FPGA 研发之道

FPGA 是个什么玩意？

FPGA 是个什么玩意？

首先来说：

FPGA 是一种器件。其英文名 feild programable gate arry 。很长，但不通俗。通俗来说，是一种功能强大似乎无所不能的器件。通常用于通信、网络、图像处理、工业控制等不同领域的器件。就像 ARM、DSP 等嵌入式器件一样，成为无数码农码工们情感倾泻而出的代码真正获得生命的地方。只不过，一样的编程，却是不一样的思想。嵌入式软件人员看到的是 C。而 FPGA 工程师看到是硬件描述语言，verilog 或 VHDL。软件看到是函数、对象、重构。FPGA 工程师则是模块、流水、复用。从现象上看，都是代码到下载程序再到硬件上运行。不能只看现象而忽略本质。FPGA 开发本质上是设计一颗 IC，“**的身子，丫鬟的命”不是所有 verilog/VHDL 代码，都能获得青睐去流片成为真正的芯片，而更多的则成为运行在 FPGA 器件上，成为完成相同功能的替代品。其实现的功能却一点也不逊色于百万身价流片的近亲。从而成为独树一帜的行业。

FPGA 开发的流程，是通过 verilog/VHDL 等硬件描述语言通过 EDA 工具编译、综合、布局布线成为下载文件，最终加载到 FPGA 器件中去，完成所实现的功能。那硬件描述语言描述的是什么？这里描述的就是组合逻辑电路和时序逻辑电路。组合逻辑电路就是大家所熟知的 与门、或门、非门。时序逻辑电路则是触发器。数字芯片上绝大部分逻辑都是这两种逻辑实现的。也就是基本上每个电子行业的人所学过的数字电路。顺便说一下，感谢香农大师，在其硕士毕业论文<继电器与开关电路的符号分析>就奠定了数字电路的的根基。只

不过在 FPGA 中，与或非的操作变成了查找表的操作。于是所有的数字电路变成了查找表和寄存器，这就构成了 FPGA 的基础。查找表负责逻辑实现，寄存器存储电路状态。二者配合，双剑合璧，天衣无缝。这是最初的 FPGA 的雏形。 现代 FPGA 内部出了查找表和寄存器之外，还有 RAM 块，用于存储大量的数据块，这是因为 RAM 块较寄存器来存储大量数据更能节省芯片实现的面积。FPGA 内部的时序电路则需要时钟的输入，通常 FPGA 内部需要时钟种类较多，因此需要在片内产生所需的的相关的时钟，如不同频率，不同相位的时钟，因此时钟管理单元 DCM/PLL 也是必不可少的内部部件。除此之外，FPGA 内部还包括接口 I/O，I/O 分为普通 I/O 和高速 I/O，高速 I/O 支持例如高速的 SERDES，用于实现 XAUI，PCIE 等高速接口，这些接口动辄几 Gbps 到 10Gbps 以上。此外种类多种多样的硬核 IP 也是各 FPGA 厂商差异化竞争利器，例如 POWERPC、ARM 等硬核 IP。从而构成 CPU+FPGA 于一体的集可编程性和可重构的处理平台。因此，相对来所，FPGA 虽然发展有二三十年的历史，其基本架构一直不变不大。

回到问题开始的地方，FPGA 的英文翻译过来是现场可编程门阵列。这是相对 ASIC 来说的，ASIC 的硬件也可看做是门阵列，但是其是非可编程的器件。流片完成其功能就固化了，而 FPGA 的可编程性就在其能够重新下载配置文件，来改变其内在的功能，这就是其可编程性的由来。从前端开发流程来说，FPGA 和 ASIC 开发并无二至。由于 ASIC 开发一次性投入成本较高，FPGA 无疑是一种经济的替代方案，用于实现的高速的数据并行处理。如业务能够支撑大规模应用并且协议固化，则能够分摊成本的 ASIC 实现就有成本的优势。

FPGA 作为一种器件，技术上主要垄断在少数大公司手中，那就是双巨头 ALTERA 和 XILINX。除此之外还有一些份额相对较小的公司，例如 ACTEL 和 LATTICE。不止是 FPGA 的硬件芯片，其配套的 EDA 工具技术壁垒更高。因此相对于 CPU 来说，FPGA 的国产化

更不乐观，不过已经有国内的厂商来从事这一行业，例如国微和京微雅格等，也在一些细分市场上推出自己的 FPGA 产品。

FPGA 和他那些小伙伴们 （一） 系统架构组成

通常来讲，“一个好汉三个帮”，一个完整的嵌入式系统中由单独一个 FPGA 使用的情况较少。通常由多个器件组合完成，例如由一个 FPGA+CPU 来构成。通常为一个 FPGA+ARM，ARM 负责软件配置管理，界面输入外设操作等操作，FPGA 负责大数据量运算，可以看做 CPU 的专用协处理器来使用，也常会用于扩展外部接口。常用的有 ARM+FPGA，DSP+FPGA，或者网络处理器+FPGA 等种种架构形式，这些架构形式构成整个高速嵌入式设备的处理形态。

不得不说的是，随着技术的进步，现在 CPU 中集成的单元也随之增加，例如 TI 的“达芬奇”架构的处理器内部通常由 ARM+DSP 构成。同时异构的处理器形态业逐渐流行，如 ARM9+ARM7 的结构。这类一个主要处理系统(ARM9)外带辅助处理系统(ARM7)的设计，同样成为现在处理器设计的流行方向。主处理系统运行嵌入式操作系统，而辅助处理单元则专注某一些的专用领域的处理。这些系统的应用减少了 FPGA 作为 CPU 协处理单元的领域。因为毕竟 FPGA 相比 ARM 等流行嵌入式处理器价格要相对较高。

在这种情形下，FPGA 的厂商似乎也感受到了压力，不约而同推出了带 ARM 硬核的 FPGA，例如 ALTERA 的 和 XILINX 的 ZYNQ 和 ALTERA 的 SOC FPGA。这是即是互相竞争的需要，也是同众多 CPU 厂商一掰手腕的杰总。即使在这两种在趋势下，经典的处理器+FPGA 的设计仍然可看做为高性能嵌入式系统的典型配置。

经典的处理器+FPGA 的配置中有多种的架构形式，即多个处理器单元,可能是 ARM，MIPS,或者 DSP，FPGA 也可能是多片的配置，具体架构形式于具体处理的业务相关和目标

设备的定位也相关。因为 FPGA 作为简单业务流大数据量的处理形态仍然是 CPU 无可比拟的优势，FPGA 内部可以开发大量业务数据并行，从而实现高速的数据处理。

在实现高速处理方面，CPU 的另一个发展趋势是多核，多核处理器也能处理大数据量的业务的并行，例如业界 TERILA 已推出 64 核的多核处理器，采用 MIPS 处理器，通过二维 MASH 网络连接在一起，形成 NOC 的结构。在性能上已经和现有的高速 FPGA 的处理能力上不相上下。但是多核处理器的不得不说的问题就是，同一业务流分配到多核处理上后，如需交互，例如访问同一资源，就会造成读写的缓存一致的问题，解决的这一问题的天然思路是加锁，即在变量访问上加自旋锁，但是带来的问题就是处理性能的急剧下降。而 FPGA 无论并行处理和同一变量的访问，都可以变成工程师的设计水平的问题，没有原理性的挑战。

没有一种器件可以满足全人类的众多需求，因此不用担心 FPGA 没有用武之地。必定是一系列产品的组合。下面主要介绍一下 FPGA 可以作为现今热门场景的几种应用。

（1）网络存储产品，特别是现在的 NAS，或者 SAN 设备上，其存储的时间、接口、安全性等都要求较高，而 FPGA 无论处理性能还是扩展接口的能力都使其在这一领域大有作为。现在高端 FPGA 单片就可以扩展 32 个或者更多 4G 或者 8G 的 FC 接口。并且其协议处理相对的固定，也使 FPGA 在这一领域有大量的可能应用。

（2）高速网络设备，现在高速网络设备 10G、40/100G 以太网设备领域，同样 FPGA 也是关键的处理部件。特别是 IPv6 的商用化及大数据对于基础设施的高要求，都使这一领域的处理应用会逐渐广泛，这一领域通常是高速网络处理器（NP）+FPGA 的典型架构。

（3）4G 等通信设备，对于新一代通信基站的信号处理，FPGA+DSP 阵列的架构就是绝配。特别是在专用处理芯片面世之前，这样的架构可以保证新一代通信基础设施的迅速研发和部署。

没有完美的架构，只有合适的组合，各种芯片和架构都是为应用服务，互相的渗透是趋势，也是必然。FPGA 相对处理器的可编程领域，仍然属于小众（虽然人数也不少）。但是正像一则笑话所说：大腿虽然比根命根子粗，但决没有命子重要。这算开个玩笑。FPGA 的实现为以后的芯片化留下了许多可能和想象空间，从而在应用大量爆发时通过芯片化来大幅降低成本，这这也正是其他可编程器件所不能比拟的。

FPGA 和他那些小伙伴们 （二） 器件互联

系统架构确定，下一步就是 FPGA 与各组成器件之间互联的问题了。通常来说，CPU 和 FPGA 的互联接口，主要取决两个要素：

（1）CPU 所支持的接口。

（2）交互的业务。

通常来说，FPGA 一般支持与 CPU 连接的数字接口，其常用的有 EMIF，PCI,PCI-E,UPP, 网口（MII/GMII/RGMII），DDR 等接口。作为总线类接口，FPGA 通常作为从设备与 CPU 连接，CPU 作为主设备通过访问直接映射的地址对 FPGA 进行访问。根据是否有时钟同步，通常总线访问分为同步或异步的总线，根据 CPU 外部总线协议有所不同，但数据、地址、控制信号基本是总线访问类型中总线信号所不能省略的。CPU 手册中会对信号定义和时序控制有着详细的说明，FPGA 需要根据这些详细说明来实现相应的逻辑。同时 CPU 还可以对访问时序进行设置，比如最快时钟，甚至所需的最小建立时间和保持时间，这些一般 CPU 都可以进行设置，而这些具体参数，不仅影响 FPGA 的实现，也决定总线访问的速度和效率。对于同步总线，只需要根据输入时钟进行采样处理即可，但对于异步总线，则需要的对进入的控制信号进行同步化处理，通常处理方式是寄存两拍，去掉毛刺。因此用于采样的时钟就与 CPU 所设置的总线参数相关，如采样时钟较低，等控制信号稳定后在译码后输出，

一个总线操作周期的时间就会相对较长，其处理的效率也相对较低；假如采样时钟过快，则对关键路径又是一个挑战，因此合理设定采样频率，便于接口的移植并接口的效率是设计的关键点和平衡点。

对于总线型的访问来说，数据信号通常为三态信号，用于输入和输出。这种设计的目的是为了减少外部连线的数量。因为数据信号相对较多一般为 8/16/32 位数据总线。总线的访问的优势是直接映射到系统的地址区间，访问较为直观。但相对传输速率不高，通常在几十到 100Mbps 以下。这种原因的造成主要为以下因素（1）受制总线访问的间隔，总线操作周期等因素，总线访问间隔即两次访问之间总线空闲的时间，而总线操作周期为从发起到相应的时间。（2）不支持双向传输，并且 FPGA 需主动发起对 CPU 操作时，一般只有发起 CPU 的中断处理一种方式。这种总线型操作特点，使其可以用作系统的管理操作，例如 FPGA 内部寄存器配置，运行过程中所需参数配置，以及数据流量较小的信息交互等操作。这些操作数据量和所需带宽适中，可以应对普通的嵌入式系统的处理需求。

对于大数据流量的数据交互，一般采用专用的总线交互，其特点是，支持双向传输，总线传输速率较快，例如 GMII/RGMII、Upp、专用 LVDS 接口，及 SERDES 接口。专用 SERDES 接口一般支持的有 PCI-E，XAUI，SGMII，SATA，Interlaken 接口等接口。GMII/RGMII，专用 LVDS 接口一般处理在 1GbpS 一下的业务形式，而 PCI-E，根据其型号不同，支持几 Gbps 的传输速率。而 XAUI 可支持到 10Gbps 的传输速率，Interlaken 接口可支持到 40Gbps 的业务传输。

对于不同所需的业务形式及处理器的类型，则可选择相应的接口形式，来传输具体的业务。现今主流 FPGA 中都提供的各种接口的 IP。选择 FPGA 与各型 CPU 互联接口，一般选择主流的应用交互方案，特殊的接口缺少支撑 IP，导致开发、调试、维护和兼容性的成本都较大，同时注意系统的持续演进的需要，如只在本项目使用一次，而下一项目或开发阶

段已摒弃此类接口，则需提前规划技术路线。毕竟一个稳定、高效的接口互联是一个项目成功的基础。

不是所有的嵌入式系统都需要“高大上”的接口形式，各类低速的稳定接口也同样在FPGA 的接口互联中有着重要的角色，其中 UART、SPI、I2C 等连接形式也非常的常见。毕竟，一个优秀的设计不是“高大上”的堆积，而是对需求最小成本的满足。适合的才是最美的。

灵活性的陷阱

如果说用一个词来描述 FPGA 的特性，灵活性肯定名列前茅。

FPGA 的灵活性在于：

（一）I/O 的灵活性，其可以通过其 I/O 组成各种接口与各种器件连接，并且支持不同的电气特性。

（二）内部存储器灵活性，可以通过 IP 生成工具生成各种深度和宽度的 RAM 或者 FIFO 等。

（三）逻辑的灵活性，内部逻辑通可生成的各种类型 IP。

对于 I/O 接口来说，FPGA 的 I/O 可以支持不同类型的电平和驱动能力，各 I/O 未定义之前其地位平等，例如一个数据信号可将其约束在任意引脚，只要其电平符合连接的规范。因此硬件工程师基于这种认识，在 PCB 布线时，基于布线需要，便调整其布线的顺序，例如互换两个信号的位置。通常情况上，这种调整是没有任何问题的。但是随着 FPGA 的接口 IP 核硬核化的趋势，逐渐由很多的接口 IP 不能支持这种调整。例如对于较早的 SDRAM 或者 DDRSDRAM 来说，在 xilinx 和 ALTERA 的 FPGA 上，其数据、地址信号等都是可调的。但是随着 DDR2，DDR3 接口的出现，其 IP 接口，只能支持在某个 BANK 并且例化结

束后直接生成相应的约束文件，而这些的改动将会导致布局布线的错误。另一些例子则是一些高速 SERDES 的组合。例如对于 XAUI 接口来说，其硬核 IP（ALTERA）上就不支持 4 组 SERDES 的顺序互换，这将会影响其硬核 FCS 的编码。如果板级连接上与 PHY 的顺序与 FPGA 例化 IP 的约束不一致，则其硬核 PCS 就不能布局布线通过（软核 FCS 可以支持调整）。这种灵活性认识导致硬件板级互联的问题可谓屡见不鲜，特别是系统复杂度的上升，板级连线的增加，将会导致设计人员疏忽从而掉入“灵活性的陷阱”。解决此类问题的方法。包括（1）预评估，在设计之前就在 FPGA 上评估所需的接口的逻辑占用、约束位置、时钟需求等等，预先评估给系统设计提供相应的数据支撑和设计参考。（2）沟通，对于设计的变更，要进行有效沟通，不能使铁路警察，各管一段。（3）设计评审，虽然老套，但每个环节上的评审能有效减少掉入类似陷阱的几率。

对于内部存储资源，大多数 FPGA 工程师就是拿来就用的状态。而缺少整体内部 memory 规划，一般来说，对于单端口、双端口、假双端口，各型芯片手册中都有明确的定义，例如 xilinx 的 SPATAN3 系列中最小 RAM 单元为 18K。一个 RAM 例化最小单位就是 18K。而新的器件中最小单位一般为 9K。也就是说虽然工程师例化的较小的 RAM，例如 256*16.只有 4K，但是其也占用一个最小单元，根据器件的不同而不同。而乱用双端口导致 RAM 资源的过分占用则是更常见的设计问题。FPGA 内部对于单个 RAM 能够支持的真双端口是有限制的。举例说明，对于 ALTERA 的 9K 的存储单元一般支持 512*18 的双端口 RAM。但如果是一个 256*32 的双端口则需要占用 2 个 9K 的存储 RAM。也就是说，RAM 器件的能力是有限的，这取决于 RAM 的外部互联线是有限的，以刚才说的 256*32 的双端口 RAM 来说，其需要数据线就是 64 根（双端口），对于单个 RAM 的连线资源来说，这是 FPGA 内部逻辑资源难以承受的。所以根据器件特定，合理规划内部 memory 资源，才能在最大限度的达到高效的利用。

FPGA 内部可以例化各型 IP，基于 IP 的复用的可以大大增加研发的进度。但是各种 IP 的互联之间则需对 IP 的特性了解清楚，明确 IP 是否为业务所需的 IP。有的 IP 和工程所需可能只是名称一致，但其功能却不是你想要的。例如网口 IP 在 MII 连接方式下，是用于 FPGA 连接 PHY 的操作。如果 FPGA 与 CPU 通过 MII 连接，现有的 IP 则难以满足需求。这是因为 MII 连接 PHY 其所有的时钟都是 PHY 提供的。CPU 的设计也是与 PHY 连接，其时钟也有 PHY 提供。而如果二者连接，就变成都等着对方提供时钟，则就变成没有时钟。这种调试问题相对来说容易解决，不过在系统规划是，就需要对整个 IP 是否能够满足系统的设计要求，有着明确的判断。

灵活性是 FPGA 最大的特性，在设计中避开那些灵活性的陷阱，才能从 FPGA 整体上提升设计能力，而不是做只会写 Verilog 的码农。毕竟 FPGA 设计不是软件设计，其最终要成为变成硬件承载的，每一行语句都要考虑其综合后的电路，才能真正领会 FPGA 设计的精髓。

从零开始调试 FPGA

“合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。” 老子《道德经》

对于新手来说，如何上手调试 FPGA 是关键的一步。
对于每一个新设计的 FPGA 板卡，也需要从零开始调试。
那么如何开始调试？
下面介绍一种简易的调试方法。
(1) 至少设定一个输入时钟 input sys_clk;
(2) 设定输出 output [N-1:0] led;
(3)设定 32 位计数器 reg [31:0] led_cnt;
(4) 时钟驱动计数器开始工作

```
always@(posedge sys_clk)
led_cnt <= led_cnt + 1
```

(5)输出 led 信号。

```
assign led = led_cnt[M:N];
```

程序完成。

(6)设定管教约束

如果为 XILINX FPGA ，在 UCF 文件中 NET “sys_clk” LOC = 管脚名称

如果为 ALTERA FPGA ，在 QSF 文件中，添加 set_location_assignment 管脚名称 -to sys_clk

其他管脚，可依次类推。

(7)编译，布局、布线，生成配置文件。

XILINX 生成 BIT 文件。

ALTERA 生成 SOF 文件

(8)连接 JTAG，下载相应的配置文件。

(9)观察是否闪灯(肉眼可见)。

关于闪灯的解释如下：

assign led = led_cnt[M:N]; led_cnt 为 32bit 的信号，需要几个闪灯，则根据输入时钟的频率和肉眼能够分辨的时间（100ms）。如输出时钟为 25Mhz。则闪灯看见的位置能够分批到 10hz。需分频 2.5M= 32‘h2625A0,因此，则需要输出至少为 led_cnt[21]位，才能看到闪灯。

虽然程序简单，但是，通过调试可以确认：

（1）首先可确定 JTAG 下载器的正确连接，能够正常下载下载文件。如不能，常见问题包括

(一)检查是否安装驱动。

(二)下载器是否由红灯变成黄灯/绿灯。如红灯亮一般情况下，JTAG 的与电路板 VCC 没有供电。

(三)检查 JTAG 连接的线序。

(四)检查 JTAG 电路，检查原理图上 TMS，TDI，TDO 的上拉和下拉电阻是否与 datasheet 中一致。

通过以上四种方式，可排除绝大部分 JTAG 下载的错误。

（2）可以判断晶振是否起振，下载后无灯闪。

(一)首先，示波器查看晶振频率，观察晶振的输出，如无输出，查看晶振的电源和地信号，如电源正常，而晶振无反应，则更换晶振。

(二)如无示波器，也有替代的方法，通过嵌入式逻辑分析仪抓信号（任意信号）。如逻辑分析仪点击采样后无反应，则无时钟输入。

这是因为逻辑分析仪也需要时钟进行逻辑值的存储。

（3）如正常下载后闪灯，证明该 FPGA 板卡硬件设计上能够达到最低限度的 FPGA 调试状态。

最后，说明一下，为什么是闪灯而不是亮灯的程序，这是因为，首先闪灯可以判断外部晶振工作正常，并且由于 LED 等通常为上拉，也就是说逻辑值 0 表示灯亮，而也不排除某硬件工程师非要下拉。逻辑 1 表示亮。因此采用闪灯更加方便。

问题：为什么 LED 灯值为什么要上拉？

这是因为：LED 上拉后，需要灯亮时电流由外部电源提供，而下拉，灯亮时电流由芯片的 CMOS 电路驱动。这种在设计中应避免。

架构设计漫谈（一）流驱动和调用式

勿用讳言，现在国内 FPGA 开发还处于小作坊的开发阶段，一般都是三、四个人，七八台机器.小作坊如何也能做出大成果。这是每个 FPGA 工程师都要面临的问题。架构设计是面临的第一关。经常有这样的项目，

需求分析，架构设计匆匆忙忙，号称一两个月开发完毕，实际上维护项目就花了一年半时间。主要包括几个问题，一，性能不满足需求。二，设计频繁变更。三，系统不稳定，调试问题不收敛。

磨刀不误砍柴工，FPGA 设计的需求分析是整个设计第一步。如何将系统的**功能需求**，转换成 FPGA 的设计需求，是 FPGA 架构设计的首要问题。首先， 需要明确划分软件和硬件的边界。软件主要处理输入输出、界面显示、系统管理、设备维护。而 FPGA 则负责大数据流的处理。

如果使用几百元 FPGA 实现了一个十几元单片机就能完成的功能，就算实现的非常完美，那么这是一个什么样的神设计?任何一个项目都要考虑成本，研发成本、物料成本、维护成本等等。FPGA 的使用位置必定是其他器件难以企及的优势。

因此对于一个 FPGA 架构设计，其首先需要考虑就是性能，如没有性能的需求，其他的处理器 ARM 就可能替代其功能。其次就是接口，用于处理器扩展其没有的接口，作为高速接口转换。最后，需要考虑就是可维护性，FPGA 的调试是非常耗时的，一个大型的 FPGA 的编译时间在几小时甚至更高（通过嵌入式分析仪抓信号，每天工作 8 小时，只能分析两到三次）。而软件调试只需 make，编译时间以秒来记（这个问题可以通过提升编译服务器性能改善而不能消失，本质上要考虑可测性设计）。如果不考虑维护性和可测性，调试成本和压力就非常之大。

通常，FPGA 的大部分架构设计可以采用数据流驱动的方式来实现，例子 1，假设一个实现视频解压缩 FPGA 的设计，输入是无线接口，输出为显示屏。那么输入输出的接口基本就能确定。以数据流为驱动可以粗略划分，输入接口->解压缩模块->视频转换模块->显示接口。如需要视频缓冲，则确定是否需要连接外部存储器。那就需要确认在什么位置进行数据的缓冲。通过要支持显示的画面的质量，就能确认最大码流，同样可以计算视频解压模块和转换模块的计算能力，从而导出所需的内部总线宽度，系统频率，以及子模块个数等等。例子 2，某支持通过有线电视网上网电视 IP 网关，同样也是输入的普通 IP 网络，输出为有线调试网的调试解调器。将 IP 报文等长填充后，在固定时隙内送入有线电视网中，同样也是基于数据流驱动的方式。

数据流驱动式架构，可以作为 FPGA 设计中一个最重要的架构。通常来说应用于 IP 领域、存储领域、数字处理领域等较大型 FPGA 设计都是**数据流驱动式架构，**主要包括输入接口单元，主处理单元，输出接口单元。还可能包括，辅助处理单元、外部存储单元。这些单元之间一般采用流水式处理，即处理完毕后，数据打包发完下一级处理。其中输入输出可能有多个，此时还需要架构内部实现数据的交换。

另一种较为常用的架构方式为**调用式架构，**即一般 FPGA 通过标准接口如 PCI、PCI-E，CPCI，PCI-X，EMIF 等等。各种接口，FPGA 内部实现某一加速单元，如视频加速，数据处理，格式转换等操作。这种结构基本基本围绕 FPGA 接口、加速单元展开，属于数据的反馈类型，即处理完数据又反馈回接口模块。

其他虽然各型各样，如 SOPC，如各型接口，但本质上其都是为上述架构服务的，或做配置管理替代外部 CPU，或在数据流中间传递中间参数。或在内部实现 CPU+协处理器的架构，因此说，无他变化。

孙子兵法云：“兵无常势，水无常形”。但是对于一种设计技术来说，没有一种固定演进的架构和设计，那么项目的整个设计层次总是推到重来，从本质上说，就是一种低水平重复。如果总结规律，提炼共性，才能在提升设计层次，在小作坊中取得大成果。

架构设计漫谈（二）稳定压倒一切

敏捷开发宣言中，有一条定律是“可以工作的软件胜过面面俱到的文档”。如何定义可可以工作的，这就是需求确定后架构设计的首要问题。而大部分看这句话的同志更喜欢后半句，用于作为不写文档的借口。

FPGA 的架构设计最首先可以确定就是外接接口，就像以前说的，稳定可靠的接口是成功的一半。接口的选择需要考虑几个问题。

1， 有无外部成熟 IP。一般来说，ALTERA 和 XILINX 都提供大量的接口 IP，采用这些 IP 能够提升研发进度，但不同 IP 在不同 FPGA 上需要不同 license，这个需要通过代理商来获得（中国国情，软件是不卖钱的）。

2， 自研接口 IP，能否满足时间、进度、稳定性、及兼容性的要求。

案例 1 设计一个网络接口在逻辑设计上相对简单，比如 MII 接口等同于 4bit 数据线的 25MHZ 样，而 RGMII 可以使用双沿 125Mhz 的采样专用的双沿采样寄存器完成（使用寄存器原语）。但是如何支持与不同 PHY 连接一个兼容性问题（所谓设计挑 PHY 的问题，这个问题后面详述）。

案例 2:CPU 通过接口连接 FPGA 时，如果 CPU 此时软复位，则有管脚会上拉，此时如果该管脚连接 FPGA 接口是控制信号且控制信号高电平有效，则此时 FPGA 逻辑必然出错。同样 FPGA 在配置时，管教输出高阻，如此时 CPU 上电且板级电路管脚上拉，则同样会导致 CPU 采样出错（误操作的问题）。

不能只是考虑编写 verilog 代码仿真能对就行，接口设计应该站在系统的角度来看问题，问题不是孤立的，还是互相联系。

设计中，如果需要存储大量数据，就需要在外部设计外部存储器，这是因为 FPGA 内部 RAM 的数量是有限的。是采用 SRAM、DDR2、DDR3。这就需要综合考虑存储数据大大小，因为 SRAM 的容量也有限，但是其接口简单，实现简单方便，且读取延时较小。DDR2、DDR3 的容量较大，接口复杂，但 FPGA 内部有成熟 IP 可用，但是读取的延时较大，从发起读信号到读回数据一般在十几个时钟周期以上。如果对数据时延有要求，需要上一次存储数据作为下一次使用，且数据量不太大（几百 K 到几兆），则 SRAM 是较好的选择。而其他方面 DDR2/DDR3 是较好的选择。为什么不用 SDRAM 或者 DDR。这是因为设计完毕，采购会告诉你，市场上这样老的芯片基本都停产了。

FPGA 接口在设计选择的原则就是：能力够用，简单易用。特别值得一提的是高速 SERDES 接口，最好使用厂商给的参考设计，有硬核则不选择软核，测试稳定后，一定要专门的位置约束，避免后面添加的逻辑拥挤后影响到接口时序，也可避免接口设计人员与最终的逻辑设计人员扯皮（不添加过多逻辑，接口是好用的）。一个分析高速 SERDES 的示波器，采样频率至少 20G 甚至更高以上，动辄上百万，出现问题，不一定有硬件条件可调试。

回到开头，如何定义“可用的”设计，稳定我想是前提，而接口的稳定性更是前提的前提。这里稳定包括，满负荷边界测试，量产、环境试验等一系列稳定可靠。而在架构设计中，就选择成熟的接口，能有效的避免后续流程中的问题，从源头保证产品的质量。

架构设计漫谈（三）时钟和复位

接口确定以后，FPGA 内部如何规划？首先需要考虑就是时钟和复位。

时钟：根据时钟的分类，可以分为逻辑时钟，接口时钟，存储器时钟等；

(1)逻辑时钟取决与逻辑的关键路径，最终值是设计和优化的结果，从经验而不是实际出发：低端 FPGA(cyclone spantan)工作频率在 40-80Mhz 之间，而高端器件（stratix virtex）可达 100-200Mhz 之间，根据各系列的先后性能会有所提升，但不是革命性的。

(2)接口时钟，异步信号的时序一般也是通过 FPGA 片内同步逻辑产生，一般需要同步化，即接口的同步化采样。某些接口的同步时钟一般是固定而精确的，例如下表所示，如 SERDES 的时钟尽量由该 BANK 的专用时钟管脚输入，这样可保证一组 SERDES 组成的高速接口时钟偏斜一致。

接口名称	IP 输入时钟	备注
MII	25Mhz	
RGMII/GMII	125Mhz	
XAUI	156.25Mhz	差分 IP 内部倍频使用
PCI	33Mhz	
PCI-e	100Mz	差分输入 IP 内部倍频使用

(3)外部存储器时钟：这里时钟主要为 LPDDR/DDR2/DDR3 等器件的时钟，一般来说 FPGA 的接口不用工作在相应器件的最高频率。能够满足系统缓存数据的性能即可，但是一般这些 IP 的接口都规定了相应的最小时钟频率，因为这些接口状态机需不停进行外部器件的刷新（充电），过低的频率可能会引起刷新的问题，造成数据丢失或者不稳定。

(4)另外一些需要输出的低速时钟，例如 I2C、MDIO、低速采样等操作，可以通过内部分频得到。不用通过 PLL/DCM 产生所需时钟。在 XILINX 的 FPGA 中，禁止 PLL 产生的时钟直接输出到管脚上，而 ALTERA 的器件可以如此操作。解决此类问题的方法可通过 ODDR 器件通过时钟及其 180 度相位时钟（反向）接入的时钟管脚分别采样 0、1 逻辑得到。

因为有了 DCM/PLL 这些专用产生时钟的器件，似乎产生任意时钟输出都是可能是，但实际例化的结果，时钟的输出只能选取某些范围和某些频率，取决于输入时钟和分频系数，CLK_OUT = CLK_IN *(M/N) 。这些分频系数基本取整数，其产生的频率也是有限的值。

复位：根据复位的分类，FPGA 内部复位可以分为硬复位，逻辑复位、软复位等；

硬复位：故名思议，即外部引脚引入的复位，可以在上电时给入，使整个 FPGA 逻辑配置完成后，能够达到稳定的状态，这种复位重要性在于复杂单板上除了 FPGA 外，可能还有多个器件（CPU、DSP），其上电顺序不同，在未完成全部上电之前，其工作状态为不稳定状态。这种复位引脚可以通过专用时钟管脚引入，也可通过普通 I/O 引入，一般由单板 MCU 或者 CPLD 给出。

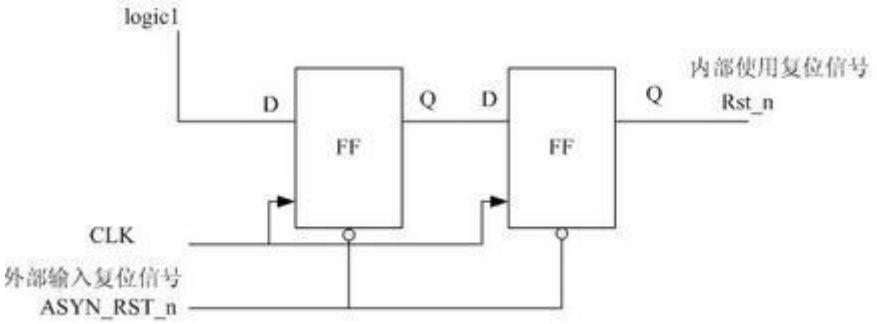
逻辑复位：则是由 FPGA 内部逻辑产生，例如可以通过计数产生，等待一段时间开始工作，一般等待外部某些信号准备好，另一种 FPGA 内部逻辑准备好的状态信号，常见的有 DCM/PLL 的 LOCK 信号；只

有内部各逻辑准备好后，FPGA 才能正常工作。另外 FPGA 内部如设计逻辑的看门狗的话，其产生的复位属于这个层次。

软复位：严格的说，应属于调试接口，指 FPGA 接收外部指令产生的复位信号，用于复位某些模块，用于定位和排除问题，也属于可测性设计的一部分。例如 FPGA 通过 EMIF 接口与 CPU 连接，内部设定软复位寄存器，CPU 通过写此寄存器可以复位 FPGA 内部单元逻辑，通过写内部寄存器进行软复位，是复杂 IP 常用的功能接口。调试时，FPGA 返回错误或无返回，通过软复位能否恢复，可以迅速定位分割问题，加快调试速度。

复位一般通过**与**或者**或**的方式（高电平或、低电平与），产生统一的复位给各模块使用。模块软复位信号，只在本模块内部使用。

问题：同步复位好、还是异步复位好？ XILINX 虽然推荐同步复位，但也不一概而论，复位的目的是使整个系统处于初始状态，这根据个人写代码经验，这些操作都可以，前提是整个设计为同步设计，时钟域之间相互隔开，复位信号足够长，而不是毛刺。下面推荐一种异步复位的同步化方式，其电路图如下：

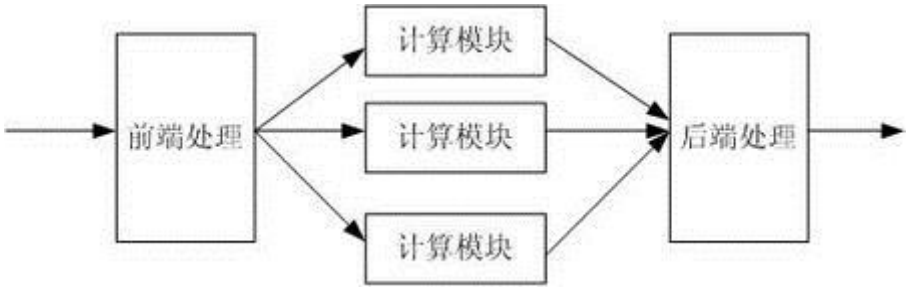


时钟和复位基本上每个模块的基本输入，也是 FPGA 架构上首先要规划的部分，而不要用到才考虑，搞的整个设计到处例化 DCM 或者输出 LOCK 进行复位，这些对于工程的可维护性和问题定位都没有益处。《治家格言》说：“宜未雨而绸缪，毋临渴而掘井。这与 FPGA 时钟和复位的规划是同一个意思。

架构设计漫谈（四）并行与复用

FPGA 其在众多器件中能够被工程师青睐的一个很重要的原因就是其强悍的处理能力。那如何能够做到高速的数据处理，数据的并行处理则是其中一个很重要的方式。

数据的并行处理，从结构上非常简单，但是设计上却是相当复杂，对于现有的 FPGA 来说，虽然各种 FPGA 的容量都在增加，但是在有限的逻辑中达到更高的处理能力则是 FPGA 工程师面临的挑战。常用并行计算结构如下图所示：



上图中：前端处理单元负责将进入数据信息，分配到多个计算单元中，图中为 3 个计算单元（几个根据所需的性能计算得出）。然后计算单元计算完毕后，交付后端处理单元整合为统一数据流传入下一级。

如果单个计算单元的处理能力为 N，则通过并行的方式，根据并行度 M，其计算能力为 N*M；在此结构中，涉及到几个问题:

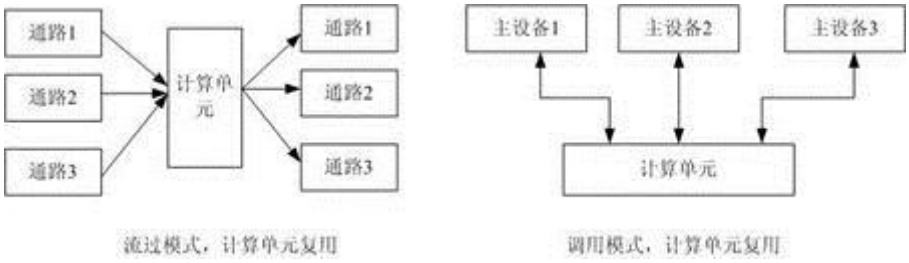
一， 前端处理单元如何将数据分配到多个计算单元，其中一种算法为 round-robin，轮流写入下一级计算单元，这种方式一般使用用计算单元计算数据块的时间等同。更常用的一种方式，可以根据计算单元的标示，即忙闲状态，如果哪个计算单元标示为闲状态，则分配其数据块。

二， 计算单元和前后端处理之间如何进行数据交互。一般来说，计算单元处理频率较低，为关键路径所在。前后端处理流量较大，时钟频率较高，因此通过异步 FIFO 连接，或者双端口 RAM 都是合适的方式。如果数据可分块计算，且块的大小不定，建议使用 FIFO 作为隔离手段，同时使用可编程满信号，作为前端处理识别计算模块的忙闲标示。

三， 如果数据有先后的标示，即先计算的数据需要先被送出，则后端处理模块需要额外的信号，确定读取各个计算模块的顺序。这是因为：如果数据等长，则计算时间等长，则先计算的数据会先被送出。但是如果数据块不等长，后送入的小的数据块肯能先被计算完毕，后端处理单元如果不识别先后计算的数据块，就会造成数据的乱序。这可以通过前端计算单元通过小的 FIFO 通知后端计算单元获知首先读取那个计算单元输出的数据，即使其他计算单元输出已准备好，也要等待按照顺序来读取。

数据的并行处理是 FPGA 常用的提升处理性能的方法，其优点是结构简单，通过计算单元模块的复用达到高性能的处理。缺点，显而易见就是达到 M 倍的性能就要要耗费 M 倍逻辑。

与之相反减少逻辑的另一种方式，则是复用，即一个处理能力较强的模块，可以被 N 的单元复用，通过复用，而不用每个单元例化模块，可以达到减少逻辑的效果，但控制复杂度就会上升。其结构图如下所示：



流过模式，计算单元复用　　调用模式，计算单元复用

上图复用的结构图中，分别介绍了流过模式复用和调用模式复用。流过模式下，计算单元处理多路数据块，然后将数据块分配到多路上，这种情况下，通过 round-robin 可以保证各个通路公平机会获得计算单元。其处理思路与上图描述并行处理类似。

调用模式下，计算单元被多个主设备复用，这种架构可以通过总线及仲裁的方式来使各个主设备能够获取计算单元的处理（有很多成熟的例子可供使用，如 AHB 等）。如果多个主设备和多个计算单元的情况，则可以不通过总线而通过交换矩阵，来减少总线处理带来的总线瓶颈。

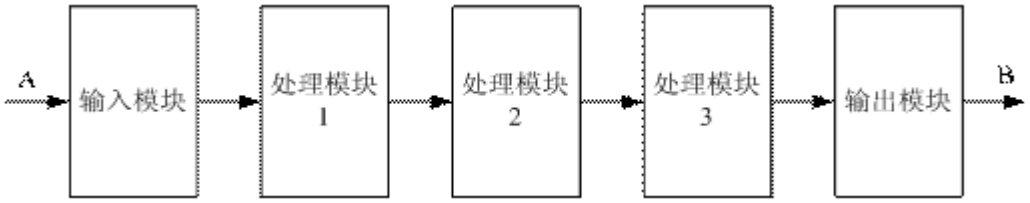
实际应用场合，设计的架构都应简单实用为好，交互矩阵虽然实用灵活，但其逻辑量，边界测试验证的难度都较大，在需要灵活支持多端口互联互通的情况下使用，可谓物尽其用。但如果仅仅用于一般计算单元能力复用的场景，就属于过度设计，其可以通过化简成上述两种简单模式，达到高速的数据处理的效果。

并行和复用，虽然是看其来属性相反的操作，但其本质上就是通过处理能力和逻辑数量的平衡，从而以最优的策略满足项目的需要。设计如此，人生亦然。

架构设计漫谈（五）数字电路的灵魂-流水线

流水线，最早为人熟知，起源于十九世纪初的福特汽车工厂，富有远见的福特，改变了那种人围着汽车转、负责各个环节的生产模式，转变成了流动的汽车组装线和固定操作的人员。于是，工厂的一头是不断输入的橡胶和钢铁，工厂的另一头则是一辆辆正在下线的汽车。这种改变，不但提升了效率，更是拉开了工业时代大生产的序幕。

如今，这种模式常常应用于数字电路的设计之中，与现在流驱动的 FPGA 架构不谋而合。举例来说：某设计输入为 A 种数据流，而输出则是 B 种数据流，其流水架构如下所示：

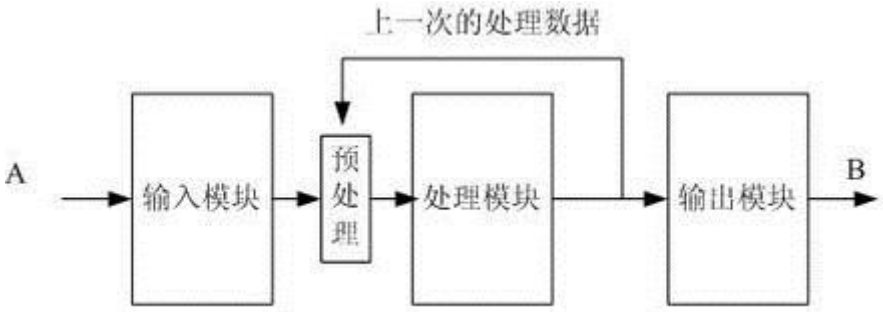


每个模块只负责处理其中的一部分，这种处理的好处是，1、简化设计，每个模块只负责其中的一个功能，便于功能和模块划分。2，时序优化，流水的处理便于进行时序的优化，特别是处理复杂的逻辑，可以通过流水设计，改善关键路径，提升处理频率，并能提升处理性能。

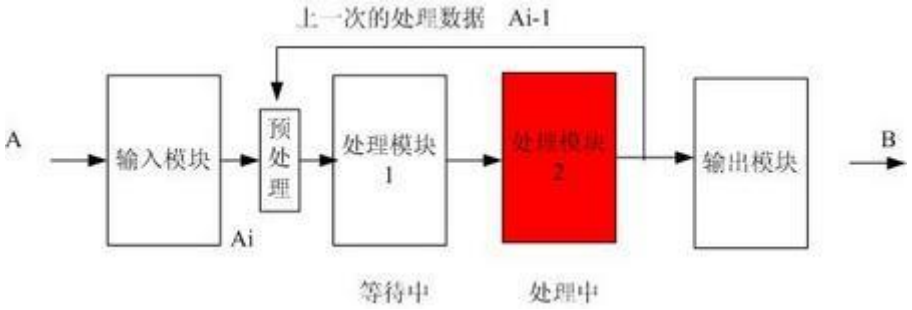
各个流水线之间的连接方式也可通过多种方式，如果是处理的是数据块，流水模块之间可以通过 FIFO 或者 RAM 进行数据暂存的方式进行直接连接、也可以通过寄存器直接透传。也可通过某些支持 brust 传输的常用业界标准总线接口进行**点对点**的互联，例如 AHB，WISHBONE，AVALON-ST 等接口，这种设计的优点是标准化，便于模块基于标准接口复用。每个模块的接收接口为从接口（SLAVE），而发送接口为主接口（MASTER）。

架构流水的好处一目了然，但另一个问题，对于某些设计就需要谨慎处理，那就是**时延**。对于进入流水线的信息 A，如果接入的流水处理的模块越多，其输出时的时延也越高，因此如对处理时延要要求的设计就需要在架构设计时，谨慎对待添加流水线。架构设计时，可以通过处理各个单元之间的延时估计，从而评估系统的时延，避免最终不能满足时延短的需求，返回来修改架构。

流水架构在另种设计中则无能为力，那就是带反馈的设计，如下图所示：



图中，需要处理模块的输入，需要上一次计算后的结果的值，也就是输出要反馈回设计的输入。例如某帧图像的解压需要解压所后的上一帧的值，才能计算得出。此时，流水的处理就不能使用，若强行添加流水，则输入需等待。



如上图中，如在需反馈的设计中强加流水，则输入信息 Ai 需要等待 Ai-1 处理完毕后，再进行输入，则**处理模块 1**，就只能等待（空闲）。因此，问题出现了，流水线等待实际上就是其流水处理的的效果没有达到，白白浪费了逻辑和设计。

流水应用在调用式的设计中，可以通过接口与处理流水并行达到。即写入、处理、读出等操作可以做到流水式架构，从而增加处理的能力。

流水是 FPGA **架构设计**中一种常用的手段，通过合理划分流水层次，简化设计，优化时序。同时流水在**模块设计**中也是一种常用的手段和技巧。这将在后续重陆续介绍。，流水本身简单易懂，而真正能在设计中活用，就需要对 FPGA 所处理的业务有着深刻的理解。正如那就话，知晓容易，践行不易，且行且珍惜。

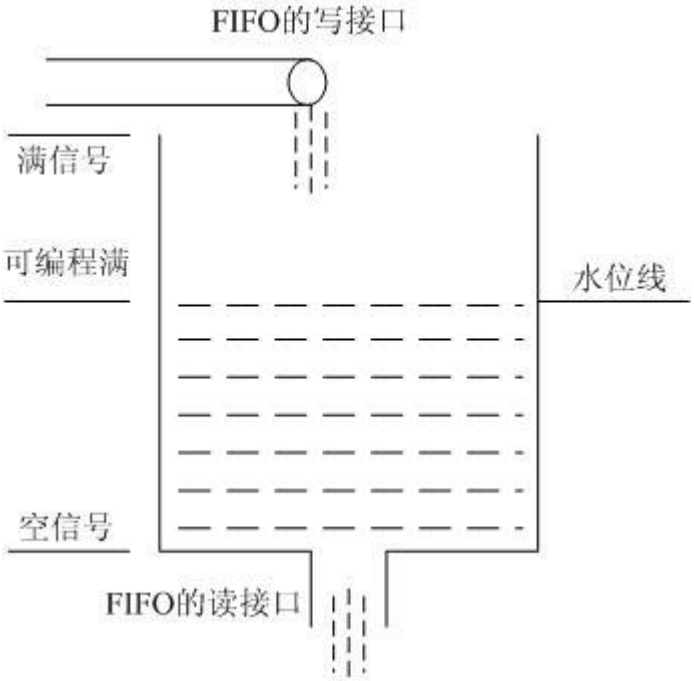
设计不是凑波形（一）FIFO（上）

FIFO 是 FPGA 内部一种常用的资源，可以通过 FPGA 厂家的的 IP 生成工具生成相应的 FIFO。FIFO 可分为同步 FIFO 和异步 FIFO，其区别主要是，读写的时钟是否为同一时钟，如使用一个时钟则为同步 FIFO，读写时钟分开则为异步 FIFO。一般来说，较大的 FIFO 可以选择使用内部 BLOCK RAM 资源，而小的 FIFO 可以使用寄存器资源例化使用。

一般来说，FIFO 的主要信号包括：

信号	数据信号
读信号	rd_en
读数据	dout
读空信号	empty
写信号	wr_en
写数据	din
写满信号	full

实际使用中，可编程满的信号（XILINX 的 FIFO）较为常用，ALTERA 的 FIFO 中，可以通过写深度（即写入多少的数据值）来构造其可编程满信号。通过配置 threshold（门限）的值可以设定可编程满起效时的 FIFO 深度。



上图所示为 FIFO 的模型，可以看做一个漏桶模型，其中输入、输出、满信号、空信号、可编程满等信号如图所示，一目了然。 其中 threshold 信号可以看做水位线，通过此信号可以设置可编程满信号。FIFO 的其他的信号也大都与其深度相关，如有特殊需求，可通过厂商提供的 IP 生成工具的图形界面进行选择使用。

FIFO 的使用场景有多种，其中主要如下所示：

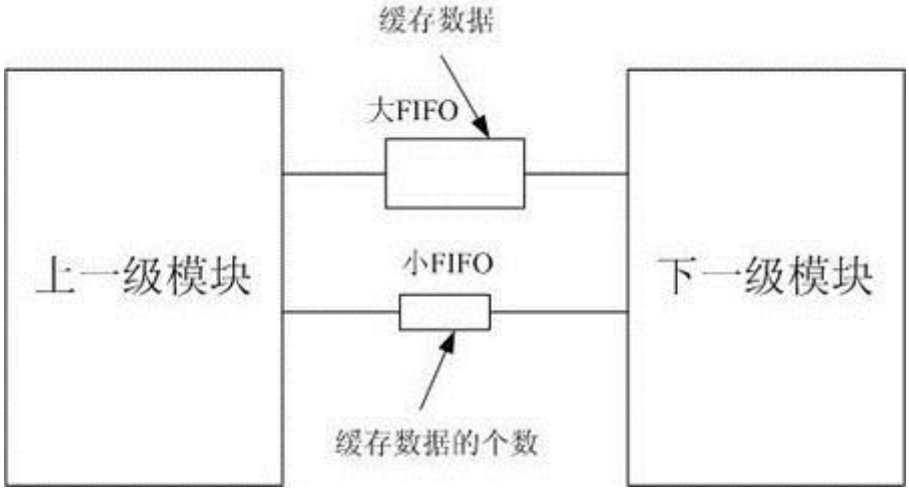
（1） 数据的缓冲，如模型图所示，如果数据的写入速率高，但间隔大，且会有突发；读出速率小，但相对均匀。则通过设置相应深度的 FIFO，可以起到数据暂存的功能，且能够使后续处理流程平滑，避免前级突发时，后级来不及处理而丢弃数据。

（2） 时钟域的隔离。对于不同时钟域的数据传递，则数据可以通过 FIFO 进行隔离，避免跨时钟域的数据传输带来的设计与约束上的复杂度。

FIFO 设计中有两个需要注意事项，首先，不能溢出，即满后还要写导致溢出，对于数据帧的操作来说，每次写入一个数据帧时，如果每写一个宽度（FIFO 的宽度）的数据，都要检查满信号，则处理较为复杂，如果在写之前没满，写过程不检查，则就容易导致溢出。因此可编程满的设定尤为必要，通过设置可编程满的水位线，保证能够存储一个数据帧，这样写之前检查可编程满即可。

其次，另一更容易出错的问题，就是空信号。对于 FIFO 来说，在读过程中出现空信号，则其没有代表该值没有被读出，对于读信号来说，如设定读出一定长度的值，只在一开始检测非空，如状态机的触发信号，容易出现过程中间也为空的信号，会导致某些数据未读出，特别是写速满而读速快的场景下。 因此 rden 与!empty 信号要一起有效才算将数据读出。

空信号处理相对容易出错，懒人自有笨方法，下面介绍一种应用于数据帧处理的 FIFO 使用方式，只需在读开始检测空信号即可，可以简化其处理读数据的流程：



其处理结构如上图所示，数据缓存以大 FIFO（BLOCK RAM 实现）为主，而每存储完毕一个数据帧向小 FIFO(寄存器实现)内写入值。当小 FIFO 标示非空时,则标示大 FIFO 中已存储一个整帧。则下一级模块可以只需检测小 FIFO 非空时，从而读出一个整帧，过程中大 FIFO 一直未非空，可以不用处理非空信号，从而简化设计和验证的流程。

此外还有 FIFO 其他应用方式，下节接着介绍。（未完待续）

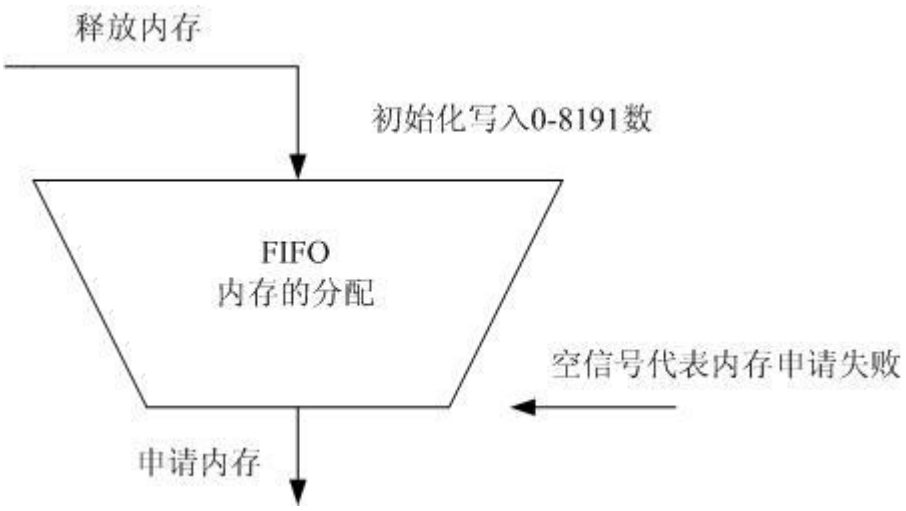
-设计不是凑波形（二）FIFO（下）

FIFO 在 FPGA 设计中除了上篇所介绍的功能之外， 还有以下作为以下功能使用：

（1） 内存申请

在软件设计中，使用 malloc（）和 free（）等函数可以用于内存的申请和释放。特别是在有操作系统的环境下，可以保证系统的内存空间被动态的分配和使用，非常的方便。如果在 FPGA 内部实现此动态的内存分配和申请，相对来说较为复杂，例如某些需要外部数据存储且需动态改变的应用需求下，需要对 FPGA 外部 DDR(或 SRAM 等)的存储空间，进行动态的分配和释放。通过使用 FIFO 作为内存分配器，虽然比不上软件的灵活和方便，但是使用也较为简便。

举例说明假设外部存储空间为 8Mbyte，可将其划分为 8192 个 1Kbyte 空间。并将数值 0-8191 存储 FIFO 中，FIFO 内部存储所标示可用的内存空间。如下图所示。



首先，进行内存的初始化，即将 0-8191 写入 FIFO 中。

如需申请内存后，从 FIFO 中读取值 A，然后根据 A 的标示，写入 A 所指示的外部存储区（DDR）中相应的位置，即申请{A，10'h0_00} ->{A,10'h3_FF}的空间区域。

如释放内存后，即可向 FIFO 中写入相应的值。即可保证下次该空间能够被设计使用。

在此种设计中，FIFO 承担了内存分配和释放器的角色。此时只能申请或释放最小单元倍数的内存空间，如本例所示：为 1Kbit。如 FIFO 读空，则代表申请内存失败，需要等待其他块内存释放后再写入 FIFO 中，才能再次申请。

（2） 串并转换

对于串并转换，可能对于 FPGA 工程师来说，非常常见，但是如果有专门的 IP 实现此功能，可简化设计，减少出错及验证的工作量。例如：对于外部输入的需要进行串并转换的信号，并进行存储的信号，如设计进行串并转换在存储等操作，设计，可以直接通过例化读写位宽不一致的 FIFO，例如 1 入 8 出的 FIFO，可直接将外部输入信号直接转换成 8BIT 信号并进行存储后，供后续处理使用（其他的）。

（3） 业务优先级划分

通过 FIFO 设置不同水位线，可以划分不同的业务优先级，保证高业务优先级数据流在带宽受限时，优先通过，而低业务优先级只能在满足高优先级需求后有多余的带宽时才能通过。并且可以划分多个优先级，满足多种业务的需求。设计将在以后篇幅中详述。

（4）固定带宽设定

通过对 FIFO 接口的读出使能，能够保证实现固定带宽的输出，例如 FIFO 读接口为 32bit，而读时钟为 50Mhz，则输出为 1.6Gbit/S。如实现固定带宽的输出（如 1Gbit/S），有两种方式,一种可以通过降低时钟频率到 31.25Mhz。另一种方式，可通过读信号中间插入等待周期，如果读出长度为 N 的数据所需时钟周期为 M，则需等待（3M/5）的周期，从而降低至 1Gbit/S 的处理能力，这在某些需要进行流量限制的业务方式中使用。

对于 FIFO 来说，作为 FPGA 内部资源的一个常用器件，最常见应用于异步时钟域划分和缓冲数据，但不仅限于此，简化设计、减少耦合、输入输出接口固定，便于仿真和验证，都是使用 FIFO 带来的设计上的益处。

设计不是凑波形（三）RAM

在 FPGA 内部资源中，RAM 是较为常用的一种资源。

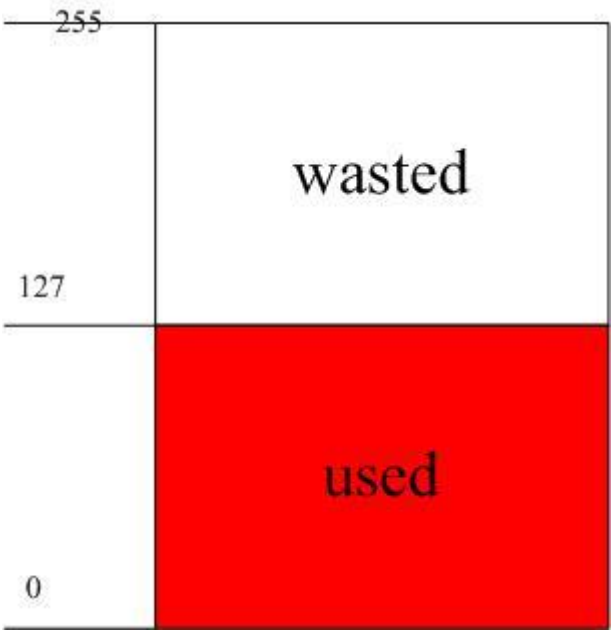
通常实例化 RAM 中，一种使用为 BLOCK RAM 也就是块 RAM 。另外资源可以通过寄存器搭，也就是分布式 RAM。前者一般用于提供较大的存储空间，后者则提供小的存储空间。

在实际应用过程中，一般使用的包括，单端口、双端口 RAM，ROM 等形式等不同的形式。 实际应用中 FIFO 也是利用 RAM 和逻辑一起实现的。

对于一块 RAM 中，其能够例化的深度是有限的。例如 cyclone4 的 RAM9k 中可以例化的资源如下所示：

Feature	M9K Blocks
Configurations (depth × width)	8192 × 1 4096 × 2 2048 × 4 1024 × 8 1024 × 9 512 × 16 512 × 18 256 × 32 256 × 36

因此：例化化深度<256 的 RAM，其同样也需要占用一块 BLOCKRAM 的资源，例如例化宽度为 64 深度为 128 的 RAM，其资源为 8K 。但是仍然需要占用两块 BOCK RAM。也就是说，只要例化 RAM 深度少于 256.则对于本器件 9K 来说，剩下的资源也是浪费。



那是不是可以例化成双端口 RAM，通过高位地址区分，变为两个单端口 RAM（width：32，depth：128）来使用，这样就可以节省资源了？

而对于双端口 RAM 来说，每个 M9K 能够配置是的最小深度就变成是 512，而最大宽度为 18，如下图所示，因此作为真双端口 RAM 使用，深度小于 512 的话，仍然会占用 1 块 RAM。且宽度》18 就会多占用额外的一块 RAM，因此上述的节省资源的方式是不正确的。

Read Port	Write Port						
	8192 × 1	4096 × 2	2048 × 4	1024 × 8	512 × 16	1024 × 9	512 × 18
8192 × 1	✓	✓	✓	✓	✓	—	—
4096 × 2	✓	✓	✓	✓	✓	—	—
2048 × 4	✓	✓	✓	✓	✓	—	—
1024 × 8	✓	✓	✓	✓	✓	—	—
512 × 16	✓	✓	✓	✓	✓	—	—
1024 × 9	—	—	—	—	—	✓	✓
512 × 18	—	—	—	—	—	✓	✓

通过查看 datasheet 的中 RAM 能够配置的方式，从而能够正确的使用 RAM 资源，从而达到高的利用效率。可以看出，RAM9K 其应用方式受限，主要是因为 RAM 的端口的连接信号受限，例如：该 RAM9K 的读端口最大支持 36 根信号线，因此对于单端口其支持的宽度为 36，双端口为 18（两个端口，总共 36 根数据线），作为 FPGA 来说，其布线资源是有限的，不可能无限制的增加其端口数。

RAM 例化时，有时需要初始化 RAM，ALTERA 和 XLINX 的初始化方法如下所示：

（1）ALTERA RAM 中，例化时为 MIF 文件，其格式为：

```
DEPTH = 32;                 -- The size of memory in words
WIDTH = 8;                  -- The size of data in bits
ADDRESS_RADIX = HEX;        -- The radix for address values
DATA_RADIX = HEX;           -- The radix for data values
CONTENT                     -- start of (address : data pairs)
BEGIN
```

```
00 : 0;                -- memory address : data
01 : 1;
END;
```

（2）在 XILINX 的 RAM 中，RAM 出示化文件为 COE 文件，其格式为：

MEMORY_INITIALIZATION_RADIX=2; 设定进制

MEMORY_INITIALIZATION_VECTOR= 初始化向量

值得注意的是：XILINX 的 RAM 初始化后会自动生成 MIF 文件，而此 mif 文件与 ALTERA mif 文件格式不同。不能用于初始化 ALTERA 的 RAM。

鉴于例化不同 IP 的复杂性，现在编译工具也支持利用 VERILOG 语言来描述 RAM，而编译工具自动识别为 RAM，自动产生相应的 IP 核，下面以 XILINX 的 EDA 工具为例简要介绍。

```
module ram(
clk, wr,addr,din,dout
);
input clk;
input [7:0] addr;
input wr;
input [31:0] din;
output [31:0] dout;
reg [31:0]   mem [0:255];
reg [31:0] dout;
always@(posedge clk)
   if(wr)
      mem[addr] <= din;
always @(posedge clk)
      dout   =   mem[addr] ;
endmodule
```

上述描述可以被描述成 RAM，自动产生 RAM.其例化的报告为：

```
================================================================================
*                     HDL Synthesis                            *
================================================================================
Performing bidirectional port resolution...
Synthesizing Unit .
    Related source file is "ram.v".
    Found 256x32-bit single-port RAM for signal .
    Found 32-bit register for signal .
    Summary:
        inferred    1 RAM(s).
        inferred   32 D-type flip-flop(s).
Unit synthesized.
================================================================================
HDL Synthesis Report
Macro Statistics
```

```
# RAMs                                          : 1
 256x32-bit single-port RAM                    : 1
# Registers                                     : 1
 32-bit register                               : 1
=========================================================================
```

综上： RAM 作为 FPGA 内部一种基本资源，掌握 RAM 的特性和基本用法，则是 FPGA 工程师的基本技能,能够充分利用 FPGA 内部的资源,毕竟对于 FPGA 来说“资源就是金钱”，节省资源就是省钱。

写在 coding 之前的铁律

写在 coding 之前的那些铁律

(1)注释： 好的代码首先必须要有注释，注释至少包括文件注释，端口注释，功能语句注释。

文件注释：文件注释就是一个说明文：这通常在文件的头部注释，用于描述代码为那个工程中，由谁写的，日期是多少，功能描述，有哪些子功能，及版本修改的标示。这样不论是谁，一目了然。即使不写文档，也能知道大概。

接口描述：module 的接口信号中，接口注释描述模块外部接口，例如 AHB 接口，和 SRAM 接口等等。这样读代码的人即可能够判断即模块将 AHB 接口信号线转换成 SRAM 接口信号。

功能语句注释： 内部关键逻辑，状态机某状态，读过程、写过程。

注释的重要性，毋庸置疑，好的注释，能够提高代码的可读性，可维护性等等。总之，养成注释的好习惯，代价不大，但是收益很大。

（2）语句：

开始写代码是，在 FPGA 设计中，特别是在可综合的模块实现中，verilog 的语句是很固定的。在 FPGA 的设计中，不外乎时序逻辑和组合逻辑，除此之外，别无他法。对于开始功能编码来说，只需知道组合逻辑信号即可生效，时序逻辑在时钟的下一拍起效就够了。

下面是编码的实例。

组合逻辑：两种组合逻辑的描述，其功能是一致的。

```
assign      A = B ? 1 : D  ?   2  : 3;
always@(*)
```

```
if(B)
    A = 1
else    if (D)
    A = 2;
else
    A = 3;
```

组合逻辑 如果是异步复位的话，描述如下

```
always@(posedge sys_clk or negedge rst_n)
if(!rst_n)
    a <= 0;
else
    a <= b;
```

也就是说，在 verilog 的可综合电路的编码中，只需要三种语句，分别是 assign, always(*) 及时序的 always(`CLOCK_EDGE clk ) 。 `CLOCK_EDGE 可以是上升沿或者下降沿。

为什么用 always@(*) 而不是 always（敏感信号列表）。“*”包含所有敏感信号列表，如果在 coding 过程中，漏掉了某个敏感信号，则会导致仿真不正确，例如本例中，敏感变量列表中，需要 B or C 但是如果漏掉一个，仿真就会在 B 或 C 有变化时，输出没有变化。导致仿真和功能不一致，但是对于综合工具来说，功能还是能够正常工作的，不会因为敏感变量列表中的值未列全而不综合某条语句。某些情况下，敏感列表的值可能有十几个甚至更多，遗漏是可能发生的事情，但是为了避免这种问题，最好采用 always(*)而不用敏感变量列表的方式,来避免仿真结果不一致的情况发生。

（3）赋值：老话重提，阻塞与非阻塞

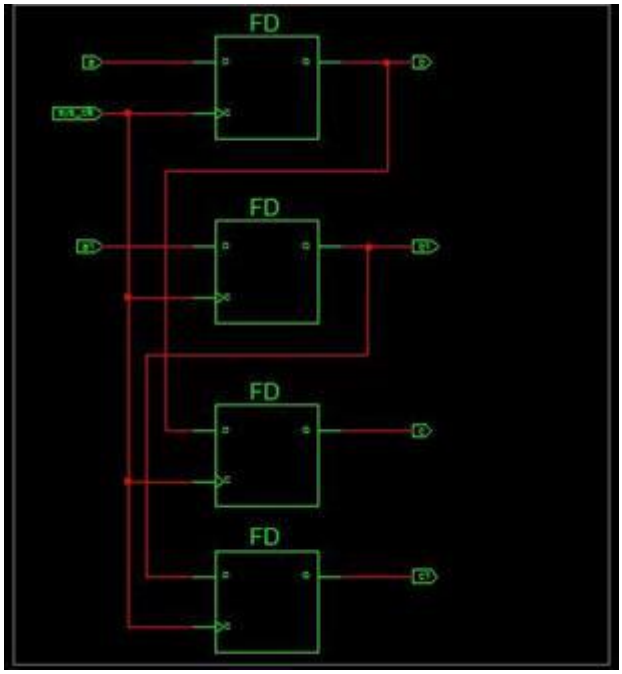
很多同志喜欢钻研阻塞赋值和非阻塞赋值，这两种赋值，分别在 always 块里面用于的阻塞“=”给组合逻辑赋值，非阻塞”<=”给时序逻辑赋值。这应该是铁律，应该在编码过程中被严格的遵守下来。“为什么？，不这么用程序也能跑”。这句话部分是正确的，疑问永远是工程师最好的老师。

诚然，某些情况下，不严格的执行也跑，但是在某些情况下，实现二者就不一样。

对于下面两个例子来说明，为什么？

```
module value1(
sys_clk,a,b,c,a1,b1,c1
);
input sys_clk;
input a;
output b;
output c;
input    a1;
output b1;
output c1;
//-----------------------------------------
//I/O type define
//-----------------------------------------
reg b;
reg c;
reg b1;
reg c1;
//-----------------------------------------
// actual code
//-----------------------------------------
always@(posedge sys_clk)
          b = a;
always@(posedge sys_clk)
          c = b;
always@(posedge sys_clk)
          b1 <=a1;
always@(posedge sys_clk)
           c1 <=b1;
endmodule
```

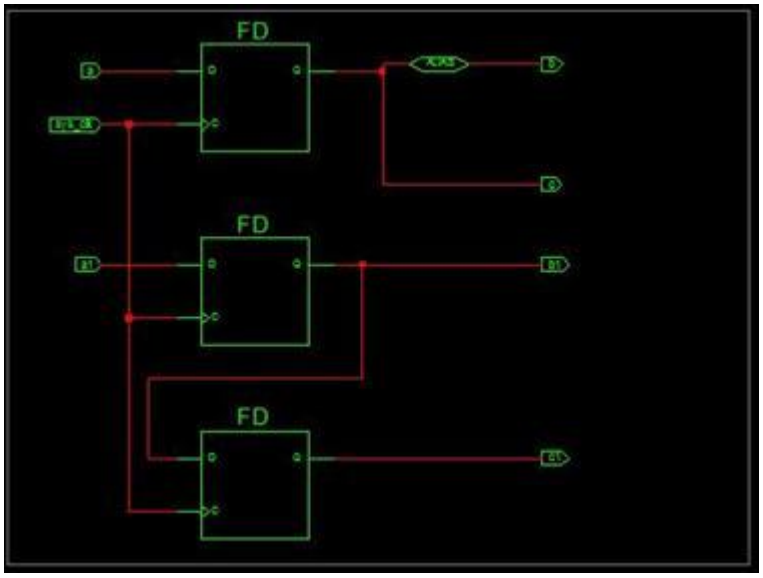
对于 value1 的描述方式：其综合后的如下所示



如果从实际的编译结果上看　b 和 b1　及 c 和 c1 其使用阻塞赋值和非阻塞赋值最终的结果是一致的，因此，也就是说，某些情况下，二者的编译结果一致。

```
module value2(
sys_clk,a,b,c,a1,b1,c1
);
input sys_clk;
input a;
output b;
output c;
input    a1;
output b1;
output c1;
//-----------------------------------------
//I/O type define
//-----------------------------------------
reg b;
reg c;
reg b1;
reg c1;
//-----------------------------------------
// actual code
//-----------------------------------------
always@(posedge sys_clk)    begin
       b = a;
       c = b;
end
always@(posedge sys_clk) begin
     b1 <=a1;
     c1 <=b1;
end
endmodule
```

而对于 value2 的描述方式：其综合后的电路图如下所示。



而对于第二中描述方式，阻塞赋值和非阻塞赋值的区别就显现出来了，从综合后的图中可以看到，c1 信号是 b1 信号的寄存，而 c 信号和 b 信号为同一信号，都为 a 信号的寄存。

作为 FPGA 工程师，一项基本的能力，就是要知道代码综合后的电路和时序，不要让其表现和你预想的不一致，“不一致”就意味着失败。即是代码的失败，也是工程的失败

对于阻塞和非阻塞赋值区别和详细说明来说，其能够编写一本书（如有时间也可专题详述），但是对 FPGA 工程师，对于 verilog 的编码而言，则只需要按照时序逻辑用“<=”非阻塞，组合逻辑用阻塞“=”赋值即可。不要挑战那些规律，试图通过语言的特性来生成特殊电路的尝试是不可取的，开个玩笑的话，是没有前途的，要把设计的精力放在通过可用的电路来实现需求上，不要舍本逐末。在数字电路设计中，我们需要的是一个确定的世界，“所见及所得”，不要让你所想的和综合编译工具得认识不一致。这也就是不要乱用和混用这两个赋值的原因。

（4）一个变量一个“家”

不要在两个 always 语句中同一个变量赋值。（这是必须的）

也尽量不要在同一个 always 语句中，对两个变量赋值。（这是可选的）

如果是一组信号，其有共同的控制条件，则在同一 always 语句中赋值能够减少代码行数，提高可读性，除此之外，最好分开来写。如果几个不太相关的信号在同一里面赋值，其可读性极差，在组合逻辑中，还容易产生 latch。

而前者赋值方式，综合工具肯定会报错，这到不用很担心，因为能够报的错误时是最容易被发现的。俗语说：“咬人的狗不叫”，而对于 FPGA 设计来说“致命的 BUG，从来不报错”。

（5）锁存

FPGA 中不要有锁存器的产生。最容易产生的是在 always(*)语句中，最后一定是所有分支条件都要描述并赋值，（一定要有最后的 else）。状态机中，同样如此，不但需要有 default 的状态,每个状态的都要有所有的分支都要赋值。

锁存器，是 FPGA 设计的大敌，因为会导致非你想要的错误功能的产生，并且导致时序分析错误，就会产生前述的问题“所见不是所得”，并且综合工具不会报错。

如果你设计的电路功能，仿真正确，而实际工作不正常，有一部分的原因是生成了锁存器，如果设计很大，不容易查的话，可以打开综合报告，搜索“LATCH”关键词，查看是否有锁存器的产生，一句话“锁存器，必杀之”。

时序逻辑会产生锁存器吗？当然不会，时序逻辑综合结果必然是触发器，因此不用检查时序逻辑的分支条件。

综上：这是写在 coding 之前的话，编码的主要功能应该是用可靠的电路来描述 FPGA 功能和需求，不要试图通过语言的特性来描述功能，设计的主要精力应放在用已知的电路（组合逻辑，时序逻辑）描述未知功能。

设计不是凑波形（五）接口设计

作为 FPGA 工程师来说，碰到新的问题是设计中最常见的事情了，技术发展趋势日新月异，所以经常会有新的概念，新的需求，新的设计等待去实现。不是每个通过 BAIDU 或者 GOOGLE 都有答案。

因此，新的设计经常会有，那如何实现？

假设，FPGA 需要设计一个接口模块，那我们就需要了解一下几个问题：

（1） 同步接口还是异步接口模块；

（2） 有哪些信号，功能是什么?

（3） 信号之间时序关系是什么?

（4） 传递的效率能够达到多少；

（5） 等等！

谁会给予这些答案，有一个好的 tutor 就是“datasheet”，一般来说 FPGA 设计一个接口模块，必然与其他硬件电路进行连接。

假如是外部连接接口为总线接口，那至少包括却不限于以下信号，

（1） 地址：能够支持的最大地址空间，数据和地址是否复用接口；

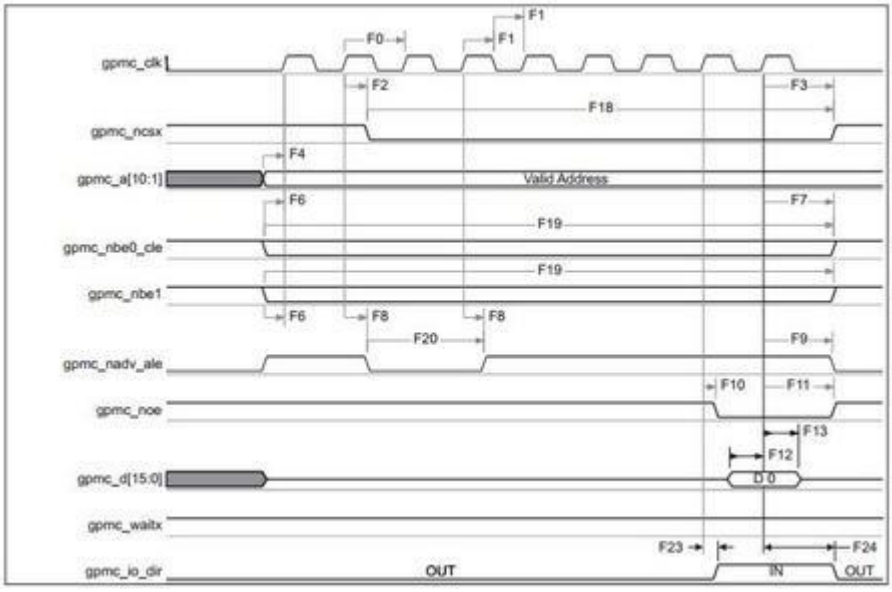
（2） 数据：一般读数据和写数据复用同一接口，一般数据信号此时都为三态。三态信号有 OE 信号。

（3） 读写命令。单次的读操作、单次的写操作

（4） 是否支持突发传输，burst 的读写操作

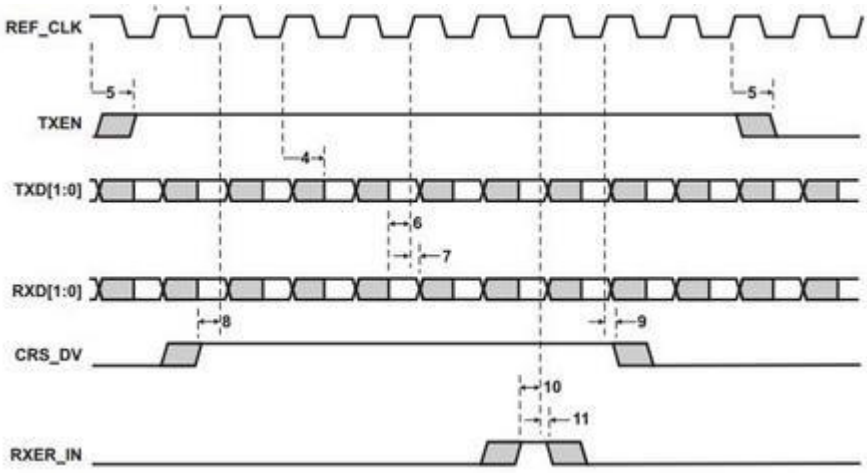
（5） 同步还是异步。

（6） 控制信号之间的相位关系及建立保持时间的要求。



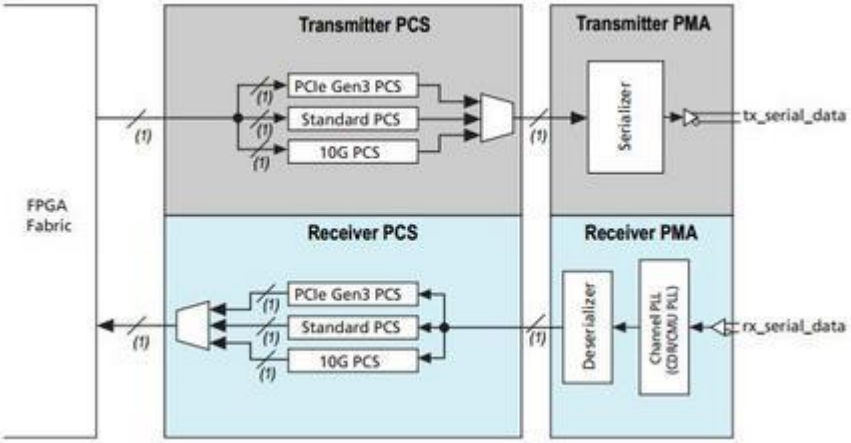
图为 TI 系列 35X 系列处理器的 GPMC 的接口，由图中可以看出其时钟、地址、数据等操作信号。

如果是同步并行接口，一般用于数据流传输，如 AD/DA 的输入或输出，网口 PHY 的输出信号等，一般的同步并行信号，通常包括，（1）时钟信号、（2）数据、信号（3）使能信号等。



上图为 RMII 的数据接口，图中可以看到其控制信号，数据信号及使能信号。作为流传输接口，一般可以支持双向双工传输。上述两种接口描述了两种主流的 FPGA 的外部接口，即总线接口和流传输接口。通过软件定义和 FPGA 内部逻辑设计，总线接口可以实现流的传输（总线接口实现双向传输，可以通过轮询和中断两种方式实现），同时流传输接口也可以实现总线读写功能。可以根据具体的使用环境进行设计。

SERDES 接口则是另一种数据流的传输接口， 现在 FPGA 的 serdes 最高可以支持到 28Gbps。实际上为了满足减少板级连线，并且提高传输速率的需求。集成更多的 SERDES 也是 FPGA 发展的趋势。



上图为 SERDES 的框图，由 PCS 和 PMA 模块构成，PMA 一般为硬核 IP，PCS 为软核或者硬核模块，收发独立，且都为差分信号（serdes 将在后续章节详述）。值得一提的是，SERDES 接口对 FPGA 逻辑的接口一般固定位同步并行接口，数据信号位宽都较大。

这些接口如何做详细设计，一方面可根据其上述共性特点，这能够对其特点有大概的认识，另一方面则是 FPGA 连接的器件的 DATASHEET。根据这些器件接口功能的描述和支持的特性。FPGA 可根据需要和功能特性，进行有选择的实现（例如总线接口不需要 brust 操作，则可只实现单次的读写，就可以满足业务的需求，进行功能裁剪和简化等等）。

接口设计完成，FPGA 就要对设计进行基本测试。对于流接口来说，能够支持环回的功能的话（即将收到数据流再发回），就极大方便测试。对于总线接口则需要支持对 FPGA

内部某地址的读写操作的测试。这就属于可测性设计的范畴。（后续将专题详述 FPGA 的可测性设计）。

如果一个模块没有任何的输入输出，其再复杂的功能也等同一块石头，或者只有输入，没有输出也等同一块石头。因此输入和输出则是一个设计的第一步。

可测性设计—从大数据开始说起

当下，最火的学问莫过于《大数据》，大数据的核心思想就是通过科学统计，实现对于社会、企业、个人的看似无规律可循的行为进行更深入和直观的了解。FPGA 的可测性也可以对 FPGA 内部“小数据”的统计查询，来实现对 FPGA 内部 BUG 的探查。

可测性设计对于 FPGA 设计来说，并不是什么高神莫测的学问。FPGA 的可测性设计的目的在设计一开始，就考虑后续问题调试，问题定位等问题。要了解 FPGA 可测性设计，只不过要回答几个问题，那就是：

（1） 设计完成如何进行测试？

（2） 设计出现问题，如何迅速定位？

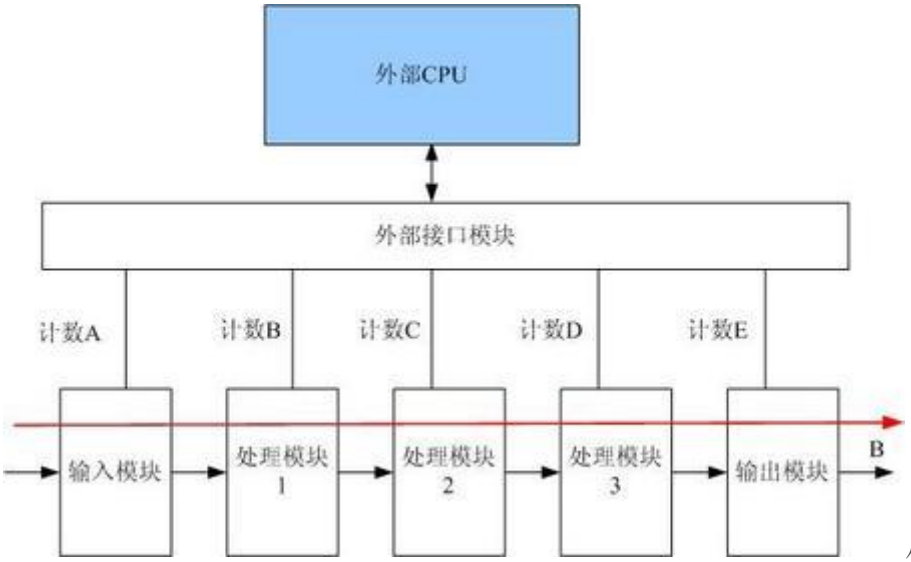
（3） 如何在设计之初就能划分故障的层次，进行问题隔离？

一般情况下，在设计的调试阶段，如果出现 BUG，则需要通过嵌入式逻辑分析仪（chipscope/signaltap）对可能出现问题的信号进行抓取。这种方式，对于较大型的设计调试速度较慢（其编译时间较长，迭代速度较慢，但是也是一种很有效的手段和 FPGA 的必备技能）。那么对于大型工程的可测试性，有什么行之有效的手段？

（1）统计计数。

FPGA 设计中的统计计数不是不是什么《大数据》，只不过是些“小数据”，例如，对于网络接口来说，收到多少包，发送多少包，收到多少字节，发送多少字节。 对于一个模块来说，收到多少次调用，或者发起多少次操作。对于读取 FIFO 的数据流操作，从 FIFO 中读取多少 frame（帧），向后级 FIFO 写入多少帧。

这些计数，毫无疑问都是需要占用资源的，但是占用这些资源是有价值的。通过这些计数，设计可以通过总线接口供外部处理器读出。于是一张 FPGA 内部设计的“大数据”图形就显现出来了”。



从上图可得，通过外部 CPU 可将各处理模块中的计数，分别读出，于是得到其内部的一张数据流图。我们可以简化设计为（计数 A->计数 B->计数 C->计数 D->计数 E）。实际使用中可根据占用的资源和实际需要的观测点来确定。

假设，我们在调试过程中发现，有输入突然没有输出，这是不需要再去内嵌逻辑分析仪来抓取信号，通过 CPU 的软件，可以打印出所有计数，在有输入驱动的情况下，假设计数 C 有变化，而计数 D 没有变化，则直接**定位处理模块 3**，此时处理有问题，简单直接而有效。

令一种情形也可以迅速定位，输入多，但是输出少，比如正常输入的码流，但是输出到显示上确实缺帧少帧，完全不流畅。通过计数分析，原本帧计数 C 和 D 应该一样多，但是 D 却计数较 C 少很多，说明此时处理模块 3 性能不够，或者设计有缺陷。这样就把整个设计的关键点定位到处理模块 3.

通过 FPGA 内部各模块的关键计数分析，来定位分析问题，在设计上没有任何难度。不过需要外部 CPU 或者 FPGA 嵌入式 CPU 的配合使用。

凡事有利就有弊，添加多的计数，会增加资源的使用量，那么如何平衡？对这种分析计数进行单独位宽设定，在全局统一的宏定义中定位`define REG_WIDTH N 。此时 N 的设定可以灵活多样，8 位/16 位/32 位等等。可以根据项目中资源的剩余量，灵活添加所需的逻辑，毕竟这些计数的值提供分析试用。

（2）状态输出。

如果向上述问题一样，我们定位到某个模块，那么如何再定位到模块中那个逻辑的问题？

解决这个问题的关键，就是状态机，向输出计数一样，一般的设计中，都会以状态机为核心进行设计，将状态机的 CS（当前状态）信号引出，如果没有外部输出的情况下，当前状态应该为 IDLE。比如上文中，我们定位到模块 3 此时死机，等待不再输入外部信号时，此时模块 3 中的状态机信号如不为 IDLE，假如此时正处于状态 B_CS,则说明此时模块的错误出现在状态 B。而 B 跳转必须由信号 X 起效。因此可以直接定位到信号 X 的问题。剩下就是要定位信号 X 为什么不起效。（也许你看看代码就能知道**不离十，是不是一秒钟变高手！ 开个玩笑！）

（3）逻辑复位。

划分 FPGA 的问题或者模块问题的另一种方式就是逻辑复位，上文讲复位时（架构设计漫谈），逻辑复位，如果 A 模块自身有逻辑复位，如果设计没有输出（通俗叫做“FPGA 死了”），如果怀疑某个模块，该模块逻辑复位后，设计又正常工作，则需要定位的则是可以是该模块、或者该模块影响的与其连接模块(该模块的非正常输出导致下一级模块出错)。

本文将的 FPGA 可测性设计，非 ASIC 讲的通过插 JTAG/BIST 进行的测试。其目的还是通过关注如何测试的设计，来定位问题，提高 FPGA 的可测性。除此之外，**逻辑探针**也是可以一个解决测试问题的方向(专题另述)。可测性的提高，意味着调试手段的增加，调试速度加快，而不是一味的依赖嵌入式逻辑分析仪。能够达到快速问题定位能力，是 FPGA 研发能力一个重要的体现，而可测性设计则是提升这一能力有力的助手。

化繁为简

有个笑话说，有个病人感冒了，于是去看医生，医生诊断后说，你得了感冒，但是我只会治疗肺炎，不如你回家再浇点凉水，把病恶化成肺炎，那我能治了。这个笑话展示了庸医误人。但是另一方面，从逻辑上来讲，医生则是一个把未知问题转化成已知问题的高手。

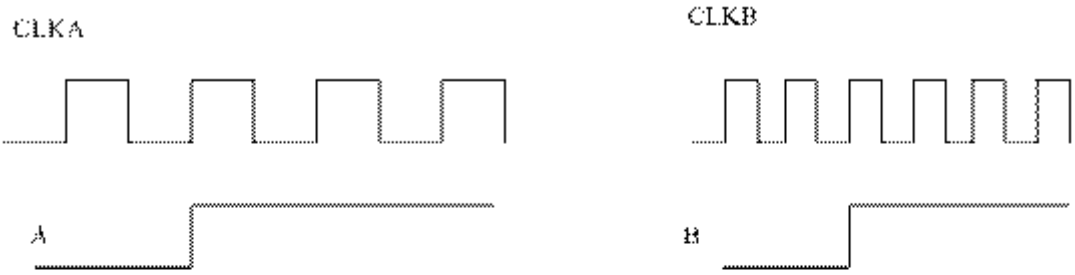
不说笑话，下面出两个题目，其分别是

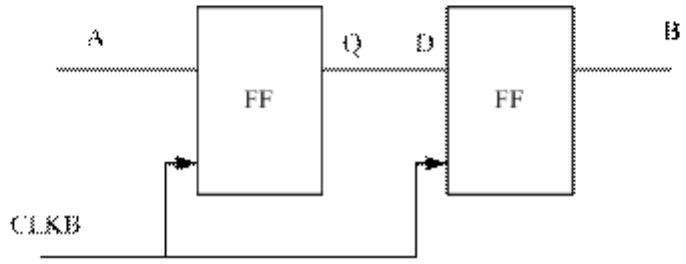
问题 1：运用数字电路，如何将一个时钟域的上升沿，转换成另一个时钟域的脉冲信号（单周期信号）。

问题 2：运用数字电路，如何将一个时钟域的脉冲信号（单周期信号），转换成另一个时钟域的上升沿。

可能乍一看，这两个题目优点难度，特别是第二个问题，答上的就更少了。那再出第三道题目，会不会让这个问题变简单些那。

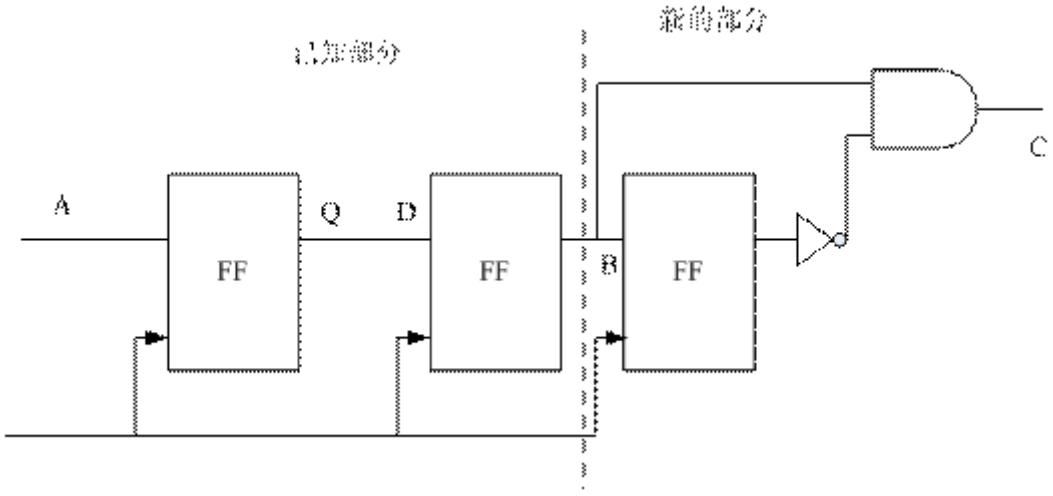
问题 3：运用数字电路，如何将一个时钟域的上升沿，转换成另一个时钟域的上升沿。



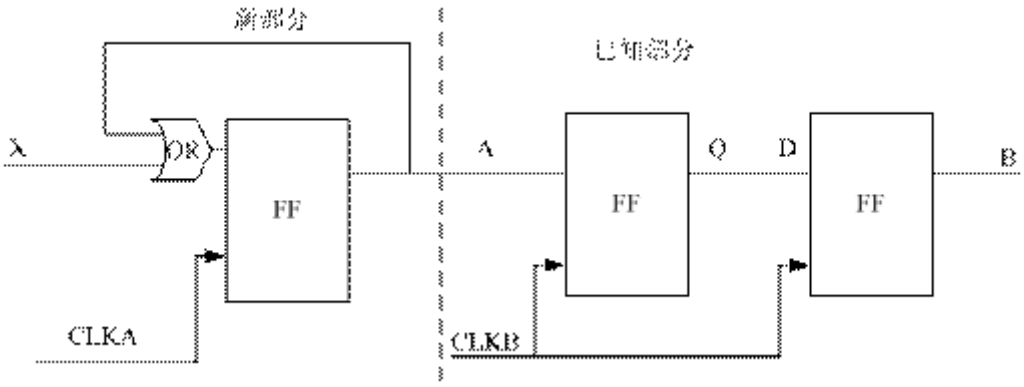


题目 3，,就会让人觉得太简单了，这既是异步信号的同步化，寄存两拍就可以了。这 CLKA 信号就被同步到 CLKB 的方法，有一定数字电路知识的都会 。不论 CLKA 与 CLKB 时钟频率的高低。

下面回到题目 1，我们按照那个庸医的做法，把未知问题转换成已知问题，那这个问题就转化成了两个部分，A 时钟域的上升沿转换成 B 时钟域的上升沿，，然后 B 时钟域的上升沿如何变成其单脉冲信号。所以问题就很简单了，最后的输出 B &! B_r(B 信号寄存一拍). 电路如下所示。



那回到题目 2.这个问题就转换成了 A 时钟的脉冲信号转换成 A 信号的上升沿，而 A 信号的上升沿，再转换成 B 信号的上升沿。



那么，如何将已是时钟信号的脉冲信号，转换成另一个时钟的脉冲信号？呵呵。

以上只是两个简单的电路，实际设计中，有许多可以进行设计，可以化繁为简或者化未知为已知的方法。

例如，实际设计中，经常有设计变更的情况，比如，原来输出的信号 A，但是需要输出信号 B。那可能大多数情况，模块不用重新设计，只需要在原有输出信号（或者数据）A 上，再添加相应的输出，或者封装一层接口，就可以快速满足需求。并且原有设计模块也可以复用，并且已验证充分。通过化未知问题为已知问题的方式，简化设计和验证，能够快速的设计需要。一方面，而这正是 IP 复用的好处，另一方面，原有设计如何达到 IP 复用的标准，也是值得探索的部分。

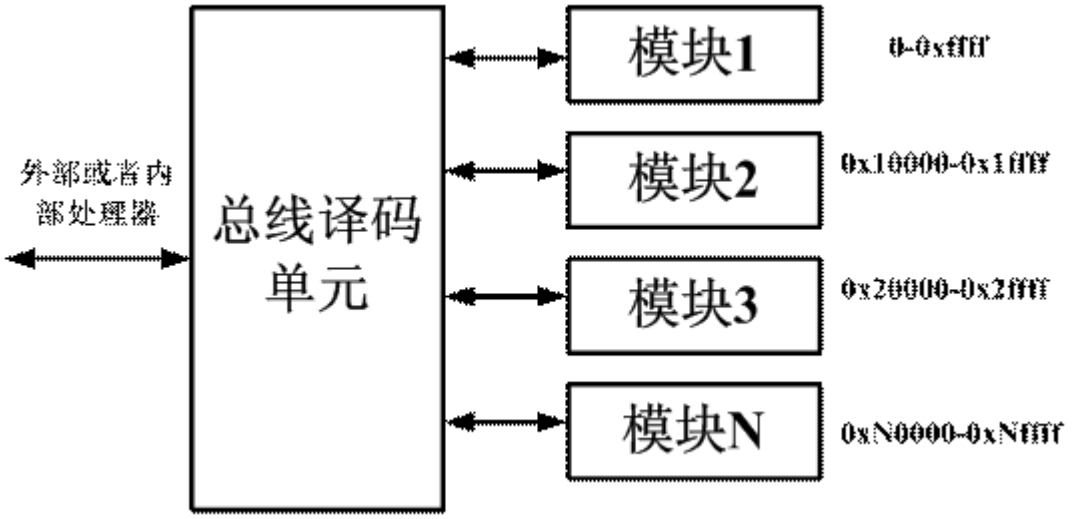
这个故事，其实还有另一版本，那就是锤子和钉子，那就是如果你已经有一把锤子，那就看什么都是钉子。如何将未知问题转化成你的锤子可以解决的钉子，则是就是设计复用的问题了。

“背靠大树好乘凉”--总线

如果设计中有多个模块，每个模块内部有许多寄存器或者存储块需要配置或者提供读出那么实现方式有多种，主要如下：

实现方式一：可以在模块顶部将所有寄存器引出，提供统一的模块进行配置和读出。这种方式简单是简单，但是顶层连接工作量较大，并且如果配置个数较多，导致顶层中寄存器的数目也会较多。

实现方式二：通过总线进行连接，为每个模块分配一个地址范围。这样寄存器等扩展就可以在模块内部进行扩展，而不用再顶层进行过多的顶层互联。如下图所示：



那如果进行总线的选择，那么有一种极为简单的总线推荐被使用，那就是 AVALON-MM 的总线

ALTERA 提出两种总线类型，分别是 AVALON-MM, AVALON-ST。分别用于连接 memeory 和数据流的传送 MM 不是你想的意思，其英文为 memory map。实现内存映射是其主要目的。主要信号包括如下表所示：

信号	例子
address	地址
read	读
readdata	读数据
write	写
writedata	写数据
waitrequest_n	等待信号

AVALON-MM 因此可以说是最简单实用的总线形态了。对于其操作来说，总线为同步类型的总线，写信号只需要在写使能有效时，同时提供写数据即可，而读数据等待信号无效时，读出数据有效。

同样数据类型读数据（readdata）和写数据（writedata）的宽度可以根据设计的需要灵活配置为（8,16,32----256---1024）BIT 等值。即可以支持非常大的位宽，但普通应用，只需要（8,16,32,64）BIT 等即可满足应用。

那假设总线宽度 32，基本上主流的数据总线的宽度。如果需要更细粒度的划分，确定读写某个字节有效，那么 byteenable 信号也是必须的。其需要 4bit 来标示 32bit（4 个 byte）中那个有效，每一 BIT 表示一个字节，因此如果要完全表示所有的字节有效，因此字节有效信号的宽度为(数据总线的宽度/8)。AVALON 还可以有 burst 的操作。主

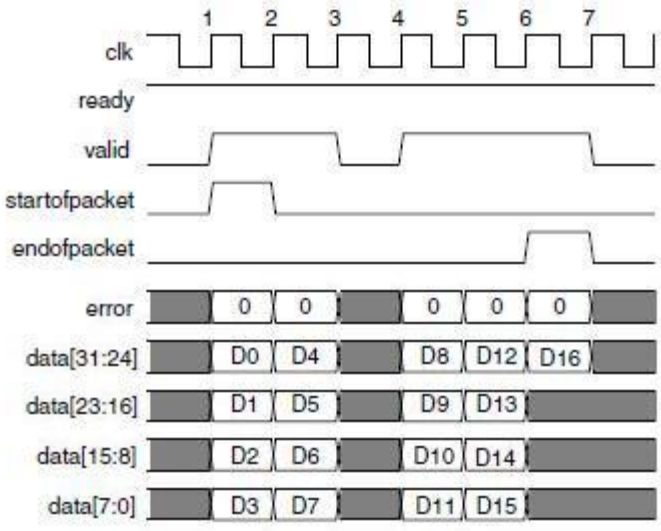
设备可以通过 burstcount 设备确定 brust 的长度，为 2 的 n-1 次方。

对于普通的应用，通过上述表格中的基本操作即可满足需求，这也正是 AVALON-mm 总线的优势。此外模块按此标准提供连接接口，各种模块可以挂在 NIOSII 的片上系统上。

如果模块之间为点对点的连接，同时传递大数据量的操作，那么的 AVALON-mm 总线就不太适合，因此 AVALON_streaming 总线就适合这种应用场景。

AVALON_streaming 总线本质上是一种同步并行总线，即在同步时钟状态下，使能有效代表传递数据有效。其基本信号如下表所示：

信号	例子
valid	主设备数据有效
data	数据信号
ready	从设备准备好
start of frame	帧开始
end of frame	帧结束
empty	帧结束信号时，空数据的位数



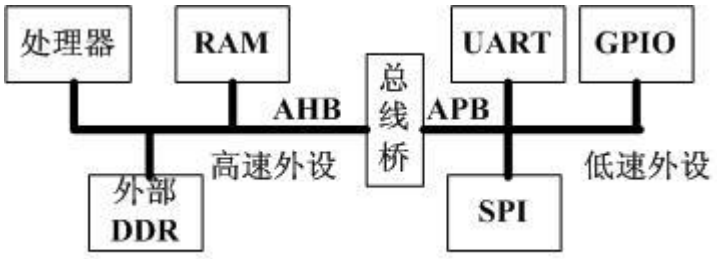
从上图中，可以看出各信号在数据传输中的作用，对于从设备获取数据的处理，就是 VALID 有效时，数据有效的采样操作，非常简单方便，易于处理。如果从设备设定 ready 永为 1，则表示没有反压的机制，则主设备，可根据自身收包情况一直向从设备发送数据包。此外还有其他辅助信号，可以根据设计需要进行添加。

使用总线使模块标准化，便于代码的移植和设计复用。同时标准总线的设定和统一定义也利于项目团队代码的标准化，便于理解和传播。

下文将介绍两种其他应用较广的总线形态，AHB（AMBA） 和 WISHBONE 总线（待续）。

如果说在 PC 时代，垄断 PC 江湖的是 WINTEL（微软和英特尔），那么在移动互联网时代，最具有这个潜质的就是谷歌的 andriod 操作系统和 ARM 芯片。基于 ARM 公司授权的各型 ARM 处理器，基本上在各型嵌入式终端设备占据了垄断地位。“背靠大树好乘凉”，因此，用于作为 ARM 处理做片上系统互联的 AMBA 总线标准亦成为业界应用最广泛的标准。

AMBA 总线事实上为三个总线标准的合集，分别是 AHB、ASB、APB。ASB 已逐渐被 AHB 所取代，现在使用最广泛的为 AHB 和 APB 总线，以及最新的扩展 AXI 总线。实际上，现今系统设计中，经常会借鉴 AHB 或 APB 总线标准，用于设计各种 IP 和片内模块之间的互联。首先来说 AHB 和 APB 总线，一家公司为什么会退出有两种类型总线，这是因为 AHB 一般认为其具备更高的性能和总线吞吐能力，而 APB 为低速总线，用于连接低速外设。两种总线互补，能够在性能和功耗方面进行互补。



如上图所示：AHB 总线与 APB 总线在一个嵌入式系统中的应用场景。 分别用于连接低速设备和高速设备。下表列举其一些主要的差别。

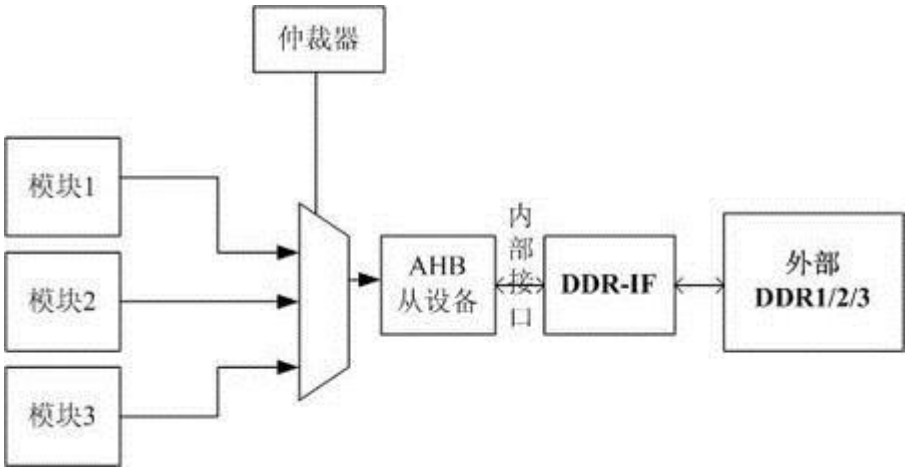
AHB	APB
多个主设备，多个从设备	一个主设备（总线桥），多个从设备
读写周期不固定，由从设备返回 hready 信号标示操作完成。	读写周期固定，两个周期完成
支持突发传输，传输类型由主设备	不支持突发传输
信号较多且复杂。	信号简单。

作为单次传输来说，AHB 与 APB 的主要区别在 AHB 周期不固定，操作完成标示由从设备返回 hready 标示，而 APB 周期固定。作为 burst 传输来说，AHB 支持增量和回环两种方式的突发。举例说，增量就是挨个地址自加，如总线宽度为 32，地址每次自加 4(字节)。而回环，比如当前地址为 0xA4，而回环突发操作位 0xA4, 0xA8, 0xAC, 0xA0。这种突发方式对于一些 cache 读写内存是非常有用的，这样可以把 0xA0-0xA15 十六个内存地址一次性读出。如果设计一条这样的 cache line，地址 0xA0-0xA15 其高位地址一致，便于匹配，这样这 16 个字节可以通过一次突发就能全部填满。（即回环

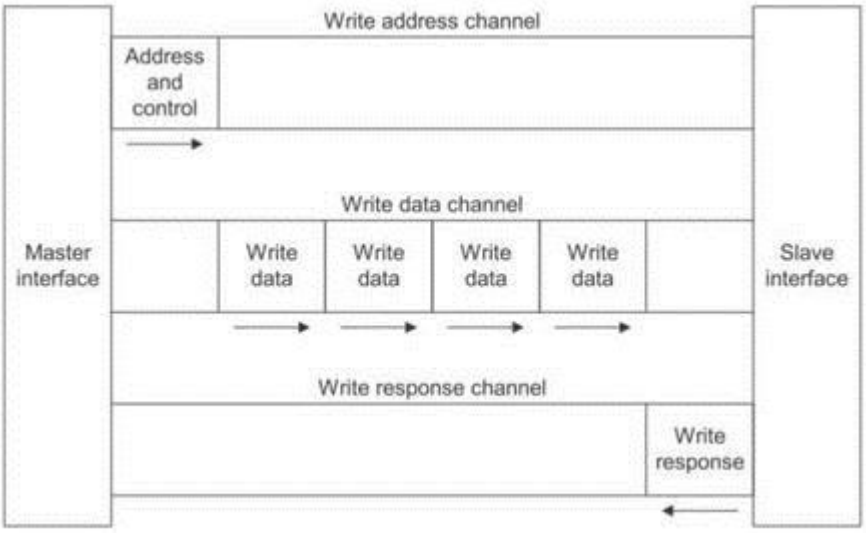
这种设计与处理器的 cache 结构是相关的，现在的 cache line 有逐渐扩大的趋势，一般 64 字节也较为常见）。

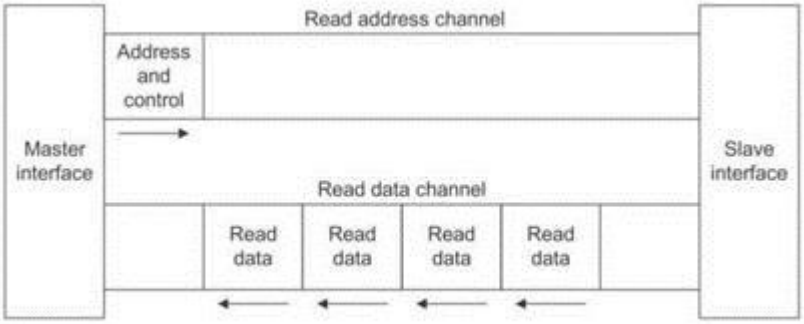
由于支持多个主设备和多个从设备进行交互，那么对于多个主设备之间就存在竞争。（从设备之间存在竞争否？从设备之间是通过地址区分的，被动接受主设备的访问，不会存储竞争的问题。）因此如何解决竞争，那就需要仲裁，即主设备谁需要访问总线，则发起 HBUSREQ，而仲裁器收到 HBUSREQ，返回给相应主设备 HGRANT。此时该设备才能访问总线。除此之外 AHB 还有其他一些信号，用于辅助整个系统的传输，感兴趣的同志，可以看一下 AMBA 的总线规范。值得一提的是，作为一个总线规范，其提供了全集的解决方案，而作为实现部分，只需要在满足规范的前提下，实现必要的功能即可，例如 AHB 总线中规定，其突发最大可 1K 字节，但是作为一个从设备，不一定需要支持这么大的操作，即实现功能可在总线框架内进行裁剪，选择实现支持的类型即可。

在 FPGA 内部设计中，经常有多个主设备访问同一从设备的例子，例如内部多个模块都需要访问外部存储器，其实现方式有多种，通过 AHB 的连接架构，可以实现一个标准、可扩展的接口单元，用于访问外部存储器。并且可以作为 IP 使用。AHB 从设备只需要根据需要，支持某些 burst 传输即可。



随着 SOC（片上系统的发展），对于高带宽、低延时的总线需求更加迫切，ARM 公司适时退出 AXI（AMBA3.0）作为扩展。





上图分别是 AXI 接口的读写操作，分别可以看出，对于 AXI 总线来说，其有 5 组独立的总线，分别是写地址，写数据，写响应，读地址，读数据信号。地址和数据信号分开，每组都有自己的控制信号。

每个通道中间没有时序关联，如何进行操作的？举例来说明，例如读数据操作，实际上，主设备向从设备中写了一个读的命令，包括读地址，burst 大小，方式等。收到后从设备按照相应的命令读取相应大小的数据，传回主设备，其操作可以简化的看做两个缓冲区类型的操作，主设备将读命令写入从设备的命令缓冲区，从设备取出后，根据命令将相应的数据返回给主设备的接收缓冲区中。这种操作的好处显而易见，能够最大限度的减少总线的开销，因此其读与读操作之间独立，不用等待读回，就可以发送下一次的读信号。写操作的流程亦然。

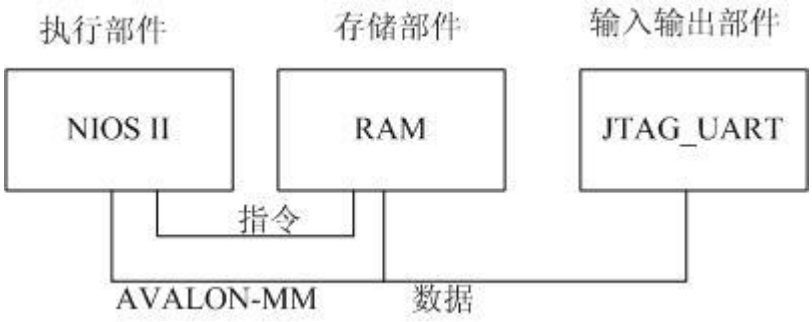
对 FPGA 设计来说，例如 xilinx 的接口 IP（DDR 例化时的接口），都已支持 AXI 的接口。FPGA 工程师熟悉相应的总线接口信号和特点，对于技术方案选择，IP 使用和验证，都是非常重要的。尽量在设计中选择标准总线接口，对于设计复用，模块共享来说，则是必由之路。而模块（IP）复用的益处随着设计不断增大将会不断显现。

PS：如要学习上述三种总线，推荐 AMBA 的手册，百度/谷歌各大搜索引擎均提供免费下载链接

片上系统

从最初的占地 170 平方的第一代 ENIAC 计算机开始，计算机开始了不断集成化、小型化的发展之旅。现今在单一芯片内部已经能够集处理器，存储，各型协处理器等，从而形成的强大的单芯片的片上系统（SOC），而这些片上系统已存在于生活的方方面面。因此 FPGA 内部支持片上系统，也算不上是新奇的事情了。ALTERA 和 XILINX 已各自推出了各自应用片上系统（FPGA 领域称之为 SOPC，因此其片上系统可以根据业务需求来定义）。

只需几 K 的资源，就能实现一个 SOC 的最小系统，对于 FPGA 工程师来说，没什么比这个更有吸引力了。那么，作为一个片上系统来说，其最小系统应该包含哪些：其至少需要三个部件，执行部件（处理器），程序执行部件（内部存储器），输出部件（输入输出单元）。（其分别相当于 PC 上的 CPU、内存条，键盘鼠标显示器）。下图所示在 ALTERA 的 QSYS 上实现 NIOS 的最小系统所需部件。



（1） 处理部件：NIOSII 为 ALTERA 器件中所专有的软核处理器，而 xilinx 所对应的为 microblaze 的软核。通过在 Qsys 的界面工具中提供许多 IP，而 NIOSII 也提供三个版本提供使用，分别是高速型，标准型，以及经济型。如果 FPGA 内部逻辑有限，可选择的经济型，其占用资源较少。如果需要内部资源丰富又需要运行嵌入式操作系统 uclinux 等复杂软件。则建议选择高速型，而要运行 linux 等操作系统，则在 NIOSII 高速型中配置 MMU 则是必须的。如无具体需求，则使用标准型即可。值得一提的是，NIOSII 为哈佛型体系结构，即数据和指令分开，从 Qsys 可以看出，其接口分为指令接口和数据接口。

（2） 存储部件:对于在 FPGA 内部实现 SOC 来说，片内的块 RAM 就是实现 SOC 内部程序与数据的存储空间。也可以使用片外的存储区，如片外 SRAM 或者 DDR 等。也可以作为程序和数据的存储空间。对于 NIOSII 处理器来说，只有选取了片内存储区或者片外存储区，才能设定程序中断向量和复位起始位置的存储区。另外，虽然 AVALON 总线支持数据总线和地址总线通过片上互联同一接口访问单端口 RAM。但建议使用时，例化为双 AVALON 接口的双端口 RAM，一则是因为一般存储区所需 RAM 深度够大，一般支持真双端口 RAM，另一方面，数据和指令分开，能够提升系统的性能。

（3） 输入输出部件：通常在嵌入式 SOC 系统中，最常用的输入输出部件就是串口（UART）。常常被应用于（打印 printf（），scanf（））函数的输入输出。如果系统设计了串口（一般为 SOC 系统中所必须的），则例化系统中的支持 avalon 接口的串口即可，如果系统中不幸没有，那么 ALTERA 公司提供了 JTAG-UART 接口提供给用户输入输出交互接口。即通过复用 JTAG 下载线来模拟串口的操作。如果系统中有多个输出输出设备，如有多个 UART，则在编程时，需在 BSP 的环境中设定，选择使用哪个 UART 作为系统的输出。

使用 ALTERA 的 Qsys 工具可以方便的在 FPGA 上构建 SOC 系统。只需选择相应的 IP（可以是系统自带，也可以使用自己构建支持 avalon-mm 接口的 IP）。通过系统的互联从而构成一个片上系统。图形化的界面只需通过 avalon 总线连接信号将 NIOSII 和外设连接在一起即可。连接完毕后，还需要下面操作：

（1） 为每个外设设定地址，例如上图中 RAM 和 JTAG-UART，每个外设都需要一个地址范围，可以点击系统中自动地址分配，也可以手动分配一个区间。只有为每个外设分配地址后（相当于整个系统的门牌号），处理器才能根据地址来访问各个外设。

（2） 如外设有中断，则为外设分配中断号，也可自动或者手动完成。如不分配中断的话，那么处理器访问外设，只有查询一种交互方式了。中断方式使用可以减少处理器的负载。

在 ALTERA 的 QSYS 工具中，硬件信息全部存储在 sopcinfo。主要是包括各个外设的地址信息等，用于产生 system.h。也就是说，作为软件和硬件的交互的渠道是每个外设的基地址，中断，和内部寄存器等信息。系统构建结束后，剩下的就可以软件编程了，运行在搭建的 SOC 系统上的第一个“hello world”的程序。

对于现在 FPGA 上的 SOC 设计，厂商为了其方便易用，做了大量的工作，只需按照其指南一步步，就可以实现相应的设计，但同时，也限制了对其基本原理的深入的理解。什么事情都有其两面性，作为 FPGA 工程师，SOC 的原理则应该能够重点关注，这样不论是 ALTERA 还是 XILINX 其基本原理也是一致的。如不使用厂家的处理器核（NIOSII、microblaze），也可以使用其他的软核（如 51 等，ARM）在 FPGA 上实现 SOC 系统。只不过原来工具做的工作，就需要手动来完成了。

鸡肋

通常认为，SOPC 是 FPGA 设计中的鸡肋，“弃之可惜，食之无味”。诚然，SOPC 一直不是 FPGA 的主流应用设计，制约主要因素则是性能，因为作为处理器使用时，处理器主频是其应用范围的瓶颈（SOPC 的软核处理器一般运行几十兆到百兆，而一般的嵌入式处理器系统在几百兆到 Ghz 的主频）。但是若因此说成“鸡肋”，也确实夸张。厂家推出 SOPC 的设计，其优点主要有一下几点，其一：是差异化竞争的需要。

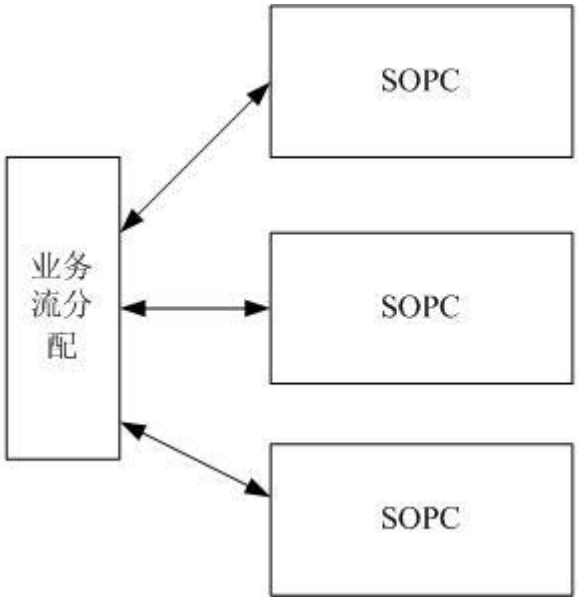
其二，扩大应用范围，争取更多的软件工程师能够从事 FPGA 设计。其三，可以替代低性能处理器，减少板级的面积和 BOM 成本。

那么，那么 FPGA 内部基于软核处理器的系统的主要应用场景有哪些，总结如下：

(1) 管理配置：对于性能无要求的管理配置功能。如某外接芯片或 FPGA 内部 IP 在其工作之前，需进行初始化和管理配置，而初始化的模块较多或者管理配置较为复杂，此种情况下，用软件处理更为方便合理。因此，假如 FPGA 内部资源较为充足，通过内部建立 SOPC 系统，利用片内的软件给多个外围模块或者内部 IP 进行初始化的配置管理，即省去配置 CPU，减少板级面积，也能便于配置的修改，同时还可以作为前面讲的可测性设计的一部分，用于内部各模块计数统计，功能测试等。

(2) 配合专用硬件加速单元使用： SOPC 系统可以通过总线扩展专用协处理单元。即将关键模块硬件化实现，实现高速的处理。举例说，如实现图像处理功能，而 SOPC 的软件性能不能支持高分辨率图像的处理能力，则可以通过逻辑实现专用的图像处理算法，通过总线接口与 SOPC 系统连接。，SOPC 只作为数据的管理和调度使用。此外厂商还这么提供了 SOPC 优化手段，例如通过算法指令分析，确定最多的操作，通过专用指令硬件实现，通过在程序中调用专用指令，也是能够提升性能的方式之一。

(3) 多核并行：这里多核并行通常意义不同，指通过多个 SOPC 系统，并行执行，可以提升系统的性能。对于单指令集多数据流的业务，通过将业务流分配到多个 SOPC 上，通过多个 SOPC 系统并行处理的方式，来提升整个系统的性能。此种情况下，通过多个 SOPC 系统并行处理，需满足几个条件：（1）即业务之间没有关联性，不需要再多个 SOPC 之间进行数据的交互，否则会影响整个系统的性能。（2）程序区不能太大，最好全部存储在片内 RAM 中。而不用占用外部存储区（DDR 或者 SRAM），否则，多个片上存储系统争抢外部存储区，可能会造成系统性能的瓶颈，如需存储在片外，则接口竞争部分则是关键设计。



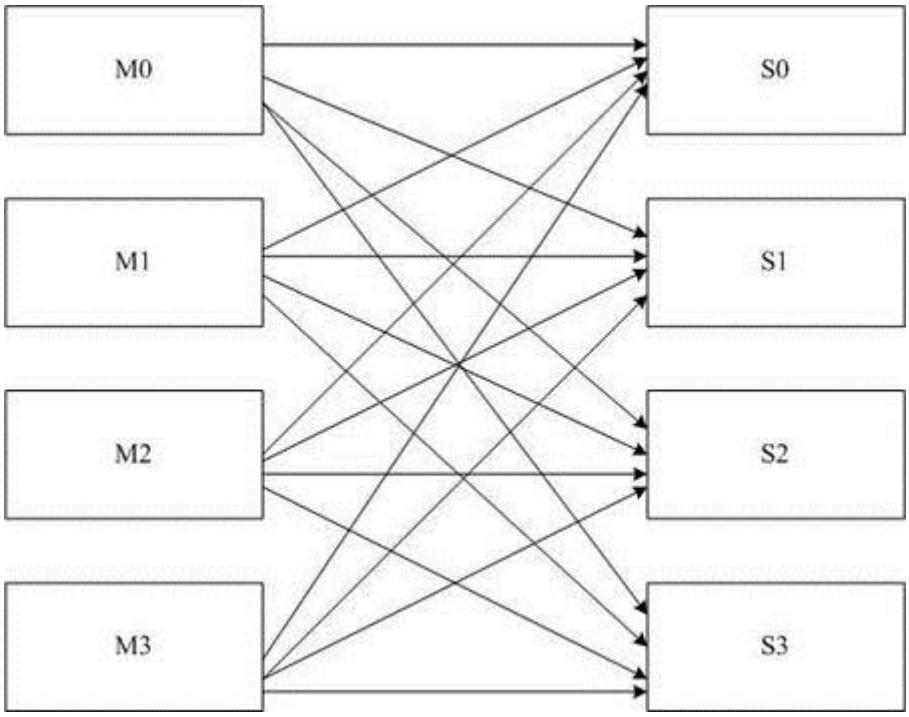
SOPC 就像一个偏科生，其优点和缺点都是那么明显，即其强大的灵活性和可编程性，配置其可怜的主频。但是通过一系列提升性能的手段，SOPC 在某些业务中也可以大显身手，尤其是需求频繁变更的业务，将变化部分通过软件实现，而不变部分硬件化实现，可以更快的满足市场的需求。

最后，软核 CPU 处理能力的瓶颈，也促使现有 FPGA 厂商提供了基于硬核 CPU 的 SOC+FPGA 的解决方案，而这些的努力将促进 FPGA 应用场景的扩展。FPGA 广阔天地，大有可为。

交换矩阵

如果在 FPGA 设计中，需要多端口，大数据量的交换，那么交换矩阵则是一个不错的实现方案。交换矩阵使用的目的主要有几个，一，灵活的端口转发。通过交换矩阵灵活实现数据流的灵活交换，减少外部负责控制。 二，高效的转发效率，交换矩阵能够实现通常单一总线不能达到的转发效率，满足高吞吐量的系统的需要。三，系统设计以交换矩阵为中心，便于 IP 集成和模块复用。

交换矩阵的实现方案较为复杂，最早的交换沿袭了共享总线式的架构，因此对于某个端口需要传输，则其他端口只能阻塞，等待总线空闲后再进行传输。而交换矩阵则是一个全互联的结构。如下图所示，如有 4 个输入，4 个输出的交换矩阵，可认为是一个 4 端口的交换单元，每个端口包含一个发送接口和一个接收模块，如端口 0 就包含发送模块 m0 和接收模块 s0。



假设每路传输的速率为 N，则整个交换矩阵的传输速率为 4N。如何实现一个简单的交换矩阵。首先可以将整速率个设计分割。将整个设计分割为接受和发送两个模块。整个交换单元可以划分为四个部分，分别是，发送模块，仲裁模块，交换模块，接收模块。

（1） 发送模块，首先根据某端口接收数据后，根据该数据帧要转发的端口，发起请求信号。

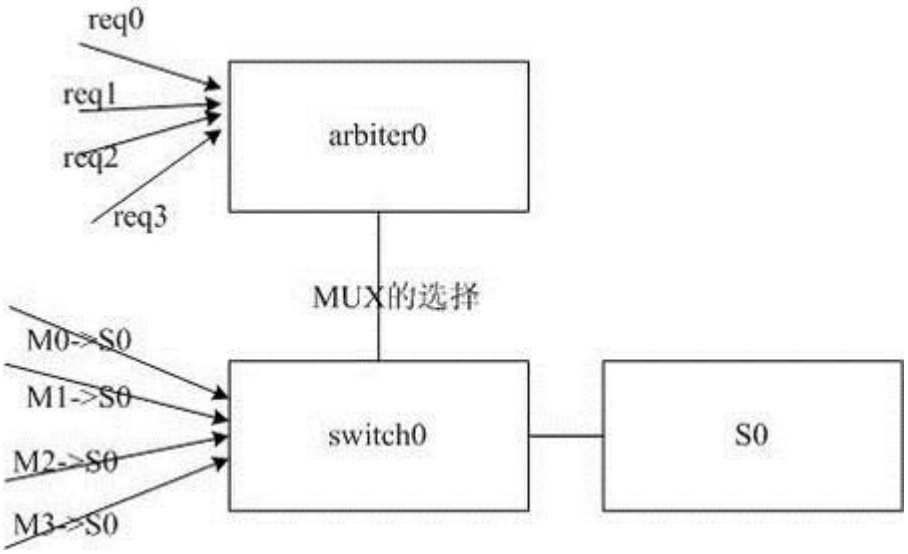
（2） 仲裁模块：根据请求信号，接收模块的忙闲状态，及各发送模块的优先级，确定当前的响应信号，如果当前的接收模块忙（上次传输未完成），则需要阻塞，等待上次传输完成（复杂的设计，可以保证高优先级能够打断当前传输，直接传输高优先级数据流，高优先级完成后，再恢复原有传输，但这种方式设计较为复杂，仿真验证的难度也较大，不建议使用）。

（3） 交换模块：根据仲裁信号确定发送模块转发的端口，交换模块本质上是多选一的 MUX，而 MUX 的选择信号，则是由仲裁模块来进行选择。

（4） 接收模块：接收交换模块交换后的数据流，向仲裁模块返回当前模块的忙闲状态（正在接收传输信号，为忙状态，而当前无传输状态，则为闲状态）。

仲裁模块的仲裁机制，一般可以使用简单的 round-robin 的设计，即轮流最高优先级。也可以通过设计带加权的优先级，保证更高优先级的端口优先进行传输。

仲裁模块的仲裁设计可以分为多种，一种是整个交换矩阵使用同一个仲裁单元，每个发送模块只使用一个请求信号及请求端口号连接到仲裁模块。这种设计对整个仲裁模块设计难度较大。另一种设计，如下图所示，即每个接收单元，配置一个仲裁模块，和一个交换模块。而每个发送模块根据要发送的端口，使用多个发送的请求信号。此种设计便于系统扩展，也可简化设计。



不仅是用于高速多端口转发的数据流传输采用交换矩阵。现在，大多高速总线机制（如 N 主设备，M 从设备之间进行数据的交换）也采用类似交换矩阵式的结构，每个从设备的连接都是多个主设备通过 MUX 来进行连接，这样保证多个主设备访问不同从设备时，可以实现并行的数据交换（如主设备 M0 访问从设备 S1，主设备 M1 访问从设备 S2，可以同时进行总线操作）。这是单一竞争式总线所不能达到的优势，但是，设计占用的逻辑量也会增加。性能的增加带来的负面影响通常就是逻辑的增加。

交换矩阵通常在基于数据包转发的 FPGA 设计应用中，交换矩阵的交换灵活性增加，也会增加设计复杂度，如果对于传输效率不需太多的需求，可以采用系统复用的方式（前文

介绍），通过一个复用模块轮流接收各个发送端口的数据流，，再根据端口转发到各个从设备中，此种方式设计简单，但是此复用模块则会成为系统的瓶颈。根据系统的设计需求，选择适合的设计，达到性能和逻辑方面的平衡，是体现 FPGA 设计艺术之一。

控制

本质上说，FPGA 的模块设计就是将输入转化成想要得到的输出结果。而除了某些简单模块，即在当拍内完成，即将输入进行逻辑操作后，再输出。（如简单加法器等）。其余大部分的设计需要通过时序逻辑和组合逻辑混合实现，时序逻辑带来就是延迟起效的问题，举例说，如实现某个信号（start）起效后，接下来五个周期需要分别进行五种操作，分别是 op0,op1,op2,op3,op4 等等。如何进行控制，这就是每个工程师要面对的问题。

对于简单控制，分别可以采用计数和移位寄存器的方式来解决问题。而对于较为复杂的控制，则需要设计状态机来解决。下面将分别介绍

计数器：对于上述操作来说，start 起效后，可以通过计数实现，设置寄存器 count[2:0]，有效信号开始时计数自加。 计数的方式带来的问题就是，计数从零开始还是从 1 开始，假如计数器初始化为 0，则从 0-4 状态可以分别输出 op0,op1,op2,op3,op4，但是在无有效信号时，计数会保持 0，从而造成 op0 的输出。 上述举例虽然简单，但是确实很多初学者或者工程师在仿真时会经常会犯的错误。从设计来说，计数需要考虑初始值对于输出的影响。同样计数带来的另一个问题就是，从零开始的计数会导致设计与实际不一致，例如，一个信号 9 拍后拉低，但从零计数到 8 时，已经到 9 拍了（0-8），这种设计会导致命名 count==8 与 9 拍存在不一致的现象。当然也可以从 1 计数到 9，这样状态在 count==9 时触发。这样就会初始化需要复位寄存器为 1。当然这个问题大端和小端的争斗一样，没有终点。一个设计中如果多种计数来驱动计数的话，就需要特别小心这个问题计数。当然也可把问题交给仿真器，仿真时根据波形调整，计数的状态。

移位寄存器：如采用移位寄存器，根据上述例子，则 start 信号有效后，设计 5bit 的移位寄存器 flag[4:0]分别利用寄存器的某 BIT 来控制输出，从而在每 BIT 有效时，分别输出 op0,op1,op2,op3,op4。假设此种状态较少，FPGA 寄存器资源较为丰富，因此利用移位寄存器是一个不错的注意。

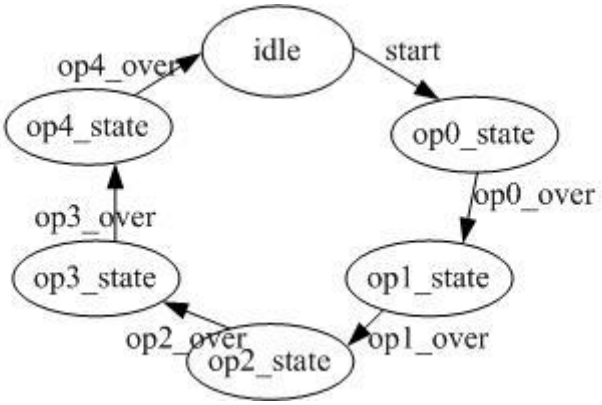
```
assign op4 = ( count == 3'b100) ;
assign op4 = flag[4] ;
```

比较上述两种输出，则可以看出，通过计数的方式占用输出资源较多，而移位寄存器在此种应用下，占用逻辑就相对简单。（仅针对小规模的计数来说，对于超过 16 的计数，

则使用计数器更优）。另外，通过移位寄存器可以方便的进行时序控制，不用纠结从零开始还是从 1 开始的问题，在某些简单的处理下能够达到更小的面积和更快的时序。

对于复杂的控制，则状态机，就是必须的。对于 FPGA 实现状态机，其实并不需要那么多的设计的方法。主要就是两个要点。（1）独热码。（2）三段式。

对于第一点来说，独热码，因为 FPGA 内部寄存器资源较多，另外独热码将会带来额外的面积和时序优化的好处。则以上述例子为例，增加状态转移的触发信号，状态转移图如下所示：



状态独热码（也可以用 define　localparam）建议使用 parameter 或者 localparam

```
parameter     idle == 6'b000001,
              op0_state == 6'b000010,
              op1_state == 6'b000100,
              op2_state == 6'b001000,
              op3_state == 6'b010000,
              op4_state == 6'b100000;
```

三段式结构如下

```
//（1）当前状态
always@(posedge sys_clk or negedge rst_n)
      if(!rst_n)
        cs_state <= idle;
      else
        cs_state <= ns_state;

//（2）下一状态的赋值
always@(*)
     case(cs_state)
       idle : if(start)
                ns_state =   op0_state;
              else
                ns_state =   idle;
```

```
        op0_state :
            if(op0_over)
              ns_state =   op1_state;
            else
              ns_state =   op0_state;

        op1_state :
            if(op1_over)
              ns_state =   op2_state;
            else
              ns_state =   op1_state;
        op2_state :
            if(op2_over)
              ns_state =   op3_state;
            else
              ns_state =   op2_state;
        op3_state :
            if(op3_over)
              ns_state =   op4_state;
            else
              ns_state =   op3_state;
        op4_state :
            if(op4_over)
              ns_state =   op4_state;
            else
              ns_state =   idle;
        default ns_state = idle;

      endcase

//（3）输出状态
assign out1 = (cs_state ==   op0_state);
always@(posedge sys_clk or negedge rst_n)
      if(!rst_n)
        out2_reg <= 1'b0;
      else if (cs_state ==   op2_state)
        out2_reg <= 1'b1;
      else
        out2_reg <= 1'b0;
```

上述例子，介绍独热码和三段式。三段式的好处不用说，就是逻辑清楚。可以看出 out1 输出为组合输出。out_2_reg 为寄存输出。那么独热码在 FPGA 内部的优势又有哪些？

（1）综合后，逻辑简单

例如 assign out1 = (cs_state ==   op0_state); 综合后的电路等同于

assign out1= cs_state(0) ;//可以看出无逻辑消耗

而 out2_reg 的电路等同于 将 cs_state（2）寄存一拍，只需一个寄存器的消耗

（2）时序优化。

从上述同样得出结论，如果是使用某状态 cs_state(n)作为其他信号的输入来说，其本身为寄存器信号，因此关键路径就会减少一级。可能运行较快的频率就会增加。如不是独**，对比这两条语句 cs_state = 3 与 cs_state(3) 一个是组合输出，一个寄存器输出。其不同也就是上述计数与移位寄存器的区别一致。

那么一般状态机会产生的错误会有哪些那？

首先；就是状态不全产生 LATCH，前文已述，这是 FPGA 设计的大敌，解决这个问题的方法可以通过所有分支都设定确定状态，如上例中。有没有更简单的方式？

其此：状态机上述描述，并不直观的显现综合后电路的描述，有没有更直接的 rtl 的描述，一眼就能看出独热码的特征和好处？

最后：状态机是一个较为成熟技术，还会有哪些值得关注的地方？

这些问题，下节再述。

首先依次回答上篇提出的几个问题：

第一个问题：如何避免状态机产生 lacth 示例如下，通过在 always（*）语句块中，添加默认赋值，ns_state = cs_state;

```
always@(*)
    ns_state = cs_state;
    case(cs_state)
      idle :
            if(start)
            ns_state =  op1_state;
      op0_state :
          if(op0_over)
            ns_state =  op1_state;
      op1_state :
          if(op1_over)
            ns_state =  op2_state;
      op2_state :
          if(op2_over)
            ns_state =  op3_state;
      op3_state :
          if(op3_over)
            ns_state =  op4_state;
      op4_state :
          if(op4_over)
```

```
            ns_state =  op4_state;
       default ns_state = idle;
      endcase
```
这样，分支没有赋值的语句全部会赋值为 ns_state = cs_state ;以 IDLE 状态为例，当前 cs_state 为 idle。因此实际上 ns_state=idle。这条语句的作用，即在没有分支赋值的情况下，默认赋值当前状态。

第二个问题：更直观的独热码的状态机实现方式。示例如下

```
//状态定义
parameter   idle == 0,
          op0_state == 1,
          op1_state == 2,
          op2_state == 3,
          op3_state == 4,
          op4_state == 5;
//（1）当前状态
always@(posedge sys_clk or negedge rst_n)
      if(!rst_n)
        cs_state <=  6'b000001;
      else
        cs_state <= ns_state;
//（2）下一状态的赋值
always@(*)
       ns_state = 0;
     case(1)
       cs_state[idle] :
            if(start)
              ns_state[op0_state] =  1'b1;
            else
              ns_state[idle] = 1'b1;
       cs_state[op0_state] :
            if(op0_over)
              ns_state[op1_state] = 1'b1;
            else
              ns_state[op0_state] = 1'b1;
       cs_state[op1_state] :
            if(op1_over)
              ns_state[op2_state] = 1'b1;
            else
              ns_state[op1_state] = 1'b1;
       cs_state[op2_state] :
            if(op2_over)
              ns_state[op3_state] = 1'b1;
            else
              ns_state[op2_state] = 1'b1;
```

```
        cs_state[op3_state]:
              if(op3_over)
                ns_state[op4_state] = 1'b1;
              else
                ns_state[op3_state] = 1'b1;
        cs_state[op4_state] :
              if(op4_over)
                ns_state[idle]= 1'b1;
              else
                ns_state[op4_state] = 1'b1;
        default    ns_state[idle]= 1'b1;
      endcase
//（3）输出状态
assign out1 = cs_state [op1_state];
always@(posedge sys_clk or negedge rst_n)
      if(!rst_n)
        out2_reg <= 1'b0;
      else if (cs_state[op2_state])
        out2_reg <= 1'b1;
      else
        out2_reg <= 1'b0;
```

上例中，定义状态机是，同样定义为 0,1,2,3,4,5,6 的值而不是独热码，只不过使用时，这些值用于赋值的为状态机的某一 bit。值得注意的是，在 ns_state 通过组合逻辑赋值时，首先需要将 ns_state 赋值为零，也就数说，除了需要赋值为 1 的状态，其他都需要赋值为 0。但此种编码方式下，就需要谨慎对待分支赋值不全的情况，因此此时，ns_state 会赋值为 0。产生非想要的后果。

通过第三段的输出赋值可以看出，其输出分别是 cs_state [op1_state]的直接输出，cs_state[op2_state]的寄存后一拍再输出。其产生的效果与前文（控制-上）中介绍的产生的效果是一致的。因此可根据习惯，选择一种的实现即可。

最后一个问题：状态机使用可以直观的通过定义的状态来控制各个信号的输出和控制，独热码本质上还是将状态机转变成一组某一时刻只有一个起效的寄存器，换个角度可以看做加强版的移位寄存器。其他需要注意问题有，

（1）如果状态机定义而没有使用，综合工具将综合掉此状态，因此综合后的状态会和工程师所定义的状态不同。通过检查综合文件，就能得知其对应关系，避免通过嵌入式逻辑分析仪抓信号时，信号与实际不一致现象。

（2）如通过单周期信号启动状态机，要注意，如单周期信号起效时，状态机未跳转回有效状态，会导致出错，应该将单周期信号转换成电平信号，等启动有效后再将电平信号拉低。

（3）状态如出现未定义的状态（如独热码出现全零状态），latch 是其中一个主要的原因，上电后未有效复位也会导致此种可能。

（4）状态机结合移位寄存器可以有效减少状态的数目。例如某个状态中，每个周期要进行多个操作，不需要再分解成多个小状态，通过移位寄存器来控制这些状态的操作能够简化设计。

总之：状态机是 FPGA 设计的一项基本设计，而“独热码”和“三段式”的设计能够使设计达到事半功倍的效果。也是事实上行业内的 FPGA 设计的标准写法。而标准化能使难以理解的 FPGA 的代码能够有更好的移植和 IP 化的基础。

管脚

管脚是 FPGA 重要的资源之一，FPGA 的管脚分别包括，电源管脚，普通 I/O，配置管脚，时钟专用输入管脚 GCLK 等。

（1）电源管脚：

通常来说： FPGA 内部的电压包括内核电压和 I/O 电压。

1.内核电压：即 FPGA 内部逻辑的供电。通常会较 I/O 电压较低，随着 FPGA 的工艺的进度，FPGA 的内核电压逐渐下降，这也是降低功耗的大势所趋。

2.I/O 电压 （Bank 的参考电压）。每个 BANK 都会有独立的 I/O 电压输入。也就是每个 BANK 的参考电压设定后，本 BANK 上所有 I/O 的电平都与参考电平等同。

是否可以通过约束来设定 IO 管脚的输出电平那，答案是否定的，如下例所示

```
set_instance_assignment -name IO_STANDARD 3.0-V LVCMOS -to pinA
set_instance_assignment -name IO_STANDARD 3.3-V LVCMOS -to pinB
```

不论设定为多大的电平，IO 的输出与 BANK 的参考电压保持一致，也就是说，PINA 和 PINB 的电平与其 BANK 的电平保持一致，而不是所约束的那样一定会是 3V 或 3.3V 的电平。那是否意味着这种约束没有作用？

如果约束同一 BANK 上的管脚为不同电平，如 PINA 和 PINB 在同一 BANK，但是电平不同，则 EDA 工具会报错。可以起到错误检查的作用。（同一 bank 上电平要一致，但是类型可以是多种，例如 CMOS，TTL 等）。

对于复杂的 FPGA 内部，一般来说 PLL 也会都单独的供电，并且其内部包括数字电源和模拟电源。

SERDES 一般也需要独立供电。一般支持高速 SERDES 的 FPGA 器件都都有独立的供电管脚，一般也都有独立的时钟管脚（一般为差分时钟）。

（2） 配置管脚：

FPGA 的配置管脚每个 FPGA 都需要，为了支持多种配置方式，例如 JTAG，从串、从并、主串、主并等。值得注意的是，对于其配置管脚的控制信号来说，是专用管脚，不能用做普通 I/O，而其数据信号，可以用作普通 I/O。在管脚资源较为紧张时，可以复用配置信号的数据信号作为普通 I/O 来用。

(3) 普通 I/O：

FPGA 的 I/O 是 FPGA 管脚上较为丰富的资源。也是做管脚约束时最常用的资源。对于例化 IP 来说（例如 serdes 和 DDR2/3 等），需要使用 EDA 工具给出了 I/O 约束。如果修改则需要预先编译进行评估。一般来说，DDR 的接口信号最好能在一个 BANK 上约束，如果不能则其控制信号要约束到同一 BANK 上，否则导致 EDA 工具布局布线报错。

对于 FPGA 的普通 I/O，可以设定包括管脚电平类型（LVTTL，LVCOMS，SSTL,HSTL 等等），还包括端接大小，驱动电流，摆率等参数。

对于 FPGA 引脚来说，通过阻抗匹配的设置（altera 的 OCT，xilinx 的 DCI 的设置）。可以有效的减少板上电阻的数目，降低 BOM 成本。端接的设置可以阻止阻抗不连续导致信号反射，保证信号完整性。

对于普通信号来说，一般不需要每个都设置阻抗匹配，，只有板级布线长度的电信号传输时间超过高速信号的时钟周期的 0.1 倍时，才需要设置端接。简单来说也就是只有高速信号，且信号输出和输入距离较远时，才需要使用端接，一般是 FPGA 连接外部 DDR 等高速器件时。对于使用外部校准的 RUP 和 RDN 电阻来说，其 RUP 和 RDN 电阻是整个电路可靠性的关键点。例如：在以一批 FPGA 板卡中，测试发现只有某一块接口不通，FPGA 工程师调试时发现，只要把接口 I/O 设定为 CMOS 电平，而不使用 DCI 的端接，则所有板卡全部能够通信正常，后来发现该板卡 RUP 上拉电阻失效，从而导致接口电路没有上拉而接口出错。也就是说，I/O 的电平设置，以满足设计需要为主，而不用锦上添花。增加的额外电路就会导致额外的失效点。

（4）时钟管脚

FPGA 内部的时钟，都需要通过专用时钟管脚连接内部 PLL 或者 DCM 等专用时钟处理单元，从而接入内部高速时钟网络。在早期的 FPGA 中内部时钟资源有限，专用处理单元也有限，需要严格的规划 PLL 等时钟处理单元和全局时钟资源，随着 FPGA 技术的发展，这个功能逐渐弱化，但是早期规划也是必须的。

值得注意的是，对于一些外部同步信号的输入，其时钟没有连接到专用时钟管脚上，只用于采样当前的同步信号，因此不用接入全局时钟网络，设计上也是允许的，需要约束其管脚不使用全局时钟资源。否者，EDA 工具会报错，提示其作为时钟输入，而没有接在专用时钟管脚上。

正如本文开始所说，管脚是 FPGA 的重要资源，FPGA 工程师熟悉管脚特性和电路设计的基础知识，对于 FPGA 系统设计和板级电路的调试都是有益的。

PS：问题回复

QUARTUS 能否设定 I/O 信号的上拉和下拉电阻的大小？

1)对于设定的输出信号来说，其是有电平格式的如 set_instance_assignment -name IO_STANDARD LVCOMS -to pin 这种情况下，其电平格式就是 LVCOMS。没有上拉或者下拉的设置（也就是说 coms 电平不包括上下拉电阻的设置）。但是可以设置其输出电流，如 set_instance_assignment -name CURRENT_STRENGTH_NEW 12MA -to pin 板级电路上信号不到位，很多情况下，是驱动能力的问题。也可以设置其输出端接电阻的大小（不是上下拉）。set_instance_assignment -name OUTPUT_TERMINATION "SERIES 50 OHM WITH CALIBRATION" -to pin 以及其他属性等等（2）对于未约束的信号时可以设定其上下拉状态的。如：set_global_assignment -name RESERVE_ALL_UNUSED_PINS "As input tri-stated with weak pull-up" 其他几种状态分别是 as inputs that are tristated, as outputs that drive ground, as outputs that drive an unspecified signal, asinput tri-stated with bus-hold