



路科芯院

---

# 路科验证笔试真题 100 题

---

大吉大利秋招训练营



JULY 12, 2019

路科验证  
助秋招一臂之力

## 为什么选择路科验证？

超过 10000 粉丝的订阅号 + 示范性微电子学院教材 + 资深验证专家 = 验证培训路科是最专业的！

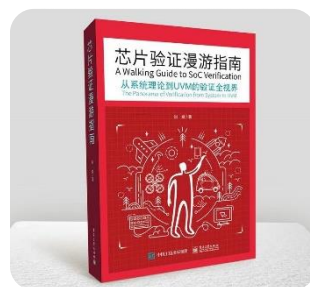
## 讲师介绍



### ● 路科验证创始人——路桑

知名外企通讯事业部资深验证专家，主持着验证架构规划和方法学研究，担任过几款亿门级通信芯片的验证经理角色。多次在国际知名的设计验证行业会议和展览中发表论文并进行演讲。

● 拥有超过 10000+关注量的垂直技术订阅号——路科验证，更新频率高，推文质量高，具有较大行业影响力。在这里，你可以看到最新行业资讯，可以看到验证的各种知识，也可以收获一群志同道合的朋友。



● 第一本示范性微电子学院的验证教材，第一本国内系统描述验证思想的书籍。这本集理论和实践为一体的综合性和实用性的图书，可作为芯片验证领域的学习用书和技术指南。



路科芯院

---

# PART ONE: VERIFICATION

---

### 1-1. 介绍常用的 EDA 验证工具和 debug 工具

目前行业内的验证工具按照验证方法来划分显得更为清晰，我们在标注工具的同时，也会附上其主要用途。在讲工具归属的时候，我们将分别采用前缀 S-（Synopsys），C-（Cadence）和 M-（Mentor）来表示。

动态验证方法依赖于仿真器（Simulator），包括 S-VCS，C-Incise & Xcelium，M-Questasim。

硬件加速模拟器（emulator），包括 S-Zebu，C-Palladium，M-Veloce。

形式验证工具（formal），包括 S-VC Formal，C-Jasper，M-Questa Formal。

仿真调试工具（debug），包括 S-Verdi，C-SimVision，M-Questa Visualizer Debug。

### 1-2. 描述 soc/ip 验证之间的区别，以及验证二者之间的侧重点

SoC 和 IP 从独立性来看，前者较后者更为独立，往往具备更加完整的功能。SoC 会由多个 IP、子系统和其它系统模块构成，从层次来看，IP 是构成 SoC 的重要组成部分。在验证 SoC 时，首先需要确保其 IP 级别都完成了验证，而在系统级别需要验证各个模块之间的交互和协调情况、集成连线情况，测试用例会更加真实，当然，仿真速度也下降很快，一般需要做门级仿真。在 IP 级验证时，如果是内部 IP，那么需要就接下来的运用场景（配置情况），展开重点性的验证，如果是向外部提供的 IP，那么需要针对其参数配置展开更为全面细致的验证工作，所以其特点不但是要求验证每一项功能，而且是每一项功能在不同配置下的行为是否是正确的。

### 1-3. 描述从芯片 spec 到 tapeout 的整个过程，重点介绍哪些步骤需要验证，以及所需的文件和验证重点

- a. 从 spec 到模块 RTL 时，除了 RTL 文件，还需要寄存器文件来生成寄存器模型，构建 UVM 验证环境，主要验证每一项 RTL 功能。
- b. 从模块到子系统时，除了之前的文件，如果在子系统级别需要模拟电源域开关，那么还需要 UPF，如果子系统单独综合且较为独立，可能还需要做门级仿真，那么需要综合网表和 SDF 文件，验证的重点将是子系统的各项完整功能。

- c. 在系统级别时，除了系统级的 RTL 仿真，也需要进行 UPF 仿真和门级仿真，因此也需要对应的 UPF 文件、网表和 SDF 文件，验证的重点是各个子系统之间的交互和协调情况、集成连线情况。

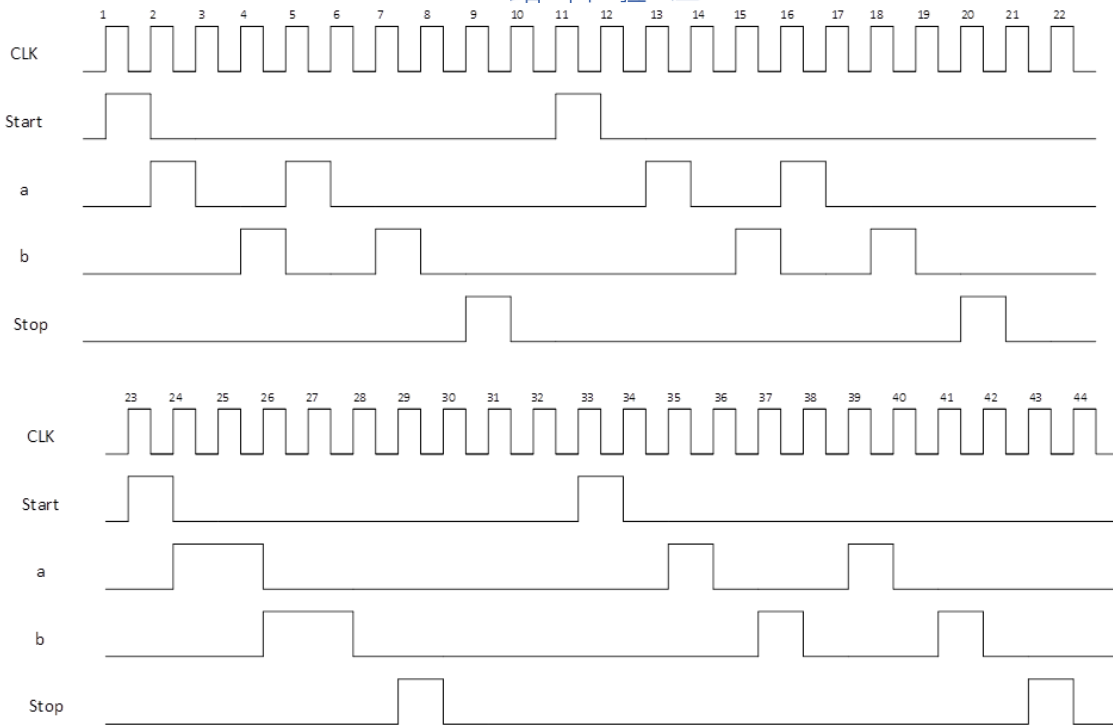
1-4. 下列关于代码覆盖率描述错误的是： CD

- A. 代码覆盖率包括语句覆盖率
- B. 代码覆盖率包括条件覆盖率
- C. 代码覆盖率包括功能覆盖率
- D. 代码覆盖率到达百分百说明代码 bug 已消除

代码覆盖率和功能覆盖率是独立的两种覆盖率，代码覆盖率 100% 只能表明代码经过了充分的执行，但是代码中是否有 bug 以及 bug 是否会被发现，取决于验证环境中的监测点是否监测了关键信号以及对这些信号的判断是否正确。

1-5. 以下断言在哪个时钟沿开始的时序可以判决成功

```
property test_seq_2;  
  @(posedge clk) $rose(start) |->  
    ##3 ((a##2 b)[*2]) ##2 stop;  
endproperty  
assert property(test_seq_2);
```



将在第 21 个时钟沿判决成功。

属性 `test_seq_2` 表示的是，在 `start` 拉高后的第 2 拍（用的是 `|->3`），要求【`a` 为高且再 2 拍后 `b` 为高】的序列重复两次，即 `(a##2b) [*2]` 可以等同于 `(a##2b ##1 a##2b)`，在最后一次 `b` 为高的 2 拍后，`stop` 应该为高。

这个属性的判决将在从 11 拍被激发，而在 21 拍（而不是 20 拍）在采样到 `stop` 为 1 时，判决成功。需要注意另外一个 `sequence` 从 33 拍到 44 拍的判决的区别。

#### 1-6. 描述一下 `code coverage` 和 `function coverage` 的区别，为什么需要这些 `coverage`?

- 没有任何一种单一的覆盖率可以完备地去衡量验证过程
- 即使我们可以达到 100% 的代码覆盖率，但这并不意味着 100% 的功能覆盖率。原因在于代码覆盖率并不是用来衡量设计内部的功能运转，或者模块之间的互动，或者功能时序的触发等。代码覆盖率可以通过仿真器完成自动收集。
- 类似地，我们即便达到了 100% 功能覆盖率，也可能只达到了 90% 的代码覆盖率。原因可能在于我们疏漏了去测试某些功能，或者一些实现的功能并没有被描述。功能覆盖率需要通过人为定义，与拆分的待测功能点做一一映射。

- 从上述关于代码覆盖率和功能覆盖率简单的论述就可以证明，如果想要得到全面的验证精度，我们就需要多个覆盖率种类的指标。

1-7. 以下关于验证的描述，正确的是 AB

- A. SystemVerilog 区别于 verilog 的一个重要特性是其具有所有面向对象语言的特性：封装继承和多态
- B. UVM 是 synopsys, cadence, mentor 等 EDA 厂商联合发布的验证平台
- C. 验证平台使用 checker 监测 DUT 行为，只有知道 DUT 的输入输出信号变化之后，才能根据这些信号变化来判断 DUT 的行为是否正确
- D. SystemVerilog, Verilog, systemC, uvm 都是验证常用的硬件语言

注意，答案 C 中，监测 DUT 行为的组件是 monitor，答案 D 中的 UVM 并不是一种语言，而是基于 SystemVerilog 之上的验证方法学。对于答案 B，UVM 是 Accellera 推出的验证平台标准，不过背后的推动实际上仍然是基于三家 EDA 厂商的统一意见。

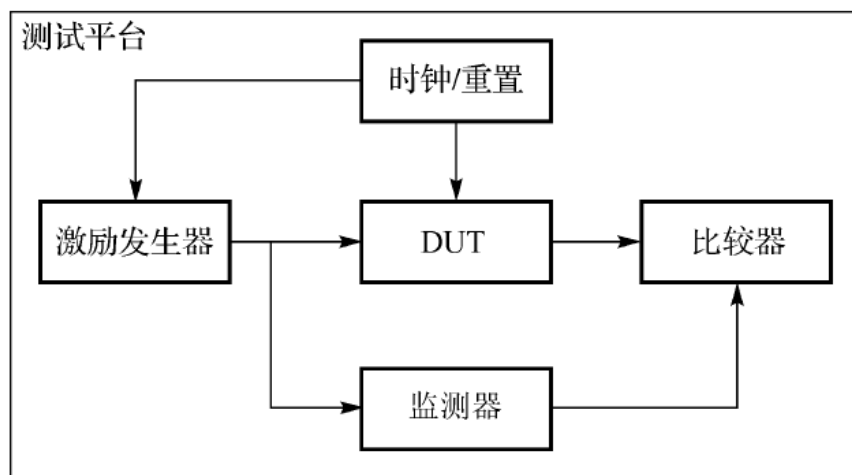
1-8. 在验证中，一种常用的方法是将输入激励同时给参考模型及被测试设计，然后比较他们的响应以确定设计是否符合预期，请问在比较其响应时需要注意什么？

如果参考模型是非时序的，则主要比较数据内容。如果参考模型有时序模拟，那么还应该比较实际输出数据和期望输出数据在时序上的差异。

对于一些数据可能经过重组，而无法通过单一参考模型来比较的情况，除了在比较数据完整性方面需要考虑，还需要针对数据重组、打包、排序等设计功能设置独立的检查机制，确保所有设计功能都得到监测和检查。

对于比较数据时的调试需要注意，无论比较成功还是失败，都应该打印必要的比较信息，便于调试。打印的信息中，应该有消息源、数据比较信息、失败信息可能原因等。如果比较失败，一般需要停止仿真，便于在比较失败现场进行调试。

1-9. The SOC design is more and more complex, and the verification is becoming more and more important for us. A very important step for verification is to setup the verification environment. Please draw a figure to show a common verification environment 's structure and give a short description for each component in the structure.



参考于《芯片验证漫游指南》

- 测试平台（testbench）是整个验证系统的总称。它包括验证结构中的各个组件、组件之间的连接关系、测试平台的配置和控制。从更系统的意义来讲，它还包括编译仿真的流程、结果分析报告和覆盖率检查等。
- 测试平台的各个组件之间是相互独立的。验证组件与设计之间需要连接，而验证组件之间也需要进行通信。验证环境也需要时钟和复位信号的驱动。
- Stimulator（激励发生器）是验证环境的重要部件，在一些场合中，它也被称为 driver（驱动器）、BFM（bus function model，总线功能模型），或者 generator（发生器）。Stimulator 的主要职责是模拟与 DUT 相邻设计的接口协议，只需要关注于如何模拟接口信号，使其能够以真实的接口协议来发送激励给 DUT。
- Monitor（监测器）的主要功能是用来观察 DUT 的边界或者内部信号，并且经过打包整理传送给其它验证平台的组件，例如 checker（比较器）。从监测信号的层次来划分 monitor 的功能，它们可以分为观察 DUT 边界信号和观察 DUT 内部信号。
- Checker（比较器）都应当是最需要时间投入的验证组件了，它肩负了模拟设计行为和功能检查的任务。Checker 一般会缓存从各个 monitor 收集到的数据，也可能将 DUT 输入接口侧的数据汇集给内置的 reference model（参考模型）。

#### 1-10. 填空题

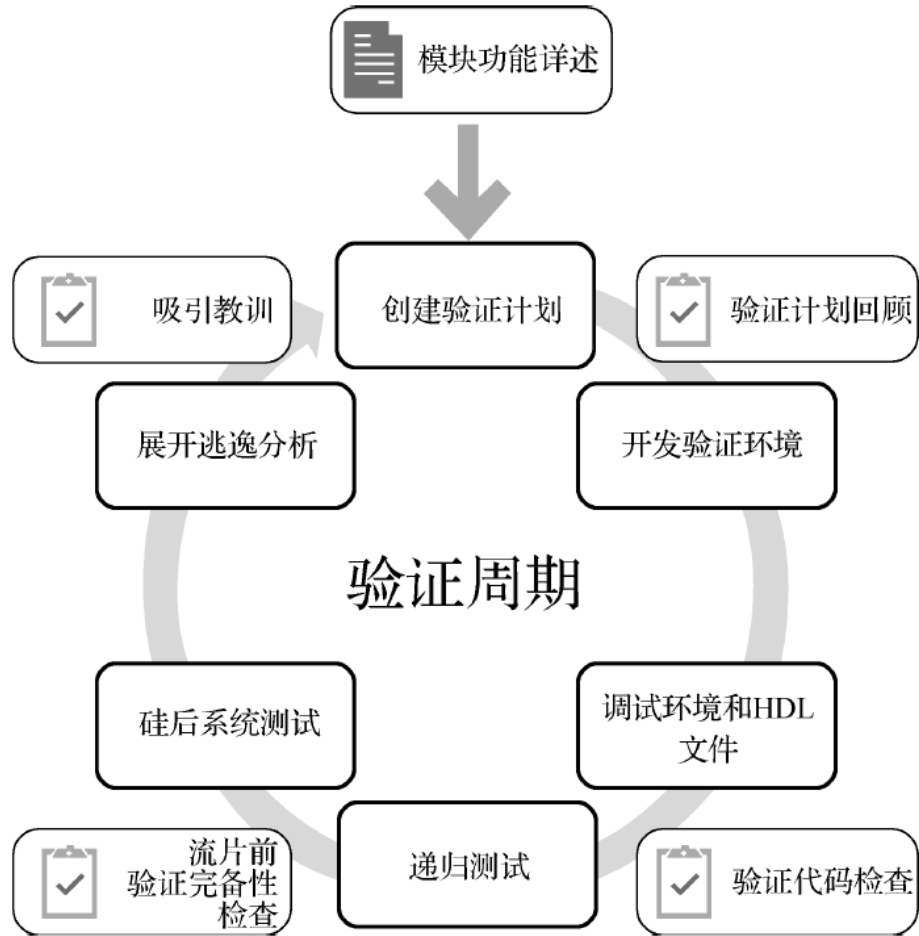
- 验证中，代码覆盖率是指（衡量哪些设计代码在激活触发，而哪一些则一直处于非激活状态的统计数据）。



- b. SystemVerilog 中，从一个类派生一个新类的关键字是（extends）
- c. SystemVerilog 中，仿真器运行一个用例需要建立多个子线程，这些子线程结束时间各不相同，此时需要使用（wait fork）语句来等待所有的线程结束
- d. SystemVerilog 中，`int_data[]={9,1,8,3,4,4}`；执行 `data.reverse()` 操作后，`data[]` 的值应该是（{4,4,3,8,1,9}）。执行 `data.rsort()` 操作后，`data[]` 的值应该是（{9,8,4,4,3,1}）。
- e. SystemVerilog 中，使用随机函数产生随机数赋值给信号 `a[11:0]`，随机范围为 3~255：（`$urandom_range(3,255)`）
- f. SystemVerilog 创建一个数据类型为 `int` 的动态数组 `a`：（`int a[]`），创建一个数据类型为 `int` 的队列 `b`：（`int b[$]`）
- g. SystemVerilog 中，如何在 `int` 类型的队列 `queue` 的后面插入数据 `data`：  
（`{queue, data}`）
- h. SystemVerilog 中，`rand int src; constrain c_dist{0 :=40, [1:2] :=60};`，此时 0, 1, 2 的权重分别是多少？ 40, 60, 60
- i. SystemVerilog 中，`rand bit[5:0]; constraint c_data {data inside [$:5], [30:$]}`，那么变量 `data` 的取值范围是（{[0:5], [30:63]}）

#### 1-11. 通用验证过程中，需要执行的关键步骤？

- a. 阅读功能描述文档
- b. 准备验证计划
- c. 提取验证功能点
- d. 绘制验证结构
- e. 实现验证环境
- f. 构建测试用例
- g. 定义功能覆盖率
- h. 发现缺陷和验证设计修复
- i. 回归测试
- j. 收集覆盖率
- k. 分析覆盖率、修改或添加测试用例
- l. 最终全面通过回归测试和完成 100%覆盖率要求



参考于《芯片验证漫游指南》

## 1-12. 什么是断言，断言在验证中的作用，断言的分类及各自的含义

断言是设计的属性的描述。如果一个在仿真中被检查的属性不像我们期望的那样表现，那么这个断言将失败。如果一个被禁止在设计中出现的属性在仿真过程中发生，那么这个断言也将失败。一系列的属性可以从设计的功能描述中推知，并且被转换成为断言。这些断言可以在功能仿真中不断地被监视。

断言可以分为即时断言和并发断言。即时断言是基于仿真事件的语义，它的表达式的求值就像在过程块中其它表达式一样。即时断言本质不是时序相关的，而是立即被求值。并发断言是基于时钟周期，它是基于时钟周期的，它会在时钟边缘根据调用的变量采样值计算测试表达式。

### 1-13. 请描述您对 UVM phase, 以及 UVM objection 的理解, 并简述 UVM phase 执行与 objection 之间的关系

九个 phase 对于一个测试环境的生命周期而言, 是有固定的先后执行顺序的; 同时对于同一个 phase 中的组件, 执行也会按照层次的顺序或者自顶向下、或者自底向上来执行。

phase	函数/任务	执行顺序	功能	典型应用
build	函数	自顶向下	创建和配置测试平台的结构	创建组件和寄存器模型, 设置或获取配置
connect	函数	自底向上	建立组件之间的连接	连接 TLM/TLM2 的端口, 连接寄存器模型和 adapter
end_of_elaboration	函数	自底向上	测试环境的微调	显示环境结构, 打开文件, 为组件添加额外配置
start_of_simulation	函数	自底向上	准备测试环境的仿真	显示环境结构, 设置断点, 设置初始运行的配置值
run	任务	自底向上	激励设计	提供激励, 采集数据和数据比较, 与 OVM 兼容
extract	函数	自底向上	从测试环境中收集数据	从测试平台提取剩余数据, 从设计观察最终状态
check	函数	自底向上	检查任何不期望的行为	检查不期望的数据
report	函数	自底向上	报告测试结果	报告测试结果, 将结果写入到文件中
final	函数	自顶向下	完成测试活动结束仿真	关闭文件, 结束联合仿真引擎

参考于《芯片验证漫游指南》

uvm\_objection 类提供了一种供所有 component 和 sequence 共享的计数器。如果有组件来挂起 objection, 那么它还应该记得落下 objection。参与到 objection 机制中的参与组件, 可以独立的各自挂起 objection, 来防止 run phase 退出, 但是只有这些组件都落下 objection 后, uvm\_objection 共享的 counter 才会变为 0, 这意味 run phase 退出的条件满足, 因此可以退出 run phase。

### 1-14. 请谈一下您对 disable fork 以及 wait fork 的理解, 利用 fork 线程, 请编码示意如何实现 timeout 的检查。

fork...join\_any 和 fork...join\_none 继续执行后, 其一些未完成的子程序仍将在后台运行。如果要等待这些子程序全部完成, 或者停止这些子程序, 可以使用 wait fork 或者 disable fork。

```
fork
    thread1();
begin #timeout; disable fork; end
join_any
```

1-15. 在验证工作中，对于激励或者配置的随机化，在进行随机过程中需要注意哪些方面？

- 如果激励或者配置已经有约束，那么在随机化时可能会添加外部约束（内嵌约束 `with{ }`），那么应该保证外部约束和类原有约束不发生冲突从而使得导致随机化失败。
- 对于一些随机化失败的场景，应该从仿真日志中获得调试信息，理解哪些约束发生了冲突。
- 无论是激励还是配置，都应该与测试功能点有关，应当避免与测试目的无关的、相违背的随机数据产生。
- 在同一个仿真、或者不同的仿真中，都可以变化随机约束，使得之前的覆盖率可以影响并驱动新的随机数据，继而不断提高覆盖率。
- 配置的随机化应该只在 `build()` phase 中完成，避免仿真过程中对配置做二次随机，从而发生不可预期的情况。

1-16. 请说明 TLM 中 `analysis_port` 和 `analysis_fifo` 之间的不同之处，并给出二者各自的使用场景。

- `Analysis port` 是一种 TLM 端口，可以实现一对多的连接，`monitor` 往往会使用 `analysis port` 将其数据写出到其它的 `subscriber` 组件，例如 `scoreboard`，`coverage collector` 等。
- `Analysysi fifo` 本身是一种 TLM FIFO 缓存，它继承于 `uvm_tlm_fifo`，在具备已有 TLM 端口的同时还具备了 `analysis imp`，它可以用来与另外一端组件（例如 `monitor`）的 `analysis fifo` 完成 TLM 端口的连接，继而从 `initiator` 端接收数据并加以缓存。

- 将 initiator 的 analysis port 连接到 tlm\_analysis\_fifo 的 analysis\_export 端口，这样数据可以从 initiator 发起，写入到各个 tlm\_analysis\_fifo 的缓存中。
- 将多个 target 的 get\_port 连接到 tlm\_analysis\_fifo 的 get\_export，注意保持端口类型的匹配，这样从 target 一侧只需要调用 get() 方法就可以得到先前存储在 tlm\_analysis\_fifo 中的数据。

```
initiator.ap.connect(tlm_analysis_fifo1.analysis_export); target1.get_port.  
connect(tlm_analysis_fifo1.get_export);  
initiator.ap.connect(tlm_analysis_fifo2.analysis_export); target2.get_port.  
connect(tlm_analysis_fifo2.get_export);  
initiator.ap.connect(tlm_analysis_fifo3.analysis_export); target3.get_  
port.connect(tlm_analysis_fifo3.get_export);
```

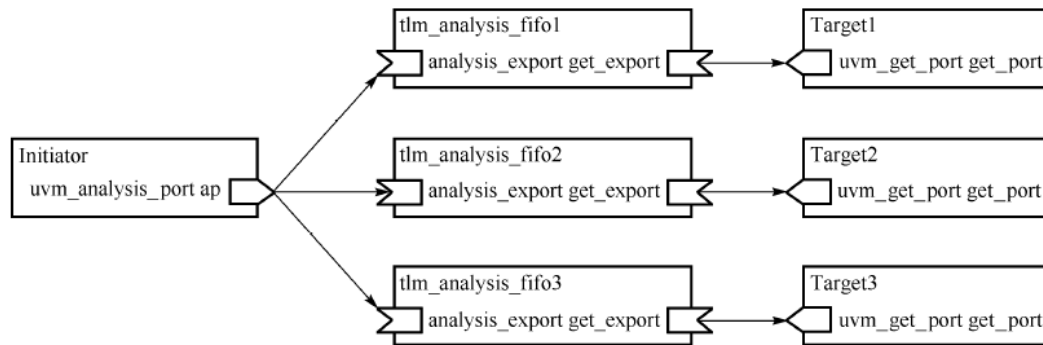


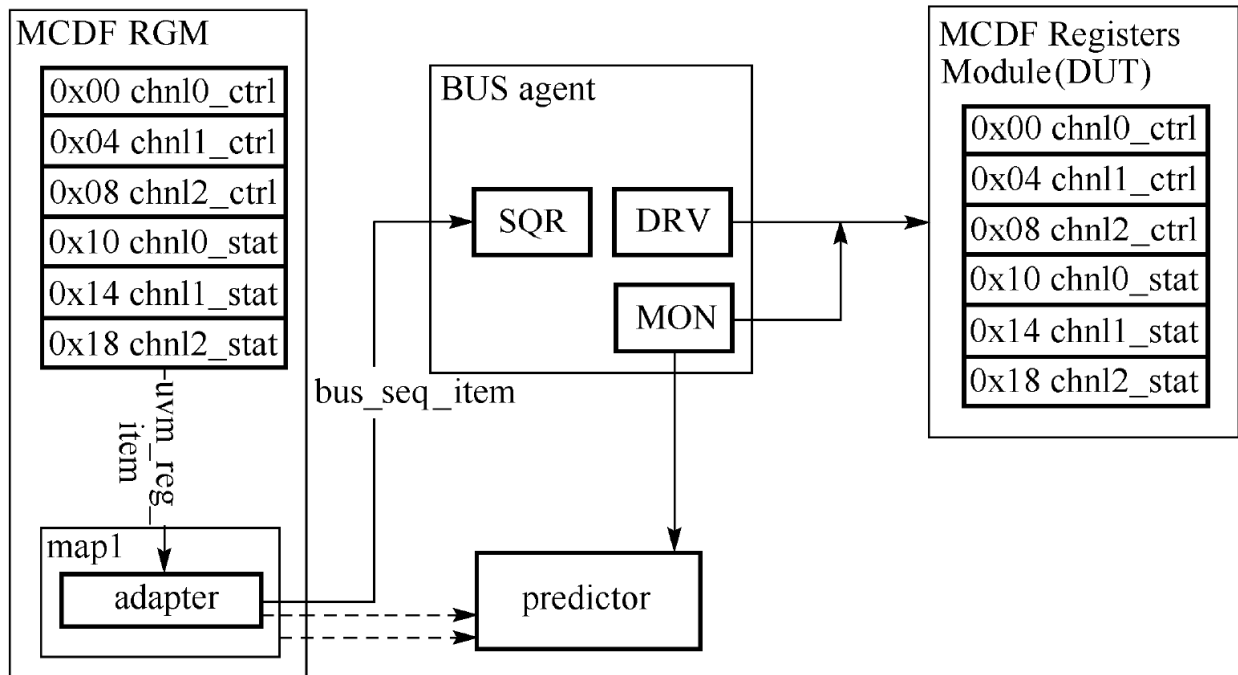
图 12.12 analysis TLM FIFO 的连接实例

参考于《芯片验证漫游指南》

1-17. 请简述 UVM RAL model 的使用机制，同时解释一下 adapter 具体做了哪些操作，并简述您对 predictor 的理解。

- 寄存器模型（RAL model）是硬件寄存器模块在 UVM 一侧构建的模型。其内部的寄存器信息包括所有的域信息（field）、地址信息（address map）、可读写方式、复位值与硬件一侧都是一一对应的。
- 寄存器模型可以在更高的抽象级用来对寄存器进行访问，继而提高寄存器访问的可读性和可维护性，同时由于其提供的后门访问方式，也可以加速寄存器的配置和访问。
- 寄存器模型本身也可以用来实现对寄存器测试的自动化，继而提高寄存器测试的效率。
- Adapter 在寄存器访问中完成了抽象级 transaction 转换的任务，即其内部的 reg2bus() 和 bus2reg() 任务，继而实现了两种不同抽象级访问事务之间的桥接。

- Predictor 的预测任务需要其它几个组件的参与，包括 uvm\_reg\_map, bus monitor 和 adapter。其原理是利用 monitor analysis port 收集到的 bus transaction，利用 adapter::bus2reg() 完成类型转换，继而利用 uvm\_reg\_map 更新对应的 uvm\_reg 的镜像值（mirror value），完成寄存器值的预测，便于后期的便捷访问和寄存器值比较。



参考于《芯片验证漫游指南》



路科芯院

---

## PART TWO: DESIGN

---

2-1. 判断电路是否存在竞争冒险的方法有哪些呢? A, B, C, D

- A. 代数法
- B. 卡诺图法
- C. 实验法
- D. 观察法

逻辑冒险的判断方法(详见题 2-66):

1. 代数法: 在逻辑函数表达式中, 若某个变量同时以原变量和反变量两种形式出现, 就具备了竞争条件。去掉其余变量(也就是将其余变量取固定值 0 或 1), 留下有竞争能力的变量, 如果表达式为  $F=A+A^{\sim}$  (因为上横杠打不出来, 故用  $A^{\sim}$  表示 A 的反变量, 下同), 就会产生 0 型冒险 (F 应该为 1 而实际却为 0); 如果表达式为  $F=AA^{\sim}$ , 就会产生 1 型冒险。一例一: 表达式  $F=AB+CB^{\sim}$ , 当  $A=C=1$  时,  $F=B+B^{\sim}$ , 在 B 发生跳变时, 可能出现 0 型冒险。

2. 卡诺图法: 将函数填入卡诺图, 按照函数表达式的形式圈好卡诺圈。

A\BC 00 01 11 10

0 0 0 0 1

1 0 1 1 1

$F=AC+BC^{\sim}$  的卡诺图 (将 101 和 111 的 1 圈一起, 010 和 110 的 1 圈一起, 这里不好表示, 自己画在纸上)

通过观察发现, 这两个卡诺圈相切。则函数在相切处两值间跳变时发生逻辑冒险。

(前提是这两个卡诺圈没有被其他卡诺圈包围)

2-2. Verilog 语言中, 下面哪些语句不可被综合: A, B

- A. #delay 语句
- B. initial 语句
- C. always 语句
- D. 用 generate 语句产生的代码

所有综合工具都不支持的结构

time, defparam, \$finish, fork, join, initial, delays, UDP, wait

2-3. 关于数字通信的特点, 下面描述不正确的是 ( ) A, B, D



- A. 抗干扰能力强，且噪声不积累
- B. 易于保密，保密性好
- C. 比模拟通信占据更窄的系统频带，系统设备简单，对同步要求更低
- D. 易于集成，使通信设备微型化

2-4. 以下关于 Latch 与 Flip\_flop 特性描述正确的是: A, B, D

- A. Latch 与 Flip\_flop，都属于时序逻辑
- B. Flip\_flop 只会在时钟触发沿采样当前输入，产生输出
- C. Latch 无时钟输入
- D. Latch 输出可能产生毛刺

2-5. 下面哪些是非易失性存储器? AB

- A. flash
- B. EPROM
- C. DRAM
- D. SRAM

2-6. 请指出多项式  $A \oplus \overline{CB}(\overline{AC} \overline{D} + \overline{AC} \overline{D})$  与下面哪一个具有逻辑等价关系 A

- ☐  $\overline{A} \overline{C} + AC + \overline{B} \overline{D}$
- ☐  $A \overline{B} + A \overline{C} + BCD$
- ☐  $A \overline{C} \overline{D} + \overline{B} D$
- ☐  $AB \overline{C} + \overline{A} C + BD$

2-7. 下面是芯片中有关 GPIO 的描述，不正确的是：C

- A. GPIO 的引脚一般是多功能复用的
- B. GPIO 作为输出接口时具有锁存功能
- C. GPIO 一般只有 0 态和 1 态，不具有高阻态
- D. GPIO 作为输入接口时具有缓冲功能

为防止信息相互干扰，要求凡挂到总线上的寄存器或存储器等，它的输入输出端不仅能呈现 0、1 两个信息状态，而且还应能呈现第三个状态——高阻抗状态

施密特触发输入的作用是将缓慢变化的或者是畸变的输入脉冲信号整形成比较理想的矩形脉冲信号

2-8. 下面关于网表仿真的描述正确的是 C

- A. 网表仿真的速度比 RTL 仿真的速度更快
- B. 网表仿真不能发现实现约束的问题
- C. 网表仿真可以发现电路设计中的异步问题
- D. 为了保证芯片的正常工作，即使在时间和资源紧张的情况下，也要将所有 rtl 仿真用例都进行网表仿真，并确保都通过

注：

异步的问题在网表仿真的时候可以发现，但不是必然会发现。这和 testcase 的选择，clock 的设置，仿真时间长短都有关系。所以对于选项 C，可以发现仅是“可以”。

稳妥的想法是将所有的 timing path 在网表级都测试一遍。但是往往不容易实现。

小规模的设计，应用要求非常高的产品，需要所有的 testcase 都通过网表仿真。

但 SoC 的规模都很大，涉及多时钟域，多电压模式，多时钟匹配，网表的仿真运行及其耗时。在项目按时交付的压力下要选择关键，有效的仿真用例，设置优先级加入仿真队列。

所以选项 D 的描述在目前 SoC 设计项目中不能全部实现。

2-9. 下列关于格雷码的描述哪些是正确的？A B C D

- A. 格雷码 0110 对应的二进制数是 0100
- B. 格雷码从编码形式上杜绝了逻辑冒险的存在
- C. 卡诺图的坐标是按照格雷码的顺序标注的

- D. 格雷码相邻两个码组间仅有一位不同
- E. 格雷码常用于提高单一时钟域内总线数据的可靠性

- 1、格雷码的特点是任意两组相邻之间只有一位不同，其余各位都相同，而且 0 和最大数（2 的 N 次方减一）对应的两组格雷码之间也有一位不同。
- 2、格雷码是一种循环码，它的特性使它在形成和传输过程中引起的误差较小。如计数电路按格雷码计数时，电路每次状态更新只有一位代码变化，从而减少了计数错误。
- 3、普通二进制码与格雷码相互转换关系为：

（1）二进制码转换成格雷码

从最右边第一位开始，依次将每一位与左邻一位异或(XOR)，作为对应格雷码该位的值，最左边一位不变。

（2）格雷码转换成二进制码

从左边第二位起，将每位与左边一位解码后的值异或(XOR)，作为该位解码后的值（最左边一位依然不变）。

十进制数	自然二进制数	格雷码	十进制数	自然二进制数	格雷码
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

2-10. 从奈奎斯特采样定理得出，要使实信号采样后能够不失真还原，采样频率  $f$  与信号最高频率  $f_s$  的关系是： (A)

- A.  $f$  大于等于  $2f_s$
- B.  $f$  小于等于  $2f_s$

- C.  $f$  大于等于  $f_s$
- D.  $f$  小于等于  $f_s$

在进行模拟/数字信号的转换过程中，当采样频率  $f_{s, \max}$  大于信号中最高频率  $f_{\max}$  的 2 倍时 ( $f_{s, \max} > 2f_{\max}$ )，采样之后的数字信号完整地保留了原始信号中的信息，一般实际应用中保证采样频率为信号最高频率的 5~10 倍；采样定理又称奈奎斯特定理。

2-11. 下列关于 FIFO 的描述正确的是 BD

- A. 外部可以直接操作 FIFO 的读写地址
- B. FIFO 可以分为同步 FIFO 和异步 FIFO
- C. 空信号是在写时钟域产生的，满信号是在读时钟域产生的
- D. FIFO 是先进先出的存储器

FIFO 是英文 First In First Out 的缩写，是一种先进先出的数据缓存器，它与普通存储器的区别是没有外部读写地址线，这样使用起来非常简单，但缺点就是只能顺序写入数据，顺序的读出数据，其数据地址由内部读写指针自动加 1 完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

FIFO 按工作时钟域的不同又可以分为：同步 FIFO 和异步 FIFO。

同步 FIFO 的写时钟和读时钟为同一个时钟，FIFO 内部所有逻辑都是同步逻辑，常常用于交互数据缓冲。异步 FIFO 的写时钟和读时钟为异步时钟，FIFO 内部的写逻辑和读逻辑的交互需要异步处理，异步 FIFO 常用于跨时钟域交互。

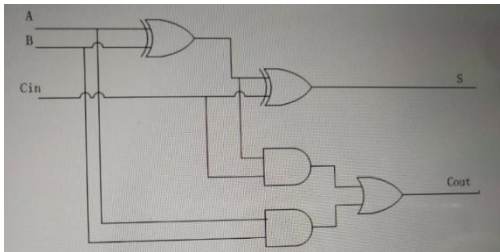
FIFO 空满状态产生：

为产生 FIFO 空满标志，引入 FIFO Count 计数器，FIFO Count 寄存器用于指示 FIFO 内部存储数据个数；

- (1) 当只有写操作时，FIFO Count 加 1；只有读操作是，FIFO Count 减 1；其他情况下，FIFO Count 保持；
- (2) 当 FIFO Count 为 0 时，说明 FIFO 为空，`fifo_empty` 置位；
- (3) 当 FIFO Count 等于 FIFO\_DEPTH 时，说明 FIFO 已满，`fifo_full` 置位

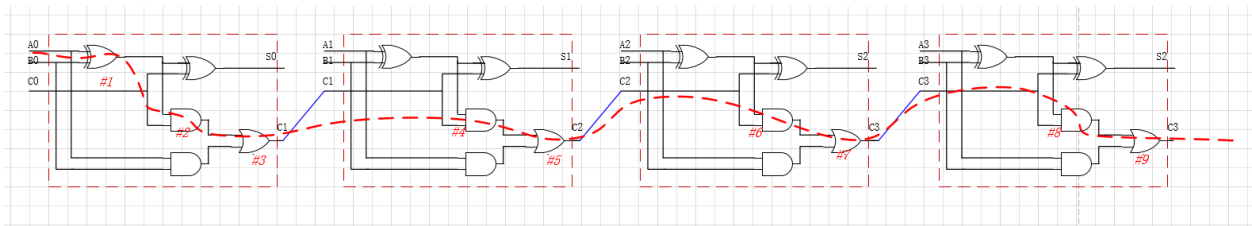
2-12. 下图为一个 full adder 全加器电路图，假设每个门延迟都为 T，不考虑延迟和扇

入扇出，下面说法正确的是 **D**



- A. 4 位 carry-lookahead adder 最大延迟为 3T
- B. 8 位 ripple-carry adder 最大延迟为 9T
- C. 8 位 carry-lookahead adder 最大延迟为 4T
- D. 4 位 ripple-carry adder 最大延迟为 9T

注：



2-13. 下面哪种逻辑门可以实现逻辑  $(A \oplus B) \text{ OR } (C \text{ AND } D)$ ? **AC**

- A. INV
- B. NOR
- C. NAND
- D. XOR

$$(AB' + A'B) + CD = (AB' + A'B + CD)' = (AB')' \cdot (A'B)' \cdot CD'$$

2-14. A 和 B 为补码表示的二进制，其中 A=10010010B，B=10001011B，请问下列选项中哪一个为 A+B 的运算结果 **C**

- A. 1100011101B
- B. 111100010B
- C. 100011101B
- D. 011100011B

2-15. 下列关于 verilog 的描述正确的是：ABCD

- A.  $Y=a+b$ ; 属于阻塞赋值语句, 执行该语句时, 先计算  $a+b$  的值, 然后更新  $y$  值, 在此过程中, 不能运行其他语句
- B. Generate, for, function 语句可综合
- C. 如果  $A=1'b1, B=1'b0, F=A\&\sim B|B\&\sim A||B$ , 则  $F=1'b1$
- D. 如果  $A=4'hb$ , 则  $\sim A=1'b1$

运算符的优先级

类 别	运 算 符	优 先 级
逻辑、位运算符	! ~	<div>高</div> <div style="text-align: center;">↓</div> <div>低</div>
算术运算符	* / %	
	+ -	
移位运算符	<< >>	
关系运算符	< <= > >=	
等式运算符	= = ! = == = ! ==	
缩减、位运算符	& ~&	
	^ ^~	
	~	
逻辑运算符	&&	
条件运算符	? :	

2-16. 关于异步处理以下说法正确的是 A

- A. 静态配置信号可以不做异步处理
- B. 异步处理需要考虑发送和接收时钟之间的频率关系 (注: 异步 FIFO 两侧的时钟就没有考虑)
- C. 异步 FIFO 采用格雷码的原因是为了提高电路速度
- D. 单比特信号打两拍后可以避免亚稳态的发生 (注: 准确的说法是降低概率)

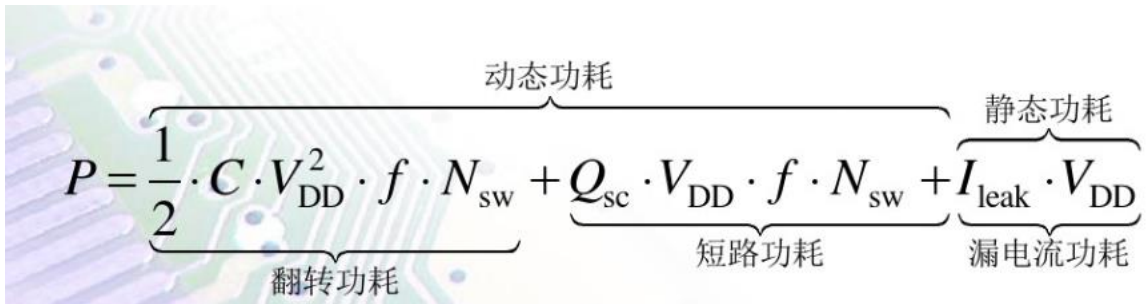
2-17. 如下为 verilogHDL 描述的一段程序，请选择对它产生的波形描述正确的是： CD

```
always
begin
    # 5 clk=0,
    # 10 clk=~clk
end
```

- A. Clk=0
- B. clk=1
- C. 周期为 15
- D. 占空比为 1/3 的时钟

2-18. 以下哪些手段可以降低 SRAM 的动态功耗 ABC

- A. 不访问 SRAM 时关闭时钟
- B. 不访问 SRAM 时地址线不翻转
- C. 不访问 SRAM 时写数据线不翻转
- D. 不访问 SRAM 时，将其 power down



$$P = \underbrace{\frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f \cdot N_{sw}}_{\text{翻转功耗}} + \underbrace{Q_{sc} \cdot V_{DD} \cdot f \cdot N_{sw}}_{\text{短路功耗}} + \underbrace{I_{leak} \cdot V_{DD}}_{\text{漏电流功耗}}$$

动态功耗      静态功耗

2-19. Please describe the synthesis result of bellow code, What's the issue of design and how to optimize this design.

```
always @( G or D )begin
    if (G)Q=D;
```

end

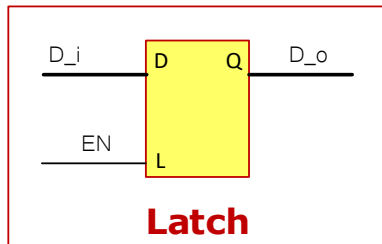
学生应该理解掌握时序单元最基本的语言描述。

Latch

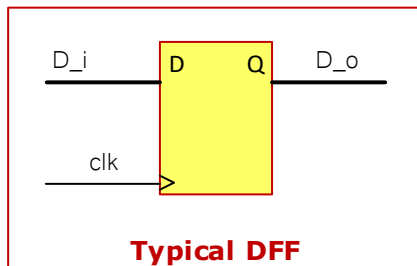
DFF

DFF with async-reset

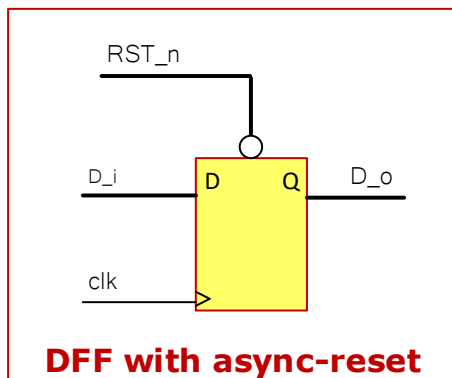
DFF with sync-reset



```
Process(EN,D_i)
begin
    if (EN='1') then --no else branch and clk edge missing
        D_o <= D_i;
    end if;
end process;
```

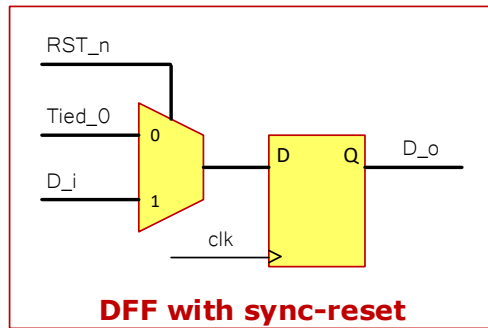


```
Process(clk)
begin
    if (clk'event and clk='1') then
        D_o <= D_i;
    end if;
end process;
```



```
Process(clk,RST_n)
begin
    if (RST_n='0') then
        D_o <= '0';
    elsif (clk'event and clk='1') then
        D_o <= D_i;
    end if;
end process;
```

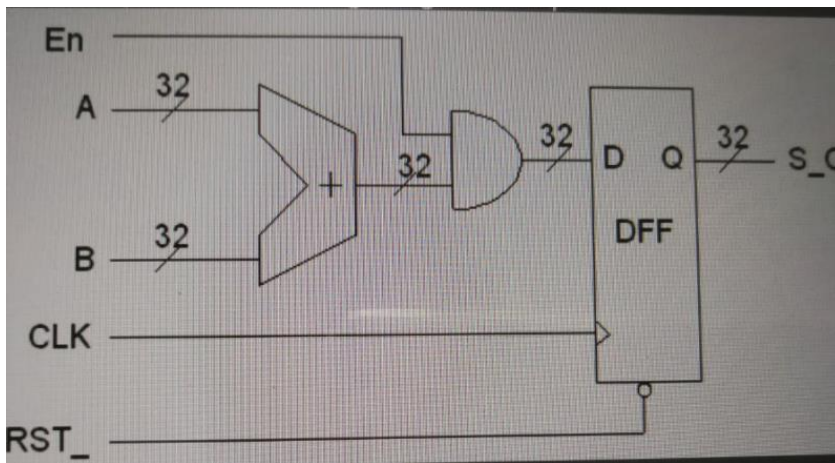


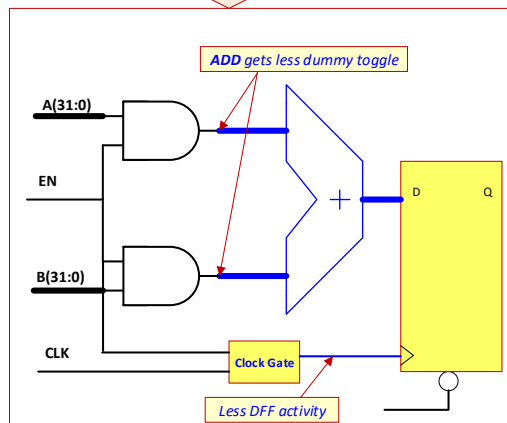
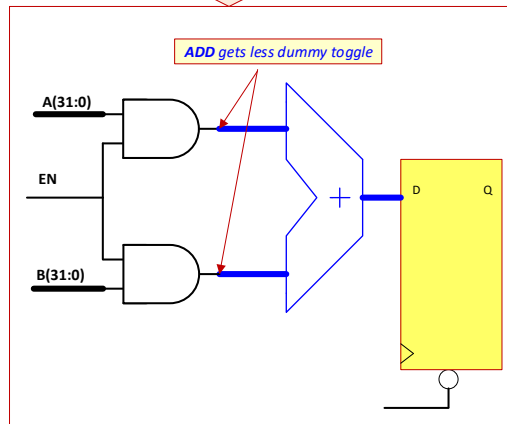
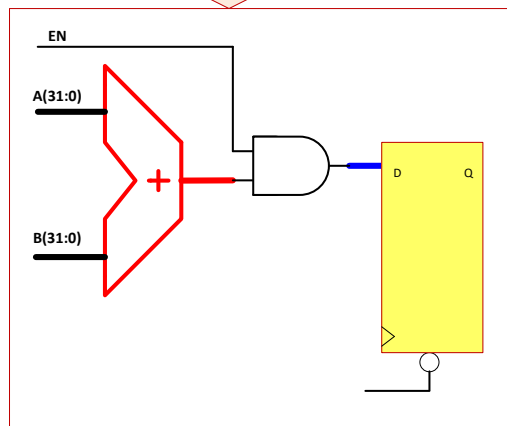
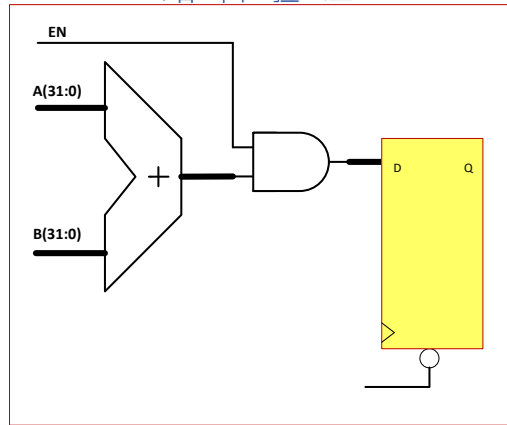


```

Process (clk) --only clk in sensitive list
begin
    if (clk'event and clk='1') then
        if (RST_n='0') then
            D_o <= '0';
        else
            D_o <= D_i;
        end if;
    end if;
end process;
    
```

2-20. Please improve the following design to save power





## 2-21. Please design rising edge detection for an input asynchronous signal (please write Verilog code )

要实现边沿检测，最直接的想法是用两级寄存器，第二级寄存器锁存住某个时钟上升沿到来时的输入电平，第一级寄存器锁存住下一个时钟沿到来时的输入电平，如果这两个寄存器锁存住的电平信号不同，就说明检测到了边沿，具体是上升沿还是下降沿可以通过组合逻辑来实现。如下图所示：

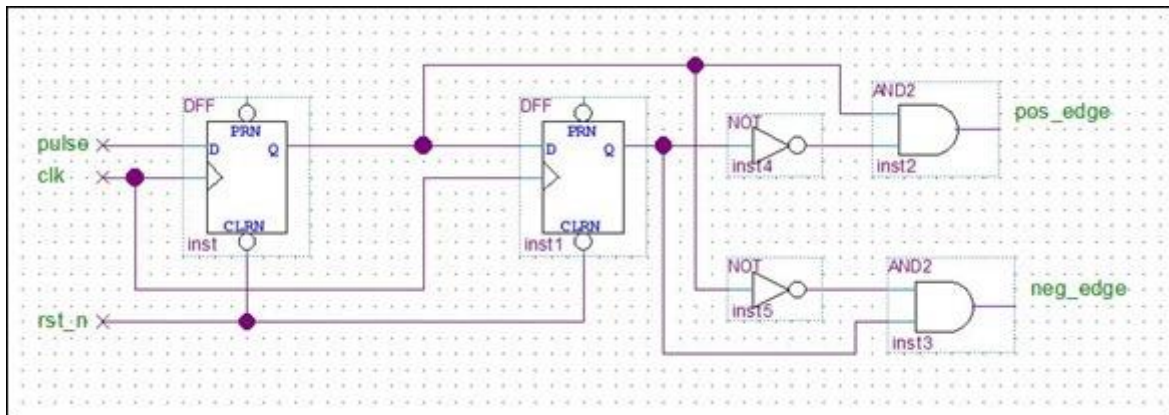


图 1 用两级寄存器实现边沿检测

上图使用 block 图表示的，也可以用 verilog 编写出来，代码如下：

```
//边沿检测电路
module edge_cap
(
    input clk, rst_n,
    input pulse,

    output pos_edge,
    output neg_edge
);
reg pulse_r1, pulse_r2;

always @ (posedge clk or negedge rst_n)
if(!rst_n)
begin
    pulse_r1 <= 1'b0;
```

```

    pulse_r2 <= 1'b0;
end
else
    begin
        pulse_r1 <= pulse;
        pulse_r2 <= pulse_r1;
    end

    assign pos_edge = (pulse_r1 && ~pulse_r2) ? 1:0;
    assign neg_edge = (~pulse_r1 && pulse_r2) ? 1:0;

endmodule

```

当检测到上升沿时，pos\_edge 信号输出一个时钟周期的高电平；检测到下降沿时，neg\_edge 输出一个时钟周期的高电平。

乍一看，这个电路似乎很简单地实现了边沿检测的功能，但是仔细分析就可以发现这种方法存在一个潜在的风险：当待测信号 pulse 是一个异步信号时，输出可能是亚稳态，如果 pulse 信号的变化刚好发生在 clk 时钟的建立时间和保持时间之内，那么第一级寄存器的输出 pulse\_r1 就会进入亚稳态，而 pulse\_r1 的亚稳态会立即传递给 pos\_edge 和 neg\_edge 信号，从而使得整个电路的输出进入亚稳态，进而影响下一级电路的正常工作，甚至导致整个系统崩溃！

因此，在进行异步信号边沿提取时，不能直接使用上面的这种电路，而应该先将异步信号同步化，一般采用多加一级寄存器的方法来减小亚稳态的发生概率，如下图所示：

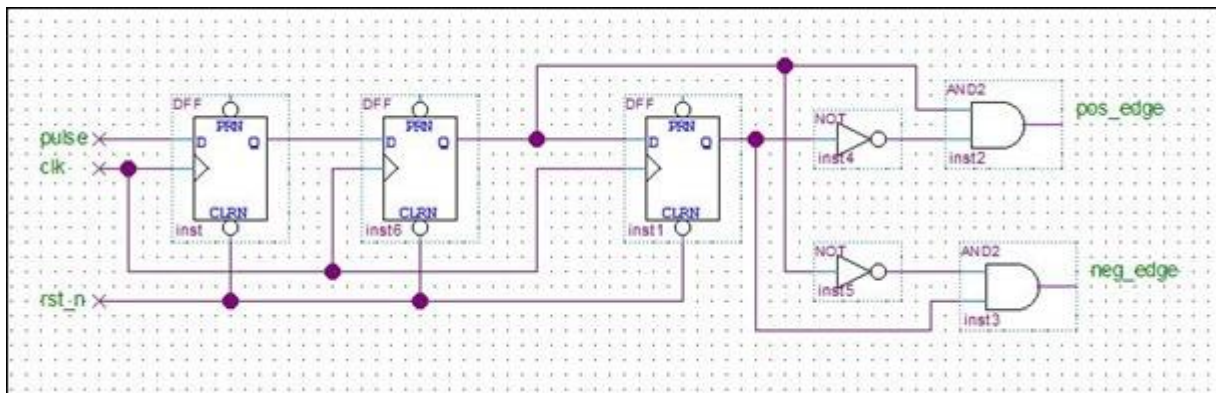


图2 异步信号边沿检测

也可以用 verilog 编写出来，代码如下：

//异步信号边沿检测电路，三级寄存器实现

```
module edge_cap
(
    input clk, rst_n,
    input pulse,

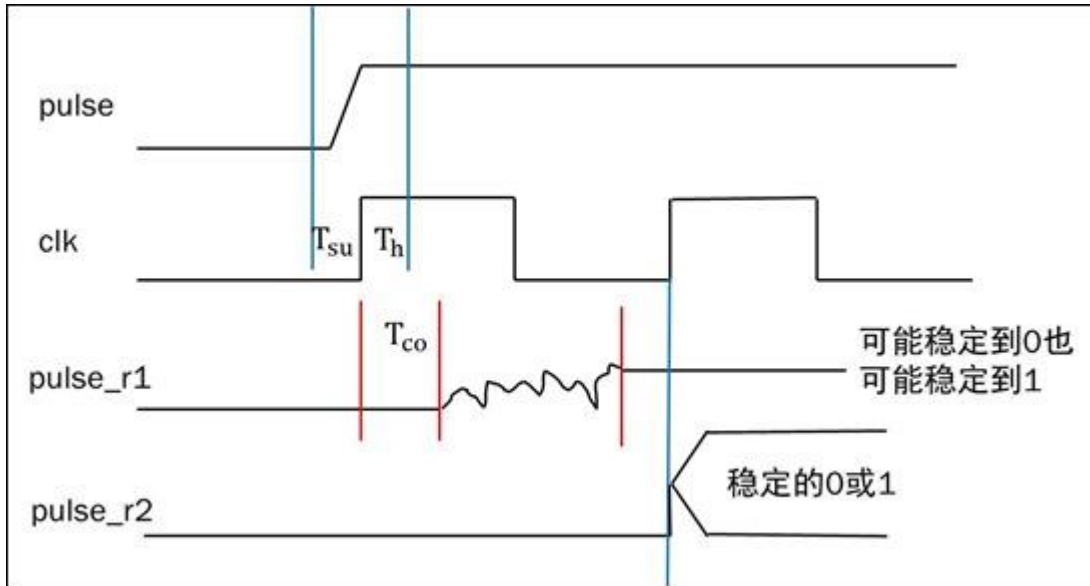
    output pos_edge,
    output neg_edge
);
reg pulse_r1, pulse_r2, pulse_r3;

always @ (posedge clk or negedge rst_n)
if(!rst_n)
    begin
        pulse_r1 <= 1'b0;
        pulse_r2 <= 1'b0;
        pulse_r3 <= 1'b0;
    end
else
    begin
        pulse_r1 <= pulse;
        pulse_r2 <= pulse_r1;
        pulse_r3 <= pulse_r2;
    end

assign pos_edge = (pulse_r2 && ~pulse_r3) ?1:0;
assign neg_edge = (~pulse_r2 && pulse_r3) ?1:0;

endmodule
```

经过这样的同步处理后，可以大大减小电路进入亚稳态的概率，如果第一级寄存器进入了亚稳态，一般也会在一个 clk 周期内稳定下来（可能稳定为 0 也可能稳定为 1），如下图所示：



图中 pulse 信号的改变刚好发生在 clk 的建立时间和保持时间之内，因而第一级寄存器的输出 pulse\_r1 可能会进入亚稳态，图中  $T_{co}$  为第一级寄存器 pulse\_r1 的状态建立时间（即 clock to output），一般情况下，亚稳态的决断时间（即从进入亚稳态到稳定下来的时间）不会超过一个时钟周期，因此在下一个 clk 上升沿到来之前，pulse\_r1 已经稳定下来（可能稳定到 0 也可能稳定到 1），这样第二级寄存器就会采集到一个稳定的状态，从而把亚稳态限制在第二级寄存器之前，保证了整个电路输出的稳定性。

综上所述，在异步信号边沿检测电路中，至少需要采用三级寄存器来实现，在对系统稳定性要求较高的数字系统中，可以采用更多级的寄存器来减小亚稳态发生概率，提高系统稳定性

## 2-22. 一个 8bit 宽的 AFIFO，输入时钟为 100MHz，输出时钟为 95MHz，设一个 package 为 4Kbit，且两个 package 之间的发送间距足够大。求 AFIFO 的最小深度？

一个异步 FIFO，读写频率不同，读写位宽相同。发送发一次 Burst 突发的数据量为 4Kbit，即 512B，在两次 Burst 突发之间有足够的时间，因此我们只用考虑在发送方 Burst 发送数据的时间  $T$  内，如果接受方没法将数据全部接受，其余数据均可存在 FIFO 内且不溢出，那么在发送方停止 Burst 发送数据的时间段内，接收方就

可以从容的从 FIFO 内读取数据。首先发送方 Burst 发送数据的时间段为  $T = 512/100\text{MHz}$ ，发送的数据量为  $B_{\text{send}} = 512\text{B}$ ，而在  $T$  这段时间内，接收方能够接受的数据量为  $B_{\text{rec}} = T \times 95\text{MHz} = 512 \times 95 / 100 \text{ B} = 486\text{B}$ ，因此  $B_{\text{remain}} = B_{\text{send}} - B_{\text{rec}} = 512 - 486 = 26$ 。那么 FIFO 的深度至少要大于等于 26 才行。

FIFO 用于缓冲块数据流，一般用在写快读慢时，

FIFO 深度 / (写入速率 - 读出速率) = FIFO 被填满时间      应大于 数据包传送时间 = 数据量 / 写入速率

例：A/D 采样率 50MHz, dsp 读 A/D 读的速率 40MHz, 要不丢失地将 10 万个采样数据送入 DSP, 在 A/D 在和 DSP 之间至少加多大容量（深度）的 FIFO 才行？

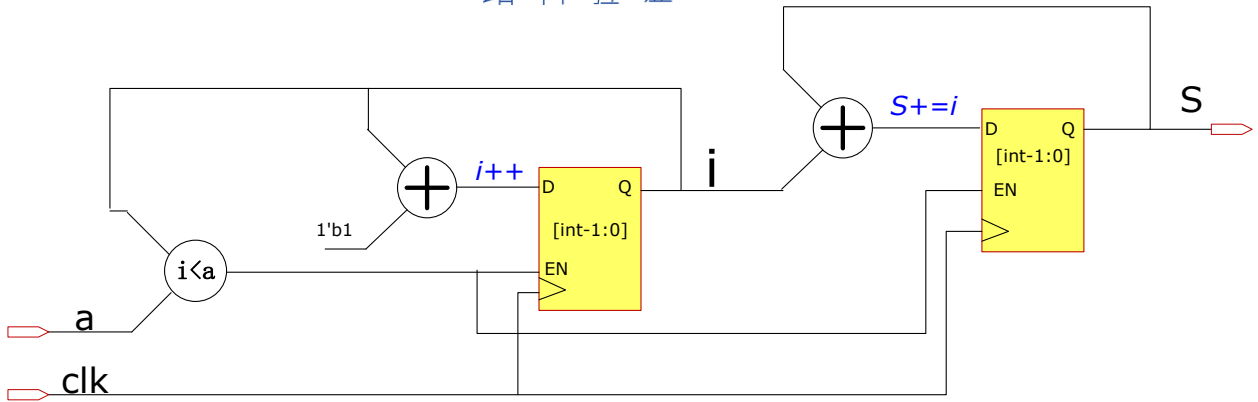
$100,000 / 50\text{MHz} = 1 / 500 \text{ s} = 2\text{ms}$

$(50\text{MHz} - 40\text{MHz}) \times 1/500 = 20\text{k}$  既是 FIFO 深度。

2-23. 阅读以下 C code (已知  $a > 0$ ) :

```
int acc(int a)
{
    int i, s=0;
    for i=0; i++; i<a
    {
        S+=i;
    }
    return s;
}
```

请用 verilog 实现以上功能，并画出电路图，要求至少使用 1 个寄存器。



## 2-24. 简述数字 IC 设计前端到后端流程和列举每步使用的 EDA 工具

逻辑设计—子功能分解—详细时序框图—分块逻辑仿真—电路设计 (RTL 级描述)—功能仿真—综合 (加时序约束和设计库)—电路网表—网表仿真—预布局布线 (SDF 文件)—网表仿真 (带延时文件)—静态时序分析—布局布线—参数提取—SDF 文件—后仿真—静态时序分析—测试向量生成—工艺设计与生产—芯片测试—芯片应用，在验证过程中出现的时序收敛，功耗，面积问题，应返回前端的代码输入进行重新修改，再仿真，再综合，再验证，一般都要反复好几次才能最后送去 foundry 厂流片。设计公司是 fabless

### 数字 IC 设计流程 (zz)

1. 需求分析 (制定规格书)。分析用户或市场的需求，并将其翻译成对芯片产品的技术要求
2. 算法设计。设计和优化芯片中所使用的算法。这一阶段一般使用高级编程语言 (如 C/C++)，利用算法级建模和仿真工具 (如 MATLAB, SPW) 进行浮点和定点的仿真，进而对算法进行评估和优化。
3. 构架设计。根据设计的功能需求和算法分析的结果，设计芯片的构架，并对不同的方案进行比较，选择性能价格最优的方案。这一阶段可以使用 SystemC 语言对芯片构架进行模拟和分析。
4. RTL 设计 (代码输入)。使用 HDL 语言完成对设计实体的 RTL 级描述。这一阶段使用 VHDL 和 Verilog HDL 语言的输入工具编写代码。
5. RTL 验证 (功能仿真)。使用仿真工具或其他 RTL 代码分析工具，验证 RTL 代码的质量和性能。
6. 综合。从 RTL 代码生成描述实际电路的门级网表文件。
7. 门级验证 (综合后仿真)。对综合产生的门级网表进行验证。这一阶段通常会使用仿



真、静态时序分析和形式验证等工具。

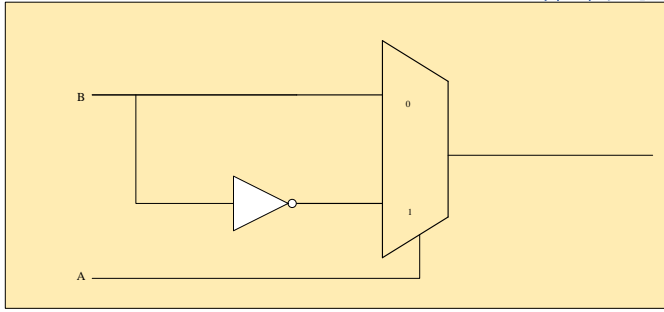
8. 布局布线。 后端设计对综合产生的门级网表进行布局规划 (Floorplanning)、布局 (Placement)、布线 (Routing)，生成生产用的版图。
9. 电路参数提取确定芯片中互连线的寄生参数，从而获得门级的延时信息。
10. 版图后验证。 根据后端设计后取得的新的延时信息，再次验证设计是否能够实现所有的功能和性能指标。
11. 芯片生产。 生产在特定的芯片工艺线上制造出芯片。
12. 芯片测试。 对制造好的芯片进行测试，检测生产中产生的缺陷和问题

### 数字 IC 后端设计流程

1. 数据准备。 对于 Cadance 的 SE 而言后端设计所需的数据主要有是 Foundry 厂提供的标准单元、宏单元和 I/O Pad 的库文件,它包括物理库、时序库及网表库,分别以 .lef、.tlf 和 .v 的形式给出。前端的芯片设计经过综合后生成的门级网表,具有时序约束和时钟定义的脚本文件和由此产生的 .gcf 约束文件以及定义电源 Pad 的 DEF (Design Exchange Format) 文件。(对 synopsys 的 Astro 而言,经过综合后生成的门级网表,时序约束文件 SDC 是一样的,Pad 的定义文件--tdf, .tf 文件 --technology file, Foundry 厂提供的标准单元、宏单元和 I/O Pad 的库文件 就与 FRAM, CELL view, LM view 形式给出(Milkway 参考库 and DB, LIB file)
2. 布局规划。 主要是标准单元、 I/O Pad 和宏单元的布局。 I/O Pad 预先给出了位置,而宏单元则根据时序要求进行摆放,标准单元则是给出了一定的区域由工具自动摆放。布局规划后,芯片的大小,Core 的面积,Row 的形式、电源及地线的 Ring 和 Strip 都确定下来了。如果必要在自动放置标准单元和宏单元之后,你可以先做一次 PNA(power network analysis) --IR drop and EM .
3. Placement -自动放置标准单元。 布局规划后,宏单元、 I/O Pad 的位置和放置标准单元的区域都已确定,这些信息 SE (Silicon Ensemble) 会通过 DEF 文件传递给 PC(Physical Compiler),PC 根据由综合给出的 .DB 文件获得网表和时序约束信息进行自动放置标准单元,同时进行时序检查和单元放置优化。如果你用的是 PC +Astro 那你可用 write\_milkway, read\_milkway 传递数据。
4. 时钟树生成(CTS Clock tree synthesis)。 芯片中的时钟网络要驱动电路中所有的时序单元,所以时钟源端门单元带载很多,其负载延时很大并且不平衡,需要插入缓冲器减小负载和平衡延时。时钟网络及其上的缓冲器构成了时钟树。一般要反复几次才可以做出一个比较理想的时钟树。

5. STA 静态时序分析和后仿真。时钟树插入后,每个单元的位置都确定下来了,工具可以提出 Global Route 形式的连线寄生参数,此时对延时参数的提取就比较准确了。SE 把.V 和.SDF 文件传递给 PrimeTime 做静态时序分析。确认没有时序违规后,将这来两个文件传递给前端人员做后仿真。对 Astro 而言,在 detail routing 之后,用 starRC XT 参数提取,生成的 E.V 和.SDF 文件传递给 PrimeTime 做静态时序分析,那将会更准确。
6. ECO(Engineering Change Order)。针对静态时序分析和后仿真中出现的问题,对电路和单元布局进行小范围的改动。
7. filler 的插入(pad fliier, cell filler)。Filler 指的是标准单元库和 I/O Pad 库中定义的与逻辑无关的填充物,用来填充标准单元和标准单元之间,I/O Pad 和 I/O Pad 之间的间隙,它主要是把扩散层连接起来,满足 DRC 规则和设计需要。
8. 布线(Routing)。Global route-- Track assign --Detail routing--Routing optimization 布线是指在满足工艺规则和布线层数限制、线宽、线间距限制和各线网可靠绝缘的电性能约束的条件下,根据电路的连接关系将各单元和 I/O Pad 用互连线连接起来,这些是在时序驱动(Timing driven )的条件下进行的,保证关键时序路径上的连线长度能够最小。 --Timing report clear
9. Dummy Metal 的增加。Foundry 厂都有对金属密度的规定,使其金属密度不要低于一定的值,以防在芯片制造过程中的刻蚀阶段对连线的金属层过度刻蚀从而降低电路的性能。加入 DummyMetal 是为了增加金属的密度。
10. DRC 和 LVS。DRC 是对芯片版图中的各层物理图形进行设计规则检查 (spacing ,width),它也包括天线效应的检查,以确保芯片正常流片。LVS 主要是将版图和电路网表进行比较,来保证流片出来的版图电路和实际需要的电路一致。DRC 和 LVS 的检查--EDA 工具 Synopsys hercules/mentor calibre/ CDN Dracula 进行的.Astro also include LVS/DRC check commands.
11. Tape out。在所有检查和验证都正确无误的情况下把最后的版图 GDS II 文件传递给 Foundry 厂进行掩膜制造

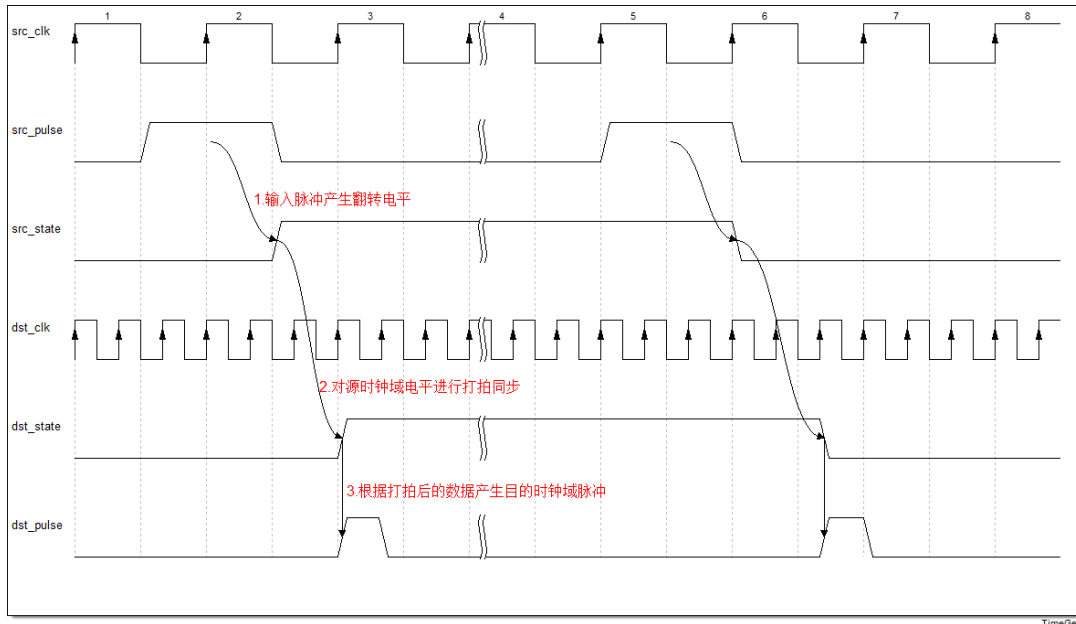
2-25. 请用一个二选一选择器和一个非门搭一个异或门。



2-26. Src\_pulse 是 src\_clk(100Mhz) 时钟域的一个单时钟脉冲信号, 如何将其同步到时钟域 dst\_clk (300MHz) 中, 并产生出 dst\_pulse 同步脉冲信号, 请用 verilog 代码描述, 并画出对应的时序波形说明图。

基于以上应用, 设计一个简单的脉冲同步器如下图所示:

- (1) 将 src\_clk 时钟域的输入脉冲转换为 src\_clk 时钟域的电平信号 src\_state;
- (2) 对 src\_data 电平信号进行打拍 (一般可打 2 拍) 同步到 dst\_clk 时钟域;
- (3) 对 dst\_clk 时钟域的电平信号进行边沿检测, 产生 dst\_clk 时钟域的脉冲;



```
module PULSE_SYNC
(
    src_clk        , //source clock
    src_rst_n      , //source clock reset (0: reset)
    src_pulse      , //source clock pulse in
```

```
        dst_clk      , //destination clock
        dst_rst_n    , //destination clock reset (0:reset)
        dst_pulse     //destination pulse out
    );
//INPUT DECLARATION
    input          src_clk      ; //source clock
    input          src_rst_n    ; //source clock reset (0: reset)
    input          src_pulse    ; //source clock pulse in

    input          dst_clk      ; //destination clock
    input          dst_rst_n    ; //destination clock reset
    (0:reset)

//OUTPUT DECLARATION
    output          dst_pulse    ; //destination pulse out

//INTER DECLARATION
    reg             src_state    ;
    reg             state_dly1   ;
    reg             state_dly2   ;
    reg             dst_state    ;
    wire            dst_pulse    ;

//-----MODULE SOURCE CODE-----

always @(posedge src_clk or negedge src_rst_n)
begin
    if(src_rst_n == 1'b0)
        src_state    <= 1'b0 ;
    else if (src_pulse)
        src_state    <= ~src_state ;
end

always @(posedge dst_clk or negedge dst_rst_n)
begin
```

```
if(dst_rst_n == 1'b0)
    begin
        state_dly1    <= 1'b0 ;
        state_dly2    <= 1'b0 ;
        dst_state      <= 1'b0 ;
    end
else
    begin
        state_dly1    <= src_state ;
        state_dly2    <= state_dly1;
        dst_state      <= state_dly2;
    end
end

assign dst_pulse = dst_state ^ state_dly2 ;

endmodule
```

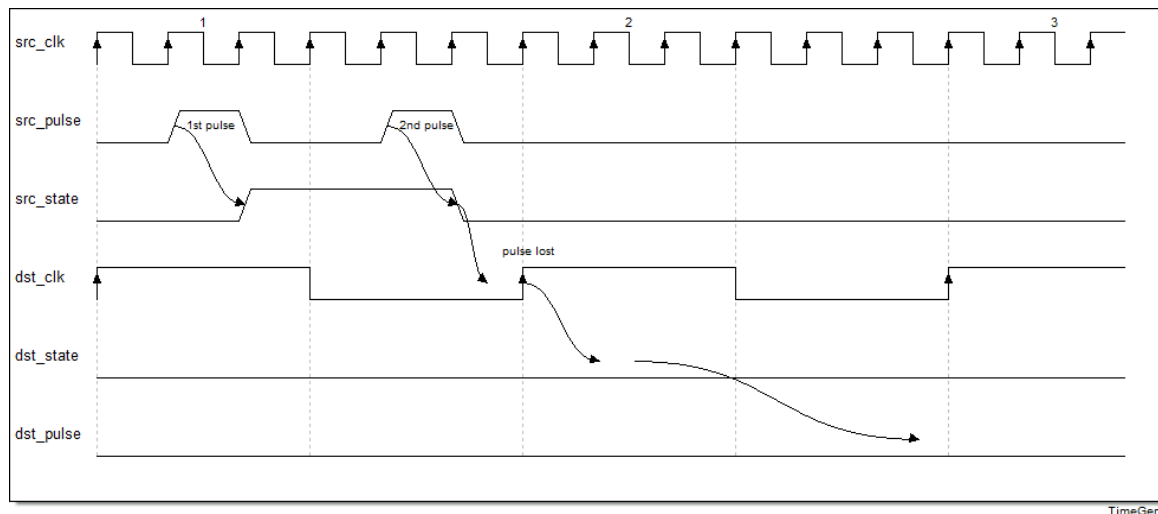
## 2-27. Please synchronize :src\_clk domain pulse “src\_pulse” to “dst\_clk” domain ,src\_clk is faster than dst\_clk(please write Verilog code )

回顾上一题中的同步器的基本设计原理：

- (1) 源时钟域脉冲转换为源时钟域电平信号；
- (2) 对单 bit 电平信号进行打拍的异步处理；
- (3) 在目的时钟域中进行脉冲还原。

从以上设计原理中，我们可以发现该同步器的控制传递是单向的，即仅从源时钟域到目的时钟域，目的时钟域并没有状态反馈。假设存在如下应用：

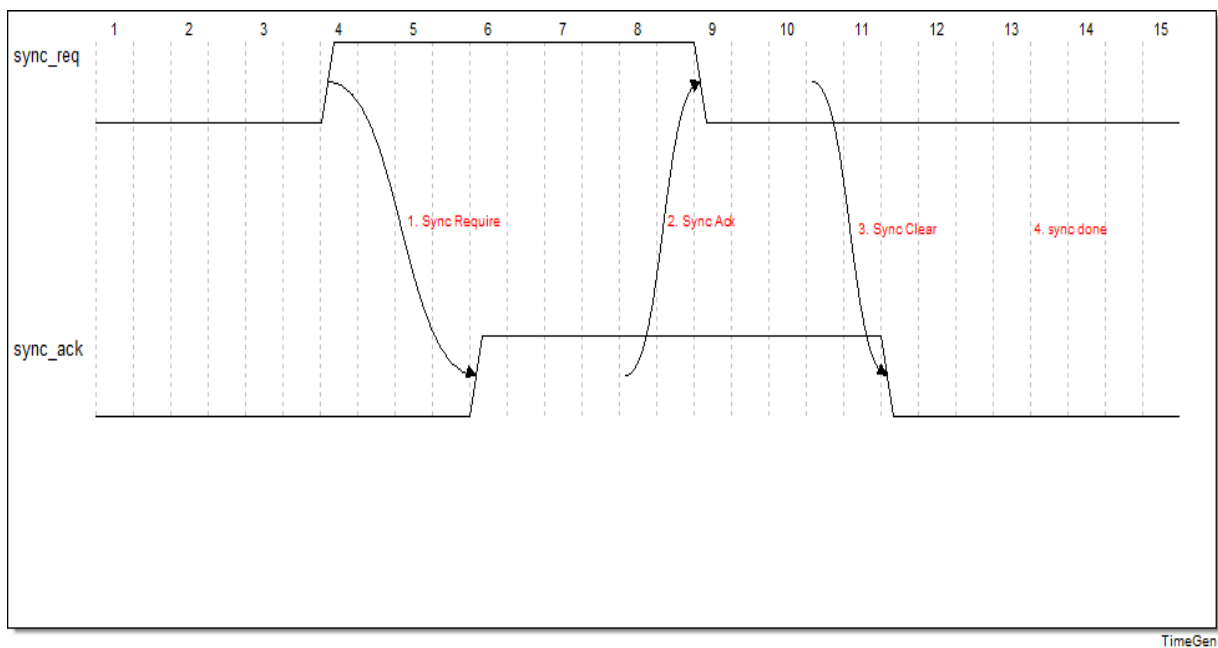
- (1) 源时钟域中的第一个脉冲和第二个脉冲间隔过短，第一个脉冲未完成同步，第二个脉冲又将状态清空，导致最终脉冲同步丢失。



要解决以上同步问题，需要引入异步握手机制，保证每个脉冲都同步成功，同步成功后再进行下一个脉冲同步。握手原理如下：

**sync\_req**: 源时钟域同步请求信号，高电平表示当前脉冲需要同步；

**sync\_ack**: 目的时钟域应答信号，高电平表示当前已收到同步请求；



完整同步过程分为以下 4 个步骤：

(1) 同步请求产生；当同步器处于空闲（即上一次已同步完成）时，源同步脉冲到达时产生同步请求信号 **sync\_req**；

(2) 同步请求信号 `sync_req` 同步到目的时钟域，目的时钟域产生脉冲信号并将产生应答信号 `sync_ack`;

(3) 同步应答信号 `sync_ack` 同步到源时钟域，源时钟域检测到同步应答信号 `sync_ack` 后，清除同步请求信号;

(4) 目的时钟域检测到 `sync_req` 撤销后，清除 `sync_ack` 应答; 源时钟域将到 `sync_ack` 清除后，认为一次同步完成，可以同步下一个脉冲。

```
module HANDSHAKE_PULSE_SYNC
(
    src_clk          , //source clock
    src_rst_n        , //source clock reset (0: reset)
    src_pulse        , //source clock pulse in
    src_sync_fail    , //source clock sync state: 1 clock pulse if sync fail.
    dst_clk          , //destination clock
    dst_rst_n        , //destination clock reset (0:reset)
    dst_pulse        //destination pulse out
);

//PARA  DECLARATION

//INPUT  DECLARATION
input          src_clk      ; //source clock
input          src_rst_n    ; //source clock reset (0: reset)
input          src_pulse    ; //source clock pulse in

input          dst_clk      ; //destination clock
input          dst_rst_n    ; //destination clock reset (0:reset)

//OUTPUT DECLARATION
output         src_sync_fail ; //source clock sync state: 1 clock pulse if
sync fail.
output         dst_pulse     ; //destination pulse out

//INTER DECLARATION
```

```

wire            dst_pulse      ;
wire            src_sync_idle  ;
reg             src_sync_fail  ;
reg             src_sync_req   ;
reg             src_sync_ack   ;
reg             ack_state_dly1 ;
reg             ack_state_dly2 ;
reg             req_state_dly1 ;
reg             req_state_dly2 ;
reg             dst_req_state  ;
reg             dst_sync_ack   ;

//-----MODULE SOURCE CODE-----

//-----
// DST Clock :
// 1. generate src_sync_fail;
// 2. generate sync req
// 3. sync dst_sync_ack
//-----

assign src_sync_idle = ~(src_sync_req | src_sync_ack );

//report an error if src_pulse when sync busy ;
always @(posedge src_clk or negedge src_rst_n)
begin
    if(src_rst_n == 1'b0)
        src_sync_fail  <= 1'b0 ;
    else if (src_pulse & (~src_sync_idle))
        src_sync_fail  <= 1'b1 ;
    else
        src_sync_fail  <= 1'b0 ;
end

//set sync req if src_pulse when sync idle ;

```



```

always @(posedge src_clk or negedge src_rst_n)
begin
    if(src_rst_n == 1'b0)
        src_sync_req    <= 1'b0 ;
    else if (src_pulse & src_sync_idle)
        src_sync_req    <= 1'b1 ;
    else if (src_sync_ack)
        src_sync_req    <= 1'b0 ;
end

always @(posedge src_clk or negedge src_rst_n)
begin
    if(src_rst_n == 1'b0)
        begin
            ack_state_dly1 <= 1'b0 ;
            ack_state_dly2 <= 1'b0 ;
            src_sync_ack   <= 1'b0 ;
        end
    else
        begin
            ack_state_dly1 <= dst_sync_ack    ;
            ack_state_dly2 <= ack_state_dly1  ;
            src_sync_ack   <= ack_state_dly2  ;
        end
end

//-----
// DST Clock :
// 1. sync src sync req
// 2. generate dst pulse
// 3. generate sync ack
//-----

always @(posedge dst_clk or negedge dst_rst_n)
begin
    if(dst_rst_n == 1'b0)

```

```

begin
    req_state_dly1 <= 1'b0 ;
    req_state_dly2 <= 1'b0 ;
    dst_req_state  <= 1'b0 ;

end
else
begin
    req_state_dly1 <= src_sync_req      ;
    req_state_dly2 <= req_state_dly1    ;
    dst_req_state  <= req_state_dly2    ;

end
end

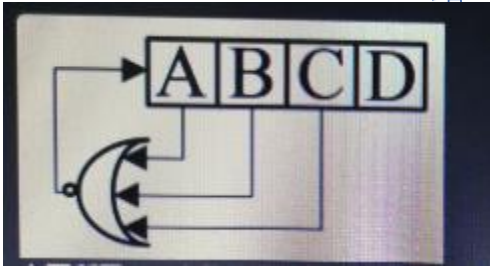
//Rising Edge of dst_state generate a dst_pulse;
assign dst_pulse = (~dst_req_state) & req_state_dly2 ;

//set sync ack when src_req = 1 , clear it when src_req = 0 ;
always @(posedge dst_clk or negedge dst_rst_n)
begin
    if(dst_rst_n == 1'b0)
        dst_sync_ack    <= 1'b0;
    else if (req_state_dly2)
        dst_sync_ack    <= 1'b1;
    else
        dst_sync_ack    <= 1'b0;
end

endmodule

```

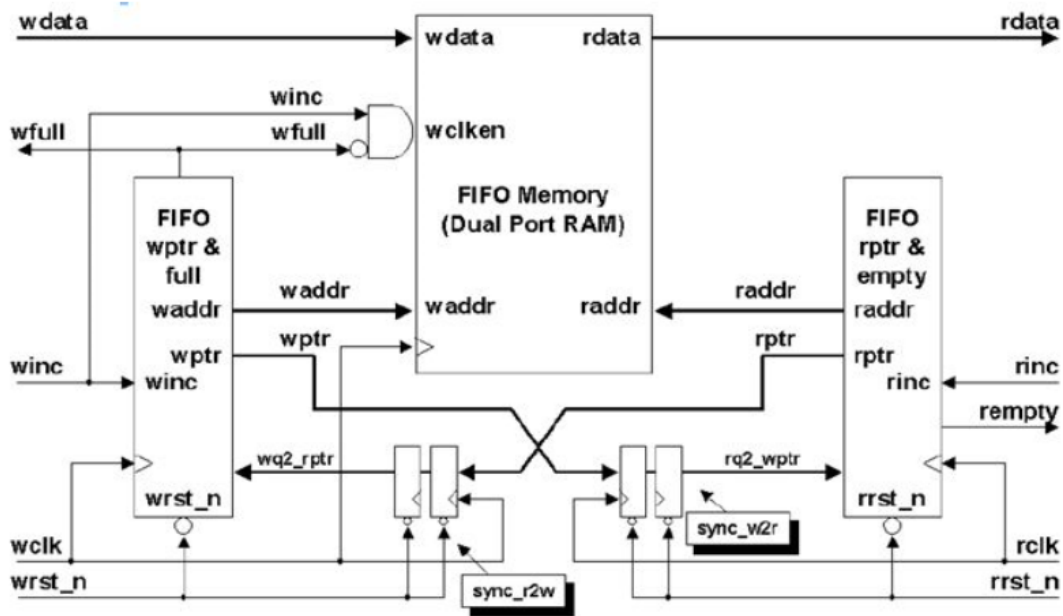
2-28. 下图所示 4bit 右移位寄存器，0 时刻 ABCD 初始状态为 0111，请写出 5 个时刻之后的 ABCD 输出：B



- A. 1010      B. 0100      C. 1101      D. 1110

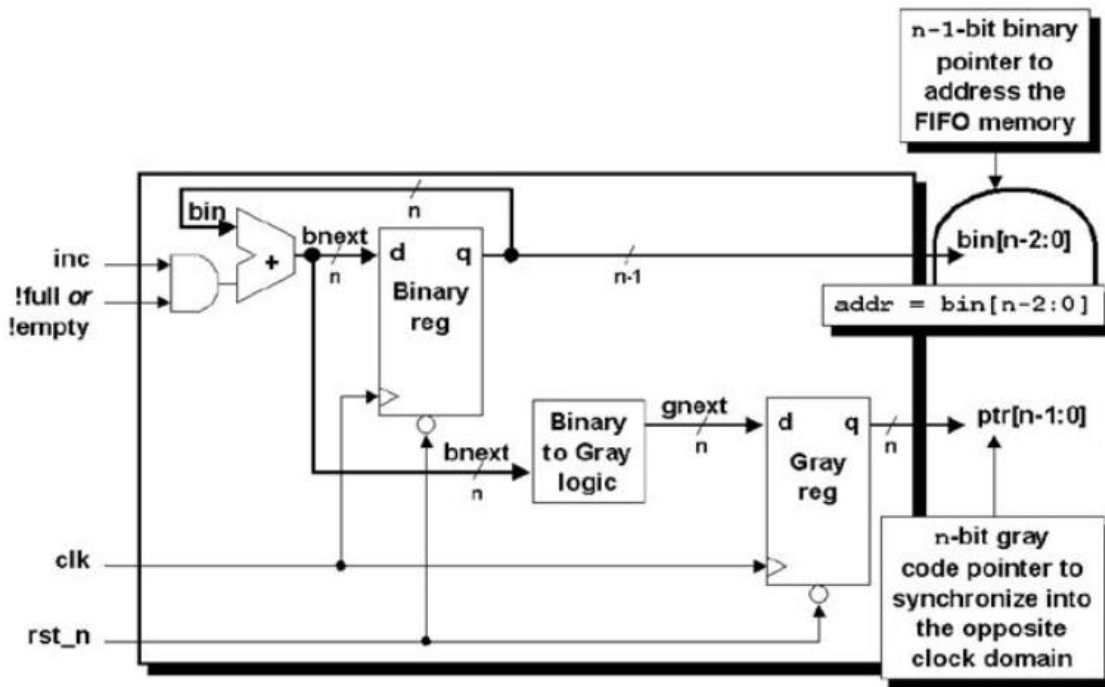
2-29. Please draw a block diagram of a typical asynchronous FIFO, and describe the interface(e.g. clock A domain write data to clock B domain through asynchronous FIFO)

系统的总体框图如下:



将一个二进制的计数值从一个时钟域同步到另一个时钟域的时候很容易出现问题，因为采用二进制计数器时所有位都可能同时变化，在同一个时钟沿同步多个信号的变化会产生亚稳态问题。而使用格雷码只有一位变化，因此在两个时钟域间同步多个位不会产生问题。所以需要有一个二进制到 gray 码的转换电路，将地址值转换为

相应的 gray 码，然后将该 gray 码同步到另一个时钟域进行对比，作为空满状态的检测。



使用 gray 码进行对比，如何判断“空”与“满”

使用 gray 码解决了一个问题，但同时也带来另一个问题，即在格雷码域如何判断空与满。

对于“空”的判断依然依据二者完全相等(包括 MSB)；

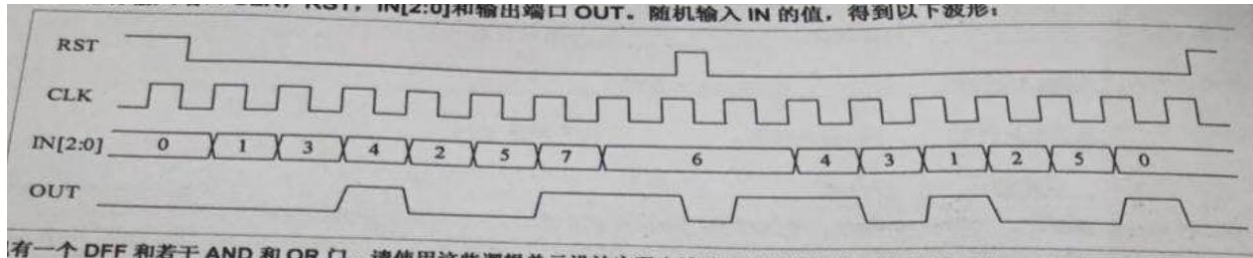
而对于“满”的判断，如下图，由于 gray 码除了 MSB 外，具有镜像对称的特点，当读指针指向 7，写指针指向 8 时，除了 MSB，其余位皆相同，不能说它为满。因此不能单纯的只检测最高位了，在 gray 码上判断为满必须同时满足以下 3 条：

wptr 和同步过来的 rptr 的 MSB 不相等，因为 wptr 必须比 rptr 多折回一次。

wptr 与 rptr 的次高位不相等，如上图位置 7 和位置 15，转化为二进制对应的是 0111 和 1111，MSB 不同说明多折回一次，111 相同代表同一位置。

剩下的其余位完全相等。

2-30. 某电路有输入端口 CLK, RST, IN[2:0]和输出端口 OUT。随机输入 IN 的值，得到以下波形：



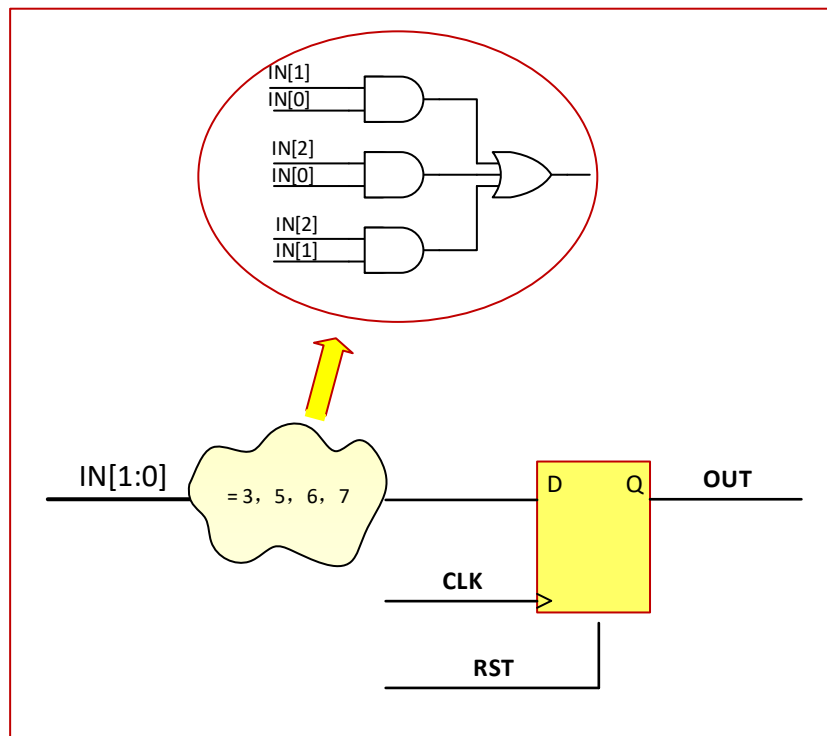
现有一个 DFF 和若干 AND 和 OR 门，请使用这些逻辑单元设计实现上述功能的最简电路，并画出电路图。

经过波形的观察可以看出，OUT 是寄存器的输出。它是由 IN 在 [3, 5, 7, 6] 取值范围下解码输出电平 ‘1’。

3, 5, 7, 6 译码最小项之和表达式经化简得到：

$Y \leq BC + AC + AB$  (Y:OUT, A:IN[2]; B:IN[1]; C:IN[0]).

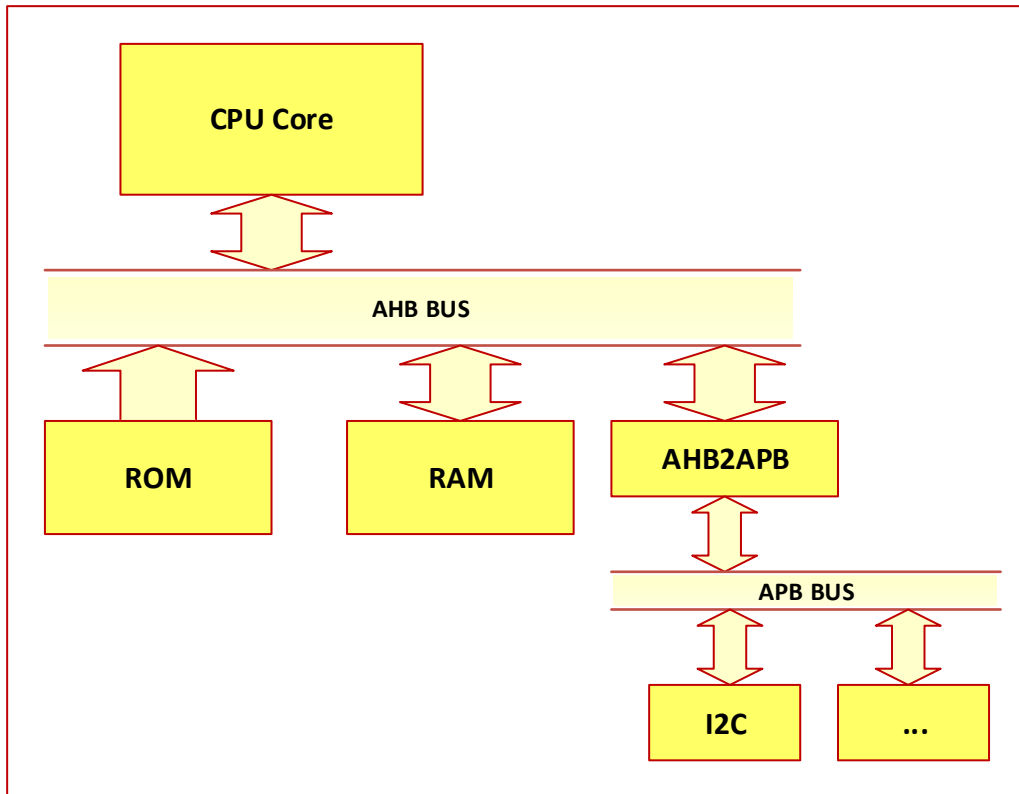
A \ BC	00	01	11	10
0			1	
1		1	1	1



2-31. 有下图电路请使用它构造一个微型 cpu 系统

Cpu core    ROM    RAM    I2C

其中 cpu core 通过 ahb 总线访问其他电路，ROM 为带有 ahb 总线接口的程序存储器，RAM 为带有 ahb 总线接口的数据存储器，I2C 为 APB 总线下的串行接口外设。



2-32. 设计一个自动饮料售卖机，饮料 10 分钱，硬币有 5 分和 10 分两种，并考虑找零，

1. 画出 fsm（有限状态机）
2. 用 verilog 编程，语法要符合 FPGA 设计的要求
3. 设计工程中可使用的工具及设计大致过程？

设计过程

- 1、首先确定输入输出， A=1 表示投入 10 分， B=1 表示投入 5 分， Y=1 表示弹出饮料， Z=1 表示找零。

2、确定电路的状态， S0 表示没有进行投币， S1 表示已经有 5 分硬币。

3、画出状态转移图。

```
module sell(clk,rst,a,b,y,z);
input  clk,rst,a,b;
output y,z;
parameter s0=0,s1=1;
reg state,next_state;
always@(posedge clk)
begin
if(!rst)
state<=s0;
else
state<=next_state;
end
always@(a or b or cstate)
begin
y=0;z=0;
case(state)
s0: if(a==1&&b==0) next_state=s1;
else if(a==0&&b==1)
begin
next_state=s0; y=1;
end
else
next_state=s0;
s1: if(a==1&&b==0)
begin
next_state=s0;y=1;
end
else if(a==0&&b==1)
begin
next_state=s0; y=1;z=1;
end
else
next_state=s0;
```

```
default: next_state=s0;
endcase
end
endmodule
```

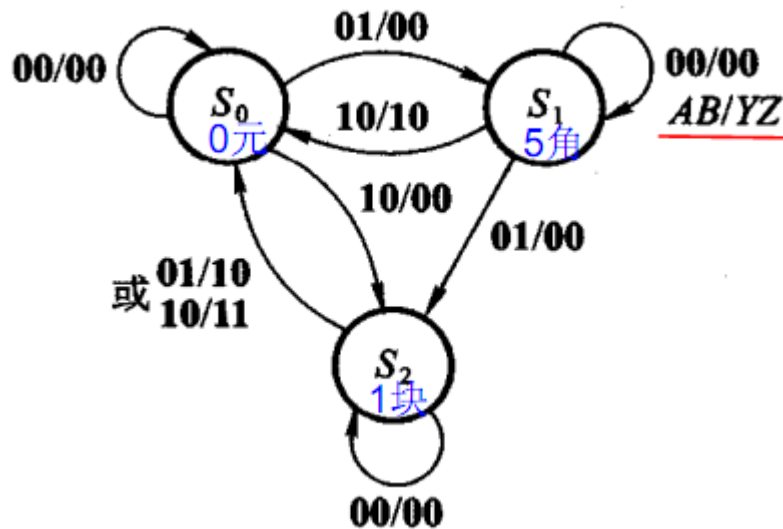
扩展：设计一个自动售饮料机的逻辑电路。它的投币口每次只能投入一枚五角或一元的硬

币。投入一元五角硬币后给出饮料；投入两元硬币时给出饮料并找回五角。

1、 确定输入输出，投入一元硬币 A=1，投入五角硬币 B=1，给出饮料 Y=1，找回五角 Z=1；

2、 确定电路的状态数，投币前初始状态为 S0，投入五角硬币为 S1，投入一元硬币为 S2。

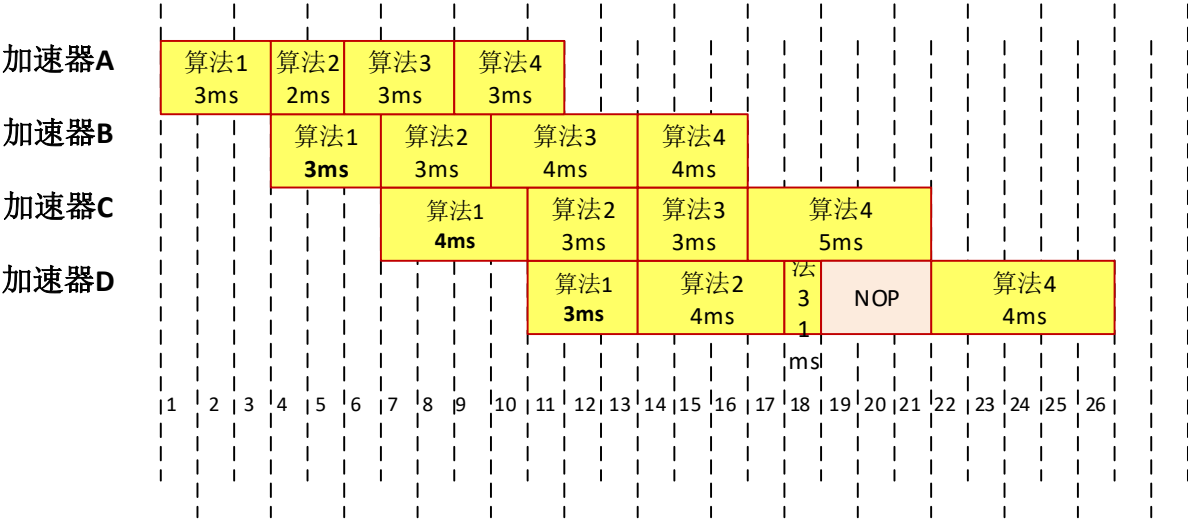
画出转该转移图，根据状态转移图可以写成 Verilog 代码。



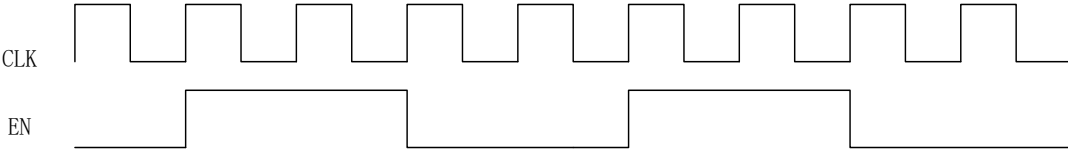
2-33. 在某安全加密芯片中有 A, B, C, D 四个硬件加速器，分辨实现模加，模乘，模除，模逆四种功能。加速器一经启动，不能停止，将连续运行直至操作完毕。现需要设计一个调度器，合理安排这四个硬件加速器协同合作实现四个加密算法。每个算法都要求按照 A→B→C→D 的调用顺序执行运算功能。每个算法中四种运算所需的硬件加速器运行时间（单位 ms）如下表，现要求调度按照算法 1→2→3→4 的顺序启动相应的加速器 A，允许利用流水线技术充分利用加速器资源。请问执行完所有的算法最快需要多长时间？



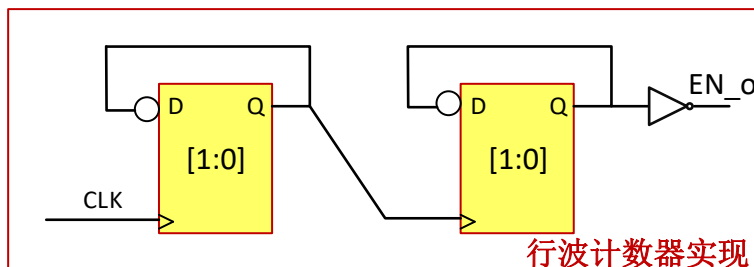
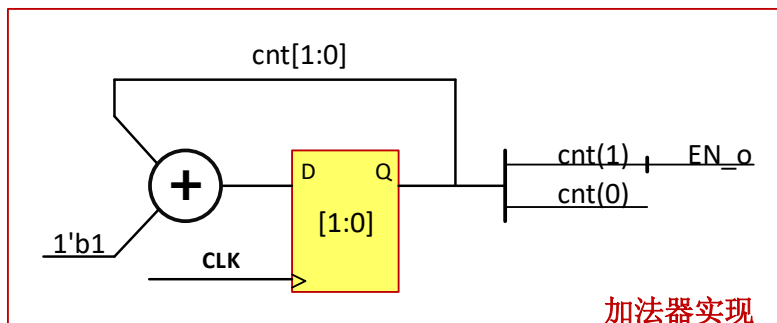
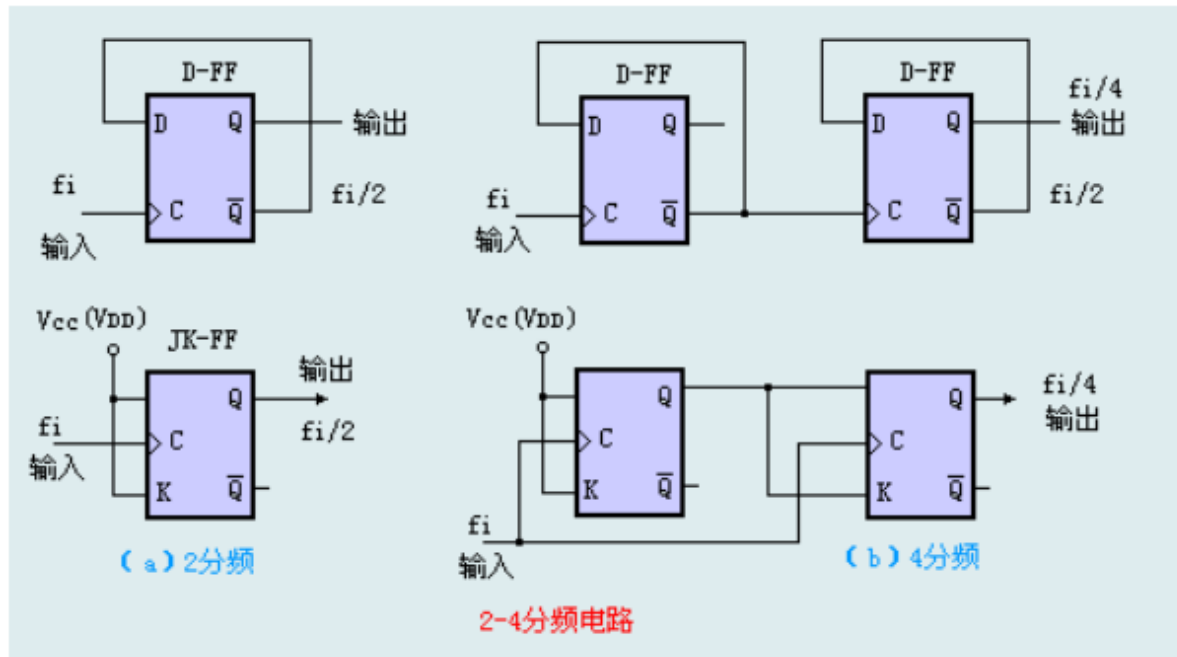
	算法 1	算法 2	算法 3	算法 4
加速器 A	3	2	3	3
加速器 B	3	3	4	4
加速器 C	4	3	3	5
加速器 D	3	4	1	4



2-34. 按照下面的时序图，用尽可能多的方法或者语言实现时序图中 en 的逻辑。



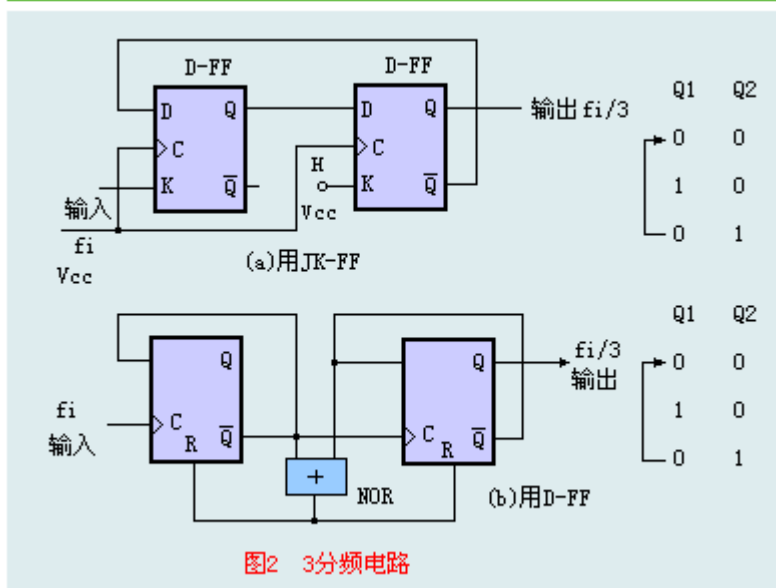
用于  $N=2-4$  分频比的电路，常用双 D-FF 或双 JK-FF 器件来构成，分频比  $n>4$  的电路，则常采用计数器(后续题目会讲解)（如可预置计数器）来实现更为方便，一般无需再用单个 FF 来组合。下图的分频电路输出占空比均为 50%，可用 D-FF，也可用 JK-FF 来组成，用 JK-FF 构成分频电路容易实现并行式同步工作，因而适合于较高频的应用场合。而 FF 中的引脚 R、S(P)等引脚如果不使用，则必须按其功能要求连接到非有效电平的电源或地线上。



## 2-35. 实现三分频电路， $3/2$ 分频电路等（偶数倍分频 奇数倍分频）

下图是 3 分频电路，用 JK-FF 实现 3 分频很方便，不需要附加任何逻辑电路就能实现同步计数分频。但用 D-FF 实现 3 分频时，必须附加译码反馈电路，如图 2 所示的译码复位电路，强制计数状态返回到初始全零状态，就是用 NOR 门电路把  $Q_2$ ， $Q_1 = "11B"$  的状态译码产生“H”电平复位脉冲，强迫 FF1 和 FF2 同时瞬间（在下一

时钟输入  $F_i$  的脉冲到来之前) 复零, 于是  $Q_2, Q_1 = "11B"$  状态仅瞬间作为“毛刺”存在而不影响分频的周期, 这种“毛刺”仅在  $Q_1$  中存在, 实用中可能会造成错误, 应当附加时钟同步电路或阻容低通滤波电路来滤除, 或者仅使用  $Q_2$  作为输出。  
D-FF 的 3 分频, 还可以用 AND 门对  $Q_2, Q_1$  译码来实现返回复零

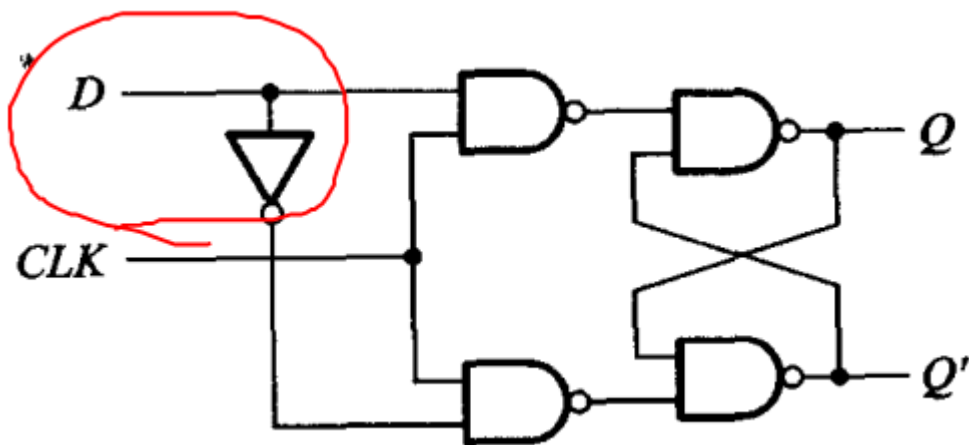


## 2-36. 用你熟悉的设计方式设计一个可预置初值的 7 进制循环计数器, 15 进制的呢?

```
module counter7(clk, rst, load, data, cout);
input clk, rst, load;
input [2:0] data;
output reg [2:0] cout;
always@(posedge clk)
begin
if(!rst)
cout<=3' d0;
else if(load)
cout<=data;
else if(cout>=3' d6)
cout<=3' d0;
else
cout<=cout+3' d1;
end
endmodule
```

电平敏感的存储器件称为锁存器。可分为高电平锁存器和低电平锁存器，用于不同时钟之间的信号同步。有交叉耦合的门构成的双稳态的存储原件称为触发器。分为上升沿触发和下降沿触发。可以认为是两个不同电平敏感的锁存器串连而成。前一个锁存器决定了触发器的建立时间，后一个锁存器则决定了保持时间。latch 是电平触发， register 是边沿触发， register 在同一时钟边沿触发下动作，符合同步电路的设计思想，而 latch 则属于异步电路设计，往往会导致时序分析困难，不适当的应用 latch 则会大量浪费芯片资源。

2-38. 用逻辑门画出 D 触发器。



电平触发的 D 触发器 (D 锁存器) 牢记!

2-39. 用传输门和倒向器搭一个边沿触发器 ( DFF )

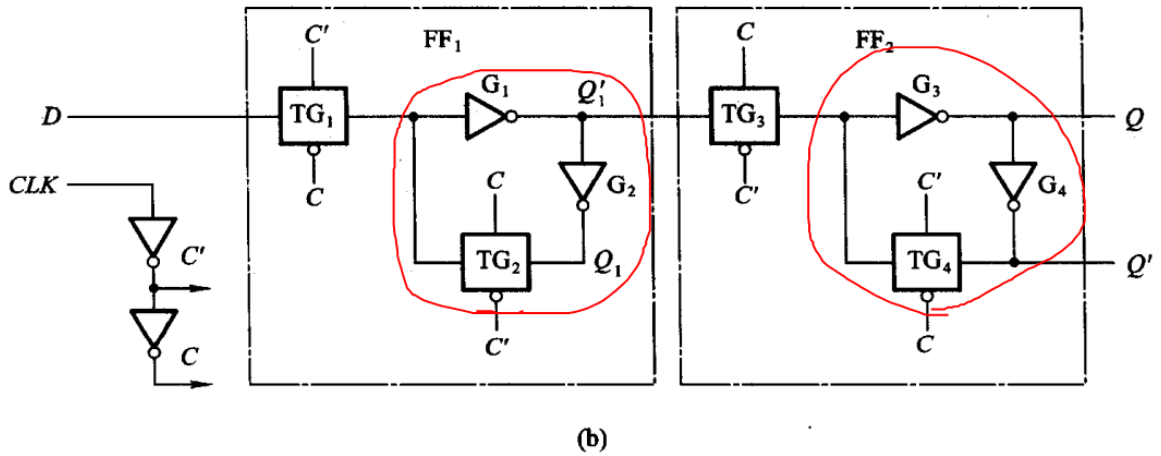
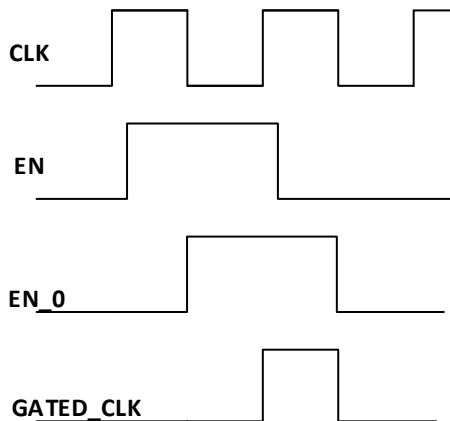
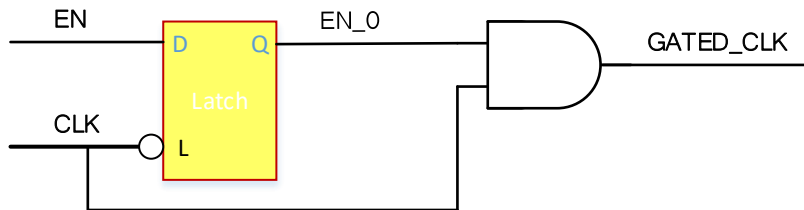


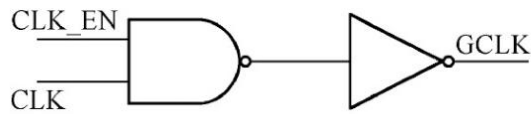
图 5.5.1 用两个电平触发 D 触发器组成的边沿触发器

2-40. 在低功耗设计中门控时钟的作用和工作原理是什么？

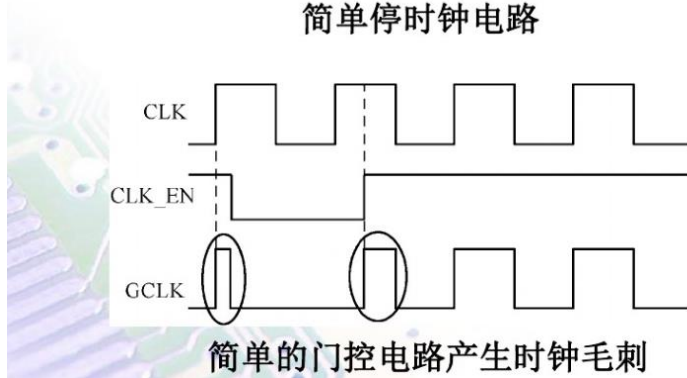
- 请画出门控时钟单元基本原理图(用逻辑门表示)，该门控单元应用于上升沿有效的系统时钟。
- 请画出该门控时钟的工作波形(门控前时钟 CLK，门控使能信号 GATE，门控后时钟 CLK\_G)。



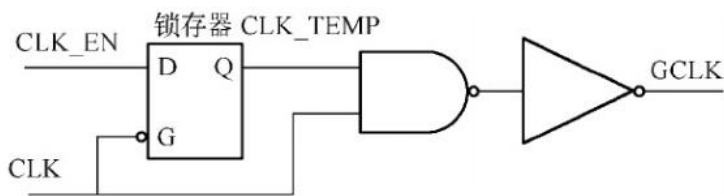
## 门控时钟即用逻辑门电路控制模块时钟的停或开



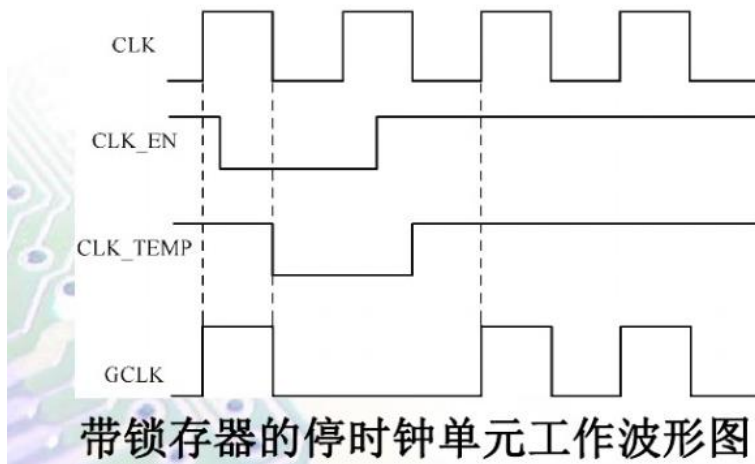
简单停时钟电路



简单的门控电路产生时钟毛刺



使用锁存器停时钟



带锁存器的停时钟单元工作波形图

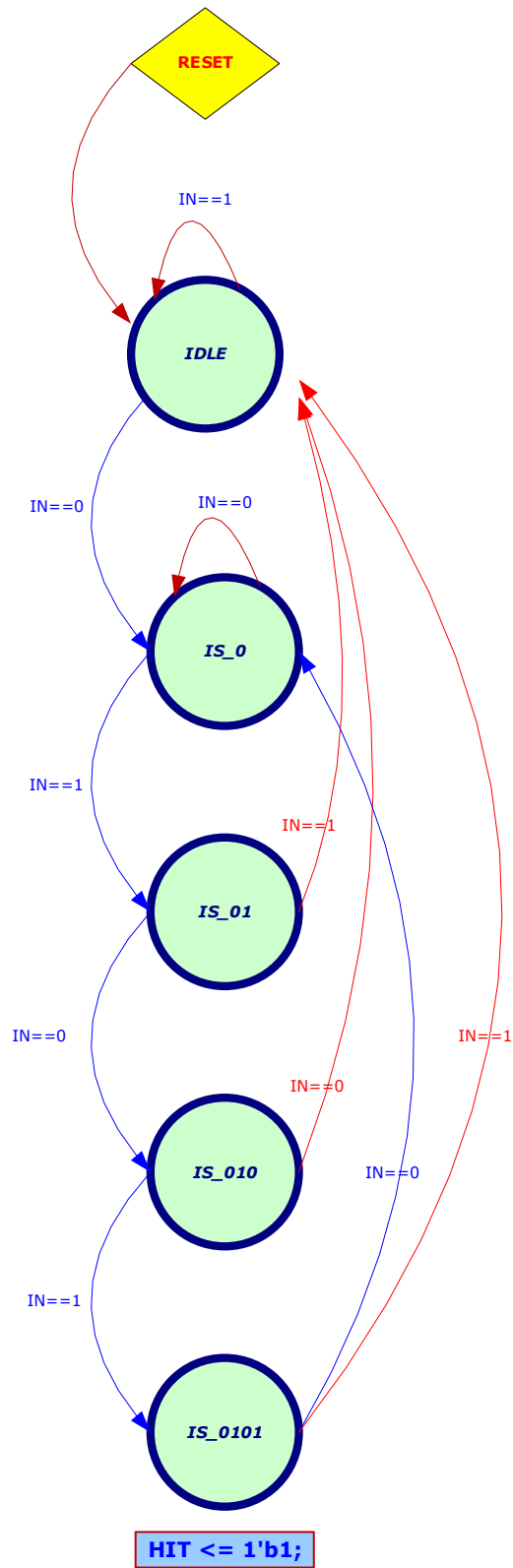
```
always @ (CLK or CLK_EN)  
if (!CLK)  
CLK_TEMP <= CLK_EN;  
assign GCLK = CLK & CLK_TEMP;
```

2-41. 以下这段代码会产生什么问题，这种问题会有什么坏处？

```
1. always@(enable or ina)
2. begin
3.     if(enable)
4.         begin
5.             data_out = ina;
6.         end
7.     else
8.         begin
9.             data_out = inb;
10.        end
11. end
```

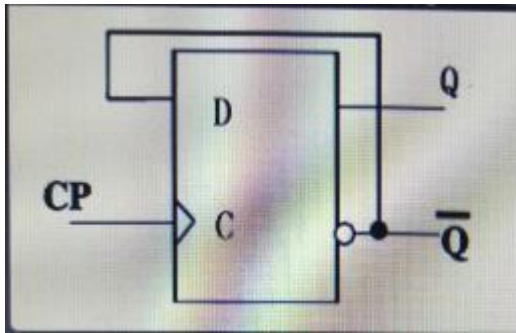
1. 应该用非阻塞赋值
2. inb 不在敏感列表

2-42. 0101 序列检测器，检测 1 位数据流中出现的 0101，出现时输出 1，画出状态机转换图并写出 RTL 代码。





2-43. 如图所示电路，若输入 CP 脉冲频率为 80KHz, 则输出 W 的频率为 (B)



- A. 80khz      B. 40khz      C. 20khz      D. 160khz

2-44. ``timescale 1ns/100ps`, 如下正确的是 (D)

- A. 时间单位是 ps  
B. 时间单位是 100ps  
C. 时间精度是 1ns  
D. 时间精度是 100ps

前面是单位，后面是精度

2-45. 常用状态机的分类

状态机由状态寄存器和组合逻辑电路构成，能够根据控制信号按照预先设定的状态进行状态转移，是协调相关信号动作，完成特定操作的控制中心。

状态机可以分为 Moore 型和 Mealy 型两种基本类型。设计时采用哪种方式的状态机要根据设计的具体情况决定，输出只由当前状态值决定则选用 Moore 型，输入信号和状态值共同决定输出则选用 Mealy 状态机

2-46. Verilog 描述的下列语句中，`a=1' b1; b=3' b101;` 那么 `{a, {2{b}}, a}` = (C)

- A. 5' b11011  
B. 7' b1110111  
C. 8' b11011011  
D. 9' b101101101

## CPU Architecture

### 2-47. RISC MCU 有什么特点?

#### 精简指令集计算机 (RISC)

采用复杂指令系统的计算机有着较强的处理高级语言的能力。这对提高计算机的性能是有益的。当计算机的设计沿着这条道路发展时，有些人没有随波逐流，他们回过头去看一看过去走过的道路，开始怀疑这种传统的做法：IBM 公司设在纽约 Yorktown 的 JhomasI. Wason 研究中心于 1975 年组织力量研究指令系统的合理性问题。因为当时已感到，日趋庞杂的指令系统不但不易实现，而且还可能降低系统性能。1979 年以帕特逊教授为首的一批科学家也开始在美国加册大学伯克莱分校开展这一研究。结果表明，CISC 存在许多缺点。首先，在这种计算机中，各种指令的使用率相差悬殊：一个典型程序的运算过程所使用的 80% 指令，只占一个处理器指令系统的 20%。事实上最频繁使用的指令是取、存和加这些最简单的指令。这样一来，长期致力于复杂指令系统的设计，实际上是在设计一种难得在实践中用得上的指令系统的处理器。同时，复杂的指令系统必然带来结构的复杂性，这不但增加了设计的时间与成本还容易造成设计失误。此外，尽管 VLSI 技术现在已达到很高的水平，但也很难把 CISC 的全部硬件做在一个芯片上，这也妨碍单片计算机的发展。在 CISC 中，许多复杂指令需要极复杂的操作，这类指令多数是某种高级语言的直接翻版，因而通用性差。由于采用二级的微码执行方式，它也降低那些被频繁调用的简单指令系统的运行速度。因而，针对 CISC 的这些弊病，帕特逊等人提出了精简指令的设想即指令系统应当只包含那些使用频率很高的少量指令，并提供一些必要的指令以支持操作系统和高级语言。按照这个原则发展而成的计算机被称为精简指令集计算机

(Reduced Instruction Set Computer-RISC) 结构，简称 RISC。

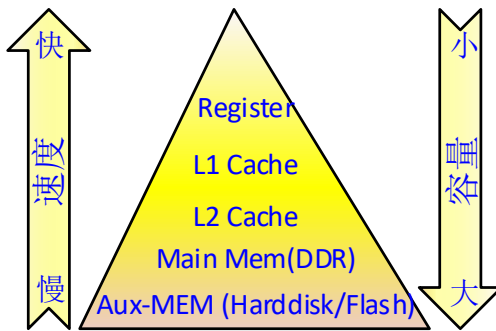
#### RISC 体系的优缺点

优点：在使用相同的晶片技术和相同运行时钟下，RISC 系统的运行速度将是 CISC 的 2~4 倍。由于 RISC 处理器的指令集是精简的，它的记忆体管理单元、浮点单元等都能设计在同一块晶片上。RISC 处理器比相对应的 CISC 处理器设计更简单，所需要的时间将变得更短，并可以比 CISC 处理器应用更多先进的技术，开发更快的下一代处理器。

缺点：多指令的操作使得程式开发者必须小心地选用合适的编译器，而且编写的代码量会变得非常大。另外就是 RISC 体系的处理器需要更快记忆体，这通常都集成于处理器内部，就是 L1 Cache（一级缓存）。

2-48. 以下是对 Cache-主存-辅存三级存储系统中各级存储器的作用，速度，容量的描述，其中完全正确的是( ) B

- A. 主存用于存放 CPU 正在执行的程序，速度慢，容量极大；
- B. Cache 用于存放 CPU 当前访问频繁的程序和数据，速度快，容量小；
- C. 辅存用于存放需要联机保存但暂不执行的程序和数据，速度快，容量极大；
- D. 加大 Cache 的容量可以使主存能够存放更多的程序和数据。



2-49. 请解释一下处理器/微控制器中所采用的流水线技术，并以取指令(fetch)、解码(decode)和执行(exe)三个步骤画出流水线指令执行过程的示意图。最后简述流水线指令执行的优点和存在的问题。

IF:Instruction Fetch, 取指令, 用到部件: 指令存储器, Adder ( 全加器, full-adder, 是用门电路实现两个二进制数相加并求出和的组合线路, 称为一位全加器。一位全加器可以处理低位进位, 并输出本位加法进位。多个一位全加器进行级联可以得到多位全加器。常用二进制四位全加器 74LS283)

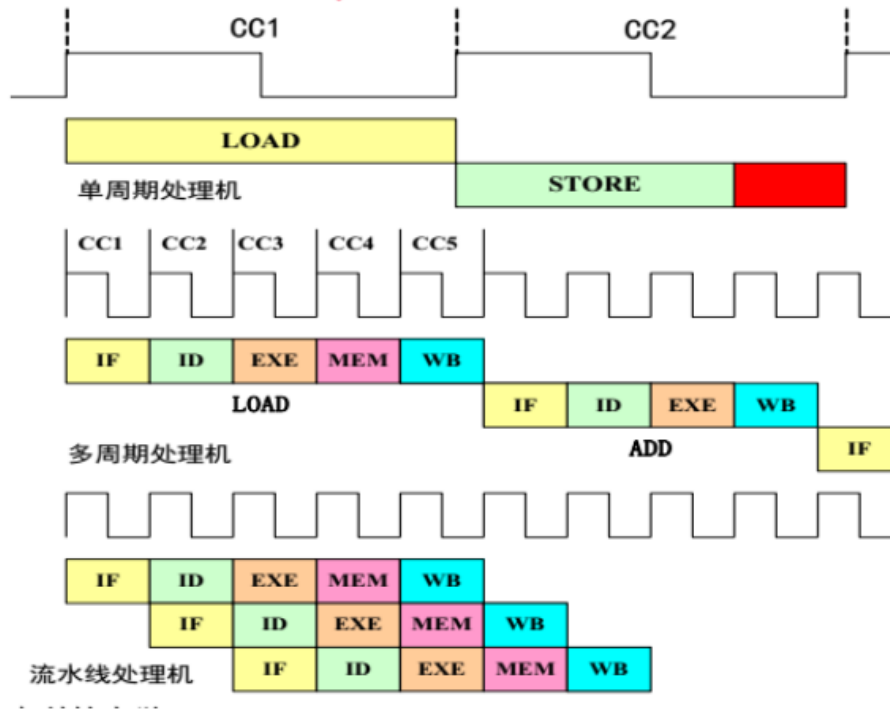
ID:Instruction Decode, 译码 (应该是取数同时译码的过程), 用到部件: 指令译码器寄存器堆读口 (这里面的寄存器堆的读口和写口可以看作两个不同的部件), 这块有大量寄存器, WB 也是从写口将数据写到这块的寄存器中。

EX:Exec, 执行, 计算内存单元地址。用到部件: ALU, 扩展器

MEM: 访存, 从数据存储器中读。用到部件: 数据存储器。

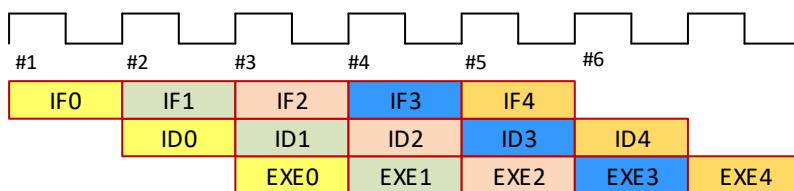
WB:Write Back, 写回, 将数据写到寄存器中。用到部件: 寄存器堆写口。

## 单周期、多周期与流水线



如果采用一个时钟周期执行一条命令，完整执行以上 5 个步骤，那么电路的频率不会高，执行效率低下。

改进的方案有单指令多周期，在多周期中按照上面五项设计流水线，各级硬件资源在每一拍都尽可能的参与工作。



途中可以看出第三拍开始三个指令都进入 CPU 内部开始执行。内部各个部件都参与工作。

优点：

有 pipeline 寄存器的插入，电路频率可以获得提升  
Pipeline 可以获得并行执行的能力

存在的问题：

会增加额外的寄存器。在各级之间需要增加寄存器以保存中间结果和控制信号。复杂度增加。需要处理相关各种问题。（结构相关/Structural hazard，数据相关/data structural hazard，转移相关/branch hazard）。

2-50. 关于流水线设计的理解，错误的是 **B**

- A. 流水线设计会消耗较多的组合逻辑资源
- B. 流水线设计的思想，是使用面积换取速度
- C. 关键路径中插入流水线，能够提高系统时钟频率
- D. 流水线设计会导致原有通路延时增加

注：因为流水的思想不是简单的面积换速度，而是硬件资源的并行合理调用。流水线设计会增加寄存器作为中间数据，状态的暂存。流水线也会增加流水控制逻辑，所以组合逻辑在控制路径上也会增加。

“延时”可以翻译为 delay，也可以是 latency。选项 D 准确讲应该是时序电路中的“latency”。多指令多周期相对单指令单周期的延时要多几拍。所以 D 的描述也是对的。

2-51. 指令系统中程序控制类指令的功能是： **C**

- A. 实现主存与 cpu 之间的数据传送
- B. 实现数字和逻辑运输
- C. 实现程序执行顺序的改变
- D. 实现堆栈操作

2-52. write-back cache 和 write-through cache 的区别：

Write-through（直写模式）在数据更新时，同时写入缓存 Cache 和后端存储。此模式的优点是操作简单；缺点是因为数据修改需要同时写入存储，数据写入速度较慢。

Write-back（回写模式）在数据更新时只写入缓存 Cache。只在数据被替换出缓存时，被修改的缓存数据才会被写到后端存储。此模式的优点是数据写入速度快，因为

不需要写存储；缺点是一旦更新后的数据未被写入存储时出现系统掉电的情况，数据将无法找回。

## 2-53. 请解释什么是 AXI outstanding, AXI out of order 以及 AXI interleaving?

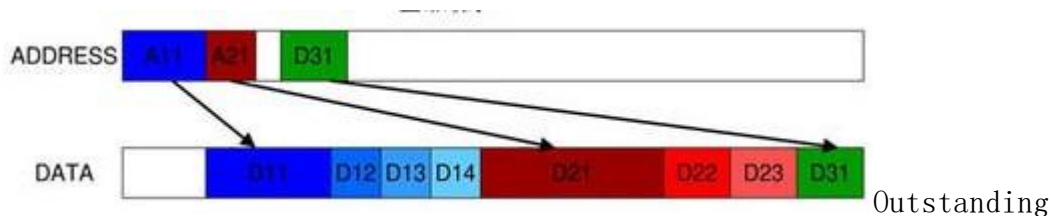
正常情况下的 master 与 slave 的操作是：

读操作：master 发送 read addr, slave 返回 data, 然后 master 发送下一笔 read addr, slave 返回下一笔 data;

写操作：master 发送 write addr 和 data, slave 返回 response, 然后 master 发送下一笔 write addr 和 data, slave 再返回下一笔 response;

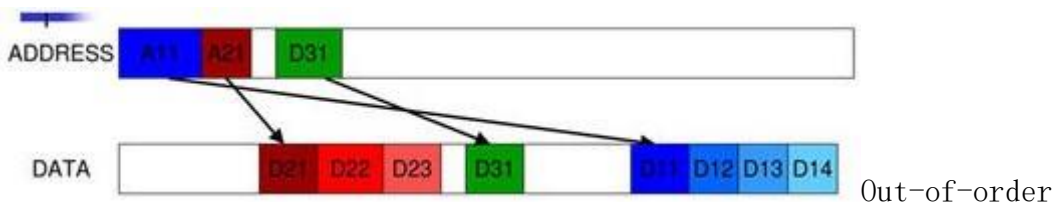
Outstanding

: The ability to issue multiple outstanding addresses means that masters can issue transaction addresses without waiting for earlier transactions to complete. This feature can improve system performance because it enables parallel processing of transactions.



发出 A11 的 addr 后，在完成 D11~D14 的 transfer 之前，发出 A21 叫做 outstanding。

Out-of-order : The ability to complete transactions out of order means that transactions to faster memory regions can complete without waiting for earlier transactions to slower memory regions. This feature can also improve system performance because it reduces the effect of transaction latency.



地址的顺序是 A11, A21, A31, 而数据顺序则可能是 D2?, D3?, D1?, 这个过程叫做 Out-of-order

Interleaving: Write data interleaving enables a slave interface to accept interleaved write data with different AWID values. The slave declares a write

data interleaving depth that indicates if the interface can accept interleaved write data from sources with different AWID values.



D11 和 D12 之间插入 D23，叫做 interleaving。

简单而言，outstanding 是对地址而言，一次 burst 还没结束，就可以发送下一相地址。而 out-of-order 和 interleaving 则是相对于 transaction，out-of-order 说的是发送 transaction 和接收的 cmd 之间的顺序没有关系，如先接到 A 的 cmd，再接到 B 的 cmd，则可以先发 B 的 data，再发 A 的 data；interleaving 指的是 A 的 data 和 B 的 data 可以交错，如 A1 B1 A2 B2 B3……

2-54. 【不定项】中断是处理器能完成并行性，实时性操作的一种重要手段，下列有关中断正确的是 **C**

- A. 中断的响应过程中，保护程序计数器的作用的是 CPU 能找到中断处理程序的入口地址
- B. CPU 在响应中断期间，原来的程序仍然可以继续运行
- C. 以上表述都不对
- D. 在中断响应中，断点保护，现场保护应该由用户编程完成

2-55. 经典的 MIPS 五级流水线是哪五级

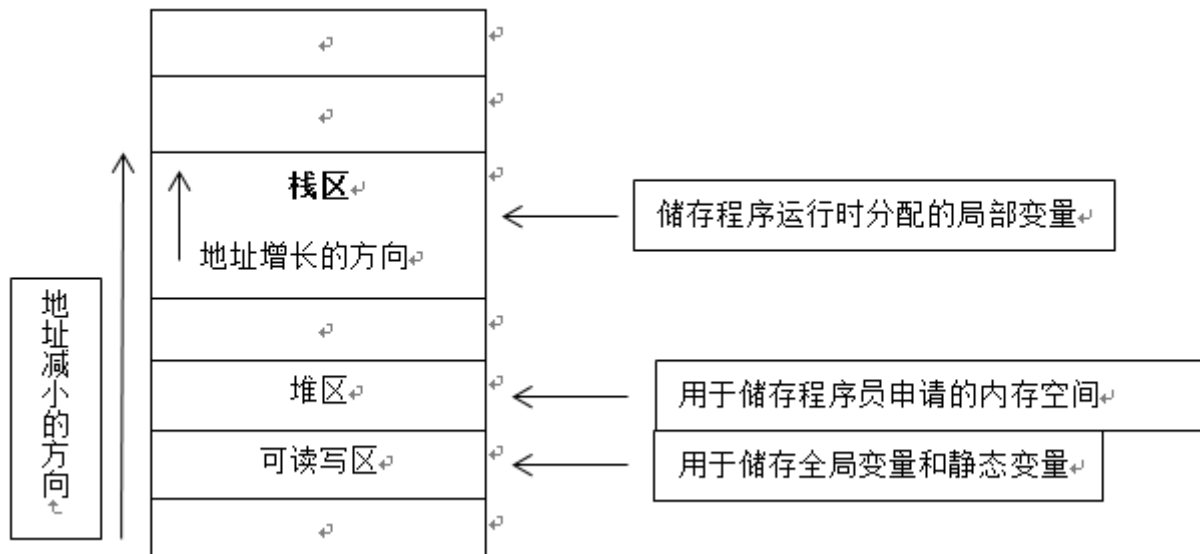
- (1) IF 取指 (instruction fetch), 从指令高速缓存 (I-cache) 获取下一条指令
- (2) RD 读取寄存器 (read register), 读取该指令的源寄存器域指定的 CPU 寄存器的内容。
- (3) ALU 算术逻辑单元 (arithmetic/logic unit) 在一个时钟周期内完成算术或者逻辑操作。
- (4) MEM 访问内存 (memory), 该阶段指令可以读写数据高速缓存 (D-cache) 中的内存变量。
- (5) WB 写回寄存器 (write back), 将操作结果值写回寄存器堆中

## 2-56. 在计算机程序中，内存被分为堆(heap)和栈(stack)，请解释二者的区别

堆(heap)：是一种非连续的树形存储数据结构，每个节点存在一个值，整个树是经过排序的，特点是根节点最小（小顶堆）或根节点最大（大顶堆），且根节点的两个子树也是一个堆。常用来实现优先队列，存取随意。

栈(stack)：是一种连续存储的数据结构，具有先进后出的性质。通常的操作有入栈（压栈），出栈和栈顶元素。想要读取某个元素就要将之前的元素全部出栈，才能完成内存中的栈区与堆区：

内存：计算机中的随机存储器(RAM)，程序都在这里运行。



栈内存：编译器自动分配和释放，存放函数的参数，局部变量，临时变量等等。

堆内存：为成员分配和释放，由程序员自己申请，自己释放。如果没有手动释放，再程序结束时由操作系统自动回收，稍有不慎会发生内存泄漏，典型为使用 new 申请堆内存。

## FPGA

### 2-57. 以下关于 FPGA 和 ASIC 的描述正确的是 ACD

- A. 相同工艺下，ASIC 能跑更快的频率
- B. FPGA 更注重对面积的要求
- C. FPGA 开发周期相对短



D. ASIC 批量生产时成本相对低

## 2-58. 简述 FPGA 可编程逻辑器件设计流程

通常可将 FPGA/CPLD 设计流程归纳为以下 7 个步骤，这与 ASIC 设计有相似之处。

1. 设计输入。 Verilog 或 VHDL 编写代码。

2. 前仿真（功能仿真）。设计的电路必须在布局布线前验证电路功能是否有效。

（ ASCII 设

计中，这一步骤称为第一次 Sign-off） PLD 设计中，有时跳过这一步。

3. 设计编译（综合）。设计输入之后就有一个从高层次系统行为设计向门级逻辑电路设转化

翻译过程，即把设计输入的某种或某几种数据格式(网表)转化为软件可识别的某种数据格式

(网表)。

4. 优化。对于上述综合生成的网表，根据布尔方程功能等效的原则，用更小更快的综合结果

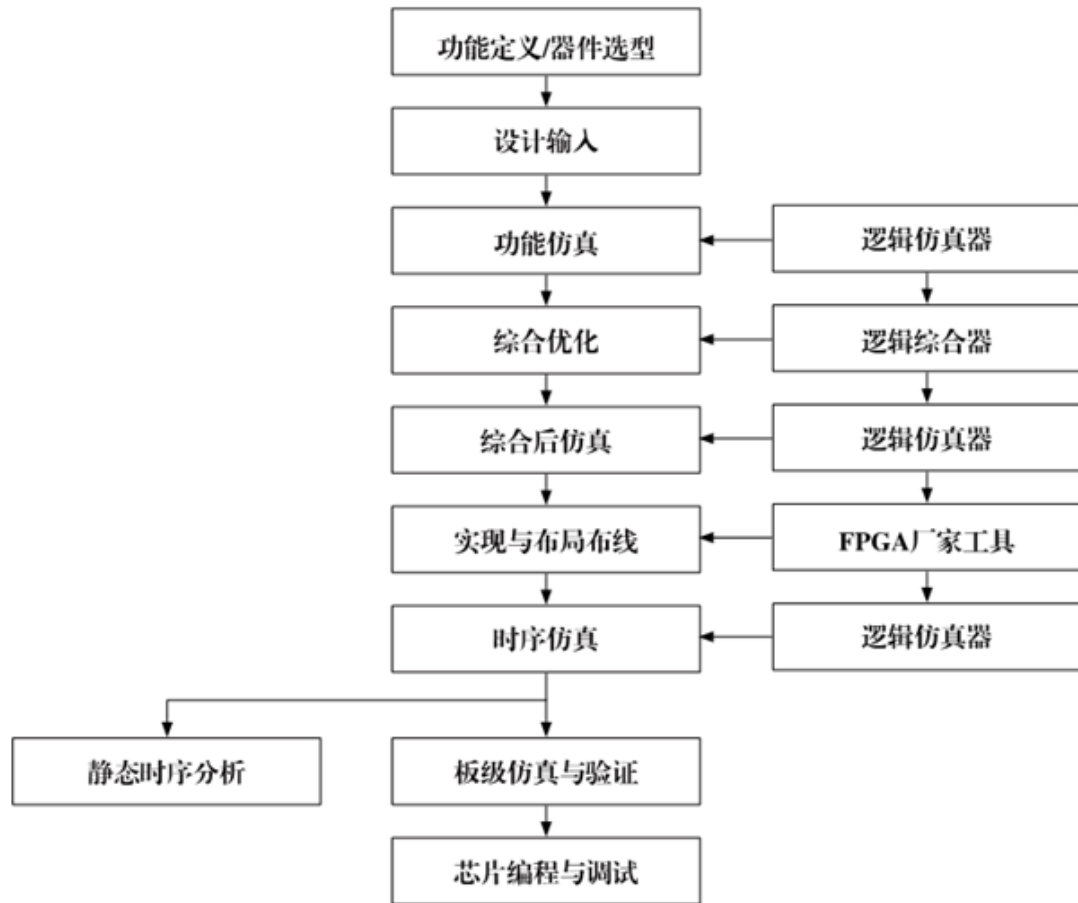
代替一些复杂的单元，并与指定的库映射生成新的网表，这是减小电路规模的一条必由之路。

5. 布局布线。

6. 后仿真（时序仿真）需要利用在布局布线中获得的精确参数再次验证电路的时序。（ ASCII

设计中，这一步骤称为第二次 Sign-off）。

7. 生产。布线和后仿真完成之后，就可以开始 ASCII 或 PLD 芯片的投产



## 2-59. FPGA 资源里的存储资源

FPGA 芯片内有两种存储器资源：一种叫 BLOCK RAM，另一种是由 LUT 配置成的内部存储器（也就是分布式 RAM）。BLOCK RAM 由一定数量固定大小的存储块构成的，使用

BLOCK RAM 资源不占用额外的逻辑资源，并且速度快。但是使用的时候消耗的 BLOCK RAM 资源是其块大小的整数倍。ADDON

## 2-60. FPGA 中可以综合实现为 RAM/ROM/CAM 的三种资源及其注意事项？

三种资源：BLOCK RAM，触发器（FF），查找表（LUT）；

注意事项：

- 1: 在生成 RAM 等存储单元时，应该首选 BLOCK RAM 资源；其原因有二：第一：使用 BLOCK RAM 等资源，可以节约更多的 FF 和 4-LUT 等底层可编程单元。使用 BLOCK RAM 可以说是“不用白不用”，是最大程度发挥器件效能，节约成本的一种体现；第二：BLOCK RAM 是一种可以配置的硬件结构，其可靠性和速度与用 LUT 和 REGISTER 构建的存储器更有优势。
- 2: 弄清 FPGA 的硬件结构，合理使用 BLOCK RAM 资源；
- 3: 分析 BLOCK RAM 容量，高效使用 BLOCK RAM 资源；
- 4: 分布式 RAM 资源（DISTRIBUTE RAM）

## 2-61. 查找表的原理与结构？

查找表（look-up-table）简称为 LUT，LUT 本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成是一个有 4 位地址线的 16x1 的 RAM。当用户通过原理图或 HDL 语言描述了一个逻辑电路以后，PLD/FPGA 开发软件会自动计算逻辑电路的所有可能的结果，并把结果事先写入 RAM，这样，每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可

## 2-62. FPGA 和 CPLD 的区别？

	CPLD	FPGA
内部结构	Product term（基于乘积项）	Look up Table（基于查找表）
程序存储	内部 EEPROM/FLASH	SRAM，外挂 EEPROM
资源类型	组合逻辑资源丰富	时序逻辑资源丰富
集成度	低	高
使用场合	完成控制逻辑	能完成比较复杂的算法
速度	慢	快 ??
其他资源	—	PLL、RAM 和乘法器等
保密性	可加密	一般不能保密

## 2-63. 你所知道的可编程逻辑器件有哪些？

#### 2-64. 什么是亚稳态？为什么两级触发器可以防止亚稳态传播？

这也是一个异步电路同步化的问题。亚稳态是指触发器无法在某个规定的时间段内到达一个可以确认的状态。使用两级触发器来使异步电路同步化的电路其实叫做“一位同步器”，他只能用来对一位异步信号进行同步。两级触发器可防止亚稳态传播的原理：假设第一级触发器的输入不满足其建立保持时间，它在第一个脉冲沿到来后输出的数据就为亚稳态，那么在下一个脉冲沿到来之前，其输出的亚稳态数据在一段恢复时间后必须稳定下来，而且稳定的数据必须满足第二级触发器的建立时间，如果都满足了，在下一个脉冲沿到来时，第二级触发器将不会出现亚稳态，因为其输入端的数据满足其建立保持时间。同步器有效的条件：第一级触发器进入亚稳态后的恢复时间 + 第二级触发器的建立时间  $\leq$  时钟周期。<sup>2</sup>更确切地说，输入脉冲宽度必须大于同步时钟周期与第一级触发器所需的保持时间之和。最保险的脉冲宽度是两倍同步时钟周期。所以，这样的同步电路对于从较慢的时钟域来的异步信号进入较快的时钟域比较有效，对于进入一个较慢的时钟域，则没有作用。

#### 2-65. 对于多位的异步信号如何进行同步？

对以一位的异步信号可以使用“一位同步器进行同步”（使用两级触发器），而对于多位的异步信号，可以采用如下方法：

- 1：可以采用保持寄存器加握手信号的方法（多数据，控制，地址）；
- 2：特殊的具体应用电路结构, 根据应用的不同而不同；
- 3：异步 FIFO。（最常用的缓存单元是 DPRAM）

#### 2-66. 什么是竞争与冒险现象?怎样判断?如何消除?

在组合电路中，某一输入变量经过不同途径传输后，到达电路中某一汇合点的时间有先有后，这种现象称竞争；由于竞争而使电路输出发生瞬时错误的现象叫做冒险。

（也就是由于竞争产生的毛刺叫做冒险）。

判断方法：

- 代数法（如果布尔式中有相反的信号则可能产生竞争和冒险现象）；

- 卡诺图：有两个相切的卡诺圈并且相切处没有被其他卡诺圈包围，就有可能出现竞争冒险；
- 实验法：示波器观测；

解决方法： 1：加滤波电容，消除毛刺的影响； 2：加选通信号，避开毛刺； 3：增加冗余项消除逻辑冒险。

门电路两个输入信号同时向相反的逻辑电平跳变称为竞争；

由于竞争而在电路的输出端可能产生尖峰脉冲的现象称为竞争冒险。

如果逻辑函数在一定条件下可以化简成  $Y=A+A'$  或  $Y=AA'$  则可以判断存在竞争冒险现象（只是一个变量变化的情况）。

消除方法，接入滤波电容，引入选通脉冲，增加冗余逻辑

## 2-67. MOORE 与 MEELEY 状态机的特征？

Moore 状态机的输出仅与当前状态值有关，且只在时钟边沿到来时才会有状态变化。

Mealy 状态机的输出不仅与当前状态值有关，而且与当前输入值有关。

## 2-68. 多时域设计中, 如何处理信号跨时域？

不同的时钟域之间信号通信时需要进行同步处理，这样可以防止新时钟域中第一级触发器的亚稳态信号对下级逻辑造成影响。

信号跨时钟域同步：当单个信号跨时钟域时，可以采用两级触发器来同步；数据或地址总线跨时钟域时可以采用异步 FIFO 来实现时钟同步；第三种方法就是采用握手信号。

## 2-69. 说说静态、动态时序模拟的优缺点？

静态时序分析是采用穷尽分析方法来提取出整个电路存在的所有时序路径，计算信号在这些路径上的传播延时，检查信号的建立和保持时间是否满足时序要求，通过对最大路径延时和最小路径延时的分析，找出违背时序约束的错误。它不需要输入向量就能穷尽所有的路径，且运行速度很快、占用内存较少，不仅可以对芯片设计进行全面的时序功能检查，而且还可利用时序分析的结果来优化设计，因此静态时序分析已经

越来越多地被用到数字集成电路设计的验证中。动态时序模拟就是通常的仿真，因为不可能产生完备的测试向量，覆盖门级网表中的每一条路径。因此在动态时序分析中，无法暴露一些路径上可能存在的时序问题；

## 2-70. 用 D 触发器做个二分频的电路？画出逻辑电路？

```
module div2(clk,rst,clk_out);  
input clk,rst;  
output reg clk_out;  
always@(posedge clk)  
begin  
    if(!rst)  
        clk_out <=0;  
    else  
        clk_out <=~ clk_out;  
    end  
endmodule
```

现实工程设计中一般不采用这样的方式来设计，二分频一般通过 DCM 来实现。通过 DCM 得到的分频信号没有相位差。

或者是从 Q 端引出加一个反相器。

## 2-71. SRAM, FLASH MEMORY, DRAM, SSRAM 及 SDRAM 的区别？

SRAM：静态随机存储器，存取速度快，但容量小，掉电后数据会丢失，不像 DRAM 需要不停的 REFRESH，制造成本较高，通常用来作为快取(CACHE) 记忆体使用。

FLASH：闪存，存取速度慢，容量大，掉电后数据不会丢失

DRAM：动态随机存储器，必须不断的重新加强(REFRESHED) 电位差量，否则电位差将降低至无法有足够的能量表现每一个记忆单位处于何种状态。价格比 SRAM 便宜，但访问速度较慢，耗电量较大，常用作计算机的内存使用。

SSRAM：即同步静态随机存取存储器。对于 SSRAM 的所有访问都在时钟的上升/下降沿启动。地址、数据输入和其它控制信号均于时钟信号相关。

SDRAM：即同步动态随机存取存储器。



压控制的一种放大器件。是组成 CMOS 数字集成电路的基本单元。

MCU(Micro Controller Unit)中文名称为微控制单元，又称单片微型计算机(Single Chip Microcomputer)或者单片机，是指随着大规模集成电路的出现及其发展，将计算机的 CPU、RAM、ROM、定时计数器和多种 I/O 接口集成在一片芯片上，形成芯片级的计算机，为不同的应用场合做不同组合控制。

RISC (reduced instruction set computer，精简指令集计算机)是一种执行较少类型计算机指令的微处理器，起源于 80 年代的 MIPS 主机（即 RISC 机），RISC 机中采用的微处理

器统称 RISC 处理器。这样一来，它能够以更快的速度执行操作（每秒执行更多百万条指令，即 MIPS）。因为计算机执行每个指令类型都需要额外的晶体管和电路元件，计算机指令集越大就会使微处理器更复杂，执行操作也会更慢。

CISC 是复杂指令系统计算机 (Complex Instruction Set Computer) 的简称，微处理器是台式计算机系统的基本处理部件，每个微处理器的核心是运行指令的电路。指令由完成任务的多个步骤所组成，把数值传送进寄存器或进行相加运算。

DSP (digital signal processor) 是一种独特的微处理器，是以数字信号来处理大量信息的器件。其工作原理是接收模拟信号，转换为 0 或 1 的数字信号。再对数字信号进行修改、删除、强化，并在其他系统芯片中把数字数据解译回模拟数据或实际环境格式。它不仅具有可编程性，而且其实时运行速度可达每秒数以千万条复杂指令程序，远远超过通用微处理器，是数字化电子世界中日益重要的电脑芯片。它的强大数据处理能力和高运行速度，是最值得称道的两大特色。

FPGA (Field-Programmable Gate Array)，即现场可编程门阵列，它是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。它是作为专用集成电 (ASIC) 领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。

ASIC:专用集成电路，它是面向专门用途的电路，专门为一个用户设计和制造的。根据一个用户的特定要求，能以低研制成本，短、交货周期供货的全定制，半定制集成电路。与门阵列等其它 ASIC(Application Specific IC)相比，它们又具有设计开发周期短、设计制造成本低、开发工具先进、标准产品无需测试、质量稳定以及可实时在线检验等优点

PCI(Peripheral Component Interconnect) 外围组件互连，一种由英特 (Intel) 公司 1991 年推出的用于定义局部总线的标准。

ECC 是 “Error Correcting Code” 的简写，中文名称是 “错误检查和纠正”。ECC



是一种能够实现“错误检查和纠正”的技术，ECC 内存就是应用了这种技术的内存，一般多应用在服务器及图形工作站上，这将使整个电脑系统在工作时更趋于安全稳定。

DDR=Double Data Rate 双倍速率同步动态随机存储器。严格的说 DDR 应该叫 DDR SDRAM，人们习惯称为 DDR，其中，SDRAM 是 Synchronous Dynamic Random Access Memory 的缩写，即同步动态随机存取存储器。

IRQ 全称为 Interrupt Request，即是“中断请求”的意思（以下使用 IRQ 称呼）。IRQ 的作用就是在我们所用的电脑中，执行硬件中断请求的动作，用来停止其相关硬件的工作状态

USB，是英文 Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通串线”，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。

BIOS 是英文“Basic Input Output System”的缩略语，直译过来后中文名称就是“基本输入输出系统”。其实，它是一组固化到计算机内主板上一个 ROM 芯片上的程序，它保存着计算机最重要的基本输入输出的程序、系统设置信息、开机后自检程序和系统自启动程序。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。



路科芯院

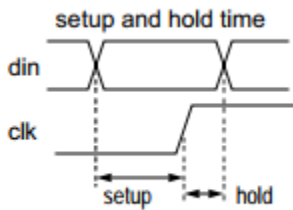
---

## PART THREE: TIMING

---

3-1. 解释以下概念(可图文并茂):

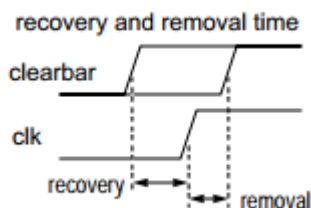
- a) 建立时间 (setup time)
- b) 保持时间 (hold time)
- c) 恢复时间 (recovery time)
- d) 移除时间 (removal time)
- e) 时钟偏差 (clock skew)
- f) 时钟抖动 (clock jitter)



参考上图。

建立时间：触发器（DFF）时钟上升沿**之前**数据需要保持稳定的最小时间间隙就是建立时间。

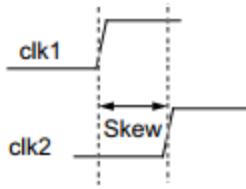
保持时间：触发器（DFF）时钟上升沿**之后**数据需要保持稳定的最小时间间隙就是保持时间。



参考上图。

恢复时间：触发器（DFF）时钟上升沿**之前** clearbar 需要保持稳定的最小时间间隙就是恢复时间。

撤除时间：触发器（DFF）时钟上升沿**之后** clearbar 需要保持稳定的最小时间间隙就是撤除时间。

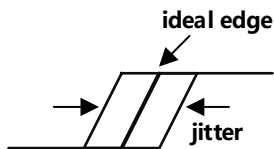


时钟树建立完毕，由于路径延时，clock 各个节点上边沿变化有早有晚。两两 DFF. CP 之间的时间差称之为时钟偏移（Clock Skew）。

### 3-2. 回答以下关于时钟的问题

- 请解释什么是 clock jitter?
- 简述 clock jitter 对时序电路的影响
- 在时序分析时如何考虑 clock jitter 对时序的影响

。



参考上图：

理想时钟没有延时，没有抖动。在实际电路中，时钟树受到温度变化，电压波动，噪声，干扰等各种影响，时钟的边沿在一个 DFF CP 端始终有波动漂移。

Clock jitter 在 STA 分析中表示为 clock uncertainty.

在时序电路中，由于 clock jitter 的不确定性，clock 边沿或早或晚，我们考虑它的最坏情况。

Setup time 计算中，数据要求延迟值（Required Time）计算公式中的 clock uncertainty 取值为负。

Hold time 计算中，数据要求延迟值（Required Time）计算公式中的 clock uncertainty 取值为正。

### 3-3. 请问亚稳态是什么，如何产生的，应如何避免亚稳态？

亚稳态是指触发器无法在某个规定时间段内达到一个可确认的状态。

当一个触发器进入亚稳态时，既无法预测该单元的输出电平，也无法预测何时输出才能稳定在某个正确的电平上。

在这个稳定期间，触发器输出一些中间级电平，或者可能处于振荡状态，并且这种无用的输出电平可以沿信号通道上的各个触发器级联式传播下去。

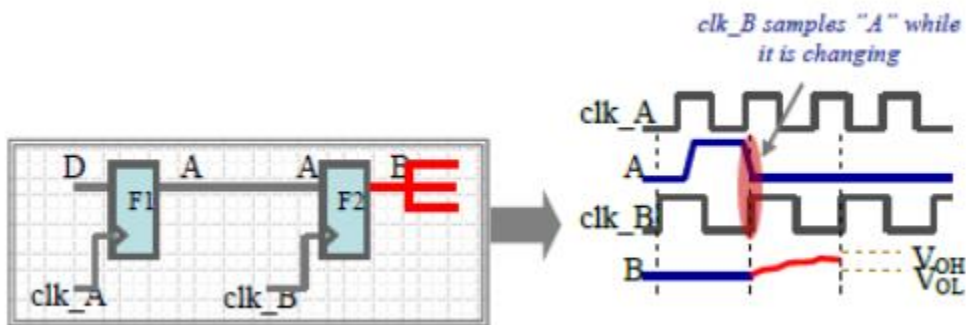
亚稳态问题的产生是由于触发器的 Timing violation (setup/hold/recovery/removal)。

同步时序电路中，因为频率，datapath 延时，clockpath 延时造成 setup/hold violation。

解决的方法有：

- 降低时钟频率
- 优化组合逻辑路径
- 优化时钟树
- 提高时钟驱动能力，改善时钟质量
- 单元库中选择反应更快的 DFF

异步时序电路中，由于 CDC (Clock Domain Crossing) 问题造成亚稳态 (Metastability)。  
如图例：



Clk\_A, Clk\_B 是完全异步的时钟，在 F2.A 端不能保证 setup/hold time，F2.B 端产生不确定的信号，并且在 B 端后面的各个路径中传播。因为线路延迟不同，不确定的信号值在 F2 后面扇出路径中各个 DFF 节点上必然更加恶化，出现信号值错乱导致逻辑功能错乱。

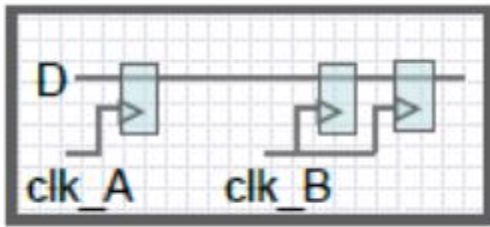
CDC 亚稳态的问题可以归纳到两个方面：

- 控制路径
- 数据路径

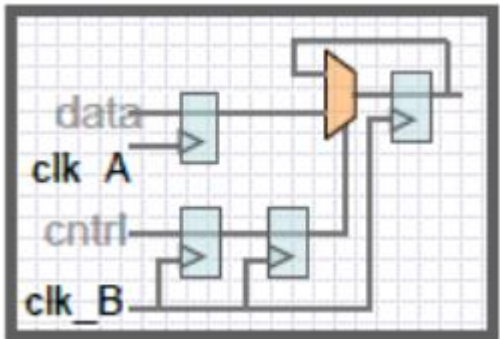
控制路径上的信号多属于 single-bit，在 signal-bit 路径上插入 sync-cell。

数据路径上的信号可以是 signal-bit 或者 data bus。采取的手段是“使能选通” (enable techniques)。首先确保数据稳定，再使能接收端的 capture 动作。

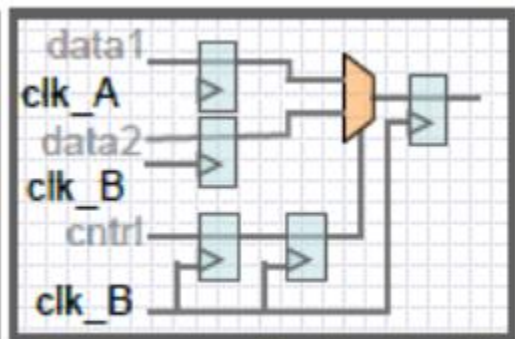
以下是解决亚稳态问题常用的技术。



2-flop



Common mux



Common mux without recirculation

- 3-4. 下列关于亚稳态描述正确的是 **B**
- A. 正常工作的电路中一定不存在亚稳态
  - B. 亚稳态是一种介于逻辑 1 和逻辑 0 之间的状态，可能引起电路解析的歧义
  - C. 在其他条件相同时，时钟频率越高，产生亚稳态的概率越低
  - D. 电路设计中的亚稳态问题可以在 RTL 阶段的仿真中发现
- 3-5. 跨时钟域的电路常用的处理方法有哪些，请举例。
- Data Sync-cell
  - Reset sync-cell
  - 选通使能 (qualifier)
  - 握手

- 格雷码
- FIFO

3-6. 关于建立(setup)和保持(hold)时间的表述哪些是正确的? C

- a) 解决 hold time violations 的方法之一是适当降低时钟频率
- b) Setup time 不受系统时钟频率影响
- c) 解决 Setup time violations 的方法之一是适当增加 capture 端 clock path delay
- d) Setup time 指的是有效时钟沿来临之后数据需要保持稳定的时间

3-7. 已知两级寄存器中含有组合逻辑, 组合逻辑延时为  $T_{cdelay}$ , 寄存器建立时间为  $T_{setup}$ , 保持时间为  $T_{hold}$ , 传输延时为  $T_{c2q}$ , 寄存器时钟周期为  $T_{clk}$ , 最小为 ( ), 组合逻辑延迟最大为 ( )

$T_{hold}$  参数和频率周期没有关系。

不考虑 clock launch path delay( $T_{launch}$ )和 capture path delay( $T_{capture}$ ) (视为同一个时刻), 周期公式可以简化为:

$$T_{c2q} + T_{cdelay} + T_{setup} \leq T_{clk}$$

$T_{clk}$  的最小值就是:  $T_{c2q} + T_{cdelay} + T_{setup}$

$$T_{cdelay} \leq T_{clk} - T_{c2q} - T_{setup}$$

$T_{cdelay}$  的最大值就是:  $T_{clk} - T_{c2q} - T_{setup}$

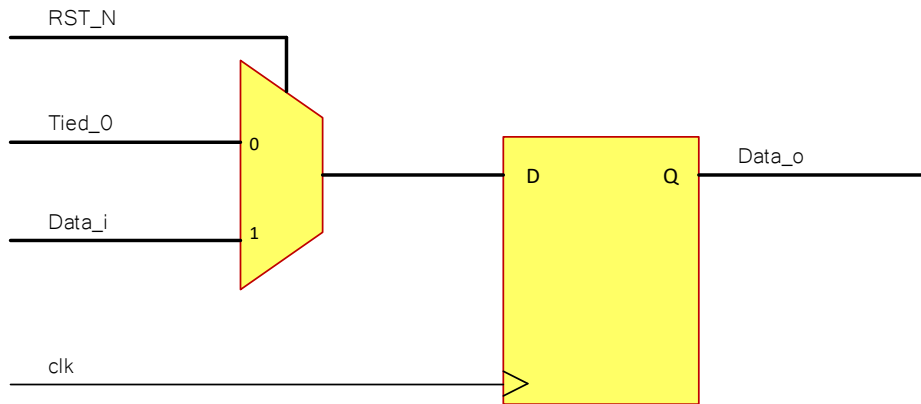
3-8. 请给出至少 3 个波形中出现 X 的原因?

1. Timing Violation (setup/hold/recovery/removal)
2. 非三态多驱动冲突
3. X 传播

3-9. 请说明数字电路中同步复位和异步复位的区别

异步复位是指 DFF 的复位清零是响应它的异步复位端(RST\_N)。复位的触发和撤销与 clock 边沿无关。(异步复位在实际应用中必须插入复位同步电路, 确保异步复位, 同步释放。)

同步复位的清零来自 DFF.D 端的组合逻辑设置, 对 RST\_N 的响应在 clock 的边沿上。如下图:



### 3-10. 关于异步复位，以下说法正确的是:A

- a) 寄存器的时钟状态对是否能复位没有影响
- b) 寄存器的时钟状态对是否能解复位没有影响
- c) 复位信号上是否有毛刺没有影响
- d) 异步复位信号不需要同步到对应的时钟域上

注:

单纯的异步复位在实际的工程中不能应用。为了能够满足 recovery/removal time 的要求，复位信号的时序是异步复位，同步释放。

### 3-11. 用 verilog 写一段代码，实现消除一个 glitch

题目关于 glitch 的描述不具体。

组合逻辑 glitch? (竞争冒险, 插入冗余项, Delay buffer, 滤波)

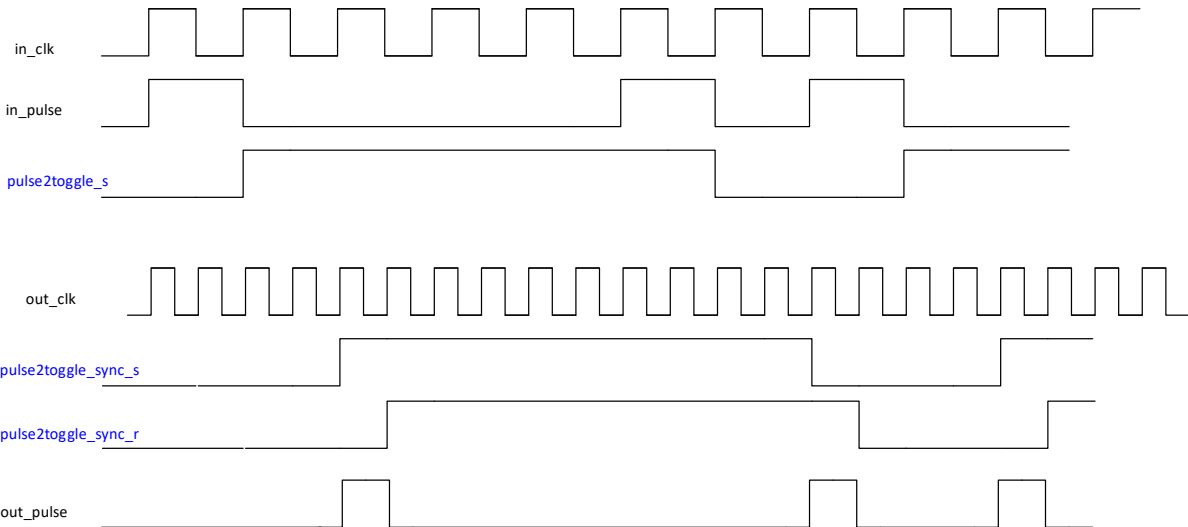
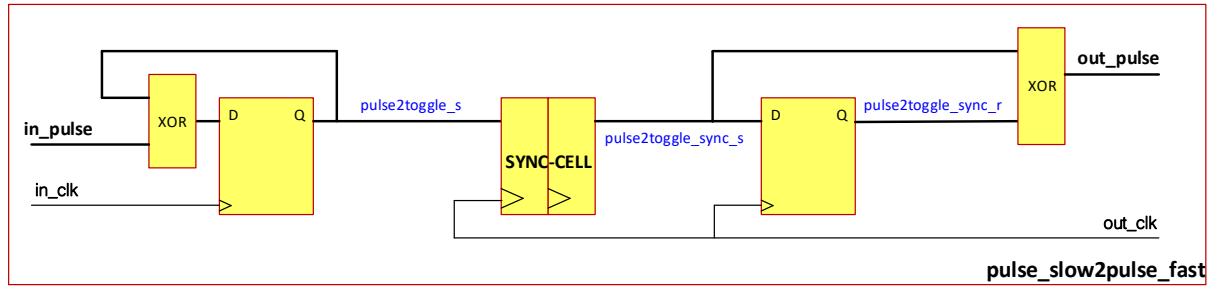
时序逻辑 data path glitch? (Insert Sync-cell)

时序逻辑 clock path glitch? (Review clock switch circuit)

### 3-12. 脉冲同步器的基本功能是从某个时钟域取出一个单时钟宽度脉冲，然后在新的时钟域中建立另一个单时钟宽度的脉冲，请按照下面的模块接口信号设计一个脉冲检测器，并说明你设计的脉冲检测器的使用限制

- a) module sync\_pulse(
- b) input in\_rst\_n,
- c) input in\_clk,
- d) input in\_pulse,
- e) output out\_rst\_n,
- f) output out\_clk,
- g) output out\_pulse);
- h) .....
- i) endmodule





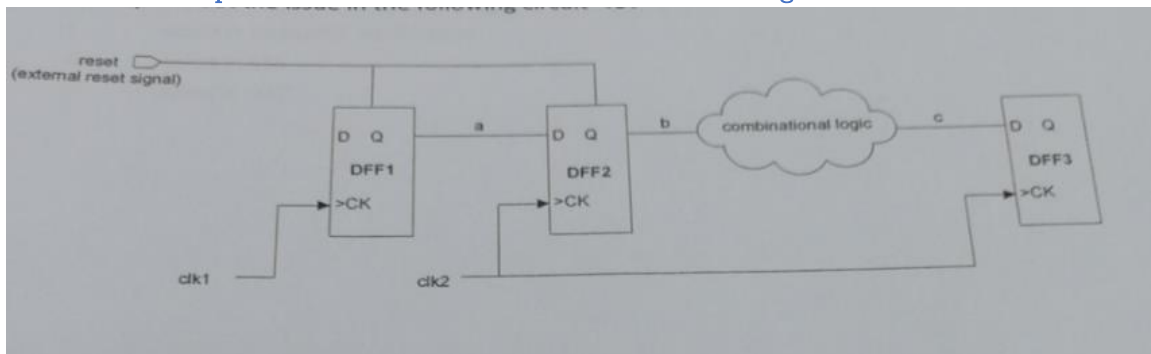
以上图示典型的 pulse2pulse 实现。

这个 pulse2pulse 电路完成从慢时钟域到快时钟域 pulse 的转换。

电路的使用限制：

1. 这个 P2P 电路普遍实用于 slow2fast 时钟域的 pulse 转换。
2. 如果要设计一个 fast2slow 的 pulse2pulse 电路，电路设计的方式是握手 (back\_pressure) 或者分路 (no back\_pressure)。

### 3-13. Please point out the issue in the following circuit

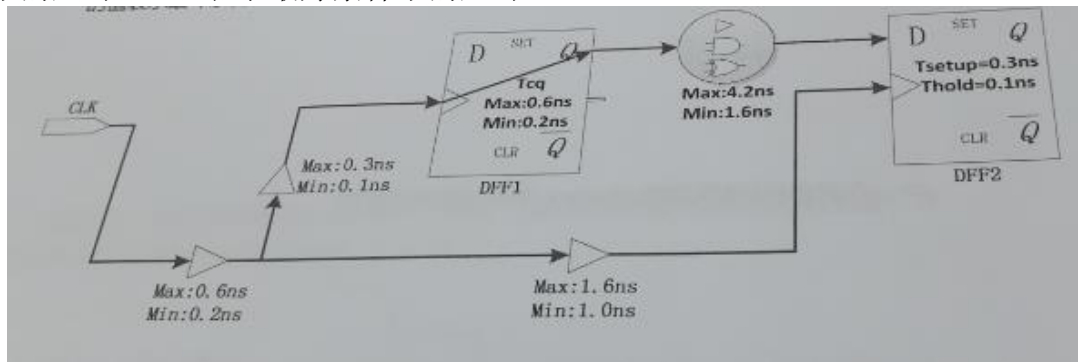


1. Missing reset-synce for clk1 domain and clk2 domain.

2. Missing data sync between clk1 and clk2 domain.
3. Missing reset control in DFF3.

### 3-14. 根据下图回答问题

如图所示为两个 D 触发器的时序路径，其中  $T_{cq}$  表示从 D 触发器时钟端到输出端的延时， $T_{setup}$  表示触发器建立时间要求， $T_{hold}$  表示触发器的保持时间要求，Max 的值表示最坏条件下的延时，Min 表示最好条件下的延时。



- a) 若时钟周期为 4ns，是否存在 setup time 和 hold time 违反，给出计算过程
- b) 假如如上电路中存在 setup 时序违反，请给出解决思路（至少三种）
- c) 假设工艺尺寸为 0.18um，请说明温度和电压对器件延时的影响

解题思路：

1. 求解 setup/hold timing 时需要考虑数据路径和时钟路径的延时
2. 参数的选择都在各自最坏工作情况下
3. CLK 后面的 buffer 作为公共 clock path 可以不用重复计算，更不适合在同一工作条件下选用相互极端的参数。由此 clock path 的起始点可以从第一个 buffer 后面开始。（概念类似“时钟路径悲观移除”，CPPR (Clock Path Pessimism Removal)

时序电路对 setup time 的要求如下公式：

$$T_{launch} + T_{clk2q} + T_{datapath} \leq T_{capture} + T_{cycle} - T_{setup}$$

考虑到最坏情况，选择各项合适的 Min/Max：

$$T_{launch\_max} + T_{clk2q\_max} + T_{datapath\_max} \leq T_{capture\_min} + T_{cycle} - T_{setup}$$

如果再考虑 clock jitter 的影响：

$$T_{launch\_max} + T_{clk2q\_max} + T_{datapath\_max} \leq T_{capture\_min} + T_{cycle} - T_{uncertainty} - T_{setup}$$

等价于：

$$(T_{capture\_min} + T_{cycle} - T_{uncertainty} - T_{setup}) - (T_{launch\_max} + T_{clk2q\_max} + T_{datapath\_max}) \geq 0$$

理解为:

$$\text{Required\_Time} - \text{Arrived\_Time} \geq 0$$

Required\_Time 与 Arrived\_Time 只差称之为 setup timing report 中的 slack。

Slack  $\geq 0$ , setup 满足。反之, setup violation。

根据以上推导手工计算 setup timing :

Required\_Time

$$= T_{\text{capture\_min}} + T_{\text{cycle}} - T_{\text{uncertainty}} - T_{\text{setup}}$$

$$= 1.0 + 4.0 - 0 - 0.3$$

$$= 4.7\text{ns}$$

Arrived\_Time

$$= T_{\text{launch\_max}} + T_{\text{clk2q\_max}} + T_{\text{datapath\_max}}$$

$$= 0.3 + 0.6 + 4.2$$

$$= 5.1\text{ ns}$$

Slack

$$= \text{Required\_Time} - \text{Arrived\_Time}$$

$$= 4.7 - 5.1$$

$$= -0.4$$

Slack 为负数说明有 setup violation。

时序电路对 Hold time 的要求如下公式:

$$T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} \geq T_{\text{capture}} + T_{\text{hold}}$$

考虑到最坏情况, 选择各项合适的 Min/Max:

$$T_{\text{launch\_min}} + T_{\text{clk2q\_min}} + T_{\text{datapath\_min}} \geq T_{\text{capture\_max}} + T_{\text{hold}}$$

如果再考虑 clock jitter 的影响:

$$T_{\text{launch\_min}} + T_{\text{clk2q\_min}} + T_{\text{datapath\_min}} \geq T_{\text{capture\_max}} + T_{\text{hold}} + T_{\text{uncertainty}}$$

等价于:

$$(T_{\text{launch\_min}} + T_{\text{clk2q\_min}} + T_{\text{datapath\_min}}) -$$

$$(T_{\text{capture\_max}} + T_{\text{hold}} + T_{\text{uncertainty}}) \geq 0$$

理解为:

$$\text{Arrived\_Time} - \text{Required\_Time} \geq 0$$

Arrived\_Time 与 Required\_Time 只差称之为 hold timing report 中的 slack。

根据以上推导手工计算 hold time:

Arrived\_Time

$$= T_{\text{launch\_min}} + T_{\text{clk2q\_min}} + T_{\text{datapath\_min}}$$

$$= 0.1 + 0.2 + 1.6$$

$$= 1.9\text{ ns}$$

Required\_Time  
= T\_capture\_max + T\_hold + T\_uncertainty  
= 1.6 + 0.1 + 0  
= 1.7 ns

Slack  
= Arrived\_Time - Required\_Time  
= 1.9 - 1.7  
= 0.2 ns

Slack 为正值，说明没有 hold violation.  
(Setup/Hold timing 到此计算完毕。)

将 setup time 计算公式重新整理：

$$T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} \leq T_{\text{capture}} + T_{\text{cycle}} - T_{\text{setup}}$$

等价与：

$$T_{\text{setup}} \leq (T_{\text{capture}} + T_{\text{cycle}}) - (T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}})$$

等价于：

$$T_{\text{setup}} \leq T_{\text{cycle}} + (T_{\text{capture}} - T_{\text{launch}}) - T_{\text{clk2q}} - T_{\text{datapath}}$$

从这个公式可以看出，如果需要增加 setup 窗口的裕度，可以从以下几个方面进行操作：

- a) 减小时钟频率，增大时钟周期
- b) 调整时钟树，提前发射时钟（变短），推后捕获时钟（变长）
- c) 采用速度更快的触发器
- d) 化简优化组合逻辑
- e) 或者更进一步打断组合逻辑，插入 pipeline

PVT 曲线表明：

- f) 电压增大，延迟减小
- g) 温度怎加，延迟增大
- h) 工艺在 best case 下，延迟减小

总结：

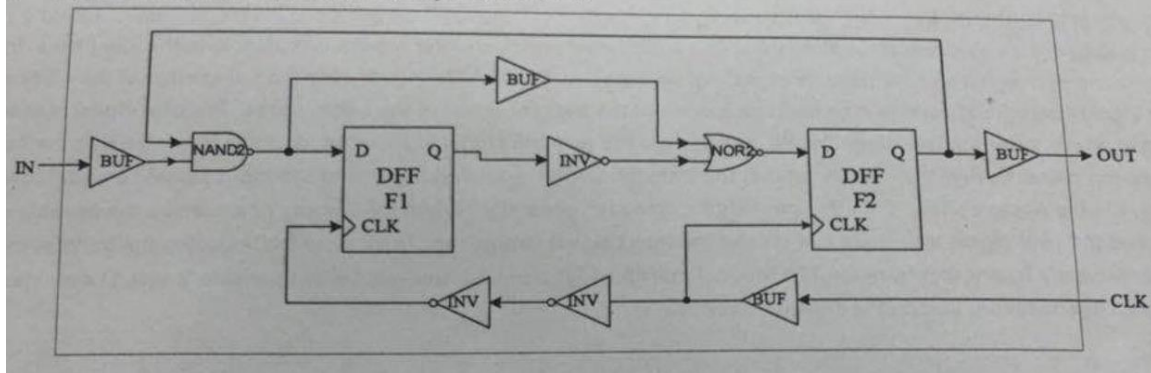
- 1. Hold time 与频率周期无关
- 2. Setup time 的公式可以变换为：

$$T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} \leq T_{\text{capture}} + T_{\text{cycle}} - T_{\text{setup}} \quad (\text{原式})$$

$$T_{\text{cycle}} \geq T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} - T_{\text{capture}} + T_{\text{setup}}$$

右侧表达式的最小值决定了频率的最大值。

3-15. 有一电路图，假设输入 IN 的 input delay 恒定为 1ns:



图中器件的时序特性为:

inverter:  $T_{INV\_MAX}$ : 1ns;  $T_{INV\_MIN}$ : 0.5ns

buffer:  $T_{BUF\_MAX}$ : 2ns;  $T_{BUF\_MIN}$ : 1ns

NAND2:  $T_{NA\_MAX}$ : 1.8ns;  $T_{NA\_MIN}$ : 0.9ns

NOR2:  $T_{NO\_MAX}$ : 2ns;  $T_{NO\_MIN}$ : 1ns

DFF:

$T_{CLK \rightarrow Q}$ : max delay: 4ns; min delay: 1ns

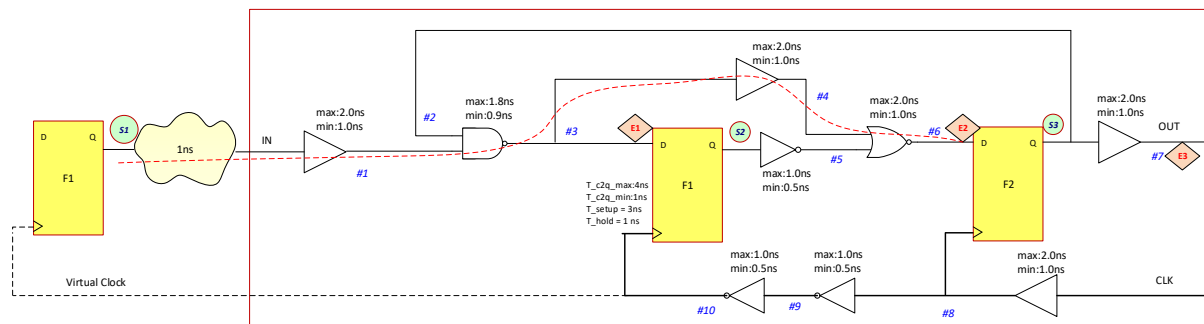
$T_{SETUP}$ : 3ns;  $T_{HOLD}$ : 1ns

根据以上信息，请回答以下问题:

- 上述电路有时序问题吗? 如有，请指出并修改。
- 该电路的最高工作频率是多少?

首先在原图中增加标记和信息:

- 时序弧(Timing Arc)起始点，结束点
- 节点编号
- 外部延时
- Virtual Clock



列出所有的时序弧的路径，找出延迟最长的路径。

Datapath1: S1→E1: S1 → #1→#3→E1

Datapath2: S1→E2: S1→#1→#3→#4→#6 → E2

Datapath3: S1→E3: NONE

Datapath4: S2→E1: NONE

Datapath5: S2→E2: S2→ #5 → #6 → E2  
 Datapath6: S2→ E3:NONE  
 Datapath7: S3→ E1:S3→#2→#3→E1  
 Datapath8: S3→ E2:S3→#2→#3→#4→#6 → E2  
 Datapath9: S3→ E3: S3→#7→E3

可以看出最长路径(critical path)是红色标注的 datapath2 (S1→E2)。

$T_{dp2\_max}$   
 = S1→#1→#3→#4→#6 → E2  
 = 1+2+1.8+2+2  
 = 8.8 ns  
 $T_{dp2\_min}$   
 = S1→#1→#3→#4→#6 → E2  
 = 1+1+0.9+1+1  
 = 4.9 ns

$T_{launch} = 0$   
 $T_{capture\_max} = 2.0ns$   
 $T_{capture\_min} = 1.0 ns$

$T_{capture} + T_{cycle} - T_{setup} \geq T_{launch} + T_{ck2q} + T_{dp}$   
 等价于:

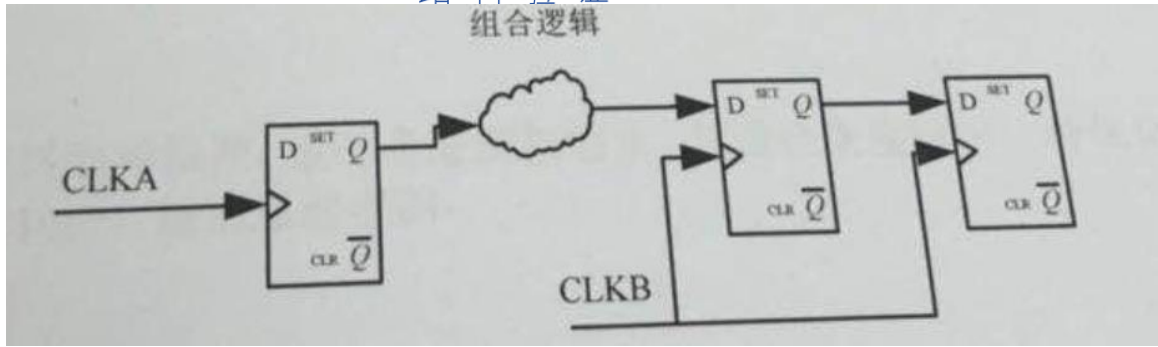
$T_{cycle} \geq T_{launch} + T_{ck2q} + T_{dp} - T_{capture} + T_{setup}$   
 频率最大即是要获得周期最小值。

电路在最优情况下可以得到更高的频率。选择 delay 的最小值。

$T_{cycle\_min}$   
 =  $T_{launch\_min} + T_{ck2q\_min} + T_{dp\_min} - T_{capture\_max} + T_{setup}$   
 = 0 + 1 + 4.9 - 2.0 + 3  
 = 6.9 ns

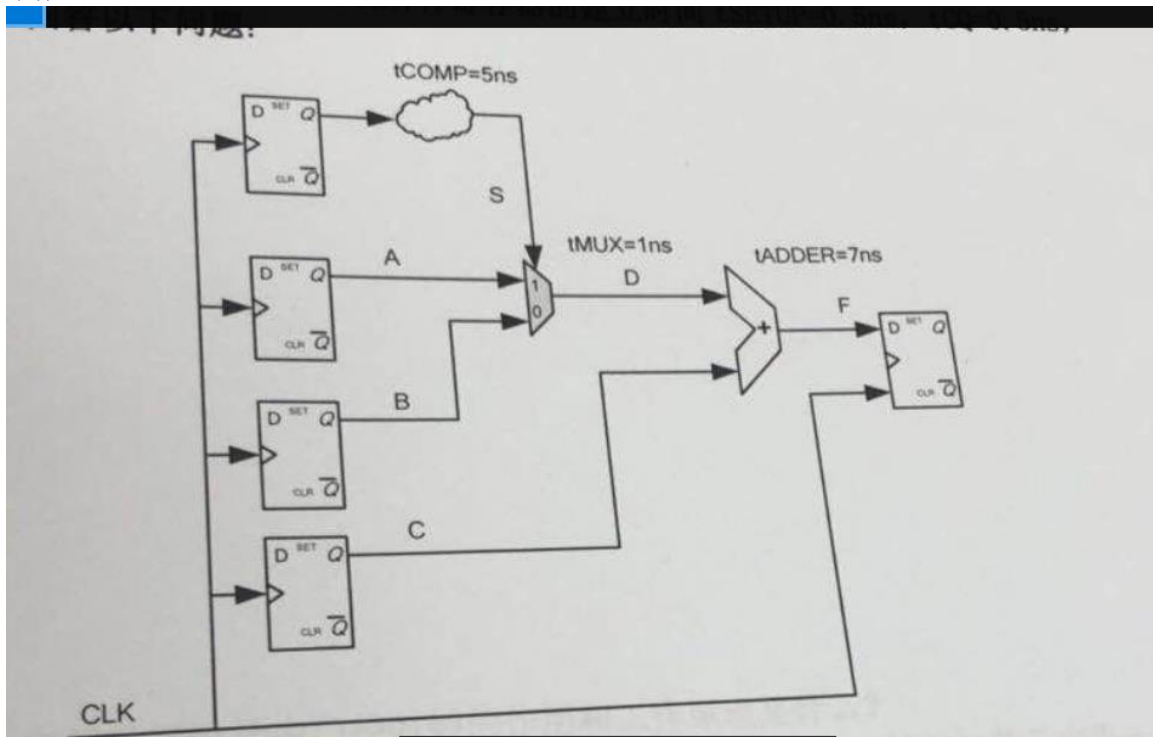
$F_{max}$   
 =  $1/T_{cycle\_min}$   
 = 1/6.9 (HZ)

3-16. CLKA 和 CLKB 是两个异步时钟，请说明以下同步电路存在的问题并给出修改方案。



组合逻辑的输出有可能存在 glitch。送到另一个 clock domain 的信号应该是 DFF 的输出。

3-17. 有以下数字电路，假设所有寄存器的建立时间  $t_{\text{SETUP}} = 0.5\text{ns}$ ,  $t_{\text{CQ}} = 0.5\text{ns}$ , 回答一下问题：



- 不考虑时钟抖动和偏差，写出电路的最小时钟周期  $t_{\text{MIN}} = \underline{\hspace{2cm}} \text{ns}$ ; (需写出运算表达式和最终时间)
- 不考虑时钟抖动和偏差，也不考虑面积因素，从设计角度给出一种优化方案，使得  $t_{\text{MIN}} < 10\text{ns}$ , 画出原理框图：

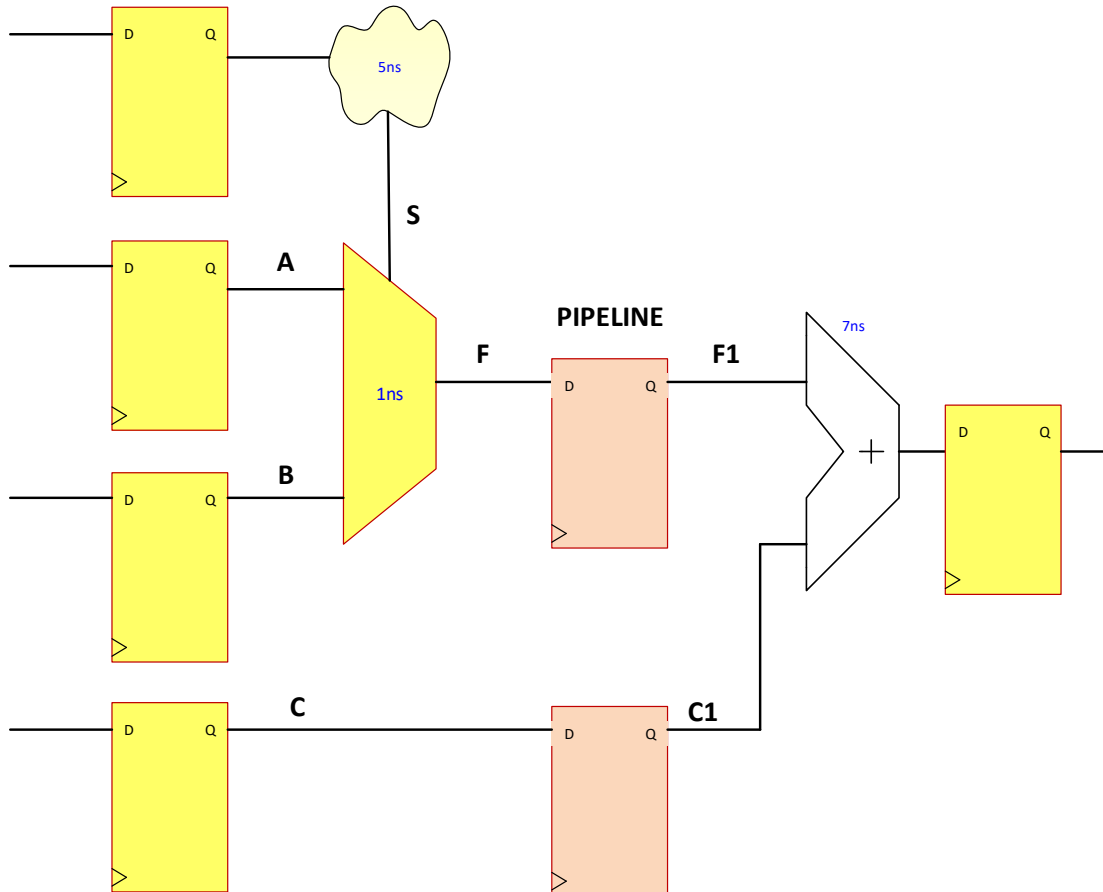
根据如下公式计算最小周期：

$$T_{\text{cycle}} \geq T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} - T_{\text{capture}} + T_{\text{setup}}$$

$T_{\text{cycle\_min}}$

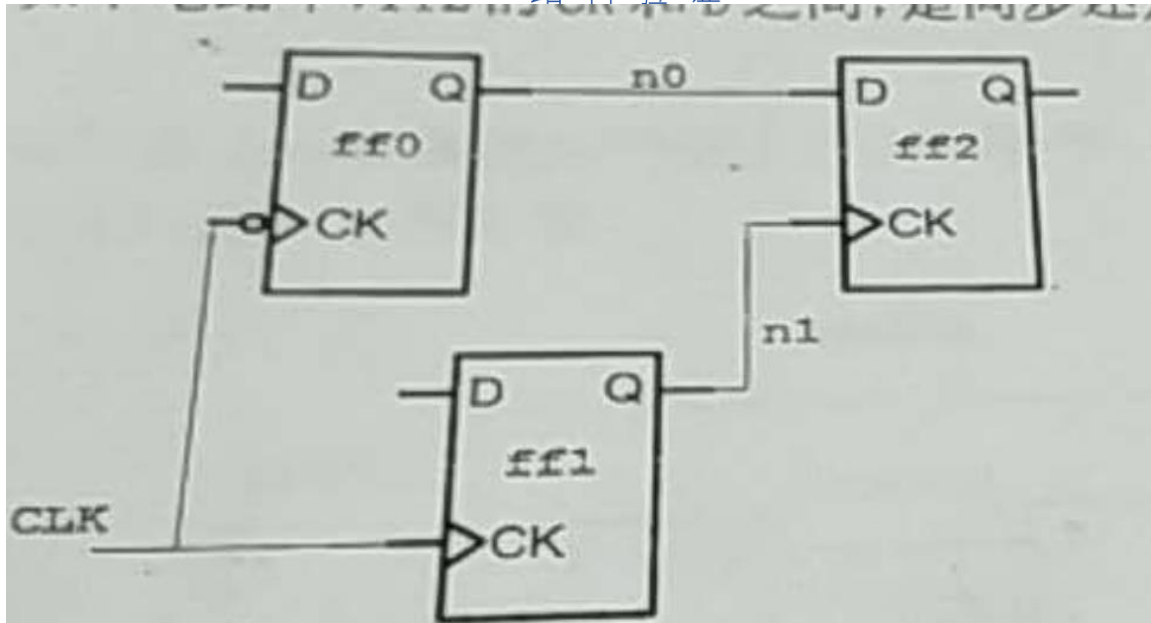
$$\begin{aligned}
 &= T_{\text{launch}} + T_{\text{clk2q}} + T_{\text{datapath}} - T_{\text{capture}} + T_{\text{setup}} \\
 &= 0 + 0.5 + (5 + 1 + 7) - 0 + 0.5 \\
 &= 14 \text{ ns}
 \end{aligned}$$

S→D→F 路径是最长路径，称之为 critical path。  
插入 pipeline 寄存器，打断组合逻辑。



3-18. 如下电路中，ff2 的 CK 和 D 之间，是同步还是异步关系？该电路是否有时序风险？为什么？





该电路中有三个 DFF，一个 CLK 源。

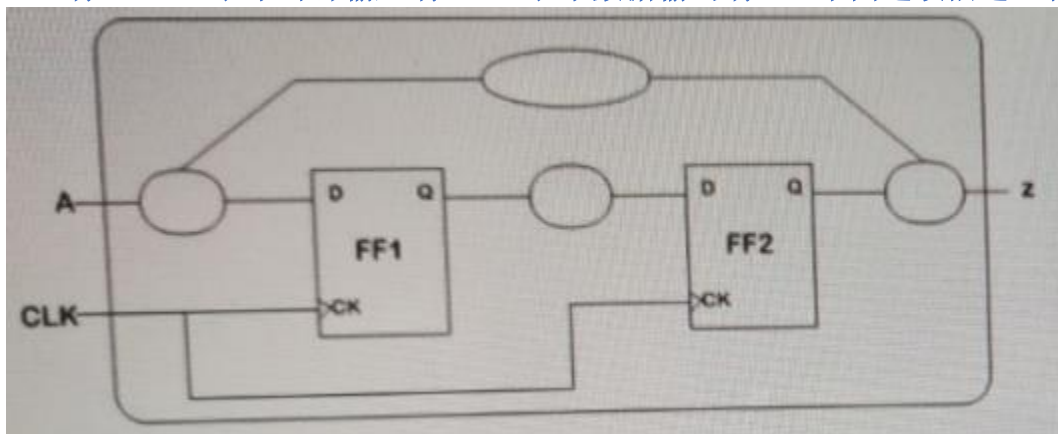
FF0.CK 来自 CLK 的取反；

FF1.CK 来自 CLK；

FF2.CK 来自 FF1.Q。属于 CLK 的分频，可以定义为 CLK 的 generated-clk。

假设数据 D 端也同样来自 CLK，那么它们三个 DFF 之间的时序路径（timing arc）依然可以由 STA 工具保证。

3-19. 如图所示路径示意图，椭圆表示组合逻辑，FF1/FF2 表示寄存器，A 表示数据输入端口，CLK 表示时钟输入端口，Z 表示数据输出端口，下面选项描述正确的是



- a) 只要在端口 Z 上设置输出延时，即可约束 FF2→Z 之间的组合逻辑的延时
- b) 只要在端口 CLK 上创建时钟，即可约束 FF1→FF2 之间的组合逻辑的延时
- c) 只要在端口 CLK 上创建时钟，即可约束 A→Z 之间的组合逻辑的延时

- d) 只要在端口 A 上设置输入延时，即可约束 A->FF1 之间的组合逻辑的延时
- e) Clock jitter 对建立时间和保持时间的影响？

解读：

如果要约束选项 a, c, d 三种路径，除了设置内外延时，还要设置 virtual clock。  
所以只有 b 是对的。

参考视频解读理解 clock jitter 的影响。



路科芯院

---

## PART FOUR: RKV INTRODUCTION

---

## 课程介绍

- 最完善的培训体系：入门+实战，你想要的路科全都有！
- 芯片验证 V2 系列课程-从零基础到实战就业

腾讯课堂报名链接：<https://ke.qq.com/course/323495?tuin=49499911>

课程分为四大部分：

1. 验证系统理论。包括验证的周期、策略、方法、计划、管理以及平台结构。
2. SystemVerilog 验证。结合 SV 核心和 MCDF 设计，从零构建基于 SV 的验证平台。
3. UVM 入门和进阶，在 SV 验证基础上，进一步学习 UVM，再构建 UVM 验证平台。
4. UVM 项目实战。结合理论和 UVM 基础，真正模拟项目周期，完成 RTL 全流程。

- 芯片验证 V1 系列课程-小白轻松入门

腾讯课堂报名链接：<https://ke.qq.com/course/375333?tuin=49499911>

课程大纲：

SystemVerilog 验证：数据类型、验证环境与类、组建的构造及通信、覆盖率

UVM 入门：UVM 核心特性、验证环境迁移、序列的发送和组织

## 路科验证 V2 系列课程的课程设置？

通向专业化验证的三部曲：

模块一：系统验证及 SV 验证（适合零基础： 利用验证系统理论和 SystemVerilog 从零构建基于 SV 的验证平台）

课程结构	教学方式
【通识】芯片开发概述、职业前景、验证任务和目标、验证周期	讲授
【SV】数据类型、过程语句、设计连接和验证结构	讲授
【SV】接口、采样、驱动、测试的始终、调试方法	讲授
【SV】随机的约束、分布、控制，随机数组，随即句柄	讲授
【SV】类的继承、方法、对象，包的使用	讲授
【通识】验证的计划、功能设计描述、内容构成、进程评估	讲授
【通识】验证的管理、检查清单、管理要素、回归测试	讲授
【通识】验证环境的结构和组件（激励、监测、比较）	讲授

【SV】线程的控制、线程间的同步（事件、旗语、信箱）	讲授
【SV】代码覆盖率、功能覆盖率（组、仓、采样、分析）	讲授
【SV】类型转换、虚方法、对象复制、回调函数、参数化的类	讲授
【通识】验证方法（仿真、形式、模拟、虚拟、功耗、性能）	讲授
【实验】实验 0 到实验 5，共 6 个实验，每周一个实验验收和答疑	上机

模块二：UVM 入门和进阶（适合具备 SV 基础： 在 SV 验证基础上，进一步学习 UVM，再构建 UVM 的验证平台）

课程结构	教学方式
方法学时代概述、UVM 类库地图、工厂机制	讲授
核心基类、phase 机制、config 机制、消息管理	讲授
组件家族，uvm_driver, uvm_monitor, uvm_sequencer	讲授
uvm_agent, uvm_scoreboard, uvm_env, uvm_test	讲授
TLM 通信，单向、双向及多向通信，通信管道	讲授
TLM2 通信，同步通信元件	讲授
Sequence 和 item, Sequence 和 driver	讲授
Sequence 和 sequence, sequence 的层次化	讲授
UVM 寄存器，寄存器模型的生成	讲授
寄存器模型的常规方法和场景应用	讲授
C-DPI 的接口使用介绍，覆盖率驱动测试探索，SVA 的应用	讲授
SV 与 UVM 的验证环境搭建比对，垂直复用和水平复用探讨	讲授
【实验】实验 0 到实验 5，共 6 个实验，每周一个实验验收和答疑	上机

模块三：UVM 项目实战（适合具备 UVM 基础： 结合系统理论和 UVM 基础，真正模拟项目周期，完成 RTL 全流程）

课程结构	教学方式
改进 MCDF 结构，实现 AHB 及 SRAM 的标准化设计接口	讲授
验证 IP 组件从零开始构建和发布的要素	讲授
原有 MCDF 验证环境的组件更新和复用	讲授
如何实现 UVM 和 C 的测试用例垂直复用	讲授

功能覆盖率和 SVA 在总线 VIP 的应用	讲授
学习寄存器标准化信息结构，实现 UVM 寄存器自动化生成	讲授
UVM 寄存器测试深度应用和寄存器覆盖率的应用	讲授
总线访问的解析和性能在线分析	讲授
【实验】实验 0 到实验 3，共 4 个实验，每周一个实验验收和答疑	上机

想更多的了解路科验证？

- 路科验证官网：<http://rockeric.com/>
- EETOP 路科首页：<http://blog.eetop.cn/space-uid-1561828.html>
- CSDN 路科首页：<http://blog.eetop.cn/space-uid-1561828.html>
- 技术培训或商业合作，请发送邮件至 rocker.ic@vip.163.com
- 路科验证公众号：我们在这里等你！

专注于系统验证思想和前沿验证资讯，  
为 IC 从业人员提供技术食粮。



长按关注 与路科验证一起进步！