FPGA/数字 IC 知识手册

B 站：Rong 晔

目录

一、 FPGA/IC 设计

1. 什么叫 FPGA

FPGA 是一种可以重构电路的芯片，是一种硬件可重构的体系结构。它的英文全称是 Field Programmable Gate Array，中文名是现场可编程门阵列。

通过编程，用户可以随时改变它的应用场景，它可以模拟 CPU、GPU 等硬件的各种并行运算。通过与目标硬件的高速接口互联，FPGA 可以完成目标硬件运行效率比较低的部分，从而在系统层面实现加速。

2. 什么叫数字 IC

IC 就是半导体元件产品的统称，IC 按功能可分为：数字 IC、模拟 IC、微波 IC 及其他 IC。数字 IC 就是传递、加工、处理数字信号的 IC，是近年来应用最广、发展最快的 IC 品种，可分为通用数字 IC 和专用数字 IC。

通用 IC：是指那些用户多、使用领域广泛、标准型的电路，如存储器（DRAM）、微处理器（MPU）及微控制器（MCU）等，反映了数字 IC 的现状和水平。

专用 IC（ASIC）：是指为特定的用户、某种专门或特别的用途而设计的电路。

3. FPGA 设计流程

以 Xilinx Vivado 开发工具为例，主要有以下步骤：
系统规划、RTL 输入、行为仿真、逻辑综合、综合后仿真（可选）、综合后设计分析（时序及资源）、设计实现（包括布局布线及优化）、布线后仿真、板级调试、bitstream 固化。

1. 系统规划：在 FPGA 设计项目开始之前，需要进行系统的功能定义和模块的划分。然后根据任务要求（系统的功能和复杂度），对工作速度和器件本身的资源，成本，以及连线的可布性进行评估，从而选择合适的设计方案和器件类型。
2. RTL 输入：RTL，Register Transfer Level，直译为寄存器转换级，要描述各级寄存器（时序逻辑中的寄存器），以及寄存器之间的信号的是如何转换的（时序逻辑中的组合逻辑）。通俗来讲，RTL 的输入一般为两种，使用硬件描述语言 Verilog HDL/VHDL 进行编写或者原理图输入。原理图就是比较老的做法了，通过门电路的拖拽连接起来设计系统，所以现在基本都是用语言来描述了。
3. 行为仿真/功能仿真：在编译前对用户所设计的电路进行逻辑功能验证，此时是没有任何延迟信息的，仅对初步的功能进行检测。
4. 逻辑综合：综合的含义就是将高级层次的描述转化为低级层次的描述，就目前层次来看，综合优化是指将设计输入编译成基本逻辑单元组成的逻辑连接网表。
5. 综合后仿真（可选）：综合后仿真检查综合结果是否和原设计一致。仿真时，将综合生成的标准延时文件反标注到综合仿真模型中，可以估计门延时带来的影响，但是无法估计线延时，因此和布线后的实际情况有一定的差距。一般的设计可以省略这一步。
6. 综合后设计分析（时序及资源）：综合之后会告诉我们，目前的系统设计消耗了多少 FPGA 的资源，比如，消耗了多少 LUT、RAM、DSP48,等等。我们可以根据这些报告来选择对设计

进行优化。

7. 设计实现（包括布局布线及优化）：利用实现工具把逻辑映射到目标器件结构的资源中。布局将逻辑网表中的硬件原语和底层单元合理的配置到芯片内部的固有硬件结构上，需要在速度最优与面积最优之间做出选择；布线在布局的基础上，利用芯片内部的各种连线资源，合理正确的连接各个元件。

8. 布线后仿真：意思与前面的综合后仿真一致，因为此时已经进行了布局布线，所以在时序中包含的延迟信息更真实。能较好地反映芯片的实际工作情况。

9. 板级调试：产生使用的数据文件（bitstream-比特流文件），然后将编程数据下载到 FPGA 芯片中，测试实际运行结果。最好的方式是外接逻辑分析仪查看，但这样需要占用一些 IO 接口，而且一般人手头没有逻辑 。那比较实用的方向就是使用内嵌式逻辑分析仪 ILA。【详情可见本人视频，开发笔记第二期-开发流程】

10. bitstream 固化：这其实是最最最最后一步了。只有在你确保当前的设计已经完美无瑕可以拿来用的时候，才会把它固化到 FPGA 上。这样，每次上电运行的就变成这个系统了。

4. 对 FPGA 开发的理解

目前 FPGA 的应用主要是三个方向：

第一个方向，也是传统方向主要用于通信设备的高速接口电路设计，这一方向主要是用 FPGA 处理高速接口的协议，并完成高速的数据收发和交换。这类应用通常要求采用具备高速收发接口的 FPGA，同时要求设计者懂得高速接口电路设计和高速数字电路板级设计，具备 EMC/EMI 设计知识，以及较好的模拟电路基础，需要解决在高速收发过程中产生的信号完整性问题。FPGA 最初以及到目前最广的应用就是在通信领域，一方面通信领域需要高速的通信协议处理方式，另一方面通信协议随时在修改，非常不适合做成专门的芯片。因此能够灵活改变功能的 FPGA 就成为首选。到目前为止 FPGA 的一半以上的应用也是在通信行业。

第二个方向，可以称为数字信号处理方向或者数学计算方向，因为很大程度上这一方向已经大大超出了信号处理的范畴。例如早就在 2006 年就听说老美将 FPGA 用于金融数据分析，后来又见到有将 FPGA 用于医学数据分析的案例。在这一方向要求 FPGA 设计者有一定的数学功底，能够理解并改进较为复杂的数学算法，并利用 FPGA 内部的各种资源使之能够变为实际的运算电路。目前真正投入实用的还是在通信领域的无线信号处理、信道编解码以及图像信号处理等领域，其它领域的研究正在开展中，之所以没有大量实用的主要原因还是因为学金融的、学医学的不了解这玩意。不过最近发现欧美有很多电子工程、计算机类的博士转入到金融行业，开展金融信号处理，相信随着转入的人增加，FPGA 在其它领域的数学计算功能会更好的发挥出来，而我也有意做一些这些方面的研究。不过国内学金融的、学医的恐怕连数学都很少用到，就不用说用 FPGA 来帮助他们完成数学_运算了，这个问题只有再议了。

第三个方向就是所谓的 SOPC 方向，其实严格意义上来说这个已经在 FPGA 设计的范畴之内，只不过是利用 FPGA 这个平台搭建的一个嵌入式系统的底层硬件环境，然后设计者主要是在上面进行嵌入式软件开发而已。设计对于 FPGA 本身的设计时相当少的。但如果涉及到需要在 FPGA 做专门的算法加速，实际上需要用到第二个方向的知识，而如果需要设计专用的接口电路则需要用到第一个方向的知识。

就目前 SOPC 方向发展其实远不如第一和第二个方向，其主要原因是因为 SOPC 以 FPGA 为主，或者是在 FPGA 内部的资源实现一个“软”的处理器，或者是在 FPGA 内部嵌入一个处理器核。但大多数的嵌入式设计却是以软件为核心，以现有的硬件发展情况来看，多数情况下的接口都已经标准化，并不需要那么大的 FPGA 逻辑资源去设计太过复杂的接口。而且就目前看来 SOPC 相关的开发工具还非常的不完善，以 ARM 为代表的各类嵌入式处理器开发

工具却早已深入人心，大多数以 ARM 为核心的 SOC 芯片提供了大多数标准的接口，大量成系列的单片机/嵌入式处理器提供了相关行业所需要的硬件加速电路，需要专门定制硬件场合确实很少。通常是在一些特种行业才会在这方面有非常迫切的需求。目前 Xilinx 已经将 ARMcortex- A9 的硬核嵌入到 FPGA 里面，未来对嵌入式的发展有很大推动，不过，不要忘了很多老掉牙的 8 位单片机还在嵌入式领域混呢，嵌入式主要不是靠硬件的差异而更多的是靠软件的差异来体现价值的。

5. FPGA 内部资源

目前主流 FPGA 都采用了 SRAM 工艺的查找表(LUT)结构，LUT 本质上就是一个 RAM。FPAG 内部组成部分主要有：可编程输入/输出块(IOB)、可配置逻辑块(CLB)、嵌入式块 RAM(BRAM)、丰富的布线资源、底层内嵌功能资源、内嵌专用硬核资源等。

(1)、可编程输入/输出块：为了便于管理和适应多种电器标准，FPGA 的 IOB 被划分为若干个组 (bank)，每个 bank 的接口标准由其接口电压 VCCO 决定，一个 bank 只能有一种 VCCO，但不同 bank 的 VCCO 可以不同。只有相同电气标准的端口才能连接在一起，VCCO 电压相同是接口标准的基本条件。

(2)、可配置逻辑块：由查找表和可编程寄存器组成，查找表完成纯组合逻辑功能，内部寄存器可配置成触发器或锁存器。

(3)、嵌入式块 RAM：可以配置成单端口 RAM、双端口 RAM、内容地址存储器(CAM)以及 FIFO 等常用存储结构。

(4)、丰富的布线资源：布线资源连通 FPGA 内部的所有单元，而连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。主要分为四类：全局布线资源、长线资源、短线资源、分布式布线资源。

(5)、底层内嵌的功能单元：主要包括 DLL、PLL、DSP、CPU 等，现在越来越丰富的内嵌的功能单元，使得 FPGA 成为了系统级的设计工具，使其具备了软硬件联合设计的能力，逐步向 SOC 平台过渡。

(6)、内嵌专用硬核资源：内嵌的专用硬核是相对底层软核而言的，指 FPGA 处理能力强大的硬核，等效于 ASIC 电路。主要有乘法器、串并收发器、PCI-E、以太网控制器等。

6. IC 设计流程

1) 确定项目需求

首先做一款芯片需要有市场，一般公司会先做市场调研，比如最近市面上比较火的人工智能芯片，物联网芯片，5G 芯片，需求量都比较大。有了市场的需求我们就可以设计芯片的 spec 了。先由架构工程师来设计架构，确定芯片的功能，然后用算法进行模拟仿真，最后得出一个可行的芯片设计方案。

有了芯片的 spec,下一步就可以做 RTL coding 了。

2) 前端设计

架构/算法设计分析

目的：完成芯片中数字部分的高层次算法或架构的分析与建模，为硬件提供一个正确的软件

功能模型，更为重要的是，通过大量的高层次仿真和调试，为 RTL 实现提供总体性的设计指导。数字部分越复杂，这一点越重要。

工具：MATLAB、C++、C、System C、System Verilog 等。不同类型的芯片都不同的选择，如数字信号处理类芯片，偏好 MATLAB。

特点：这部分工作至关重要，基本上奠定了整个芯片的性能和功耗的基础。这部分工作主要由具有通信、信号处理、计算机、软件专业背景的工程师完成，也有很多微电子专业背景的工程师参与。

3) RTL 实现

目的：依据第一步的结果，完成由高层次描述到 Verilog HDL 实现的过程。

工具：GVim/Emac、Verilog/VHDL

特点：这一步能明显区别中训练有素的工程师和初学者。前者在写代码的过程中，具有极强的大局观，能够在书写 Verilog HDL、描述逻辑功能的同时，还能够兼顾逻辑综合、STA、P&R、DFX、功耗分析等多方面因素，最终提供一份另其他环节的工程师都赏心悦目的代码。初学者则处处留地雷，一不小心就引爆。

Coding Style Check

目的：排除 RTL 代码中 CDC(Clock Domain Cross)、Lint 等问题。

CDC:跨时钟域检查

LInt：代码潜在问题检查（例如 A+B 的进位溢出）

工具：Syglass、LEDA、0inCDC

特点：目前大部分芯片中的数字部分基本上都采用局部同步和全局异步的设计策略，因此，在设计中需要小心注意跨时钟域的数据同步问题。

输入：RTL，SDC，lib/sglib

输出：wave file，report

4) 功能验证

目的：在无延迟的理想情况在，通过大量的仿真，发现电路设计过程中的人为或者非人为引起的 bug。主要指标是功能覆盖率。

工具：Modelsim、VCS、NC-Verilog、（DVE/Verdi 波形查看器）

语言：C++、C、System C、System Verilog，基于 UVM 的方法学等。主要是 System Verilog，一般哪个方便用哪个。

特点：验证工程师近年来已经成为 IC 设计中需求量最大的岗位。这个阶段会占用大量的时间，数以月计。

5) 逻辑综合+DFT

目的：将 RTL 代码映射为与工艺库相关的网表。

工具：DesignCompiler、RTL Compiler。DesignCompiler 在市场中占有垄断性地位，几乎成为逻辑综合的标准。

特点：

a.从芯片生产的角度来看，在该步骤之前，所有的工作都可近似看做一个虚拟性的，与现实无关。而从逻辑综合起，后续所有的工作都将与工艺的物理特性、电特性等息息相关。逻辑综合工具的功能主要是将 VerilogHDL 格式的文本映射为网表格式的文本，因此，它的功能等同于文本编译器。那么转换的方式有很多种，工具如何选取呢？逻辑综合过程中，整个文本格式的编译过程是在给定的人为约束条件下进行的，通过这些约束和设定的目标来指导工具完成 Compiler 的工作。所以，逻辑综合过程可以看成一个多目标（频率、面积、功耗）多约束的工程优化问题。

b.该步骤中，通常会插入 DFT、clock gating 等。

c.该步骤中通常加入 Memory、各种 IP 等。为了在各种工艺库以及 FPGA 原型验证平台之间有一个更方便的移植，注意适当处理这些 Memory、IP 等的接口。该步骤中也可加入 I/O、PLL 等。

DFT(Design For Test): 为了保证芯片内部的制造缺陷尽量能够哦被检测到，通过在电路中插入扫描链（Scan Chain）的方式，测试 IC 在生产制造过程中是否出现问题。加入 DFT 会增加 20%~30%的面积。但为了保证良率，为了给客户的片子是合格的需要将具有缺陷（工厂制造缺陷）的片子筛除。

输入文件：

a.RTL 代码：由 ASIC design engineers 团队提供；交接前，必须保证在第 3 步的 check 中没有任何问题；

b.工艺库（.db）：由晶圆厂提供；

c.约束（SDC）：由逻辑综合工程师和 ASIC design engineers 共同商定。

输出文件：

a.网表：包含了 RTL 中的所有的逻辑信息，除此以外，可能还会有 DFT、clock gating、I/O 等；网表主要用于 P&R 等流程；

b.标准延迟文件 SDF：主要包含了网表中所有器件的延迟信息，用于时序仿真；PT 会结合后端工具生成的一个更为精确的 sdf ，所以，通常会用 PT 的 sdf 文件做后仿真。

Project 文件：.ddc;

c.各种报告：timing report、area report、constrain report、clock report、violation report 等等，以及工具的 log 文件。（此处最好能够熟悉各种脚本语言，将各种 report 处理为友好易读形式）

6) 形式验证

目的：RTL 代码和逻辑综合后的网表是否具有一致的功能。（尤其是在后端做 ECO 的时候）

工具：Formality、Conformal

输入文件：RTL 代码、netlist（逻辑综合输出）、约束

输出文件：match 和 verify 报告。

7) 静态时序分析

目的：分析设计中所有的路径，确保满足内部时序单元对建立时间和保持时间的要求。

工具：PrimeTime、Tempus。PrimeTime 在市场中占有垄断性地位，几乎成为 STA 的标准

特点：

输入：spef 文件、PnR 网表，标准单元库 db 文件，IP 的 dp 文件

输出：SDF，timing ECO 文件

特点：

a.从逻辑综合开始，基本上每做一步大的调整，都会完成一次 STA 分析，以保证每步都能实现时序收敛。鉴于该特性非常重要，PrimeTime 成为了 Signoff 的重要工具。

b.所用到的 SDC 同逻辑综合；

c.通常设计中会存在大量的违例路径，STA 要修大量的 setup、hold 等，如何修这些违例，可以体现工作经验的重要性。此外，如果是前端修 timing 违例，一般会修的很快，但是会带来一个重大的问题，代码被前端修改后是否存在新的 bug，还需重新仿真确认，仿真会消耗掉数以月计的时间，所以除非万不得已，不会找前端修 timing。

注：静态时序分析和形式验证这属于验证范畴。

STA 主要是在时序上对电路进行验证，检查电路是否存在建立时间（setup time）和保持时间（hold time）的违例（violation）。而形式验证是从功能上对综合后的网表进行验证。常用的就是等价性检查 (LEC) 方法，以功能验证后的 HDL 设计为参考，对比综合后的网表功能，

他们是否在功能上存在等价性。
静态时序分析和形式验证出现在设计流程中前后端过程，有一些地方在前端中没有提到，应该是在后端当中比较重要，作为验证设计工作的一部分，在前端中也加入静态时序分析和形式验证可以提高设计的可靠性。

3.后端设计

下图给出了后端设计的流程及主要工作。
Place & Route 一般由后端工程师来做，Physical Design Engineer.
后端里 DRC 就是要检查设计规则是否符合芯片制造商的要求，这样才能正确的生产芯片。
后端完成工作后，最终会生成 GDSII 格式的文件，交由芯片制造商流片。

7. 对数字 IC 设计的理解

Verilog HDL 和 VHDL 是世界上最流行的两种硬件描述语言，都是在 20 世纪 80 年代中期开发出来的。前者由 Gateway Design Automation 公司（该公司于 1989 年被 Cadence 公司收购）开发。两种 HDL 均为 IEEE 标准。
是的，20 世纪 80 年代，这个语言距今已经 40 年，期间的修修改改并没有出现翻天覆地的大改动，也没有被时间和市场抛弃。原因很简单，在现有工艺没有突破性创新并推广之前，对以晶体管为基础的数字电路设计描述方法并不会变。
其次，数字电路就是一个由严密的逻辑组成的巨大建筑，经验的累积非常重要。小到一个加法器能不能加，加上去会不会又 timing 问题，performance bug 如何去修正,各种逻辑描述的优劣，对各种 EDA 的掌握和熟悉，如何才能写出物理实现最平衡的电路.....更别说大到各种各样的算法建模前的研讨，复杂算法的 RTL 实现，对新的系统的预研，如此种种不一而足。恐怕不是上上培训班然后教你个 XXX 天学会 XXX 就能教给你的，不在完全匹配的岗位上磨练十年，或许连我列出的这些都学不完。
最重要的是，没人敢把较高风险的系统交给有"青春"但是没有经验的人去做，一次流片，特别是高端工艺失败的代价可以顷刻让小公司破产，是的，驱动这个社会的是资本，资本是逐利的，但是资本也是最聪明的，他清楚应该选谁。十年之后，你大约 35 岁的日子，别人的“中年危机”正是你在 IC 这个领域成熟的时间段，可以说是风华正茂。可以在技术上继续前进，或者以经验来带领团队，可以抽身出来完成创业。
客观的说，ICer 并不是严格意义上的程序员，我们是数字电路的设计者和维护者，我们的每一行语言都会转换成物理意义上真实存在的与或非门，被光刻机印刻在芯片的最深处，成为所有现代工业物品中最核心的“大脑”，为它们赋予智慧。

8. 查表法 LUT

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，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。

9. 可编程逻辑块 CLB

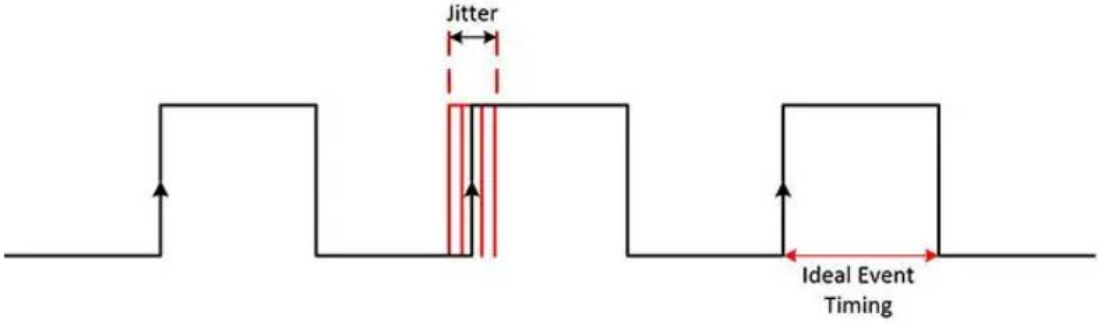
CLB 是 FPGA 内的基本逻辑单元。CLB 的实际数量和特性会依器件的不同而不同，但是每个 CLB 都包含一个可配置开关矩阵，此矩阵由 4 或 6 个输入、一些 选型电路（多路复用器等）

和触发器组成。 开关矩阵是高度灵活的，可以对其进行配置以便处理组合逻辑、移位寄存器或 RAM。在 Xilinx 公司的 FPGA 器件中，CLB 由多个（一般为 4 个或 2 个） 相同的 Slice 和附加逻辑构成。每个 CLB 模块不仅可以用于实现组合逻辑、时序逻辑，还可以配置为分布式 RAM 和分布式 ROM。

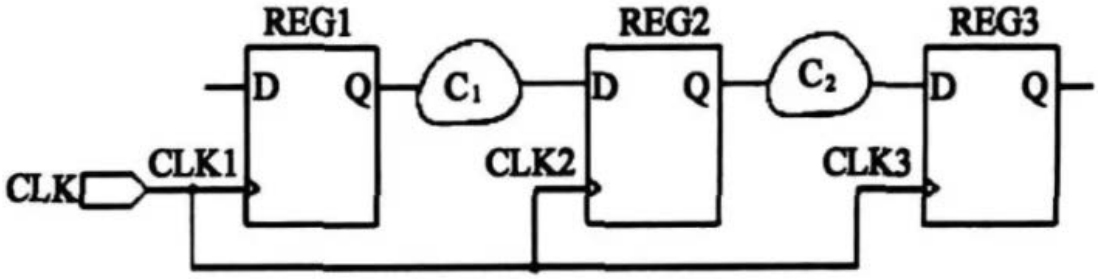
Slice 是 Xilinx 公司定义的基本逻辑单位，其内部结构如图 1-4 所示，一个 Slice 由两个 4 输入的函数、进位 逻辑、算术逻辑、存储逻辑和函数复用器组成。算术逻辑包括一个异或门（XORG）和一个专用与门（MULTAND），一个异或门可以使一个 Slice 实现 2bit 全加操作，专用与门用于提高乘法器的效率；进位逻辑由专用进位信号和函数复用器（MUXC）组成，用于实现快速的算术加减法操作；4 输入函数发生 器用于实现 4 输入 LUT、分布式 RAM 或 16 比特移位寄存器（Virtex-5 系列芯片的 Slice 中的两个输入函数为 6 输入，可以实现 6 输入 LUT 或 64 比特移位寄存器）；进位逻辑包括两条快速进位链，用于提高 CLB 模块的处理速度

10. 时间抖动 jitter/偏移 skew

jitter:由于晶振本身稳定性，电源以及温度变化等原因造成了时钟频率的变化，就是 jitter,指的是时钟周期的变化，也就是说**时钟周期在不同的周期上可能加长或缩短。**它是一个平均值为 0 的平均变量。指两个时钟周期之间存在的差值，这个误差是在时钟发生器内部产生的，和晶振或者 PLL 内部电路有关，布线对其没有影响。由于跟晶振本身的工艺有关，所以在设计中无法避免它能带来的影响，通常只能在设计中留有一定的余量。



skew:是指同样的时钟产生的多个子时钟信号之间的延时差异。**skew 通常是时钟相位上的不确定。**由于时钟源到达不同寄存器所经历路径的驱动和负载的不同，时钟边沿的位置有所差异，因此就带来了 skew。完成布局布线后，物理路径延时是固定的，所以在设计中考虑到时钟偏移，就可以避免偏移带来的影响。



11. 毛刺 glitch

在 FPGA 的设计中，毛刺现象是长期困扰电子设计工程师的设计问题之一，是影响工程师设计效率和数字系统设计有效性和可靠性的主要因素。**由于信号在 FPGA 的内部走线和通过逻辑单元时造成的延迟,在多路信号变化的瞬间，组合逻辑的输出常常产生一些小的尖峰**，即毛刺信号，这是由 FPGA 内部结构特性决定的。**毛刺现象在 FPGA 的设计中是不可避免的，**

有时任何一点毛刺就可以导致系统出错,尤其是对尖峰脉冲或脉冲边沿敏感的电路更是如此。

1 利用冗余项法

利用冗余项消除毛刺有 2 种方法：代数法和卡诺图法，两者都是通过增加冗余项来消除险象，只是前者针对于函数表达式而后者针对于真值表。以卡诺图为例，若两个卡诺圆相切，其对应的电路就可能产生险象。因此，修改卡诺图，在卡诺图的两圆相切处增加一个圆，以增加多余项来消除逻辑冒险。但该法对于计数器型产生的毛刺是无法消除的。

2 采样法

由于冒险多出现在信号发生电平跳变的时刻，即在输出信号的建立时间内会产生毛刺，而在保持时间内不会出现，因此，在输出信号的保持时间内对其进行采样，就可以消除毛刺信号的影响，常用的采样方法有 2 种:一种使用一定宽度的高电平脉冲与输出相与，从而避开了毛刺信号，取得输出信号的电平值。这种方法必须保证采样信号在合适的时间产生，并且只适用于对输出信号时序和脉冲宽度要求不严的情况。另一种更常见的方法叫锁存法，是利用 D 触发器的输入端 D 对毛刺信号不敏感的特点，在输出信号的保持时间内，用触发器读取组合逻辑的输出信号。由于在时钟的上升沿时刻，输出端 Q=D，当输入的信号有毛刺时，只要不发生在时钟的上升沿时刻，输出就不会有毛刺。这种方法类似于将异步电路转化为同步电路，实现简单，但同样会涉及到时序问题。

3 吸收法

由于产生的毛刺实际上是高频窄脉冲，故增加输出滤波，在输出端接上小电容 C 就可以滤除毛刺。但输出波形的前后沿将变坏，在对波形要求较严格时，应再加整形电路，该方法不宜在中间级使用。

4 延迟法

因为毛刺最终是由于延迟造成的，所以可以找出产生延迟的支路。对于相对延迟小的支路，加上毛刺宽度的延迟可以消除毛刺。但有时随着负载增加，毛刺会继续出现，而且，当温度变化，所加的电压变化或要增加逻辑门时，所加的延迟是不同的，必须重新设计延迟线，因而这种方法也是有局限性的。而且采用延迟线的方法产生延迟会由于环境温度的变化而使系统可靠性变差。

5 硬件描述语言法

这种方法是从硬件描述语言入手，找出毛刺产生的根本原因，改变语言设计，产生满足要求的功能模块，来代替原来的逻辑功能块。一个 3 位计数器可能会在 011 到 100 和 101 到 110 发生跳变时产生毛刺，究其原因是因为一次有 2 位发生跳变，可以采用 VHDL 语言对计数器编写如下，产生的计数模块代替原来普通的计数器。

12. 锁存器/触发器

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。

一般将信号经过两级触发器就可以消除毛刺。

为什么锁存器不好？

1) 锁存器对毛刺不敏感，很容易在信号上产生毛刺；

2) 而且也没有时钟信号，不容易进行静态时序分析。

正是因为这两个原因，我们在 FPGA 设计时，尽量不用锁存器。

注意，“FPGA 中只有 LUT 和 FF 的资源，没有现成的 Latch，所以如果要用 Latch，需要更多的资源来搭出来。”这个观点不完全是正确的。

在 Xilinx 的 FPGA 中，6 系列之前的器件中都有 Latch；6 系列和 7 系列的 FPGA 中，一个 Slice 中有 50%的 storage element 可以被配置为 Latch 或者 Flip-Flop，另外一半只能被配置为 Flip-Flop。比如 7 系列 FPGA 中，一个 Slice 中有 8 个 Flip-Flop，如果被配置成了 Latch，该 Slice 的另外 4 个 Flip-Flop 就不能用了。这样确实造成了资源的浪费。

但在在 UltraScale 的 FPGA 中，所有的 storage element 都可以被配置成 Flip-Flop 和 Latch。

13. D 触发器

D 触发器是一个具有记忆功能的，具有两个稳定状态的信息存储器件，是构成多种时序电路的最基本逻辑单元，也是数字逻辑电路中一种重要的单元电路。

因此，D 触发器在数字系统和计算机中有着广泛的应用。触发器具有两个稳定状态，即"0"和"1"，在一定的外界信号作用下，可以从一个稳定状态翻转到另一个稳定状态。

D 触发器有集成触发器和门电路组成的触发器。触发方式有电平触发和边沿触发两种，前者在 CP(时钟脉冲)=1 时即可触发，后者多在 CP 的前沿（正跳变 0→1）触发。

D 触发器的次态取决于触发前 D 端的状态，即次态=D。因此，它具有置 0、置 1 两种功能。

对于边沿 D 触发器，由于在 CP=1 期间电路具有维持阻塞作用，所以在 CP=1 期间，D 端的数据状态变化，不会影响触发器的输出状态。

D 触发器应用很广，可用做数字信号的寄存，移位寄存，分频和波形发生器等等。

代码实现：

```
module d_flip_flop(d,clk,q);
    input d;
    input clk;
    output q;
    reg q;
    always @ (posedge clk)
    begin
        q <= d;//上升沿有效的时候，把 d 捕获到 q
    end
endmodule
```

14. 常见触发器

(1) 按逻辑功能：

1. RS 触发器/SR 触发器

方程：Q*=S'|R'Q

1) S=0 R=0，Q 保持不变

2) S=1 R=0，Q*=1

3) S=0 R=1，Q*=0

4) S=1 R=1，Q*=1，当 S 与 R 归零时状态不定，若 R 先归零，则与②情况相同，置位优先，称为 RS；若 S 先归零，则与③情况相同，复位优先，称为 SR.

2. D 触发器
方程：Q*=D
1) D=0，Q*=0
2) D=1，Q*=1

3. JK 触发器
方程：Q*=JQ'+K'Q
1) J=0 K=0，Q 保持不变
2) J=1 K=0，Q*=1
3) J=0 K=1，Q*=0
4) J=1 K=1，Q 反转

4. T 触发器
方程：Q*=TQ'+T'Q
1) T=0，Q 保持不变
2) T=1，Q 反转

(2) 按触发方式：
1.电平触发器
时钟电平触发的触发器。以时钟脉冲作为控制信号 CLK 来控制，在触发状态下根据信号的变化而变化。这个信号可以是 SR/RS，T，D，JK 等逻辑功能对应的方式输入。

1. 边沿触发器
接收时钟信号 CLK 的某一约定跳变(上升沿或下降沿)来到时的输入数据。在 CLK=1 及 CLK=0 期间以及未跳变时，触发器不接收数据。

2. 主从/脉冲触发器
为克服电平触发的 SR 触发器在一个 CLK 周期内输出状态可能发生多次反转的缺点，用时钟脉冲触发的触发器。简成脉冲触发的触发器，其结构为主从结构。
若 CLK 高有效，则下降沿触发；若 CLK 低有效，则上升沿触发。

(3) 按电路结构：
1.基本 RS 触发器
2.CLK 钟控触发器

(4) 按存储数据：
1.静态触发器
2.动态触发器

15. 组合逻辑/时序逻辑

1 组合逻辑：

组合逻辑的特点是任意时刻的输出仅仅取决于该时刻的输入，与电路原本的状态无关，逻辑中不牵涉跳变沿信号的处理，组合逻辑的 verilog 描述方式有两种：

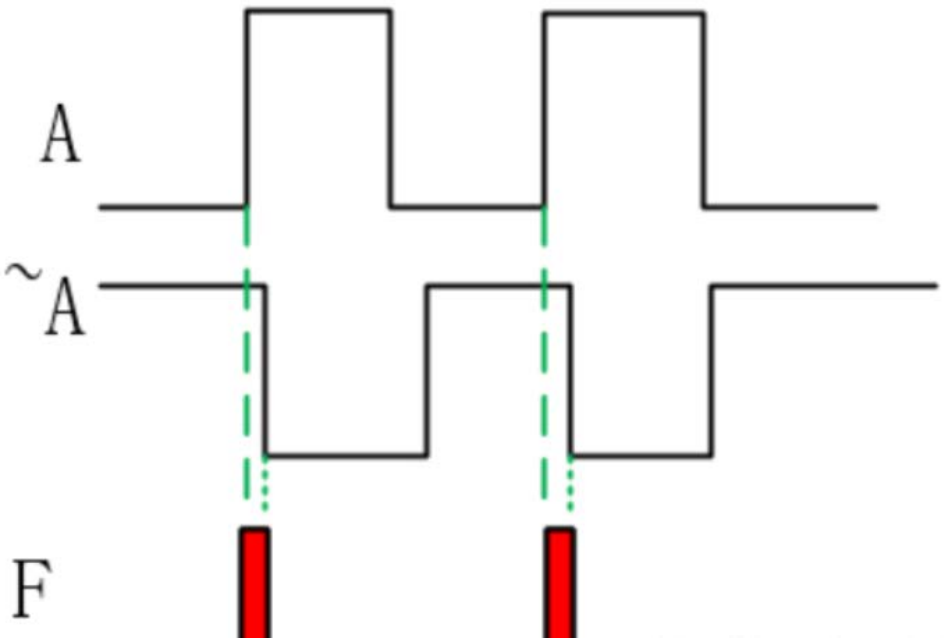
（1）always @（电平敏感信号列表）
always 模块的敏感列表为所有判断条件信号和输入信号，但一定要注意敏感列表的完整性。在 always 模块中可以使用 if、case 和 for 等各种 RTL 关键字结构。由于赋值语句有阻塞赋值和非阻塞赋值两类，建议读者使用阻塞赋值语句“=”。always 模块中的信号必须定义为 reg 型，不过最终的实现结果中并没有寄存器。这是由于在组合逻辑电路描述中，将信号定义为 reg 型，只是为了满足语法要求。
（2）assign 描述的赋值语句。
信号只能被定义为 wire 型。

2 时序逻辑：
时序逻辑是 Verilog HDL 设计中另一类重要应用，**其特点为任意时刻的输出不仅取决于该时刻的输入，而且还和电路原来的状态有关。**电路里面有存储元件（各类触发器，在 FPGA 芯片结构中只有 D 触发器）用于记忆信息，从电路行为上讲，不管输入如何变化，仅当时钟的沿（上升沿或下降沿）到达时，才有可能使输出发生变化。

与组合逻辑不同的是：
（1）在描述时序电路的 always 块中的 reg 型信号都会被综合成寄存器，这是和组合逻辑电路所不同的。
（2）时序逻辑中推荐使用非阻塞赋值“<=”。
（3）时序逻辑的敏感信号列表只需要加入所用的时钟触发沿即可，其余所有的输入和条件判断信号都不用加入，这是因为时序逻辑是通过时钟信号的跳变沿来控制的。

16. 竞争/冒险

信号经过逻辑门电路都需要一定的时间，由于不同路径上门的级数不同，信号经过不同路径传输的时间不同，或者门的级数相同但各个门延迟时间有差异，也会造成传输时间不同。



竞争与冒险是逻辑门因输入端的竞争而导致输出产生不应有的尖峰干扰脉冲（又称过渡干扰脉冲）的现象。在门电路中，两个输入信号同时向两个相反方向的逻辑状态转换，即一个从低电平变为高电平，一个从高电平变为低电平，或反之，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冒险。竞争不一定会产生冒险，但冒险就一定有竞争。

消除竞争冒险的方法常见有 4 种:
1）修改逻辑设计，这主要包括去除互补逻辑变量和增加冗余项。
2）输出端并联电容，这主要利用了电容的充放电特性，对毛刺滤波，对窄脉冲起到平波的作用。
3）利用格雷码每次只有一位跳变，消除了竞争冒险产生的条件。
4）利用 D 触发器对毛刺不敏感的特性。

在 Verilog 编程时，需要注意以下几方面，在绝大多数情况下可避免综合后仿真出现冒险问题。
1）时序电路建模时，用非阻塞赋值。
2）锁存器电路建模时，用非阻塞赋值。
3）用 always 和组合逻辑建模时，用阻塞赋值。
4）在同一个 always 块中建立时序和组合逻辑模型时，用非阻塞赋值。
5）在同一个 always 块中不要既使用阻塞赋值又使用非阻塞赋值。
6）不要在多个 always 块中为同一个变量赋值。

17. 建立时间/保持时间

概念:
建立时间 Tsu：触发器在时钟上升沿到来之前，其数据输入端的数据必须保持不变的时间。
建立时间决定了该触发器之间的组合逻辑的最大延迟。
保持时间 Th：触发器在时钟上升沿到来之后，其数据输入端的数据必须保持不变的时间。
保持时间决定了该触发器之间的组合逻辑的最小延迟。

为什么要满足建立时间和保持时间：
因为触发器内部数据的形成是需要一定的时间的。
如果不满足建立和保持时间，触发器将进入亚稳态，进入亚稳态后触发器的输出将不稳定，在 0 和 1 之间变化，这时需要经过一个恢复时间，其输出才能稳定，但稳定后的值并不一定是你的输入值。

需要建立时间是因为：
假设 Clock 的到达时刻为传输门 A 关闭、传输们 B 打开的时刻。如果 Data 没有在这之前足够早的时刻到达，那么很有可能内部的 feedback 线路上的电压还没有达到足够使得 inv1 翻转的地步（因为 inv0 有延时，Data 有 slope，传输门 B 打开后原来的 Q 值将通过 inv2 迫使 feedback 保持原来的值）。如果这种竞争的情况发生，Q 的旧值将有可能获胜，使 Q 不能够寄存住正确的 Data 值；当然如果 feedback 上的电压已经达到了足够大的程度也有可能在竞争中取胜，使得 Q 能够正确输出。

如果 inv0、inv1 和 inv2 的延时较大（Data 的变化影响 feedback 和 Q 的时间越长），那么为了保证正确性就需要更大的 setuptime。所以在实际测量 setup time 的时候，需要选取工艺中最慢的 corner 进行仿真测量。

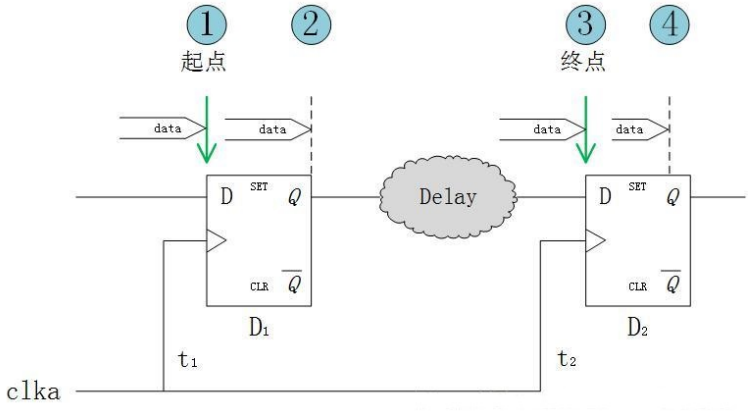
需要保持时间是因为：

和 setuptime 的情况不一样，因为 Clock 到达时刻并不等同于 latch 的传输门 A 完全关闭的时刻。所以如果 Data 没有在 Clock 到达之后保持足够长的时间，那么很有可能在传输门 A 完全关闭之前 Data 就已经变化了，并且引起了 feedback 的变化。如果这种变化足够大、时间足够长的话，很有可能将 feedback 从原本正确的低电压拉到较高电压的电压。甚至如果这种错误足够剧烈，导致了 inv1 和 inv2 组成的 keeper 发生了翻转，从而彻底改变了 Q 的正确值，就会导致输出不正确。当然，如果这种错误电压不是足够大到能够改变 keeper 的值，就不会影响到 Q 的正确输出。

如果 inv0、inv1 和 inv2 的延时较小（Data 的变化影响 feedback 和 Q 的时间越短），那么为了保证正确性，就需要更大的 holdtime。所以在实际测量 holdtime 的时候，需要选取工艺中最快的 corner 进行仿真测量。

模型分析：

理解建立时间保持时间需要一个模型，如图所示。起点是源触发器 D1 的采样时刻，终点是目的触发器 D2 的采样时刻，假设起点已经满足了建立时间和保持时间要求，现在分析终点采样时刻是否同样满足要求。



用到的参数:

1) Tco: 数据正确采样后从 D 端到达 Q 端的延时，触发器固有属性，不可改变
2) TDelay: D1 输出端到 D2 输入端的组合逻辑延时和布线延时
3) Tsu: 触发器的建立时间，触发器固有属性，不可改变
4) Th: 触发器的保持时间，触发器固有属性，不可改变
5) Tclk: 时钟周期
6) t1: 假设源时钟为 clka，clka 到达 D1 的延时
7) t2: 同 t1，是 clka 到达 D2 的延时

触发器的行为：

时钟沿到来时采样数据 D，将采到的数据寄存下来，并输出到 Q 端，所以如果没有新的时钟沿到来，则 Q 端输出的一直是上次采样的数据，每来一个时钟沿，采样一次数据 D。

假设 clk 传输没有延时：

先假设 clk 的传输没有任何延时，则每一个时钟上升沿都会同时到达 D1 和 D2。
时间起点，第一个时钟沿 D1 的采样时刻，时间终点，第二个时钟沿 D2 的采样时刻。

物理起点，D1 的输入端，物理终点，D2 的输入端。

建立时间满足（关注数据头）：

第一个时钟沿到来时，数据 data 的头部从起点①开始传输，经过 Tco 到达②，在经过 TDelay 到达③。根据建立时间的要求，数据在时钟沿到来之前需要保持稳定的最小时间为 Tsu，假设这里刚好满足建立时间的要求，数据 data 到达③后经过 Tsu 的时间到达了④。计时是从第一个时钟沿开始的，采样在第二个时钟沿，采样的时候 data 在位置④，则有

$Tco+TDelay+Tsu=Tclk$

如果不是最极限的情况，则第二个时钟沿到达③时，data 的头部已经超过④，假设超过④的时间为 t，则有

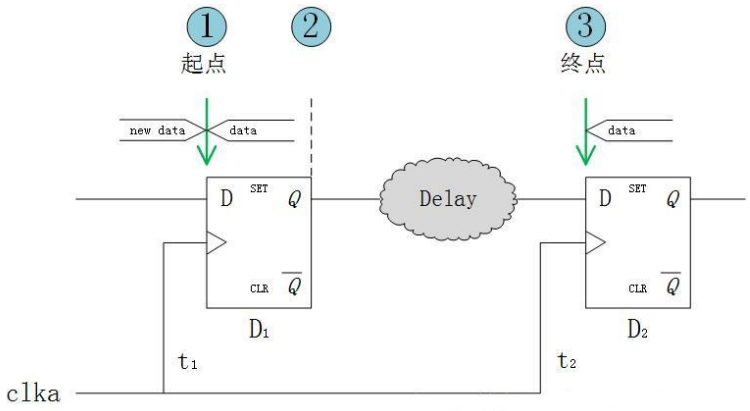
$Tco+TDelay+Tsu+t=Tclk,t\geqslant 0$

即有

$Tco+TDelay+Tsu\leqslant Tclk$

如果这条路径是关键路径（Delay 最长），那么系统能跑到的最大频率为

$Fmax=1min(Tclk)=1Tco+TDelay+Tsu$



保持时间满足（关注数据尾）：

第二个时钟沿到来时，D2 采集数据 data，同时 D1 在采集数据 new data，所以 data 的尾部在第二个时钟沿到来时开始从 D1 的输入端①开始向前传输，经过 Tco 和 TDelay 后到达 D2 的输入端③，所以第二个时钟沿到来之后 data 稳定的时间为

$Tco+TDelay$

根据保持时间的定义，有

$Tco+TDelay\geqslant Th$

加上 CLK 的传输时延：

事实上 Clock 的传输也是有延时的，如图所示，两个触发器的源时钟为 clka，到达 D1 需要 t1 的时间，到达 D2 需要 t2 的时间，t2−t1 其实就是我们常说的 clock skew（时钟偏斜），就是同一个时钟沿达到 D1 和 D2 的时延差别，如果 D1 和 D2 离的很远，那么相应的 clock

skew 就会更大。

建立时间：

加上 clk 传输时延后，变了的是第二个时钟沿到达 D2 的时间，从 TclkTclk 变为 Tclk+t2−t1Tclk+t2−t1，所以有

Tco+TDelay+Tsu≤Tclk+t2−t1

因为 t2 大于 t1，所以对左边时间的限制其实是放宽了
保持时间：
与建立时间相反，因为第二个时钟沿晚来的原因，实际上对保持时间的要求更严格了

Tco+TDelay−(t2−t1)≥Th

从上面的分析可以看到，数据跑得越快（TDelay 越小），时钟传输时延越大（clock skew 越大）对建立时间的满足越有利，而对保持时间的满足越不利，相反则对满足保持时间越有利，对满足建立时间越不利。建立时间还跟时钟周期有关系，时钟周期越小，越容易发生建立时间违例，而保持时间则跟时钟周期没有关系。在设计中，我们常常关注的是建立时间是否满足要求，因为它关系到我们能使用的最小时钟周期有多小，能否跑到预定的工作频率，而因为时钟通常都是走专门的快速线路，很难存在时钟传输时延过大的问题，所以一般也不会出现保持时间违例的情况。

对于建立时间违例的解决办法：

1) 降低时钟频率，即增大时钟周期
2) 再时钟路径上加缓冲器（buffer），让时钟晚到来。
3) 更换具有更小器件延迟的触发器。
4) 组合逻辑优化或插入流水线，缩短关键路径。

对于保持时间违例的解决办法：

1) 在数据路径上插 buffer
2) 更换具有更大器件延迟的触发器
3) 优化时钟路径，让时钟更早到来。

***可以看出，保持时间和建立时间基本是相反的。但是保持时间与时钟周期无关。**

18. 亚稳态

概念：
亚稳态，是因违反寄存器的建立时间和保持时间而产生的。

逻辑电路中绝大多数的时序问题基本都是因为这个原因产生的。
由于寄存器在任何信号通路中都有特定的建立时间和保持时间，这要求输入信号要保持稳定。但如果信号在这段时期发生了变化，那么输出将是未知的。

这个未知的状态便称为亚稳态。

危害：
只要系统中存在异步元件，那么亚稳态就是无法完全避免的。产生亚稳态后，寄存器输出在稳定之前可能是毛刺、振荡、固定的一个电压值。其他与之相连的数字元件收到它的亚稳态，也会发生逻辑混乱，最糟糕的情况就是系统直接崩掉。

常见情况：
输入信号是异步信号
时钟偏移/摆动（上升/下降时间）高于容限值。（时钟信号质量不好）
信号在两个不同频率或者相同频率但相位和偏移不同的时钟域下跨时钟域工作。
组合延迟使寄存器的数据输入在亚稳态窗口内发生变化。

解决方法：
在实际的 FPGA 设计中，人们不会想着怎么降低发生亚稳态的概率，而是想着怎么减小亚稳态的影响。**常见的方法有以下几种：**
1) 降低系统时钟频率
2) 用反应更快的 FF
3) 引入同步机制，防止亚稳态传播（可以采用前面说的加两级触发器）。
   异步信号同步化（两级触发器）
   采用 FIFO 对跨时钟域数据通信进行缓冲
   对复位电路采用异步复位、同步释放处理
4) 改善时钟质量，用边沿变化快速的时钟信号

19. 同步时钟

当**两个时钟间的相位是固定关系的，则可以称这两个时钟为同步时钟(synchronous clock)**。
注意，这个表述很重要。
那么这里就又有一种情况，经过一个 PLL 产生相位不同，但相位固定的两个时钟，他们依旧是同步时钟。

而如果是两个晶振产生的时钟，因为两个晶振在上电时相位差是随机的，而且**不同晶振时钟漂移抖动也不一样，所以相位是不固定的。**
当无法判断两个时钟间的相位时，则可以称这两个时钟为异步时钟(asynchronous clocks)。

20. 同步电路/异步电路

注意，这里不是说用的是什么时钟就是什么电路，不太一样。
异步电路：
a) 电路核心逻辑有用组合电路实现;
b) 异步时序电路的最大缺点是容易产生毛刺;
c) 不利于器件移植;
d) 不利于静态时序分析(STA)、验证设计时序性能。

同步时序电路：

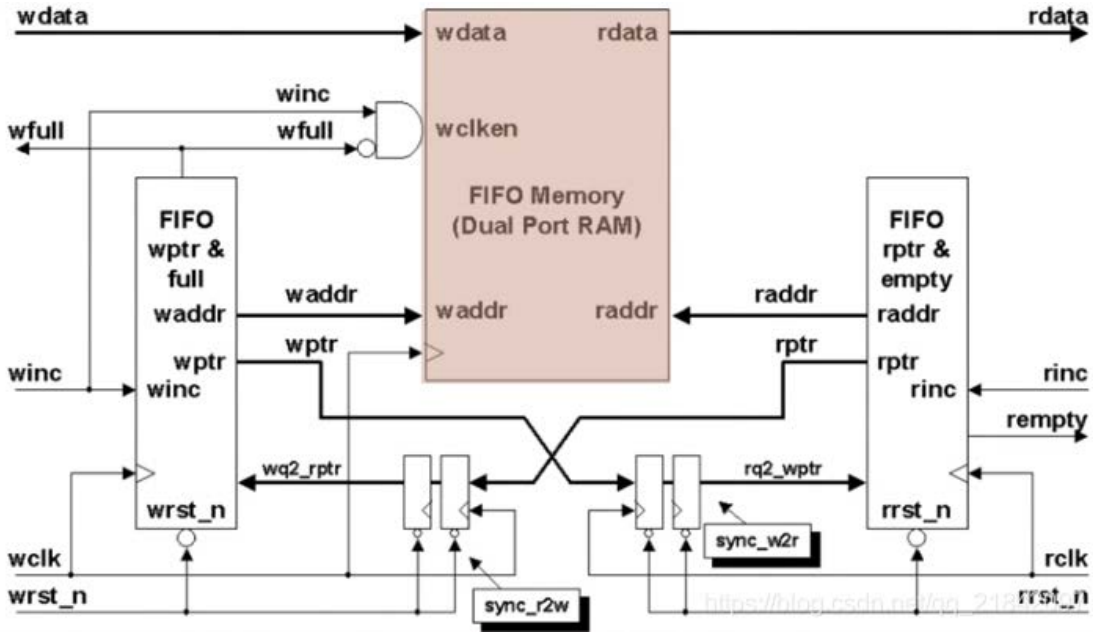
a) 电路核心逻辑是用各种触发器实现;
b) 电路主要信号、输出信号等都是在某个时钟沿驱动触发器产生的;
c) 同步时序电路可以很好的避免毛刺;
d) 利于器件移植;
e) 利于静态时序分析(STA)、验证设计时序性能。

21. 同步 FIFO/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。

异步 FIFO
数据流的传输与指示信号不同在于：数据流大多具有连续性，及背靠背传输；数据流要求信号具有较快的传输速度。**主要的方案是利用 FIFO 进行传输**。

异步 FIFO 结构如下：



1、FIFO 的参数
宽度：一次读写操作的数据位
深度：可以存储的 N 位数据的数目(宽度为 N)
满标志：FIFO 已满或将要满时，由 FIFO 的状态电路送出的信号，阻止 FIFO 写操作
空标志：FIFO 已空或将要空时，由 FIFO 的状态电路送出的信号，阻止 FIFO 读操作
读时钟：读操作所遵循的时钟
写时钟：写操作所遵循的时钟

异步 FIFO 的设计主要有 5 部分组成：

- FIFO Memory
  双口 RAM 存储数据
- sync_r2w

同步读数据指针到写时钟域

- sync_w2r

同步写数据指针到读时钟域

- wptr_full

处理写指针和满信号的逻辑

- rptr_empty

处理读指针和空信号的逻辑

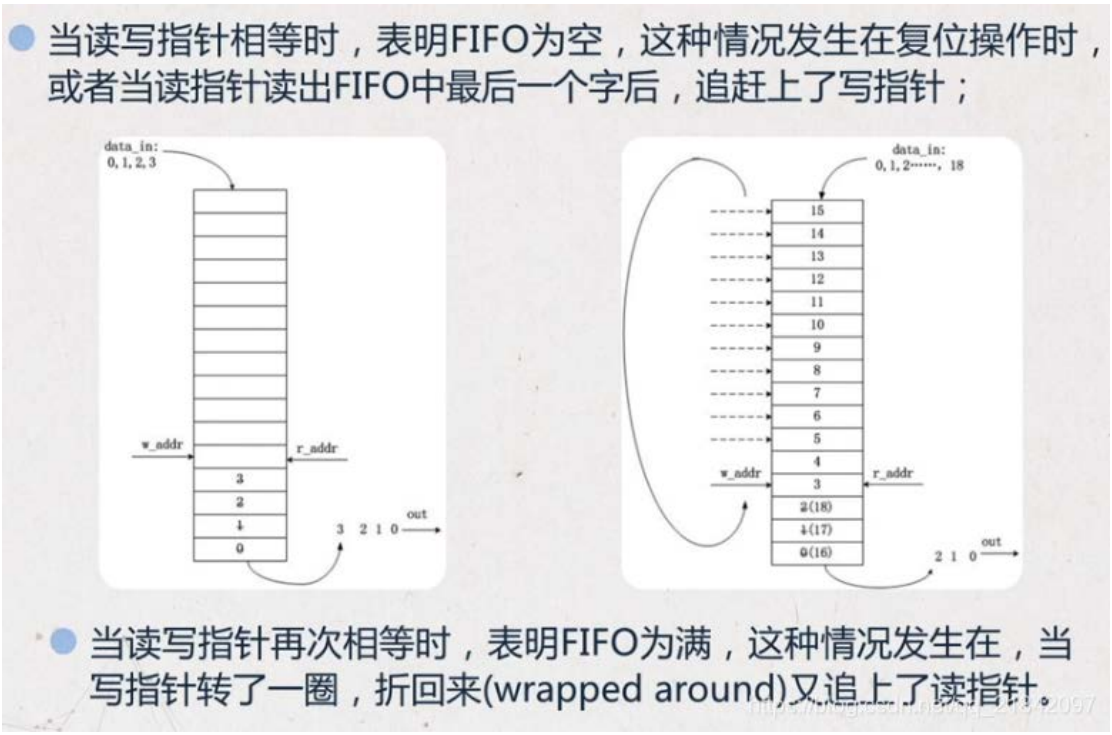
2、设计空满标志位电路

正确产生空满信号是任何 FIFO 设计的关键。其设计原则是：能写满而不溢出，能读空而不多读。

在写时钟域下，判断读写指针的关系，生成满标志

在读时钟域下，判断读写指针的关系，生成空标志

对于空满判断，可看下图

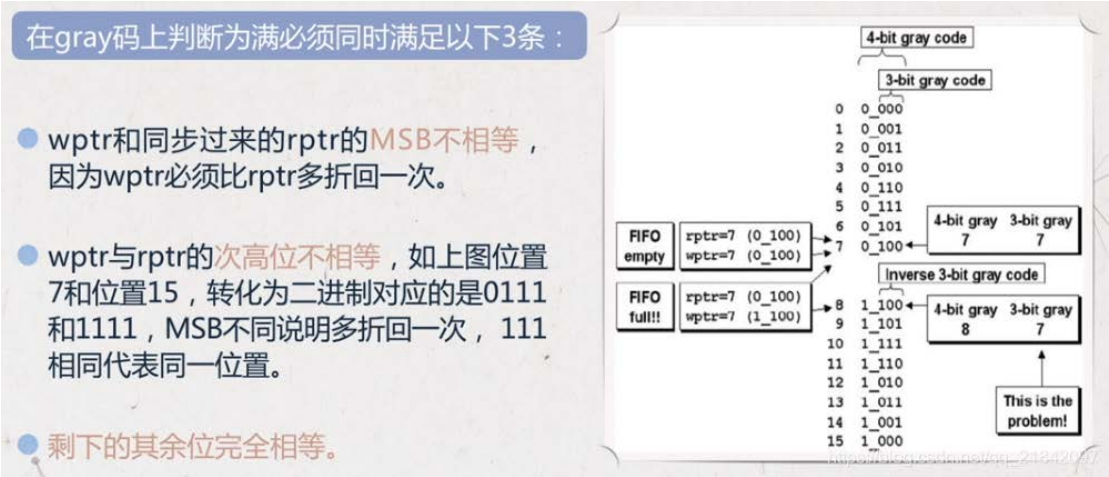


在判断空和满的情况下，都会有读写指针相等的情况，可采用以下方法进行区分：

在地址中额外添加一个位，当写指针增加并越过最后一个地址位时，就将这个额外的位(MSB)加一，其他位回 0。读指针也进行同样的操作。额外的位作为折回标志位。如果两个指针的 MSB 不同，其余位相等，说明读指针比写指针少折回了一次，这种情况下，FIFO 为满。而读写指针所有位都相等，说明折回次数相等，此时 FIFO 为空。

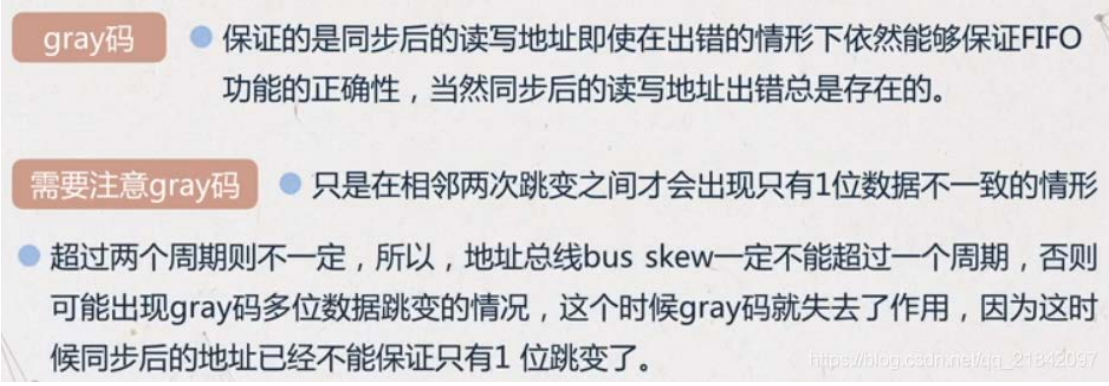
解决空满判断后，则是读写指针同步的问题。二进制的读写指针通常位宽超过了 1bit，而多比特信号是不可以直接使用两级同步器的。这时，考虑到 FIFO 地址是连续变化的，我们使用格雷码来进行传输。格雷码相邻两个数值只有一位发生变化，0 和最大数之间也只有一位不同。

而转换为格雷码进行传输后，如何进行空满判断？如下图：



至于为什么选择格雷码进行传输，我们来看格雷码传输失效的情况：

格雷码传输失效，只有一位同步出错，此时地址没有跳变。如果是写地址同步失效，用这个错误的写地址在读时钟域进行空判断时不会出错，最多是让空标志在 FIFO 不是真正空的时候产生，而不会产生空读的情况。格雷码保证的是同步后的读写地址即使在出错的情况下依然能够保证 FIFO 功能的正确性。



3、设计 FIFO 的深度

假设 FIFO 的写时钟为 100MHZ，读时钟为 80MHZ。在 FIFO 输入侧，每 100 个时钟，写入 80 个数据；FIFO 读入测，每个时钟读取一个数据。设计合理的 FIFO 深度，使 FIFO 不会溢出。

我们需要考虑数据轻载和重载的情况，**对缓存能力要求最高的情况为背靠背传输，则 FIFO 深度为 160-(160/100)*80=32**

我们将问题一般化：

1) 写时钟频率 WCLK
2) 读时钟频率 RCLK
3) 写入测每 B 个时钟周期有 A 个数据写入
4) 读取测每 Y 个时钟周期有 X 个数据读出

则 FIFO 的深度为，$$Depth = burst\_length - \frac{burst\_length}{wclk} * (rclk * \frac{X}{Y})$$

其中 burst_length 与写入测情况有关。

22. 同步复位/异步复位

同步复位

同步复位只有在时钟沿到来时复位信号才起作用，则复位信号持续的时间应该超过一个时钟周期才能保证系统复位。

同步复位的优点：一般能够确保电路是百分之百同步的。确保复位只发生在有效时钟沿，可以作为过滤掉毛刺的手段。
同步复位的缺点：复位信号的有效时长必须大于时钟周期，才能真正被系统识别并完成复位。同时还要考虑如：时钟偏移、组合逻辑路径延时、复位延时等因素。由于大多数的厂商目标库内的触发器都只有异步复位端口，采用同步复位的话，就会耗费较多的逻辑资源。

异步复位

异步复位只要有复位信号系统马上复位，因此异步复位抗干扰能力差，有些噪声也能使系统复位，因此有时候显得不够稳定，要想设计一个好的复位最好使用异步复位同步释放。

异步复位优点：异步复位信号识别方便，而且可以很方便的使用全局复位。由于大多数的厂商目标库内的触发器都有异步复位端口，可以节约逻辑资源。
异步复位缺点：复位信号容易受到毛刺的影响。复位结束时刻恰在亚稳态窗口内时，无法决定现在的复位状态是 1 还是 0，会导致亚稳态。

异步复位同步释放

使用异步复位同步释放就可以消除上述缺点。所谓异步复位，同步释放就是在复位信号到来的时候不受时钟信号的同步，而是**在复位信号释放的时候受到时钟信号的同步。**

23. 跨时钟域处理

单 bit 信号

1) 电平检测：

最为简单的方法。通过寄存器打两拍进行同步，也就是所谓的电平同步器。

存在问题为，输入信号必须保持两个接收时钟周期，每次同步完，输入信号要恢复到无效状态。所以，如果是从快到慢，信号很有可能被滤除。

适用于慢时钟域向快时钟域。

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。
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 。

为什么是打两拍呢，打一拍、打三拍行不行呢？
两级并不能完全消除亚稳态危害，但是提高了可靠性减少其发生概率。总的来讲，就是一级

概率很大，三级改善不大。如果再加上第三级寄存器，**由于第二级寄存器对于亚稳态的处理已经起到了很大的改善作用，第三级寄存器在很大程度上可以说只是对于第二级寄存器的延拍，所以意义是不大的。**

2) 边沿检测：

在电平同步器的基础上，通过输出端的逻辑组合，可以完成对于信号边沿的提取，**识别上升沿，下降沿以及双边沿，并发出相应的脉冲。**

比起电平同步器，更适合要求脉冲输出的电路。

但同样，输入信号必须保持两个接收时钟周期。

适用于慢时钟域向快时钟域。

3) 脉冲同步：

先**将原时钟域下的脉冲信号，转化为电平信号（使用异或门），再进行同步**，同步完成之后再把新时钟域下的电平信号转化为脉冲信号（边沿检测器的功能）。

这就从快时钟域的取出一个单时钟宽度脉冲，在慢时钟域建立新的单时钟宽度脉冲。

结合了之前所提到的两种同步器。

但存在一个问题，输入脉冲的时间间距必须在两个接收时钟周期以上，否则新的脉冲会变宽，这就不再是单时钟脉冲了。

适用于快时钟域向慢时钟域。

对于多 bit 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）

1) 握手：

握手指的是两个设备之间通信的一种方式，用来通信的信号就是握手信号。**最简单的握手信号是 valid 和 ready**，也可以叫 request 和 grant。假设设备 1 向设备 2 发送数据，设备 1 不知道设备 2 什么时候可以接收数据，设备 2 也不知道设备 1 什么时候会发送数据，那么它们之间如果用握手通信可能是这样的顺序：

设备 1 将 valid 信号置 1，告诉设备 2，数据准备就绪了，请查收

设备 2 此刻正处于忙碌状态无法接收数据，设备 2 将 ready 信号保持为 0

设备 2 空闲了，将 ready 信号置 1 接收设备 1 的数据

设备 1 看到设备 2 的 ready 为 1，它知道设备 2 已经接收好数据了，将 valid 置 0 同时撤销数据，准备下一次发送。

可以看到因为有握手控制，可以确保数据的正确传输，不会丢失。**跨时钟域的握手设计就是利用握手控制这种优势，从而避免因为跨时钟域引起的数据传输错误。**

24. MEALY/MOORE 状态机

Moore 型状态机：**输出只由当前状态决定**，即次态=f(现状，输入)，输出=f（现状）；

Mealy 型状态机：**输出不但与当前状态有关，还与当前输入值有关**，即次态=f(现状，输入)，输出=f（现状，输入）；

不同点在于：
Moore 型状态机的输出信号是直接由状态寄存器译码得到，而 Mealy 型状态机则是以现时的输入信号结合即将变成次态的现态，编码成输出信号。

Mooer 状态机的输出只与当前的状态有关，也就是数当前的状态决定输出，而与此时的输入无关，输入只决定状态机的状态改变，不影响电路最终的输出。（注意：这里所说的输出不是状态机的状态机状态的输出，而是当前状态的所代表的含义，比如：检测 110 序列的状态机，当状态机跳转到 STA_GOT110 时，电路会有一个输出信号，假如说是 find，此时 find 就会为高电平，其他（状态时）时 find 就会为低电平。find 是我们最后电路的输出，find 的值置于我们的转台机当前所处的状态有关，而与输出无关）。用一本书上的话说就是：Moore 状态机的每一状态指定它的输出独立于电路的输入 。

Mealy 状态机的输出不仅与当前的状态有关，还与当前的输入有关(同样，不要误认为状态机的输出只能是状态机的状态)，即当前的输入和当前的状态共同决定当前的输入。

25. 有限状态机 FSM 设计

状态机描述时关键是要描述清楚几个状态机的要素，即如何进行状态转移，每个状态的输出是什么，状态转移的条件等。具体描述时方法各种各样，最常见的有三种描述方式：

（1）一段式：整个状态机写到一个 always 模块里面，在该模块中既描述状态转移，又描述状态的输入和输出；

（2）二段式：用两个 always 模块来描述状态机，其中一个 always 模块采用同步时序描述状态转移；另一个模块采用组合逻辑判断状态转移条件，描述状态转移规律以及输出；

（3）三段式：**在两个 always 模块描述方法基础上，使用三个 always 模块，一个 always 模块采用同步时序描述状态转移，一个 always 采用组合逻辑判断状态转移条件，描述状态转移规律，另一个 always 模块描述状态输出(**可以用组合电路输出，也可以时序电路输出)。

三段式状态机示例模板：

```
//第一个进程，同步时序 always 模块，格式化描述次态寄存器迁移到现态寄存器
always @ (posedge clk or negedge rst_n)    //异步复位
if(!rst_n)
        current_state <= IDLE;
else
        current_state <= next_state; //注意，使用的是非阻塞赋值

//第二个进程，组合逻辑 always 模块，描述状态转移条件判断
always @ (current_state) //电平触发，现存状态为敏感信号
begin
        next_state = x; //要初始化，使得系统复位后能进入正确的状态
case(current_state)
        S1: if(...)
```

```
                    next_state = S2; //阻塞赋值
        S2: if(...)

                    next_state = S3; //阻塞赋值
.....
endcase
end

//第三个进程，同步时序 always 模块，格式化描述次态寄存器输出
always @ (posedge clk or negedge rst_n)
begin
...//初始化
case(next_state)
        S1:
                out1 <= 1'b1; //注意是非阻塞逻辑
         S2:
                out2 <= 1'b1;
          default:... //default 的作用是免除综合工具综合出锁存器
endcase

end
```

26. 系统工作频率计算

最高运行速度：

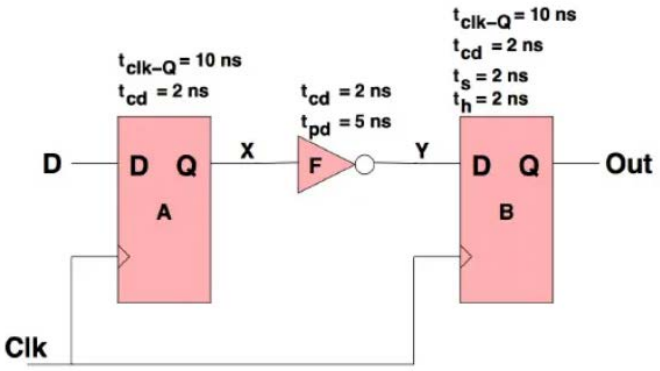
Tco ——触发器的输入数据被时钟打入到触发器到数据到达触发器输出端的延时时间；
Tdelay——组合逻辑的延时；
Tsetup ——D 触发器的建立时间。

假设数据已被时钟打入 D 触发器，那么数据到达第一个触发器的Q输出端需要的延时时间是 Tco，

经过组合逻辑的延时时间为 Tdelay，

然后到达第二个触发器的D端，要希望时钟能在第二个触发器再次被稳定地打入触发器，则时钟的延迟必须大于 Tco + Tdelay + Tsetup，也就是说：

最小的时钟周期 Tmin =Tco + Tdelay + Tsetup，
即最快的时钟频率 Fmax=1/Tmin。



如图中例子，组合逻辑模块其实只有一个反相器。
最小的时钟周期为 Tmin = tClk-Q(A) + tpd(F) + ts(B) = 10+5+ 2=17 ns
最大时钟频率为 fmax = 1/Tmin = 58.5 MHz。

Tdelay < Tperiod - Tsetup – Thold
Tperiod > Tsetup + Thold +Tdelay （用来计算最高时钟频率）
Tco= Tsetup + Thold 即触发器的传输延时

27. 关键路径

关键路径通常是指同步逻辑电路中，组合逻辑时延最大的路径（这里我认为还需要加上布线的延迟），也就是说关键路径是对设计性能起决定性影响的时序路径。

对关键路径进行时序优化，可以直接提高设计性能。对同步逻辑来说，常用的时序优化方法包括 Pipeline、Retiming、逻辑复制、加法/乘法树、关键信号后移、消除优先级等解决。

静态时序分析能够找出逻辑电路的关键路径。通过查看静态时序分析报告，可以确定关键路径。

1. **组合逻辑中插入寄存器（插入流水线）**

组合逻辑的延时过长，就会成为关键路径，这时可以考虑在该路径上插入额外的寄存器，这种方法也称为插入流水线，多用于高度流水的设计中，因为这种设计中额外插入寄存器增加的时钟周期延时并不会违反整个设计的规范要求，从而不会影响设计的总体功能性实现，也即额外插入的寄存器在保持吞吐量不变的情况下改善了设计的时序性能。当然，其不可避免地会带来部分面积的增加。

2. 寄存器平衡（重定时 Retiming）

在不增加寄存器个数的前提下，通过改变寄存器的位置来优化关键路径，可以对比和流水线插入寄存器的不同。

3. **操作符平衡（加法树、乘法树）**

平衡前，a 和 b 均经过 3 个乘法器带来的延时，c 经历 2 个，d 经历 1 个，最长延时为 3 个乘法器延时。平衡后，树形结构，a、b、c、d 均经历 2 个乘法器延时，最长延时为 2 个乘法器延时。

4. 消除代码优先级（case 代替 if…else）
本身确实不需要优先级的地方，可以使用 case 代替 if…else，使得顺序执行的语句编程并行执行。如果确实有优先级兵，则不能这样做。这种消除代码中的优先级的策略也称为代码结构平坦化技术，主要针对那些带优先级的编码结构。

5. 逻辑复制
当某个信号的扇出 fanout 比较大时，会造成该信号到各个目的逻辑节点的路径变得过长，从而成为设计中的关键路径，此时可以通过对该信号进行复制来降低扇出。高扇出的危害是大大增加了布局布线的难度，这样其扇出的节点也就无法被布局得彼此靠近，所以就导致了布线长度过大的问题。

6. 关键信号后移
关键输入应该在逻辑最后一级提供，其中关键输入为芯片、Slice、或者 LUT 提供的时延最大的输入，比如在 if…else if…链中，将关键信号放在第一级。

28. PLL

PLL 的英文全称是 Phase Locked Loop，即锁相环，是一种反馈控制电路。
锁相环作为一种反馈控制电路，其特点是利用外部输入的参考信号控制环路内部震荡信号的频率和相位。因为锁相环可以实现输出信号频率对输入信号频率的自动跟踪，所以锁相环通常用于闭环跟踪电路。锁相环在工作的过程中，当输出信号的频率与输入信号的频率相等时，输出电压与输入电压保持固定的相位差值，即**输出电压与输入电压的相位被锁住**，这就是锁相环名称的由来

29. FPGA 开发工具

目前 FPGA 的生产厂家主要有 ALTERA，Xilinx，Actel，Lattice。
Altera 公司生产的 FPGA 和 CPLD，开发工具主要用 Quartus;
Xilinx 公司的 FPGA 主要用 ISE，vivado 作为开发工具;

30. EDA 开发工具

数字逻辑仿真工具：
cadence: Incisive
synopsys: VCS
mentor: QuestaSim
ModelSim

数字逻辑综合工具：
Cadence:Genus
Synopsis: design
Compiler (DC)

数字后端设计工具：
1，自动布局布线工具
Cadence: Innovus

Synopsis: IC Compiler

2. 物理验证工具

1, Mentor: Calibre

2, Synopsis: ICV

3, Cadence: PVS/Pegasus

31. 面积/速度问题

面积（资源）优化

无论是在 ASIC 还是 FPGA 中，硬件设计资源即面积(Area)是一个重要的技术指标。

面积优化意义：

1.可使用规模更小的可编程器件，降低系统成本，提高性价比。

2.当耗用资源过多时会严重影响时序性能。

3.为后续技术升级留下更多的可编程资源。

4.资源耗用太多会使器件功耗显著上升。

优化方法如下

1) 资源共享：

主要针对数据通路中耗费逻辑资源较多的模块，通过选择、复用的方式共享使用该模块，达到减少资源使用、优化面积的目的。

并不是在任何情况下都能以此法实现资源优化，输入与门之类的模块使用资源共享是无意义的，有时甚至会增加资源的使用（多路选择器的面积显然要大于与门）。高级的 HDL 综合器，如 QuartusII 和 Synplify Pro 等，通过设置能自动识别设计中需要资源共享的逻辑结构，自动地进行资源共享。

2) 逻辑优化

使用优化后的逻辑进行设计，可以明显减少资源的占用。

状态机的设计要尽可能简洁，不要搞出不必要的，多个状态对同一个数据做处理。

if 语句中用单 bit 的标志位代替多 bit 的数据来进行条件判断。

设计尽可能简洁，不要弄出冗余赋值。

3) 串行化

将原来耗用资源巨大、单时钟周期内完成的并行执行的逻辑块分割开，提取出相同的逻辑模块（一般为组合逻辑块），在时间上利用该逻辑模块，用多个时钟周期完成相同的功能，其代价是工作速度被大为降低。

速度优化

一般来说，速度优化比资源优化更重要，需要优先考虑。速度优化包括：FPGA 的结构特性、HDL 综合器性能、系统电路特性、PCB 制版情况等，也包括 Verilog 的编程风格。下面主要讨论电路结构方面的速度优化方法。

优化方法如下：

1) 流水线设计

流水线(Pipelining)是一种在复杂组合逻辑之间添加寄存器的方法，是最常用的速度优化技术之一。它能显著地提高设计电路的运行速度上限。

2) 寄存器配平 (Register Balancing)

寄存器配平是使较长路径缩短，较短路径加长，使其达到平衡从而提高工作频率的一种技术。若设计中，若两个组合逻辑块的延时差别过大，若 T1>T2，则总体的工作频率 fmax 取决于 T1，即最大的延时模块。对不合理设计进行改进，即将原本设计中的组合逻辑 1 的部分逻辑转移到组合逻辑 2 中，使 t1≈t2，且 T1+T2=t1+t2， T1>t1 则总体的工作频率 fmax 提高。

3) 关键路径法

关键路径：指设计中从输入到输出经过的延时最长的逻辑路径。优化关键路径是提高设计工作速度的有效方法。

EDA 工具中的综合器及设计分析器都提供关键路径的信息以便设计者改进设计。Quartus 中的静态时序分析工具可以帮助找到延时最长的关键路径。

4) **乒乓操作法**

乒乓操作法是 FPGA 开发中的一种数据缓冲优化设计技术，可视作另一种形式的流水线技术。乒乓操作本质是使用 2 倍的硬件资源，通过将数据产生时间和数据使用时间重叠，解决一个数据生产效率低于数据使用效率的问题。是一种拿面积换性能的方法。通过“输入数据流选择单元”和“输出数据流选择单元”按节拍、相互配合的切换，将经过缓冲的数据流“无缝”地送到“数据流运算处理模块。

5) 树形结构法

若要实现 A+B+C+D。首先实现 AB=A+B，CD=C+D，将 AB/CD 锁存一个时钟周期再相加。树形结构法和上面面积优化提到的串行化是相反的。

32. 时序约束

时序约束主要包括周期约束，偏移约束，静态时序路径约束三种。通过附加时序约束可以综合布线工具调整映射和布局布线，使设计达到时序要求。

附加时序约束的一般策略是先附加全局约束，然后对快速和慢速例外路径附加专门约束。

附加全局约束时，首先定义设计的所有时钟，对各时钟域内的同步元件进行分组，对分组附加周期约束，然后对 FPGA/CPLD 输入输出 PAD 附加偏移约束、对全组合逻辑的 PAD TO PAD 路径附加约束。

附加专门约束时，首先约束分组之间的路径，然后约束快、慢速例外路径和多周期路径，以及其他特殊路径。

约束的作用？

1：提高设计的工作频率（减少了逻辑和布线延时）；

2：获得正确的时序分析报告；

（静态时序分析工具以约束作为判断时序是否满足设计要求的标准，因此要求设计者正确输入约束，以便静态时序分析工具可以正确的输出时序报告）

3：指定 FPGA/CPLD 的电气标准和引脚位置。

33. 静态/动态时序分析

静态时序分析（static timing analysis，STA）是遍历电路存在的所有时序路径，根据给定工

作条件（PVT）下的时序库.lib 文件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约束要求，**根据最大路径延时和最小路径延时找出违背时序约束的错误。**

静态时序分析的优点：
1) 不需要给输入激励；
2) 几乎能找到所有的关键路径（critical path）；
3) 运行速度快；

静态时序分析的缺点：
1) 只适用同步电路；
2) 无法验证电路的功能；
3) 需要比较贵的工具支持；
4) 对于新工艺可能还需要建立一套特征库，建库的代价可能要几百万。

静态时序分析的工具：
Synopsys 的 prime time，
Cadence 的 Encounter Timing System 等

动态时序分析（dynamic timing analysis，DTA）通常是所有的输入信号都会给一个不同时刻的激励，在 testbech（.sp 或者.v）中设置一段仿真时间，最后**对仿真结果进行时序和功能分析**。 这里的仿真可以是门级或者晶体管级，包括 spice 格式和 RTL 格式的网表。

动态时序分析的优点：
1) 晶体管级的仿真比较精确，直接基于工厂提供的 spice 工艺库计算得到；
2) 适用于任何电路，包括同步、异步、latch 等等；
3) 不需要额外搞一套特征库；
4) 不需要很贵的时序分析工具。

缺点：
1) 需要给不同的测试激励；
2) 关键路径无法检查全 （致命性的）；
3) 规模大的电路 spice 仿真特别慢 （致命性的）。

动态时序的工具 ：
spice 仿真器: hspice, finesim, hsim, spectre 等；
verilog 仿真器: ModelSim,VCS,NC-Verilog,Verilog-XL 等。

34. DMA

DMA(Direct Memory Access，直接存储器访问）是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载。否则，CPU 需要从来源把每一片段的资料复制到暂存器，然后把它们再次写回到新的地方。在这个时间中，CPU 对于其他的工作来说就无法使用。
DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，

传输动作本身是由 DMA 控制器来实行和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理器工作拖延，反而可以被重新排程去处理其他的工作。DMA 传输对于高效能 嵌入式系统算法和网络是很重要的。
在实现 DMA 传输时，是由 DMA 控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即 DMA 传输前，CPU 要把总线控制权交给 DMA 控制器，而在结束 DMA 传输后，DMA 控制器应立即把总线控制权再交回给 CPU。一个完整的 DMA 传输过程必须经过 DMA 请求、DMA 响应、DMA 传输、DMA 结束 4 个步骤。

35. 逻辑电平

常用逻辑电平：12V，5V，3.3V。
TTL 和 CMOS 不可以直接互连，由于 TTL 是在 0.3-3.6V 之间，而 CMOS 则是有在 12V 的有在 5V 的。CMOS 输出接到 TTL 是可以直接互连。TTL 接到 CMOS 需要在输出端口加一上拉电阻接到 5V 或者 12V。
用 CMOS 可直接驱动 TTL;加上拉电阻后,TTL 可驱动 CMOS.

36. 逻辑最小项

在一个有 n 个变量的逻辑函数中，包括全部 n 个变量的乘积项（每个变量必须而且只能以原变量或反变量的形式出现一次）称为最小项。n 个变量有 2^n 个最小项，比如当 n = 3 时，此逻辑函数应有 2^3 = 8 个最小项。，分别是：A'B'C', A'B'C, A'BC', A'BC, AB'C', AB'C, ABC', ABC 最大项就是全部 n 个变量的加和了。

性质
1) 对于任意一个最小项，输入变量只有一组取值使得它的值为 1，而在变量取其他各组值的时候，这个最小项的值都为 0。
2) 不同的最小项，使得它的值为 1 的那一组输入变量取值也不同。
3) 对于输入变量的任何一组取值，任意两个最小项的乘积为 0。
4) 对于输入变量的任何一组取值，全体最小项的和为 1。

37. 乒乓 buffer

概念
乒乓 buffer 是由两个单口 sram 背靠背组成的一种电路结构，假设我们称其为 s1 和 s2。则乒乓 buffer 的工作方式如下。
1) 首先向 s1 中写入数据，此时 s2 是空的，因此没有操作。
2) 当向 s1 写入完毕，通过逻辑操作，使得接下来向 s2 中写入数据，于此同时其他模块可以从 s1 中读出已经写入的数据;
3) 待 s2 中写完，再次转换，重新向 s1 中写入数据，同时其他模块从 s2 中读出数据。
4) 由于这个过程中两个 buffer 总是一个读一个写，并且互相交换读/写角色，因此称其为乒乓 buffer。

应用场景：
当后面的处理单元在工作期间，前面的 buffer 的内容不能被释放。或者，在处理单元工作期间，buffer 的特定地址的内容不止被访问一次。

注：对于 buffer 的内容用一次就可以被释放的应用场景（如处理图像数据）：直接用 FIFO 结构，或者移位寄存器 即可实现。

38. 门控时钟

为什么要门控时钟：

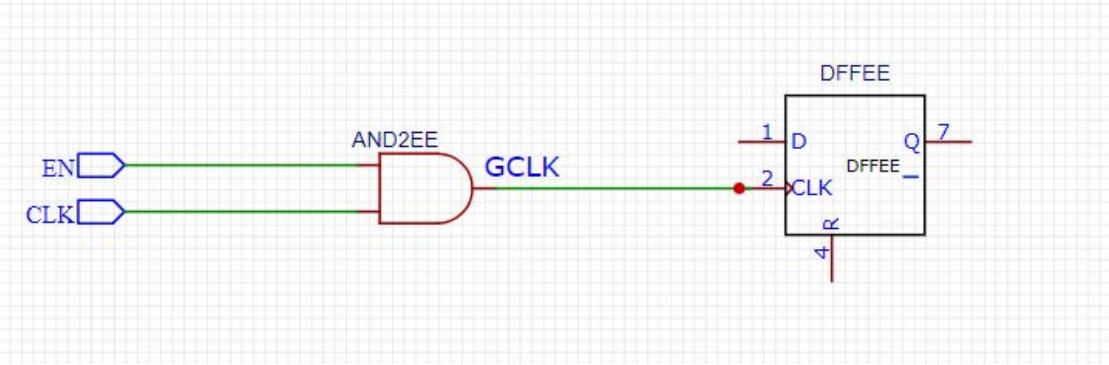
芯片功耗组成中，有高达 40%甚至更多是由时钟树消耗掉的。这个结果的原因也很直观，因为这些时钟树在系统中具有最高的切换频率，而且有很多时钟 buffer，而且为了最小化时钟延时，它们通常具有很高的驱动强度。此外，即使输入和输出保持不变，接收时钟的触发器也会消耗一定的功耗。而且这些功耗主要是动态功耗。

那么**减少时钟网络的功耗消耗，最直接的办法就是如果不需要时钟的时候，就把时钟关掉。这种方法就是大家熟悉的门控时钟：clock gating。**(电路图中看到的 CG cell 就是门控时钟了)。

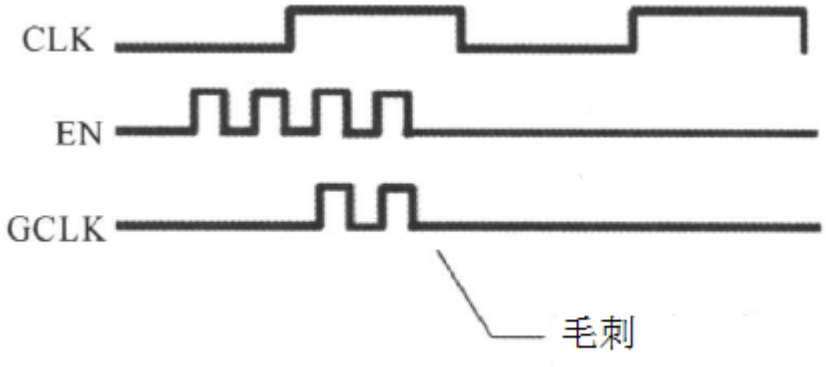
门控时钟的结构

1) 与门门控

如果让我们设计一个门控时钟的电路，我们会怎么设计呢？最直接的方法，不需要时钟的时候关掉时钟，这就是与操作，我们只需要把 enable 和 CLK 进行“与”操作不就行了么，电路图如下：



这种直接将控制 EN 信号和时钟 CLK 进行与操作完成门控的方式，可以完成 EN 为 0 时，时钟被关掉。但是同时带来另外一个很大的问题：毛刺



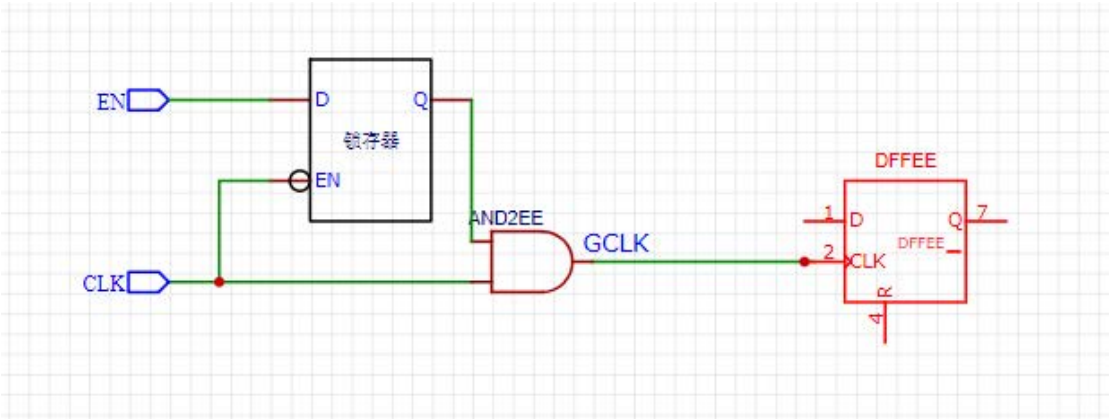
如上图所示，EN 是不受控制的，随时可能跳变，这样纯组合输出 GCLK 就完全可能会有毛刺产生。时钟信号上产生毛刺是很危险的。实际中，这种直接与门的方式基本不会被采样。所以我们需要改进电路，为了使门控时钟不产生毛刺，我们必须对 EN 信号进行处理，使其在 CLK 的高低电平期间保持不变，或者说 EN 的变化就是以 CLK 为基准的。

1 很自然的我们会想到触发器，只要把 EN 用 CLK 寄存一下，那么输出就是以 CLK 为基准

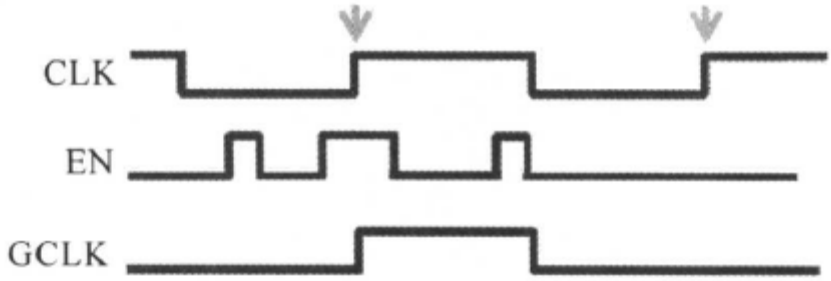
的;
2 其实还有一种办法是锁存器，把 EN 用锁存器锁存的输出，也是以 CLK 为基准的。

2) 锁存门控

我们先看一下第二种电路，增加锁存器的电路如下:



对应的时序如下:



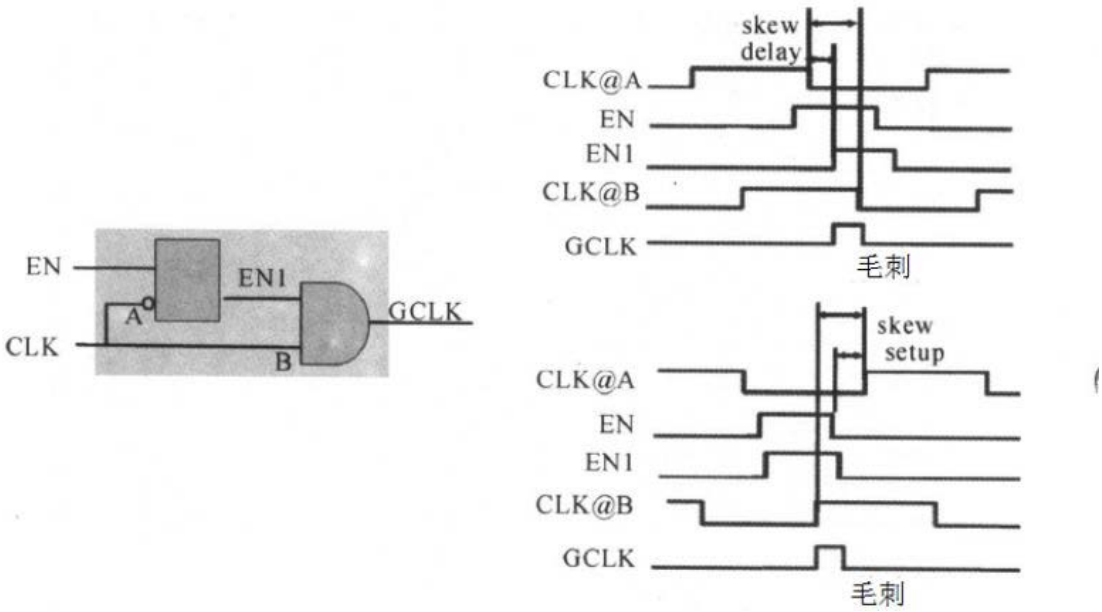
可以看到，只有在 CLK 为高的时候，GCLK 才可能会输出高，这样就能消除 EN 带来的毛刺。这是因为 D 锁存器是电平触发，在 clk=1 时，数据通过 D 锁存器流到了 Q；在 Clk=0 时，Q 保持原来的值不变。

虽然达到了我们消除毛刺的目的，但是这个电路还有两个缺点:

1 如果在电路中，锁存器与与门相隔很远，到达锁存器的时钟与到达与门的时钟有较大的延迟差别，则仍会出现毛刺。

2 如果在电路中，时钟使能信号距离锁存器很近，可能会不满足锁存器的建立时间，会造成锁存器输出出现亚稳态。

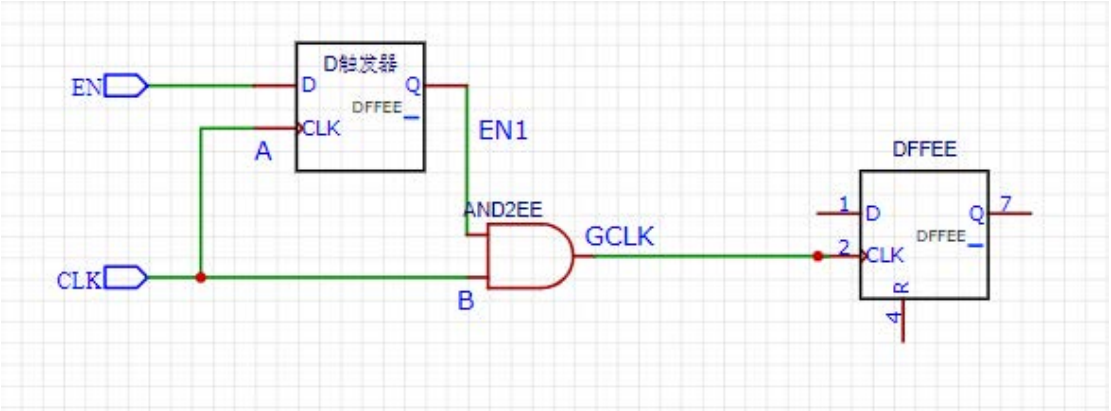
如下图分析所示:



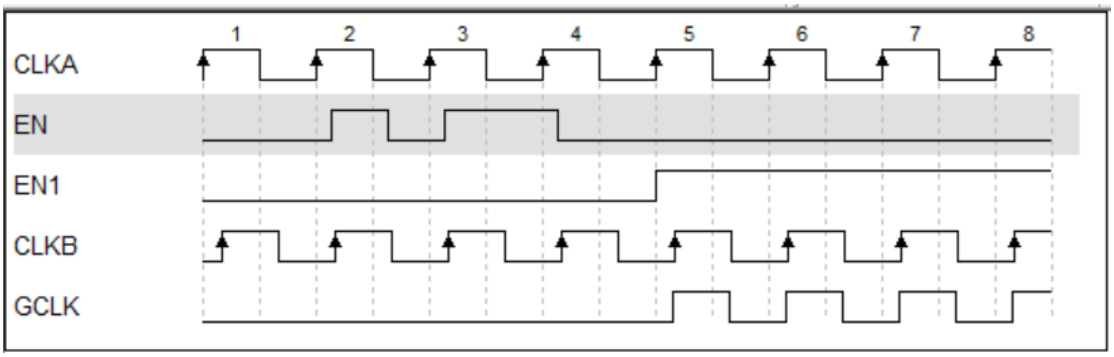
上述的右上图中，B 点的时钟比 A 时钟迟到，并且 Skew > delay，这种情况下，产生了毛刺。为了消除毛刺，要控制 Clock Skew，使它满足 Skew >Latch delay（也就是锁存器的 clk-q 的延时）。上述的右下图中，B 点的时钟比 A 时钟早到，并且|Skew| > ENsetup 一 (D->Q)，这种情况下，也产生了毛刺。为了消除毛刺，要控制 Clock Skew，使它满足|Skew|< ENsetup 一(D->Q)。

3) 寄存门控

如 1.1 中提到的，我们还有另外的解决办法，就是用寄存器来寄存 EN 信号再与上 CLK 得到 GCLK，电路图如下所示：



时序如下所示：



由于 DFF 输出会 delay 一个周期，所以除非 CLKB 上升沿提前 CLKA 很多，快半个周期，才会出现毛刺，而这种情况一般很难发生。但是，这种情况 CLKB 比 CLKA 迟到，是不会出现毛刺的。
当然，如果第一个 D 触发器不能满足 setup 时间，还是有可能产生亚稳态。

SOC 芯片设计中使用最多的是锁存结构的门控时钟：
原因是:在实际的 SOC 芯片中，要使用大量的门控时钟单元。所以通常会把门控时钟做出一个标准单元，有工艺厂商提供。那么锁存器结构中线延时带来的问题就不存在了，因为是做成一个单元，线延时是可控和不变的。而且也可以通过挑选锁存器和增加延时，总是能满足锁存器的建立时间，这样通过工艺厂预先把门控时钟做出标准单元，这些问题都解决了。

那么用寄存器结构也可以达到这种效果，为什么不用寄存器结构呢？那是因为面积！一个 DFF 是由两个 D 锁存器组成的，采样 D 锁存器组成门控时钟单元，可以节省一个锁存器的面积。当大量的门控时钟插入到 SOC 芯片中时，这个节省的面积就相当可观了。

所谓门控时钟就是指连接触发器时钟端来自于组合逻辑，凡是组合逻辑在布局布线之后肯定会产生毛刺，而如果采用这种有毛刺的信号来作为时钟使用的话将会出现功能上的错误。

39. BRAM/DRAM

1 Block RAM（BRAM）
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。如 Xilinx 公司的结构中每个 BRAM 有 36Kbit 的容量，既可以作为一个 36Kbit 的存储器使用，也可以拆分为两个独立的 18Kbit 存储器使用。反过来相邻两个 BRAM 可以结合起来实现 72Kbit 存储器，而且不消耗额外的逻辑资源。

Block RAM 都有两套访问存储器所需的地址总线、数据总线及控制信号灯信号，因此其既可以作为单端口存储器，也可以作为双端口存储器。需要注意的时访问 BRAM 需要和时钟同步，异步访问不支持的。

2 查找表存储器——分布式 RAM（DRAM）
只有成为 SLICEM 的逻辑块里的查找表才可以用做分布式 RAM。
利用查找表为电路实现存储器，既可以实现芯片内部存储，又能提高资源利用率。**分布式 RAM 的特点是可以实现 BRAM 不能实现的异步访问。
**不过使用分布式 RAM 实现大规模的存储器会占用大量的 LUT，可用来实现逻辑的查找表就会减少。因此建议仅在需要小规模存储器时，使用这种分布式 RAM。

40. 功耗问题

一般分为静态功耗和动态功耗。
动态功耗：发生在门开关（或状态翻转）的瞬间。是由于对 电容充电和 电源和地之间短暂电流通路造成的。它正比于开关频率。
静态功耗：总是存在，是由电源和地之间的静态导通电流（或漏电流）引起的。

降低动态功耗：

1、降低内核供电电压。降低供电电压会影响到时序性能。为了弥补这个影响，一般采用流水线（pipelining）和并行（parallelism）设计方法来提高设计性能。当然，这样同时也会增加设计面积。

2、降低电容负载和逻辑开关频率。

41. 波形 X 问题

看波形首先先看 X 和 Z。

任何一个波形，无论是验证的前期、中期、后期，到手之后，先刷屏，找 X 和 Z，确认。某些 Z 和 X 是可以存在的，例如某些 IP 模型，或者未初始化的寄存器和 RAM，但芯片开始正常后，Z 和 X，都不应当存在。

X 和 Z，所对应的 Bug，可能有如下几种：

1、IP（包括 Memory、PLL、Serdes 等等）例化时，某些信号悬空未接。也许某些模型允许 Power 信号悬空，或者某些信号是悬空给 DFT 处理（当下给 DFT 处理的信号是接零），但大多数 IP，输入信号是不可悬空的；

2、信号位宽不匹配、信号多驱动、声明的信号名称写错、TB 级互联错误或 TB 中遗漏的 Force（额外小心隐藏的 Force）；

3、后仿真时序不满足时的 X 态传递；

4、功能错误，某些模拟 IP 未能正确操作；

5、功能错误，导致管脚冲突；

6、功能错误，未能合理使用无复位端的寄存器和未初始化的 Memory。

42. 设计描述方式

概述： verilog 通常可以使用三种不同的方式描述模块实现的逻辑功能：**结构化、数据流、行为描述方式。**

结构化描述方式： 是使用实例化低层次模块的方法，即调用其他已经定义过的低层次模块对整个电路的功能进行描述，或者直接调用 Verilog 内部预先定义的基本门级元件描述电路的结构。

数据流描述方式： 是使用连续赋值语句(assign)对电路的逻辑功能进行描述，该方式特别便于对组合逻辑电路建模。

行为级描述方式： 是使用过程块语句结构(always)和比较抽象的高级程序语句对电路的逻辑功能进行描述。

43. 延迟设计

异步电路一般是通过加 buffer、两级与非门等来实现延时，但这是不适合同步电路实现延时的。

在同步电路中，对于比较大的和特殊要求的延时，一般通过高速时钟产生计数器，通过计数器来控制延时；对于比较小的延时，可以通过触发器打一拍，不过这样只能延迟一个时钟周

期。

44. DDR 带宽计算

基本公式：带宽=内存核心频率×内存总线位数×倍增系数

DDR 倍增系数为 2，DDR2 倍增系数为 4，DDR3 倍增系数为 8。

如果直接标为 DDR3-800，其中内存的核心频率只有 100MHz，有效数据传输频率则为 800MHz。

计算时就不管什么核心频率了，如果内存总线位宽为 64bits，

直接 800x64/8，要注意的是，这里除以 8 是为了将 bit 位转为 Byte 字节，而不是因为倍增系数。

二、 Verilog 语法

1. 关键字

关键字	含义
module	模块开始定义
input	输入端口定义
output	输出端口定义
inout	双向端口定义
parameter	信号的参数定义
wire	wire信号定义
reg	reg信号定义
always	产生reg信号语句的关键字
assign	产生wire信号语句的关键字
begin	语句的起始标志
end	语句的结束标志
posedge/negedge	时序电路的标志
case	Case语句起始标记
default	Case语句的默认分支标志
endcase	Case语句结束标记
if	if/else语句标记
else	if/else语句标记
for	for语句标记
endmodule	模块结束定义

2. 运算符

按其功能可分为以下几类:

1) 算术运算符(+,－,×, /,%)
2) 赋值运算符(=,<=)
3) 关系运算符(>,<,>=,<=)
4) 逻辑运算符(&&,||,!)
5) 条件运算符(？：)
6) 位运算符(,|,^,&,^)
7) 移位运算符(<<,>>)
8) 拼接运算符({ })

9) 其它

按其所带操作数的个数运算符可分为三种:

1) 单目运算符(unary operator):可以带一个操作数,操作数放在运算符的右边。
2) 二目运算符(binary operator):可以带二个操作数,操作数放在运算符的两边。
3) 三目运算符(ternary operator):可以带三个操作,这三个操作数用三目运算符分隔开。

见下例:

clock = ~clock; // ~是一个单目取反运算符, clock 是操作数。
c = a | b; // 是一个二目按位或运算符, a 和 b 是操作数。
r = s ? t : u; // ?: 是一个三目条件运算符, s,t,u 是操作数。

优先顺序	
! ~	
* / %	
+ -	
<< >>	最高优先级别
< <= > >=	↓
== !== === !===	↓
&	↓
^ ^~	↓
|	最低优先级别
&&	
||	
? :	

3. 数据类型

Verilog 中共有 19 种数据类型。

基本的四种类型: reg 型、wire 型、integer 型、parameter 型。
其他类型: large 型、medium 型、small 型、scalared 型、time 型、tri 型、trio 型、tril 型、triand 型、trior 型、trireg 型、vectored 型、wand 型和 wor 型。

1) wire 型

wire 型数据常用来表示以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输入、输出信号类型默认为 wire 型。wire 型信号可以用做方程式的输入，也可以用做“assign”语句或者实例元件的输出。

wire 型信号的定义格式如下:

wire [n-1:0] 数据名 1，数据名 2，……数据名 N;

这里，总共定义了 N 条线，每条线的位宽为 n。下面给出几个例子:

wire [9:0] a, b, c; // a, b, c 都是位宽为 10 的 wire 型信号
wire d;

2) reg 型

reg 是寄存器数据类型的关键字。寄存器是数据存储单元的抽象，通过赋值语句可以改变寄存器存储的值，其作用相当于改变触发器存储器的值。**reg 型数据常用来表示 always 模块内的指定信号，代表触发器。通常在设计中要由 always 模块通过使用行为描述语句来表达逻辑关系。在 always 块内被赋值的每一个信号都必须定义为 reg 型，即赋值操作符的右端变量必须是 reg 型。**

reg 型信号的定义格式如下：
reg [n-1:0] 数据名 1，数据名 2，……数据名 N;
这里，总共定义了 N 个寄存器变量，每条线的位宽为 n。下面给出几个例子：
reg [9:0] a, b, c; // a, b, c 都是位宽为 10 的寄存器
reg d;

reg 型数据的缺省值是未知的。reg 型数据可以为正值或负值。但当一个 reg 型数据是一个表达式中的操作数时，它的值被当作无符号值，即正值。如果一个 4 位的 reg 型数据被写入 -1，在表达式中运算时，其值被认为是+15。

reg 型和 wire 型的区别在于：reg 型保持最后一次的赋值，而 wire 型则需要持续的驱动。

3) integer 型

也是一种寄存器数据类型，integer 类型的变量为有符号数，而 reg 类型的变量则为无符号数，除非特别声明为有符号数。
还有就是 integer 的位宽为宿主机的字的位数，但最小为 32 位，用 integer 的变量都可以用 reg 定义，只是用于计数更方便而已。
reg, integer, real，time 都是寄存器数据类型，定义在 Verilog 中用来保存数值的变量，和实际的硬件电路中的寄存器有区别。

4) parameter 型

在 Verilog HDL 中用 parameter 来定义常量，即用 parameter 来定义一个标志符表示一个常数。采用该类型可以提高程序的可读性和可维护性。
parameter 型信号的定义格式如下：
parameter 参数名 1 = 数据名 1;
下面给出几个例子：
parameter s1 = 1;
parameter [3:0] S0=4'h0,
S1=4'h1,
S2=4'h2,
S3=4'h3,
S4=4'h4;

4. 缩位运算

缩减运算符是单目运算符,也有与或非运算。
其与或非运算规则类似于位运算符的与或非运算规则,但其运算过程不同。位运算是对操作数的相应位进行与或非运算,操作数是几位数则运算结果也是几位数。
而缩减运算则不同,缩减运算是对单个操作数进行或与非递推运算,最后的运算结果是一位的二进制数。
缩减运算的具体运算过程是这样的:
1) 第一步先将操作数的第一位与第二位进行或与非运算,
2) 第二步将运算结果与第三位进行或与非运算,
3) 依次类推,直至最后一位。
例如：reg [3:0] B;
reg C;
C = &B;
相当于:
C =((B[0]&B[1]) & B[2]) & B[3];

5. if-else

设计要点
1) 条件语句必须在过程块中使用。所谓过程块语句是指由 initial、always 引导的执行语句集合。除了这两个语句块引导的 begin end 块中可以编写条件语句外，模块中的其他地方都不能编写。
2) if 语句中的表达式一般为逻辑表达式或者关系表达式。系统对表达式的值进行判断；若为 0，z，X；按照假处理；若为 1 按照真处理，执行指定的语句；
3) if（a）等价于 if（a == 1）；
4) if 语句可以·嵌套·使用
5) end 总是与离它最近的一份 else 配对。

如果 if 语句使用不当，没有 else，
可能会综合出来意想不到的锁存器

在 always 块里面，如果在给定的条件下变量没有被赋值，这个变量将会保持原来的值，也就是说会生成一个锁存器。

需要注意的是，这里说的是可能，
因此，不代表没有 else 就一定会出现锁存器，
同时，不代表有 else 就一定不会出现锁存器。
这个是根据具体设计来看的。

6. case

case 语句检查给定的表达式是否与列表中的其他表达式之一相匹配，并据此进行分支。它通常用于实现一个多路复用器。
如果要检查的条件很多，if-else 结构可能不合适，因为它会综合成一个优先编码器而不是多路复用器。

语法

一个 Verilog case 语句以 case 关键字开始，以 endcase 关键字结束。在括弧内的表达式将被精确地评估一次，并按其编写顺序与备选方案列表进行比较，与给定表达式匹配的备选方案的语句将被执行。一块多条语句必须分组，并在 begin 和 end 范围内。

```
// Here 'expression' should match one of the items (item 1,2,3 or 4)
case (<expression>)
    case_item1 :  <single statement>
    case_item2,
    case_item3 :  <single statement>
    case_item4 :  begin
                      <multiple statements>
                  end
    default    : <statement>
endcase
```

如果所有的 case 项都不符合给定的表达式，则执行缺省项内的语句，缺省语句是可选的，在 case 语句中只能有一条缺省语句。case 语句可以嵌套。

如果没有符合表达式的项目，也没有给出缺省语句，执行将不做任何事情就退出 case 块。

避免锁存器

同 if else，case 应当加上 default，以避免锁存器出现。

注意，如果 case 的情况是完备的，可以不加。（完备意为所有情况都设计了）

7. for

在 C 语言中，经常用到 for 循环语句，但在硬件描述语言中 for 语句的使用较 C 语言等软件描述语言有较大的区别。

for 循环会被综合器展开为所有变量情况的执行语句，每个变量独立占用寄存器资源。

简单的说就是：for 语句循环几次，就是将相同的电路复制几次，因此循环次数越多，占用面积越大，综合就越慢。

注意，i 的变化不跟时钟走：

在 Verilog 中使用 for 循环的功能就是，把同一块电路复制多份，完全起不到计数的作用，所以这个 i 的意思是复制多少份你这段代码实现的电路，和时钟没有任何关系。主要是为了提高编码效率。

8. generate

Verilog 中的 generate 语句常用于编写可配置的、可综合的 RTL 的设计结构。它可用于创建模块的多个实例化，或者有条件的实例化代码块。然而，有时候很困惑 generate 的使用方法，因此看下 generate 的几种常用用法。

我们常用 generate 语句做三件事情。一个是用来构造循环结构，用来多次实例化某个模块。一个是构造条件 generate 结构，用来在多个块之间最多选择一个代码块，条件 generate 结构包含 if--generate 结构和 case--generate 形式。还有一个是用来断言。

在 Verilog 中，generate 在建模（elaboration）阶段实施，出现预处理之后，正式模拟仿真之前。因此。generate 结构中的所有表达式都必须是常量表达式，并在建模（elaboration）时确定。例如，generate 结构可能受参数值的影响，但不受动态变量的影响。

generate 循环的语法与 for 循环语句的语法很相似。但是在使用时必须先在 genvar 声明中声明循环中使用的索引变量名，然后才能使用它。genvar 声明的索引变量被用作整数用来判断 generate 循环。genvar 声明可以是 generate 结构的内部或外部区域，并且相同的循环索引变量可以在多个 generate 循环中，只要这些环不嵌套。genvar 只有在建模的时候才会出现，在仿真时就已经消失了。

在“展开”生成循环的每个实例中，将创建一个隐式 localparam，其名称和类型与循环索引变量相同。它的值是“展开”循环的特定实例的“索引”。可以从 RTL 引用此 localparam 以控制生成的代码，甚至可以由分层引用来引用。

Verilog 中 generate 循环中的 generate 块可以命名也可以不命名。如果已命名，则会创建一个 generate 块实例数组。如果未命名，则有些仿真工具会出现警告，因此，最好始终对它们进行命名。

```
1   `timescale 1 ns /1 ps
2
3   module nbit_xor
4   #(parameter SIZE = 16)
5   (
6   input  [SIZE-1 : 0] a,
7   input  [SIZE-1 : 0] b,
8   output [SIZE-1 : 0] y
9   );
10
11  genvar gv_i; // 这这类型的变量只能在generate循环语句中使用
12
13  generate
14      for (gv_i = 0 ; gv_i < SIZE ; gv_i = gv_i +1)
15      begin : label
16          // label用来表示generate循环的实例名称
17          xor u_xor(y[gv_i] , a[gv_i] , b[gv_i]);
18          // 实例化后的结果如下：
19          // label[0].u_xor (y[0] , a[0] , b[0]);
20          // label[1].u_xor (y[1] , a[1] , b[1]);
21          // label[2].u_xor (y[2] , a[2] , b[2]);
22          // ... ...
23          // label[SIZE-1].u_xor (y[SIZE-1] , a[SIZE-1] , b[SIZE-1]);
24          // 实例化后的层次路径如下：
25          // nbit_xor.label[0].u_xor;
26          // ... ...
27          // 同理，还可以引用别的已经定义的module在generate语句中实例化
28      end
29  endgenerate
30
31  endmodule
```

9. 函数 function

function 函数的目的返回一个用于表达式的值。

verilog 中的 function 只能用于组合逻辑;
1 定义函数的语法
function <返回值的类型或范围> <函数名>
<端口说明语句>
<变量类型说明>
begin
<语句>
…
end
endfunction
说明:

```
1   function [7:0] getbyte ;
2   input [15:0] address ;
3       begin
4           <说明语句>                          //从地址字节提取低字节的程序
5           getbyte = result_expression ;   //把结果赋给函数的返回字节
6       end
7   endfunction
```

① <返回值的类型或范围>这一项为可选项，如果缺失，则返回值为一位寄存器类型数据。
② 从函数的返回值：函数的定义蕴含声明了与函数同名、位宽一致的内部寄存器。例子中，getbyte 被赋予的值就是调用函数的返回值。
③ 函数的调用：函数的调用是通过将函数作为表达式中的操作数来实现的。其调用格式：
<函数名> （<表达式> ,…, <表达式>）;
其中函数名作为确认符。下面的例子中，两次调用 getbyte，把两次调用的结果进行位拼接运算，以生成一个字。
word = control ? {getbyte(msbyte),getbyte(lsbyte)} : 8'd0 ;
④ 函数使用的规则
1 函数定义不能包含有任何的时间控制语句，即任何用#、@、wait 来标识的语句。
2 函数不能调用“task”。
3 定义函数时至少要有一个输入参数。
4 在函数的定义中必须有一条赋值语句给函数中与函数名同名、位宽相同的内部寄存器赋值。
5 verilog 中的 function 只能用于组合逻辑;

2 具体实例
函数功能：实现两个 4bit 数的按位“与”运算。
实验现象：如果函数操作正确，则 led 灯闪烁；如果函数操作不正确，则 led 灯常灭。

```
module func_ex_01 (
                    input       clk ,   //E1 25M
                    output      led     //G2   高电平  灯亮
                   );
    ////////////////////*counter_01*//////////////////////
    reg [25:0] counter_01 = 26'd0 ;
```

```
always @ (posedge clk)
begin
    counter_01 <= counter_01 + 1'b1 ;
end
//////////////*& function*/////////////////////////////
function [3:0] yu ;
input [3:0] a ;
input [3:0] b ;
begin
    yu = a & b ;
end
endfunction
reg [3:0] reg_a = 4'b0101 ;
reg [3:0] reg_b = 4'b1010 ;
wire [3:0] result ;
assign result = yu(reg_a , reg_b) ;
/////////////////////////////*verify and display*//////////////////
assign led = (result == 4'd0) ? counter_01[25] : 1'b0 ;
endmodule
```

说明：verilog 中的 function 只能用于组合逻辑；

10. 任务 task

任务就是一段封装在“task-endtask”之间的程序。任务是通过调用来执行的，而且只有在调用时才执行，如果定义了任务，但是在整个过程中都没有调用它，那么这个任务是不会执行的。调用某个任务时可能需要它处理某些数据并返回操作结果，所以任务应当有接收数据的输入端和返回数据的输出端。另外，任务可以彼此调用，而且任务内还可以调用函数。

1．任务定义

任务定义的形式如下：

```
task task_id;
    [declaration]
    procedural_statement
endtask
```

其中，关键词 task 和 endtask 将它们之间的内容标志成一个任务定义，task 标志着一个任务定义结构的开始；task_id 是任务名；可选项 declaration 是端口声明语句和变量声明语句，任务接收输入值和返回输出值就是通过此处声明的端口进行的；procedural_statement 是一段用来完成这个任务操作的过程语句，如果过程语句多于一条，应将其放在语句块内；endtask 为任务定义结构体结束标志。下面给出一个任务定义的实例。

```
：定义一个任务。
task task_demo;                    //任务定义结构开头，命名为 task_demo
    input   [7:0] x,y;             //输入端口说明
    output [7:0] tmp;              //输出端口说明

    if(x>y)                        //给出任务定义的描述语句
```

```
        tmp = x;
  else
     tmp = y;

endtask
```

上述代码定义了一个名为“task_demo”的任务，求取两个数的最大值。在定义任务时，有下列六点需要注意：

（1）在第一行“task”语句中不能列出端口名称；

（2）任务的输入、输出端口和双向端口数量不受限制，甚至可以没有输入、输出以及双向端口。

（3）在任务定义的描述语句中，可以使用出现不可综合操作符合语句（使用最为频繁的就是延迟控制语句），但这样会造成该任务不可综合。

（4）在任务中可以调用其他的任务或函数，也可以调用自身。

（5）在任务定义结构内不能出现 initial 和 always 过程块。

（6）在任务定义中可以出现“disable 中止语句”，将中断正在执行的任务，但其是不可综合的。当任务被中断后，程序流程将返回到调用任务的地方继续向下执行。

2．任务调用

虽然任务中不能出现 initial 语句和 always 语句语句，但任务调用语句可以在 initial 语句和 always 语句中使用，其语法形式如下：

task_id[(端口 1, 端口 2, ..., 端口 N)];

其中 task_id 是要调用的任务名，端口 1、端口 2，…是参数列表。参数列表给出传入任务的数据（进入任务的输入端）和接收返回结果的变量（从任务的输出端接收返回结果）。任务调用语句中，参数列表的顺序必须与任务定义中的端口声明顺序相同。任务调用语句是过程性语句，所以任务调用中接收返回数据的变量必须是寄存器类型。下面给出一个任务调用实例。

例：通过 Verilog HDL 的任务调用实现一个 4 比特全加器。

```
module EXAMPLE (A, B, CIN, S, COUT);

input [3:0] A, B;
input CIN;
output [3:0] S;
output COUT;

reg [3:0] S;
reg COUT;
reg [1:0] S0, S1, S2, S3;

task ADD;

input A, B, CIN;
output [1:0] C;
```

```
reg [1:0] C;
reg S, COUT;

begin

S = A ^ B ^ CIN;
COUT = (A&B) | (A&CIN) | (B&CIN);
C = {COUT, S};
end
endtask

always @(A or B or CIN) begin
ADD (A[0], B[0], CIN, S0);
ADD (A[1], B[1], S0[1], S1);
ADD (A[2], B[2], S1[1], S2);
ADD (A[3], B[3], S2[1], S3);
S = {S3[0], S2[0], S1[0], S0[0]};
COUT = S3[1];
end
endmodule
```

在调用任务时，需要注意以下几点：
（1）任务调用语句只能出现在过程块内；
（2）任务调用语句和一条普通的行为描述语句的处理方法一致；
（3）当被调用输入、输出或双向端口时，任务调用语句必须包含端口名列表，且信号端口顺序和类型必须和任务定义结构中的顺序和类型一致。需要说明的是，任务的输出端口必须和寄存器类型的数据变量对应。
（4）可综合任务只能实现组合逻辑，也就是说调用可综合任务的时间为“0”。而在面向仿真的任务中可以带有时序控制，如时延，因此面向仿真的任务的调用时间不为“0”。

11. 过程块

过程块是行为模型的基础。

过程块有两种：
initial 块，只能执行一次
always 块，循环执行

过程块中有下列部件：
过程赋值语句：在描述过程块中的数据流
高级结构（循环，条件语句）：描述块的功能
时序控制：控制块的执行及块中的语句。

initial 语句与 always 语句和 begin_end 与 fork_join 是一种高频搭配：

1.initial 语句
initial 语句的格式如下：

```
initial
    begin
        语句 1;
        语句 2;
        ……
        语句 n;
end
```

举例说明：

[例 1]：

```
initial
    begin
        areg=0; //初始化寄存器 areg
        for(index=0;index<size;index=index+1)
            memory[index]=0; //初始化一个 memory
end
```

在这个例子中用 initial 语句在仿真开始时对各变量进行初始化。

[例 2]：

```
initial
    begin
        inputs = 'b000000; //初始时刻为 0
        #10 inputs = 'b011001;
        #10 inputs = 'b011011;
        #10 inputs = 'b011000;
        #10 inputs = 'b001000;
end
```

从这个例子中，我们可以看到 initial 语句的另一用途，即用 initial 语句来生成激励波形作为电路的测试仿真信号。一个模块中可以有多个 initial 块，它们都是并行运行的。

initial 块常用于测试文件和虚拟模块的编写，用来产生仿真测试信号和设置信号记录等仿真环境。

2.always 语句
always 语句在仿真过程中是不断重复执行的。

其声明格式如下：
always <时序控制> <语句>
always 语句由于其不断重复执行的特性，只有和一定的时序控制结合在一起才有用。如果一个 always 语句没有时序控制，则这个 always 语句将会发成一个仿真死锁。见下例：

[例 1]:
always areg = ~areg;
这个 always 语句将会生成一个 0 延迟的无限循环跳变过程，这时会发生仿真死锁。如果加上时序控制，则这个 always 语句将变为一条非常有用的描述语句。见下例:

[例 2]:
always #10 areg = ~areg;
这个例子生成了一个周期为 20 的无限延续的信号波形，常用这种方法来描述时钟信号，作为激励信号来测试所设计的电路。

[例 3]:
```
reg[7:0] counter;
reg tick;

always @(posedge areg)
    begin
        tick = ~tick;
        counter = counter + 1;
    end
```
这个例子中,每当 areg 信号的上升沿出现时把 tick 信号反相，并且把 counter 增加 1。这种时间控制是 always 语句最常用的。

always 的时间控制可以是沿触发也可以是电平触发的，可以单个信号也可以多个信号，中间需要用关键字 or 连接，如:

```
always @(posedge clock or posedge reset) //由两个沿触发的 always 块
begin
……
end
always @( a or b or c ) //由多个电平触发的 always 块
begin
……
end
```
沿触发的 always 块常常描述时序逻辑，如果符合可综合风格要求可用综合工具自动转换为表示时序逻辑的寄存器组和门级逻辑，而电平触发的 always 块常常用来描述组合逻辑和带锁存器的组合逻辑，如果符合可综合风格要求可转换为表示组合逻辑的门级逻辑或带锁存器的组合逻辑。一个模块中可以有多个 always 块，它们都是并行运行的。

always 是一个极高频的语法，always@（）用法总结如下
① always@(信号名)
• 信号名有变化就触发事件
例:
```
always@( clock)
a=b;
```

② always@(posedge 信号名)
• 信号名有上升沿就触发事件
例:
always@(posedge clock)
a=b;

③ always@(negedge 信号名)
• 信号名有下降沿就触发事件
例:
always@(negedge clock)
a=b;

④ always@(敏感事件 1or 敏感事件 2or…)
• 敏感事件之一触发事件
• 没有其它组合触发
例:
always@(posedge reset or posedge clear)
reg_out=0;

⑤ always@(*)
• 无敏感列表，描述组合逻辑，和 assign 语句是有区别的
例:
always@(*)
b= 1'b0;
assign 赋值语句和 always@(*)语句。两者之间的差别有：

1.被 assign 赋值的信号定义为 wire 型，被 always@(*)结构块下的信号定义为 reg 型，值得注意的是，这里的 reg 并不是一个真正的触发器，只有敏感列表为上升沿触发的写法才会综合为触发器，在仿真时才具有触发器的特性。

2.另外一个区别则是更细微的差别：举个例子，
wire a;
reg b;
assign a = 1'b0;
always@(*)
b= 1'b0;
在这种情况下，做仿真时 a 将会正常为 0，但是 b 却是不定态。这是为什么？
verilog 规定，always@(*)中的*是指该 always 块内的所有输入信号的变化为敏感列表，也就是仿真时只有当 always@(*)块内的输入信号产生变化，该块内描述的信号才会产生变化，而像 always@(*)b = 1'b0，敏感源是什么呢？1'b0，它始终是不会发生变化的。

12. 位宽计算

N 位二进制数所能表示的数据范围

有符号数（补码）： -2^(N-1) ~ 2^(N-1)-1
*正数范围-1，是因为 0 算在正数范围中
如，N = 8，则表示范围是：-128 ~ 127.

无符号数：0~2^N-1
如，N = 8，则表示范围是：0~255.

有符号定点数：
如 N = 8, 3Q5 ： 意思为共 8 位，3 位整数位 5 位小数位。
如果有符号，那么整数位被占去一位。
所以整数部分其实只有 2 位，最大为 4。
小数位有 5 位，那么最大值为：
4 + ∑ 2^(-i) (i = 1···5) - 2^(-5) ;
整数 + 所有小数位为 1 时的值 - 精度（最小的小数值）

为什么最大值要减 1 个精度，同样是因为 0 占掉了一个范围。
最小值则为 -4 - ∑ 2^(-i) (i = 1···5)

N bit 数和 M bit 数相加、相乘后需要多少 bit?
相加相乘后需要的数据位宽，若无已知数据范围，按照基本规律：
相加位宽+1，相乘位宽为 N+M（保证宽度足够）
若已知操作数范围，根据运算操作数所能表示数的绝对值最大值，求出运算结果极限值。

例如，两个 8bit 有符号数相乘，其结果需要的位宽是多少?
8bit 有符号数补码所能表示的数据范围是：-128 到 127，具有最大绝对值的数是-128，所以极限情况下是-128*（-128） = 16384 = 2^14.
15bit 有符号数所能表示范围是：-2^14 到 2^14-1，并不能表示 2^14。综上，需要 16bit 位宽。
正好位宽扩大一倍，也就是 8+8。

13. 阻塞/非阻塞

阻塞赋值
阻塞赋值使用的赋值运算符为“=”。阻塞赋值的过程是立刻执行的，即阻塞赋值运算符右侧表达式求值完后立刻会更新至运算符左侧，并且这个执行的过程不受其他语句执行的影响，其后的语句只有当前的赋值操作执行完成后才能顺序执行。

非阻塞赋值
非阻塞赋值使用的赋值运算符为“<=”。非阻塞赋值执行过程为：在当前仿真时间槽（time-slot）开始分析计算获得右侧表达式的值，在当前时间槽执行结束时更新左侧表达式的值，在右侧表达式分析计算和左侧表达式被更新之间，任何其他事件都可以执行，同时也有可能修改已经计算完成的右侧表达式的值，即非阻塞赋值的过程不影响其他语句的执行.

根据非阻塞赋值的特点,其赋值运算符左侧操作数只能为寄存器类型,因此非阻塞赋值只能用于过程性语句中（initial 和 always），不允许在连续赋值语句中使用非阻塞赋值。

所谓阻塞的概念是指在同一个 always 块中，其后面的赋值语句从概念上是在前一条赋值语句结束后开始赋值的。

Verilog 模块编程的 8 个原则：

(1) 时序电路建模时，用非阻塞赋值。
(2) 锁存器电路建模时，用非阻塞赋值。
(3) 用 always 块建立组合逻辑模型时，用阻塞赋值。
(4) 在同一个 always 块中建立时序和组合逻辑电路时，用非阻塞赋值。
(5) 在同一个 always 块中不要既用非阻塞赋值又用阻塞赋值。
(6) 不要在一个以上的 always 块中为同一个变量赋值。
(7) 用$strobe 系统任务来显示用非阻塞赋值的变量值。
(8) 在赋值时不要使用#0 延时。

例题：
下面两段代码中 in、q1、q2 和 q3 的初值分别为 0、1、2、3，那么经历 1 个时钟周期后，左侧 q3 的值和右侧 q3 的值分别变成了（）

```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    q3 = q2;
end
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    q3 <= q2;
end
```

【A】0，0
【B】0，3
【C】2，0
【D】0，2

解析：“=”是阻塞赋值，当 clk 的上升沿到来时，in 的值赋给 q1，然后 q1 的值赋给 q2，然后 q2 的值赋给 q3。最终结果 q3 等于 in 的值，为 0。“<=”是非阻塞赋值，当 clk 的上升沿到来时，in 的值赋给 q1，同时 q1 的值赋给 q2，同时 q2 的值赋给 q3。最终结果 q3 等于 q2 的值，为 2。
正确答案：D

14. 时间尺度

timescale 是 Verilog HDL 中的一种时间尺度预编译指令，它用来定义模块的仿真时的时间单位和时间精度。格式如下：

`timescale<时间单位>/<时间精度>

注意：用于说明仿真时间单位和时间精度的数字只能是 1、10、100，不能为其它的数字。而且，**时间精度不能比时间单位还要大。最多两则一样大。

**比如：下面定义都是对的：
`timescale 1ns/1ps
`timescale 100ns/100ns

下面的定义是错的：
`timescale 1ps/1ns

时间精度就是模块仿真时间和延时的精确程序，比如：定义时间精度为 10ns， 那么时序中所有的延时至多能精确到 10ns，而 8ns 或者 18ns 是不可能做到的。
在编译过程中,timescale 指令影响这一编译器指令后面所有模块中的时延值，直至遇到另一个 timescale 指令 resetall 指令。
在 verilog 中是没有默认 timescale 的，一个没有指定 timescale 的 verilog 模块就有可能错误的继承了前面编译模块的无效 timescale 参数.

15. 存储器设计

Verilog 中提供了两维数组来帮助我们建立内存的行为模型。具体来说，就是可以将内存定义为一个 reg 类型的数组，这个数组中的任何一个单元都可以通过一个下标去访问。这样的数组的定义方式如

数组（内存）定义

```
reg [wordsize : 0] array_name [0 : arraysize];
```

例如：

```
reg [7:0] my_memory[0:255];
```

其中 [7:0] 是内存的宽度（位宽），而 [0:255] 则是 内存的深度（也就是有多少存储单元），其中宽度为 8 位，深度为 256。地址 0 对应着数组中的 0 存储单元。

写操作：如果要存储一个值到某个单元中去，可以这样做：

```
my_memory[address] = data_in;
```

读操作：而如果要从某个单元读出值，可以这么做：

```
data_out = my_memory[address];
```

读取内存中的某一位或者多位
只需要读一位或者多个位，就要麻烦一点，因为 Verilog 不允许读/写一个位。这时，就需要使用一个变量转换一下：
例如：

```
data_out = my_memory[address];
data_out_it_0 = data_out[0];
```

这里首先从一个单元里面读出数据，然后再取出读出的数据的某一位的值

16. 三态门设计

三态指其输出既可以是一般二值逻辑电路，即正常的高电平（逻辑 1）或低电平（逻辑 0），又可以保持特有的高阻抗状态，高阻态相当于隔断状态（电阻很大，相当于开路）。

下面是三态门的 Verilog 代码实现：

```
module Tri(
    input din,
    input en,
    output reg dout
    );

    always @(din or en)
        if (en)
            dout <= din;
        else
            dout <= 1'bz;

// 数据流描述
// assign dout = en ? din : 1'bz;

endmodule
```

17. 可综合

verilog 可综合和不可综合语句

（1）所有综合工具都支持的结构：always，assign，begin，end，case，wire，tri，generate，aupply0，supply1，reg，integer，default，for，function，and，nand，or，nor，xor，xnor，buf，not，bufif0，bufif1，notif0，notif1，if，inout，input，instantitation，module，negedge，posedge，operators，output，parameter。

（2）所有综合工具都不支持的结构：time，defparam，$finish，fork，join，initial，delays，UDP，wait，force。

（3）有些工具支持有些工具不支持的结构：casex，casez，wand，triand，wor，trior，real，disable，forever，arrays，memories，repeat，task，while。

建立可综合模型的原则

要保证 Verilog HDL 赋值语句的可综合性，在建模时应注意以下要点：

（1）不使用 initial。

（2）不使用#10。

（3）不使用循环次数不确定的循环语句，如 forever、while 等。

（4）不使用用户自定义原语（UDP 元件）。

（5）尽量使用同步方式设计电路。

（6）除非是关键路径的设计，一般不采用调用门级元件来描述设计的方法，建议采用行为语句来完成设计。

（7）用 always 过程块描述组合逻辑，应在敏感信号列表中列出所有的输入信号。

（8）所有的内部寄存器都应该能够被复位，在使用 FPGA 实现设计时，应尽量使用器件的全局复位端作为系统总的复位。

（9）对时序逻辑描述和建模，应尽量使用非阻塞赋值方式。对组合逻辑描述和建模，既可以用阻塞赋值，也可以用非阻塞赋值。但在同一个过程块中，最好不要同时用阻塞赋值和非阻塞赋值。

（10）不能在一个以上的 always 过程块中对同一个变量赋值。而对同一个赋值对象不能既使用阻塞式赋值，又使用非阻塞式赋值。

（11）如果不打算把变量推导成锁存器，那么必须在 if 语句或 case 语句的所有条件分支中都对变量明确地赋值。

（12）避免混合使用上升沿和下降沿触发的触发器。

（13）同一个变量的赋值不能受多个时钟控制，也不能受两种不同的时钟条件（或者不同的时钟沿）控制。

（14）避免在 case 语句的分支项中使用 x 值或 z 值。

不可综合 verilog 语句

1、initial

只能在 test bench 中使用，不能综合。（写了 initial 虽然可以通过综合，但那个不叫“能综合”）

2、events

event 在同步 test bench 时更有用，不能综合。

3、real

不支持 real 数据类型的综合。

4、time

不支持 time 数据类型的综合。

5、force 和 release

不支持 force 和 release 的综合。

6、assign 和 deassign

不支持对 reg 数据类型的 assign 或 deassign 进行综合，支持对 wire 数据类型的 assign 或 deassign 进行综合。

7、fork join

不可综合，可以使用非块语句达到同样的效果。

8、primitives

支持门级原语的综合，不支持非门级原语的综合。

9、table

不支持 UDP 和 table 的综合。

10、敏感列表里同时带有 posedge 和 negedge

如：always @(posedge clk or negedge clk) begin...end

这个 always 块不可综合。

11、同一个 reg 变量被多个 always 块驱动

12、延时

以#开头的延时不可综合成硬件电路延时，综合工具会忽略所有延时代码，但不会报错。

如：a=#10 b;

这里的#10 是用于仿真时的延时，在综合的时候综合工具会忽略它。也就是说，在综合的时候上式等同于 a=b;

13、与 X、Z 的比较

可能会有人喜欢在条件表达式中把数据和 X(或 Z)进行比较，殊不知这是不可综合的，综合工具同样会忽略。所以要确保信号只有两个状态：0 或 1。

18. VHDL 的结构

VHDL 基本结构
实体(Entity)：描述所设计的系统的外部接口信号，定义电路设计中所有的输入和输出端口；
结构体（Architecture）：描述系统内部的结构和行为；
包集合（Package）：存放各设模块能共享的数据类型 、常数和子程序等 ；
配置（Configuration）：指定实体所对应的结构体；
库（Library）：存放已经编译的实体 、结构体 、包集合和配置。

VHDL 基本设计单元结构：程序包说明、实体说明和结构体说明三部分：

19. VHDL:WAIT 语句格式

WAIT FOR
WAIT UNTIL
WAIT ON

20. 原语

原语，即 primitive。不同的厂商，原语不同；同一家的 FPGA，不同型号的芯片，可以也不一样；**原语类似最底层的描述方法。**使用原语的好处，可以直接例化使用，不用定制 IP；即可通过复制原语的语句，然后例化 IP，就可使用。
常见原语：

IBUF 和 IBUFDS（IO）
IBUF 是输入缓存，一般 vivado 会自动给输入信号加上，IBUFDS 是 IBUF 的差分形式，支持低压差分信号（如 LVCMOS、LVDS 等）。在 IBUFDS 中，一个电平接口用两个独特的电平接口（I 和 IB）表示。一个可以认为是主信号，另一个可以认为是从信号。主信号和从信号是同一个逻辑信号，但是相位相反。

IDDR（Input/Output FuncTIons）
被设计用来接收 DDR 数据，避免额外的时序复杂性。

IBUFG 和 IBUFGDS（IO）
IBUFG 即输入全局缓冲，是与专用全局时钟输入管脚相连接的首级全局缓冲。所有从全局时钟管脚输入的信号必须经过 IBUF 元，否则在布局布线时会报错。

21. False-path

总得来说，**FALSE PATH 就是我们在进行时序分析时，不希望工具进行分析的那些路径。一般不需要工具时序分析的路径指的是异步的路径**，异步路径就是指的不同时钟域的路径。

22. 编译预处理

编译预处理是 VerilogHDL 编译系统的一个组成部分，指编译系统会对一些特殊命令进行预处理，然后将预处理结果和源程序一起在进行通常的编译处理。以"`"(反引号)开始的某些标识符是编译预处理语句。在 Verilog HDL 语言编译时，特定的编译指令在整个编译过程中有效(编译过程可跨越多个文件)，直到遇到其他不同的编译程序指令。常用的编译预处理语句如下：

(1)`define,`undef
(2)`include
(3)`timescale
(4)`ifdef,`else.`endif
(5)`default_nettype;
(6)`resetall
(7)`unconnect_drive,`nounconnected-drive;
(8)`celldefine,`endcelldefine

三、 SystemVerilog

1. 数据类型

四值变量：（0、1、x、z）四种状态
四值逻辑类型：integer、reg、logic、reg、net-type（如 wire、tri）；

SV 并不太常用变量类型是 wire(assign 语句中)还是 reg(initial 和 always 语句中)。
logic 用的比较多。可以被连续赋值语句驱动，可用在 assign、initial、always 语句中。
四值变量与二值变量的特性：
四值变量的默认初始值为 x，二值变量的默认初始值为 0，在 initial 中可以直接使用~clk 变成 1，但是如果是 logic，必须设置初值为 0、或者 1.**
将四值变量赋值给二值变量，x 和 z 状态会转变为 0;

二值变量：（0、1）两种状态
二值逻辑类型：byte、shortint、int、longint、bit。

有符号类型：byte、shortint、int、longint、integer。
无符号类型：bit、logic、reg、net-type（如 wire、tri）。

对于转换方式，可以分为隐式转换和显式转换。显式转换又可以分为静态转换和动态转换
静态转换：unsigned'(signed)；注意单引号。
动态转换：$cast(tgt,src)

2. Logic 类型

在 SystemVerilog 中，可以在过去 verilog 中用 reg 型或是 wire 型的地方用 logic 型来代替。

实际上 logic 是对 reg 数据类型的改进，使得它除了作为一个变量之外，还可以被连续赋值、门单元和模块所驱动，显然，logic 是一个更合适的名字。
抛去 inout 类型外，其余所有的变量都可以声明为 logic 类型，不用区分线网和变量 reg 之间的不同点。

值得注意的是，logic 不能有多个结构性的驱动，也就是说，logic 不允许使用多于一次的持续赋值语句和输出端口连接的给同一变量赋值。这是因为没有类似于线网的多重驱动变量的定论。因此，假如你通过这些方式给一个变量赋过值，你将不能再用过程赋值语句给变量赋值。

3. 类 class

在 SystemVerilog 中，class 也是一种类型（type），你可以把类定义在 program、module、package 中，或者在这些块之外的任何地方定义。类可以在程序或者模块中使用。

类可以被声明成一个参数（方向可以是 input、output、inout 或者 ref），此时被拷贝的是这个对象的句柄，而不是这个对象的内容。

4. 结构体

Verilog 中没有结构体，用户常常以相同的字符开始或结尾来命名信号名，以此表示一组相关的信号。
SystemVerilog 中增加了类似 C 语言中的结构体类型，可以方便的表示一组相关的信号。结构体表示如下：

```
struct {
    int             a,b;        //32bit variables
    opcode_t        opcode;     //user-defined type
    logic[23:0]  address;    //24-bit variable
    bit             error;      //1-bit 2-state var
} Instruction_Word;
```

5. 构造函数

new()的作用有三点：

例化（创建）对象，也就是申请新的内存块来保存对象的变量
初始化变量（二值→0；四值→x）
返回句柄

class 只有经过了 new()函数才真正开辟了内存，否则只是一个空的、没有实际存在。

6. 动态数组

1：基础
在 run-time 才知道元素个数，在 compile-time 不知道
可以在仿真的时候再确定元素个数
2：表示
data_type name_of_dynamic_array[];
name_of_ dynamic_array = new[number of elements];
实例：int dyn[]; dyn = new[5];dyn.delete();
3：可将固定数组赋值给动态数组，要求是元素个数相同

只要基本数据类型相同，定宽数组和动态数组之间就可以相互赋值，在元素数目相同的情况下，可以把动态数组的值复制到定宽数组。

四、 验证

1. UVM

通用验证方法学（Universal Verification Methodology, UVM）是一个以 SystemVerilog 类

库为主体的验证平台开发框架，验证工程师可以利用其可重用组件构建具有标准化层次结构和接口的功能验证环境。

UVM TEST

UVM TEST 是 UVM 的顶层 Component 组件，主要完成例化和配置顶层 env，并通过 env 调用 sequence 产生激励从而给到 DUT。

一般情况下，会有一个基本的 base_test 用于对 env 的例化以及其他共同的 UVM 元素组件的生成例化。然后其他的 test 则继承该 base_test 并有针对性地配置 env 或者选择不同的 sequence 来测试。

UVM Environment

UVM env 主要用于将 UVM 组件进行有关联的分层。一般在 env 里例化 agent、scoreboard，设置是其他的 env，顶层的 env 用于封装用来对 DUT 测试的环境。

UVM Scoreboard

Scb 主要用于检查 DUT 的行为功能是不是符合预期。通过 agent 的 analysis port 来接收 DUT 的输入和输出的 transaction。将输入 transaction 灌入到 reference model 来产生期望的结果，然后与 DUT 实际的输出结果作比较。

UVM Agent

agent 用于对一些 UVM 组件进行分层和连接，主要用于完成 DUT 的接口。

一个典型的 agent 包括一个用于管理激励序列的 sequencer，一个用于施加激励到 DUT 接口的 driver，以及一个用于监测 DUT 接口的 monitor，另外还可能包括一些如覆盖率收集、协议检查等。

agent 可以工作在 active 和 passive 两种模式下，前一种用于产生激励，后一种则只是监测接口并不具备控制能力。

UVM Sequencer

sequencer 用于从 sequence 中选择控制 transaction 序列，类似一个仲裁器。简单说主要用于控制 transaction。

UVM Sequence

sequence 是用于产生激励的 object。

UVM Driver

driver 从 sequencer 中获取 transaction 的 sequence 并将其驱动到 DUT 的接口上，这里使用 TLM 端口来完成。这里涉及到抽象层数据转换成具体的端口信号。

UVM Monitor

monitor 采样 DUT 接口信号并将其送到验证平台的其他组件来作分析。类似 driver，同样涉及到抽象数据和具体数据的转换。为了实现上述功能，通常 monitor 有用于访问 DUT 的接口并且通过 TLM analysis port 来广播产生的 transaction。

monitor 可以在内部对产生的事务做一些处理，比如覆盖率收集、检查、日志记录等，也可以将这些事务的处理由一个专用的组件来完成。

2. 功能覆盖率/代码覆盖率

基于覆盖率驱动的验证技术

采用覆盖率驱动的验证方式可以量化验证进度，保证验证的完备性。一般在验证计划中会指定具体的覆盖率目标。通过覆盖率验证可以确定验证是否达到要求。当然，达到目标覆盖率并不意味着验证就通过了，因为功能覆盖率是由人为定义的，有时候即便达到 100%，也未必将所有的功能场景全部覆盖了，因为人为主观定义的功能场景有时候可能存在遗漏，所以还需要对测试用例进行迭代。

代码覆盖率与功能覆盖率

代码覆盖率：工具会自动搜集已经编写好的代码，常见的代码覆盖率如下：

1) 行覆盖率（line coverage）：记录程序的各行代码被执行的情况。
2) 条件覆盖率（condition coverage）：记录各个条件中的逻辑操作数被覆盖的情况。
3) 跳转覆盖率（toggle coverage）：记录单 bit 信号变量的值为 0/1 跳转情况，如从 0 到 1，或者从 1 到 0 的跳转。
4) 分支覆盖率（branch coverage）：又称路径覆盖率（path coverage），指在 if，case，for，forever，while 等语句中各个分支的执行情况。
5) 状态机覆盖率（FSM coverage）：用来记录状态机的各种状态被进入的次数以及状态之间的跳转情况。
6) 功能覆盖率：`是一种用户定义的度量，主要是衡量设计所实现的各项功能，是否按预想的行为执行，即是否符合设计说明书的功能点要求，功能覆盖率主要有两种如下所示：
7) 面向数据的覆盖率（Data-oriented Coverage）-对已进行的数据组合检查.我们可以通过编写覆盖组（coverage groups）、覆盖点（coverage points）和交叉覆盖（cross coverage）获得面向数据的覆盖率.
8) 面向控制的覆盖率（Control-oriented Coverage）-检查行为序列（sequences of behaviors）是否已经发生.通过编写 SVA 来获得断言覆盖率(assertion coverage).

需要指出的是：**代码覆盖率达到要求并不意味着功能覆盖率也达到要求，二者无必然的联系。而为了保证验证的完备性，在收集覆盖率时，要求代码覆盖率和功能覆盖率同时达到要求。**

功能覆盖率建模

功能覆盖率主要关注设计的输入、输出和内部状态，通常以如下方式描述信号的采样要求；

1) 对于输入，它检测数据端的输入和命令组合类型，以及控制信号与数据传输的组合情况。
2) 对于输出，它检测是否有完整的数据传输类别，以及各种情况的反馈时序。
3) 对于内部设计，需要检查的信号与验证计划中需要覆盖的功能点相对应。通过对信号的单一覆盖、交叉覆盖或时序覆盖来检查功能是否被触发，以及执行是否正确。

3. 断言

断言是对设计违例的一种严查，能够在违例时立刻报出错误。

断言的优势：

1. 断言能够缩短你的开发时间，断言的代码是比较简单的，相比 systemverilog 能够很好的

处理信号的电平和边沿变化的检测。

2. 断言是可以观测的，能够通过断言直接看到是哪里的 bug；
3. 断言能够提供覆盖率收集；

添加在哪里？

1.rtl 设计中，内部 module，如非法状态转换、死锁、fifo 等
2.module 接口 intrerace 中，内部模型接口中，如在 req 是 0 时 ack 不能是 1。
3.芯片功能断言
4.用来检测芯片接口违例的断言，如独立的 PCIX 和 AXI 接口等
5.用来性能的断言，如读响应的最大时延不能超过 5 个 clk；

断言的类型

1.立即断言（不支持 formal 验证）
用来检测待测设计的信号值，可以理解为设计中的 if。

2.并发断言
存在 edge 采样的检测信号的断言

4. 约束

虽然我们可以通过仿真器生成输入激励，但有时候我们不需要完全随机的值。
一般会在测试设计时**考虑设计规范的边界处**，甚至测试设计规范之外的行为。

权重分布的约束（注意两种写法的区别）

：=取权重

```
class bus
    ....
    constrain data_c {
        data dist {0:=10,[1:3]:=80};
        //data = 0,权重为 10/250,250=10+80*3;
        //data = 1/2/3 的权重均为 80/250;
    }
endclasss
```

：/取权重

```
class bus
    ....
    constrain data_c {
        data dist {0:/10,[1:3]:/90};
        //data = 0,权重为 10/100;
        //data = 1/2/3 的权重均为 30/100;
    }
endclasss
```

5. 验证透明度-黑/白/灰盒验证

黑盒测试

应用场合：对设计细节缺乏认识。

该部分测试只需要将激励给入设计的外部接口，检查设计输出，建立参考模型，无需关心设计本身。但是该测试透明度较低，导致无法深层次地定位问题，从而解决一些较深的缺陷。

白盒测试

应用场合：了解内部工作逻辑、层次，信号等。

无需参考模型，只需要植入监视器和断言来检查各个内部逻辑，从而对底层的设计细节进行测试。但是该测试专注于设计内部逻辑检查而忽视整体功能测试，且一旦设计更新，验证文件的可复用性很差。

灰盒测试

实际应用场合中通常将黑百盒两种测试方法结合。其特点是前两种测试折中。

6. 形式验证 Formality

1. 基本特点：

Synopsys Formality 是形式验证的工具，你可以用它来比较一个修改后的设计和它原来的版本，或者一个 RTL 级的设计和它的门级网表在功能上是否一致。

在 IC 设计中通常用于进行不同步骤的 netlist 的比较：逻辑综合 netlist，floorplannetlist，placement netlist ，CTSinserted netlist,P&R netlist，在每一个步骤后都有新的逻辑加入到 netlist 中，但是这个新的逻辑的加入不能改变原 netlist 的 l 逻辑功能。

2. Reference Design 和 Implementation Design：

形式验证的过程中涉及到两个设计：一个是标准的、其逻辑功能符合要求的设计，在 Synopsys 的术语中称之为 Reference Design；另一个是修改后的、其逻辑功能尚待验证的设计，称之为 Implementation Design

3. container：

我们可以把 container 理解为 Formality 用来读入设计的一个空间，或者说一个“集装箱”，一般情况下要建立两个 container 来分别保存 Reference Design 和 Implementation Design

4. 读入共享技术库：

在开始验证流程之前，首先要读入所有的会被用到的共享技术库

5. 设置 Reference Design：

1）建立一个新的 container；

2）读入门级网表；

3）确认该设计为 Reference Design；

4）链接 Reference Design；

6. 设置 Implementation Design：

1）建立一个名为 impl 的 container，读入 clk_insert1.v 文件；

2）确认 Implementation Design；

3）链接该设计；
4）把该设计设置为当前设计，然后把其中的 test_se 端口设置为 0

7. 保存及恢复所做的设置

8. 运行 verify 命令
Formality 将根据所作的设置，对 ref 和 impl 中的两个设计进行验证。

相比于动态仿真的优势在于：
不需要开发验证 pattern
速度比较快
覆盖率 100%
纯逻辑上的验证，不考虑物理和 timing 信息

缺点在于：
由于不考虑 timing，因此需要配合 STA 工具使用。

7. 验证方法学

什么是验证方法学?
验证工程师绕不开的一个基本概念。芯片规模越大功能越复杂，潜在的问题也就越多，验证的难度也就越大.
验证方法学，就是研究怎样降低验证工程复杂度的同时，还能保证验证的可靠性，提升验证效率的一门学问。
从它被提出到今天，其核心依然是带约束的随机激励、覆盖率驱动以及重用，并利用面向对像语言的特性对常用功能进行高度封装，再提供统一的事物层接口使不同抽象层级的建模数据得以共享和有效通信，从而极大地提升验证平台的构建效率，并加速 EDA 仿真，最终对缩短芯片面市周期做出贡献。

8. 测试点

概述
测试点实际上是把设计的功能按层级分解成一个个最简单、最底层的功能点，化繁为简，方便测试用例的实现。测试点主要从功能规格(FS)与架构规格(AS)中提取。

几点原则:
1) 完备性，即不能遗漏任何功能点，特别是异常处理，边界处理，容错处理这些往往容易被忽视;
2) 低耦合，不同测试点之间的相关性越低越好，这也直接决定了分解粒度，并影响 testcase 的开发难度;
3) 无歧义，测试点的描述要直接而明确，不同测试点之间不存在矛盾之处。
4) 扩展性，包含异常和边界特性

为什么需要测试点分解
验证规格分解到特性，粒度比较粗，无法保证完备性，特性的理解会存在歧义，特性和测试

用例的对应关系不确定。

测试点分解的规则

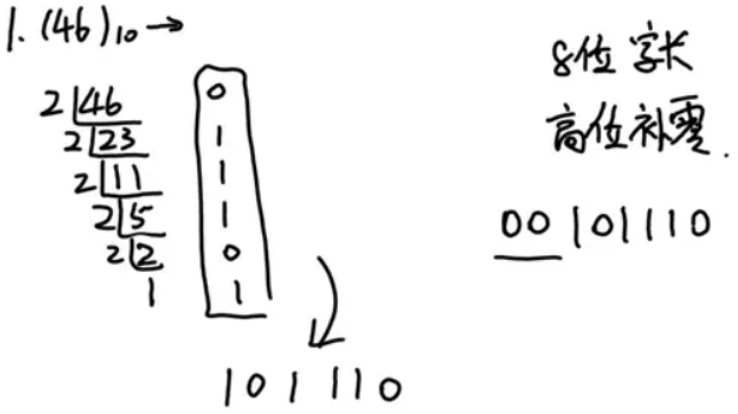
1) 测试点分解的粒度要求细化到无法再细化，保证无歧义，不遗漏，同时兼顾效率不过度验证，
2) 测试点的描述必须明确激励和期望
3) 测试点分解是一个持续的过程，在整个芯片的验证过程中是不断更新迭代补充的。

五、 一些概念

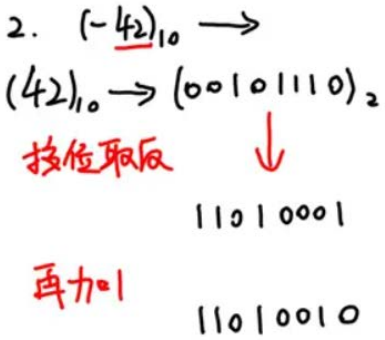
1. 进制转换

10 to 2

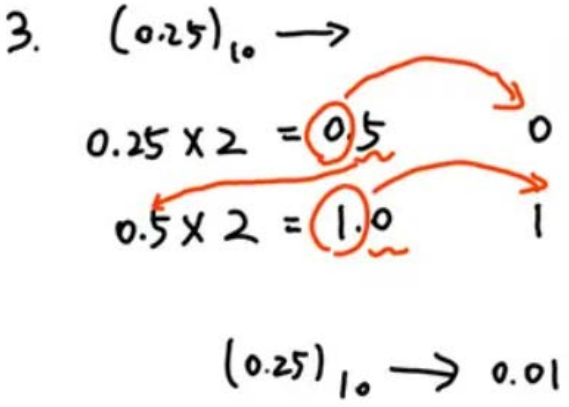
1. 正整数转二进制：除二取余，然后倒序排列，高位补零。



2. 负整数转二进制：先是将对应的正整数转换成二进制后，对二进制取反，然后对结果再加一。



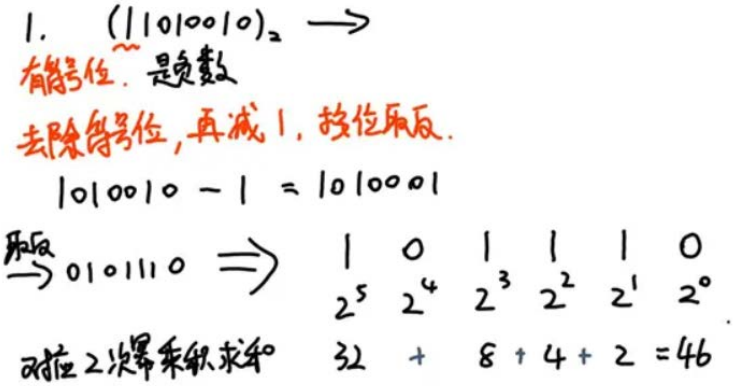
3. 小数转二进制：对小数点以后的数乘以 2，取整，剩余小数再乘以 2，直至小数部分为 0（有可能永远有小数，此时按需停止）



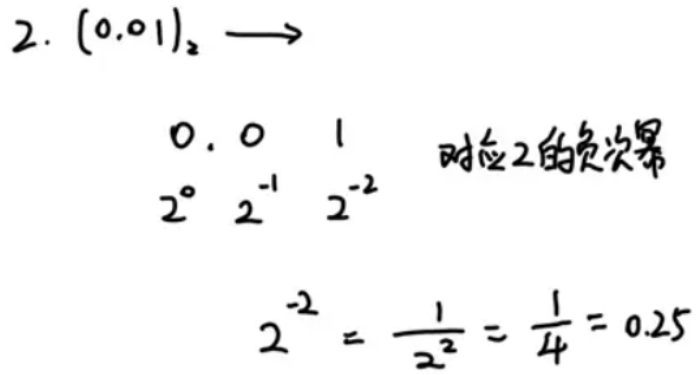
*小数部分和整数部分互不影响，分开转换，最后合在一起即可。

2 to 10

1. 整数二进制转十进制：高位补零（比如数据为 8 位），再确定是有符号还是无符号，若是有符号整数，当最高位是 0 为正数，1 为负数；
正整数，去除符号位，各个位数与对应 2 次幂相乘，最后求和。
负整数，去除符号位，减一，取反，再换算，记得添加整数的负号。



2. 小数二进制十进制：各个位数与对应 2 的负次幂相乘，最后求和。



2. 补码

计算机中的有符号数有三种表示方法，即原码、反码和补码。
三种表示方法均有符号位和数值位两部分，符号位都是用 0 表示“正”，用 1 表示“负”，而数值位，三种表示方法各不相同。
在计算机系统中，数值一律用补码来表示和存储。
原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理

采用补码运算具有如下两个特征：

1）因为使用补码可以将符号位和其他位统一处理，同时，减法也可以按加法来处理，即如果是补码表示的数，不管是加减法都直接用加法运算即可实现。

2）两个用补码表示的数相加时，如果最高位（符号位）有进位，则进位被舍弃。

这样的运算有两个好处：

1）使符号位能与有效值部分一起参加运算，从而简化运算规则。从而可以简化运算器的结构，提高运算速度；（减法运算可以用加法运算表示出来。）

2）加法运算比减法运算更易于实现。使减法运算转换为加法运算，进一步简化计算机中运算器的线路设计。

3. 格雷码

格雷码(Gray code),又叫循环二进制码或反射二进制码。

在数字系统中只能识别 0 和 1，各种数据要转换为二进制代码才能进行处理。
格雷码属于可靠性编码，是一种错误最小化的编码方式，因为，自然二进制码可以直接由数/模转换器转换成模拟信号，但某些情况，例如从十进制的 3 转换成 4 时二进制码的每一位都要变，使数字电路产生很大的尖峰电流脉冲。
而格雷码则没有这一缺点，**它是一种数字排序系统，其中的所有相邻整数在它们的数字表示中只有一个数字不同。它在任意两个相邻的数之间转换时，只有一个数位发生变化。**它大大地减少了由一个状态到下一个状态时逻辑的混淆。
另外由于最大数与最小数之间也仅一个数不同，故通常又叫格雷反射码或循环码。

十进制	二进制	格雷码	十进制	二进制	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

格雷码常用于通信，FIFO 或者 RAM 地址寻址计数器中。

转换:

1) 二进制码->格雷码（编码）：从最右边一位起，依次将每一位与左边一位异或(XOR)，作为对应格雷码该位的值，最左边一位不变(相当于左边是 0);
2) 格雷码-〉二进制码（解码）：从左边第二位起，将每位与左边一位解码后的值异或，作为该位解码后的值（最左边一位依然不变）.

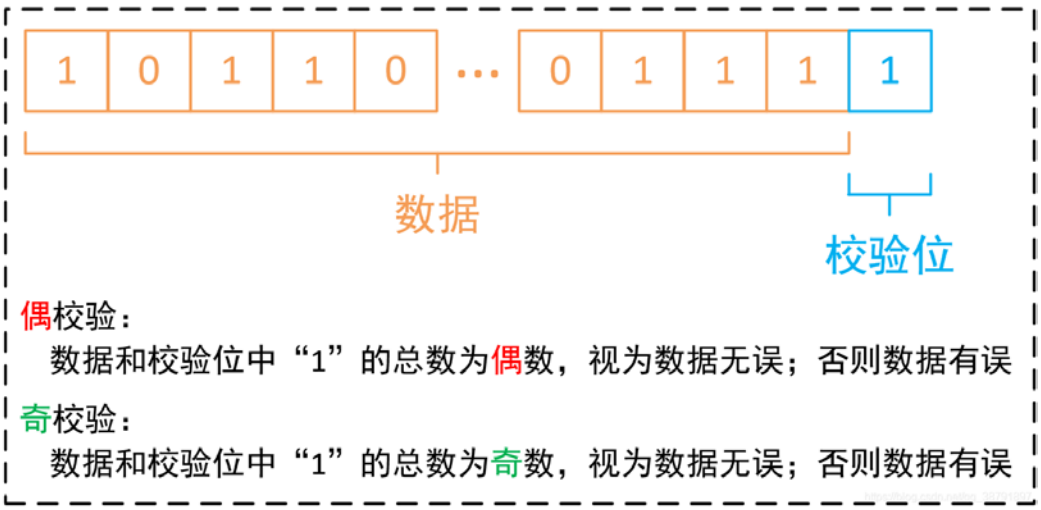
优点：属于压缩状态编码，使用的触发器位数少；相邻状态变换时，仅一位发生改变，电噪声小，转换速度较快;
缺点：译码复杂，没有固定大小，很难直接进行比较大小和算术运算，需要转换为自然二进制码来判断。

4. BCD 码

BCD 码（Binary-Coded Decimal），用 4 位二进制数来表示 1 位十进制中的 0~9 这 10 个数码，是一种二进制的数字编码形式，用二进制编码的十进制代码。

十进制	BCD表达格式
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
10	0 0 0 1 0 0 0 0
11	0 0 0 1 0 0 0 1

5. 奇偶校验



6. 独热码

独热码，在英文文献中称做 one-hot code，直观来说就是有多少个状态就有多少比特，而且只有一个比特为 1，其他全为 0 的一种码制。通常，在通信网络协议栈中，使用八位或者十六位状态的独热码，且系统

例如，有 6 个状态的独热码状态编码为：000001，000010，000100，001000，010000，100000。

优点：状态比较时仅仅需要比较一个位，从而一定程度上简化了译码逻辑，译码简单，减少了毛刺产生的概率。

缺点：速度较慢，触发器资源占用较多，面积较大；

7. 指针

指针相对于一个内存单元来说，指的是单元的地址，该单元的内容里面存放的是数据。

在 C 语言中，允许用指针变量来存放指针，因此，一个指针变量的值就是某个内存单元的地

址或称为某内存单元的指针。
不同类型的指针变量所占用的存储单元长度是相同的，而存放数据的变量因数据的类型不同，所占用的存储空间长度也不同。有了指针以后，不仅可以对数据本身，也可以对存储数据的变量地址进行操作。

8. 链表

定义：一个含有 int 类型和指向 node 类型的指针的数据类型，其中每个指针都指向下一个数据储存的内存地址即可。

对于数组，想要在一列数中插入一个或多个数，就需要使数据依次向后移到下一个下标的空间里，造成程序的运行时间也很大，用链表这种结构就可以很简单的插入和删除数据了。

9. 二叉树

二叉树（Binary tree）是树形结构的一个重要类型。许多实际问题抽象出来的数据结构往往是二叉树形式，即使是一般的树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单，因此二叉树显得特别重要。二叉树特点是每个结点最多只能有两棵子树，且有左右之分。

遍历是对树的一种最基本的运算，所谓遍历二叉树，就是按一定的规则和顺序走遍二叉树的所有结点，使每一个结点都被访问一次，而且只被访问一次。由于二叉树是非线性结构，因此，树的遍历实质上是将二叉树的各个结点转换成为一个线性序列来表示

10. 堆栈

堆栈都是一种数据项按序排列的数据结构，只能在一端(称为栈顶(top))对数据项进行插入和删除。
堆（数据结构）：堆可以被看成是一棵树，如：堆排序。
栈（数据结构）：一种先进后出的数据结构。

11. 反馈电路

反馈，就是在电路系统中，把输出回路中的电量（电压或电流）输入到输入回路中去。
反馈的类型有：电压串联负反馈、电流串联负反馈、电压并联负反馈、电流并联负反馈。
负反馈的优点：降低放大器的增益灵敏度，改变输入电阻和输出电阻，改善放大器的线性和非线性失真，有效地扩展放大器的通频带，自动调节作用。
电压负反馈的特点：电路的输出电压趋向于维持恒定。
电流负反馈的特点：电路的输出电流趋向于维持恒定。

12. 占空比

占空比是指在一个脉冲周期内，高电平时间相对于总时间所占的比例。

13. 面向对象

面向对象是相对于面向过程来讲的，面向对象方法，把相关的数据和方法组织为一个整体来看待，从更高的层次来进行系统建模，更贴近事物的自然运行模式。

三大特性：封装、继承、多态。
封装：是把客观事物抽象成类，并且把自己的属性和方法让可信的类或对象操作，对不可性的隐藏。

继承：是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

多态性（polymorphism）：是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。

14. 流水线

计算机中的流水线是把一个重复的过程分解成为若干个子过程，每个子过程与其他子过程分段展开。由于这种工作方式与工厂中的生产流水线十分相似，因此称作流水线技术。

通常，可以从两个方面来提升处理机内部的并行性，一个是所谓的空间并行性，即在一个处理机内设置多个独立国家的操作者部件，并且使这些部件分段工作。另一个是所谓的时间并行性,就是使用流水线技术。

流水线技术是一种非常经济，对提升计算机的运算速度非常有效地的技术，使用流水线技术需减少少量硬件就能把计算机的运算速度提升几倍，沦为计算机中广泛用于的一种并行处理技术。

将一个指令的执行过程分为多个阶段，一般把一条指令的解释过程分为 3 个（取指，分析，执行）或者 5 个（取值，译码，执行，访存，写回）阶段。

无论是三级还是五级流水线，当出现多周期指令，跳转分支指令和中断发生的时候，流水线都会发生阻塞，而且相邻指令之间也可能因为寄存器冲突导致流水线阻塞，降低流水线的效率。

计算机各个部分几乎都可以使用流水线技术，
如果指令的继续执行过程使用流水线，那么称作指令流水线。
运算器中的操作者部件，如浮点加法器等可以使用流水线，称作操作者部件流水线。
多个计算机之间，通过存储器相连，也可以使用流水线，称作宏流水线。

15. 数字/模拟电路

电子电路中的信号包括模拟信号和数字信号两种。模拟信号是时间连接的信号，如正弦波信号，锯齿波信号等。如图(a)所示即为模拟信号。数字信号是时间和幅度都离散的信号，如产

品数量的统计，数字表盘的读数，数字电路信号等。

（1） 数字电路的特点：在电子设备中，通常把电路分为模拟电路和数字电路两类。其中，用来传输，控制或变换数字信号的电路称为数字电路，数字电路工作时通常只有两种状态，即高电位（又称高电平）或低电位（又称低电平）。通常把高电位用代码“1”表示，称为逻辑“1”；低电位用代码“0”表示称为逻辑“0”。

（2） 模拟电路的特点：
模拟电路的电信号是连接变化的电量，其幅值的大小在一定范围内是任意的，所以要求电路对这种信号不失真地进行放大或处理，因而对元器件及电路参考和外界条件的要求比较严格，例如，放大电路中的半导体器件通常要工作在线性放大状态。

（3） 模拟电路的优缺点
优点：模拟电路可以括放大电路、信号运算和处理电路等，处理模拟信号的电子电路模拟信号，操作方便简单。
缺点：模拟电路的保密性差、抗干扰能力弱。模拟通信很容易被窃听，只要收到模拟信号就可得到通信内容；电信号在沿线路的传输过程中会受到外界的和通信系统内部的各种噪声干扰，噪声和信号混合后难以分开，从而使得通信质量下降。

（4） 数字电路的优缺点
优点：数字电路同时具有算术运算和逻辑运算功能，不像模拟电路那样易受噪声的干扰。数字电路便于计算机处理和高度集成化。
缺点：数字电路中的电流和电压会是脉动变化的，当数字信号采用断续变化的电压或光脉冲来表示时，一般则需要用双绞线、电缆或光纤介质将通信双方连接起来，才能将信号从一个节点传到另一个节点。

16. 数模转换

数模转换器，D/A 转换器，简称 DAC，它是把数字量转变成模拟的器件。D/A 转换器基本上由 4 个部分组成，即权电阻网络、运算放大器、基准电源和模拟开关。

模数转换器，A/D 转换器，简称 ADC，它是把连续的模拟信号转变为离散的数字信号的器件。

17. 可编程逻辑器件

programmable logic device 即 PLD
常见的有 PROM、EPROM、EEPROM、PAL、GAL、CPLD、EPLD、EPLA、FPGA

18. 波特率

波特率表示每秒钟传送的码元符号的个数，是衡量数据传送速率的指标，它用单位时间内载波调制状态改变的次数来表示。

在信息传输通道中，携带数据信息的信号单元叫码元，每秒钟通过信道传输的码元数称为码元传输速率，简称波特率。波特率是传输通道频宽的指标。

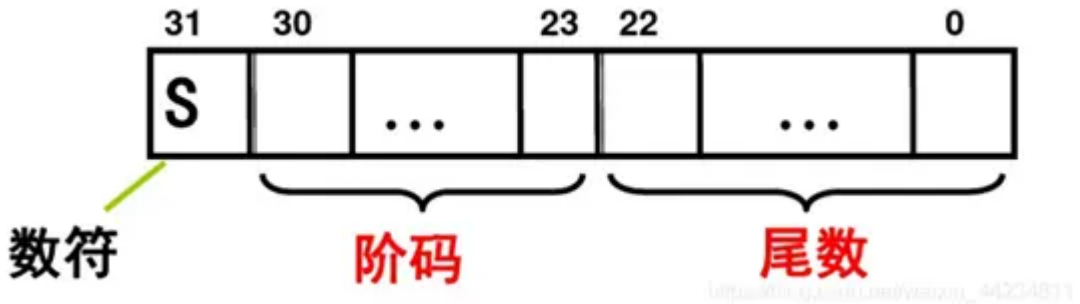
一个数字脉冲就是一个码元，即一个 bit。

19. 差分信号

在高速电路设计中的应用越来越广泛，电路中最关键的信号往往都要采用差分结构设计。何为差分信号？通俗地说，就是驱动端发送两个等值、反相的信号，接收端通过比较这两个电压的差值来判断逻辑状态“0”还是“1”。而承载差分信号的那一对走线就称为差分走线。
差分信号的优点：

差分信号和普通的单端信号走线相比，最明显的优势体现在以下三个方面：
1．抗干扰能力强，因为两根差分走线之间的耦合很好，当外界存在噪声干扰时，几乎是同时被耦合到两条线上，而接收端关心的只是两信号的差值，所以外界的共模噪声可以被完全抵消。
2．能有效抑制 EMI，同样的道理，由于两根信号的极性相反，他们对外辐射的电磁场可以相互抵消，耦合的越紧密，泄放到外界的电磁能量越少。
3．时序定位精确，由于差分信号的开关变化是位于两个信号的交点，而不像普通单端信号依靠高低两个阈值电压判断，因而受工艺，温度的影响小，能降低时序上的误差，同时也更适合于低幅度信号的电路。

20. 32 位浮点数



IEEE 754 规定，对于 32 位的浮点数，最高的 1 位是符号位 s，接着的 8 位是指数 E，剩下的 23 位为有效数字 M。

数符：1 位，即符号位，0 正 1 符，（-1）^S
阶码：8 位，以 2 为底，阶码 = 阶码真值+127
尾数：23 位，默认最高位为 1，实际尾数 24 位，尾数=尾数真值-1

真值：±（1+尾数）*2^(阶码-127)

例3：将$(100.25)_{10}$转换成短浮点数格式。

(1) 十进制数→二进制数 $(100.25)_{10}=(1100100.01)_2$

(2) 非规格化数→规格化数

$1100100.01=1.10010001\times2^6$

(3) 计算移码表示的阶码（偏置值＋阶码真值）

1111111+110=10000101

(4) 以短浮点数格式存储该数。

符号位=0

阶码=10000101

尾数=10010001000000000000000

短浮点数代码为

0,100 0010 1,100 1000 1000 0000 0000 0000

表示为十六进制的代码：42C88000H

六、 计算机体系结构

1. CISC/RISC

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。

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。

2. 冯诺依曼架构/哈佛架构

冯诺依曼架构，也叫普林斯顿架构，其特点是程序空间和数据空间是一体的，数据和程序采用同一数据总线和地址总线。指令和数据地址指向同一个存储器的不同物理位置，指令和数据的宽度相同。由于冯诺依曼架构的指令和数据储存在同一存储器，而且由同一总线进行读写，因而指令和数不能同时进行操作，只能顺序执行。也是这个原因限制了计算机的性能和数据处理速度。
采用冯诺依曼架构的优点是硬件简单，最典型的应用便是 intel 的 x86 微处理器。
冯诺依曼架构芯片采用的是复杂指令集（CISC）。

哈佛架构，其特点则是数据和程序分别存放，存储器分为数据存储器和程序存储器，同时，总线则分为程序存储器的数据总线和地址总线以及数据存储器的数据总线和地址总线。这种总线方式也为同时对数据和程序进行操作提供了可能，因而哈佛结构具有较高的执行效率。同时，由于指令和数据分开存放，因而指令和数据可以有不同的宽度。
目前，采用哈佛架构的芯片以 DSP 和 ARM 为代表。
哈佛架构芯片采用的是精简指令集（RISC）

3. 时间局部性/空间局部性

1）时间局部性
时间局部性是指被引用过一次的内存位置很可能在不远的将来再被多次引用。

2）空间局部性
空间局部性是指如果一个内存位置被引用了一次，那么程序很可能在不远的将来引用其附近的一个内存位置。

3）局部性原理举例
在硬件层，局部性原理允许计算机设计者通过引入小而快的高速缓存存储器来保存最近被引用的指令和数据项，从而提高对主存的访问速度。
在操作系统级，局部性原理允许系统使用主存作为虚拟地址空间最近被引用块的高速缓存。

类似的，操作系统用主存来缓存磁盘文件系统中最近被使用的磁盘块。
在应用程序的设计，如 Web 浏览器将最近被请求的文档放在本地磁盘上，利用的就是时间局部性。大容量的 Web 服务器将最近被请求的文档放在前端磁盘高速缓存中，不需要服务器的干预就可满足对这些文档的请求。

4. 计算机存储结构

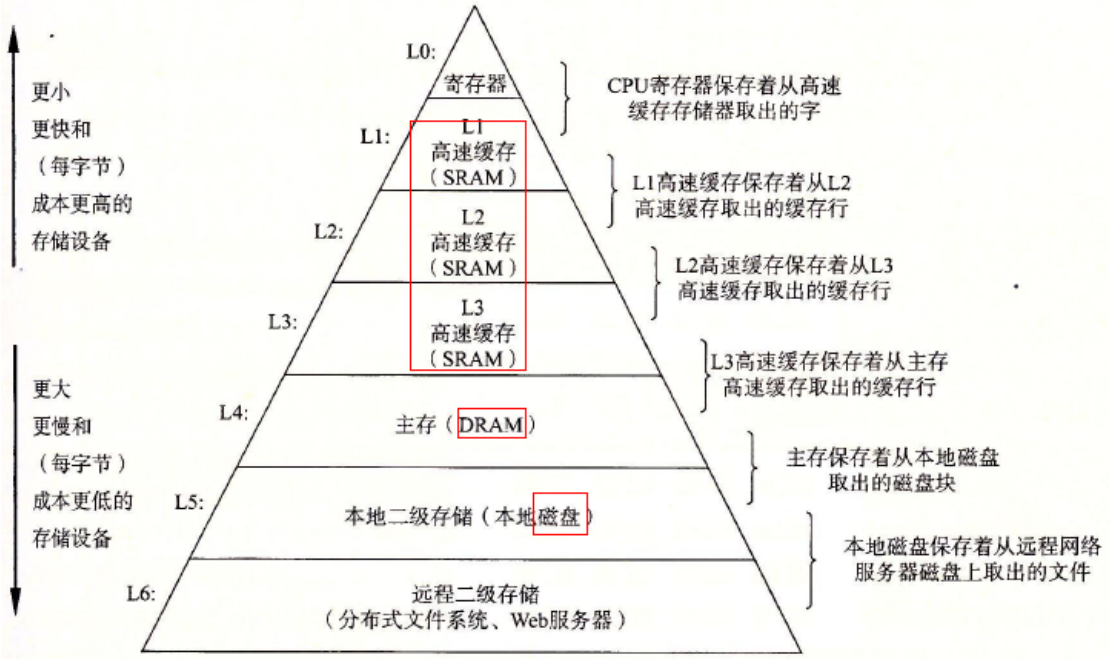


图 6-21 存储器层次结构

5. RAM/SRAM/DRAM/SDRAM/DDR

RAM 随机存储器：存储单元的内容可按需随意取出或存入，且存取的速度与存储单元的位置无关的存储器。这种存储器在断电时将丢失其存储内容，故主要用于存储短时间使用的程序。

SRAM 表示静态态随机存取存储器：因此只要供电它就会保持一个值。一般而言，SRAM 比 DRAM 要快，这是因为 SRAM 没有刷新周期。每个 SRAM 存储单元由 6 个晶体管组成，而 DRAM 存储单元由一个晶体管和一个电容器组成。相比而言，DRAM 比 SRAM 每个存储单元的成本要高。照此推理，可以断定在给定的固定区域内 DRAM 的密度比 SRAM 的密度要大。

DRAM 表示动态随机存取存储器：这是一种以电荷形式进行存储的半导体存储器。DRAM 中的每个存储单元由一个晶体管和一个电容器组成。数据存储在电容器中。电容器会由于漏电而导致电荷丢失，因而 DRAM 器件是不稳定的。为了将数据保存在存储器中，DRAM 器件必须有规律地进行刷新。

SRAM 常常用于高速缓冲存储器，因为它有更高的速率；而 DRAM 常常用于 PC 中的主存储器，因为其拥有更高的密度。

SDRAM 同步动态随机存储器：意思是指理论上其速度可达到与 CPU 同步。

DDR SDRAM(Dual Date Rate SDRAM)：简称 DDR，也就是“双倍速率 SDRAM“的意思。DDR 可以说是 SDRAM 的升级版本，DDR 在时钟信号上升沿与下降沿各传输一次数据，这使得 DDR 的数据传输速度为传统 SDRAM 的两倍。

6. ROM/PROM/EPROM/E2PROM/FLASH

ROM 指的是“只读存储器”，即 Read-Only Memory。这是一种线路最简单半导体电路，通过掩模工艺， 一次性制造，其中的代码与数据将永久保存(除非坏掉)，不能进行修改。

PROM 指的是“可编程只读存储器”既 Programmable Red-Only Memory。这样的产品只允许写入一次，所以也被称为“一次可编程只读存储器”(One Time Progarmming ROM，OTP-ROM)。PROM 在出厂时，存储的内容全为 1，用户可以根据需要将其中的某些单元写入数据 0(部分的 PROM 在出厂时数据全为 0，则用户可以将其中的部分单元写入，以实现对其“编程”的目的。

EPROM 指的是“可擦写可编程只读存储器”，即 Erasable Programmable Read-Only Memory。它的特点是具有可擦除功能，擦除后即可进行再编程，但是缺点是擦除需要使用紫外线照射一定的时间。

EEPROM 指的是“电可擦除可编程只读存储器”，即 Electrically Erasable Programmable Read-Only Memory。它的最大优点是可直接用电信号擦除，也可用电信号写入。
EEPROM 不能取代 RAM 的原因是其工艺复杂，耗费的门电路过多，且重编程时间比较长，同时其有效重编程次数也比较低。

Flash memory 指的是“闪存”，所谓“闪存”，它也是一种非易失性的内存，属于 EEPROM 的改进产品。它的最大特点是必须按块(Block)擦除(每个区块的大小不定，不同厂家的产品有不同的规格)， 而 EEPROM 则可以一次只擦除一个字节(Byte)。

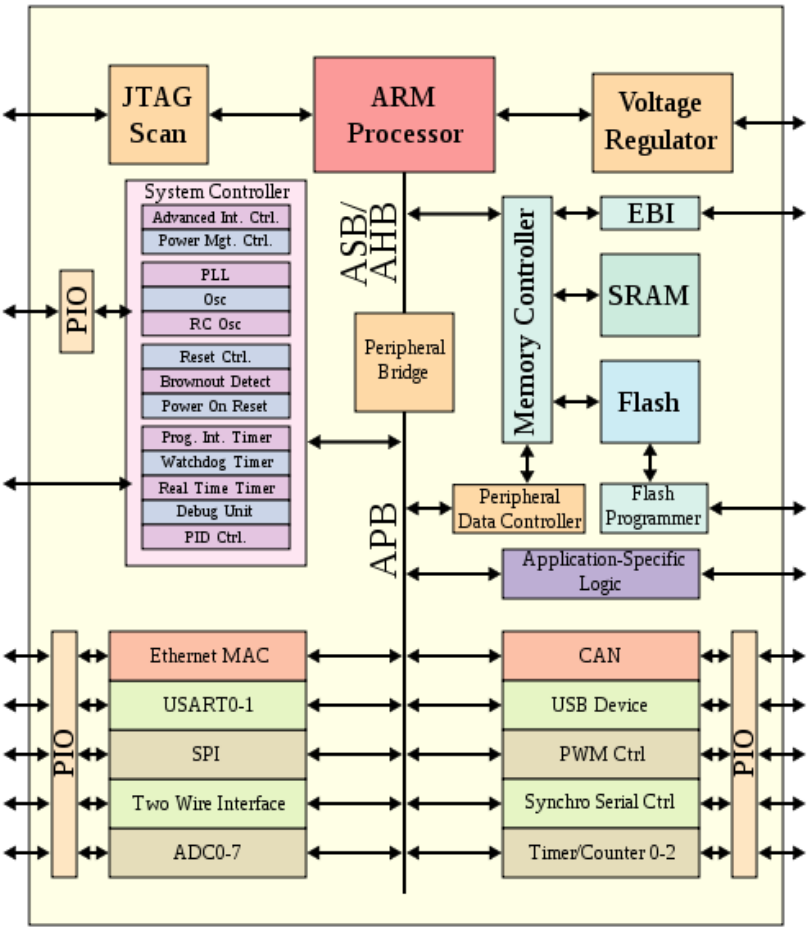
7. SOC 片上系统

System on Chip 的缩写，称为系统级芯片，也有称片上系统。

狭义角度讲，它是信息系统核心的芯片集成，是将系统关键部件集成在一块芯片上；
从广义角度讲，SoC 是一个微小型系统，如果说中央处理器(CPU)是大脑，那么 SoC 就是包括大脑、心脏、眼睛和手的系统。
国内外学术界一般倾向将 SoC 定义为将微处理器、模拟 IP 核、数字 IP 核和存储器(或片外存储控制接口)集成在单一芯片上，它通常是客户定制的，或是面向特定用途的标准产品。

SoC 关键技术主要包括总线架构技术、IP 核可复用技术、软硬件协同设计技术、SoC 验证技术、可测性设计技术、低功耗设计技术、超深亚微米电路实现技术，

SOC 的组成



1)一个微控制器、微处理器或 DSP 核。有些包含不止一个处理器核的 SoC 称为 multiprocessor system on chip (MPSoC)。
2）内存模块，可以是 ROM、RAM、EEPROM 和 flash。
3）时钟源。
3）外设，包括计数器。
4）外部接口，如 USB、FireWire、Ethernet、SPI。
5）数模转换器和模数转换器。
6）电源和电压管理电路。

SOC 特征

并不是包含了微处理器、存储器以及其他外围设备和电路的芯片就是 SoC，就像我们不能将一块 51 单片机称为 SoC。SoC 是建立在 IP 核（具有复杂系统功能且能独立出售的超大规模集成电路块）上的，可以对 IP 核进行复用，以达到快速开发的目的。由于 SoC 芯片的高集成度以及较短的布线，它的功耗也相对低的多。SoC 将多芯片都集成到一起，不需要单独的配置更多芯片，这样更能够有效的降低生产成本，因此使用 SoC 方案成本更低。

SOC 的应用

SoC 的应用十分广泛，最为常见的当属我们日常生活中使用的智能手机。比如苹果 A4 处理器就是基于 ARM 处理器架构的 SoC，它集成基于 45 纳米制程的一颗 ARM Cortex-A8 处理器内核以及一颗 PowerVR SGX 535 图形处理内核。不过，在企业级的服务器和 HPC 等领域，

SoC 并不是最好的选择，但 SoC 会在整个计算设备中挤占传统的 CPU 市场，比如移动端（手机、平板、传感器等等）和低端服务器、存储等设备。

8. 互联

Die 内互联

1) Crossbar
2) Ring
3) Mesh

互联相关模块

1) Access Agent 访问代理：AA 是系统级连接外设子系统和互联总线的通路模块。
2) Decoder 解码器：根据系统的地址分配来决定去哪个节点的调度器。Decoder 在译码侧主要分为三个类别：DAW,MSD,VF，其中 MSD 为 DDR 空间的专门译码空间，支持各种交织算法。

9. 总线

由 ARM 公司推出的 AMBA 片上总线受到了广大 IP 开发商和 SOC 系统集成者的青睐，已成为一种流行的工业标准。

AMBA 4：可以根据应用不同可选 AXI4/AXI4-LITE/AXI4-STREAM。

AXI 协议

AXI 协议包含 5 个相对独立的通道，读操作通过 AR 和 R 通道完成，写操作通过 AW、W 和 B 通道完成。

AXI 所有五个通道均使用相同的 VALID/READY 握手机制来传送数据和控制信息，这种机制使 master 和 slave 可以自由控制信息传输的速率。

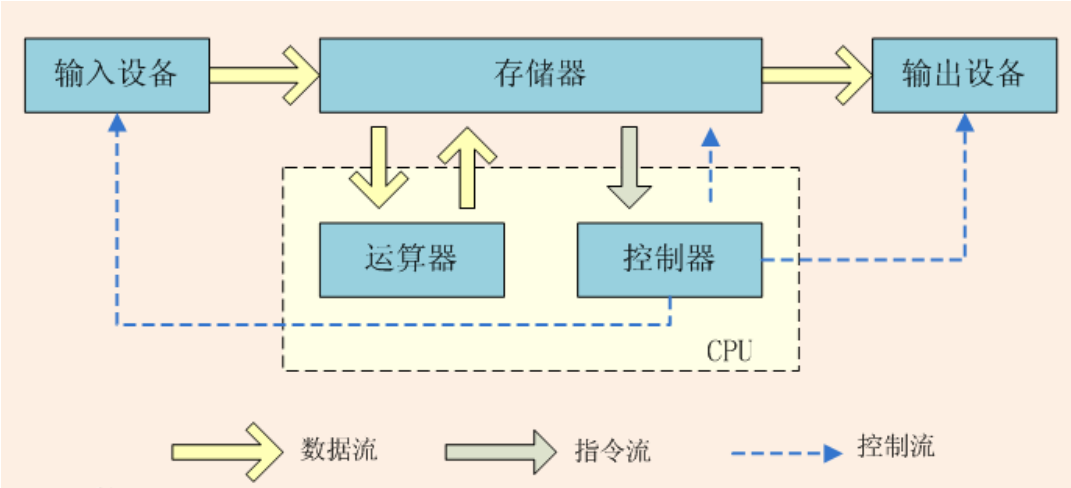
AXI 协议是基于 burst 进行数据串数，一个 burst 可能包含多拍数据，每个拍数据称为一个 transfer 或者 beat。AXI 协议有一个总体限制，就是 burst 操作不能越过 4K 边界，以保证 burst 操作访问的都是同一个 slave。

10. CPU 处理器

冯诺依曼体系结构提出了存储程序的思想。**“存储程序”的意思就是将程序存到计算机内部，计算机自动执行。**

主要特点如下：

1) 计算机处理的数据和指令一律用二进制数表示。
2) 顺序执行程序。
3) 计算机硬件由运算器，控制器，存储器，输入设备和输出设备五大部分组成。
4) 指令集可以分为 CISC 和 RISC 两部分。



11. Cache

Cache 存储器，电脑中为高速缓冲存储器，是位于 CPU 和主存储器 DRAM（Dynamic Random Access Memory）之间，规模较小，但速度很高的存储器，**通常由 SRAM（Static Random Access Memory 静态存储器）组成。**

它是位于 CPU 与内存间的一种容量较小但速度很高的存储器。**CPU 的速度远高于内存，当 CPU 直接从内存中存取数据时要等待一定时间周期，而 Cache 则可以保存 CPU 刚用过或循环使用的一部分数据，如果 CPU 需要再次使用该部分数据时可从 Cache 中直接调用，这样就避免了重复存取数据，减少了 CPU 的等待时间，因而提高了系统的效率。**Cache 又分为 L1Cache（一级缓存）和 L2Cache（二级缓存），L1Cache 主要是集成在 CPU 内部，而 L2Cache 集成在主板上或是 CPU 上。

Cache 原理：局部性原理

空间局部性：在最近的未来要用到的信息（指令和数据），很可能与现在正在使用的信息在存储空间上是邻近的

时间局部性：在最近的未来要用到的信息，很可能是现在正在使用的信息

映射

直接映射

主存中的每一个块只能被放置到 Cache 中唯一的一个位置。

优点：映射方式简单，可以得到比较快的访问速度。

缺点。：效率低。

全相联映射

指主存的任意一块可以映射到 cache 的任意一块。

优点：命中率比较高，cache 存储空间利用率高。

缺点：速度低，成本高。

组相联映射

指主存中的每一块可以被放置到 Cache 中唯一的一个组（相当于直接映射）中的任何一路（相当于全相联映射）

优点：块的冲突率比较低，块的利用率大幅度提高，块的缺失率明显降低。

缺点：实现难度和造价要比直接映射方式高。

Cache 一致性问题：
在多处理器中，不仅 cache 同共享存储器中的同一数据拷贝可能不一致，而且多个处理器异步的独立操作也会使多个 cache 中同一存储块的拷贝不一致。

监听协议
对于被处理器独占的 Cache 中的缓存的内容，该处理器负责监听总线，如果该内容被本处理器改变，则需要通过总线广播；反之，如果该内容状态被其他处理器改变，本处理器的 Cache 从总线收到了通知，则需要相应改变本地备份的状态。

目录协议
需要缓存在 Cache 的内存块被统一存储在一个目录中，目录表统一管理所有的数据，协调一致性问题。该目录表类似于一个仲裁者，当处理器需要把一个数据从内存中加载到自己独占的 Cache 中时，需要向目录表提出申请；当一个内存块被某个处理器改变之后，目录表负责改变其状态，更新其他处理器的 Cache 中的备份，或者使其他处理器的 Cache 的备份无效。

MESI 协议
MESI 中每个缓存行都有四个状态，分别是 E（exclusive）、M（modified）、S（shared）、I（invalid）。下面我们介绍一下这四个状态分别代表什么意思。

M：代表该缓存行中的内容被修改了，并且该缓存行只被缓存在该 CPU 中。这个状态的缓存行中的数据和内存中的不一样，在未来的某个时刻它会被写入到内存中（当其他 CPU 要读取该缓存行的内容时。或者其他 CPU 要修改该缓存对应的内存中的内容时（个人理解 CPU 要修改该内存时先要读取到缓存中再进行修改），这样的话和读取缓存中的内容其实是一个道理）。

E：E 代表该缓存行对应内存中的内容只被该 CPU 缓存，其他 CPU 没有缓存该缓存对应内存行中的内容。这个状态的缓存行中的内容和内存中的内容一致。该缓存可以在任何其他 CPU 读取该缓存对应内存中的内容时变成 S 状态。或者本地处理器写该缓存就会变成 M 状态。

S:该状态意味着数据不止存在本地 CPU 缓存中，还存在别的 CPU 的缓存中。这个状态的数据和内存中的数据是一致的。当有一个 CPU 修改该缓存行对应的内存的内容时会使该缓存行变成 I 状态。

I：代表该缓存行中的内容时无效的。

Cache 性能指标
命中率、失效率、平均访问时间
平均访存时间 = 命中时间 + 失效率×失效开销

12. DDR

DDR 层次结构

1) Channel：也就是通道，简单理解就是 DDRC(DDR 控制器)，一个通道对应一个 DDRC。芯片支持多少个 DDRC 就支持多少个通道。
2) DIMM：双列直插式内存模块。说白了就是内存条/插槽。
3) RANK: 由多个颗粒并联，位宽与通道的数据位宽一样。比如 channel 的数据位宽是 x64，颗粒是 x8 的，那就需要 8 个颗粒组成一个 rank。一个 DIMM 可以多个 rank。
4) CHIP：就是内存条上贴的存储芯片，也叫作颗粒。根据数据位宽，可分为 x4，x8 和 x16 的。
5) BANK：chip 往下拆分就是 bank。
6) ROW&COL：BANK 往下拆分就是一个个的存储单元，横向一排称之为 row，纵向一列称之为 column。

Burst lenth 突发长度：

Burst Lengths，简称 BL，指突发长度，突发是指在同一行中相邻的存储单元连续进行数据传输的方式，连续传输所涉及到存储单元（列）的数量就是突发长度(SDRAM)，在 DDR SDRAM 中指连续传输的周期数。

只要指定起始列地址与突发长度，内存就会依次自动对后面相应数量的存储单元进行读/写操作而不再需要控制器连续地提供列地址。

在 DDR3 SDRAM 时代,内部配置采用了 8n prefetch(预取)来实现高速读写.这也导致了 DDR3 的 Burst Length 一般都是 8。

Prefetch 预取：

所谓 prefetch，就是预加载，这是 DDR 时代提出的技术。在 SDR 中，并没有这一技术，所以其每一个 cell 的存储容量等于 DQ 的宽度（芯片数据 IO 位宽）。

进入 DDR 时代之后，就有了 prefetch 技术，DDR 是两位预取（2-bit Prefetch），有的公司则贴切的称之为 2-n Prefetch（n 代表芯片位宽）。DDR2 是四位预取（4-bit Prefetch），DDR3 和 DDR4 都是八位预取（8-bit Prefetch）。而 8-bit Prefetch 可以使得内核时钟是 DDR 时钟的四分之一，这也是 Prefetch 的根本意义所在。

DDR 关键参数：

tRCD：行寻址到列寻址的延迟时间。

tCL：内存读写操作前列地址控制的潜伏时间。

tRP：内存行地址控制器预充电时间。

13. ARM 体系结构

ARM（Advanced RISC Machines）是一个 32 位 RISC（精简指令集）处理器架构，ARM 处理器则是 ARM 架构下的微处理器。ARM 处理器广泛的使用在许多嵌入式系统。ARM 处理器的特点有指令长度固定，执行效率高，低成本等。

RISC 设计主要特点：

1) 指令集——RISC 减少了指令集的种类，通常一个周期一条指令，采用固定长度的指令格式，编译器或程序员通过几条指令完成一个复杂的操作。而 CISC 指令集的指令长度通常不固定；
2) 流水线——RISC 采用单周期指令，且指令长度固定，便于流水线操作执行；

3) 寄存器——RISC 的处理器拥有更多的通用寄存器，寄存器操作较多。例如 ARM 处理器具有 37 个寄存器；
4) Load/Store 结构——使用加载/存储指令批量从内存中读写数据，提高数据的传输效率；
5) 寻址方式简化，指令长度固定，指令格式和寻址方式种类减少。

Arm 的基本数据类型：

双字节（DoubleWord）：64 位
字（Word）：在 ARM 体系结构中，字的长度为 32 位。
半字（Half-Word）：在 ARM 体系结构中，半字的长度为 16 位。
字节（Byte）：在 ARM 体系结构中，字节的长度为 8 位。

ARM 处理器存储格式：

ARM 体系结构将存储器看作是从 0 地址开始的字节的线性组合。作为 32 位的微处理器，ARM 体系结构所支持的最大寻址空间为 4GB。ARM 体系结构可以用两种方法存储字数据，分别为大端模式和小端模式。

大端模式（高地高低）：字的高字节存储在低地址字节单元中，字的低字节存储在高地址字节单元中。
小端模式（高高低低）：字的高字节存储在高地址字节单元中，字的低字节存储在低地址字节单元中。

14. 虚拟内存

基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分留在外存，就可以启动程序执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存,然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。这样，**系统好像为用户提供了一个比实际内存大得多的存储器，称为虚拟存储器。**

之所以将其称为虚拟存储器，是因为这种存储器实际上并不存在，只是由于系统提供了部分装入、请求调入和置换功能后（对用户完全透明），给用户的感觉是好像存在一个比实际物理内存大得多的存储器。虚拟存储器的大小由计算机的地址结构决定，并非是内存和外存的简单相加。

虚拟存储器有以下三个主要特征：
1) 多次性，是指无需在作业运行时一次性地全部装入内存，而是允许被分成多次调入内存运行。
2) 对换性，是指无需在作业运行时一直常驻内存，而是允许在作业的运行过程中，进行换进和换出。
3) 虚拟性，是指从逻辑上扩充内存的容量，使用户所看到的内存容量，远大于实际的内存容量。

15. 内核 kernel

内核是操作系统最基本的部分，它是为众多应用程序提供对计算机硬件的安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。直接对硬件操作是非常复杂的， 所以内核通常提供一种硬件抽象的方法来完成这些操作。硬件抽象隐藏了复杂性，**为应用软件和硬件提供了一套简洁，统一的接口(为系统调用开发的接口)。**一句话就是对硬件进行管理。

功能：
为系统调用开发的接口：为了开发人员更高效的操作内核,进而操作系统。
程序管理：例如多任务环境,一台计算机可能同一时间需要处理多个任务,需要内核分配好,高效运转。
内存管理：所有需运行的任务,都得放到内存里,内核要处理好。
文件系统管理：I/O 输入/输出设备,还有不同文件格式的支持。
设备驱动：通过加载驱动程序,使计算机与相关硬件连接起来。

16. MCU

微控制单元(Microcontroller Unit；MCU)，又称单片微型计算机(Single Chip Microcomputer)或者单片机，是把中央处理器(Central Process Unit；CPU)的频率与规格做适当缩减，并将内存(memory)、计数器(Timer)、USB、A/D 转换、UART、PLC、DMA 等周边接口，甚至 LCD 驱动电路都整合在单一芯片上，形成芯片级的计算机。
CPU（Central Processing Unit，中央处理器）发展出来三个分枝，一个是 DSP（Digital Signal Processing/Processor，数字信号处理），另外两个是 MCU（Micro Control Unit，微控制器单元）和 MPU（Micro Processor Unit，微处理器单元）。

17. AXI

AXI(Advanced eXtensible Interface)是一种总线协议，该协议是 ARM 公司提出的 AMBA3.0 中最重要的部分，是一种面向高性能、高带宽、低延迟的片内总线。AMBA4.0 将其修改升级为 AXI4.0。

AMBA4.0 包括 AXI4.0、AXI4.0-lite、ACE4.0、AXI4.0-stream。

AXI 协议是基于 burst 的传输，并且定义了以下 5 个独立的传输通道：
读地址通道、读数据通道、写地址通道、写数据通道、写响应通道。

地址通道携带控制消息，用于描述被传输的数据属性；
数据传输使用写通道来实现 master 到 slave 的传输，slave 使用写响应通道来完成一次写传输；
读通道用来实现数据从 slave 到 master 的传输。

AXI 使用基于 VALID/READY 的握手机制数据传输协议，传输源端使用 VALID 表明地址/控制信号、数据是有效的，目的端使用 READY 表明自己能够接受信息。

读/写地址通道：读、写传输每个都有自己的地址通道，对应的地址通道承载着对应传输的

地址控制信息。
读数据通道：读数据通道承载着读数据和读响应信号包括数据总线和指示读传输完成的读响应信号。
写数据通道：写数据通道的数据信息被认为是缓冲（buffered）了的，master 无需等待 slave 对上次写传输的确认即可发起一次新的写传输。写通道包括数据总线（8/16...1024 bit）和字节线（用于指示 8 bit 数据信号的有效性）。
写响应通道：slave 使用写响应通道对写传输进行响应。所有的写传输需要写响应通道的完成信号。

18. IIC

I2C 总线在物理连接上非常简单，分别由 SDA(串行数据线)和 SCL(串行时钟线)及上拉电阻组成。通信原理是通过对 SCL 和 SDA 线高低电平时序的控制，来产生 I2C 总线协议所需要的信号进行数据的传递。在总线空闲状态时，这两根线一般被上面所接的上拉电阻拉高，保持着高电平。

I2C 通信方式为半双工，只有一根 SDA 线，同一时间只可以单向通信。
I2C 总线上的每一个设备都可以作为主设备或者从设备
I2C 总线上的主设备与从设备之间以字节(8 位)为单位进行双向的数据传输。
I2C 协议规定，总线上数据的传输必须以一个起始信号作为开始条件，以一个结束信号作为传输的停止条件。

起始和结束信号总是由主设备产生(意味着从设备不可以主动通信。所有的通信都是主设备发起的，主可以发出询问的 command，然后等待从设备的通信)。

19. SPI

SPI，是一种高速的，全双工，同步的通信总线。
（1）MOSI – 主器件数据输出，从器件数据输入
（2）MISO – 主器件数据输入，从器件数据输出
（3）SCLK –时钟信号，由主器件产生,最大为 fPCLK/2，从模式频率最大为 fCPU/2
（4）NSS – 从器件使能信号，由主器件控制,有的 IC 会标注为 CS(Chip select)

I2C 是多主机总线，通过 SDA 上的地址信息来锁定从设备；
SPI 只有一个主设备，主设备通过 CS 片选来确定从设备。

20. GPIO

GPIO（英语：General-purpose input/output），通用型之输入输出的简称，功能类似 8051 的 P0—P3，其接脚可以供使用者由程控自由使用，PIN 脚依现实考量可作为通用输入（GPI）或通用输出（GPO）或通用输入与输出（GPIO），如当 clk generator, chip select 等。

接口至少有两个寄存器，即“通用 IO 控制寄存器”与“通用 IO 数据寄存器”。
数据寄存器的各位都直接引到芯片外部，而对这种寄存器中每一位的作用，即每一位的信号流通方向，则可以通过控制寄存器中对应位独立的加以设置。

21. JTAG 接口

简介：

JTAG（Joint Test Action Group，联合测试工作组）是一种国际标准测试协议，主要用于芯片内部测试。**标准的 JTAG 接口是 4 线：TMS、TCK、TDI、TDO，分别为模式选择、时钟、数据输入和数据输出线**。

功能：
1.下载器，即下载软件到 FLASH 里。
2. DEBUG，在线进行调试。可以边运行观察硬件的信息，如查看内存。
3. 边界扫描，可以访问芯片内部的信号逻辑状态，还有芯片引脚的状态等等。

详解：
具有 JTAG 口的芯片都有如下 JTAG 引脚定义：
TCK——测试时钟输入；
TDI——测试数据输入，数据通过 TDI 输入 JTAG 口；
TDO——测试数据输出，数据通过 TDO 从 JTAG 口输出；
TMS——测试模式选择，TMS 用来设置 JTAG 口处于某种特定的测试模式。
可选引脚 TRST——测试复位，输入引脚，低电平有效。

22. 指令执行步骤

几乎所有的冯·诺伊曼型计算机的 CPU，其工作都可以分为 5 个阶段：

1.取指令阶段

取指令（Instruction Fetch，IF）阶段是将一条指令从主存中取到指令寄存器的过程。

程序计数器 PC 中的数值，用来指示当前指令在主存中的位置。

2.指令译码阶段

取出指令后，计算机立即进入指令译码（Instruction Decode，ID）阶段。

在指令译码阶段，指令译码器按照预定的指令格式，对取回的指令进行拆分和解释，识别区分出不同的指令类别以及各种获取操作数的方法。

在组合逻辑控制的计算机中，指令译码器对不同的指令操作码产生不同的控制电位，以形成不同的微操作序列；在微程序控制的计算机中，指令译码器用指令操作码来找到执行该指令的微程序的入口，并从此入口开始执行。

3.访存取数阶段

根据指令需要，有可能要访问主存，读取操作数，这样就进入了访存取数（Memory，MEM）阶段。

此阶段的任务是：根据指令地址码，得到操作数在主存中的地址，并从主存中读取该操作数用于运算。

4.执行指令阶段

在取指令和指令译码阶段之后，接着进入执行指令（Execute，EX）阶段。

此阶段的任务是完成指令所规定的各种操作，具体实现指令的功能。为此，CPU 的不同部分被连接起来，以执行所需的操作。

例如，如果要求完成一个加法运算，算术逻辑单元 ALU 将被连接到一组输入和一组输出，输入端提供需要相加的数值，输出端将含有最后的运算结果。

5.结果写回阶段

作为最后一个阶段，结果写回（Writeback，WB）阶段把执行指令阶段的运行结果数据“写回”到某种存储形式：结果数据经常被写到 CPU 的内部寄存器中，以便被后续的指令快速地存取；在有些情况下，结果数据也可被写入相对较慢、但较廉价且容量较大的主存。许多指令还会改变程序状态字寄存器中标志位的状态，这些标志位标识着不同的操作结果，可被用来影响程序的动作。

七、 其它

1. 稳压二极管

稳压二极管，英文名称 Zener diode，又叫齐纳二极管。利用 PN 结反向击穿状态，其电流可在很大范围内变化而电压基本不变的现象，制成的起稳压作用的二极管。

此二极管是一种直到临界反向击穿电压前都具有很高电阻的半导体器件.**在这临界击穿点上，反向电阻降低到一个很小的数值，在这个低阻区中电流增加而电压则保持恒定，**

稳压二极管是根据击穿电压来分档的，因为这种特性，稳压管主要被作为稳压器或电压基准元件使用。

稳压二极管可以串联起来以便在较高的电压上使用，通过串联就可获得更高的稳定电压。

2. 三极管

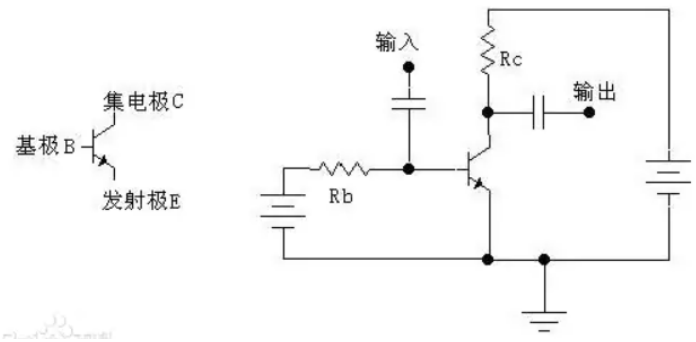
三极管，全称应为半导体三极管，也称双极型晶体管、晶体三极管，是一种电流控制电流的半导体器件.其作用是把微弱信号放大成幅度值较大的电信号，也用作无触点开关。

晶体三极管，是半导体基本元器件之一，具有电流放大作用，是电子电路的核心元件。
三极管是在一块半导体基片上制作两个相距很近的 PN 结，两个 PN 结把整块半导体分成三部分，中间部分是基区，两侧部分是发射区和集电区，**排列方式有 PNP 和 NPN 两种。**

有三个极，分别叫做集电极 C，基极 B，发射极 E。

3. 放大电路

以 NPN 型硅三极管为例，
我们把从基极 B 流至发射极 E 的电流叫做基极电流 Ib;
把从集电极 C 流至发射极 E 的电流叫做集电极电流 Ic。
这两个电流的方向都是流出发射极的，所以发射极 E 上就用了一个箭头来表示电流的方向。



三极管的放大作用：
集电极电流受基极电流的控制(假设电源能够提供给集电极足够大的电流的话)，并且**基极电流很小的变化，会引起集电极电流很大的变化，且变化满足一定的比例关系：**

集电极电流的变化量是基极电流变化量的 β 倍，即电流变化被放大了 β 倍，所以我们把 β 叫做三极管的放大倍数(β 一般远大于 1，例如几十，几百)。
如果我们将一个变化的小信号加到基极跟发射极之间，这就会引起基极电流 Ib 的变化，Ib 的变化被放大后，导致了 Ic 很大的变化。
如果集电极电流 Ic 是流过一个电阻 R 的，那么根据电压计算公式 U=R*I 可以算得，这电阻上电压就会发生很大的变化。
我们将这个电阻上的电压取出来，就得到了放大后的电压信号了。

4. 逻辑门晶体管数量

非门 2 个
与非门 4 个
或非门 4 个
与门 = 与非+非 =6 个
或门 = 或非+非 =6 个

5. FPGA 器件结温范围

商用级结温范围为 0~85 摄氏度，
工业级结温范围是-40~100 摄氏度

6. FPGA 加载方式

粗略可以分为主动和被动两种。
主动加载是指由 FPGA 控制配置流程，被动加载是指 FPGA 仅仅被动接收配置数据。
最常见的被动配置模式就是 JTAG 下载 bit 文件。
主动配置就是 FPGA 主动发起对 Flash 的读写

7. 施密特触发器

施密特触发器有两个稳定状态，但与一般触发器不同的是，施密特触发器采用电位触发方式，其状态由输入信号电位维持；对于负向递减和正向递增两种不同变化方向的输入信号，施密特触发器有不同的阈值电压。

对于标准施密特触发器，当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变，也就是说输出由高电准位翻转为低电准位，或是由低电准位翻转为高电准位时所对应的阈值电压是不同的。只有当输入电压发生足够的变化时，输出才会变化，因此将这种元件命名为触发器。这种双阈值动作被称为迟滞现象，表明施密特触发器有记忆性。**从本质上来说，施密特触发器是一种双稳态多谐振荡器。**

施密特触发器可作为波形整形电路，能将模拟信号波形整形为数字电路能够处理的方波波形，而且由于施密特触发器具有滞回特性，所以可用于抗干扰，其应用包括在开回路配置中用于抗扰，以及在闭回路正回授/负回授配置中**用于实现多谐振荡器**。

8. C 语言结构化编程

结构化程序设计方法
自顶向下；逐步细化；模块化设计；结构化编码;

9. 中断向量地址

中断向量地址，即存储中断向量的存储单元地址，中断服务例行程序入口地址的地址。
在 PC/AT 机中，中断向量是指中断服务程序的入口地址。

10. 寄生效应

所谓寄生效应就是那些溜进你的 PCB 并在电路中大施破坏、令人头痛、原因不明的小故障。**它们就是渗入高速电路中隐藏的寄生电容和寄生电感。其中包括由封装引脚和印制线过长形成的寄生电感；焊盘到地、焊盘到电源平面和焊盘到印制线之间形成的寄生电容；通孔之间的相互影响，以及许多其它可能的寄生效应。**
理想状态下，导线是没有电阻，电容和电感的。而在实际中，导线用到了金属铜，它有一定的电阻率，如果导线足够长，积累的电阻也相当可观。两条平行的导线，如果互相之间有电压差异，就相当于形成了一个平行板电容器（你想象一下）。通电的导线周围会形成磁场（特别是电流变化时），磁场会产生感生电场，会对电子的移动产生影响，可以说每条实际的导线包括元器件的管脚都会产生感生电动势，这也就是寄生电感。
在直流或者低频情况下，这种寄生效应看不太出来。而在交流特别是高频交流条件下，影响就非常巨大了。根据复阻抗公式，电容、电感会在交流情况下会对电流的移动产生巨大阻碍，也就可以折算成阻抗。这种寄生效应很难克服，也难摸到。只能通过优化线路，尽量使用管脚短的 SMT 元器件来减少其影响，要完全消除是不可能的。

11. 上拉电阻的作用

最基本的作用是: **将状态不确定的信号线通过一个电阻将其箝位至高电平(上拉)或低电平(下拉)**

01、提高电路稳定性，避免引起误动作
上拉电阻示例中的按键如果不通过电阻上拉到高电平，那么在上电的瞬间可能就发生误动作，因为在上电瞬间 FPGA 芯片的引脚电平是不确定的，上拉电阻 R12 的存在就保证了其引脚处于高电平状态，而不会发生误动作。

02、提高输出管脚的带载能力
受其他外围电路的影响 FPGA 在输出高电平时能力不足，达不到 VCC 状态，这会影响整个系统的正常工作，上拉电阻的存在就可以使管脚的驱动能力增强。

12. FIR/IIR 滤波器

FIR（有限冲激响应）滤波器：非递归（没有反馈通道），具有线性相位。

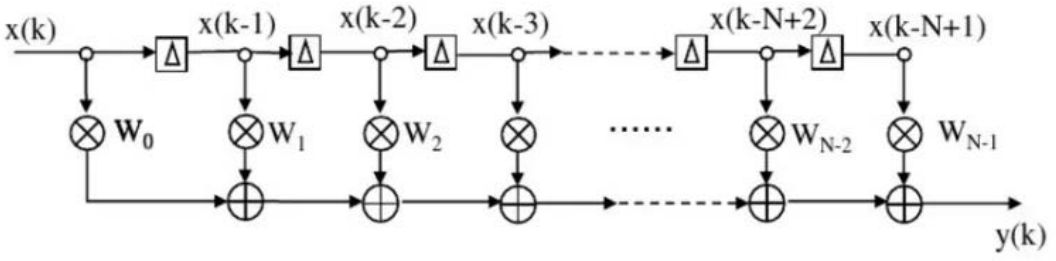
IIR（无限冲激响应）滤波器：递归结构（含反馈通道），非线性相位。

两者的取舍：相同阶数 FIR 和 IIR 滤波器，IIR 滤波器滤波效果较好，但会产生相位失真。FIR 滤波器则性能稳定，但同样幅度指标所需阶数比 IIR 要高 5——10 倍，成本很高。在 DSP 设计中，FIR 比 IIR 需要更多的参数，也就是说需要增加更多的计算量，资源消耗更多的同时也需要更多计算时间，对 DSP 的实时性会有一定的影响。

FIR 滤波器：对 N 个采样数据进行加权和平均（卷积）处理。
其处理过程用下列式表示：

$$y(k)=\sum_{n=0}^{N-1}W_n x(k-n)$$

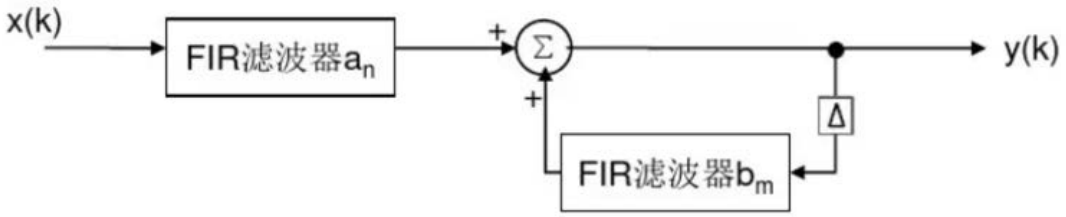
结构图：（图中的三角符号就是延迟的意思）



IIR 滤波器：包含递归部分也包含非递归部分。一个 IIR 滤波器可以看做由两个 FIR 滤波器构成。其中一个滤波器位于反馈回路中，设计 IIR 滤波器的关键就是确保递归部分的稳定。
其处理过程用下列式表示：（具有 N 个前馈系数和 M-1 个反馈系数）

$$y(k)=\sum_{n=0}^{N-1}a_n x(k-n)+\sum_{m=1}^{M-1}b_m y(k-m)$$

结构图：



FIR 滤波器，可能是我在基于 FPGA 的数字信号处理系统需要滤波时，第一时间会想到的东西。因为它非常的方便。关于滤波器的使用，xilinx 官方提供了 FIR Compiler IP 核进行调用。传统的设计方法为，使用 matlab 的 FDATool 工具来设计滤波器，并导出抽头文件，也就是上图中的 W(0)—W(N-1)这些权值，并在 vivado 中调用 FIR IP，配置导入抽头文件。如果是使用 system generator 来设计则会更方便，在 Sysgen 中我们可以直接把 FDATool 和 FIR Compiler 进行一个关联，不需要再手动进行抽头的导出导入就可以完成 FIR 滤波器的设计。

13. 硬核/软核/固核

硬核 (Hard IP Core) :硬核在 EDA 设计领域指经过验证的设计版图；具体在 FPGA 设计中指布局和工艺固定、经过前端和后端验证的设计，设计人员不能对其修改。不能修改的原因有两个：首先是系统设计对各个模块的时序要求很严格，不允许打乱已有的物理版图；其次是保护知识产权的要求，不允许设计人员对其有任何改动。IP 硬核的不许修改特点使其复用有一定的困难，因此只能用于某些特定应用，使用范围较窄。

软核(Soft IP Core)：软核在 EDA 设计领域指的是综合之前的寄存器传输级(RTL) 模型；具体在 FPGA 设计中指的是对电路的硬件语言描述，包括逻辑描述、网表和帮助文档等。软核只经过功能仿真，需要经过综合以及布局布线才能使用。其优点是灵活性高、可移植性强，允许用户自配置；缺点是对模块的预测性较低，在后续设计中存在发生错误的可能性，有一定的设计风险。软核是 IP 核应用最广泛的形式。

固核(Firm IP Core) :固核在 EDA 设计领域指的是带有平面规划信息的网表；具体在 FPGA 设计中可以看做带有布局规划的软核，通常以 RTL 代码和对应具体工艺网表的混合形式提供。将 RTL 描述结合具体标准单元库进行综合优化设计，形成门级网表，再通过布局布线工具即可使用。和软核相比，固核的设计灵活性稍差，但在可靠性上有较大提高。 目前，固核也是 IP 核的主流形式之一。

14. PWM/SPWM

PWM，英文名 Pulse Width Modulation，是脉冲宽度调制缩写，它是通过对一系列脉冲的宽度进行调制，等效出所需要的波形（包含形状以及幅值），对模拟信号电平进行数字编码，也就是说通过调节占空比的变化来调节信号、能量等的变化，**占空比就是指在一个周期内，信号处于高电平的时间占据整个信号周期的百分比**，例如方波的占空比就是 50%.

SPWM，英文名 Sinusoidal PWM，脉冲宽度按正弦规律变化而和正弦波等效的 PWM 波形

15. 大端模式存储

大端模式（Big-endian），是指数据的高字节，保存在内存的低地址中，而数据的低字节，保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

所谓小端模式（Little-endian），是指数据的高字节保存在内存的高地址中,而数据的低字节保存在内在的低地址中,这种存储模式将地址的高低和数据位 权有效结合起来,高地址部分权值高,低地址部分权值低,和我们的逻辑方法一致;

为什么有大小端之分:
因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以由硬件来选择是大

端模式还是小端模式。

16. 斐波那契数列

斐波那契数列指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……
这个数列从第 3 项开始，每一项都等于前两项之和。

17. 傅里叶变换

傅立叶变换，表示能将满足一定条件的某个函数表示成三角函数（正弦和/或余弦函数）或者它们的积分的线性组合。在不同的研究领域，傅立叶变换具有多种不同的变体形式，如连续傅立叶变换和离散傅立叶变换。

18. 奈奎斯特采样定律

奈奎斯特抽样定理指若频带宽度有限的，要从抽样信号中无失真地恢复原信号，抽样频率应大于 2 倍信号最高频率。

抽样频率小于 2 倍频谱最高频率时，信号的频谱有混叠。
抽样频率大于 2 倍频谱最高频率时，信号的频谱无混叠。

19. 基尔霍夫定律

基尔霍夫定律包括电流定律和电压定律：
电流定律：在集总电路中，在任一瞬时，流向某一结点的电流之和恒等于由该结点流出的电流之和。
电压定律：在集总电路中，在任一瞬间，沿电路中的任一回路绕行一周，在该回路上电动势之和恒等于各电阻上的电压降之和。

20. 芯片选型

首先当然是基本功能。功能得满足你的需求。
其次需要考虑性能，例如信噪比、带宽、传输速度、存储容量、工作电压、静态电流损耗等等，取决于你芯片的功能。当然在考虑性能的同时还需要兼顾成本。
此外，还需要考虑到芯片的封装形式、工作温度范围、ESD 防护等级、是否无铅等等因素。
接下来从生产工艺的角度问一问“这种芯片好焊吗？”例如在工艺水平较低的加工厂是无法焊接/检验 BGA 系封装元件的。
接下来从供应链管理的角度问一问“这种芯片好买吗？有稳定可靠的来源渠道吗？”如果千辛万苦选定了一款芯片、却发现走遍世界都买不到货，那就白搭了。