Clock Domain Crossing(CDC)筆記-知乎老李版本

CDC（Clock Domain Crossing）：指某個信號在 clock domain A（稱為發送端）下的暫存器的輸出，經過某條路徑，抵達了 clock domain B（稱為接收端）下的暫存器的輸入，則視為發生了CDC (clock domain crossing)。

1.0 跟老李一起學習晶片設計-- CDC 的那些事（1）

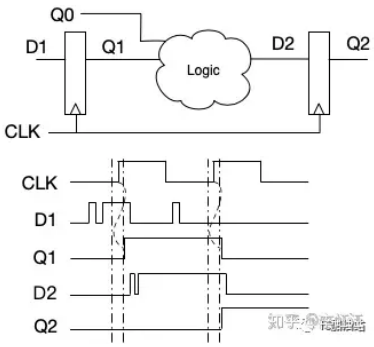
一轉眼，老李在數位晶片設計領域又摸爬滾打了四年。上篇推送已經四年前了，久到我差點忘記了還開過這個公眾號。前兩天偶然想起，便覺得還是要在這個平臺輸出一些內容。這些年在工作中也算是積累了一些經驗，其中有些經驗值得記錄下來，和大家做一個交流。

從這篇開始，老李推送一系列晶片設計中跨時鐘域（CDC）的知識。跨時鐘域設計幾乎是現代數位晶片設計中所有設計人員必須要掌握的基本知識，但是從老李自身的經歷來說，這方面的知識在本科和研究生的課程當中都沒有講過，卻是面試中經常被考到的知識點。同時工作中工程師也必不可少地要和 CDC 打交道，考慮如何進行跨時鐘域設計，以及掌握 CDC 相關的 EDA 工具。以下文章我們就從跨時鐘域設計的最基本開始講起，在之後的推送中，老李還會依次分享跨時鐘域的進階內容。

這裡我們先複習一下同步電路和非同步電路的概念。在現代 SoC 設計中，絕大多數的電路都是同步電路。同步電路是由時鐘驅動記憶元件的電路，也就是說記憶元件的狀態只在時鐘沿到來的時候才能發生變化。因為組合邏輯電路在輸入變化時輸出可能出現毛刺(glitch)，而記憶元件因為只會在時鐘沿到來時才會更新狀態，那麼在時鐘沿之間的時間狀態是穩定的，這樣的同步電路可以消除組合電路中的毛刺，如下圖所示。

NOTE：毛刺(glitch)是短暫的非預期訊號變化。

NOTE：毛刺(glitch)如果造成資料有誤，稱為危障(hazard)，解決方式有增加卡諾圖的共通項。https://www.cnblogs.com/iczero/p/17187349.html

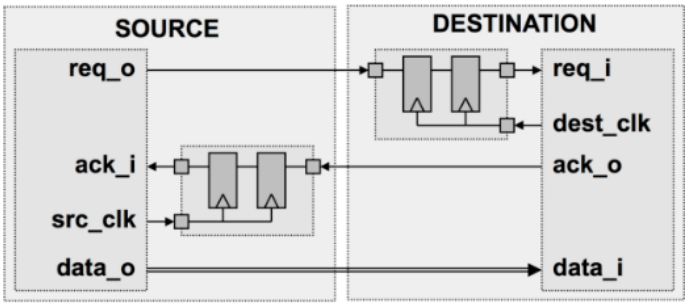


相應的，時鐘週期的大小取決於最長的傳輸延時(propagation delay)。同步電路的好處是時序很清晰，電路中的記憶元件例如觸發器都是依照一個固定的節拍來工作，便於 EDA 工

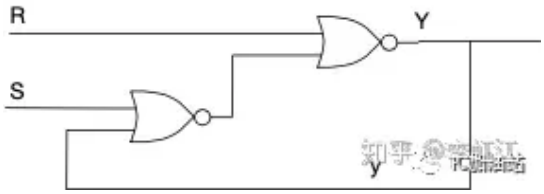
具來進行延時的分析和計算，所以同步電路幾乎佔據了當前數位晶片的絕大多數部分。

而非同步電路就沒有時鐘的概念了，記憶元件所存的狀態跟隨了輸入信號的變化立刻發生變化。信號之間的傳遞通常通過握手(handshake)來完成，因為沒有時鐘的約束，每一級記憶元件之間的邏輯電路都是各自獨立的，可以各自進行優化，這樣可以達到很好的性能。但是這既是優勢，也是劣勢。劣勢就在於 EDA 工具沒有滿足每一級都單獨優化的計算能力，而且由於相鄰的級之間互相影響，使得計算總的時序時變得異常複雜，所以非同步電路的規模通常無法做大，進而也限制了它的用途。目前更多的也只是在學術研究方面，並沒有成為當今 SoC 設計的主流。下面這個例子實現了一個狀態存儲單元，next state Y 會依據當前狀態以及 R,S 的值立刻發生變化，顯然這樣的狀態變化更加快速，但是分析起來也更加複雜。

NOTE：hand shake(要互相溝通，增加資料傳遞的時間。Latency 太大，BAD！)



當資料準備好的時候，利用 req_o 同步到對面的 clk domain B，這時 clk domain B 就知道 A 已經將資料準備好了，並且這時候 A 的 data_o 會完全靜止不動，等到 B 已經取樣資料完畢，利用 ack_i 告訴 clock domain A 資料已經取樣完畢。



我們接下來的跨時鐘域分析，當然都是基於同步電路的。同步電路的核心就是觸發器，觸發器的種類有很多種，最常用的就是 D 觸發器。在這裡我們還是首先複習一下基本概念: 建立時間 setup time 和保持時間 hold time, 以及亞穩態 metastability。

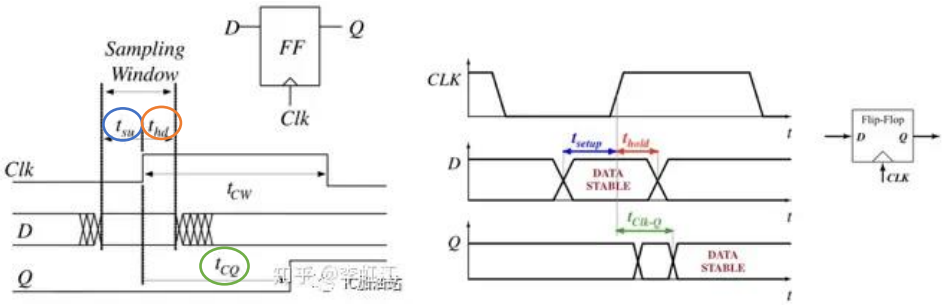
setup time: 時鐘沿到來之前輸入信號 D 必須保持穩定的最小時間

hold time: 時鐘沿到來之後輸入信號 D 必須保持穩定的最小時間

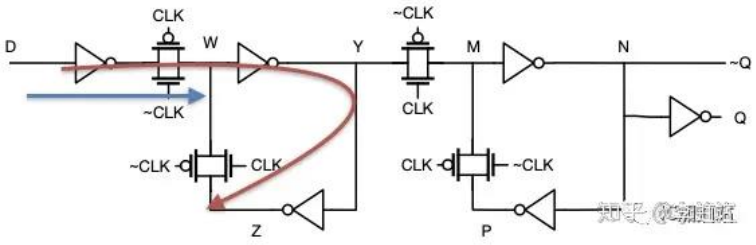
clk-to-q time: 輸入 D 滿足 setup/hold time 要求，從時鐘沿到來時刻到輸出端 Q 變化至穩定的時間

那麼當輸入信號 D 無法滿足 setup time 或者 hold time 的要求，我們稱之為產生了 setup

time / hold time violation, Flip Flop Q 的輸出這個時候是 0 還是 1 是不確定的，需要一定的時間才能夠穩定在 0 或者 1。所以如果當 Q 端在 clk-to-q 時間之後才變得穩定的話，我們就說這個觸發器產生了亞穩態 metastability。



很多工程師在面試的時候都可以回答得上 setup time 和 hold time 的定義，但是回答不上**為什麼 D 觸發器有 setup time 和 hold time 的要求**。這個問題其實大家在學校裡學過，如果你在面試的時候被問到卻不知道，那麼你應該回去好好複習一下基礎知識。它們與 D 觸發器的內部結構有關係。D 觸發器的內部是一個主從鎖存器(master-slave latch)，一個常見的 D 觸發器結構如下圖所示



67/

Latch 能夠存儲住狀態，靠的是上面的背靠背的反相器。而這個背靠背的反相器能夠鎖住狀態是需要時間的。由此，我們可以分析出

setup time: 在 clk 的上升沿到來之前，D 要傳輸到 Z 的時間。因為當 Z 的值還沒有穩定的時候，D 如果變化，那麼這個背靠背的反相器就無法鎖住值。

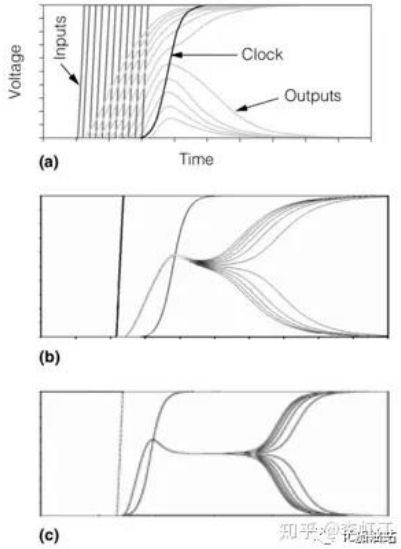
hold time: 第一個傳輸門關閉需要的時間，在傳輸門關閉期間，D->W 要保持穩定，這樣在傳輸門關閉之後，W 穩定才不會導致背靠背反相器鎖住的值發生變化。

所以我們可以看出，當 D 在 setup/hold time window 內發生變化，鎖存器可能無法鎖住一個穩定的值，會發生的結果是

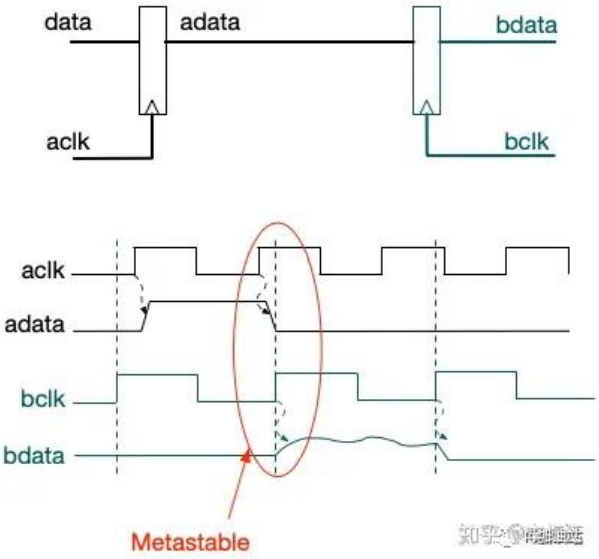
- Q 的值可能不是正確的 D
- 隨著 D 的變化越靠近時鐘沿，Q 變穩定的時間越長
- 最後 Q 穩定到的值可能是隨機的

注意我們並不是說 Q 最後的值不是穩定的 1 或者 0， Q 的值最後一定會穩定下來，穩定在高電平或者低電平，這是由於背靠背的反相器會產生正回饋，最終一定會穩定下來。但

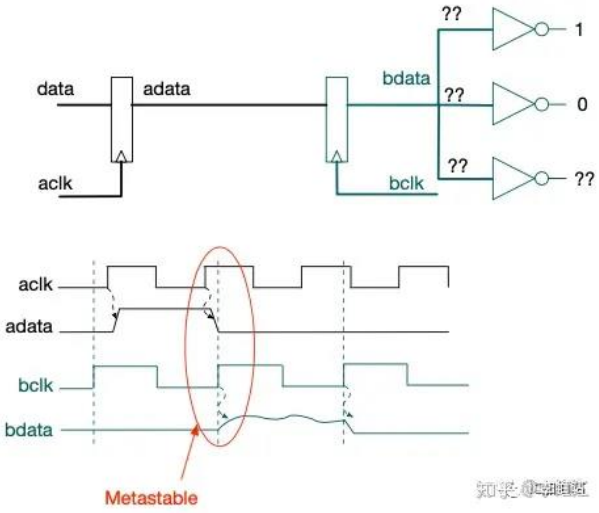
是當這個穩定的時間超出了 clk-to-q 的限制，我們就說產生了亞穩態。



接下來我們就可以正式進入跨時鐘域的討論了。當只有一個時鐘存在時，life is simple，只要保證 setup/hold time 就好了，那麼**當有多個時鐘存在的時候會發生什麼呢？**我們來看最簡單的情況，如下圖所示，aclk 時鐘域的信號需要傳輸到 bclk 時鐘域去。在各自時鐘域內，EDA 工具可以保證觸發器不會產生 metastable，但是當 aclk 和 bclk 非同步的時候，我們是無法保證 aclk 和 bclk 之間的關係的，也就是說 adata 相對於 bclk 的沿來說，可能在任何時候發生變化，這樣 bdata 這個 flip flop 就可能產生亞穩態。



當 bdata 發生亞穩態的時候，會造成什麼影響呢？影響主要發生在 bclk 時鐘域的後級電路上，讓後級電路無法 sample 到一個確定的正確的值，進而導致運算邏輯錯誤。



在這裡要澄清的一點是，亞穩態的出現並導致邏輯錯誤並且晶片失效是一個**概率事件**，而不是一個 100%會發生的確定性事件。這一點可能有點難以理解，舉例來說明，很有可能 bdata 這個 flip flop 的後面組合邏輯的 delay 很小，而這個 flip flop 在發生亞穩態之後所需要穩定的時間也很短，這樣即使 flip flop 發生了亞穩態，而後級的 flip flop 的 setup time/hold time 也可能可以得到滿足，這樣的話在實際晶片工作中，我們可能觀察不到產生錯誤輸出的情況。但也正是這樣的原因，很多看似正常工作的晶片內部可能其實有跨時鐘域設計上的問題，卻從來沒有暴露出來。這種情況其實非常危險，因為這種問題一旦出現，則會非常難以 debug，因為出現的概率很低，看起來很隨機。或者很可能同樣的設計換一個工藝，以前可以工作(但)在新的工藝上突然產生問題，造成很嚴重的後果。所以晶片在設計的時候需要儘量在流片前發現並解決所有的 CDC 問題，這一點大家要銘記在心。

晶片一旦出現 CDC 的問題，可能會導致以下後果

- 邏輯功能發生錯誤，比如控制信號，握手信號錯位
- 資料發生錯誤，比如 data bus 的值，memory 中存的值發生錯誤
- 發生錯誤的時間可能隨機，很難以複現相同的錯誤
- 很難以通過軟體修復，即使能夠修復，也可能需要犧牲性能和功耗

那麼我們能夠完全消除亞穩態嗎？答案是否定的。其實我們關心的並不是亞穩態，而是說**能否避免由於亞穩態而造成的邏輯問題**。在這裡要引入一個 **MTBF** 的概念。MTBF--mean time between failure. 意思是兩次失效之間的平均時間。簡單來說，就是這個晶片或者這個 IP 或者這個電路發生兩次發生錯誤之間的間隔。對於不同的系統和應用場景，MTBF 的要求也不同。比如說對於我們的手機，沒有人拿一個手機用二三十年吧？那麼如果能夠保證 MTBF 大於 30 年，那麼也等效於在整個手機的使用壽命中，這個邏輯錯誤不會發生 2 次，那麼針對這個錯誤來說，這樣的 MTBF 是可以接受的。但是對於有些應用場景，比如通訊衛星，一個通訊衛星的壽命可能超過二三十年，那麼這種情況下 MTBF 如果只有 30 年，那麼就無法接受了。

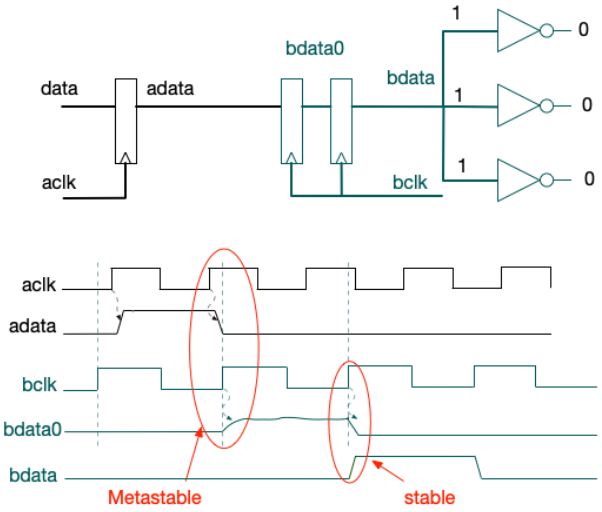
關於 MTBF，還有一個誤區需要澄清，就是 MTBF 只要大於產品的設計使用壽命就可以了，這其實是不嚴謹的。因為一個產品可能由多個系統組成，每個系統又是由多個子系統組成，每個子系統可能細分下去是由更小的單元組成。整個產品不發生失效的概率是所有部分不發生失效概率的乘積。所以越是小的單元，越要保證 MTBF 越高，這樣才能不會導致整個產品的 MTBF 有顯著下降。

說回來觸發器的 MTBF，MTBF 的具體公式將在下一篇推送中呈現。這裡大家只需要知道，MTBF 反比於採樣時鐘頻率(destination clock frequency)，反比於資料變化頻率(source data change frequency)，還和工藝、電壓、溫度等因素相關即可。

2.0 你真的懂 2- flip flop synchronizer 嗎-- CDC 的那些事（2）

上一篇中我們回顧了一些基礎知識，其中最重要的概念就是亞穩態。我們接下來所要看到的各種 CDC 的設計方法，本質上都是圍繞在如何解決亞穩態帶來的問題。

我們首先來看最基本的問題，**single bit level 信號的跨時鐘域**。single bit 直接被 destination domain 的 flip flop 去 sample 產生的問題我們在上一篇已經討論過，那麼解決的辦法呢？看起來很簡單 -- 之後再加一個 flip flop，也就是說用兩級的 flip flop 來同步 source domain 的 signal。我們通常把這種 synchronizer 叫做 2 flip flop synchronizer 或者 double flip flop synchronizer，俗稱“打兩拍”。



說實話，老李在最開始學習到這個辦法的時候，內心的聲音是：“這 TM 在逗我？這麼簡單就可以了嗎？憑什麼第二級的輸出就沒有亞穩態了？” 相信有很多初學者也和我當初有同樣的困惑。在這裡我們要再次回顧一下 metastable 產生的原因。第一級 flip flop 產生 metastable 的原因是 flip flop 裡面沒有及時鎖住該鎖的值，所以我們無法直接使用第一級 flip flop 的 Q 來直接用於 bclk 時鐘域。但是要注意，我們之前說過，第一級 flip flop 的 Q 會最終穩定下來的，而且在絕大多數時候，可以在一個 bclk 週期內穩定下來，這樣第二級 flip flop 的 D 輸入就是一個穩定的值，進而第二級 flip flop 的 Q 是滿足 clk-to-q 的，沒有亞穩態的產生。如上圖所示，儘管 bdata0 產生了 metastable，但是 bdata 是 stable 的。

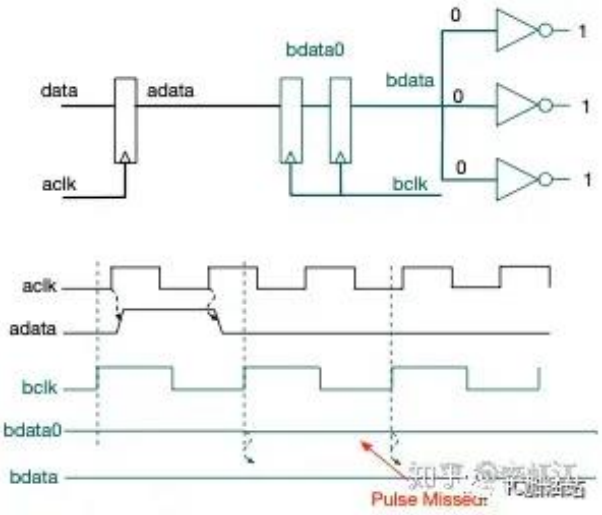
那你可能會問，上一講你不是說第一級的 flip flop 的輸出可能需要很長時間才能穩定嗎？你憑什麼說在第二個時鐘沿到來的時候 bdata0 就能穩定呢？如果還是不穩定，讓第二級 flip flop 產生了 setup/hold time violation，那第二級 flip flop 不還是有可能產生 metastable 麼？

這是一個非常正確的問題，的確，double flip flop synchronizer 不能完全消除亞穩態！但是很多有經驗的工程師會告訴你，用個 double flip flop synchronizer 就夠了，那是因為 double flip flop 會使得 metastable 產生的概率顯著降低，這就又回到了我們上一講的 MTBF 的概念。在使用 double flip flop 的時候，由於給了第一級 flip flop 一個週期的時間去穩定，使得兩級發生 metastable 的概率大大降低。我知道大家都不喜歡看公式，下面給了一個 100MHz 時鐘的例子，大家只需要關心一下最終的結果，是 957 億年，而地球的年齡是 46 億年，太陽的壽命是 100 億年，也就是說，直到太陽毀滅，你也碰不到下一次 metastable 的產生。

$$MTBF(t_r) = \frac{e^{t_r/\tau}}{T_0 \cdot f_{clk} \cdot f_{data}} \cdot \frac{e^{t_r/\tau}}{T_0 \cdot f_{clk}} = 9.57 \times 10^{10} \text{years}$$

但是注意，隨著 sample clock frequency 的提升，以及工藝節點越來越小，有些時候打兩拍已經不太夠了，那麼就簡單粗暴來一個打三拍，就能夠保證了。而老李目前還沒有看到過有誰在設計中實現打四拍的，如果你見過，請聯繫我，老李更願意和他聊聊地球毀滅和人類未來的話題。

想當初，老李剛剛學到一點 double flip flop synchronizer，以為 CDC 就不過如此，可以仰天大笑出門去也。但是之後的一個 fail test 就把當時還是小李的我給搞懵了。打開波形一看，搞什麼鬼，信號沒有同步過來啊？

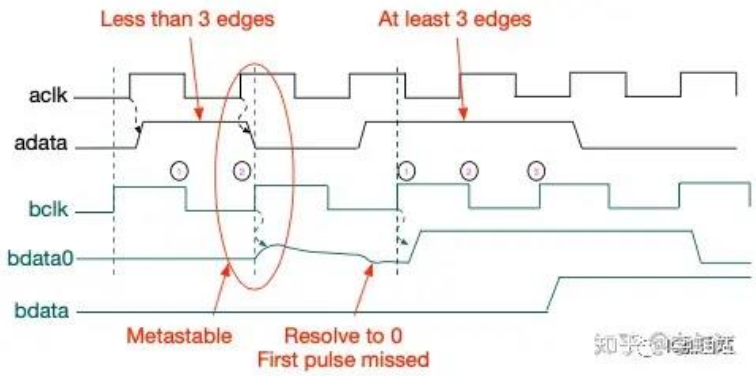


不是說好了單 bit 信號用 double flip flop 麼，肯定是哪裡不對？老李仔細一看，這 adata 怎麼在 bclk 的沿到來之前又變了呢，這樣 bdata0 這個 flip flop 壓根看不到 adata 的變化啊，看來 double flip flop 來做 synchronizer 是有條件的，不是隨便什麼單 bit 信號都可以直接無腦用！

其實使用 double flip flop 來同步，有個最基本的“3 個沿”要求，就是 source data 必須保證穩定不變，至少碰見 destination clock 3 個連續的沿，這個沿可以是上升沿也可以是下降沿，持續 3 個沿之後才能變，否則就有可能在 destination clock domain 根本看不到這個 data 的變化。例

如下圖所示：adata 第一次變高，只碰到了 bclk 的 2 個沿，就可能導致 bdata 根本沒有看到這個 pulse，而第二次 adata 變高，持續了 3 個沿，這樣 bdata 就能夠確保也可以變高了。

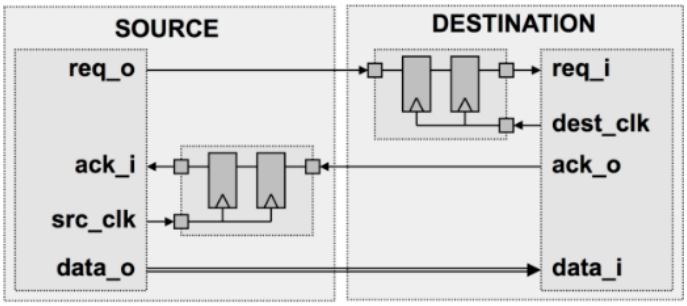
NOTE：a_data 必須保證穩定不變，至少碰見 b_clk 的 3 個連續的沿，這個沿可以是上升沿也可以是下降沿，持續 3 個沿之後才能變，否則就有可能在 destination clock domain 根本看不到這個 data 的變化

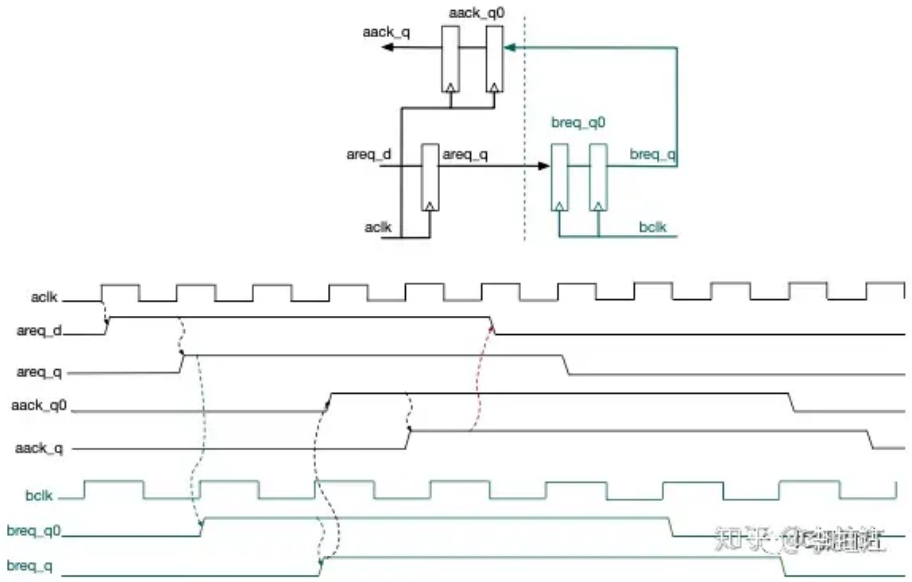


所以如果 bclk 的頻率是 1.5 倍的 aclk 頻率以上，即使 adata 是 aclk 域的一個短 pulse，也可以保證 3edge 要求。如果沒有這樣頻率的關係，那就得對 adata 有要求了，adata 的變化不能很迅速，要穩定足夠長的時間，這樣才不會讓 bclk 域錯過值，具體怎麼做呢？在回答這個問題之前，我們應該先停一下，問自己一個問題：我是不是要將 adata 的每一次翻轉都同步到 bclk 呢？

其實這個問題應該是在考慮要將一個信號跨到另一個時鐘域的時候首先要問自己的問題。大多數情況下，回答都是肯定的，也就是說 adata 變化了，bclk 這邊的信號也要變化。但是也有些時候，即使漏了 adata 的變化，可能也沒關係，這就是和設計的要求相關了.

如果必須要求 adata 的每一次翻轉都同步到 bclk，一個辦法就是利用**回饋(Hand Shake)**。也就是說信號從 aclk 域同步到 bclk 域，再同步回 aclk 域。aclk 的 data 只有看到同步回來的值之後才能再翻轉。如下圖所示(類似第一張)





但是這樣做的缺點也很明顯，就是將 aclk 的 data 進行了擴展，兩次的同步也增加了延時，這是為了達到每次變化都同步而付出的代價。大多數時候，設計者知道 adata 變化的頻率很低，比如是一個軟體配置位元，配置好之後可能不輕易更改；比如是一個中斷信號，中斷發生之後可能需要很長時間才會被軟體清除，這些時候就沒有必要設計回饋電路了。

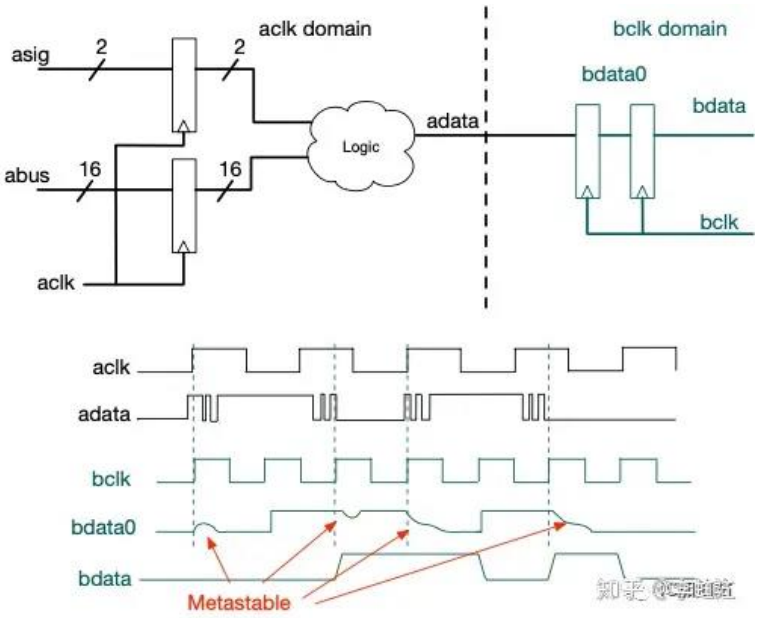
最後再講兩個知識點，也是非常關鍵的知識點。

第一，利用 double flip flop，bdata 發生變化可能是在 adata 翻轉之後 1 個週期，也可能是 2 個週期，這是由於第一級 flip flop 的 metastable 可能會 resolve 在不同的值。如果第一級 flip flop 穩定在和 adata 相同的值，那麼就只需要 1 個週期就能看到 bdata 翻轉。而如果第一級 flip flop 穩定在和 adata 相反的值，那麼則需要再多一個週期。所以在設計和模擬驗證中，不能假定 bdata 一定會在 2 個週期之後發生變化，而是將這個因素隨機在模擬中，有的時候真的會暴露出設計中的問題。

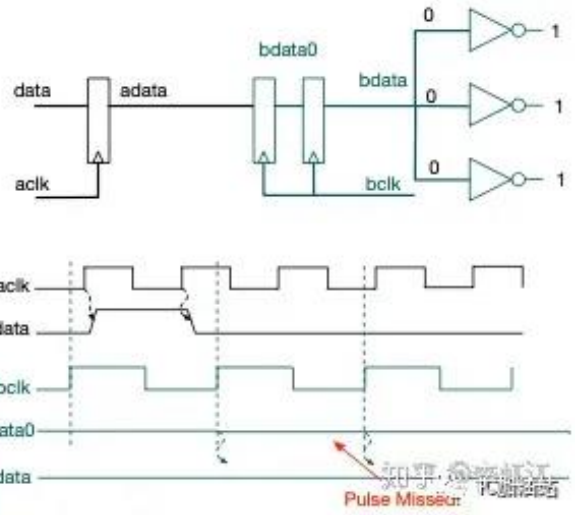
NOTE：打兩極只會降低發生機率，不一定會完全解決 metastable 問題。

第二，我們說的單 bit 信號，有人可能會說，組合邏輯的輸出可不可以用 double flip flop 呢？比如一個 AND 閘的輸出，不也是單 bit 嗎？答案很簡單，不可以。原因就是組合邏輯的輸出可能會有毛刺，這些毛刺會增大第一級 flip flop 產生 metastable 的概率，進而影響整個 synchronizer 的 MTBF。所以對於任何單 bit 信號，在跨時鐘域之前一定要先寄存（flip flop)，只有 flip flop 的輸出才能經過 synchronizer. 下面的圖就是不 flip flop 的情形。

NOTE：如果要做 CDC，盡量讓輸出使用 reg out。原因就是組合邏輯的輸出可能會有毛刺，這些毛刺會增大第一級 flip flop 產生 metastable 的概率，進而影響整個 synchronizer 的 MTBF。



結尾之前，要重新再討論一下當初讓小李搞懵的那個例子。

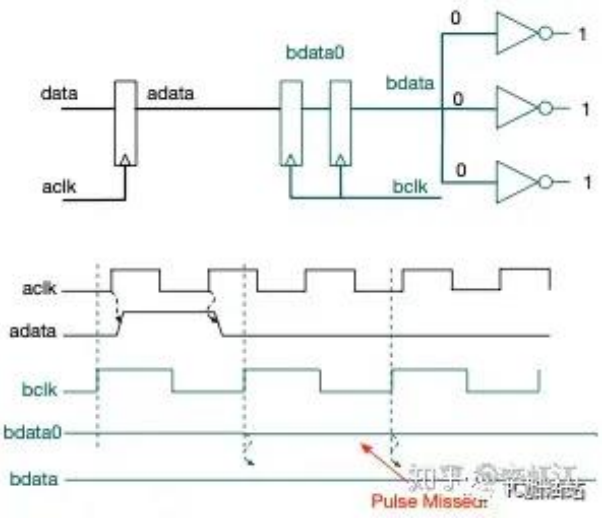


其實在這個例子中，adata 在 aclk 域一個週期就翻轉了，其實是 aclk 域的一個 pulse 信號。對於 pulse 信號，我們其實不應該用 double flip flop 來同步，原因很簡單，就是上面所示的可能丟失 pulse。那麼面對 pulse，我們該怎麼同步呢，那就且聽下回分解吧。

3.0 常見數電面試題 Pulse Synchronizer -- CDC 的那些事（3）

上一期老李挖了個坑，是關於**同步一個時鐘域的單週期脈衝(pulse)**的問題。想當年這個問題老李在面試某幾家大廠的時候被問到過不止一次，足以見得這是一個常考的知識點。在這篇裡，老李帶領大家破解這道常考面試題，讓你在面試時能夠遊刃有餘。

pulse 信號在設計當中很常見，通常在某個時鐘沿變高，在下一個時鐘沿變低。我們上期舉了下面的例子，可見當 aclk 頻率比 bclk 頻率高的時候，adata 變高一個週期，那麼有可能 bclk 的時鐘沿根本看不到這個變化，或者有時候即使能被 bclk 採到一次，也可能無法滿足 3 個沿的要求，導致無法用常見的 2 flip flop synchronizer 來去 sync。



那你肯定就會問了，如果 bclk 的頻率比 aclk 高，比 1.5 倍還高（3 個沿的基本要求），比如 10 倍，一個 aclk 週期內可以見到 bclk 的 10 個上升沿，那不是肯定滿足 3 個沿要求嗎？難道這種情況也不能用 2 flip flop synchronizer 嗎？在回答這個問題之前，我們首先要回答一個問題：

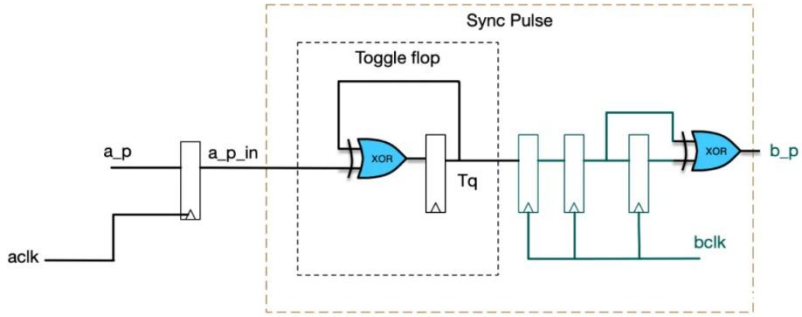
aclk 時鐘域的一個 pulse，到了 bclk 時鐘域內，應該是什麼樣的信號呢？

如果這個問題不好回答，那麼換一個問法：為什麼要把 aclk 的 pulse 同步到 bclk 時鐘域呢？

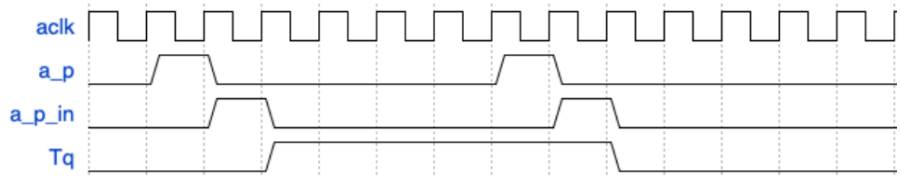
這才是觸及靈魂的問題，對於普通的面試者，能夠正確回答上 **pulse synchronizer** 當然不錯，但是能夠從根本上理解並且講清楚 pulse synchronizer 存在的理由，那才是真正優秀的候選者。其實這個問題在工程師設計電路的時候也要自己問自己，當你無法給出確信的理由說你一定要把 aclk 的 pulse 同步到 bclk 時鐘域，那你應該重新思考，也許你就會發現可能你真的不需要一個 pulse synchronizer。

NOTE：脈衝同步器 (Pulse Synchronizer)

- 使用時機：A domain 的訊號同步到 B domian 得時候也需要是一個脈衝(Pulse)
  - 例如: Memory WE(Write Enable)的寫操作，如果原本在 A domain 是一個脈衝，但是同步到 B domain 變成不是一個脈衝，那就可能導致 Memory 被寫多次
- 方法
  - 將 aclk 的 pulse 訊號轉換成一個準位(level)訊號
  - 用同步器来同步這個 level 訊號
  - 在 bclk 將同步過來的 level 訊號轉換成 pulse



- 將原本 clka 的脈衝訊號轉換成一個準位訊號



- 要求
  - 從上方圖看到 Tq 後接了一個同步器，代表需要滿足同步器訊號的穩定時間要有三個邊緣的時間，代表 A domain 的脈衝時間不可以太短。滿足 B domain 三個邊緣時間的要求。

在電路設計中，經常我們需要設計一些 pulse 信號，比如說，有一個 counter 在不停地計數，每個週期加一或者減一，當 counter 的值等於一個特定的值的時候，我們就輸出一個週期的 pulse，可以用這個 pulse 來作為使能信號(enable)來做其他的事情，例如去 set 或者 clear 某個寄存器。反過來，也有可能用這個信號去作為某個 counter 增加 1 或者減去 1 的條件。再比如說，需要對 memory 進行讀操作或者寫操作，現在 memory 通常都有兩個使能埠 CE(Chip Enable)和 WE(Write Enable)，CE 和 WE 同時為 1 的週期表示要對 memory 進行寫操作，CE 為 1 但 WE 為 0 的表示要對 memory 進行讀操作。再比如對於一個 FIFO，push 為高一個週期就表示給 FIFO 加入一個數，pop 為高一個週期就是表示給 FIFO 減去一個數。以上這些例子，都說明瞭當這些信號為高時，就要有相應的操作發生，為高一個週期，就操作一次，再次為高時，就需要再操作一次，這是和另外一些狀態信號(status signal)的差別。對於那些狀態信號，它們為高或低只表示一種狀態，而與它們為高為低經過了多少個時鐘週期沒有關係。我們在上一篇講到用 2 flip flop 來同步的單 bit 信號，幾乎都是針對的那些狀態信號。而對於 active 時需要進行相應操作的信號來說，很顯然由於 2 flip flop synchronizer 的限制，adata 同步到 bclk 時鐘域就無法保證持續相應的週期數，這裡可能是最開始的例子 bclk 連一個 cycle 的 pulse 都沒有，也可能是持續了多個 cycle，自然不能用 2 flip flop synchronizer 了。

我們現在可以回答之前提出的問題了，當我們要同步 aclk 時鐘域的一個單週期的 pulse 到 bclk 時鐘域時，我們期望 bdata 是什麼樣呢？答案就是，bclk 時鐘域也是單週期的一個 pulse。

那麼如何克服 2 flip flop synchronizer 的問題呢？咱們來看，由於 pulse 只持續一個週期，2 flip flop synchronizer 可能會 miss 掉 pulse，那我們想個辦法讓產生過 pulse 這個之前發生過的事件

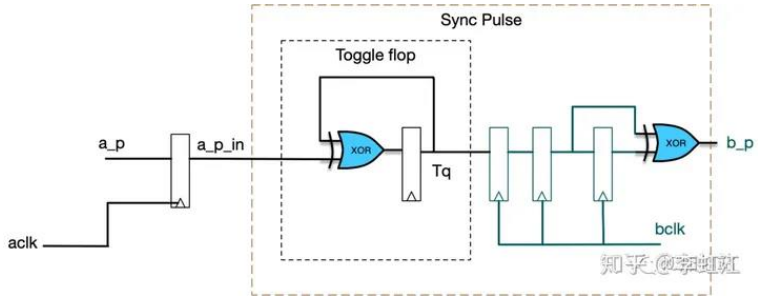
記錄在那裡不就好了嗎？這就是破解這個問題的思路：

將 aclk 時鐘域的 pulse 信號轉為一個 level 信號

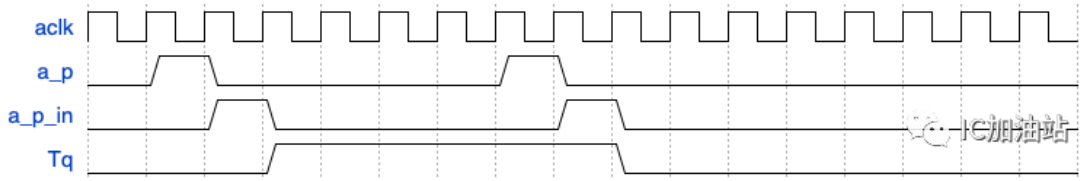
用 2- flip flop synchronizer 來同步這個 level 信號

在 bclk 時鐘域將同步過來的 level 信號轉化為 pulse

好了，廢話不多說，直接上圖



其中的關鍵就是圖中所示的 Toggle Flip Flop。可以看到當輸入是一個單週期的 pulse 時，裡面 Toggle flip flop 的輸出只會翻轉一下。
NOTE：T 型暫存器，輸入是 1 才會將輸出反向。



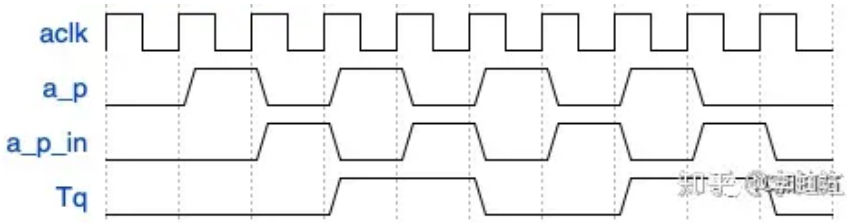
這樣我們就把一個 pulse 轉化成為了 level，這個 level 信號直到下一個 pulse 來之前都是穩定的，於是我們就可以利用 2 flip flop synchronizer 來將這個 level 信號同步到 bclk 時鐘域，然後我們再借助一個 XOR 和一個 flip flop 來重新創造出一個 pulse。

如果你面試的時候已經能夠正確回答出上面的電路圖，在面試官心裡你已經及格了，但是距離完勝其他求職者還要再深入思考一些。

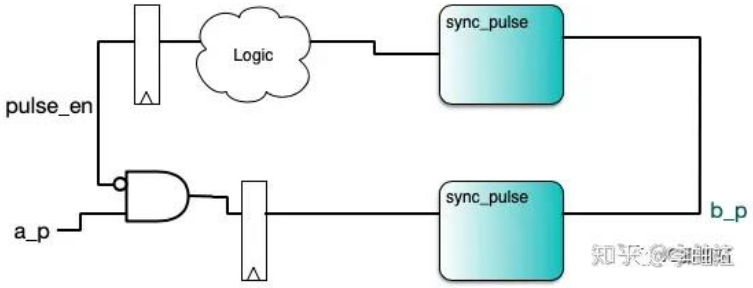
我們繼續看，既然這個 pulse synchronizer 中間利用了 2 flip flop，那麼 2 flip flop 的 3edge 要求就必須要滿足，換句話說，我們轉化成為的 level 的信號 Tq 要足夠長。如果 Tq 不滿足 bclk 的 3edge 要求，那麼這個 level 信號我們就無法同步過去，也就無法產生 bclk 的 pulse 了。而 Tq 每次變化是由於 aclk 來了一個新的 pulse，這也就是要求 aclk 的連續兩個 pulse 之間的間隔要足夠大，要滿足 bclk 的 3edge 要求。

如果你回答出來了這個 pulse synchronizer 的局限性，那麼面試官很可能會接著問，如果我不知道下一個 pulse 是什麼時間來怎麼辦呢？老李當年還就真被這麼問到過，把老李差點整懵逼了。我們先來看，aclk 時鐘域最接近的兩個 pulse 能靠多近呢，顯然就是兩個 pulse 中間只有一個 aclk 週期，這其實就是將 aclk 進行了 2 分頻。那麼相應的，Tq 就是對 aclk 進行了 4 分頻，每個 Tq 的 level 持續時間是 2 個 aclk cycle，這 2 個 cycle 需要滿足 bclk 的 3edge 要求，如

果滿足，那麼就可以保證 bclk 域也可以產生每個 cycle 的 pulse 的要求。



如果無法滿足，那麼我們就要另想辦法，如果還是要使用 pulse synchronizer 這個電路，我們繼續祭出回饋大法，這一方法我們上一篇已經見過。



本質思路就是我們不能讓 aclk 域的 pulse 產生得很快，而是要等到 pulse 同步到 bclk 之後才能繼續產生下一個 pulse，於是我們要將 b_p 重新 synchronize 回到 aclk 域，作為放行下一個 pulse 的條件。

可是這個回饋的辦法要求每個 pulse 產生都要同步回來，這樣做的效率不高，有沒有更快的辦法呢？我們想像有一個細長的管子，管子的一頭我們往裡面塞玻璃球，每塞進去個玻璃球對應一個 aclk 時鐘域的 pulse，管子的另一頭我們往外彈玻璃球，每彈出一個玻璃球對應一個 bclk 時鐘域的 pulse。（哎，是不是嗅到了一點 FIFO 的味道了？）那麼除非假定管子無限長，那麼我們往管子裡塞入球的平均速率不能永遠大於從管子裡彈出球的速率，我們可以允許某一段時間塞入球的速率高，管子作為一個緩衝區，來容納之前塞入但是還沒有彈出的球，但是當管子塞滿的時候，我們必須得停下塞入球的動作，等待彈出球的那邊彈出球之後才能騰出空間來繼續塞。也就是說，當管子不是無限長時，儘管兩邊的時脈速率不同，能夠保持一一對應的條件是兩邊塞球彈球的平均速率是一樣的。這就是利用 FIFO 來解決的思路。如果 FIFO 告訴 aclk 說 FIFO 滿了，那麼 aclk 域就得停止產生 pulse，如果不滿，就可以繼續產生 pulse，最後我們把 FIFO 裡面的元素完全彈完，就可以做到一一對應了。

於是，又給下一篇埋下坑了，下一篇老李會帶大家瞭解 CDC 面試中最愛考察的 asynchronous FIFO，敬請關注。

注：題圖與內容無直接關係，我們這裡討論的 pulse 通常都是方波。題圖就是吸引大家眼球用的。

4.0 多 bit 信號跨時鐘域怎麼辦？ -- CDC 的那些事（4）

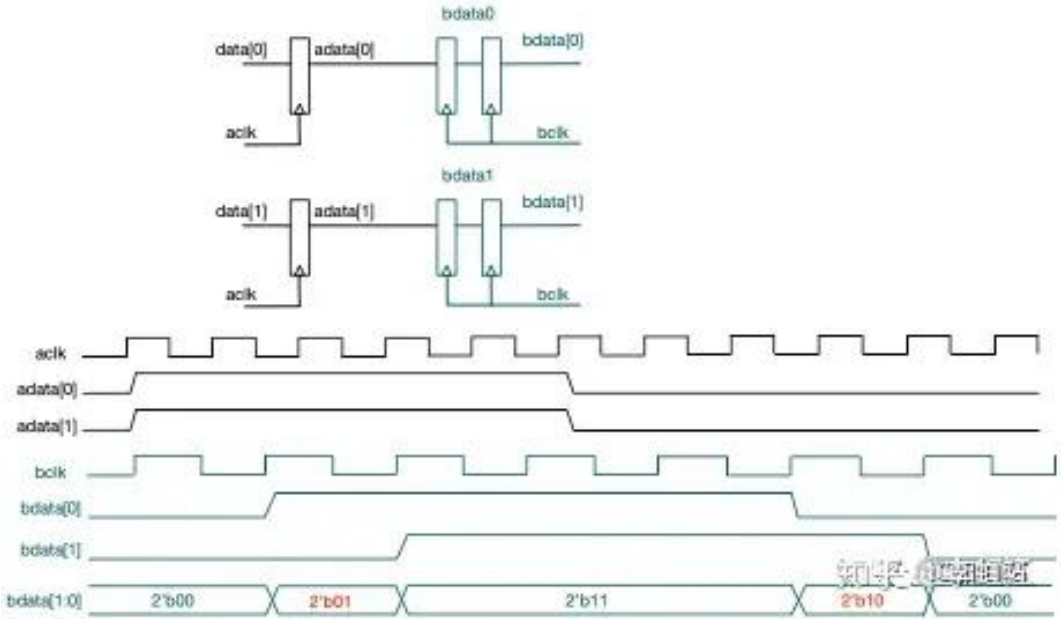
相信經過前面三篇 CDC 的那些事，大家對於單 bit 信號的跨時鐘域有了相應瞭解。下面老李

帶大家破解多 bit 信號的 CDC。

這次咱先不廢話，直接上一個結論：**在絕大多數情況下，我們不能直接利用 2 flip flop synchronizer 來同步一個多 bit 信號**。（為什麼說絕大多數情況，在這篇文章最後我們會講用 2 flip flop synchronizer 來同步的例子）。

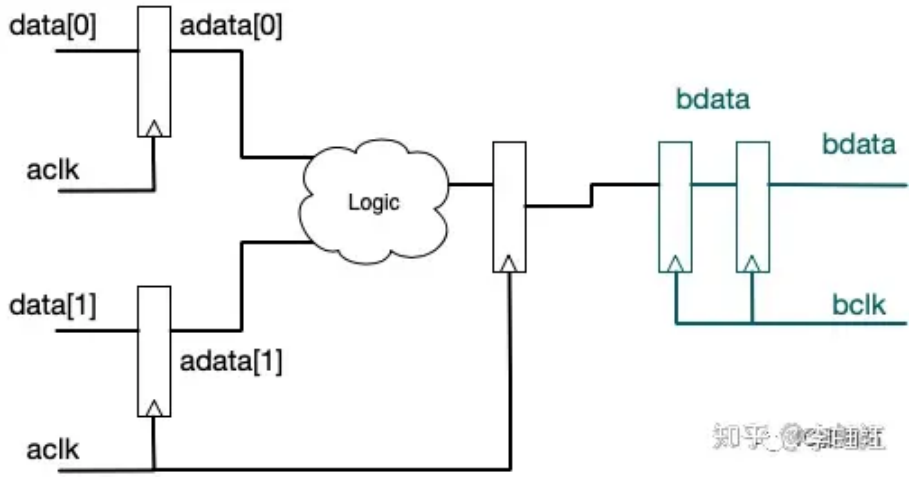
注意我們說多 bit 信號，是說這個信號是由多於 1 個 bit 來表示的。比如說一個 couter 的值，或者是一個 address，再或者本身就是一個多 bit 的 data bus，這些 bits 之間是相關的，單獨拿出來是沒有意義的。（有的時候有人會把幾個單 bit 的控制信號 group 成一個多 bit 的信號，但實際上各自 bit 是各自獨立的，在同步到 bclk 時鐘域之後也是各自起作用，這種情況其實是“偽”多 bit 信號，是可以對每個信號用 2 flip flop synchronizer 同步的，多說一句，這個在 CDC 檢查工具裡面大家可以加 unrelated attribute 來告訴工具這些 bit 雖然看起來是屬於一個多 bit 信號，但其實是不相關的。）

那麼來看一個為什麼不能用 2 flip flop synchronizer 來同步各個 bit 的例子。



注意 adata 從 2'b00 變到 2‘b11，一段時間之後再變為 2'b00，但是因為 2 flip flop synchronizer 的 delay 有隨機性，可能是一個週期之後就同步過去了，也可能需要兩個週期。這樣我們就可能在 bdata 上看到一個週期的 2'b01，之後也可能看到一個週期的 2'b10，這兩個值都是 adata 沒有出現過的，也就是說 bdata 出現了錯誤的值。

那麼怎麼解決這個問題呢？老李建議大家這個時候先別著急往下看，而是停下來想一想，你是不是真的需要同步一個多 bit 的信號。有的時候你把 CDC 的分界線挪一挪，可能你就不需要同步一個多 bit 的信號了，而是只需要同步一個單 bit 信號就行，當然要注意同步單 bit 信號之前要寄存一下。

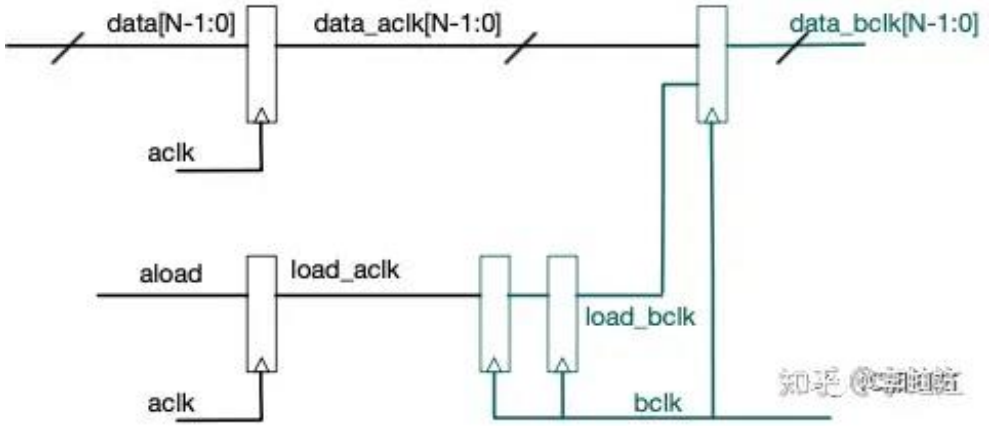


好了，經過你的思考，你告訴老李，不行啊，必須得同步多 bit 信號過去，咋辦呢？

方案一

我們說直接用 2 flip flop synchronizer 同步多 bit 信號 adata，如果 adata 的信號在同步的時候變化，就會導致上面出錯的問題。那麼我能不能想個辦法，說 bclk 在採樣 adata 的時候，adata 的所有 bit 都穩定不變呢？這樣就不存在不同 bit 之間 delay cycle 不同的問題了。於是思路如下

- 在 aclk 時鐘域產生一個 load_aclk 信號，load_aclk 為 1'b1 時代表多 bit data 信號穩定
- load_aclk 信號本身利用 double flip flop 同步到 bclk 時鐘域得到 load_bclk
- bclk 時鐘域可以直接利用 flip flop 來 load bus 信號



看起來很簡單不是嗎？但其實這裡面有幾個隱藏的坑要注意。

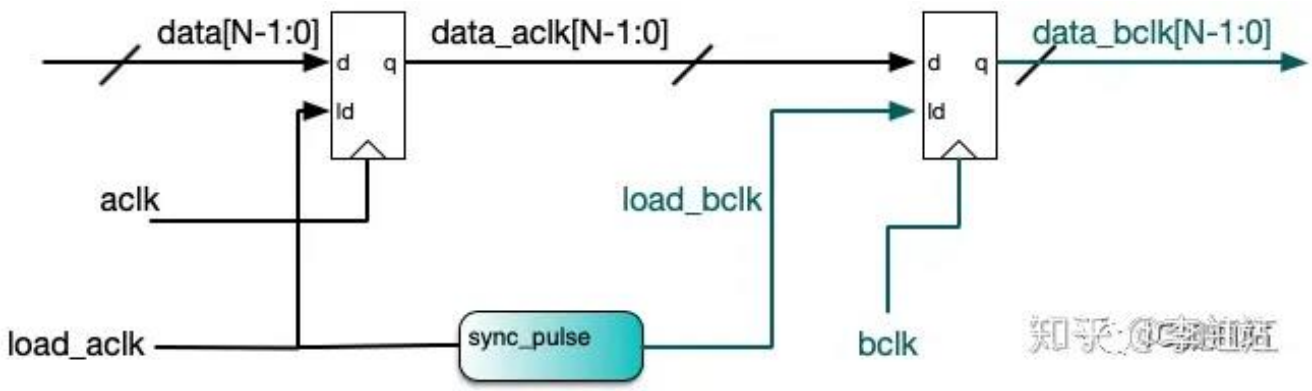
第一，要有專門的邏輯保證 aload 為高的時候 data_aclk 不變。

第二，在 aload 為 1'b1 的時候，data_bclk 會持續 load data_aclk, aload 從 0—>1 是 ok 的，但是 1—>0 會發生錯誤，因為 data_aclk 是不穩定的！

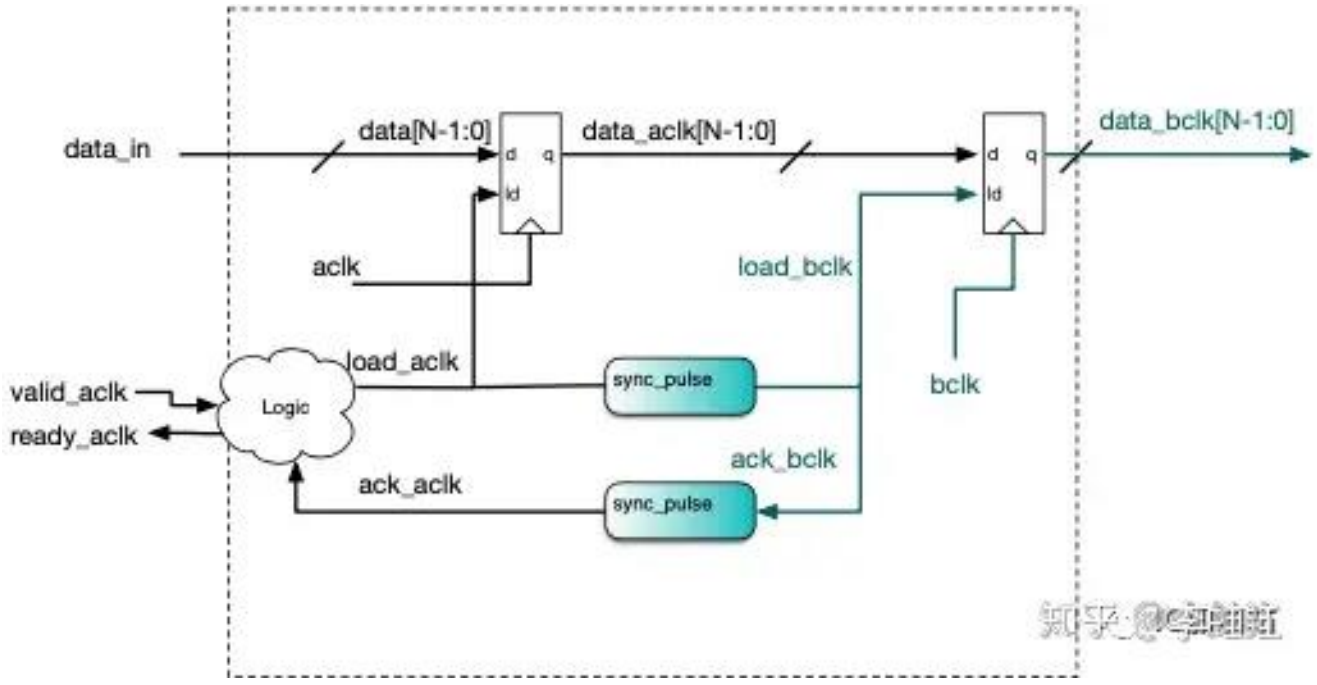
第三，aclk 時鐘域怎麼知道 data_aclk 已經被成功傳到 bclk 時鐘域，從而可以更新下一組 data

了呢？

我們首先看如何解決第二個問題。我們其實需要的是：當 load_aclk 變高的時候，把 data_aclk 當前的值同步過去之後就行了，並不需要持續 load。這個時候我們上一篇講的 pulse synchronizer 就派上用場了，我們讓 load_aclk 是一個 pulse，然後把這個 pulse 同步過去，這樣 data_bclk 只會 load 一次。



可是這個還是沒有辦法解決第三個問題，要解決它，我們只能繼續引入回饋大法：把信號從 bclk 時鐘域回饋回來，告訴 aclk 時鐘域 load 成功，可以更新下一個資料了。如下圖所示，aclk 時鐘域的 load_aclk 是由一個 valid/ready 的握手邏輯產生。我們可以把 load_bclk 再利用 pulse synchronizer 同步回去，從而讓 ready_aclk 為 1，這樣我們就知道 data_aclk 肯定已經被同步到了 bclk 時鐘域，可以更新下一個 data 了。



似乎大功告成了是吧？別急，再思考一下上面的電路，如果 data_bclk 的後級還沒來得及用它，而 data_aclk 卻開始更新了下一個數，那 data_bclk 是不是會被新的數覆蓋呢？這個問題要怎麼解決老李就不直說了，相信聰明的你很快就可以想出來了。

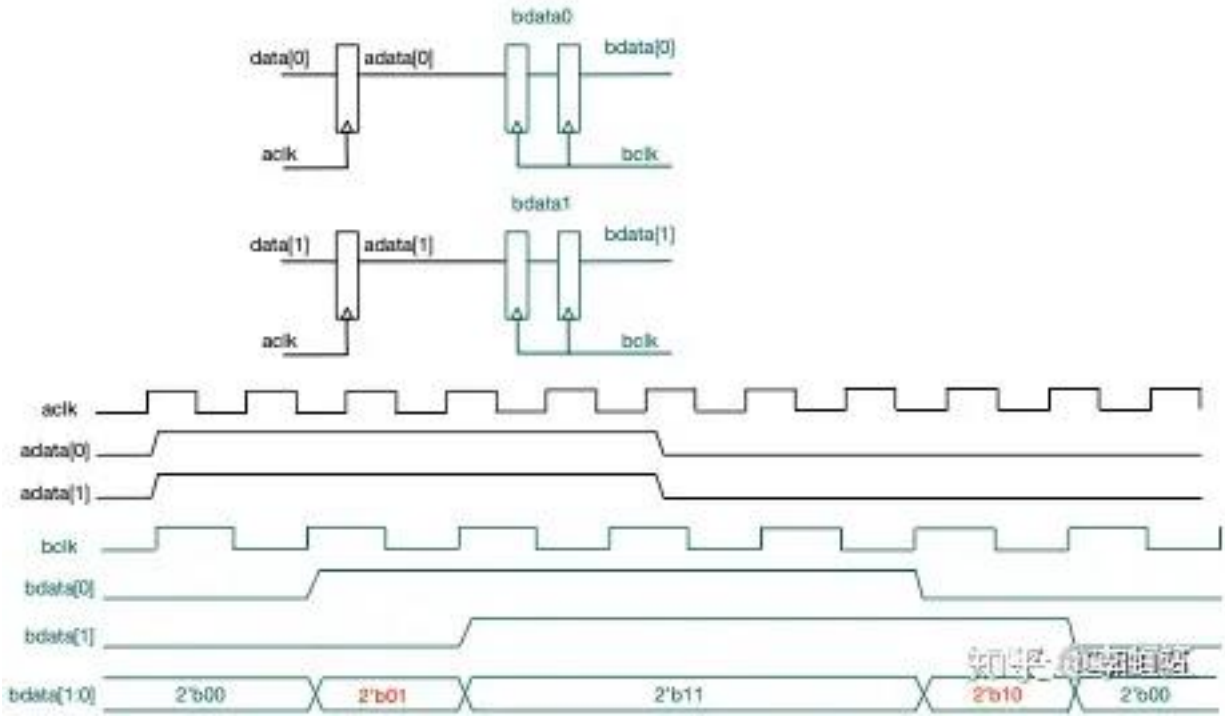
總結一下，方案一 (就是 Hand shake) 適用於

- 無法化簡為單 bit 信號跨時鐘域傳遞
- 適用於非高速傳輸的場合，即在 source 時鐘域的多 bit 信號可以保持穩定一段時間，而不是時刻都在變化，可以有一個明確的 load 視窗
- load 信號為高時必須保證多 bit 信號穩定不變
- 如果沒有連續同步資料的要求，可以適當使用不帶回饋的

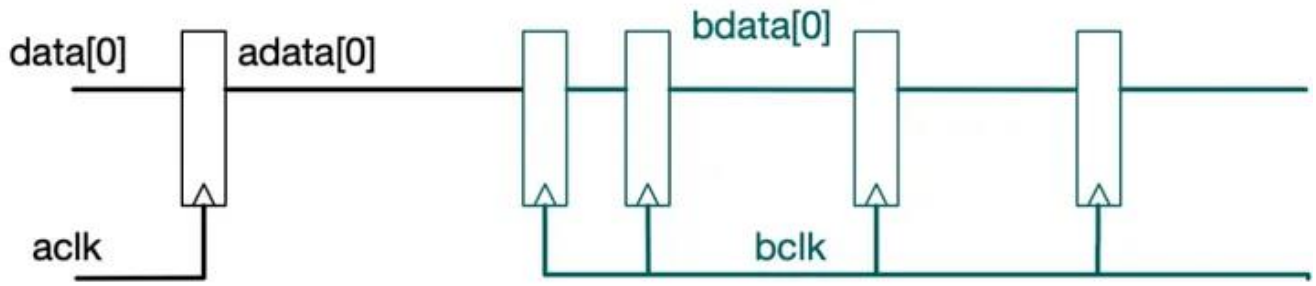
方案二

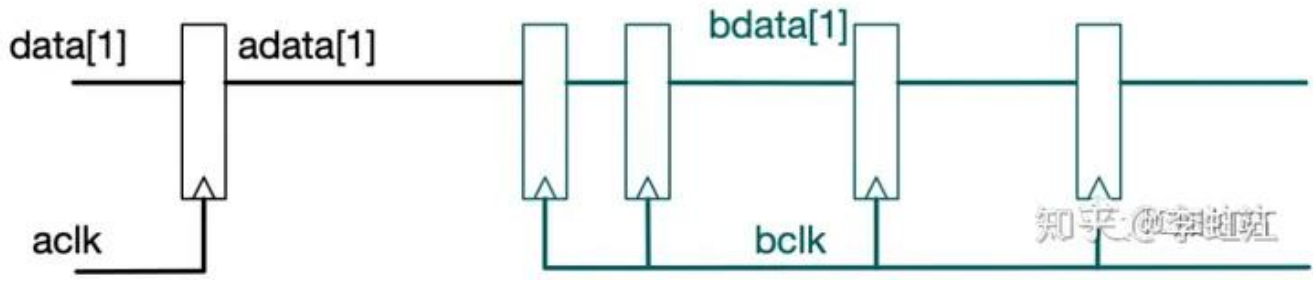
話說老李當年掌握了方案一，以為掌握了降龍十八掌，加上 Asynchronous FIFO 就可以橫行天下了。結果有一天碰到一個別人設計的模組，要做 CDC，多 bit 信號但是沒有 load 信號，因為多少年的老設計了沒人願意動，這可咋整？

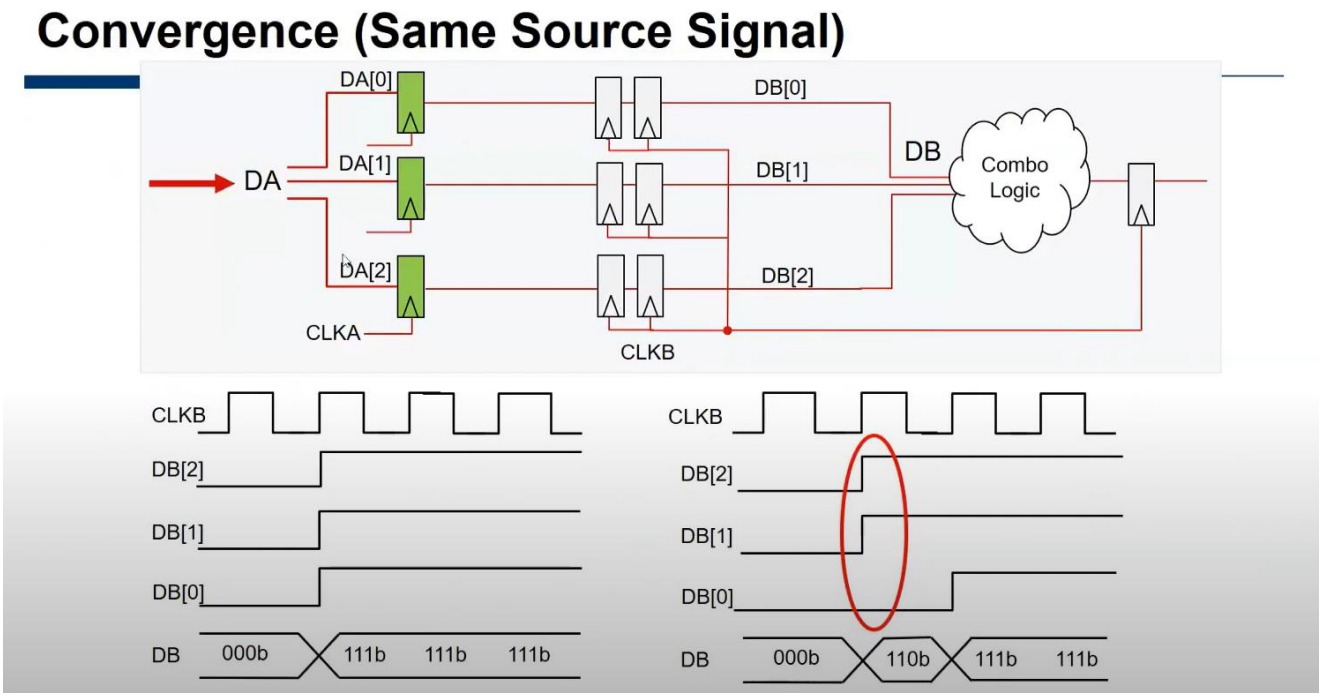
我們再重新看一下上面的那個出問題的圖



儘管我們看到了 2'b01 和 2'b10 這兩個錯誤的值，但是這兩個值中間可是 2’b11 是正確的值啊，而且 2'b11 至少持續了 3 個週期，那麼我們其實可以設計一個比較邏輯，利用 2 flip flop synchronizer 同步到 bclk 時鐘域之後，再用兩級 flip flop 把 bdata 打兩拍，然後比較這 3 級的值，如果這三級 flip flop 的值是相同的，那不就證明 2 flip flop synchronizer 同步到的值是穩定的嗎？我們可以用三級 flip flop 的值相等作為一個 update 信號，來 updata 最後輸出級的 flip flop（輸出級沒有畫）。







當然我們需要注意的是，這樣做的話要求 adata 變化不是很頻繁，因為 bclk 這邊要等好幾個週期去比較值是不是穩定，如果 adata 變化非常快，可以想像，bclk 這邊的三級 flip flop 可能始終沒有辦法達到彼此相等，從而就無法更新輸出級了。

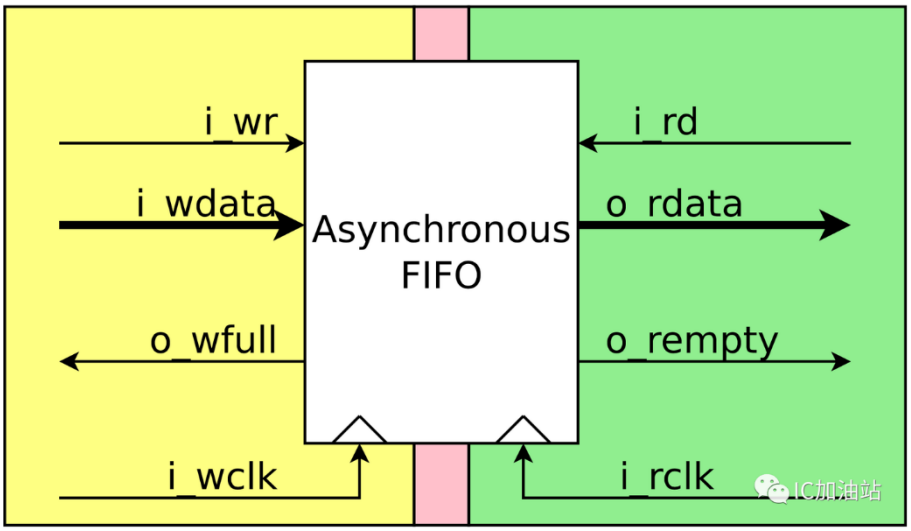
這也是這個方案的缺點，和方案一中沒有回饋的結構一樣，無法保證每次 adata 的變化都被 bclk 實際同步到了。但是如果你確實知道 adata 變化頻率很低，每變一次之後會穩定很長時間，或者說 bclk 這邊不在乎是不是錯過了些 data，那麼你確實可以用方案二。方案二是當你

沒有辦法拿到 aclk 域的 load 信號時的 back up 方案，所以你必須深刻瞭解它的限制條件。最後還有一點，方案二需要很多級 flip flop，三級 flip flop 可能有的時候還不夠，要具體分析，但是很明顯方案二需要的 flip flop 數目更多，尤其是 bit 數大的時候，面積的花費可能要更高。

方案三就是非同步 FIFO 了。Asynchronous FIFO 裡面的知識點太多了，值得再來兩篇好好探討一下。下周咱們不見不散。

最後把文章開始埋的坑填一半，什麼情況下可以用 2 flip flop synchronizer 來同步多 bit 信號呢？其實方案二已經算一種了，還有一種情況就是這個信號是格雷碼編碼，這個咱們下一篇細細講。

5.0 面試必殺技：非同步 FIFO（上） -- CDC 的那些事（5）



這一篇老李終於要開始聊非同步 FIFO(Asynchronous FIFO)了。在知乎上曾經老李見過一個問題：

碩士生找工作的時候把非同步 fifo 寫成一個專案經歷，是不是顯得很 low 啊？

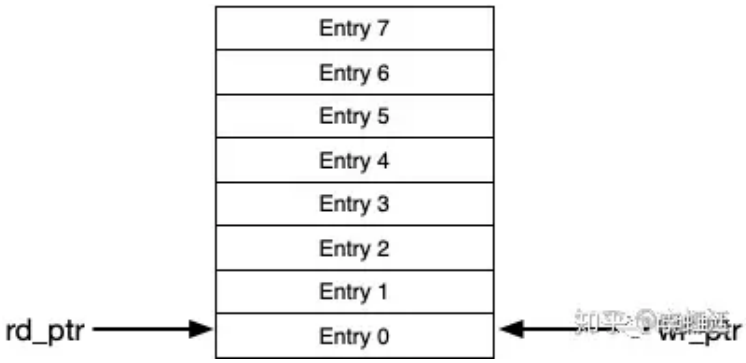
下面回答很有意思，老李發現那些回答很 low 的以在校學生和剛畢業的應屆生居多，大家都覺得簡歷裡面一定要寫上什麼無線接收機，USB 控制器等等才顯得高大上。而幾個來自真正工業界大佬的回答反而不說 low，比如下面的這個回答來自海思的架構師說

我如果面試，不會介意。但會很樂意基於這個話題繼續問你對非同步原理的理解，例如格雷碼的原理，兩拍同步或者三拍同步的差異，或者 FIFO 內 DATA 如何保證絕對穩定等問題來進一步試探，這才是決定結果的

老李自己也基本認同這個觀點，即非同步 FIFO 裡面其實需要瞭解的東西很多，幾乎涵蓋了 CDC 中所有相關的知識點，面試官可以就非同步 FIFO 裡面很多的點進行提問。老李當年面試矽谷各大晶片公司，幾乎都被問到了非同步 FIFO。所以說，深刻理解非同步 FIFO，是一個合格的前端晶片設計工程師必須掌握的基本技能。

在開始講非同步 FIFO 之前，老李先帶大家簡單回顧一下同步 FIFO。FIFO 就是一個存儲的管

道，有進的口，有出的口。同步 FIFO 就是說進口（寫入端）和出口（讀出端）是同一個時鐘域。FIFO 一般深度多於 1，就需要兩個指標: write pointer 和 read pointer。



對於 write pointer 和 read pointer 我們一般用 2 進制，寫入操作(Push)使得 write pointer + 1，讀出操作(Pop)使得 read pointer + 1。這就像是兩個人在一個環形跑道上賽跑。當 write pointer 領先了 read pointer 一圈之後，也就是說 FIFO 裡面所有的存儲單元都存了資料，FIFO 沒有空餘的存儲單元了，我們就說 FIFO 滿了。反過來，當 read pointer 追上了 write pointer，所有的存儲單元都空閒了，我們就說 FIFO 空了。

對於**非同步 FIFO** 來說，Push 和 Pop 分別在不同的 clock domain，那麼最核心的問題就是空滿的判斷了。在 Pop 的這一側，FIFO 空不空是關鍵，因為空的時候不能 Pop，滿不滿反而不重要。在 Push 的這一側，反過來，滿不滿才關鍵，因為滿的時候不能繼續往進 Push。因此，我們就要在讀的這一側判斷 FIFO 是否空，在寫的這一側來判斷 FIFO 是否滿。當然我們還是要有 read pointer 和 write pointer，在 pop 這一側更新 read pointer，在 push 這一側更新 write pointer。那麼當我們要把 pointer 同步到另外的時鐘域進而去比較的時候，我們就遇到了上一講討論的 multi-bit 同步的問題，即 binary counter 不能直接利用 double flip flop 來同步。

那麼我們上一篇講到的帶回饋的 asynchronous load 模組可以用來同步 pointer 嗎？可以是可以，但是缺點也很明顯，即回饋的話要跨兩次時鐘域，對於效率很有影響，比如說 push 這一側要等到回饋信號回來之後才能繼續下一個 push，哪怕 FIFO 裡面還有很多空閒的單元。pop 的這一側也是一樣。這樣對於 FIFO 的整體性能影響太大。

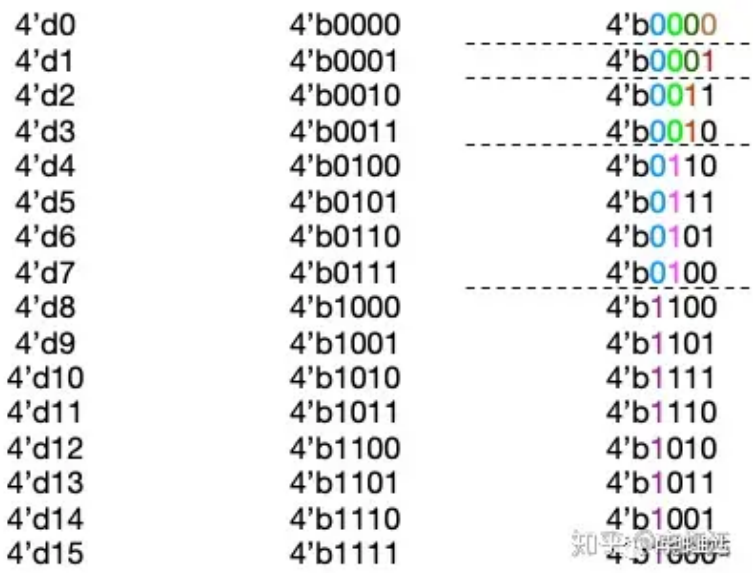
那有沒有更快的辦法呢？答案就是老李上一期最後埋的坑 -- 用格雷碼 Gray Code。

格雷碼是以美國學者 Frank Gray 于 1947 年提出的一種二進位編碼方式，後面這種編碼方式就以他的名字命名。這種編碼方式的特點老李以兩句話概況

- 每相鄰的兩個編碼之間有且只有一位不同
- 當第 N 位從 0 變到 1 的時候，之後的數的 N-1 位會關於前半段軸對稱，而比 N 位高的位是相同的。

記住了以上兩點，老李保證你可以現場推出來 Gray code 是什麼樣的。說實話，老李自己也記不住二進位到 Gray code 的轉換公式，每次都是寫出來現場推。（相信老李，面試官問你的時候除非他自己面你之前背了公式，否則他也得現推。）如果你只記得 1，你現場可能推不出

來，所以老李還是建議你把 2 也記住。下面這幅圖就是用 10 進制，傳統 2 進制以及 Gray code 來表示 0-15 個數。



4'd0	4'b0000	4'b0000
4'd1	4'b0001	4'b0001
4'd2	4'b0010	4'b0011
4'd3	4'b0011	4'b0010
4'd4	4'b0100	4'b0110
4'd5	4'b0101	4'b0111
4'd6	4'b0110	4'b0101
4'd7	4'b0111	4'b0100
4'd8	4'b1000	4'b1100
4'd9	4'b1001	4'b1101
4'd10	4'b1010	4'b1111
4'd11	4'b1011	4'b1110
4'd12	4'b1100	4'b1010
4'd13	4'b1101	4'b1011
4'd14	4'b1110	4'b1001
4'd15	4'b1111	4'b1000

老李所說的軸對稱是什麼意思呢？請看 Gray code 前兩個數 4'b0000, 4'b0001，它們倆之間可以畫一條對稱軸，第 1-3 位都是相同的。再看前 4 個數，在 4'b0001 和 4'b0011 之間畫一條對稱軸，第 2、3 位是相同的，第 0 位則是軸對稱的，從 0-1 到 1-0。之後的規律老李也在圖上標出來了，一看就懂。有了這兩個規律，更多位數的 Gray code 老李相信你也可以現場直接寫出來。

NOTE：0-1 會是對稱最後一位，0-4 對稱最後兩位，0-7 對稱最後三位…(自己寫看看就知道啦)

然後咱們就來推一推關係，你只要大概記住需要用到異或操作，就能推出來

bnary to Gray
g(3) = b(3) ^ 0
g(2) = b(3) ^ b(2)
g(1) = b(2) ^ b(1)
g(0) = b(1) ^ b(0)
g(n) = b(n+1) ^ b(n)
Gray to Binary
b(3) = g(3)
b(2) = g(3) ^ g(2)
b(1) = g(3) ^ g(2) ^ g(1)
b(0) = g(3) ^ g(2) ^ g(1) ^ g(0)

Gray code 有什麼魔法之處，能夠突破 muti-bit 不能用 2 flip flop synchronzier 的限制呢？關鍵就在於它的第一個特點：相鄰兩個編碼之間有且只有 1 位不同。我們說 multi-bit 如果在一個時鐘沿有多個 bit 同時翻轉，在另外一個時鐘域采到的時候由於 2 flip flop 穩定需要 1 個或 2 個週期，所以可能會出現錯誤的值。Gray code 這種編碼，從根本上就沒有這個問題，因為以 Gray

code 編碼作為計數器，每個時鐘沿來的時候只會有 1 個 bit 發生了翻轉，其餘所有 bit 都是穩定的！這樣即使這一個 bit 在用 2 flip flop synchronizer 同步到另外一個時鐘域時，可能需要 1 個週期發生變化，或者 2 個週期，在發生變化前，另一個域的值就是之前的穩定值，變化後就是新的值，而不會出現其他不該出現的值。

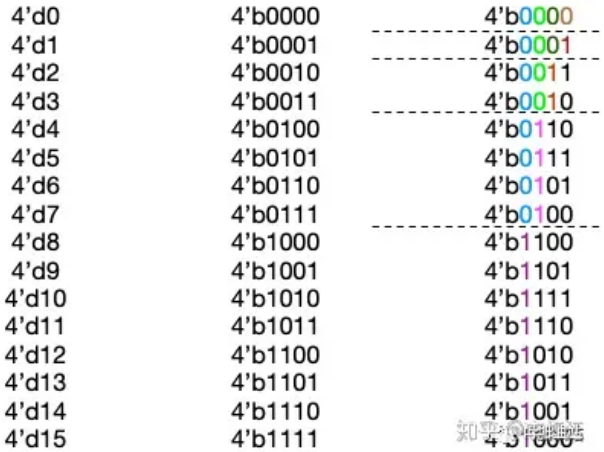
用了 2 flip flop synchronizer 來同步，省去了回饋，把 read pointer 同步到 write domain 來判斷滿，把 write pointer 同步到 read domain 來判斷空，只需要跨一次 domain，就可以判斷，這樣可以提高 push 和 pop 的效率。

而對於 memory 的取址還是得用 2 進制編碼，我們需要做的就是在同步 pointer 的時候把 binary pointer 轉化為 gray code pointer，然後用 2 flip flop synchronizer 同步到對面時鐘域之後，再來判斷空滿。這裡有個問題，我們需不需要把 gray code 再轉化為 binary code 之後再來比較空滿呢？當然可以，我們學習新知識就是不斷把問題轉換為已經解決的問題，在同步 FIFO 的時候我們已經知道怎麼用 read pointer 和 write pointer 判斷空滿，那麼自然而然我們可以這樣做。那麼有沒有更快的辦法呢？我們能不能直接利用 gray code 來判斷空滿呢？

有！我們再仔細觀察 gray code。和同步 FIFO 一樣，我們對於 2^n 個 entry 的 FIFO, 需要 N+1 個 bit 來表示 address 和 gray code。以下面的編碼為例，假設 FIFO 有 8 個 entry，我們用 4 位來表示。

FIFO 空比較好判斷，write pointer == read pointer，用 binary 或者 gray code 都行，要求每位都相同。

滿稍微複雜一點，我們舉例來說，假設 FIFO 一開始一直寫，不讀，寫滿 8 個 entry 後 write pointer 的 binary 變成 4’b1000, gray code 是 4‘b1100， 而 read pointer 的 gray code 是 4’b0000，可以看到高兩位是相反的，之後的低位是相同的。再舉個例子，假設 write pointer 的 gray code 到了 4'b1011, 而這個時候 read pointer 如果是 4'b0111，那麼也是 8 個 entry 滿了。



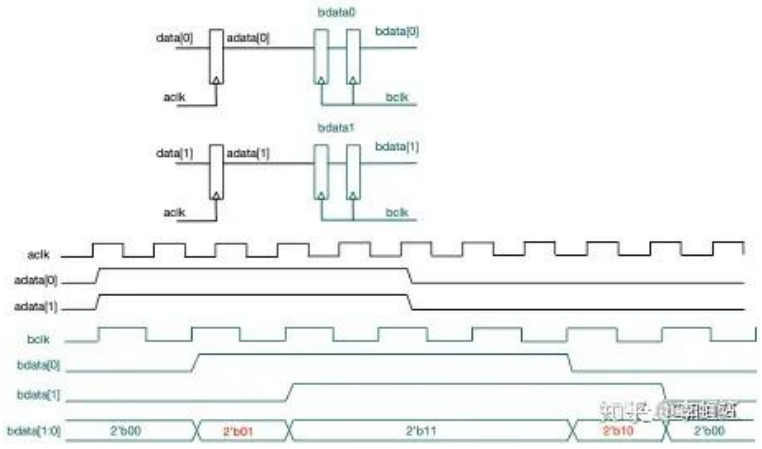
所以我們歸納出，利用 gray code 判斷滿的條件為:

```
assign full = (write_ptr_gray[N:N-1] == ~read_ptr_gray[N:N-1]) &&(write_ptr_gray[N-2:0] == read_ptr_gray[N-2:0]);
```

最後再說一個面試中的常見問題，這個問題知乎上也有人提

慢時鐘域同步快時鐘域格雷碼時候，在慢時鐘域的一個週期中，經歷了兩次或多次快時鐘域的上升沿，那麼對應的格雷碼就會有兩個或多個 bits 發生變化，這個不會產生多個 bits 同步的問題嗎？

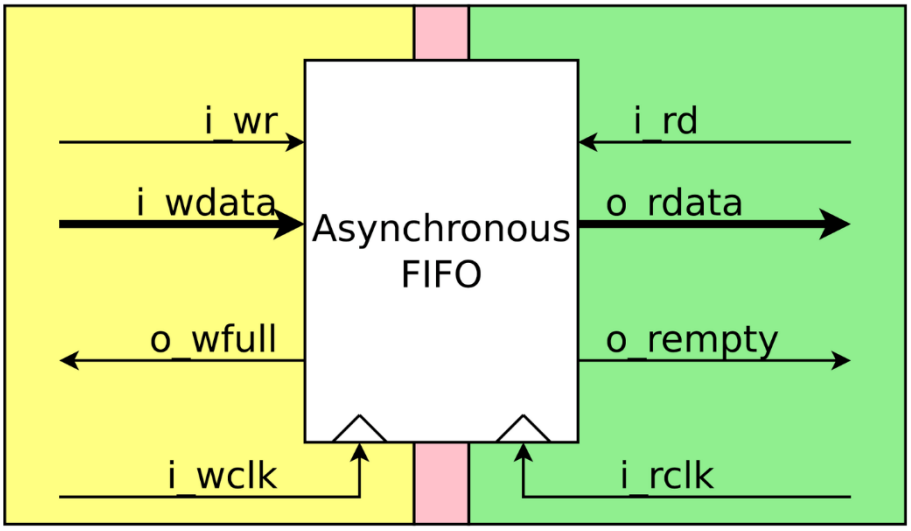
說實話老李當年面試就被問過好幾次這個問題。這個問題很有迷惑性，但是回答起來也很簡單，其實老李上面已經回答了，這裡再複習一下，繼續搬出我們之前看過的圖。我們說多個 bit 發生變化其實是針對 source clock 的每一個 edge 來說的，因為不同 bit 之間發生翻轉的時間不能嚴格對齊，所以會導致 destination clock 可能看到不同的值，導致最後 synchronizer 輸出會出現錯誤的值，從而影響 FIFO 的空滿判斷。而 gray code 在每個 source clock 的沿只會有一個 bit 發生翻轉，其餘 bit 保持穩定，這樣每個 destination clock edge 來的時候最多也只可能碰到 1bit 在翻轉，這個翻轉的 bit 可能會給 synchronizer 的第一級引入 metastable，但是最後 synchronizer 的輸出無非就是保持前值或者是更新後的值，而這兩個值都是合理的值，不會出現一個錯誤的值從而導致 FIFO 空滿判斷邏輯錯誤。雖然慢時鐘域同步過來的值可能和之前的值相比有多個 bit 發生變化，但是這些 bit 的翻轉不是同時發生的，這是回答這道題的關鍵。



這周老李工作較忙，上周的推送晚了幾天，拖到了這週末。下周老李繼續帶大家瞭解非同步 FIFO 裡的其他細節內容。比如前面講的判斷空滿是真滿、真空還是假滿、假空？如果 FIFO 的深度不是 2^n，還能用 gray code 嗎？

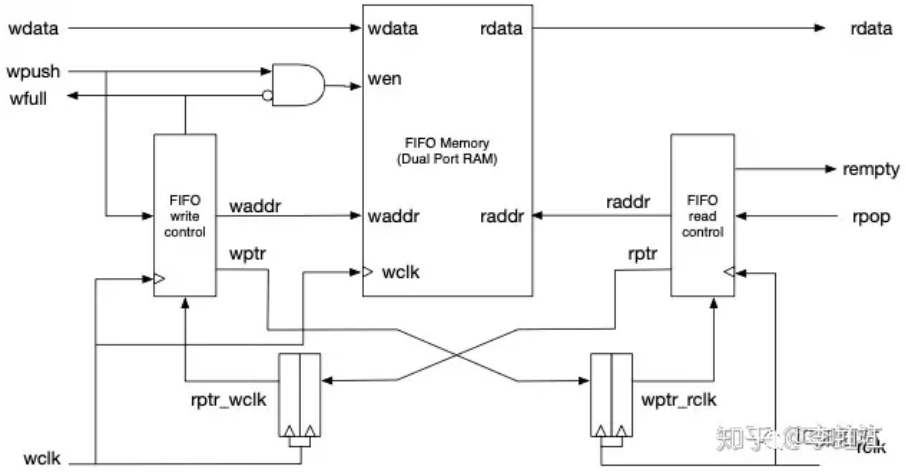
如果你覺得這篇文章對你有所幫助，請微信搜索公眾號”IC 加油站“。 最好能夠分享到群裡或者朋友圈，你的支持就是老李更新的動力！

6.0 面試必殺技：非同步 FIFO（下）-- CDC 的那些事（6）



上一篇老李介紹了非同步 FIFO 的基礎部分，包括為什麼用 Gray Code 來同步 read pointer, write pointer。這一篇咱們從頭一起過一遍非同步 FIFO 的具體設計，然後再討論幾個常見的問題。

有的面試官可能上來讓你先畫非同步 FIFO 的框圖，老李建議大家自己手畫一下，能夠記住。



要注意，wptr 和 rptr 都是 gray code，在上一篇我們已經討論過 gray code 是可以直接利用 2 flip flop synchronizer 來同步的。而用來讀寫實際的 memory 必須是 binary address，在 FIFO write control 和 FIFO read control 裡面我們進行 binary to gray code 的轉換。

下面是上圖中間 fifo memory 部分的簡單實現

```
module fifomem #(parameter  DATASIZE = 8, // Memory data word width
                 parameter  ADDRSIZE = 4) // Number of mem address bits
        (output [DATASIZE-1:0] rdata,
         input  [DATASIZE-1:0] wdata,
         input  [ADDRSIZE-1:0] waddr, raddr,
         input                 wen, wfull, wclk);
        `ifdef VENDORRAM
         // instantiation of a vendor's dual-port RAM
         vendor_ram mem (.dout(rdata), .din(wdata),
                         .waddr(waddr), .raddr(raddr),
                         .wen(wen), .wclken_n(wfull), .clk(wclk));
        `else
         // RTL Verilog memory model
         localparam DEPTH = 1<<ADDRSIZE;
         reg [DATASIZE-1:0] mem [0:DEPTH-1];
         assign rdata = mem[raddr];
         always @(posedge wclk)
           if (wclken && !wfull) mem[waddr] <= wdata;
        `endif
endmodule
```


上面的 code 簡化了 rdata 的邏輯，如果使用 SRAM，可能需要加一級 flip flop 來存儲 SRAM 讀出來的值。

這裡插一句，在設計非同步 FIFO 或者使用非同步 FIFO 的時候，需要計算清楚 FIFO 的深度，（如何 FIFO 的深度計算老李打算以後單獨開一篇文章來討論）然後要比較使用 SRAM 和 flip flop array 的 cost。依據老李的經驗，目前較新的 7nm/5nm 的工藝下，當存儲位元大於 2k bit，使用 vendor 的 compile memory 在面積上開始划算起來，低於 2k bit，使用 flip flop array 划算。這個部分需要大家在實際工作中自己去比較計算。

```
module sync_r2w #(parameter ADDRSIZE = 4)
        (output reg [ADDRSIZE:0] rptr_wclk,
         input      [ADDRSIZE:0] rptr,
         input                   wclk, wrst_n);
        reg [ADDRSIZE:0] rptr_wclk;
//        always @(posedge wclk or negedge wrst_n)
//          if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
//          else         {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
        sync_lib #(.STAGE(2), .WIDTH(4))
                 (.d    (rptr),
                  .q    (rptr_wclk),
                  .clk  (wclk),
                  .rstn (wrst_n)
                 );
endmodule
```


關於 read pointer 和 write pointer 的同步很簡單，用 2 flip flop synchronizer 即可，老李這裡要強調一下，大家在實際工作中不要自作聰明去用 verilog 的 behaivor code 去實現 2 flip flop syncronzier（上面注釋掉的行），而是要直接例化現成的 cdc 庫元件。只要你不是在創立不到一個月的創業公司，這種 cdc library 一定是你們公司已經有的，輪子已經造出來了，千萬別自己再造。

下面我們再看一下 write control 部分的 RTL 實現。

```
module wptr_full  #(parameter ADDRSIZE = 4)
        (output reg               wfull,
         output     [ADDRSIZE-1:0] waddr,
         output reg [ADDRSIZE  :0] wptr,
         input      [ADDRSIZE  :0] rptr_wclk,
input
wpush, wclk, wrst_n);
reg  [ADDRSIZE:0] wbin;
wire [ADDRSIZE:0] wgraynext, wbinnext;
// GRAYSTYLE2 pointer
always @(posedge wclk or negedge wrst_n)
  if (!wrst_n) {wbin, wptr} <= 0;
  else         {wbin, wptr} <= {wbinnext, wgraynext};
// Memory write-address pointer (okay to use binary to address memory)
assign waddr = wbin[ADDRSIZE-1:0];
assign wbinnext  = wbin + (wpush & ~wfull);
assign wgraynext = (wbinnext>>1) ^ wbinnext;
//------------------------------------------------------------------
// Simplified version of the three necessary full-tests:
// assign wfull_val=((wgnext[ADDRSIZE] !=wq2_rptr[ADDRSIZE] ) &&
//                   (wgnext[ADDRSIZE-1]  !=wq2_rptr[ADDRSIZE-1]) &&
// (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
//------------------------------------------------------------------
assign wfull_val = (wgraynext=={~rptr_wclk[ADDRSIZE:ADDRSIZE-1],
                                rptr_wclk[ADDRSIZE-2:0]});
always @(posedge wclk or negedge wrst_n)
  if (!wrst_n) wfull  <= 1'b0;
    else
        wfull  <= wfull_val;
endmodule
```


這裡的滿的判斷用到了我們上一篇講的判斷邏輯，即高兩位元相反，低位都相同。下麵是 read side 判斷空的邏輯。

```
module rptr_empty #(parameter ADDRSIZE = 4)
        (output reg               rempty,
         output     [ADDRSIZE-1:0] raddr,
         output reg [ADDRSIZE  :0] rptr,
         input      [ADDRSIZE  :0] wptr_rclk,
         input                     rpop, rclk, rrst_n);
  reg  [ADDRSIZE:0] rbin;
  wire [ADDRSIZE:0] rgraynext, rbinnext;
  //---------------------
  // GRAYSTYLE2 pointer
  //---------------------
  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) {rbin, rptr} <= 0;
    else         {rbin, rptr} <= {rbinnext, rgraynext};
  // Memory read-address pointer (okay to use binary to address memory)
  assign raddr     = rbin[ADDRSIZE-1:0];
  assign rbinnext  = rbin + (rpop & ~rempty);
  assign rgraynext = (rbinnext>>1) ^ rbinnext;
//---------------------------------------------------------------
// FIFO empty when the next rptr == synchronized wptr or on reset
//---------------------------------------------------------------
assign rempty_val = (rgraynext == wptr_rclk);
  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rempty <= 1'b1;
    else         rempty <= rempty_val;
endmodule
```


下面我們來討論幾個面試中常見的問題。

問題：假設 wclk 速度比 rclk 快，那麼當 raddr+1，再同步到 wclk 後，如果這期間有了 push 操作，那會不會使得 wptr 超過了 rptr，造成 FIFO overflow 呢？

回答：不會，當 rptr 在傳過去之前，如果 wptr 已經追上了 rptr-1，那麼 wfull 已經是 1 了，FIFO 是不允許在 FIFO 滿的時候進行 push 操作的（在實際工程中我們通常要利用 assertion 來 check 保證在 wfull 為 1 的時候 push 不能為 1）。而如果這個時候有了 pop 操作，raddr+1，這個時候實際上 FIFO 有了一個 free entry，但是 push 這一側看到的 FIFO 依然是滿的。這就是我們所說的非同步 FIFO 的假滿。相應的，FIFO 的 empty 為 1 時，也可能 FIFO 此時有個 push 操作，導致 FIFO 為假空。假空和假滿並不會影響 FIFO 的正確性，無非就是早一點告訴 push side 停止 push，或者早一點告訴 pop side 停止 pop，但是 FIFO 是不會產生 overflow 和 underflow 的。如果要說有什麼缺點的話，就是在性能上有一些損失，當 FIFO 的深度很大的時候，這通常不是什麼問題。

問題：如何判斷 FIFO 是真空/真滿呢？

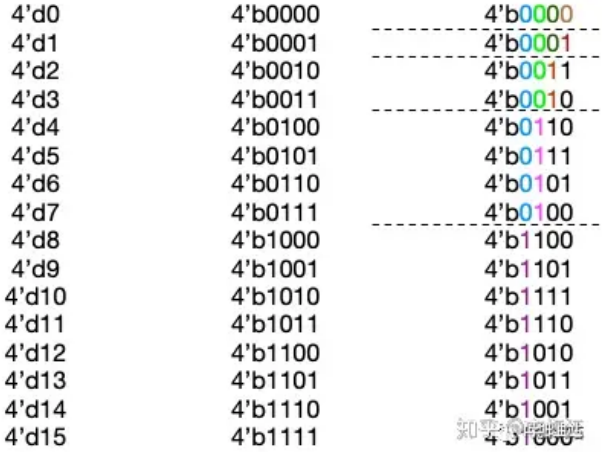
回答：判斷假空假滿剛好相反，在 push side 我們來判斷空，在 pop side 來判斷滿，為什麼要這樣留給大家自己思考。

問題：設計一個 depth=1 的非同步 FIFO

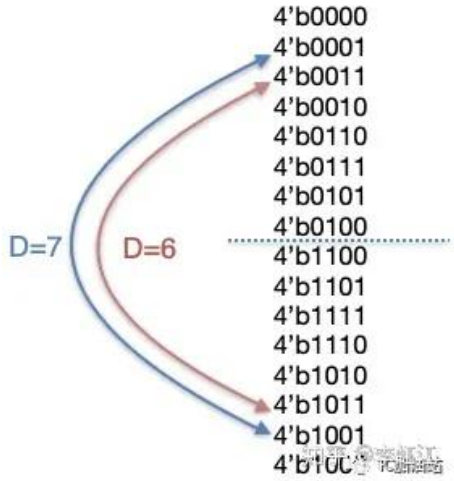
回答：這個大家就是要活學活用了，不能死板套用。只需要考慮一個問題，只有 1 個 entry，那麼需要幾位元的 address 或者 pointer 呢？當然是 1 位就夠了，那我們真的還需要一個 pointer 嗎？因為只有一個 entry，當一次 push，FIFO 就滿了，一次 pop，FIFO 就空了。1 個 bit 用來表示滿和空就足夠了。其實這樣的 FIFO 我們已經見過了，老李在多 bit 信號跨時鐘域怎麼辦？-- CDC 的那些事（4）裡面講到的帶回饋的 asynchronous load 其實就是 depth=1 的非同步 FIFO！

問題：如何設計 depth 不是 2 的冪次的非同步 FIFO？

回答：我們在上一講裡面看到的 gray code，只有當 depth=2 的冪次個數的時候，才能做到 wrap around 時繼續保持 gray code 的性質：即連續兩個碼之間只有 1 位不同。下面這個圖是表示 depth=8 的時候我們利用 16 個 gray code 來表示 pointer。



比如從 4'd15 到 4'd0，也只有 1 位不同。但是如果不是 2 的冪次，比如 DEPTH=7，那我們怎麼樣來利用 Gray code 呢？直接從 4'b0000 到 4'b0101 肯定是不行的，因為 4'b0101 變到 4'b0000 有兩個 bit 發生了變化，這樣我們就沒法利用 2 flip flop synchronizer 來同步了。解決這個辦法的訣竅其實就是老李上一篇提到的 gray code 的第二個性質：gray code 每一位是有個對稱軸的。我們可以這樣編碼，addr==0 的時候 gray code 不從 4'b0000 開始，而是從 4'b0001 開始，直到 4'b1001 來 wrap around，這樣從 4'b1001->4'b0001 依然只有一個 bit 翻轉。同理，如果是 depth=6，那麼我們繼續往裡收縮 1 位，只利用 gray code 關於對稱軸兩側的部分編碼，從 4'b0011 到 4'b1011，我們可以看到，這樣的編碼依然可以保證相鄰兩個碼之間只會有 1 位變化。



注意，利用這種編碼，FIFO 的滿判斷邏輯就不是簡單的高兩位取反，低位相同了，比如 depth=7, rd_ptr=4'b0001, wr_ptr=4'b1100 表示 7 個 entry 已經滿了，如何得出正確的滿判斷邏輯就交給大家思考吧，歡迎大家在評論區留言討論。

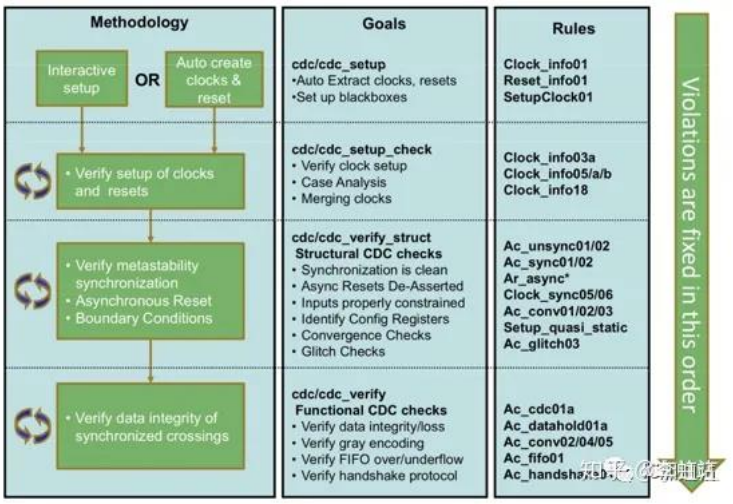
7.0 乾貨大放送之 CDC 工程經驗總結--CDC 的那些事（7）完結篇

這一篇老李給大家簡單介紹一下工業界常用的 CDC 檢查工具 Spyglass，然後奉上 CDC 設計和驗證中的工程經驗總結。如果你已經熟悉 Spyglass CDC，那麼你可以跳過第一部分。全篇乾貨滿滿，總計三千多字，希望大家一定能夠讀到最後，歡迎點贊以及分享朋友圈。

一、CDC 檢查工具

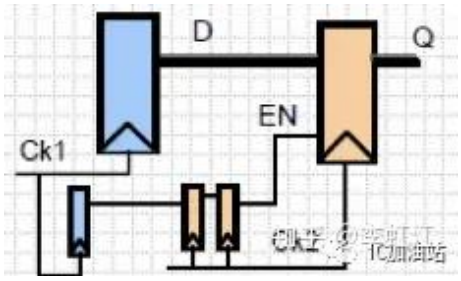
我們先來說 CDC 檢查工具。業界三大 EDA 公司 Synopsys, Cadence, Mentor 都有各自的 CDC 工具：Synopsys Spyglass CDC®, Cadence Conformal Constraint Designer®, Mentor Questa®. 而這其中以 Synopsys 公司的 Spyglass CDC 最為常用，市場佔有率最高。在很多公司裡都以 Spyglass CDC 作為 Sign-off 的標準。

Spyglass CDC 的基本功能是它已經定義好了一系列的設計規則(rule)，然後基於這些規則來對設計進行檢查，看設計是不是滿足了這些規則，如果有違反規則的設計，它會把這些違反的地方報告出來讓設計人員進行 debug。不同的規則被劃分為幾個大類，稱之為 goal。舉例來說，首先它會對設計和 constraint 進行設置檢查(setup rule check)，比如是不是每一個 flip flop 的時鐘都有定義，reset 信號有沒有定義等等。當 setup rule check 過了之後，它會進行一系列的 CDC 檢查，比如是否有信號跨時鐘域但是沒有經過 synchronizer 等等。每一個 rule 下面所有的不滿足規則的設計它都會報出來，可以通過 schematic, spreedsheet, text report 的形式呈現給設計人員，同時會有詳細的規則介紹以及為什麼它認為這個地方的設計違反了該設計規則，並且給出了相應的修改建議。



上面這個圖列出了 Spyglass CDC 的檢查流程，從 setup 開始，進行 setup check，之後再進行 structure check，最後再進行 functional check。structual check 是靜態的，工具會直接分析設計的有沒有滿足基本的跨時鐘域規則，但是工具是不理解這個設計的功能是什麼。而 functional CDC check 需要更進一步理解設計的功能，從實際功能運行的角度進一步分析設計，找出設計中的缺陷。

絕大部分情況下，structure check 就可以找出大多數的 CDC 問題。那為什麼還要進行 functional check 呢？舉個例子，如下面的電路



Q flip flop 是有一個 enable 的，這個 enable 信號是來自 clk1，並且用 2 flip flop synchronizer 進行了同步，在 structure check 中，這個 enable 信號會被當做 qualifier，工具會認為這是設計者有意為之，並不會報錯。因為很可能設計者的意圖就是利用這個 enable 信號來杜絕 D 信號的 CDC 問題。回想一下，我們在講多 bit 信號同步裡面其實就是用這個思路的：即同步一個單 bit 的 load 信號到 clk2 去，只有 load 信號為 1 時，Q 才會采 D 的值。但是注意，利用這種方法來同步有個假定：即在 clk1 域，enable 信號為 1 的時候，D 必須是穩定的。如果 D 不穩定，Q 依然會產生 metastable 的問題。而 structure check 是無法得知 D 在 EN 為 1 的時候是否穩定，必須要進行 functional 的分析才能進一步得知 D 是否穩定，如果不穩定則會報出 Ac_datahold01 錯誤。

和其他檢查工具一樣，我們不僅要給工具提供我們的 RTL design，還要提供一個 constraint file 來告訴工具一些 design 的基本資訊，比如 clock，reset 都是什麼，以及告訴工具一些分析 CDC 時要用到的資訊，這樣產生的最終分析結果才能有效。Sypgass CDC 的 constraint file 我們稱之為 sgdc。相信很多人已經瞭解 sdc， sdc 全稱是 synopsys design constraint, 主要是定義 design

的 clock, clock relationship，case analysis 等等，主要用在綜合和 STA 工具裡。對於 CDC 檢查來說，光有 clock 的定義還不夠，通常還要有 reset 相關的資訊，而且用作 timing 分析的 case analysis 以及 false path 並不能直接適用在 CDC 的 check 上，這些資訊都要在 sgdc 裡提供。

在 structure check 裡，大家最常見的 violation 有以下幾種：

Ac_unsync01/02

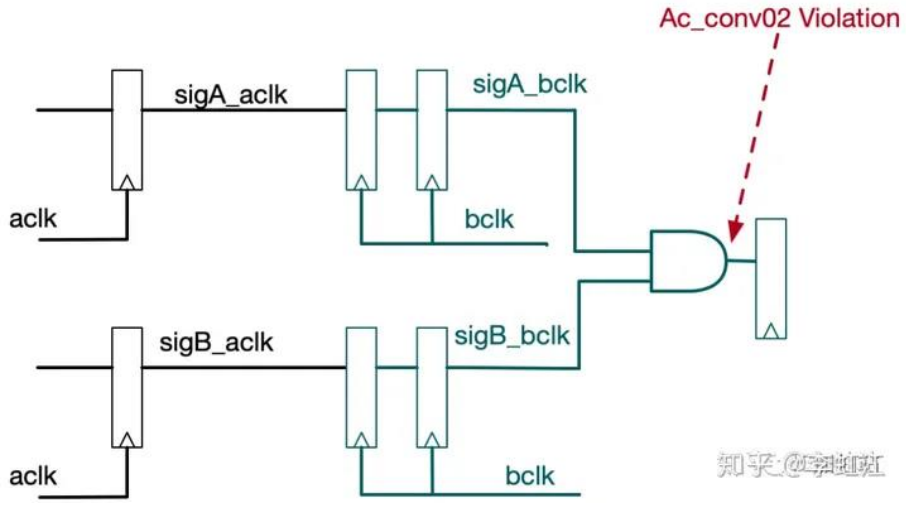
Ac_unsync 簡單來說就是該同步的信號沒有進行同步，這是 CDC 裡最重要的 rule，所有 Ac_unsync 的 violation 都不應該 waive，而是要仔細分析 design，來給信號加上適當的 synchronizer。Ac_unsync 分為以下兩個子類：

Ac_unsync01 ：一個單 bit 信號跨時鐘域，但是工具發現並沒有進行同步

Ac_unsync02: 一個 vector 即多 bit 信號跨了時鐘域，但是工具沒有發現進行同步

Ac_conv02

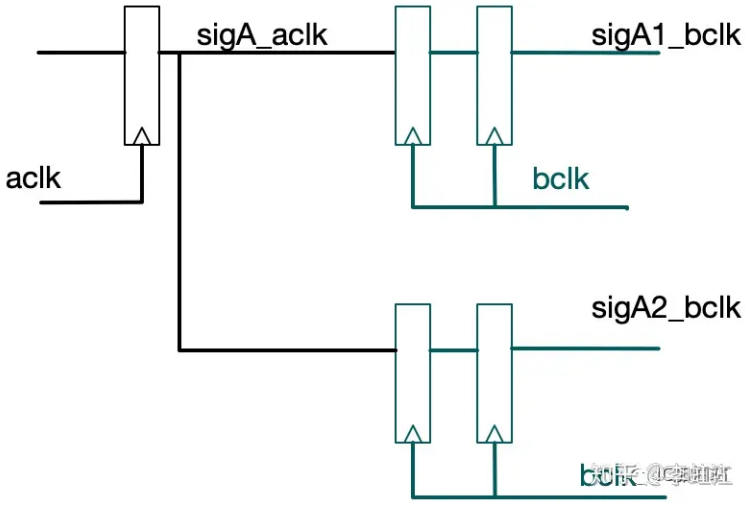
Convergence Violation 也是在 CDC 分析中非常常見的一類 violation。設計人員通常會記得給每個跨時鐘域的信號加 synchronizer，但是很多人一旦加完 synchronizer 之後就以為萬事大吉，在 destination clock 域對信號隨便進行操作了，從而容易產生 convergence 的問題。舉個簡單例子，單 bit 信號 sigA 和 sigB，用 2 flip flop synchronizer 同步到 bclk 域，然後進行 AND 操作，會有什麼問題呢？其實問題和多 bit 信號利用 2 flip flop synchronizer 產生錯誤是一個道理: 由於 2 flip flop synchronizer 有隨機性，當 sigA_aclk 和 sigB_aclk 同時變化時，sigA_bclk 和 sigB_bclk 可能會出現不期望的組合。



有些時候當 tool 報告了 Ac_conv02，但並不是必須要修改 design。比如說如果設計本身可以保證 sigA_aclk 和 sigB_aclk 不會在同一個 cycle 變化，即它們本身滿足 gray code 的特性，那這個電路其實是沒有問題的。我們可以給 CDC tool 加 gray_signals 這個 command 來告訴工具這裡沒有問題。當然，每一條 Ac_conv02 的 violation 都要仔細分析。

Ac_coherency06

Ac_coherency06 也是一個常見的 violation，指的是同一個信號在同一個時鐘域裡被同步了多次。特別是 design 變得比較大，有多個設計人員來做一個系統，不同人負責不同模組的時候容易出現這個這個問題。因為可能你看到一個信號到了你設計的時鐘域，你加了 synchronizer，而這個信號可能還去了別人的模組，別人也加了 synchronizer，但是你不知道，而你們兩個模組恰好又是同一個時鐘域，這個時候對整個大的系統進行分析的時候就會報出這個 violation。（注：這個 violation Spyglass CDC 默認是 warning, 不是 error，但是還是建議大家看這個 violation）CDC 有的時候就是這麼有意思，你不同步不行，你同步多了也會出問題。



首先，由於用了多個 synchronizer 產生了浪費，其次，依然是由於 synchronizer 的隨機性，導致很可能 sigA1_bclk 和 sigA2_bclk 在不同的 cycle 發生了變化，相應的後級電路也可能因為它們變化不一致導致出錯，因為從 clean design 的角度，既然是同一個信號，那麼就不應該不一致。這種 violation 通常也比較好修，拿掉多餘的 synchronizer 就可以了。

Ac_glitch02

Ac_glitch02 也是非常常見的 violation，原因就是 synchronzier 的 input 不是來自於 flip flop，而是 combo 電路的輸出。這個老李在你真的懂 2- flip flop synchronizer 嗎-- CDC 的那些事（2）裡已經講過了。

當然，Spyglass CDC 還包含了其他上百個 rule，更多的細節有待大家在工作中自己去學習，看文檔掌握。

二、CDC 工程經驗總結

以下經驗都是老李在工作中向前輩學習到以及自己總結的經驗，每一條單獨拿出來說都不複雜，希望大家可以參考。

設計前三思：確實需要跨時鐘域嗎？

我們在設計之前，其實要問自己，這一個功能我能不能不跨時鐘域就實現呢？或者仔細想一想能不能將邏輯簡化呢？比如可以將多 bit 信號簡化為單 bit 信號再同步來省下一些 synchronizer， 還要注意看是不是在別的地方已經對相關的信號進行了同步來避免 coherency

的問題。提前想好為什麼要跨時鐘域並進行恰當的設計，要遠比在設計後期來一個一個去修 CDC 的問題要有價值的多。

一個 module 包含一個 clock domain

在設計時，儘量把相同時鐘域的邏輯放在同一個 module 內，這樣對這個 module 可以認為是 synchronous 的設計。老李比較喜歡把所有的 synchronization logic 放在一個單獨的 module 裡，這樣做的好處就是 CDC 的分界很清晰，以後 debug 以及修 cdc 的問題就只需要改這一個 module 了。

同步時不能根據 source clock 的頻率和 destination clock 的頻率關係來同步，而是要假定它們之間的關係是任意的。這樣才能保證設計的健壯性和魯棒性。

要例化已經驗證過的 CDC synchronizer library，比如 2 flip flop synchronizer, pulse synchronizer, async FIFO 等等，成熟有一定積累的設計公司都已經有這些模組，在你的設計裡直接例化它們，千萬不要自己在 RTL 去造輪子。使用這些 library 對於其他 flow 來說也有幫助，比如說 STA 檢查中可以直接將 synchronizer 上的 path 設為 timing 的 false path，這個時候可以直接利用 synchronizer 的 module name 把這些 cell 找出來，非常方便。

所有的 IP 的 output 必須來自一個 register，禁止將組合邏輯的輸出當做 IP 的輸出。

一個信號同步到另外一個時鐘域只能同步一次，禁止同步多次，否則會有 coherency 的問題。

將要跨時鐘域的信號標記上 clk，要選擇個統一的 naming style。比如 req_aclk, req_pclk 分別表示 req 信號在 aclk 和在 pclk，這樣可以使得 code 清晰可讀。

在功能模擬驗證中，test bench 要對 2 flip flop synchronizer 來 deposit 隨機的值，來模擬 flip flop 在產生 metasability 之後可能穩定在 0，也可能穩定在 1，用這種方法能夠發現一些隱藏得很深的 bug。

對於非同步的時鐘，在 testbench 裡要使它們的沿要錯開，而不要讓他們的沿對齊。特別是有的時候兩個 clock 的頻率相同，但是如果它們是非同步的，更要讓沿錯開，或者說要隨機化它們之間的相位差。

在 Spyglass CDC 的 check 中

要首先 fix 所有的 setup error，之後再看 CDC violation。因為很可能 CDC 的 violation 是由於設置不正確造成的。

當有大量的相似的 CDC error 出現時，要考慮是不是 setup 沒有設置正確，比如 clock 定義錯誤，quasi_static 沒有 define 等等。

能從設計上修改最好，儘量減少 cdc 的 waiver。

不要相信 tool 給你找的 qualifier，要麼你自己指定 qualifier，要麼把 tool 幫你找到的 qualifier 從而認為 path 沒有問題當做是有問題來進行分析。

不要 waive Ac_unsync01/02 的 violation，一定要從 design 上來 fix。

如果不打算 fix design，那麼也不要用 waiver，考慮下面幾種辦法

set_case_analysis: 這樣 tool 就只分析當這個信號固定在 0 或 1 的情形，通常我們會把 MUX 的選擇端固定在某個值來去反映實際運行時的情況。

quasi_static: 這樣 tool 就會認為這個信號是不會 toggle 的

qualifier: tool 可以利用這個 quailifer 來去看 multi-bit 的 signal 是不是被這個 qualifier 來控制住從而不產生 metastable。

cdc_false_path: 如果並不存在這樣的 functional path，比如一個信號去了 DFT 的邏輯但沒有被 synchronize。

當然要慎重使用 quasi_static, cdc_false_path, set_case_analysis，除非你有充足的理由。

對於較大的 SoC，如果直接對整個 design 來跑 CDC 可能需要很長時間，而且最後產生的 violation 太多，非常不便於 debug，這個時候可以考慮利用 hierarchical flow。

對一些小一些的模組先進行 spyglass cdc check，把 violation 都清乾淨之後產生出一個 abstract model。abstract model 其實就是一個 sgdc file，它描述了這個模組的 CDC 相關的所有資訊，比如所有的 input output port 是在哪個 clock domain 上的，是 combo 的輸出還是 flip flop 的輸出等等。Abstract model 還包含了這個模組的 waiver。

在上層模組 run cdc 的時候讀入下層的 abstract model，好處就是可以節省 run time，同時下層模組內部的 violation 也不會報出來，這樣可以使得最終的 report 可讀性比較好。

當然 abstract model 也不是萬能的，有的時候甚至會掩蓋掉一些真正的問題，所以使用 abstract model 時要謹慎。

8.0 Reset 信號 如何同步？

這周老李帶大家補上一個 CDC 裡常考的基礎知識點：Reset 信號如何同步。

首先來複習一個更加基礎的概念：同步 reset 和非同步 reset。

同步 reset(synchronous reset)是說，當 reset 信號為 active 的時候，寄存器在下一個時鐘沿到來之後被重定，時鐘沿到來之前寄存器還是保持其之前的值。

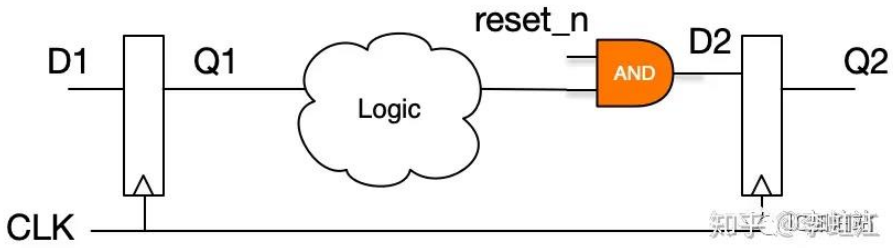
非同步 reset(asynchronous reset)是說，當 reset 信號為 active 的時候，寄存器立刻被重定，與時鐘沿到來與否沒有關係。

注意這裡老李沒有說 reset 信號為 1 的時候，而是說 active，因為有的時候是為 1 能夠使寄存器復位，這個時候我們說 high active，而有的時候是 0 能夠使寄存器復位，這個時候我們說 low active。

同步 reset 和非同步 reset 的區別算是數位晶片設計的入門知識點，一般第一輪面試就會考察。如果這個問題回答不好的話，那麼你在面試官心裡的印象一定是低於平均值的。老李在這裡帶大家簡單複習一下，如果你對下面這些內容已經爛熟於心，就直接跳過吧。

首先同步 reset 和非同步 reset 最主要的區別，從定義上就可以看得出，同步 reset 需要時鐘，而非同步 reset 不需要時鐘。如果說你的模組需要在沒有時鐘的時候重定，那只有非同步 reset 能夠做到，這也是絕大多數晶片的上電重定信號（PowerOn Reset）以及一些 PHY 比如 USB 的內部需要非同步 reset 的原因。而在一些 IP 中，如果你可以等到時鐘開始翻轉之後再重定，時鐘開始翻轉之前內部即使沒有復位也沒有關係的話，那麼就可以用同步 reset。

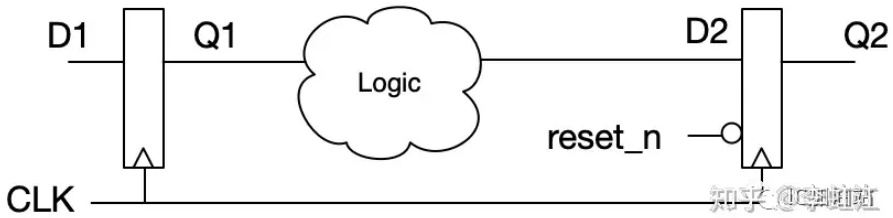
其次一個差別，即同步 reset 信號在綜合後，reset 信號和其他的 datapth 信號一樣，是一起算在兩個寄存器之間的 logic 深度裡，D 寄存器本身是沒有復位的 pin 的。而非同步 reset 信號通常會綜合出一個帶有復位 pin 的 D 寄存器。一般來說工藝廠家的 standard cell library 都會提供兩種不同的寄存器，只要你的 coding style 正確，綜合工具會選擇適當的 flip flop。



上圖是同步 reset 綜合出來之後的 netlist，可以看出 reset_n 使得兩級寄存器之間的組合邏輯多加了一個 AND 門。

```
always_ff @(posedge clk) begin
   if (!reset_n) begin
       q2 <= 1'b0;
   end
   else begin
      // q2 <= ...
   end
end
```

對於非同步 reset，綜合出來的 flip flop 自帶 reset pin，所以 reset 不參與中間的組合邏輯，如下圖 D2 所示



非同步 reset

```
always_ff @(posedge clk or negedge reset_n) begin
   if(!reset_n) begin
      q2 <= 1'b0;
   end
   else begin
      //q2 <= ...
   end
end
```

從綜合出來的邏輯可以看出，非同步 reset 由於對寄存器之間的 datapath 沒有貢獻，所以在 timing 上面能夠略微比同步 reset 好一些，特別是 reset 信號作為一個負載很大的信號，如果 reset tree 做得不好可能使得 reset path 的 combo delay 變得很大，反而限制了 performance 的提高。所以在對 logic depth 摳得很細的設計中，可以使用非同步 reset 來避免引入更多的 combo delay。

但是同步 reset 還有一個優勢，由於 reset 信號會最終起作用在寄存器的 D 輸入端，那麼通過 reset 的組合邏輯都會被 STA 所約束，也就是說 reset 信號和其他 datapath 的信號一起要滿足寄存器的 setup time, hold time, min pulse 等一系列 check，在 timing close 的情況下我們可以拍著胸脯保證：寄存器不會因為 reset 信號的變化產生 metastable。(所以同步 reset 信號的跨時鐘域咱們就不廢話了）可以對於非同步 reset 就沒有這麼簡單了，既然是非同步，那麼就是在任何時候都可能變化。現在 STA 之所以叫 static timing analysis，是因為工具是靜態分析電路的：給定一個時鐘沿的起始點，然後後面每一級的 delay 都是純粹的累加，最後再和 required time 來進行比較，比 required time 早到，就是滿足 timing，晚到就是 violation。可是如果一個信號什麼時候來都無法確定，那麼就無法判斷這個信號的 datapath 上最後能否滿足 required time。換句話說，純粹的非同步 reset 在當前的 STA check 中是沒有辦法檢查的。

那麼怎麼辦呢？難道對於非同步 reset 信號就聽之任之放任不管嗎？當然不是，我們做 IC 的當然要對每一個細節都要研究清楚。我們這裡要分兩種情況：1. reset assertion; 2. reset release。老李直接上結論：

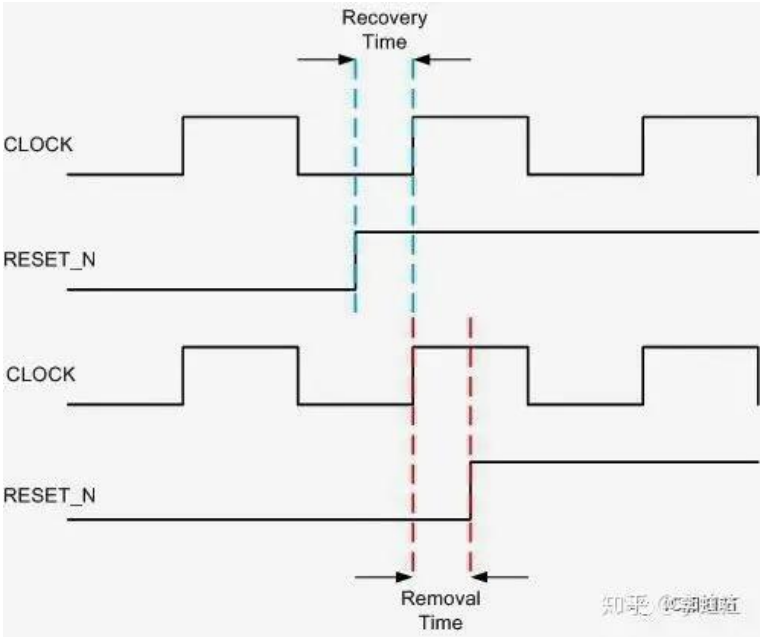
如果使用非同步 reset，reset assertion 是非同步的，但是 reset release 一定要和時鐘同步！

因為對於 reset assertion，reset active 之後 flip flop 的值是穩定在 reset value 的，只要 reset 繼續 active，來多少個 clock，其他 datapath 上的信號怎麼變，flip flop 的值都不會變化，所以 reset 在什麼時候 assertion 都沒關係。但是 reset release 就不一樣了，一旦 reset 從 active 變為 in-active，那麼 flip flop 之後的值就得取決於其他信號輸入和時鐘沿的關係了，所以對於 reset de-assertion，在 STA 裡有兩個專門的參數來 check，叫做 recovery time 和 removal time。

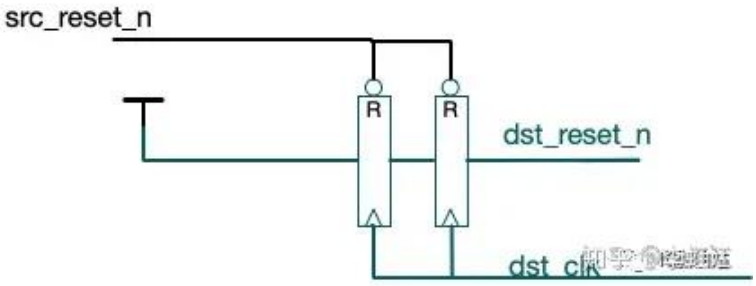
recovery time 指的是 reset release 之後要求距離下一個時鐘沿的最小間隔，可以類比於其他信號 datapth 上的 setup time。

removal time 指的是 reset release 之後要求距離上一個時鐘沿的最小間隔，可以類比於其他信號 datapth 上的 hold time。

換句話說，reset release 必須在 recovery time 和 removal time 加起來這個視窗之外，這樣才能保證寄存器不會產生 metastable。



好，下面的問題就變成了，我們如何設計可以使得非同步 reset 信號是非同步 assertion, 同步 release 呢? 終於要引出我們今天的主題，將一個非同步的 reset 信號同步到一個時鐘域，並且還要保證 assertion 是非同步的，但是 release 是同步於這個時鐘的，我們把這樣的電路叫做 reset synchronizer, 如下圖所示。



可以看到，當 src_reset_n assert 時，兩個 flip flop 被非同步 reset，它們的 Q 會經過 reset-to-q 的延時之後立刻發生變化，使得 dst_reset_n assert。

而當 src_reset_n release 後，dst_reset_n 並不是立刻發生變化，而是要等待 dst_clk 的時鐘沿，並且打兩拍之後才能將 1 傳遞到 dst_reset_n。因為 dst_reset_n 是來自於 flip flop 的 Q，而 Q 是經過 dst_clk 上的同步信號。那為什麼要兩級 flip flop，還是為了減小產生 metastable 的概率。

注意，對於一個使用非同步 reset 的模組，reset synchronizer 是必須的，但是也不要對一個非同步 reset 信號進行多次 reset synchronizer，否則同樣的會產生 coherency 的問題。

非同步 reset 在設計中還有一些其他需要注意的點，特別是在 DFT 中要特殊對待，還有非同步 reset 在 STA 中 timing constraint 要如何正確設置等等。老李以後抽時間再和大家聊。

9.0 面試題分析--獨熱碼檢測

今天老李帶大家分析一道數字電路面試題，這道題也是老李面試別人的時候喜歡問的一道問題。歡迎大家一起討論。

題幹很簡單：給定一個 4bit 的信號 A，設計邏輯來判斷 A 是不是獨熱碼，設輸出為 Y，如果 A 是獨熱碼，則 Y 輸出 1，如果不是，則輸出 0.

首先我們來看什麼是獨熱碼(one-hot)。獨熱碼是一種二進位編碼方式，它的特點是，用來編碼這個數的 N 位元 bit 中，有且只有一位是 1，其餘位元全部為 0。因為只有 1 位是 1，所以叫做 one-hot （對應的，還有一種編碼方式是只有 1 位元是 0，其餘位都是 1，叫做 one-cold)

其實 one-hot 編碼在電路設計中很常見。舉個最常見的例子，給定一個 SRAM 的位址，要讀出 SRAM 中的一行，SRAM 內部就是利用 address 的某幾位來轉換為 one-hot 的 select 從而選中對應的 word line 和 bit line。還有一些場合，利用 one-hot 來編碼狀態機，好處就是一個 flip flop 就表示一個狀態，用來判斷狀態機在哪一個狀態的時候就只需要看第幾個 flip flop 為 1 即可，而不需要像 binary 編碼那樣是所有 flip flop 在一起參與比較。這樣可以省下一點點邏輯的比較電路的延時，代價顯而易見，用 one-hot 編碼狀態機需要的寄存器比 binary 編碼要多，這就是一個典型的利用面積換性能的 trade off。

回到這個問題本身，其實直接回答上這個問題難度不大，相信所有學過數位邏輯的同學都能夠立刻想到解決辦法--利用真值表和坎諾圖。這也是面試官要考察你的第一個層次，對於學校課程的基礎知識掌握是不是扎實。

真值表咱們略過，直接上坎諾圖

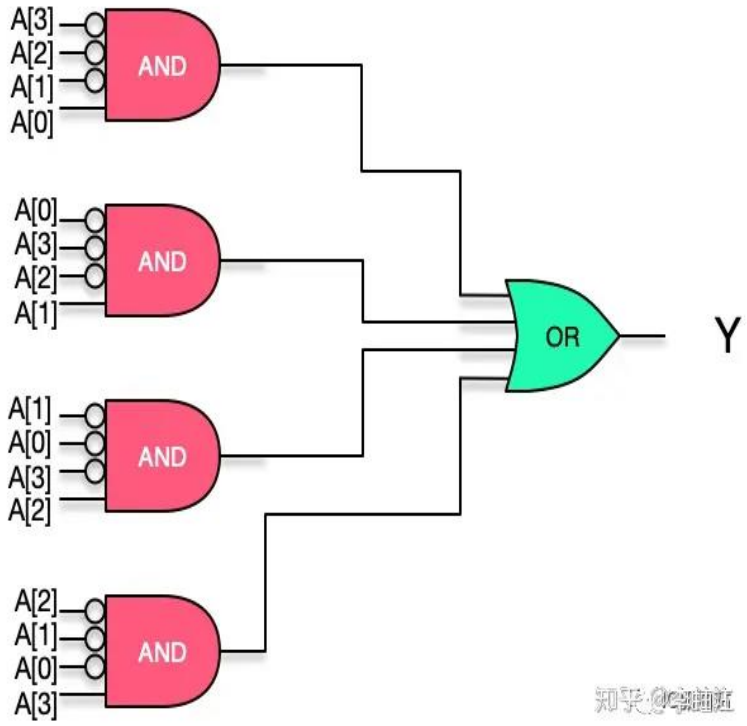
卡诺图

A[0]A[1] \ A[2]A[3]	00	01	11	10
00	0	1	0	1
01	1	0	0	0
11	0	0	0	0
10	1	0	0	0

很明顯，在這個例子中，坎諾圖並不能幫我們簡化邏輯（為什麼？），最後的運算式其實都可以直接想到

```
Y = (A[0] & (!A[1]) & (!A[2]) & (!A[3]))
    |((!A[0]) & A[1] & (!A[2]) & (!A[3]))
    |((!A[0]) & (!A[1]) & A[2] & (!A[3]))
    |((!A[0]) & (!A[1]) & (!A[2]) & A[3]);
```

這個時候面試官可能會問你，需要多少個邏輯門。關於上面的運算式，假設我們有四輸入的及閘和四輸入的或閘，而且假設輸入可以取反，那麼我們就需要 4 個及閘，1 個或閘，如下圖所示：



回答至此，面試官可以認為你在數位電路的課沒有完全睡覺，至少最基本的知識你是掌握的。能夠回答出這個答案不一定會給你 offer，但是如果回答不上那肯定是對不起謝謝了。

那麼接下來老李會怎麼問呢？如果輸入變成了 16bit，甚至 32bit 或者更多，你要怎麼設計電路呢？如何用 verilog 來表示你的電路呢？

你當然可以回答繼續用坎諾圖和真值表，直接寫運算式，16 個最小項，再把它們或起來。可是你知道這不是我想要的答案。因為面試官想要你設計的，是一個參數化的表達，參數化的設計是數字邏輯設計中很重要的一個點，它體現了你的設計是不是可以複用，而且能夠匹配各種應用需求。以這個為例，面試官想要你設計的其實是下面這個 function

```
parameter WIDTH = 16;
function is_onehot (input[WIDTH-1:0] sig)
//put    your code here
endfunction
```

那麼讓我們想想該如何解決這個問題。其實有個很簡單的思路，就是從獨熱碼的定義出發：只有一位是 1，其餘位都是 0。那麼不管我輸入信號有多少位元，有一個性質是不變的 -- 把這些位各自相加，最後的結果肯定得是 1。那麼我們就可以利用一個 for 迴圈，把每一位相加，最後再把最終的結果和 1 比較一下，如果是 1，那就是獨熱碼，如果不是 1，那就是其他的數，非常直觀吧？

當然，如果你告訴我這個思路，我會很樂於讓你寫一下 RTL code，code 不長，只要你能寫出來，沒有語法錯誤，那你在面試官裡的心裡又加了一分。下面是一個上面思路的參考版本

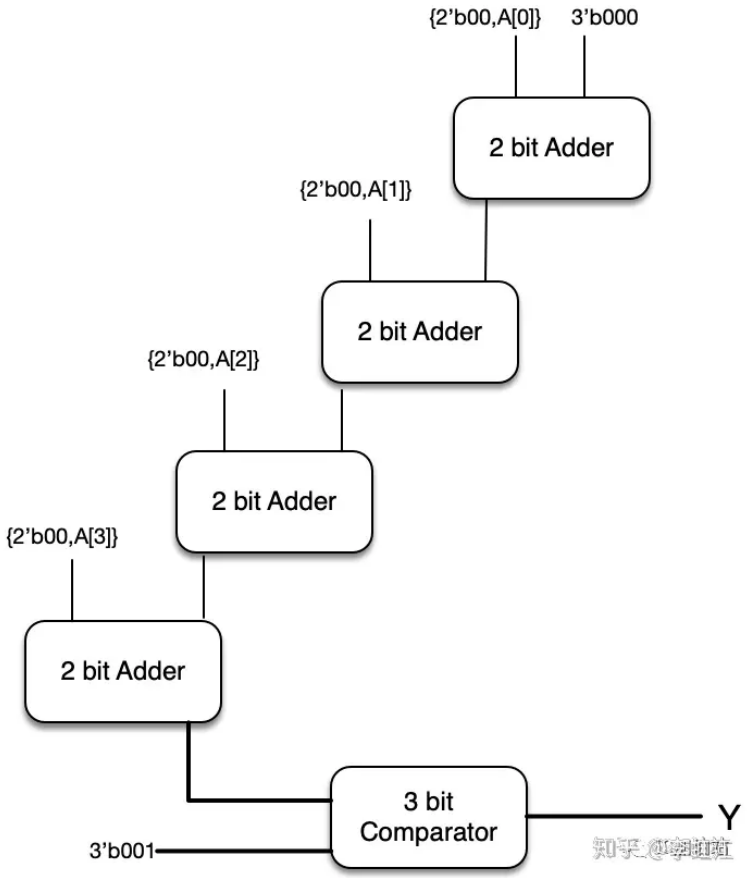
```
function automatic logic is_onehot(input [WIDTH-1:0] sig);
   localparam SUM_WIDHT = $clog2(WIDTH) + 1;
```

```
    logic [SUM_WIDTH-1:0] sum;
    sum = '0;
    for(int i = 0; i < WIDHT; i++)
        sum = sum + sig[i];
    is_onehot = (sum == 1);
endfunction
```

如果你能寫出上面的代碼，特別是能夠注意用到 automatic, $clog2 等等，說明你對 SystemVerilog 掌握得還不錯，是寫過幾行代碼的人。如果你是一個應屆畢業生，能夠寫出這樣的 code，並且能夠回答出我針對這幾行 code 的 follow up 問題，老李對你已經很感興趣了。這裡面有個小點需要大家注意，SUM_WIDTH 要用$clog2 之後再加 1，為什麼需要這樣留給大家思考。

那麼老李接下來就會問你，這段 code 綜合出來的電路是什麼樣的呢？這裡老李其實想考察你對 for loop 在 RTL 中實際理解。對於許多 RTL 的初學者，通常會搞不清楚 for loop 的作用，錯誤地把程式設計中的 for 迴圈的想法濫用在 RTL 設計中，從而很容易寫出看起來好像正確，但是實際無法綜合出電路的 RTL code。老李一再要和大家強調，RTL 是硬體描述語言，你寫的每一行 code 都是在描述一個電路，大家一定要做到心中有電路，手下才有 code。

上面的電路以 WIDTH=4 為例，綜合出的電路（優化前）其實是



很明顯，這樣的電路可以達到設計的目的，但是並不是最優的，大家可以計算一下每個 2bit

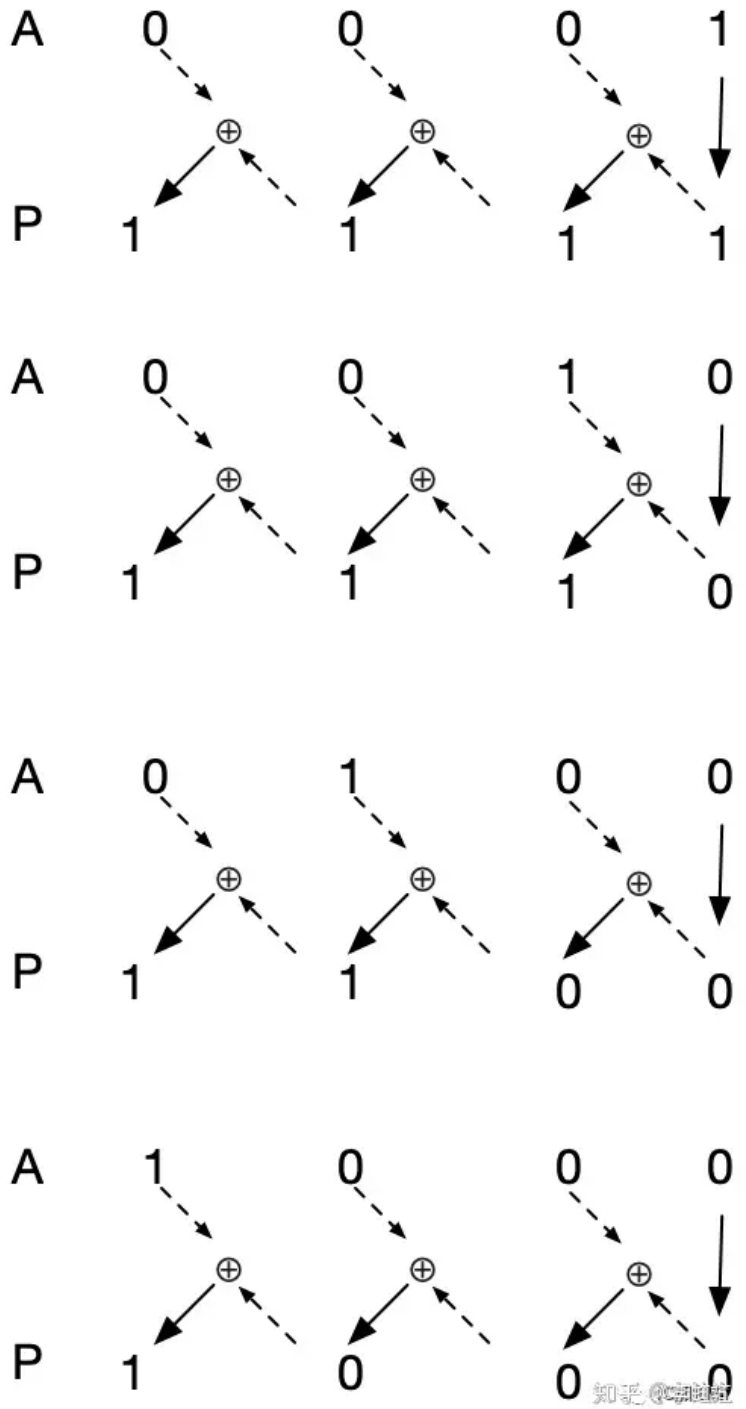
full adder 需要多少門，comparator 需要多少門，再和之前利用坎諾圖方法得出的最簡電路比較一下。

那麼有什麼辦法可以優化呢？當然如果你繼續對上面的思路進行優化，比如第一級，第二級其實不需要一個 2bit 的 adder, 而是可以將 a[0]，a[1]送到一個 half adder 裡面，這樣就可以替換掉前面兩個 2bit adder，還有 half adder 本質其實就是一個 XOR gate 等等... 這些都是我很樂於和你探討的，你探討的越細，在老李心中的加分越多。老李說句掏心窩子的話，你在面試的過程中是不是給出最優解並不關鍵，重要的是給面試官展現出你思考的過程，展現出你利用你現有的知識去一點一點解決未知問題的能力。一般來說，如果這道題你經過這麼一個思考過程，包括寫代碼畫電路圖還有和麵試官討論，你在這個問題上的表現已經在面試官心裡有譜了。

那麼這個問題再上一個層次要怎麼回答呢？其實思路來自一個稍微冷門但其實又非常常用的知識點：對一個 binary number 進行同位。

舉例來說，對於一個 4 位的二進位數字，我們對這 4 位同位，利用 XOR 門依次進行每一位，最後的結果如果是奇數個 1 那麼 4 位 XOR 之後結果就是 1，如果偶數個 1 那麼結果就是 0。看到了嗎，我們要找的獨熱碼其實是奇數個 1 的特殊情況，即只有 1 位元是 1。所以更加巧妙的方法就來自於這個按位 XOR。

我們再來看幾個例子，看能不能找出獨熱碼按位 XOR 的特點。還是以 4 位為例，我們定義如下運算，把 A 相應按位 XOR 的結果記為 P，其中 P[0] = A[0]， P[i] = A[i] ^ P[i-1]。



我們可以看到一個什麼規律？即對於獨熱碼，我們得到的 P 會是這樣一個數：如果獨熱碼 A 的第 i 位為 1，那麼 P 的第 i-1 到第 0 位都是 0，而第 i 位元和更高位都是 1。注意，這個規律只對獨熱碼適用，大家可以試驗一下其他任何數，只要多於 1 位是 1，那麼就不會出現高位連續的都是 1，高位必然會出現 0。（思考，什麼時候 P 的高位會出現 0？）

那麼我們就可以繼續觀察，如果我們將 A 按位取反，然後再和 P 按位 OR 起來會得到什麼？A 既然是獨熱碼，那麼除了為 1 的那一位（假設是第 i 位元），取反之後都會變成 1，只有第 i 位會是 0。而 P 的第 i 位是 1，那麼最後按位 OR 的結果是什麼？全部每一位元都是 1。

那麼如果 A 不是獨熱碼，而是有兩個 1 或者更多 1，假設第 i 位和第 k 位是 1(k 是 i 之後第一

個 1），我們進行上面的操作會得到什麼？P[k]會得到 0。A 反和 P 按位 OR 之後第 k 位也是 0.

至此，我們距離我們的最終答案只有一步之遙，A 反和 P 按位 OR 之後的結果我們再對每一位進行按位 AND，得到的結果如果是 0，那麼一定不是獨熱碼！

為什麼是一步之遙呢？我們這裡漏了一個特殊情況，即 A 為全 0。當 A 為全 0 的時候，P 也是全 0，但是 A 取反之後是全 1，所以 A 反和 P 按位 OR 之後也會得到全 1。幸好，特殊情況就只有這一種，我們只需要對 A 進行一下全 0 判斷就可以了。

至此，我們更加高效的通用解法就出來了

```
function automatic logic is_onehot(input [WIDTH-1:0] sig);
  logic [WIDTH-1:0] parity;
  parity[0] = sig[0];
  for(int i = 1; i < WIDTH; i++)
      parity[i] = parity[i-1] ^ sig[i];
  is_onehot = parity[i-1] & (&(parity | ~sig));
endfunction
```

終於至此，這道面試題算是講差不多了，說實話能夠回答出最後一種解法的人寥寥無幾，所以作為一道面試題，這道題的區分度其實並不怎麼樣。老李還是更看重面試者是否能夠順利地利用前面基本思路來給出一個可以 work 的解法。至於如果哪天遇到一個求職者給出了最後的解法，老李肯定對他是十分佩服的，肯定會給 strong hire。當然，老李在這裡寫這篇文章不是希望大家去背答案，而是去思考這麼一道看似簡單題的解題思路，還是那句話，你給出的答案其實不那麼重要，重要的是你的思考過程。

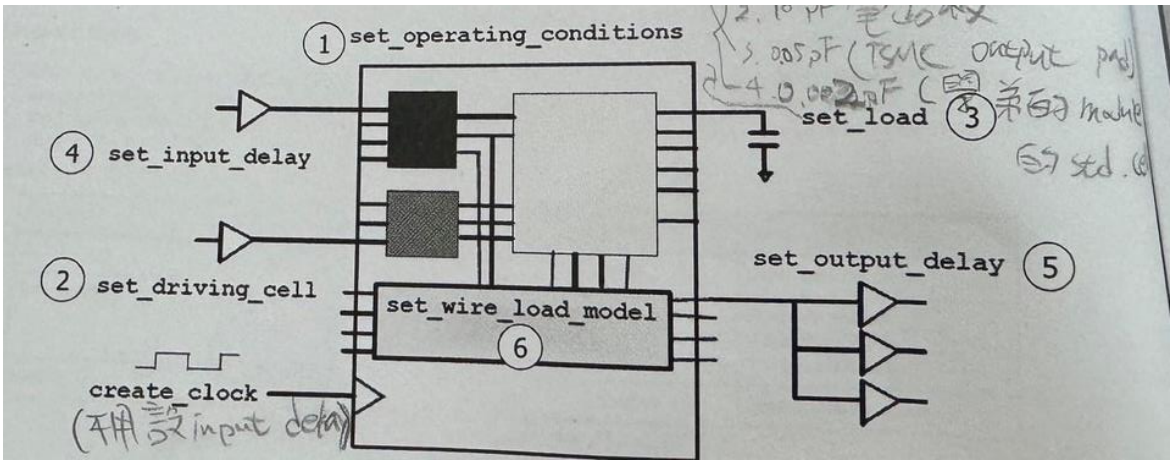
感謝閱讀到最後的你。如果你覺得老李的文章有用，歡迎點贊，搜索微信公眾號“IC 加油站”，老李最新的文章都在那裡更新。

時脈閘控（英語：Clock gating）是一種在同步序向邏輯電路的一種定時器訊號技術，可以降低晶片功耗。時脈閘控通過在電路中增加額外的邏輯單元、優化時鐘樹結構來節省電能。[1]

可以通過以下幾種方式在設計中添加時脈閘控邏輯：

- 通過暫存器傳輸級編程中的條件選擇來實現使能訊號，從而在邏輯綜合過程自動被翻譯為時脈閘控
- 通過實例化特殊的時脈閘控單元，來把時脈閘控插入到設計中去
- 使用專門的時脈閘控工具添加。

10.0 DC 問題-陳老哥筆記



設定七大法則

1. set_operating_conditions：設定 PVT(製程 process、電壓 volt、溫度 temp)，為了讓測試的時候可以有 worst 跟 best case，測試晶片在最壞或最好的狀況下都可以使用。有 slow.lib 跟 fast.lib(lib 給人看得、db 給電腦看的)。

   PPA(功率 Power、效能 performance、面積 area)

2. set_driving_cell：設定前一級的推動力。

   set_transition：V93000 是設定。

   set_drive：設定 pin 的電阻值，前一級是 I/O pad 的時候用。

3. set_load：設定電容值，給下一級的負載(傳送過去的推動力)。

   set_fanout_load：同上，但是以標準 load 為基礎。

4. set_input_delay：知道外部進來的時間，讓 input 可以順利吃到。(吃到代表沒有 setup 跟 hold time 問題)

5. set_output_delay：知道傳去外部的時間，讓 Output 被順利吃到。(吃到代表沒有 setup 跟 hold time 問題)

6. set_wire_load_model：設定模擬繞線會產生的 delay。

7. create_clock：clock 可能會遇到的問題，讓 DC 模擬(worst case)；clock 週期(要合出來的週期)、uncertainty(時鐘不確定性包含 skew、jitter) 、clock gating。

   時鐘抖動(clock jitter)：指芯片的某一個給定點上時鐘週期發生暫時性變化，使得時鐘週期在不同的週期上可能加長或縮短。(clock 實際抖動有可能會變快或變慢)

   時鐘偏移(clock skew)：是由於佈線長度及負載不同引起的，導致同一個時鐘信號到達相鄰兩個時序單元的時間不一致。

clock jitter 跟 clock skew 差別：jitter 是在時脈產生器內部產生的，和晶振或 PLL 內部電路有關，佈線對其沒有影響；skew 是由不同佈線長度導致的不同路徑的時脈上升沿到來的延遲不同。

時脈閘控（clock gating）：是一種在同步序向邏輯電路的一種定時器訊號技術，可以降低晶片功耗。時脈閘控通過在電路中增加額外的邏輯單元、優化時鐘樹結構來節省電能。

可以通過以下幾種方式在設計中添加時脈閘控邏輯：

- 通過暫存器傳輸級編程中的條件選擇來實現使能訊號，從而在邏輯綜合過程自動被翻譯為時脈閘控
- 通過實例化特殊的時脈閘控單元，來把時脈閘控插入到設計中去
- 使用專門的時脈閘控工具添加