveform. edge_list] [source_list]
```

例如，周期为 20，占空比为 40%(上升沿在 0 时刻，下降沿在 8 时

刻), 的时钟, 输入的管教名称为 CLK

```
dc_shell>create_clock - period 20 - waveform. {0 8} CLK
```

默认为 50%占空比

关于公共基准周期的问题, 查询书。10, 20, 30, 公共基准就是 60. 尽量设置使得公共基准周期小。否则, 运算复杂。

设置时钟不确定性绕开这个问题, 如下: 10,10.1,公共基准周期为 1010.0

```
>create_clock -period 10 clk1
```

```
>create_clock -period 10 clk2
```

```
>set_clock_uncertainty -setup 0.1 clk2
```

两个时钟都没有小数部分, 计算时间也不需要扩展其周期, 由于对 clk2 设置了建立时间的不确定性, 最终结果仍然考虑 0.1 的时间。

用 set_clock_latency 和 set_clock_uncertainty 定义时钟网络的时延信息。

2.确定 I/O 端口相当于以时钟的时间约束

相当于给输入输出信号信号相对于时钟的时间要求。

对输入: 定义了信号到达模块时已经经过的延时。

对输出: 限定了信号必须在哪个时间前有效。

据此, DC 可以计算出这个模块内部各个路径的时延要求, 根据这些信息进行优化。

默认, 都为 0, 与实际相悖。

例如: 定义的是时间约束相对于时钟上升沿的关系

```
>set_input_delay 20 -clock CLK DATA_IN //DATA_IN 为输入端口
```

```
>set_output_delay 15 -clock CLK DATA_OUT
```

3.组合时延

无法与时钟对应的情况

对复位信号 `rst` 设置了最大的时延 5

```
>set_max_delay 5 -from rst
```

```
>set_min_delay
```

4.确定例外情况

伪路径:

无时间要求或无实际操作的路径

```
dc_shell>set_false_path -from U1/G -to U1/D
```

最小时延要求和最大时延要求:

```
set_min_delay set_max_delay
```

多周期路径:

```
set_multicycle_path
```

结合 `-setup` 和 `-hold` 命令可以将建立时间和保持时间检查从缺省的位置向后移动到实际的位置。

例如: 定义一个从 A 到 B 的路径伪一个两周期的路径

```
>set_multicycle_path 2 -from A -to B
```

面积约束:

```
>set_max_area 100 //单位与工艺库中的一致
```

