**IC 设计学习笔记**

个人日常笔记

目录

[1 面试题集 8](#_Toc77586051)

[1.1 芯司机 8](#_Toc77586052)

[1.1.1 设计面试题 8](#_Toc77586053)

[1.1.2 验证面试题 9](#_Toc77586054)

[1.1.3 中端面试题 13](#_Toc77586055)

[1.1.4 DFT面试题 23](#_Toc77586056)

[1.1.5 后端面试题 25](#_Toc77586057)

[1.1.6 每日一题 25](#_Toc77586058)

[2 经典设计 27](#_Toc77586059)

[2.1 Sync FIFO 27](#_Toc77586060)

[2.1.1 标志位 27](#_Toc77586061)

[2.2 Async FIFO 27](#_Toc77586062)

[2.2.1 标志位 27](#_Toc77586063)

[2.2.2 深度 28](#_Toc77586064)

[2.3 CRC 29](#_Toc77586065)

[2.4 Encryption/Decryption 29](#_Toc77586066)

[2.4.1 AES/SM4 不同加解密模式的比较 29](#_Toc77586067)

[2.4.2 SM2 31](#_Toc77586068)

[2.4.3 测信道攻击防护 31](#_Toc77586069)

[2.5 跨时钟域处理--最终详尽版 32](#_Toc77586070)

[2.5.1 参考资料 32](#_Toc77586071)

[2.5.2 异步时序定义 32](#_Toc77586072)

[2.5.3 亚稳态 32](#_Toc77586073)

[2.5.4 单比特同步策略 34](#_Toc77586074)

[2.5.5 多比特同步策略 40](#_Toc77586075)

[3 名词释义 55](#_Toc77586076)

[3.1 CPU 55](#_Toc77586077)

[3.2 GPU 55](#_Toc77586078)

[3.3 NPU 55](#_Toc77586079)

[3.4 PHY 55](#_Toc77586080)

[4 接口协议 57](#_Toc77586081)

[4.1 PCIE 57](#_Toc77586082)

[4.2 DDR 57](#_Toc77586083)

[5 设计流程 & EDA\_Tools 58](#_Toc77586084)

[5.1 综合 58](#_Toc77586085)

[5.1.1 Design Compiler Flow 58](#_Toc77586086)

[5.1.2 随手记 58](#_Toc77586087)

[5.1.3 Timing Arc 60](#_Toc77586088)

[5.1.4 OCV 64](#_Toc77586089)

[5.1.5 SDC 69](#_Toc77586090)

[5.1.6 UPF 70](#_Toc77586091)

[5.2 Power 73](#_Toc77586092)

[5.2.1 功耗分类： 73](#_Toc77586093)

[5.2.2 Clock Gating 73](#_Toc77586094)

[5.3 DFT 74](#_Toc77586095)

[5.3.1 SCAN 74](#_Toc77586096)

[5.4 Verdi 75](#_Toc77586097)

[5.4.1 随手记 75](#_Toc77586098)

[5.5 PT 75](#_Toc77586099)

[5.5.1 PTPX 75](#_Toc77586100)

[6 Verilog 78](#_Toc77586101)

[6.1 实例分析 78](#_Toc77586102)

[6.1.1 always block中阻塞赋值与非阻塞赋值的执行顺序 78](#_Toc77586103)

[6.2 function和task 78](#_Toc77586104)

[6.2.1 Function 78](#_Toc77586105)

[6.2.2 Task 79](#_Toc77586106)

[6.2.3 Function和task的对比 80](#_Toc77586107)

[6.2.4 Glitch free clock switch 81](#_Toc77586108)

[7 Analog 82](#_Toc77586109)

[7.1 PLL 82](#_Toc77586110)

[8 附录 83](#_Toc77586111)

[8.1 SGDC Template 83](#_Toc77586112)

[8.2 SDC Template 83](#_Toc77586113)

[8.3 DC get\_\*\* Command Example 84](#_Toc77586114)

[8.4 Linux 85](#_Toc77586115)

[8.4.1 压缩解压缩 85](#_Toc77586116)

**FIGURE**

Figure 1‑1 看图画电路 7

Figure 1‑2 SRAM 控制器功能列表 7

Figure 1‑3 SVA 断言设计 8

Figure 1‑4 System Verilog 应用 9

Figure 1‑5 UVM 应用 10

Figure 1‑6 脚本处理 11

Figure 1‑7 clock 时序约束 12

Figure 1‑8 setup-hold分析。 13

Figure 1‑9 1.1.3.2 答案 13

Figure 1‑10 1.1.3.5 15

Figure 1‑11 1.1.3.6 16

Figure 1‑12 clock gating cell 18

Figure 1‑13 1.1.3.10 18

Figure 1‑14 DFT Fault Simulation 19

Figure 1‑15 ATPG D算法的应用 20

Figure 5‑1 Timing ARC 28

Figure 5‑2 Combinational Timing Arc Sense 29

Figure 5‑3 Sequence Timing Arc Sense-1 30

Figure 5‑4 Sequence Timing Arc Sense-2 31

Figure 6‑1 Coding style with circuit-always 37

Table 1‑1 SDC 约束时钟 17

# 面试题集

## 芯司机

### 设计面试题

#### 请描述你对芯片设计的了解

参考每日一题第1题、第13题。

#### 谈谈你所了解的低功耗设计方法和思路

参考每日一题第2题。

#### 如何理解Setup/Hold Time,为什么DFF有Setup/Hold time要求

参考每日一题第6题，第7题，第53题。

#### 何为同步复位和异步复位？何为异步复位同步释放

参考每日一题第15题。

#### 逻辑化简：用两输入或非门实现如下逻辑？B=en? Mask&A: A

参考每日一题第22题。

#### 异步FIFO代码实现

参考每日一题第25题，第28题，第51题。

#### 算法的代码实现：CRC算法。（g(x)=xˆ7+xˆ3+1,完成CRC7代码）

参考每日一题第29题。

#### 画出AHB incr4 burst读写传输时序图，请问incr4、wrap4的区别

参考每日一题第30题，第89题。

#### 看图画电路或者写代码

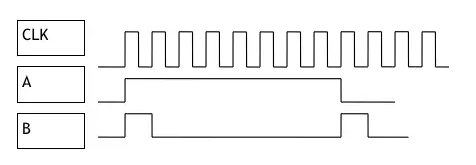


Figure 1‑1 看图画电路

参考每日一题第56题，第57题。

#### 状态机实现

在一个扔硬币的实验中，设扔到正面为“0”，扔到反面是“1”。请设计一个状态机，报告扔硬币过程中所有连续三次结果恰好为“正正反”出现的总个数。并用verilog写出该模块。

参考每日一题第59题，第62题。

### 验证面试题

#### SoC芯片验证包括哪些内容，各自目的是什么

参考每日一题第19题。

#### 如何制定验证计划？（根据功能列表制定验证计划）

#### 

Figure 1‑2 SRAM 控制器功能列表

参考每日一题第47题，第87题。

#### SVA断言设计：

根据下图和说明写出这个sequence的程序的断言，并统计这种情况是否在case中出现的次数。

图中addr和data为高表示accept信号拉高后的第三个clk开始addr和data信号不能为32’hxxxx，并且统计出accept信号变化的次数以及addr[7:0]的coverage。

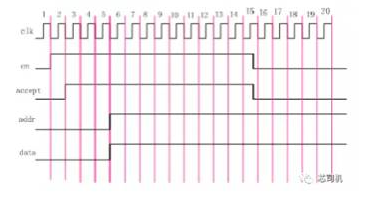


Figure 1‑3 SVA 断言设计

参考每日一题第74题。

#### System Verilog应用

根据表一和表二数据包的约束，在class中产生约束的数据，在程序中每次调用这个函数都产生一个新的数据包，并且产生后的包数据可以从另外一个class更改，不能用force，并且两个函数不能同时执行。

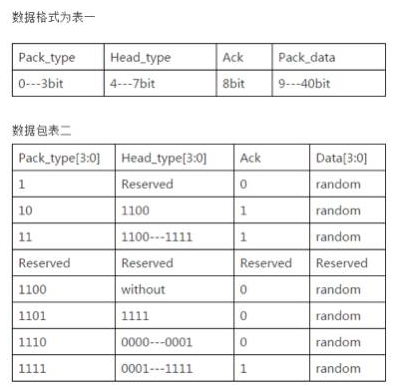


Figure 1‑4 System Verilog 应用

参考每日一题第77题。

#### UVM应用

根据如下总线的读写时序，用systemverilog语言写出总线的driver,并通过interface的方式在testcase中调用读写函数，并且使用UVM的方式把一个操作通过上面写的driver驱动给DUT。

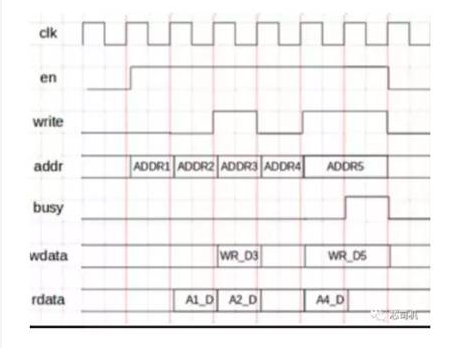


Figure 1‑5 UVM 应用

参考每日一题第78题。

#### 低功耗设计方法和思路

参考每日一题第2题，第54题。

#### 谈谈对静态时序分析和动态时序仿真的理解及优缺点

参考每日一题第4题。

#### 脚本处理



Figure 1‑6 脚本处理

参考每日一题第84题，第64题。

#### 画出典型的MCU（SoC）系统框图

参考每日一题第60题。

#### FPGA 内部资源有哪些

参考每日一题第43题，第5题。

### 中端面试题

#### 芯片A端口OUT1/OUT2分别接到芯片B的D/CLK。芯片A端口约束如下：

create\_clock -name CLK-period 10 -waveform {0 5} [get\_pins U1/Y]

create\_generate\_clock -name GCLK -source [get\_pins U1/Y] -edge "1 3 7" -master\_clock CLK [get\_pins U2/Y]

create\_generate\_clock -name OUTCLK -source [get\_pins U2/Y] -master\_clock GCLK -divide\_by 1 [get\_por

ts OUT2]

set\_output\_delay -max 8-clock OUTCLK [get\_ports OUT1]

set\_output\_delay -min 1-clock OUTCLK [get\_ports OUT1]

点评：这道题非常深刻的考察了同学们对create\_generate\_clock这条约束的掌握程度，题目中提到了-edge的用法，当然还有其他option，如-combination等。另外还考察了同时当clock做输出时，约束应该如何写的问题。

答：

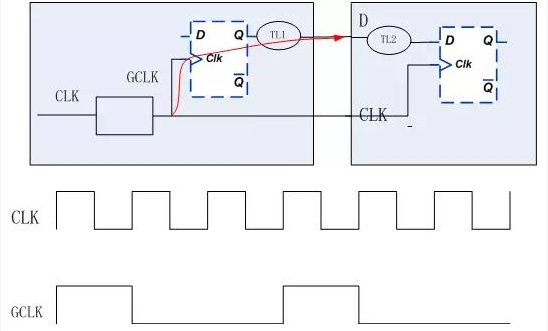


Figure 1‑7 clock 时序约束

电路结构和波形示意如下，由于使用了-edge 1 3 7 , 因此CLK和GCLK的波形如上，另外假设芯片A出口是GCLK域出来，那么：

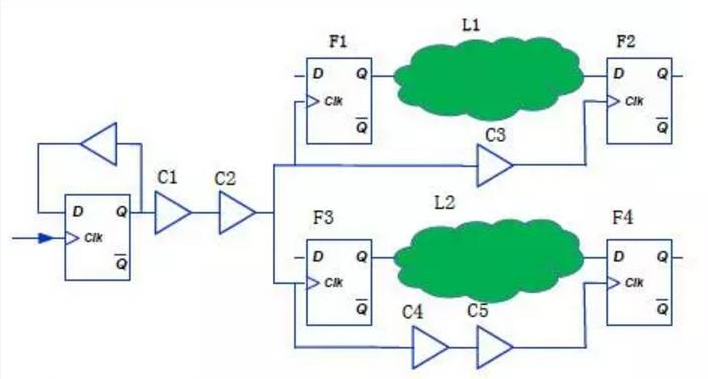
Q1：T(c->q) +TL1+TL2 +Tsetup  < Tcycle , 其中Tcycle =80ns, TL2+Tsetup = max output delay = 8ns

 T(c->q)+TL1+TL2> Thold , 其中TL2-Thold= min output delay

Q2: 波形如上图

Q3：第三问考察的是随路时钟的问题。一般都要做special timing check，保证data和CLK之间的skew =0. 如果要加约束，可以用set\_min\_delay和set\_max\_delay

#### from schematic and delay picture below, calculate the WC setup slack at D pin of F2 and BC hold slack at D pin at F4. input CLK period is 4ns。



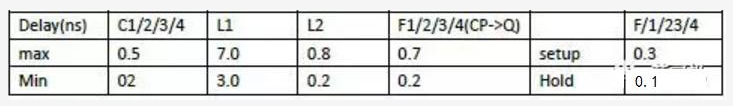
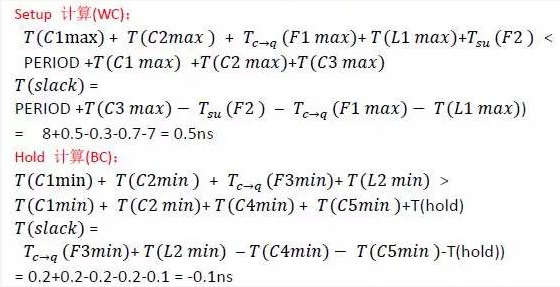
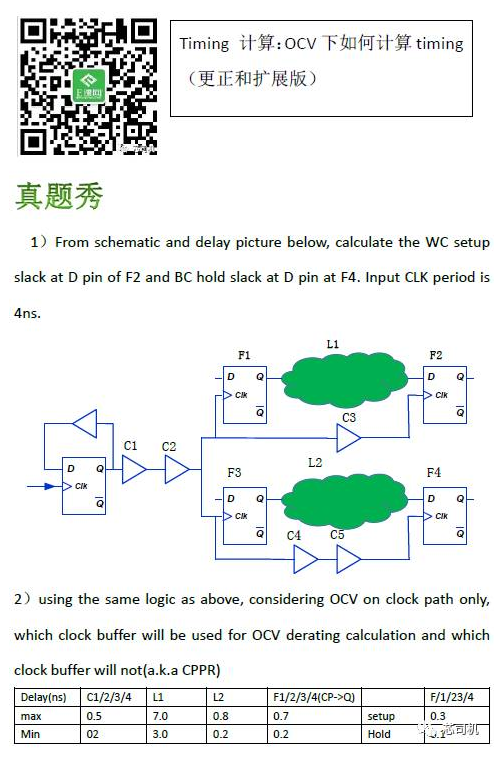


Figure 1‑8 setup-hold分析。



下面是这个问题的更正和补充。



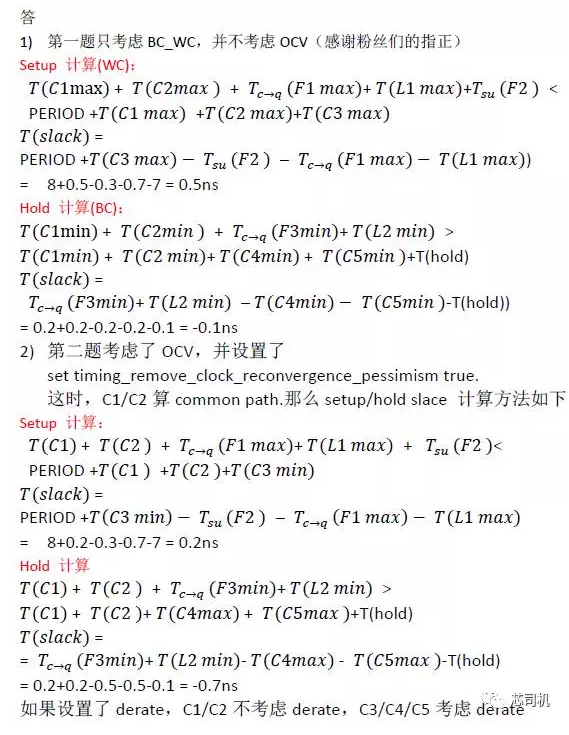
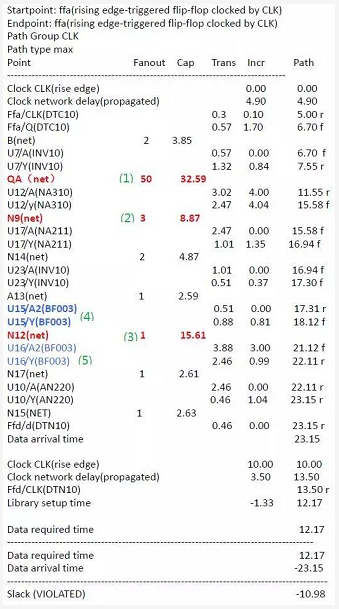


Figure 1‑9 1.1.3.2 答案

#### 下面的时序报告，报的是什么类型的时序违例，违例的原因是什么？修复建议是什么



点评：时序报告分析，不管是设计工程师还是DC实现，ICC实现。都需要非常精通，因此时序报告可以说是年年的必考题。

答：

1．这是setup timing check 报告；从path type = max可以看出来

2．上面绿色标记的6处是引起这次timing violation的最大原因。1）处是由于fanout太大，没有做buffer tree造成；2）处由于前面没有做buf tree从而导致U12输出的transition 非常的差，简介导致U17delay 变大；3）3处N12前面的BUF驱动能力太弱，导致U16 delay变大；4，5）2处的BUF是否可以合并，采用驱动能力比较强的BUF。

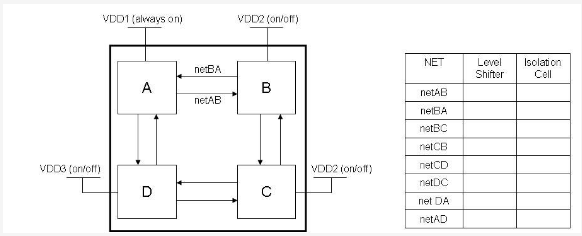
#### 请描述静态时序分析中2种不同的模式GBA和PBA的区别

点评：STA分析的最基本，最基础的概念，也是经常被忽略的。GBA和PBA到底有什么区别呢？

答： PBA compute path-specific cell and net delays and PBA compute path-specific slew degradation across nets and then re-compute resulting path slack. GBA use the worst input slew of a cell to calculate that cell’s output transition.

#### 低功耗设计相关

There are 4 power supplies in the design. VDD1/2/3 are different voltage. VDD1 is always on, but VDD2 and VDD3 can be turn off and on. VDD2 to block B and block C don't switch at sometimes. Please fill in the blank which net needs to be added level shifter and/or isolation cell. If you think Level shifter/isolation cell should be added in netAB at B side, then write B; if you think it's not necessary, write X.



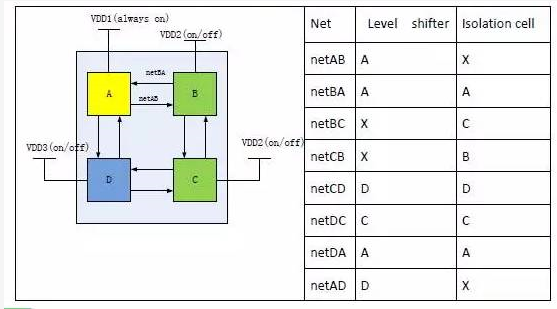


Figure 1‑10 1.1.3.5

根据答案总结：

1. 加level shift时，如果有一端不掉电的话，则首先应该加在always on block，否则应该加在信号终点侧。
2. 插isolation cell时，也应该插在信号终点侧。不断电区域output的信号不需要插isolation。

#### 请分析如下电路

1. 标注图中所有的timing path
2. 假设电路中相关延时如下所示，计算各条path的timing

所有逻辑门的最长的延时时间为3ns

所有连线（Net）最长的延时时间为2ns，最短为1ns

所有寄存器的clock到Q的延时时间为3ns

所有寄存器的setup time为1ns(Ts)

所有寄存器的hold time为1ns(Th)

所有Clock周期为14ns(Dclkp)

Clock source latency为2ns(Dclks)

Clock network latency 为3ns(Dclkn)

Clock uncertainty 为1ns(Dclku)

B及C的input delay 皆为1ns(Da, Db,Dc)

Y的output delay 为3ns(Dy)

1. 检查每条path的timing slack，确认是否有violation

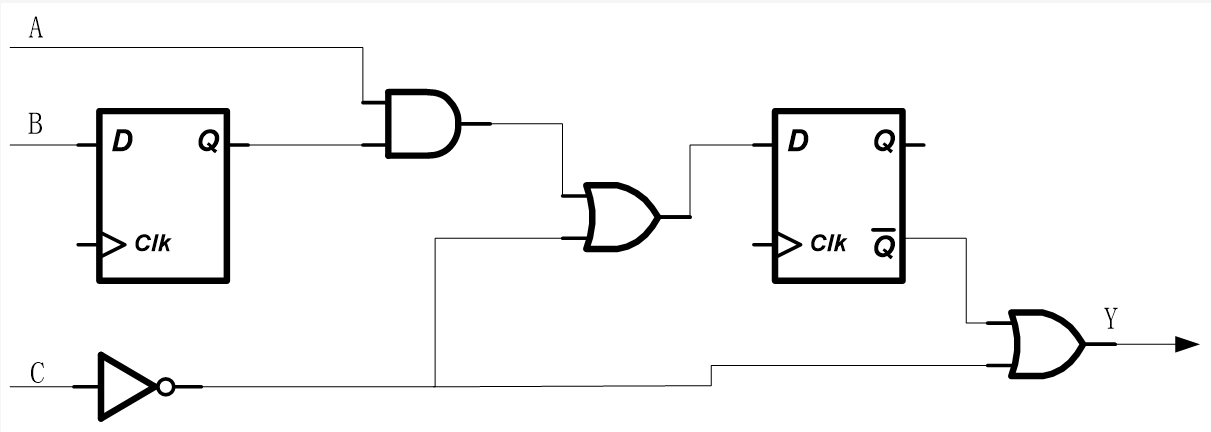


Figure 1‑11 1.1.3.6

点评：在有clk network delay的时候，我们该如何计算timing时序呢？

答：

Path1：T require =Tperiod+Tclk\_source +Tclk\_network-Tun –T setup = 14+2+3-1-1=17Tactual =2+3+2+3+2 = 12  slack = 17-12 5ns

Path2：T require =Tperiod + T clk\_source +T clk\_network –Tun-T setup = 14+2+3-1-1 =17 Tactual =Tclksource+Tclknetwork+3+2+3+2+3+2 = 20 slack = -3ns

Path3：Trequire =Tperiod –Ty = 14 –3 =11ns T actual = Tclk\_source +Tclk\_network+3+2+3+2 =15nsslack = Trequire-T actual = 11 -15 = -4ns

Path4：Trequire =Tclk\_virtual = 14  Tactual = 2+3+2+3 =10nsslack = 14-10 4ns

因此path2/3 有violation

#### 请用SDC，约束时钟的jitter，时钟的skew，时钟的latency，时钟的margin。DC和后端ICC在时钟约束上有什么样的区别？

Table 1‑1 SDC 约束时钟

|  |  |  |
| --- | --- | --- |
|  | PreCTS | Post CTS |
| transition | set\_clock\_transiton0.2 [get\_clockClk] | Propagate + input transition |
| latency | set\_clock\_latency1.2 [get\_clock Clk]  set\_clock\_latency –source 0.2 [get\_clock Clk] | Propagate +source latency  set\_propagate\_clocksClk  set\_clock\_latency–source 0.2 [get\_clockClk] |
| uncertainty | set\_clock\_uncertainty 0.5 [get\_clock Clk] | set\_clock\_uncertainty0.1[get\_clockClk] |

 点评：时钟的特性可以说也是最基本的。DC综合和ICC CTS在时钟约束上有很大的区别，也是考察读者的基本能力。

#### 请画出带测试功能的clk gatingcell电路图。并描述clk gating cell的时序检查特性

点评：clk gate cell在设计中及其实现中是非常常见的一个cell类型。因此，作为工程师，必须要深入了解它的特性。

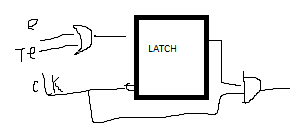


Figure 1‑12 clock gating cell

#### 9、上面的clock gate cell中的latch可以换成寄存器吗？为什么？

点评： ICG cell中为什么是latch，而不是寄存器呢？考察对问题的理解深入程度。

答：

#### 写出下面电路的SDC约束

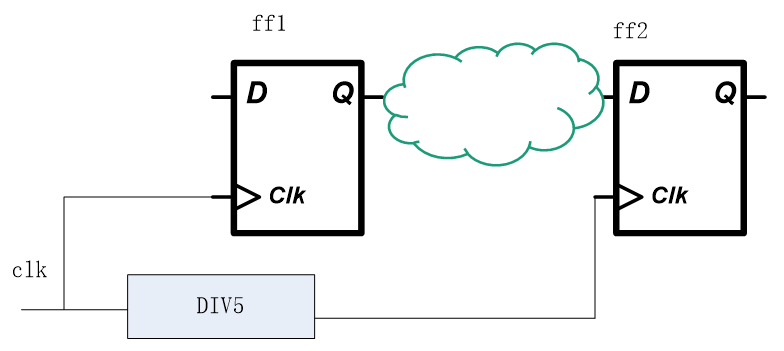


Figure 1‑13 1.1.3.10

点评：上面的电路看似简单，在工程中在也简单不过，但是很多人都写的不太对。

set\_multicycle\_path -setup 5 -from clk1 -to clk2 -start

set\_multicycle\_path -hold 4 -from clk1 -to clk2 –end

### DFT面试题

#### 谈谈对DFT（可测性设计）的理解

参考每日一题第16题。

#### 谈谈对MBIST的理解

参考每日一题第31题。

#### 谈谈对MarchC算法的理解

参考每日一题第32题。

#### 何为WAT、CP、FT

参考每日一题第49题。

#### DFT：Fault simulation

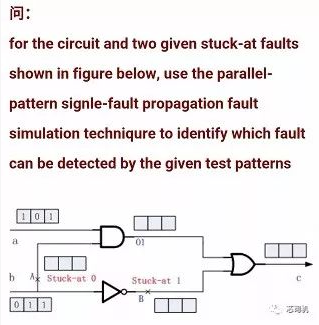


Figure 1‑14 DFT Fault Simulation

参考每日一题第80题

#### ATPG： D算法的应用

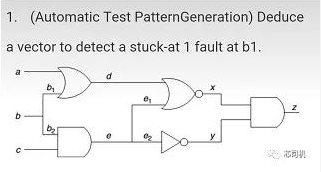


Figure 1‑15 ATPG D算法的应用

参考每日一题第82题

#### 谈谈芯片的IO排布形式与封装之间的关系

参考每日一题第40题。

### 后端面试题

### 每日一题

#### 太太你所了解的低功耗设计思路和方法

1. 系统设计层面：合理规划芯片的工作模式，通过功耗管理模块控制芯片各个模块的clock，起到控制功耗的目的，比如stop、sleep、deep sleep等工作状态，以及各个状态下各个模块的工作状态；合理的划分power domain，可以在sleep的时候将某个power domain直接掉电；
2. Clock gate：通过给DFF增加合理的enable信号，有效降低动态功耗；
3. Multi-VDD：各个power domain使用不同的电压；
4. Multi-VT：合理选择器件以降低漏电功耗，在满足timing的前提下尽可能的使用LVT；

#### 谈谈对PVT三项因素的理解，数字电路设计过程中需要如何考虑PVT

1. Process：加工工艺的正常波动会导致生产出来的器件的速度不一致；
2. Voltage：芯片工作电压的正常波动，±10％；
3. Temperature：芯片的工作温度；

数字电路设计时会做STA分析，分析在各个PVT工作条件下，芯片的时序是否满足要求。在比较旧的工艺中，会通过base case（BC，-40℃+V（+10%）\*P（FF））-Hold，worse case（WC，125℃+V（-10%）+P（SS））-Setup。比较先进的工艺中foundry会提供建议的sign-off corner。

#### 谈谈对静态时序分析、动态时序分析仿真的理解以及优缺点

静态时序分析-STA：根据电路网表的拓扑结构，检查电路设计中的所有路径是否满足时序要求。

特点：分析速度快，但是不关心功能的正确性，不能处理异步逻辑；

动态时序仿真-Post-Sim：利用PR之后的网表+RC反标文件+仿真激励，对电路的时序和功能进行验证。

特点：功能+时序的验证，但是速度慢，而且不一定能够仿真到关键路径。

#### 谈谈对FPGA和ASIC的理解，各自的特点

FPGA：现场可编程门阵列。Lut是其可编程的核心，高端FPGA会包括众多的IP，像PCIe，高速serdes等，甚至基于arm core。

开发流程：RTL+IP🡪综合🡪布局布线🡪bitstream🡪download🡪show。

特点：开发周期短，小批量有成本优势。

ASIC：专用集成电路，特定用户需求，特定的电子系统。

特点：开发周期长，大批量会有成本优势，体积小，功耗低，性能高。

# 经典设计

## Sync FIFO

### 标志位

* wr\_full = wr\_ptr\_nxt == {~rd\_ptr\_nxt[ADDR\_WIDTH], rd\_ptr\_nxt[ADDR\_WIDTH-1:0]}
* rd\_empty = rd\_ptr\_nxt == wr\_ptr\_nxt
* al\_wr\_full=(wr\_ptr\_nxt+alf\_thrh)=={~rd\_ptr\_nxt[ADDR\_WIDTH],rd\_ptr\_nxt[ADDR\_WIDTH-1:0]}
* al\_rd\_empty = (rd\_ptr\_nxt+ale\_thrh) == wr\_ptr\_nxt)

## Async FIFO

### 标志位

* wr\_full = wr\_ptr\_nxt == {~rd\_ptr\_nxt[ADDR\_WIDTH: ADDR\_WIDTH-1], rd\_ptr\_nxt[ADDR\_WIDTH-2:0]}（注意：因为是Async FIFO，所以write/read point需要sync到另一个clock domain使用，这个时候sync的是gray code，所以这个和sync FIFO之间有所不同）
* rd\_empty = rd\_ptr\_nxt == wr\_ptr\_nxt（这个用gray code或者bin code 都一样，不过一般用gray code）
* almost full/empty的判断和sync FIFO一样，但是因为运算需要用bin code，所以需要将gray code转为bin code，转换的code如下：

for（i=0; i<WIDTH;i=i+1）begin

bin\_code[i] = ^gray\_code[WIDTH-1:i]

end

### 深度

同步FIFO的深度可以是任意一个正数，取值可能为8，9或者12，。然而，异步FIFO有一些限制，深度通常是2的n次幂，这里n是正数。一些有效的数值是2，4，8，16，32，64，128等。这是因为异步FIFO指针传递时使用格雷码进行跨时钟域传输。在格雷码方案中，完整的序列长度通常是2的n次幂。如果序列小于2的n次幂，相邻格雷码之间只有一位发生变化的特性也就不再有了，在计数到边界并返回时会有不止一位发生变化。

以23为例，格雷码序列是：

000🡪001🡪011🡪010🡪110🡪111🡪101🡪100🡪000

返回发生在从100到000，此时也只有一位发生变化。如果按照这种做法设计深度为6的FIFO，此时的序列为：

000🡪001🡪011🡪010🡪110🡪111🡪000

现在返回发生在从111到000时，有三位发生变化，使得指针数值不合法，导致不正确的操作。这意味着，我们通常需要的深度为2的n次幂。当深度很小时，这不是个大麻烦，额外空间也不多，但是当深度很大时，额外空间非常巨大。例如，如果我们需要深度为300的AFIFO，应该如何做？我们可以使用深度为512的AFIFO，也就是2的9次幂。此时能正常工作，但是有近200个额外空间没有真正使用，此时可以使用两种方法来处理。

一种方法时使用两个FIFO，一个时深度为300的同步FIFO，后面跟随一个小的异步FIFO（深度为8），用于将数据于其它时钟域同步。另一种方法是仍使用格雷码方案，但是需要进行一些修改。FIFO中的指针通常从0到最大值计数，然后返回0。如果计数到299，然后返回到0，会有多位发生变化（可以比较一下0和299的格雷码之间有多少位发生了变化）。我们可以对任何数值建立偶数格雷码计数器（不一定是2的n次幂），这通过增加一个计数偏移量来实现。此时，计数值不是从0到2的n次幂，而是从(2n)/2-(FIFO\_DEPTH/2)到(2n)/2+(FIFO\_DEPTH/2)-1。因为格雷码是中心对称的（所以从两头计数其实也是可以的，但是从中间计数方便计算地址偏移）。这种做法只能解决深度为偶数的AFIFO的设计问题，对于深度为奇数的AFIFO，将其深度加一就好。

对于本例（深度为300），计数值范围将是从(512/2)-(300/2)到(512/2)+(300/2)-1，也就是从106到405。表示为格雷码之后，计数值范围是106(0\_0101\_1111)到405(1\_0101\_1111)。当计数器的计数值达到405，之后返回到106，此时仍然只有一位发生变化。当然，此时我们需要对地址译码逻辑进行一些调整。指针值106对应FIFO的第一个存储位置，指针 值405对应的是FIFO的第300个位置。为了完成这个地址偏移，需要从write point和read point减去106，然后送给FIFO地址译码器。

## CRC

CRC网页版计算器：<http://www.ip33.com/crc.html>。

CRC可以通过LFSR实现。

发送方发送Message+CRC，接收方将Message+CRC通过相同的CRC计算，最终的结果为0表示没有CRC错误。

## Encryption/Decryption

### AES/SM4 不同加解密模式的比较

|  |  |  |
| --- | --- | --- |
| 模式 | 优点 | 缺点 |
| ECB | * 可以并行计算 * 设计简单 * 错误的密文只会影响对应的明文 | * 相同的明文会得到相同的密文 * 不能验证消息的完整性（消息被篡改之后不知道）（验证消息的完整性需要用到MAC（Message Authentication Code），密文的收发双发需要提前共享一个秘钥。密文发送者将密文的MAC值随密文一起发送，密文接收者通过共享秘钥计算收到密文的MAC值，这样就可以对收到的密文做完整性校验。当篡改者篡改密文后，没有共享秘钥，就无法计算出篡改后的密文的MAC值） * 不支持明文长度不是128bit对齐 |
| CBC/CFB/OFB | * 设计简单 * 相同的明文不会得到相同的密文 | * 不能并行计算（CFB解密时可以并行计算） * 不能随机访问各个区块了，访问某个区块必须解出前一个区块数据 * 错误的密文会影响同一个区块之后所有的明文 * 不能验证消息的完整性 * 不支持明文长度不是128bit对齐 |
| XTS/CTR | * 相同的明文不会得到相同的密文 * 错误的密文只会影响对应的明文 * 可以并行计算 * 不同的区块可以单独处理 * 支持明文长度不是128bit对齐 | * 不能验证消息的完整性 * 和CBC/CFB/OFB 模式对比来看，如果同一个初始值/调整值/计数值只影响较小区块的数据加解密，则CBC/CFB/OFB模式和XTS/CTR在这一点上是类似的，但是存储成本会增加 * CTR模式可以很方便的支持数据长度非16Byte对齐，XTS虽然也可以支持这点，但是实现比较复杂 |
| CMAC | * 能够验证消息的完整性 * 支持明文长度不是128bit对齐 | * 对数据不做加密 * 不能并行运算 |
| GCM | * 相同的明文不会得到相同的密文 * 可以验证消息的完整性 * 可以并行计算 * 支持明文长度不是128bit 对齐 | * 设计相对比较复杂 |

思考：密文加上明文的hash，是否能保证消息的机密性并且验证消息的完整性?（注意是明文的hash而不是密文的hash）

CCM Mode提供了类似的做法，可以参考。

### SM2

蒙哥马利模乘

<https://max.book118.com/html/2017/0630/119098520.shtm>

### 测信道攻击防护

理论上就是隐藏和掩码两个大的方法：

* 隐藏可以通过调用功耗引擎产生噪声，在原有噪声的基础上再增加一个无关的随机噪声以混淆视听；
* 掩码需要在明文或密钥上加入一个随机数，并在最后再解掉这个随机数，一般掩码都是做在电路内部的，掩码能够从源头上削弱噪声和明文或者密钥之间的关联性；

这两种方法都是削弱甚至接触电路工作时的电磁，能量，噪声等和明文以及密钥之间的关联性。

## 跨时钟域处理--最终详尽版

### 参考资料

* <https://zhuanlan.zhihu.com/p/45186793> 跟IC君一起学习集成电路
* <https://www.cnblogs.com/PG13/p/10329678.html> 新芯时代
* 格雷码 <https://www.cnblogs.com/zhuruibi/p/8988044.html>
* 主要参考来源：“ASIC 中的异步时序设计” 王夏泉 华中科技大学
* 主要参考来源: "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog"
* <https://blog.csdn.net/lovingduo/article/details/77723219>
* 2.5节网络来源：[跨时钟域处理方法总结--最终详尽版 - love小酒窝 - 博客园 (cnblogs.com)](https://www.cnblogs.com/lyc-seu/p/12441366.html?wework_cfm_code=OJtSP5eXxO3QXacvX0iA6M2VwEewcl9%2FuauIrBvatobkiTMw8qkf81hUuaWU2EA6g2AqKMmunRAW%2Bm5l0VIl6v8k21Knk8V2eXcByUwLlnR3kxU5twKemFvAN4Rum3wjvQ%3D%3D)

### 异步时序定义

异步时序设计指的是在设计中有两个或以上的时钟， 且时钟之间是同频不同相或不同频率的关系。而异步时序设计的关键就是把数据或控制信号正确地进行跨时钟域传输。

### 亚稳态

每一个触发器都有其规定的建立(setup)和保持(hold)时间参数, 在这个时间参数内， 输入信号在时钟的上升沿是不允许发生变的。 如果在信号的建立时间中对其进行采样， 得到的结果将是不可预知的，即亚稳态。

触发器进入亚稳态的时间可以用参数 MTBF(mean time between failures)来描述， MTBF即触发器采样失败的时间间隔，其公式描述如下：

MTBF=etr/τ/T0faMTBF=etr/τ/T0fa

其中：

trtr = 分辨时间（时钟沿开始）

τ,T0τ,T0 = 触发器参数

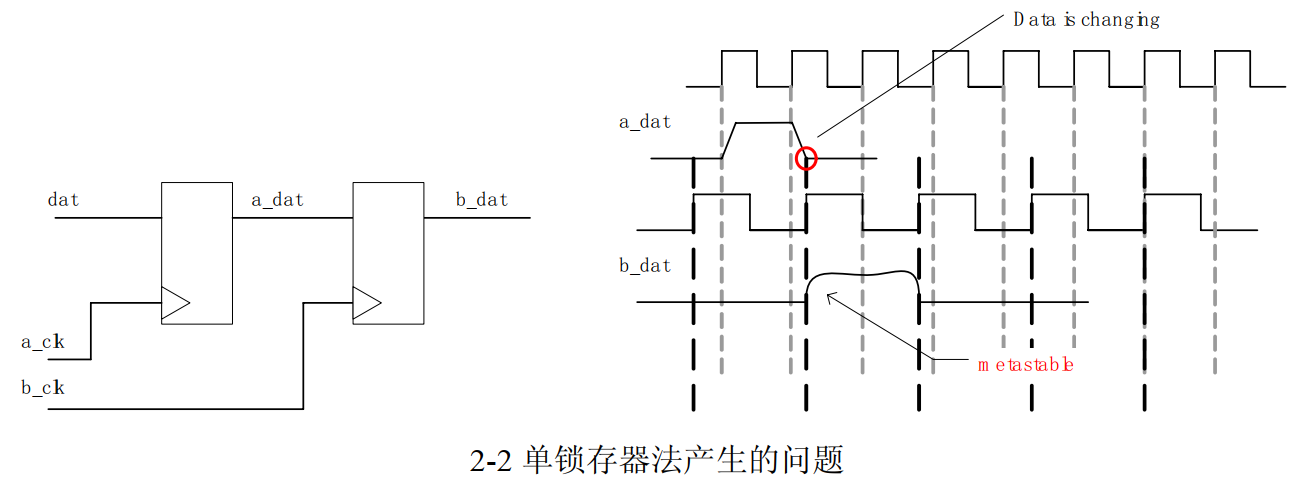
ff = 采样时钟频率

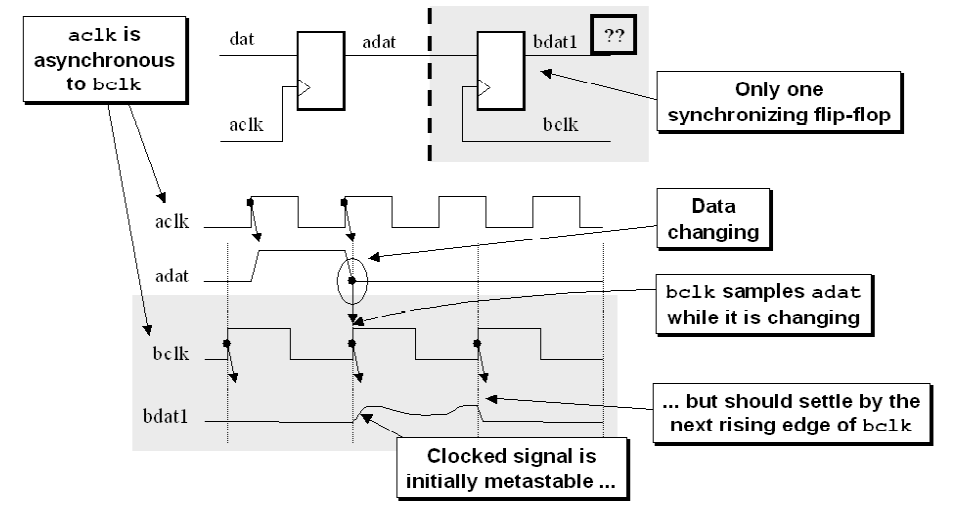
a = 异步事件触发频率

通常，MTBF越大说明系统采样失败的可能越小。可见，对于高速的设计，MTBF是更容易发生的。对于一个典型的 0.25µm 工艺的 ASIC 库中的一个触发器，我们取如下的参数：

tr = 2.3ns, τ = 0.31ns, T0 = 9.6as, f=100MHZ, a = 10MHZ, MTBF = **2.01 days**

也就是说触发器以100MHZ工作，假设异步事件触发的频率，也就是数据变化的频率跟采样时钟频率相同。我们用10MHZ的频率去不停地采它的数据，每个上升沿数据都会发生变化，则每两天就可能采集到一次亚稳态（个人理解，如有误请指正）。如果使用单锁存器同步，b的时钟上升沿采集a的数据时很可能采到亚稳态数据。

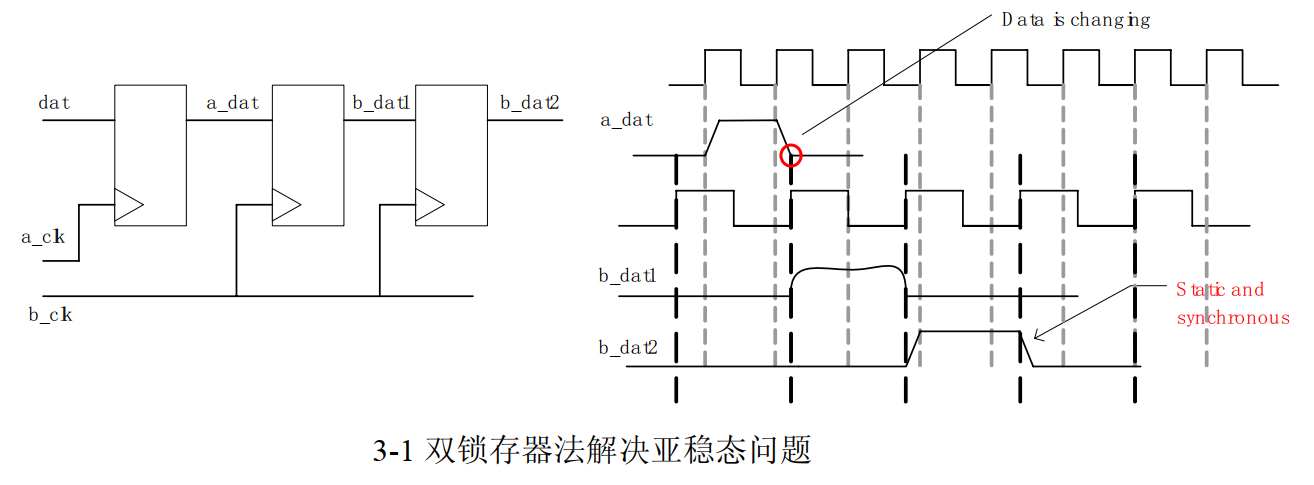
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200302172619.png)

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327201226.png)

### 单比特同步策略

#### 方法一：双锁存器

为了避免亚稳态，应该使得MTBF尽量大。采用双锁存器可以改善这一问题：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200302173321.png)

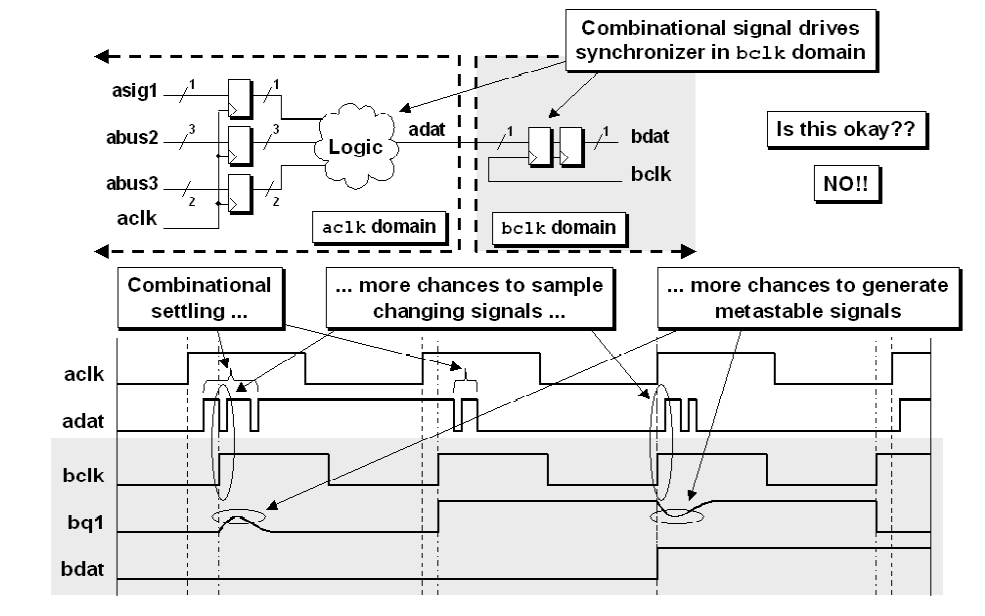
当使用了双锁存器以后， b\_dat2 的MTBF由以下公式可以得出：

MTBF=e(tr/τ)/T0fa×e(tr/τ)/T0faMTBF=e(tr/τ)/T0fa×e(tr/τ)/T0fa

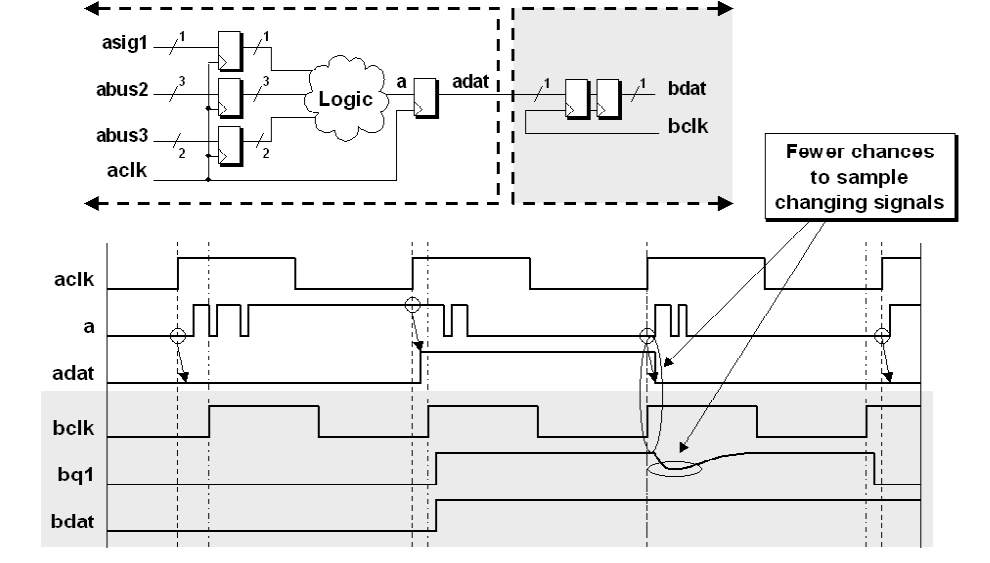
如果我们仍然使用上一节所提供的参数，则b\_dat2 的MTBF为 **9.57\*109(years)**。由上述结果可以看出，双锁存器法可以消除亚稳态问题。

#### 注意问题1

时钟域B两级同步的寄存器跟时钟域A的输出寄存器之间不能有组合逻辑。组合逻辑电路各个输入信号的不一致性以及组合逻辑内部路径的延时时间不一样，运算后的信号存在毛刺如下图,我们无法预先知道CLKB 的上升沿何时会到来，CLKB 采样到的信号就无法预知。

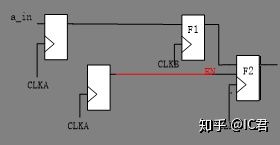
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327202401.png)

因此，要想CLKB 能采到稳定的信号，时钟域A的信号必须是经过CLKA 敲过，在一个时钟周期内是稳定的信号，如下图所示:

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327202458.png)

#### 注意问题2

Clock-gating enable 信号没有经过异步处理：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/v2-3fd99ab9b5abb64b4fc3cd9e17c7d194_720w.jfif)

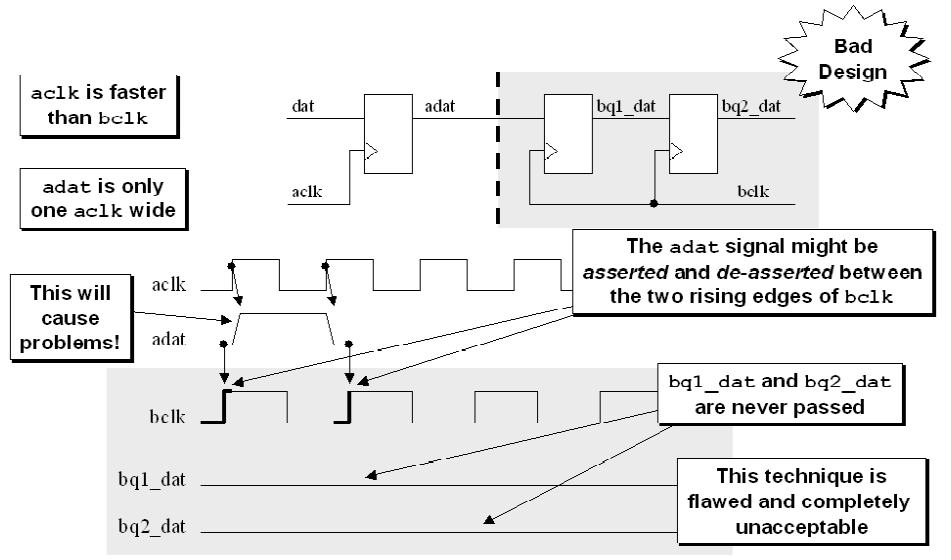
在下图中a\_in 信号经过CLKA的DFF敲过，再送到两级DFF 同步器处理，完全没毛病。但是F2的使能信号EN是从时钟域A来的，当EN信号变化的时候，由于时钟域不一样，无法保证使能之后的CLKB信号采样数据时满足setup/hold time 要求，这时F2输出信号也就变得无法预测了。因此对clk gating的信号也要做处理。

#### 注意问题3

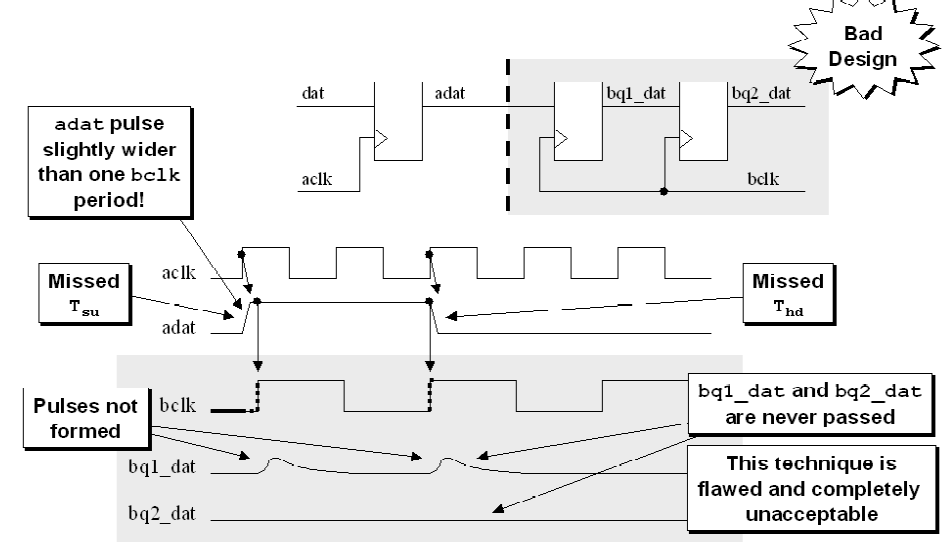
如果较快的时钟域是较慢时钟域频率的1.5倍(或更多)，那么将较慢的控制信号同步到较快的时钟域通常不是问题，因为较快的时钟信号将采样较慢的CDC信号一次或多次。因此将快采慢比慢采快引起的潜在问题更少，所以对于快采慢的情况，使用简单的两个触发器同步器在时钟域之间传递单个Clock Domain Crossing (CDC)信号即可。

而对于慢采快的情况则最好更加稳妥些，一般要求在接收时钟域中采样信号要保持三个时钟上升沿的时间（"three edge" requirement），也就是1.5倍的采样时钟周期。一般来说1.25倍也够。

如果采样信号维持时间过短，则慢时钟域很可能会漏采：

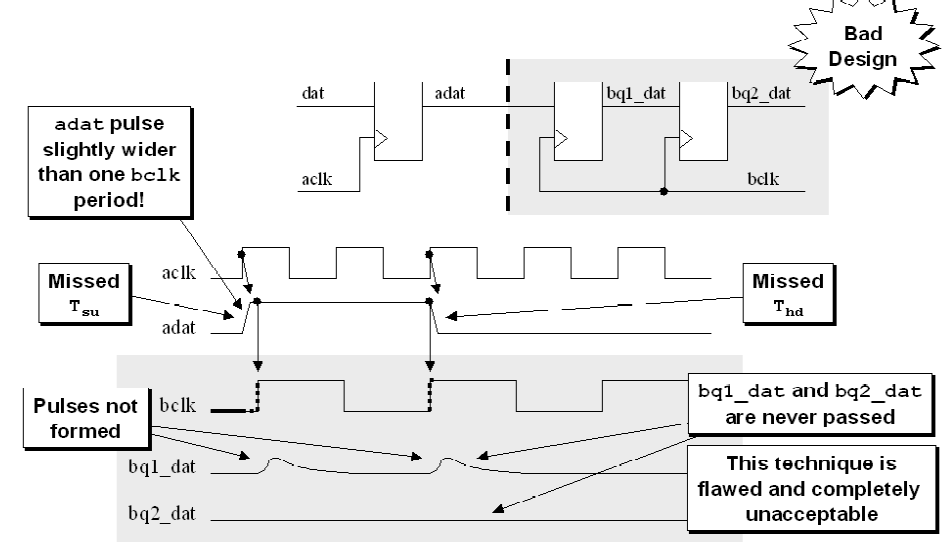
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327203843.png)

即使采样信号比采样周期略长，也可能会面临信号改变正好落在时钟沿的setup和hold violation的区域：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327204212.png)

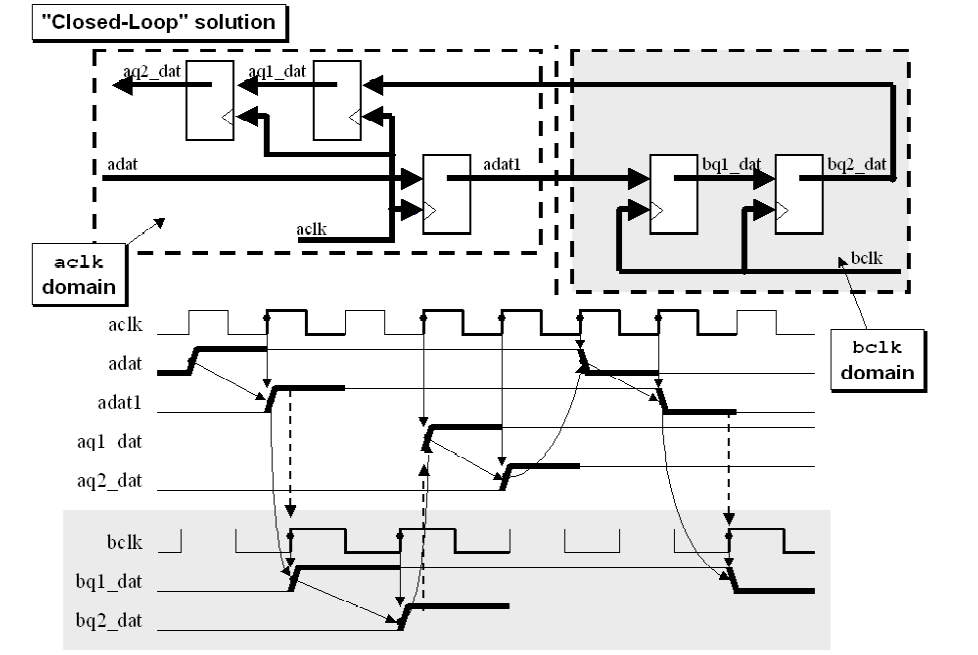
#### 解决方案

1、开环地控制信号的长度

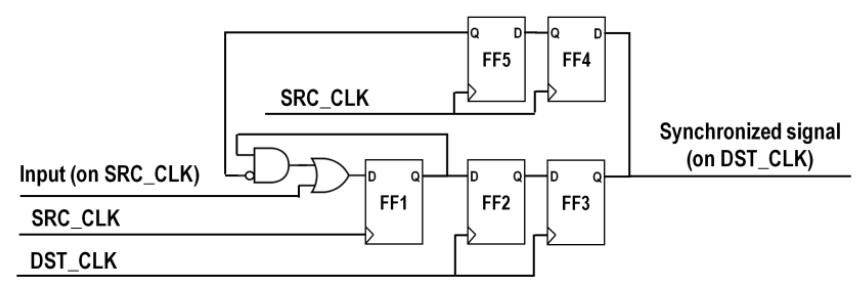
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327204212.png)

不需要知道接收时钟域的边沿信息，只要在发送域保证发送信号的长度就可以。

2、闭环控制：采用反馈信号自动延长快时钟域的信号：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327205648.png)

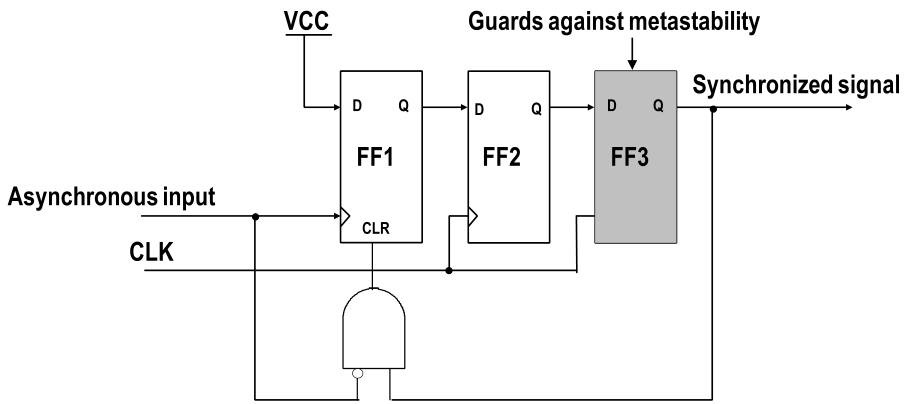
将A时钟域要传输的信号打一拍送到B域中进行双reg同步，同步之后的信号在反馈会A中，A中再进行双reg同步，当A域中反馈同步信号变为高时才将被采信号拉低，这样的方法时比较稳妥的，但是会有更多的延时，并且需要更多同步reg。 例如下图就是采用闭环延长信号的一个方法：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/bb.jpg)

FF1由源时钟驱动，输入变高，FF1输出变高。FF1的Q输出反馈通过与门和或门保证了在FF5输出为0时，只要FF1输出变高，FF1输出就一直保持高。直到FF3同步输出变成1后，FF5输出变成1，与门输出0。这时只要输入为0，FF1输出即为0。

#### 扩展

同样一个快时钟域的脉冲信号，其宽度只有快时钟域的**一个周期的长度**，现在需要将该信号同步到一个相对较慢的时钟域，**如果要求不能在快时钟域将脉冲信号展宽，并且同样采用双寄存器同步，该如何实现？**  
下面的这个电路可以解决这一问题：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200515174430.png)

这个电路跟异步复位同步释放很像，其原理是通过把要处理的异步脉冲当作第一个DFF的时钟输入，这样当有脉冲时该级会拉高，之后后面两级同步器检测到后将第一个DFF置位，这样第一个DFF就置为0，后级之后也变为0. 这种同步器的好处在于不需要再源时钟域对信号做处理，但是缺点也很明显，因为需要将信号作为时钟，会消耗额外的时钟资源，并且该信号的毛刺会导致不可预知的错误。 所以上面这种同步器实际上不推荐使用。还是老老实实展宽脉冲吧。

### 多比特同步策略

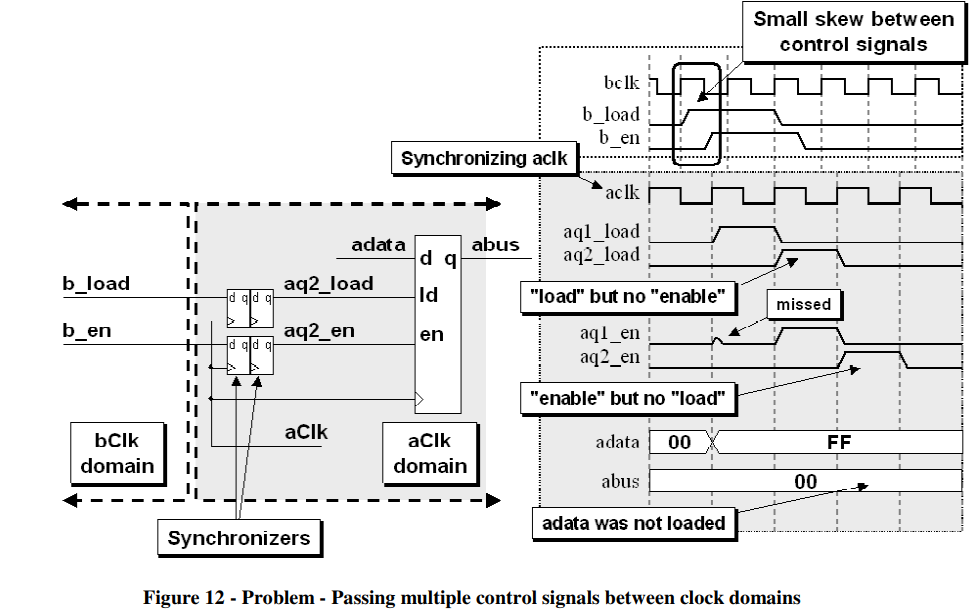
经常会遇到多个比特的控制信号或者数据信号跨时钟域传输的问题，解决这类问题的出发点主要有：

* 尽可能将这些信号合并成单bit
* Multi-cycle path （MCP). 使用同步信号。
* 使用Gray码

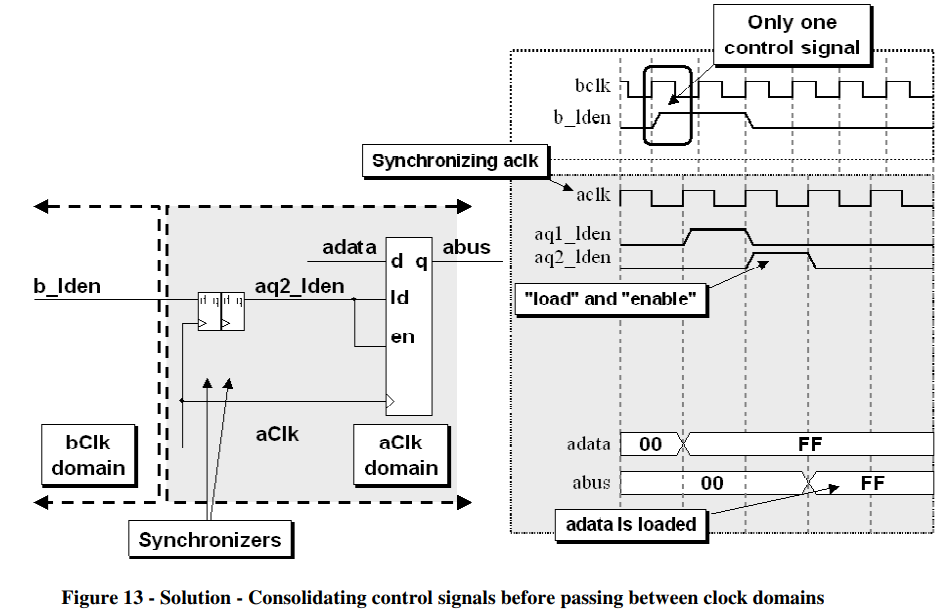
#### 控制信号多比特同步

##### 同步变化的控制信号

例如下面的例子，load和en作为两个控制信号需要跨时钟同步。两个信号可能需要同时拉高才能正确操作，但是实际同步时可能两者存在一定skew，导致再同步域两级reg同步后两个信号错开，导致控制失败：

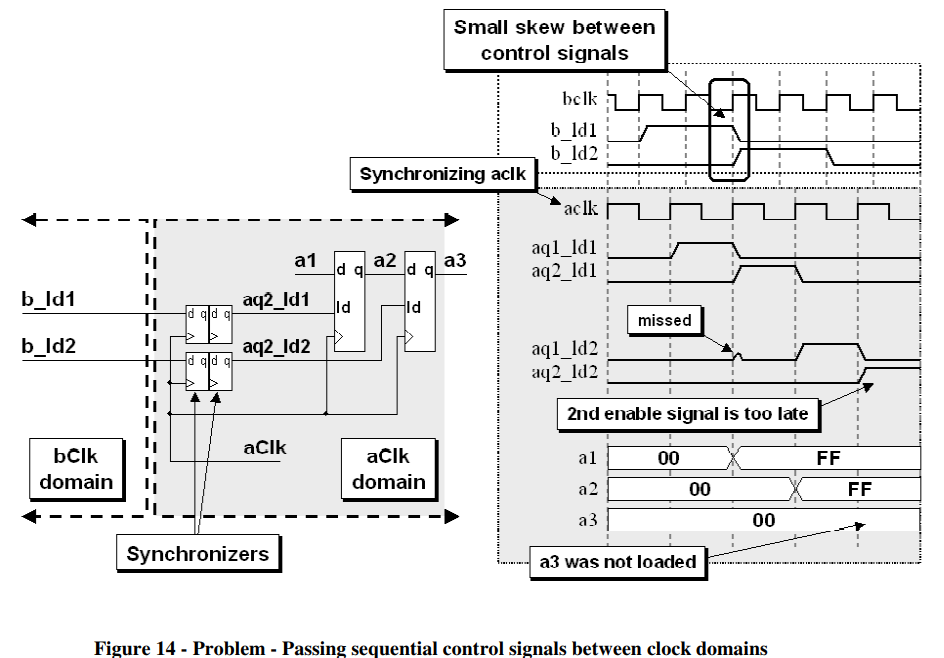
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327210545.png)

这种解决办法也很简单，只要将他们合并成一个即可：

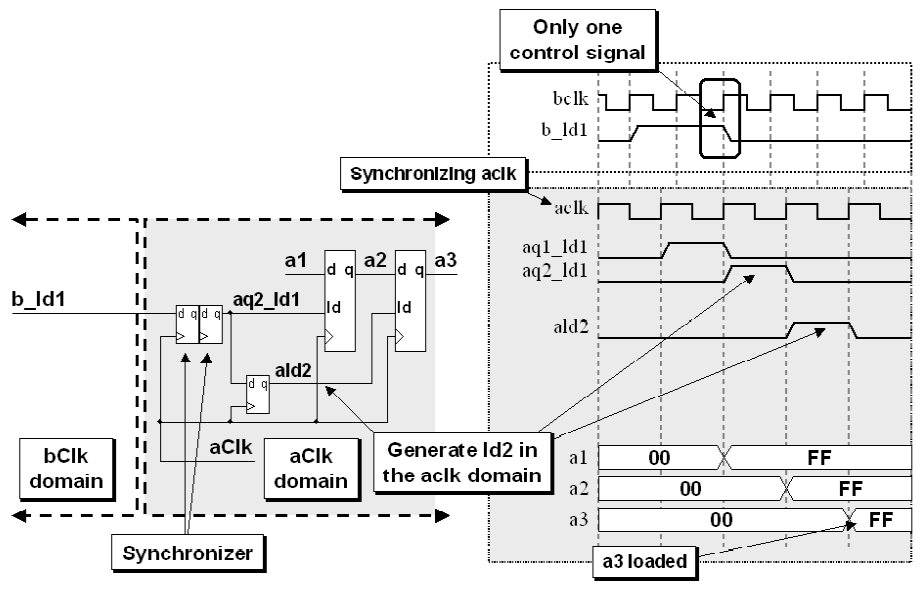
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327211421.png)

##### 控制信号多比特之间有一定时钟相位差

这种情况下，如果两个信号隔开固定的相位，在传输过程中可能会有一定偏移，则同步时也会导致下面的问题：

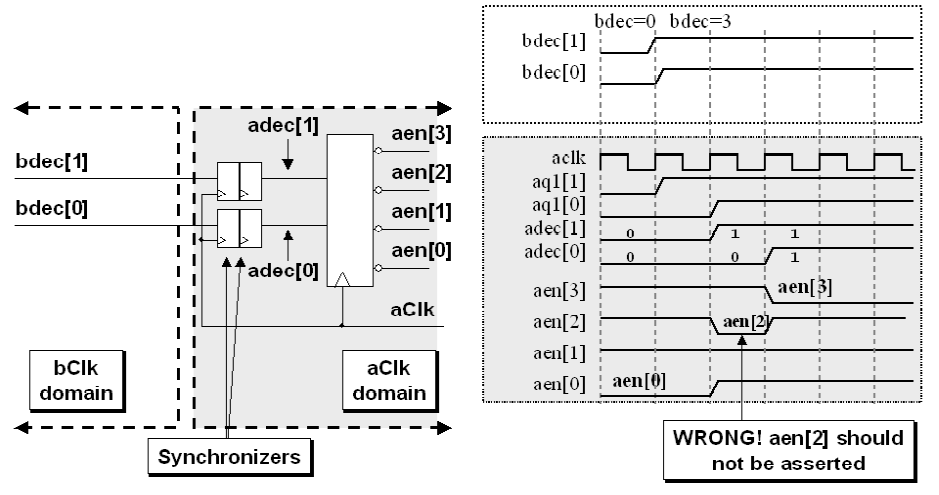
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327211609.png)

可见，ld2信号和ld1信号的相位间隔被改变了，而这通常不是我们希望的。解决办法是只传输其中一个信号，另一个信号在同步域内产生。例如上面的例子可以将ld1在B时钟域通过ld1打一拍得到：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327212106.png)

#### 数据多比特同步

在数据的多比特传输时更要注意skew的问题，如果bit之间有skew，则传输值很可能出错：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327212635.png)

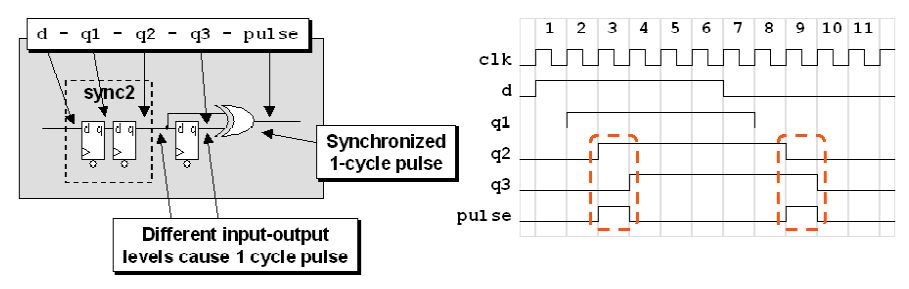
主要有两类解决方法：

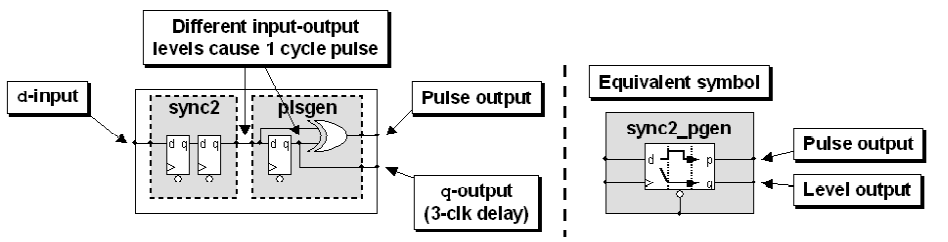
1. Multi-Cycle Path (MCP) formulation 多周期路径规划（名字怪怪的，感觉就是握手机制）；

2. FIFO；

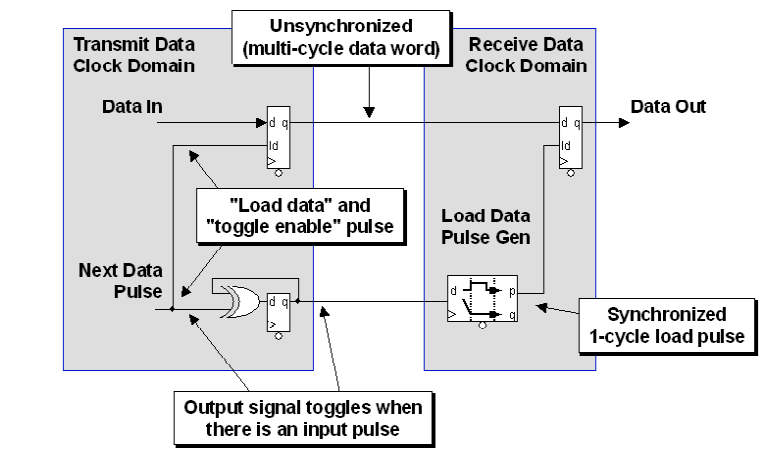
##### 方法一：脉冲同步法（开环的结绳法）

首先用到的一个电路是脉冲产生电路：

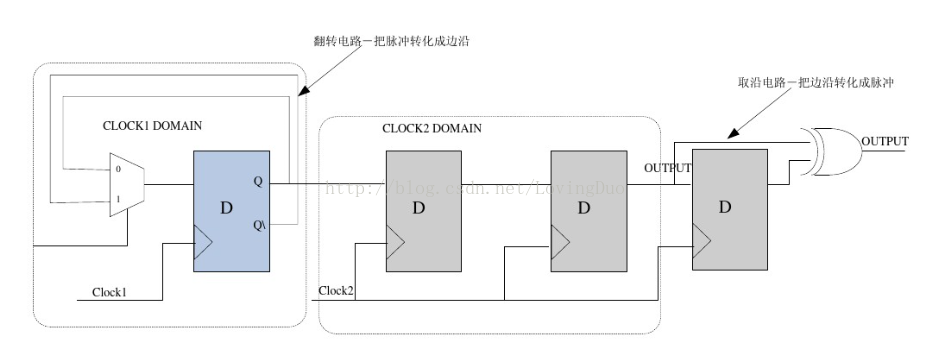
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327214434.png)

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327214508.png)

一个典型的脉冲同步器用于信号同步的例子：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327214807.png)

或者还有另一个版本更加清楚一些：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/1164424-20200313212129892-579612526.png)

脉冲同步器的工作方式如下：

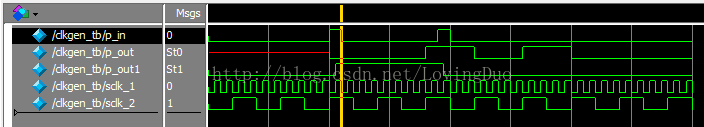
在A时钟域产生一组单周期脉冲，其间隔至少需要比B时钟域的**两个**时钟周期大，否则B中无法进行边沿检测。同时这组脉冲也表示了A中数据传送的开始和结束。

在A时钟域中，两个脉冲被电平翻转器（可以由异或门或者mux+inv构成）将脉冲之间的区域变为一段高电平（结绳toggle)

A中的结绳信号在B中通过边沿产生器翻译为两个单周期脉冲

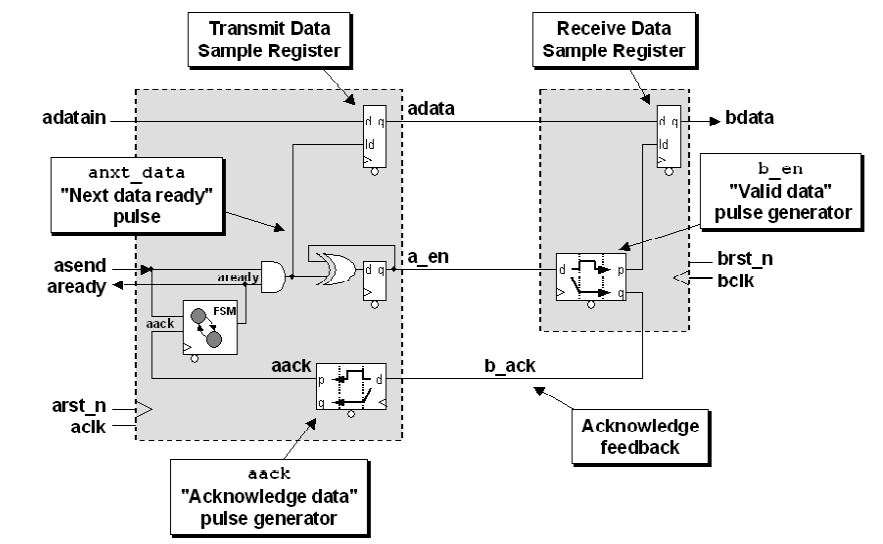
B中需要由逻辑去监测脉冲并进行信号的采样

仿真案例：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20170830204442642.png)

##### 方法二：闭环结绳法

上面的脉冲同步信号实际上是开环产生的，需要考虑B时钟域的周期宽度。而下面这种闭环方式解决了这个问题：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327222325.png)

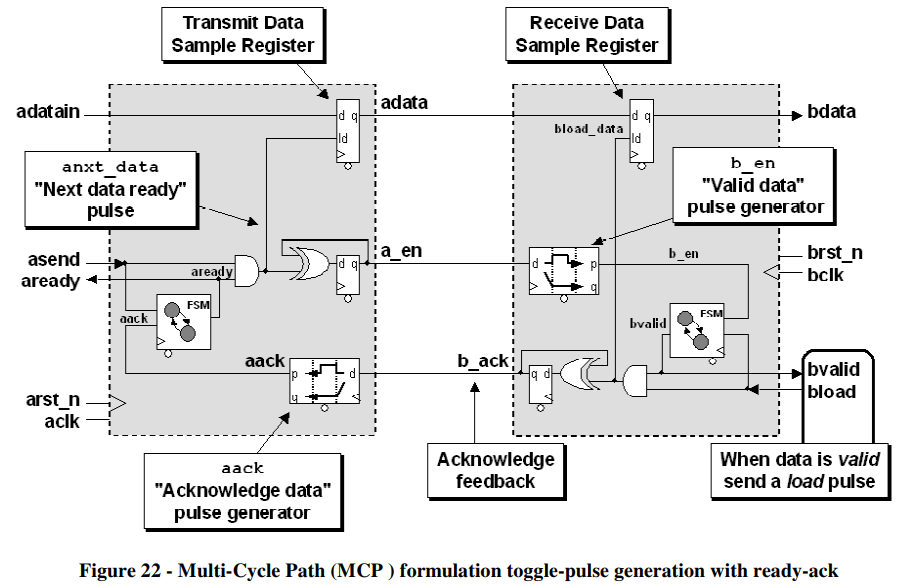
其具体原理是：

首先，asend生成一个脉冲，经过结绳、解绳到B中变为另一个脉冲

该脉冲反馈回A中通过边沿采样变回A中的单周期脉冲（因为这边讨论的是慢采快，所以A总能采到B的单周期脉冲）

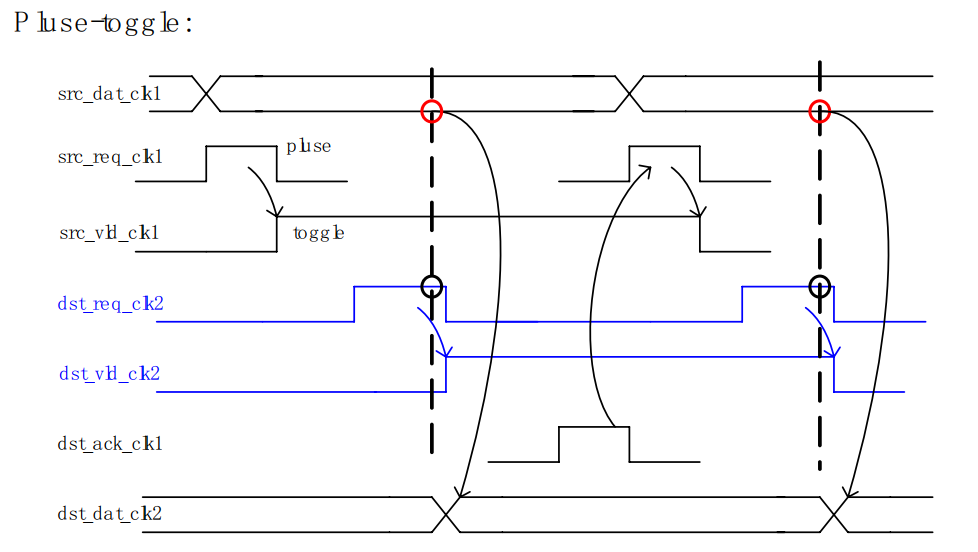
一个简单的FSM接受反馈的脉冲并给出ready信号，表示单次采样结束，A模块可以改变adata的数据了，同时控制asend信号给出一个新的脉冲，代表单次传输结束，结绳解绳到B中后B也接受到了结束脉冲。

上面的这种方法还是不够完备，因为反馈信号也是单周期脉冲，在上面第二步中可以发现，如果A的频率比B慢，采不到脉冲岂不是僵硬，所以为了通用，慢采快还是快采慢都能适配，干脆在B中也加入A的的结绳以及FSM来控制整个流程，原理图如下：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327225022.png)

可见，B中的状态机接收到脉冲后输出valid信号告诉外面B可以采数据了，bload置高，数据开始采样，之后b\_ack置高，送入A中产生结束脉冲，结束脉冲返回B中后FSM跳回not valid状态，一次传输结束。

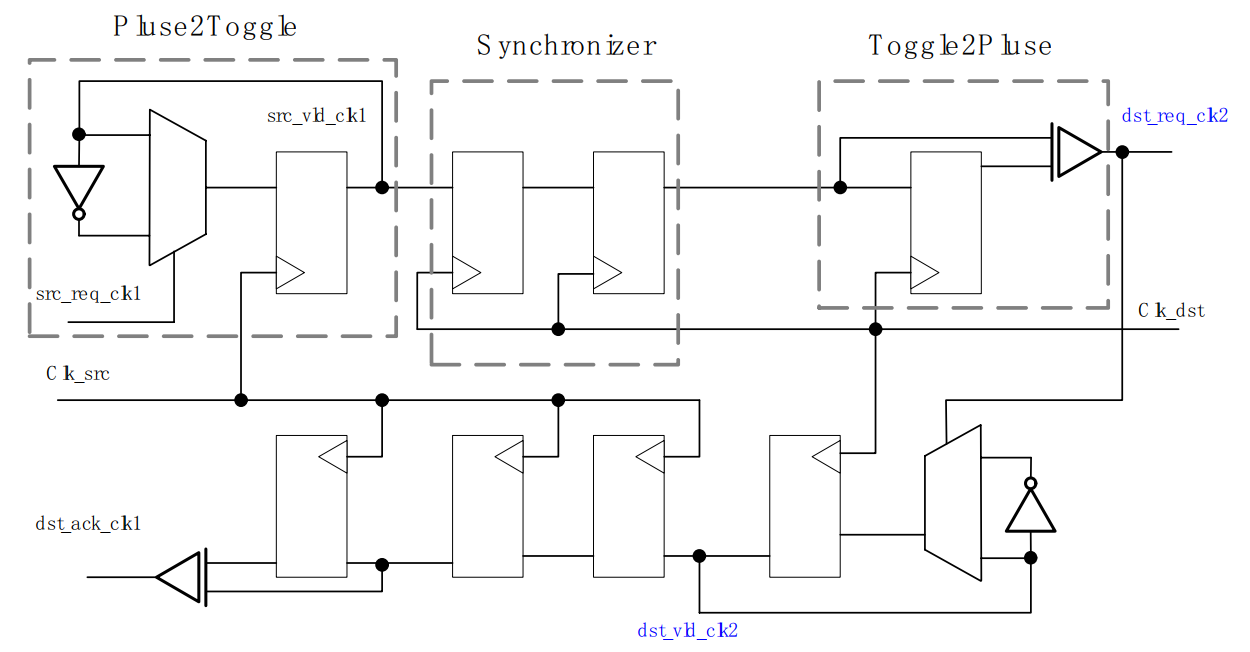
下面是上面的简陋版本：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200302194822.png)

undefined

其中标明\_clk1 的信号表示该信号属于 clk1 时钟域， 同理标明\_clk2 的信号表示该信号属于 clk2 时钟域。在两次src\_req\_clk1 之间被 src\_vld\_clk1“ 结绳” (pluse2toggle)，在将src\_vld\_clk1 用双锁存器同步以后， 将该信号转换为 dst\_req\_clk2(toggle2pluse)。同理，用dst\_vld\_clk2 将 dst\_req\_clk2“结绳”， dst\_vld\_clk2 表明在 clk2 时钟域中， src\_dat\_clk1 已经可以进行正确采样了。 最后将 dst\_vld\_clk2 转换为 dst\_ack\_clk1(synchronizer and toggle2pluse), dst\_ack\_clk1 表明 src\_dat\_clk1 已经被 clk2 正确采样了， 此后 clk1 时钟域就可以安全地传输下一个数据了。 可以看出，“结绳法” 关键是将信号结绳以后， 使其保持了足够长的时间，以便另一个时钟可以正确地采样。

电路图如下：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200302200302.png)

上图描述了握手协议的完整流程，其中三角带横线的符号是异或门。同时给出了两个脉冲之间结绳信号（vld信号）的产生方法。

总之：

**优点**：“结绳法” 可以解决快时钟域向慢时钟域过渡的问题， 且其适用的范围很广。

**缺点**：实现较为复杂，特别是其效率不高，在对设计性能要求较高的场合应该慎用。

##### 方法三：异步双口RAM+格雷码（异步FIFO)

处理多bit数据的跨时钟域，一般采用异步双口RAM。假设我们现在有一个信号采集平台，ADC芯片提供源同步时钟60MHz，ADC芯片输出的数据在60MHz的时钟上升沿变化，而FPGA内部需要使用100MHz的时钟来处理ADC采集到的数据（多bit）。

在这种类似的场景中，我们便可以使用异步双口RAM来做跨时钟域处理。先利用ADC芯片提供的60MHz时钟将ADC输出的数据写入异步双口RAM，然后使用100MHz的时钟从RAM中读出。

但我们读出RAM中的数据时，肯定不是一上电就直接读取，而是要等RAM中有ADC的数据之后才去读RAM。这就需要100MHz的时钟对RAM的写地址进行判断，当写地址大于某个值之后再去读取RAM。

在这个场景中，其实很多人都是使用直接用100MHz的时钟对RAM的写地址进行打两拍的方式，但RAM的写地址属于多bit，如果单纯只是打两拍，那不一定能确保写地址数据的每一个bit在100MHz的时钟域变化都是同步的，肯定有一个先后顺序。如果**在低速的环境中不一定会出错，在高速的环境下就不一定能保证了。所以更为妥当的一种处理方法就是使用格雷码转换。**

##### 格雷码简介

在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码（Gray Code），另外由于最大数与最小数之间也仅一位数不同，即“首尾相连”，因此又称循环码或反射码。格雷码（Gray Code）又称Grey Code、葛莱码、格莱码、戈莱码、循环码、反射二进制码、最小差错码等。

格雷码有多种编码形式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **十进制数** | **4位自然二进制码** | **4位典型格雷码** | **十进制余三格雷码** | **十进制空六格雷码** | **十进制跳六格雷码** | **步进码** |
| 0 | 0000 | 0000 | 0010 | 0000 | 0000 | 00000 |
| 1 | 0001 | 0001 | 0110 | 0001 | 0001 | 00001 |
| 2 | 0010 | 0011 | 0111 | 0011 | 0011 | 00011 |

表中典型格雷码具有代表性。若不作特别说明，格雷码就是指典型格雷码，它可从自然二进制码转换而来。

二进制格雷码的生成方法有很多，具体可自行搜索或见：

<https://www.cnblogs.com/zhuruibi/p/8988044.html>

回到刚才的问题，多比特利用双寄存器打两拍在高速场合不再适用，而使用格雷码可以将这种多比特变为单比特传输（格雷码每次变化只有一位会变）如果先将RAM的写地址转为格雷码，然后再将写地址的格雷码进行打两拍，之后再在RAM的读时钟域将格雷码恢复成10进制。这种处理就相当于对单bit数据的跨时钟域处理了。

##### 异步FIFO

使用异步双口ram的场合其实用异步fifo也是一样的。

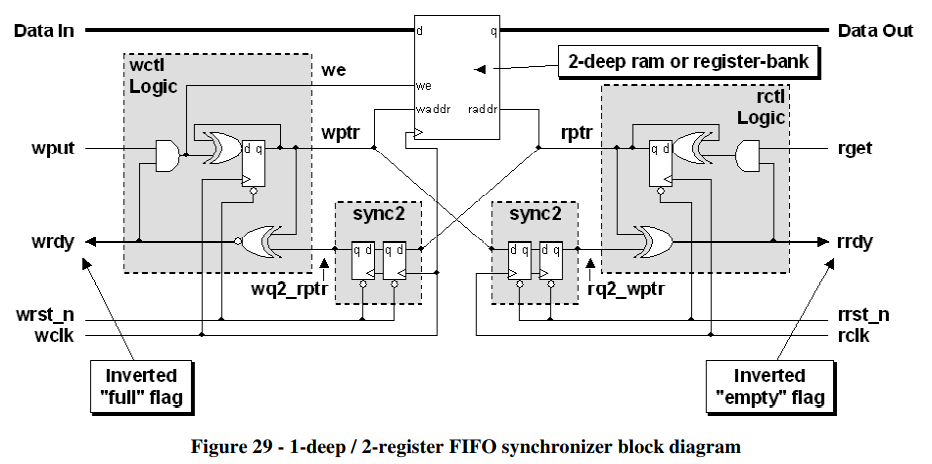
使用场景：在有**大量的数据**需要进行跨时钟域传输， 并且对数据**传输速度**要求比较高的场合 。

一个异步 FIFO 一般由如下部分组成：  
1． Memory, 作为数据的存储器；  
2． 写逻辑部分，主要负责产生写信号和地址；  
3． 读逻辑部分，主要负责产生读信号和地址；  
4． 地址比较部分，主要负责产生 FIFO 空、满的标志。

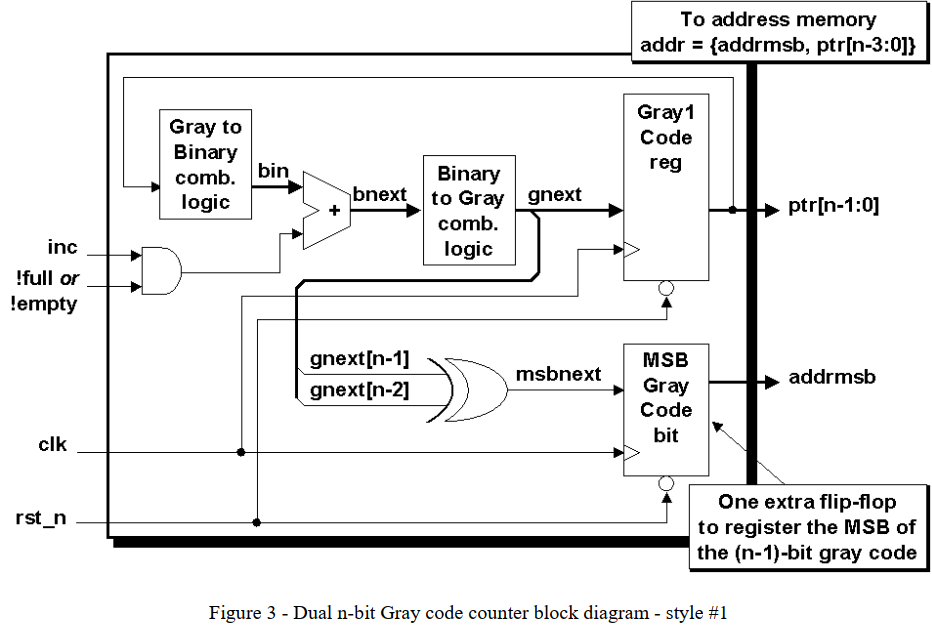
异步FIFO代码可以参考我的另一篇文章：

[https://www.cnblogs.com/lyc-seu/p/12439203.html#示例代码](https://www.cnblogs.com/lyc-seu/p/12439203.html#%E7%A4%BA%E4%BE%8B%E4%BB%A3%E7%A0%81)

##### 方法四：二深度FIFO同步器

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327230725.png)

二深度FIFO的地址实际上只有单bit,所以不需要进行gary编码，所以异步FIFO中的编码模块可以简化：

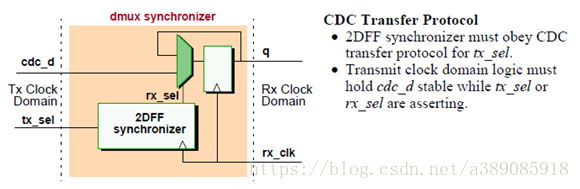
[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/20200327232002.png)

其中的gray2bin,bin2gray都可以省略，并且空满信号的产生之间通过同或门和异或门即可产生。

限制： 由于深度只有2，所以只适合某些特定场景（突发传输要求FIFO最小深度小于2的场景）

##### 方法五：DMUX同步器

对于多bit的data信号，还可以使用使能技术，也就是通过一个使能信号来判断data信号是否已经稳定，当使能信号有效的时候说明data处于稳定状态，在这种情况下终点寄存器才对信号进行采样，可以保证没有setup/hold违例。而使能信号一般使用double FF的方法来进行同步。下面是DMUX的同步示意图：

[](https://lyc-picture.oss-cn-shanghai.aliyuncs.com/screenshot/2018042513565555.png)

# 名词释义

## CPU

Center Process Units。平时用的RISC的ARM也好，CISC的X86也好，都是通用处理器，设计的目的是为了能够进行普遍且广泛的运算需求。

## GPU

Graphics Processing Unit。又称显示核心、视觉处理器、显示芯片，是一种专门在[个人电脑](https://baike.baidu.com/item/%E4%B8%AA%E4%BA%BA%E7%94%B5%E8%84%91/3688503)、[工作站](https://baike.baidu.com/item/%E5%B7%A5%E4%BD%9C%E7%AB%99/217955)、游戏机和一些[移动设备](https://baike.baidu.com/item/%E7%A7%BB%E5%8A%A8%E8%AE%BE%E5%A4%87/9157757)（如[平板电脑](https://baike.baidu.com/item/%E5%B9%B3%E6%9D%BF%E7%94%B5%E8%84%91/1348389" \t "_blank)、[智能手机](https://baike.baidu.com/item/%E6%99%BA%E8%83%BD%E6%89%8B%E6%9C%BA/94396)等）上做图像和图形相关运算工作的[微处理器](https://baike.baidu.com/item/%E5%BE%AE%E5%A4%84%E7%90%86%E5%99%A8/104320" \t "_blank)。GPU使显卡减少了对CPU的依赖，并进行部分原本CPU的工作，尤其是在3D图形处理时GPU所采用的核心技术有硬件T&L（几何转换和光照处理）、立方环境材质贴图和顶点混合、纹理压缩和凹凸映射贴图、双重纹理四像素256位渲染引擎等，而硬件T&L技术可以说是GPU的标志。

## NPU

Neural-network Process Units。NPU简单的理解就是专用处理器，为AI算法专向进行优化，拥有的指令集只与AI算法有关，从硬件层面实现快速处理AI算法中一些特定公式的计算。

## PHY

在ASIC的设计中，PHY是经常要遇到的，它是链接数字逻辑和物理电路的必要环节。通常这部分设计由模拟电路来完成。PHY是物理层接口的意思，是一个AD电路(数模混合电路)，PHY自身可以看作是一种ADC/DAC电路，TX是DAC，RX是ADC。

以DDR PHY为例说明如下：

PHY-物理层，顾名思义就是匹配物理实现，增强物理电路性能的东西，包括数字或者模拟部分，撤了上层应用协议或者接口协议之外的东西，比如DDR Controller 接口🡨🡪PHY🡨🡪外部DDR接口，因为DDR的并行信号可能很难达到更高的运行频率，所以需要PHY将其转换成高速串行数据在PCB上走线，这样会更好的提高信号完整性。

# 接口协议

## PCIE

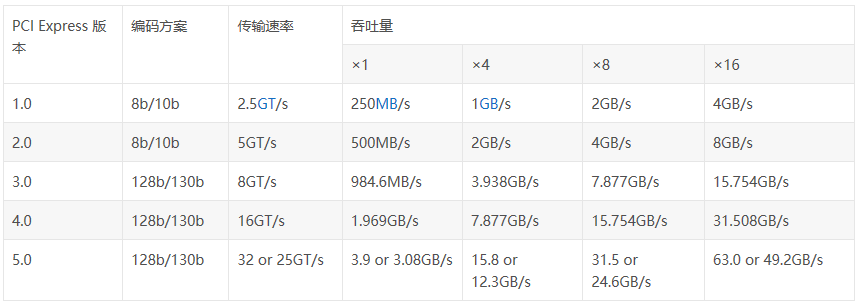


Figure 4‑1 PCIE带宽

## DDR

DDR3-1600（8-8-8）（tCL-tRCD-tRP）

# 设计流程 & EDA\_Tools

## 综合

### Design Compiler Flow

### 随手记

1. write\_file –format ddc： Synopsys internal database format（the default format）
2. 综合的基本流程：translate (analysis+elaborate/read\_verilog(read Verilog时会自动做translate))🡪 link 🡪 mapping 🡪optimization
3. non-unate clock即非单调时钟。理论上来说，时钟电路只能经过寄存器时钟端、buffer(正相关、inverter(负相关)，clock 经过or ，xor等除了与门和非门外的其他逻辑，就会被叫做non-unate。非单调时钟逻辑是指在传播过程中时钟的两种方向(非反转和反转)一起到达同一引脚的情况。
4. get\_cells instance\_name/instance\_name：只会按照hierarchy找到这一层
5. get\_cells –hier {instance\_name}：current design 内所有匹配instance
6. filter时的判断：==/！= 等于和不等于，不识别通配符；=~/！~匹配和不匹配，支持通配符
7. 帮助

* create\_clock -help ：查看create\_clock这个命令的简单用法
* man create\_clock ：查看create\_clock这个命令的详细信息
* printvar Mibrary ：查看 Mibrary这个变量的内容

1. filter时的属性：

* full\_name：带hierarchy信息，比如instance\_name0/instance\_name1/cell\_name
* ref\_name：cell 在library中的名字

1. get\_\*命令返回current\_design，DC memory，libraries中的对象，对象可以使用通配符，返回一个集合，或者返回空集合

* get\_cells -hier {instance\_name}：get instance\_name匹配的所有module
* get\_cells 时如果要用full\_name做filter，就需要加上-hier
* get\_attribute [filter xxx "exp"] sources
* get\_cells -hier -filter “exp“
* get\_pins -of\_objects cell\_list -filter “exp”

1. all\_\*命令：all\_inputs、all\_outputs、all\_clocks、all\_registers、all\_ideal\_nets、all\_fanin、all\_fanout、all\_connected、 all\_dont\_touch、all\_high\_fanout返回对应集合。
2. 将RTL中instance的standard cell设为dont\_touch，set\_dont\_touch [get\_cells -h -f “is\_hierarchical == false && is\_mapped == true && ref\_name ！= \*\*logic\_0\*\* && ref\_name ！= \*\*logic\_1\*\* && ref\_name ！~ gf\_cmos\* ]
3. 报告所有寄存器的个数：reoprt\_cell [all\_registers -edge\_trigger]
4. 启动DC时传递参数：dc\_shell -f main.tcl -x “set TOP A; set SDC B”
5. 比RTL更高的行为级的综合由Synopsys的另外一个工具Behavior Compiler完成
6. read\_file 是可以读取任何synopsys支持格式的；analyze和eloborate只支持verilog和VHDL两个格式，但是他们支持在中间过程中加入参数而且以便以后可以加快读取过程
7. 在design中，instance的子设计，称为cell；没有子模块的cell 统称为leaf cell，我们在定义clock constraint的时候，一般需要把path指到leaf cell
8. 综合时连接dynamic clock gating TE pin:

* set\_dft\_configuration -scan disable -clock\_controller enable
* set\_dft\_clock\_gating\_pin [get\_cells -hier -f “ref\_name =~ xx” -pin\_name TE -control\_signal scan\_enable
* insert\_dft

1. create\_clock -waveform时为了避免{}中不能使用变量的限制可以使用“”
2. gen\_clk不会继承master\_clk的latency和uncertainty（从DC的timing report中看到generated clock并没有继承master clock的latency，但是STA那本书中说generated clock会继承master clock的latency）
3. set\_host\_option -max\_cores integer

### Timing Arc

[Ti](http://www.elecfans.com/tags/ti/)[mi](http://www.hqpcb.com/quote/)ng arc是基本的时序概念，中文名时序弧。这是timing计算最基本的组成元素，在昨天的lib介绍中，大部分时序信息都以Timing arc呈现。如果两个[pi](http://www.elecfans.com/tags/pi/)n之间在timing上存在因果关系，我们就把这种时序关系称为Timing arc，主要分为定义时序延迟，和定义时序检查两种。为啥叫它时序弧？因为时序图中经常用一条弧形线段来表示它。如下图所示：cell的timing arc定义在lib中，net之间没有timing arc, 它的delay则有RC参数计算而出。

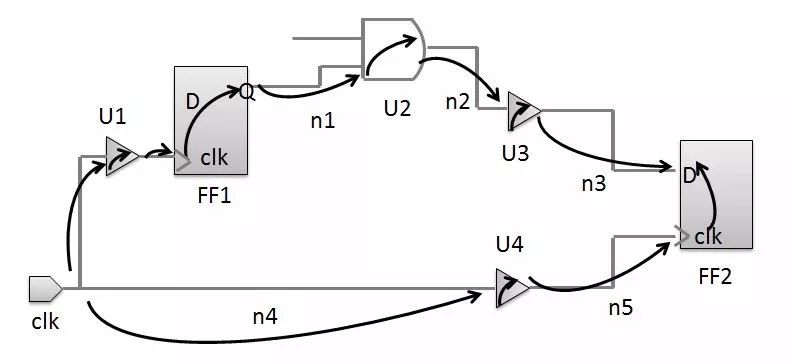
[](http://www.elecfans.com/uploads/allimg/180102/093305A02-0.jpg)

Figure 5‑1 Timing ARC

**时序延迟的Timing arc有以下几种：**

1. Combinational Timing Arc
2. Edge Timing Arc
3. Preset and Clear Timing Arc
4. Three State Enable & Disable Timing Arc

**时序检查的Timing arc有以下几种：**

1. Setup Timing Arc
2. Hold Timing Arc
3. Recovery Timing Arc
4. Removal Timing Arc
5. Width Timing Arc

首先看Combinational Timing Arc，Combinational Timing Arc 是最基本的Timing Arc。Timing Arc 如果不特别指明的话，就是属于此类。如下图所示，定义了从特定输入到特定输出（A到Z）的延迟时间。Combinational Timing Arc 的Sense有三种，分别是inverting（或 negative una[te](http://www.elecfans.com/tags/te/)），non-inverting（或 positive unate）以及non-unate。当Timing Arc 相关之特定输出（下图Z）信号变化方向和特定输入（下图A）信号变化方向相反（如输入由0变1，输出由1变0），则此Timing Arc 为inverting sense。反之，输出输入信号变化方向一致的话，则此Timing Arc 为non-inverting sense。当特定输出无法由特定输入单独决定时，此Timing Arc 为non-unate。

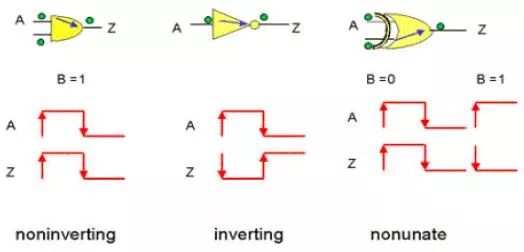
[](http://www.elecfans.com/uploads/allimg/180102/0933052341-1.jpg)

Figure 5‑2 Combinational Timing Arc Sense

**其它的Timing Arc 说明如下：**

Setup Timing Arc：定义时序组件（Sequential Cell，如Flip-Flop、Latch 等）所需的Setup Time，依据Clock上升或下降分为2类（图五）。

Hold Timing Arc：定义时序组件所需的 Hold Time，依据Clock 上升或下降分为2类（图六）。

Edge Timing Arc：定义时序组件Clock Active Edge 到数据输出的延迟时间（CK-to-Q），依据Clock上升或下降分为2类（图七）。

Preset and Clear Timing Arc：定义时序组件清除信号（Preset或Clear发生后，数据被清除的速度，依据清除信号上升或下降及是Preset或Clear分为4类（图八）。这个Timing Arc 通常会被取消掉，因为它会造成信号路径产生回路，这对STA而言是不允许的。

Recovery Timing Arc：定义时序组件Clock Active Edge 之前，清除信号不准启动的时间，依据Clock上升或下降分为2类（图九）。

Removal Timing Arc：定义时序组件Clock Active Edge 之后，清除信号不准启动的时间，依据Clock上升或下降分为2类（图十）。

Three State Enable & Disable Timing Arc：定义 Tri-State 组件使能信号（Enable）到输出的延迟时间，依据Enable或Disable分为2类。（图十一）

Width Timing Arc：定义信号需维持稳定的最短时间，依据信号维持在0或1的基准分为2类。（图十二）

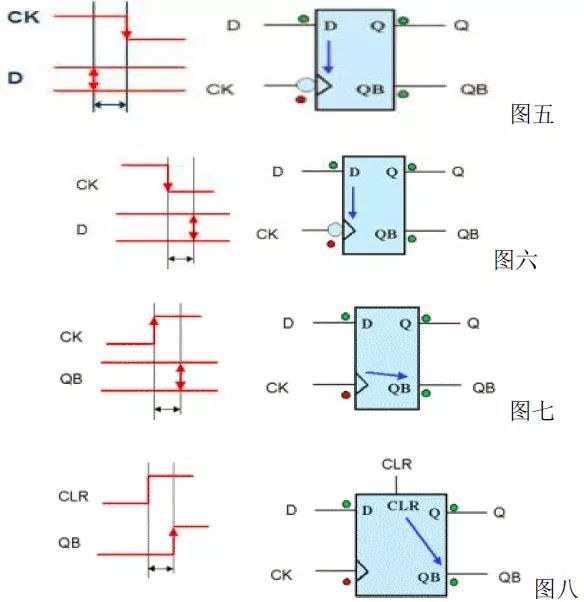


Figure 5‑3 Sequence Timing Arc Sense-1

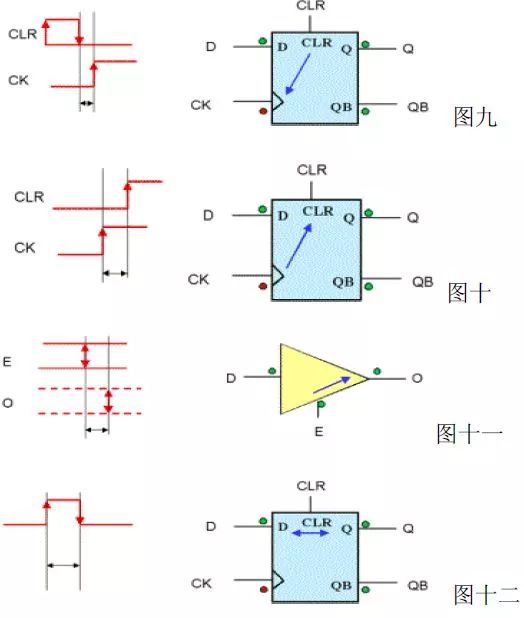


Figure 5‑4 Sequence Timing Arc Sense-2

### OCV

时序分析模式随着工艺尺寸的进步也在不断演化，从早期的single mode，best case-worst case到 OCV，以及现在较为先进的AOCV和SOCV。

Single mode和best-case worst-case（简称BCWC mode）用在较老的工艺节点上。 Single timing analysis mode主要是指整个时序分析过程中都使用同一套相同PVT情况的library，如下图所示：

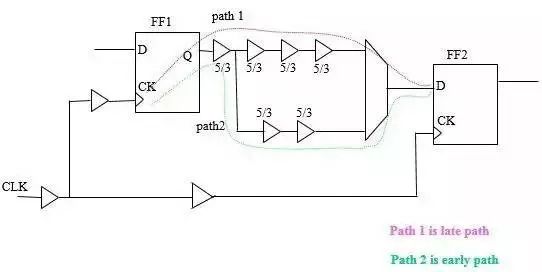


Figure 5‑5 OCV analysis-1

对于setup Analysis

Launch clock: late path

Data path: late path

Capture clock: early path

对于hold Analysis

Launch clock: early path

Data path: early path

Capture clock: late path

BC-WC timing analysis mode: BC-WC模式下，工具会去使用两套max，min的library，也是两种PVT条件下，去计算timing path的delay. 同样setup和hold的计算方式不一样。如上图所示：

对于setup Analysis

Launch clock: late path from max lib

Data path: late path from max lib

Capture clock: early path from max lib

对于hold Analysis

Launch clock: early path from min lib

Data path: early path from min lib

Capture clock: late path from min lib

在90nm以上的工艺，我们通常都使用OCV的分析模式，OCV全称on-chip variations，中文名“片上误差”。OCV的出现使得我们对时序分析提出了更严格的要求。

那为什么需要OCV呢？因为制造工艺的限制，同一芯片上不同位置的MOS晶体管的性能会有一些差异。单元库中的PVT是一个"点"，比如1.2V，250℃，工艺1.0. 但实际芯片的PVT永远不会落在一个点上，而是一个范围；比如说有时序关系的几个cell，可能这几个cell的PVT是1.18V，20℃，工艺0.98。而那个cell的PVT是1.21V，35℃，工艺1.01。这些cell的PVT都不在那个点上，怎么去分析呢？这时候就需要OCV了。

OCV可以采用多套library实现。如果只用一套library，那么可以设置time derate，进一步加紧约束，也可以为后面的步骤预留一些margin。考虑timing derate以后，在setup check中，Arrival time即data path和launch clock path需要使用-late 选项，使得路径变慢。Require time即capture clock path需要使用 -early 选项，加快路径延迟。

这里需要注意的是：考虑time derate需要在某个单一条件下，比如说Best Corner或者Worst Corner条件下，把指定path的延迟再放大或者缩小一点，要么是Best Corner，要么是Worst Corner。不要把Best Corner和Worst Corner混在一起，再OCV，那样太过于悲观。

还是用原来那张图，参照single mode和BC-WC模式，OCV的检查方式更加严格，如下所示：

For setup Analysis

Launch clock path: late path delay from max lib

Data Path: late path delay from max lib

Capture Clock path: early path from min lib

For Hold Analysis

Launch clock path: early path from min lib

Data Path: early path from min lib

Capture Clock path: late path from max lib

最后我们看个具体OCV分析计算的例子

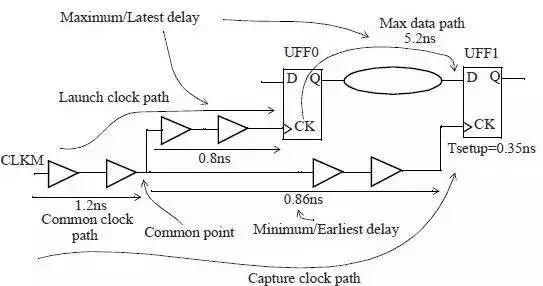


Figure 5‑6 OCV analysis-1

OCV for setup check

先讲setup ocv check，按照上诉定义可以画出OCV mode下，setup检查规则如上图所示：

先假设PVT情况可以在整个chip上变化，（不加timing derate）

slack>0的最小时钟周期

T=launch clockpath + max data path- capture clock path + Tsetup = 1.2 + 0.8 + 5.2 - 1.2 -0.86+ 0.35 = 5.49ns

setup check 一般是工作在Worst Corner PVT条件下，因此不需要在late path上，即launch clock path以及data path上再加time derate，因为在Worst Corner条件下，launch clock path以及data path上的延迟已经是所有条件下最差的delay了，没有必要再加大延迟，但是Worst Corner条件下capture clock path上的delay肯定不是最小的，因此需要进一步加快。

所以上面的timing path做setup check, time derate只需要这样设置：

set\_timing\_derate -early 0.9

set\_timing\_derate -late 1.0

我们可以计算一下加了timing derate以后setup check的变化：

上图中：

launch clock path = （1.2+0.8）\*1.0 = 2.0

max data path =5.2 \* 1.0 =5.2

capture clock path= (1.2 + 0.86) \* 0.9 = 1.854

所以最小时钟周期 T = 2.0 + 5.2 -1.854 + 0.385= 5.731

可以看到：考虑timing derate以后，会降低整个design的工作频率。

CPPR：大家肯定发现了对于path最前端那1.2ns的延迟，在launch clock path中没有被derate, 而在capture clock path中被time derate1.2\*0.9 =1.08. 显然这是相互矛盾的。这种现象称为“悲观时钟收敛”。对于common path, 解决的方法可以采用“共同路径悲观去除”（CPPR，common path pessimism removal）的方法解决。

对于上诉设计，考虑CPPR之后，我们必须减去一个CPP因子=1.2-1.08=0.12

所以最小时钟周期T=5.731-0.12 =5.611

OCV for Hold check

对于Hold check,同样可以参考上图分析模式：

不考虑time derate情况下

hold slack = launch clock path +min data path - capture clock path -Thold= 0.25+ 0.6 +1.7 -0.25 -0.75-1.25 = 0.3 >

考虑time derate的情况与setup正好相反

Require time中的capture clock path使用-late选项，使路径变慢。

Arrival time中的data path和launch clock path使用-early选项，使路径加快。

实际上，Hold check一般在Best Corner条件下，因此，launch clock path与data path不需要再进一步减小delay, 因为已经是最小delay, 但是Best Corner条件下的capture clock path需要进一步变慢。可以使用如下设置：

set\_timing\_derate-early 1.0

set\_timing\_derate-late 1.2

这样添加time derate后：

Launch clock path= 0.85 \* 1.0 = 0.85

Min data path =1.7 \* 1.0 = 1.7

Capture clock path= 1.0 \* 1.2 = 1.2

同样需要考虑CPPR = 0.25 \* （1.2-1.0）= 0.05

所以slack=0.85+1.7-1.2-1.25+0.05=0.15

### SDC

#### SDC Rule

1. Clock：所有的root clock和generated clock都必须cell的PIN上；所有的generated clock都必须相对于root clock；
2. don’t touch：
3. IO：
4. timing exception：

#### Don’t use cell

1. Small drive strength cells: 对PVT工艺参数的漂移很敏感
2. CLK\*\*：这种cell中有可能会有比较大的capacitance或者shielding
3. Delay cell：由于OCV效应，这种cell的delay在芯片的不同位置会有很大差异
4. ANTEN/NABWP
5. Capacitor cell/ Over driver cell/ tie1 tie0 cell/ bus hold/ SFF

#### Example-Sync



Figure 5‑7 SDC for async data transmitter

从上面这个例子可以引申一下：async\_fifo的gray code的每个bit之间，最快的和最慢的相差不能超过1T，不然就不满足相邻两个数据之间只有1bit发生变化这个条件

### UPF

UPF（Unified Power Format），由于硬件描述语言不支持对电源连接的描述，因此要仿真门控电源的工作需要扩展RTL代码或者使用UPF来描述电源开关及其连接。

**一些基本的概念和command介绍：**

1. set\_scope用来定位当前scope，表明UPF command应该在hierarchy中的哪个位置执行；
2. set\_design\_top用来联系UPF和design module，该UPF将应用到每个由该module例化的instance；
3. port和net的区别：port是指module的接口，连接相邻的hierarchy，而net是module内部的，连接各个port；
4. power domain：在物理上，同一个power domain的instance会放在一起，并由相同的电源供电。power domain定义时，一般会指定该处在domain的最高层instance。子instance的默认domain跟它的上层instance一致。子instance如果处在不同的domain, 需要显式地从该domain中移除，或者在对应的domain中指定；
5. isolation的作用是，防止大电流损害电路，以及防止不定态影响电路逻辑。isolation cell有两种模式：normal模式，作用类似buffer，isolation模式，输出(clamp)一个特定值。模式通过isolation enable信号来选择。isolation cell 可放在power domain的input或者output；
6. 如果两个相邻的power domain电压不一样，需要用到level-shifter；
7. isolation和level-shifter功能通常放在同一个cell里面；
8. add\_power\_state 与 create\_pst/add\_pst\_state combination的作用类似，都用来指定power状态信息；
9. create\_power\_switch的作用是插入一个switch可以用来关断输出电压；
10. 通过insert\_mv\_cells –isolation根据UPF file插入isolation cell；

#### 一个简单的UPF file

{

set\_scope

set\_design\_attributes –elemets {.} –attribute enable\_state\_propagation\_in\_add\_power\_state true

load\_upf xxx.upf

check\_upf

set\_voltage 0.99 –object\_list VDD

set\_voltage 0.99 –object\_list VSS

set\_voltage 0.99 –object\_list pw1\_VDD

}

#######################################################

xxx.upf

set\_design\_top pw1\_domain

upf\_version 2.0

## Actual supply network needs to be fully defined

create\_supply\_port VDD

create\_supply\_port VSS

create\_supply\_net VDD

create\_supply\_net VSS

create\_supply\_net pw1\_VDD // 两个VDD公用一个VSS

connect\_supply\_net VDD –ports {VDD}

connect\_supply\_net VDD –ports {VSS}

## All supply sets must be updated with supply nets

create\_supply\_set ssTop –function {power VDD}

create\_supply\_set ssTop –function {ground VSS} –update // 定义top的电源和地

create\_supply\_set ssPD1 –function {power pw1\_VDD}

create\_supply\_set ssPD1 –function {ground VSS} –update // 定义pw1的电源和地

## pw1\_domain

create\_power\_domain pw1\_domain –supply {extra\_supplies\_0 “ssTop”} // 定义pw1\_domain，并且ssTop是pw1\_domain外部的电源域

set\_domain\_supply\_net pw1\_domain –primary\_power\_net ssPD1.power

–primary\_ground\_net ssPD1.ground // 定义pw1\_domain的电源和地

create\_power\_switch pw1\_sw

-domain pw1\_domain

-input\_supply\_port {VDDP ssTop.power}

-output\_suuly\_port {VDDC ssPD1.power}

-control\_port {EN pw1\_psw\_on}

-on\_state {on VDDP {EN}}

-off\_state {off {!EN}}

-ack\_port {ENX pw1\_psw\_wrapper/psw}

map\_power\_switch pw1\_sw

-domain pw1\_domain

-lib\_cells {xxxx}

set\_isolation pw1\_iso\_out\_l

-domain pw1\_domain

-iso\_lation\_supply\_set ssTop

-calmp\_value 0

-applies\_to outputs

set\_isolation pw1\_iso\_out\_h

-domain pw1\_domain

-iso\_lation\_supply\_set ssTop

-calmp\_value 1

-elements {xxx}

set\_isolation pw1\_no\_iso

-domain pw1\_domain

-no\_isolation

-elements {xxx}

set\_isolation\_control pw1\_iso\_out\_l

-domain pw1\_domain

-isolation\_signal xxx

-isolation\_sense high

-location parent

set\_isolation\_control pw1\_iso\_out\_h

-domain pw1\_domain

-isolation\_signal xxx

-isolation\_sense high

-location parent

map\_isolation\_cell pw1\_iso\_out\_l

-domain pw1\_domain

-lib\_cells {xxxx}

map\_isolation\_cell pw1\_iso\_out\_h

-domain pw1\_domain

-lib\_cells {xxxx}

## UPF power state definitions

add\_power\_state ssTop –state TOP\_ON {-supply\_expr {power ==`{FULL\_ON, 0,99}}}

add\_power\_state ssTop –state GROUND {-supply\_expr {power ==`{FULL\_ON, 0,0}}}

add\_power\_state ssPD1 –state PD1\_ON {-supply\_expr {power ==`{FULL\_ON, 0,99}}}

add\_power\_state ssPD1 –state PD2\_OFF {-supply\_expr {power ==`{OFF}}}

UPF power state table

create\_pst gk2303\_pst –supplies {ssTop.power ssPD1.power ssTop.ground}

add\_pst\_state q0 –pst gk2303\_pst –state {TOP\_ON PD1\_ON GROUND}

add\_pst\_state q1 –pst gk2303\_pst –state {TOP\_ON PD1\_OFF GROUND}

## Power

### 功耗分类：

1. Leakage Power

Leakage Power由标准单元库的cell .db提供相关数据。

1. Dynamic Power（包括internal和switching power）

internal power由标准单元库的cell .db提供相关数据，但与激励有关系。

switching power由电压、网表电容、线开关计算得出（我觉得net不能仅仅说是线，而应该是管子源漏栅的连线。电容是管子和线的寄生电容）。与激励有关系。

原文链接：<https://blog.csdn.net/u011729865/article/details/54138703>

### Clock Gating

clock gating通过ICG-AND、ICG-OR实现。

* 使用ICG-AND时其前后的DFF必须是正沿触发，并且latch的enable是低有效；
* 使用ICG-OR时其前后的DFF必须是负沿触发，并且latch的enable是高有效

以ICG-AND为例，clock上升沿之后开始计算sleep信号，在clock为高的这一段时间，由于latch的enable是低有效，所以sleep信号不会通过latch影响AND的输出，等到clock为低之后，AND的输出为0，不受sleep的影响，所以这一个时钟周期内能保证AND的输出没有毛刺，而且这个时钟周期结束时已经完成sleep的计算，进而完成clock gating。

## DFT

DFT可以分为三个大的方向：

1. JTAG：主要用于测试各个芯片的连接
2. MBIST：主要用于测试memory中是否有错误，有时也会做memory repair
3. SCAN：主要用于测试电路内部的错误

### SCAN

将数字电路内部的各个DFF串称不同的扫描链（scan chain），在输入和输出pad侧插入EDT逻辑，完成解压缩和压缩，使其能够通过pad控制scan chain的输入和观测其输出。同时需要控制clock和reset，保证能够正确的shift in/capture/shift out。

SCAN中有两个很重要的要求就是可控制（controller）和可观测（observable），可控制是为了很好的控制scan chain的输入，可观测是为了能够很好的观测pattern的测试结果。为了能够做到这两点，就一定需要控制clock/reset。为了控制clock和reset，RTL阶段需要注意两个问题：

1. Clock path上插入OCC（On-Chip Clock Control），注意OCC后不能有clock divider，这样做一是为了能够在stack阶段保证clock是想要的频率，二是为了保证在transition时用的clock和实际function时一致。
2. reset sync后插入mux，保证reset-sync-DFF串入scan chain之后reset仍然可控。

## Verdi

### 随手记

1. 打开verdi的时候自动load 波形：-ssf wavename
2. 打开Verdi的时候设置模式：-workMode hardwareDebug
3. $fsdbAutoSwitchDumpfile(10,"test.fsdb",20),表示在fsbd文件到10M大小之后，创建新的fsbd文件，最多创建20个fsdb文件，超过20个之后会覆盖掉之前的波形

## PT

### PTPX

#### Report\_power:

Before generating the report, power information is updated if necessary

#### Report\_switching\_activity:

Activity propagation for other unannotated nodes does no occur until the update\_power dommand is run

* -hierarchy: When the command is used with this flag, the report generated includes about subblocks in the design;
* -average\_activity: When combined with the -hierarchy flag, average switching activity is computed for each subblock in the design

#### Power\_analysis\_mode:

PrimeTime PX provides three diffeent analysis modes: averaged, time\_based, and leakage\_variation.

* averaged: Default value. PrimeTime PX calculates power based on toggle-rate and state-probability.
* time\_based: PrimeTime PX calculates power based on the events from VCD. For more information, see the update\_power man page.
* Leakage\_variation: PrimeTime PX performs leakage variation analysis. For more information, see the power\_enable\_leakage\_variation\_analysis man page.

#### Set\_power\_analysis\_options:

Just support option -static\_leakage\_only, -cells, -sdpd\_tracking, -pdpd\_tracking\_cells when power\_analysis\_mode is averaged.

#### Set\_switching\_activity:

Use this command to annotate design nets, ports, pins and cells with the different kinds of switching acitivity. These inlcude simple toggle rate, glitch rate, and static\_probability on nets, ports and pins; state and path dependent toggle rate and glitch rate in cell pins, and state dependent static probabilities on cells.

* -state\_condition state\_condition(string)
* -static\_probability static\_probability(float): Specifies the value of the static\_provavility switching activity.. The satic\_probability value represents the percentage of time the signal is at logic state 1.
* -toggle\_rate toggle\_rage(float): Specifies the value of the toggle rage switching acitvity.
* -glicth\_rage glitch\_rage(float):Specifies the value of the glitch rate switching activity. Note: If either of the -toggle\_rate or the -glitch\_rate option is specified, the other is assumed to be zero.
* -period perido\_value(float): Specifies the time period for which the number of transitions given in the toggle\_rate and glitch\_rate occur.
* -base\_clock clock: Specifies a clock by which the toggle and glitch rate values are referenced.
* -clock\_domain: If a n object belongs to multiple clock domains, the fastest of the clocks, is slected as the related clock.
* -hierarchy: Use with the -type option to specify that the objects in all the hierarchies in the current instance that sataify the selection criteria. If not spedified, the tool annotated only the top level objects in the current instance that satisfy the selection criteria.
* -type object\_type\_list:
* Registers: sequential cell outputs
* Three\_states: tristate cell outputs
* Inputs: input design ports/hierarchical instance pins
* Outputs: output design ports/hierarchial instance pins
* Inout: inout design ports/hierarchial instance pins
* Ports: design port/hierarchial instance pins
* Nets: nets
* Clock\_gating\_cells
* Black\_boxes
* Non\_clock\_network
* memory

# Verilog

## 实例分析

### always block中阻塞赋值与非阻塞赋值的执行顺序



Figure 6‑1 Coding style with circuit-always

## function和task

### Function

**function的定义：**

function <返回值类型> <返回值位宽> <函数名>

<输入变量和类型声明>

<局部变量和类型声明>

行为语句

endfunction

**定义function时，要注意以下几点：**

1. function定义结构不能出现在任意一个过程块(always块或者initial块)的内部；
2. function定义不能包括有任何时间控制语句，即任何用#，@或wait来标识的语句；
3. 定义function时至少要有一个输入参量；
4. 定义function时，在function内部隐式地将函数名声明成一个寄存器变量，在函数体中必须有一条赋值语句对该寄存器变量赋以函数的结果值，以便调用function时能够得到返回的函数值。如果没有指定的返回值的宽度，function将缺省返回1位二进制数；
5. function的调用：<函数名> (<输入表达式1>,...,<输入表达式n>) ;
6. 输入表达式与函数定义结构中的各个输入端口一一对应，这些输入表达式的排列顺序必须与各个输入端口在函数定义结构中的排列顺序一致。
7. function的调用既可以出现在过程块中又可以出现在assign连续赋值语句之中；
8. function定义中声明的所有局部变量寄存器都是静态的，即function中的局部寄存器在function的多个调用之间保持他们的值。

### Task

任务(task)类似于一般编程语言中的Process(过程)，它可以从描述的不同位置执行共同的代码。通常把需要共用的代码段定义为task,然后通过task调用来使用它。在task中还可以调用其他的task和function。

**task的定义：**

task<任务名>；

端口与类型说明；

变量声明；

行为语句；

endtask

**在定义一个task时，必须注意以下几点：**

1. task的定义结构不能出现在任何一个过程块内；
2. 一个task可以没有输入/输出端口，当然也可以有；
3. 一个task可以没有返回值，也可以通过输出端口或双向端口返回一个或多个值；
4. 除了task的输入和内部定义的变量之外，task还可以使用声明task的模块中的变量；
5. 在task内部定义的局部变量必须是reg类型，并且只能作用于task本身；
6. Task内部可以有timing control；
7. task的调用：task调用语句给出传入任务的参数值和接收结果的变量值

<任务名>  (端口1，端口2，... ，端口n)

1. 在调用task时，必须注意一下几点：

* task调用是过程性语句，因此只能出现在always过程块和initial过程块中，调用task的输入与输出参数必须是寄存器类型的；
* task调用语句中的列表必须与任务定义时的输入、输出和双向端口参数说明的顺序相匹配；
* 在调用task时，参数要按值传递，而不能按地址传递(和其他语言的不同)；
* 在一个task中，可也直接访问上一级调用模块中的任何寄存器；
* 可以使用循环中断控制语句disable来中断任务执行，在task被中断后，程序流程将返回到调用task调用的地方继续向下执行。
* Task的输入变量只有在调用时传递一次；

### Function和task的对比

**task和function的不同点**：

1. function只能与主模块共用一个仿真时间单位，而task可以定义自己的仿真时间单位；
2. function中只有组合逻辑，可以综合，task中没有时序逻辑的时候可以综合，有时序逻辑的时候不能综合；
3. function能调用function而不能调用task，而task可以调用task和function；
4. function至少需要一个输入变量，而task可以没有或者有很多个任意类型的变量；
5. function返回一个值，而task则不返回值。

**task和function的相同点：**

1. 调用function和task都是相当于直接复制电路单元，如果使用module的话可以通过切换不同输入实现分时复用，节省电路资源；

### Glitch free clock switch

# Analog

## PLL

# 附录

## SGDC Template

##=================================================

## Case analysis setting

##=================================================

## set\_case\_analysis 1 [get\_ports xxxx]

##=================================================

## Reset setting

##=================================================

#reset -name <rst-name> [-async | -sync] [-value <0 | 1>] [-soft]

reset -name presetn -async -value 0

reset -name swp\_rstn -async -value 0

##=================================================

## Static configure register

##=================================================

## quasi\_static -name KM6022\_swp\_host.u\_swp\_reg\_ctrl.u\_swp\_regs.

quasi\_static -name KM6022\_swp\_host.u\_swp\_reg\_ctrl.u\_swp\_regs.swp\_rate

## SDC Template

##=================================================

## Global setting

##=================================================

set over\_cons 1

set pclk pclk

set sclk swp\_clk

set pclk\_freq 40

set sclk\_freq 13.56

set pclk\_period [expr 1e9/(${over\_cons}\*${pclk\_freq}\*1e6) ]

set sclk\_period [expr 1e9/(${over\_cons}\*${sclk\_freq}\*1e6) ]

##=================================================

## Clock setting

##=================================================

#create\_clock -period $pclk\_period -waveform {0 [expr ${pclk\_period}/2]} -name $pclk [get\_ports $pclk]

#create\_clock -period $sclk\_period -waveform {0 [expr ${sclk\_period}/2]} -name $sclk [get\_ports $sclk]

create\_clock -period $pclk\_period -waveform {0 12.5 } -name $pclk [get\_ports $pclk]

create\_clock -period $sclk\_period -waveform {0 36.87} -name $sclk [get\_ports $sclk]

set\_clock\_groups -asynchronous -group [get\_clocks $pclk] -group [get\_clocks $sclk]

##=================================================

## Input/Output delay setting

##=================================================

set pclk\_in [get\_ports {presetn psel penable pwrite paddr pwdata}]

set pclk\_out [get\_ports {prdata pready pslverr swp\_intr}]

set sclk\_in [get\_ports {swp\_rstn swp\_usr\_i}]

set sclk\_out [get\_ports {swp\_usr\_o swp\_en}]

# set in\_list [all\_inputs]

# set out\_list [all\_outputs]

# remove\_from\_collection $in\_list [get\_ports $clk]

set pclk\_dly [expr $pclk\_period\*0.6]

set sclk\_dly [expr $sclk\_period\*0.6]

set\_input\_delay $pclk\_dly -clock $pclk $pclk\_in -add\_delay

set\_output\_delay $pclk\_dly -clock $pclk $pclk\_out -add\_delay

set\_input\_delay $sclk\_dly -clock $sclk $sclk\_in -add\_delay

set\_output\_delay $sclk\_dly -clock $sclk $sclk\_out -add\_delay

##=================================================

## Timing exception setting

##=================================================

## set max delay to capture clock period\*2

set\_max\_delay -from [get\_clocks $pclk] -to [get\_clocks $sclk] ${sclk\_period}

set\_max\_delay -from [get\_clocks $sclk] -to [get\_clocks $pclk] ${pclk\_period}

## DC get\_\*\* Command Example

## get\_clocks

* get\_clocks {clk0\_name …}

## get\_cells

* get\_cells top/subsystem/xxx
* get\_cells –hier {aes\_enc\*}
* get\_cells –hier –filter “is\_hierarchical === false && is\_mapped == true && ref\_name != \*\*logic\_0\*\* && ref\_name != \*\*logic\_1\*\* && ref\_name !~ gf\_cmos\*”
* get\_cells –hier –filter “full\_name =~xxx/\* && is\_sequential == false && is\_hierarchical == false && is\_mapped == true && ref\_name != \*\*logic\_0\*\* && @is\_unmapped=true && is\_black\_box==true”
* ==等于；!=不等于；=~匹配；!~不匹配

## get\_pins

* get\_pins –of\_objects [get\_cells –hier –filter ”ref\_name=~RF1P\*”] –filter “full\_name =~ \*/SD && lib\_pin\_name =~ FBDIV\* && direction != out”
* get\_pins –hier –filter “(full\_name =~ \*/MVDD\* || full\_name =~\*/vdd\*) && direction == inout”
* get\_pins xxx/xxx/A

## get\_object\_name

* get\_object\_name [get\_cells $xxx]
* get\_object\_name [all\_connected $xxx/X]
* get\_object\_name [get\_pins –of\_objects [get\_nets $xxx]]

## get\_attribute

* get\_attr [get\_cells –of\_object $port] (ref\_name || full\_name)
* get\_attr [filter [all\_clocks] “full\_name !~ v\_\*”] sources
* get\_attr [get\_pins $pin] (clocks || case\_value || pin\_direction)
* get\_attr [[get\_pins $pin] is\_clock\_pin] = “true”
* get\_attr $path points
* get\_attr $xxx arrival
* get\_attr $cell area

## get\_nets

* get\_nets xxx/xxx/xxx
* get\_nets –of\_objects [get\_pin xxx/xxx]

## get\_lib\_cell

* get\_lib\_cell {$lib\_name/\*BUF\_1}

## get\_designs

* get\_disign “xxx”

## Linux

### 压缩解压缩

压缩：

* tar –cvf jpg.tar \*.jpg //将目录里所有jpg文件打包成tar.jpg
* tar –czf jpg.tar.gz \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用gzip压缩，生成一个gzip压缩过的包，命名为jpg.tar.gz
* tar –cjf jpg.tar.bz2 \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用bzip2压缩，生成一个bzip2压缩过的包，命名为jpg.tar.bz2
* tar –cZf jpg.tar.Z \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用compress压缩，生成一个umcompress压缩过的包，命名为jpg.tar.Z

解压缩

* tar –xvf file.tar //解压 tar包
* tar -xzvf file.tar.gz //解压tar.gz
* tar -xjvf file.tar.bz2 //解压 tar.bz2
* tar –xZvf file.tar.Z //解压tar.Z