

# FPGA 基础知识

## 1、查找表 LUT 和编程方式

### 第一部分： 查找表 LUT

FPGA 是在 PAL、GAL、EPLD、CPLD 等可编程器件的基础上进一步发展的产物。它是作为 ASIC 领域中的一种半定制电路而出现的，即解决了定制电路的不足，又克服了原有可编程器件门电路有限的缺点。

由于 FPGA 需要被反复烧写，它实现组合逻辑的基本结构不可能像 ASIC 那样通过固定的与非门来完成，而只能采用一种易于反复配置的结构。查找表可以很好地满足这一要求，目前主流 FPGA 都采用了基于 SRAM 工艺的查找表结构，也有一些军品和宇航级 FPGA 采用 Flash 或者熔丝与反熔丝工艺的查找表结构。通过烧写文件改变查找表内容的方法来实现对 FPGA 的重复配置。

根据数字电路的基本知识可以知道，对于一个  $n$  输入的逻辑运算，不管是与或非运算还是异或运算等等，最多只可能存在  $2^n$  种结果。所以如果事先将相应的结果存放于一个存储单元，就相当于实现了与非门电路的功能。FPGA 的原理也是如此，它通过烧写文件去配置查找表的内容，从而在相同的电路情况下实现了不同的逻辑功能。

查找表 (Look-Up-Table) 简称为 LUT，LUT 本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成是一个有 4 位地址线的 RAM。当用户通过原理图或 HDL 语言描述了一个逻辑电路以后，PLD/FPGA 开发软件会自动计算逻辑电路的所有可能结果，并把真值表（即结果）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。

下面给出一个 4 与门电路的例子来说明 LUT 实现逻辑功能的原理。

例 1-1：给出一个使用 LUT 实现 4 输入与门电路的真值表。

表 1-1 4 输入与门的真值表

实际逻辑电路		LUT的实现方式	
a, b, c, d输入	逻辑输出	RAM地址	RAM中存储的内容
0000	0	0000	0
0001	0	0001	0
...	...	...	...
1111	1	1111	1

从中可以看到，LUT 具有和逻辑电路相同的功能。实际上，LUT 具有更快的执行速度和更大的规模。

## 第二部分： 编程方式

由于基于 LUT 的 FPGA 具有很高的集成度，其器件密度从数万门到数千万门不等，可以完成极其复杂的时序与逻辑组合逻辑电路功能，所以适用于高速、高密度 的高端数字逻辑电路设计领域。其组成部分主要有可编程输入/输出单元、基本可编程逻辑单元、内嵌 SRAM、丰富的布线资源、底层嵌入功能单元、内嵌专用单元等，主要设计和生产厂家有 Xilinx、Altera、Lattice、Actel、Atmel 和 QuickLogic 等公司，其中最大的是 Xilinx、Altera、Lattice 三家。

FPGA 是由存放在片内的 RAM 来设置其工作状态的，因此工作时需要对片内 RAM 进行编程。用户可根据不同的配置模式，采用不同的编程方式。FPGA 有如下几种配置模式：

- 1、并行模式：并行 PROM、Flash 配置 FPGA；
- 2、主从模式：一片 PROM 配置多片 FPGA；
- 3、串行模式：串行 PROM 配置 FPGA；
- 4、外设模式：将 FPGA 作为微处理器的外设，由微处理器对其编程。

目前，FPGA 市场占有率最高的两大公司 Xilinx 和 Altera 生产的 FPGA 都是基于 SRAM 工艺的，需要在使用 时外接一个片外存储器以保存程序。上电时，FPGA 将外部存储器中的数据读入片内 RAM，完成配置后，进入工作状态；掉电后 FPGA 恢复为白片，内部逻辑 消失。这样 FPGA 不仅能反复使用，还无需专门的 FPGA 编程器，只需通用的 EPROM、PROM 编程器即可。Actel、QuickLogic 等公司还 提供反熔丝技术的 FPGA，只能下载一次，具有抗辐射、耐高低温、低功耗和速度快等优点，在军品和航空航天领域中应用较多，但这种 FPGA 不能重复擦写，开发初期比较麻烦，费用也比较昂贵。Lattice 是 ISP 技术的发明者，在小规模 PLD 应用上有一定的特色。早期的 Xilinx 产品一般不涉及军品和宇航级市场，但目前已经有 Q Pro-R 等多款产品进入该类领域。

## 2、FPGA 芯片结构

目前主流的 FPGA 仍是基于查找表技术的，已经远远超出了先前版本的基本性能，并且整合了常用功能（如 RAM、时钟管理 和 DSP）的硬核（ASIC 型）模块。如图 1-1 所示（注：图 1-1 只是一个示意图，实际上每一个系列的 FPGA 都有其相应的内部结构），FPGA 芯片主 要由 6 部分完成，分别为：可编程输入输出单元、基本可编程逻辑单元、完整的时钟管理、嵌入块式 RAM、丰富的布线资源、内嵌的底层功能单元和内嵌专用硬件 模块。

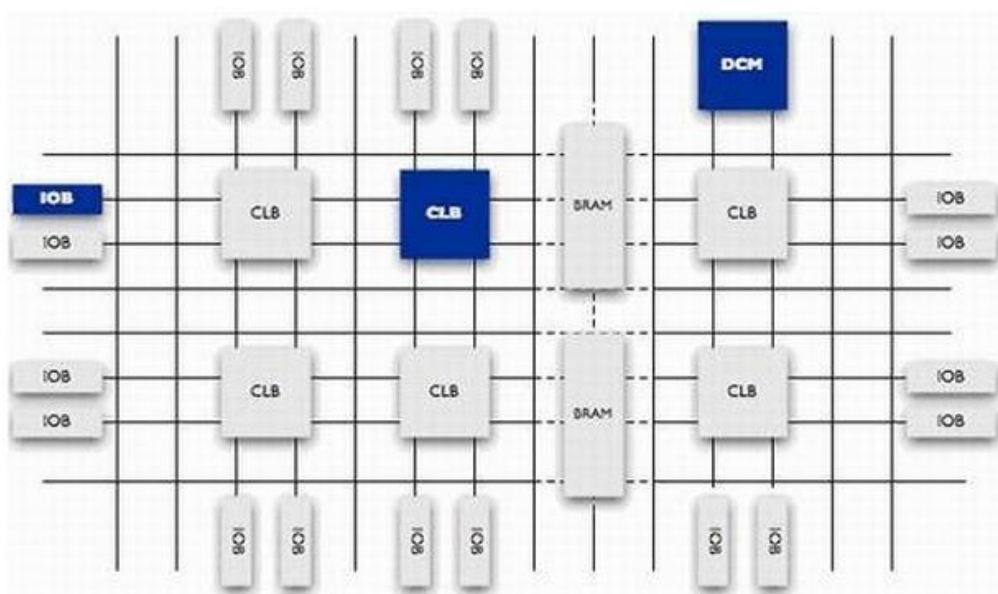


图 1-1 FPGA 芯片的内部结构

每个模块的功能如下：

#### 1. 可编程输入输出单元（IOB）

可编程输入/输出单元简称 I/O 单元，是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配要求，其示意结构如图 1-2 所示。FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性，可以调整驱动电流的大小，可以改变上、下拉电阻。目前，I/O 口的频率也越来越高，一些高端的 FPGA 通过 DDR 寄存器技术可以支持高达 2Gbps 的数据速率。

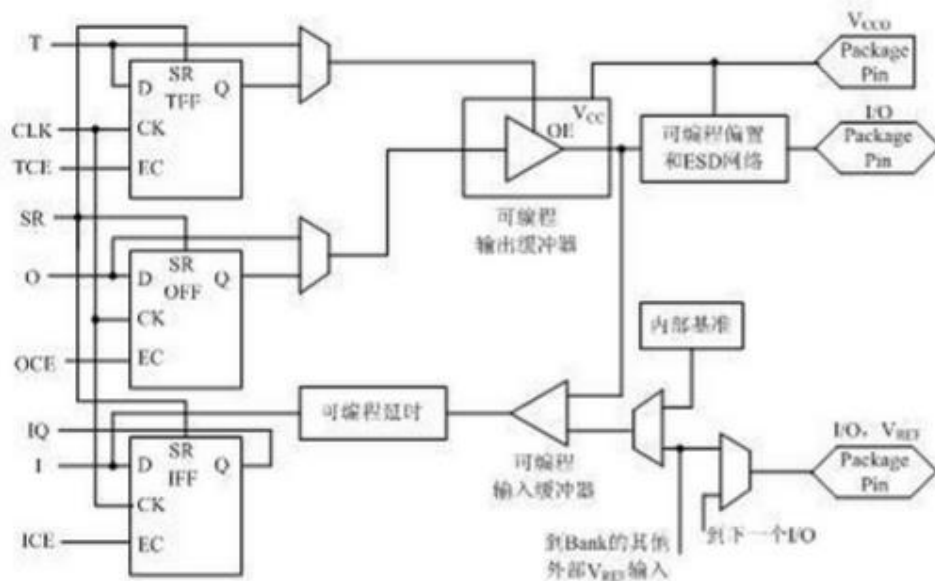


图 1-2 典型的 IOB 内部结构示意图

外部输入信号可以通过 IOB 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。当外部输入信号经过 IOB 模块的存储单元输入到 FPGA 内部时，其保持时间（Hold Time）的要求可以降低，通常默认为 0。

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 VCC0 决定，一个 bank 只能有一种 VCC0，但不同 bank 的 VCC0 可以不同。只有相同电气标准的端口才能连接在一起，VCC0 电压相同是接口标准的基本条件。

## 2. 可配置逻辑块（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，如图 1-3 所示。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。

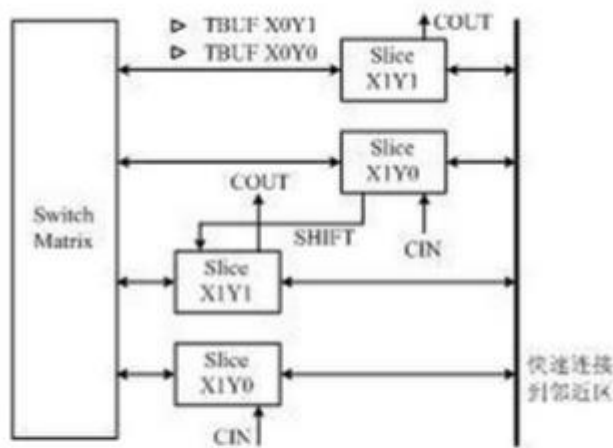


图 1-3 典型的 CLB 结构示意图

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

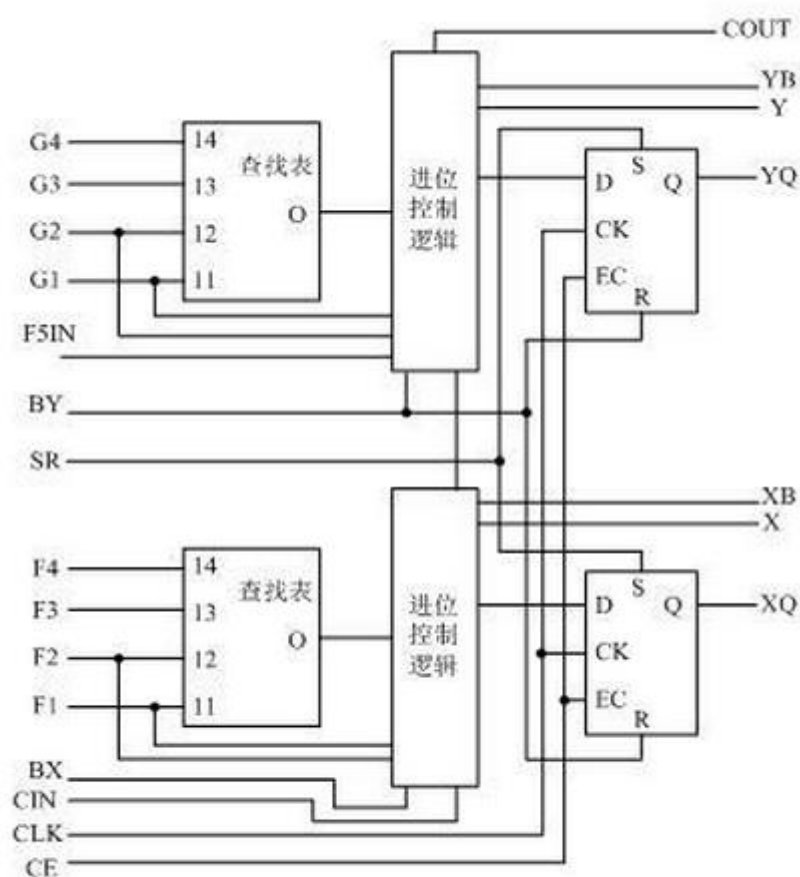


图 1-4 典型的 4 输入 Slice 结构示意图

### 3. 数字时钟管理模块（DCM）

业内大多数 FPGA 均提供数字时钟管理（Xilinx 的全部 FPGA 均具有这种特性）。Xilinx 推出最先进的 FPGA 提供数字时钟管理和相位环路锁定。相位环路锁定能够提供精确的时钟综合，且能够降低抖动，并实现过滤功能。

### 4. 嵌入式块 RAM（BRAM）

大多数 FPGA 都具有内嵌的块 RAM，这大大拓展了 FPGA 的应用范围和灵活性。块 RAM 可被配置为单端口 RAM、双端口 RAM、内容地址存储器（CAM）以及 FIFO 等常用存储结构。RAM、FIFO 是比较普及的概念，在此就不冗述。CAM 存储器在其内部的每个存储单元中都有一个比较逻辑，写入 CAM 中的数据会和内部的每一个数据进行比较，并返回与端口数据相同的所有数据的地址，因而在路由的地址交换器中有广泛的应用。除了块 RAM，还可以将 FPGA 中的 LUT 灵活地配置成 RAM、ROM 和 FIFO 等结构。在实际应用中，芯片内部块 RAM 的数量也是选择芯片的一个重要因素。

例如：单片块 RAM 的容量为 18k 比特，即位宽为 18 比特、深度为 1024，可以根据需要改变其位宽和深度，但要满足两个原则：首先，修改后的容量（位宽 深

度)不能大于 18k 比特;其次,位宽最大不能超过 36 比特。当然,可以将多片块 RAM 级联起来形成更大的 RAM,此时只受限于芯片内块 RAM 的数量,而不再受上面两条原则约束

## 5. 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。FPGA 芯片内部有着丰富的布线资源,根据工艺、长度、宽度和分布位置的不同而划分为 4 类不同的类别。第一类是全局布线资源,用于芯片内部全局时钟和全局复位/置位的布线;第二类是长线资源,用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线;第三类是短线资源,用于完成基本逻辑单元之间的逻辑互连和布线;第四类是分布式的布线资源,用于专有时钟、复位等控制信号线。

在实际中设计者不需要直接选择布线资源,布局布线器可自动地根据输入逻辑网表的拓扑结构和约束条件选择布线资源来连通各个模块单元。从本质上讲,布线资源的使用方法和设计的结果有密切、直接的关系。

## 6. 底层内嵌功能单元

内嵌功能模块主要指 DLL (Delay Locked Loop)、PLL (Phase Locked Loop)、DSP 和 CPU 等软处理核 (Soft Core)。现在越来越丰富的内嵌功能单元,使得单片 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。

DLL 和 PLL 具有类似的功能,可以完成时钟高精度、低抖动的倍频和分频,以及占空比调整和移相等功能。Xilinx 公司生产的芯片上集成了 DLL,Altera 公司的芯片集成了 PLL,Lattice 公司的新型芯片上同时集成了 PLL 和 DLL。PLL 和 DLL 可以通过 IP 核生成的工具方便地进行管理和配置。DLL 的结构如图 1-5 所示。

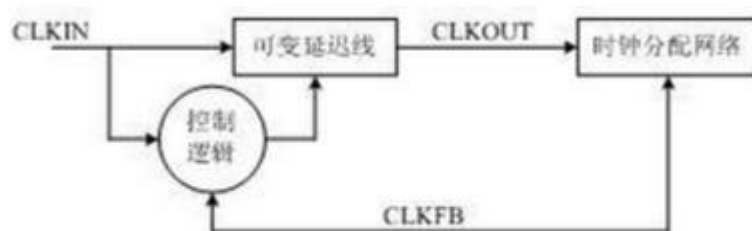


图 1-5 典型的 DLL 模块示意图

### 1. 内嵌专用硬核

内嵌专用硬核是相对底层嵌入的软核而言的,指 FPGA 处理能力强大的硬核(Hard Core),等效于 ASIC 电路。为了提高 FPGA 性能,芯片生产商在芯片内部集成了一些专用的硬核。例如:为了提高 FPGA 的乘法速度,主流的 FPGA 中都集成了

专用乘法器；为了适用通信总线与接口标准，很多高端的 FPGA 内部都集成了串并收发器（SERDES），可以达到数十 Gbps 的收发速度。Xilinx 公司的高端产品不仅集成了 Power PC 系列 CPU，还内嵌了 DSP Core 模块，其相应的系统级设计工具是 EDK 和 Platform Studio，并依此提出了片上系统（System on Chip）的概念。通过 PowerPC、Miroblaze、Picoblaze 等平台，能够开发标准的 DSP 处理器及其相关应用，达到 SOC 的开发目的。

### （1）软核

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。

### （2）固核

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。

### （3）硬核

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。

## 3、比较 CPLD 和 FPGA

### 一. 基于乘积项（Product-Term）的 PLD 结构

采用这种结构的 PLD 芯片有：Altera 的 MAX7000，MAX3000 系列（EEPROM 工艺），Xilinx 的 XC9500 系列（Flash 工艺）和 Lattice, Cypress 的大部分产品（EEPROM 工艺）

我们先看一下这种 PLD 的总体结构（以 MAX7000 为例，其他型号的结构与此都非常相似）：

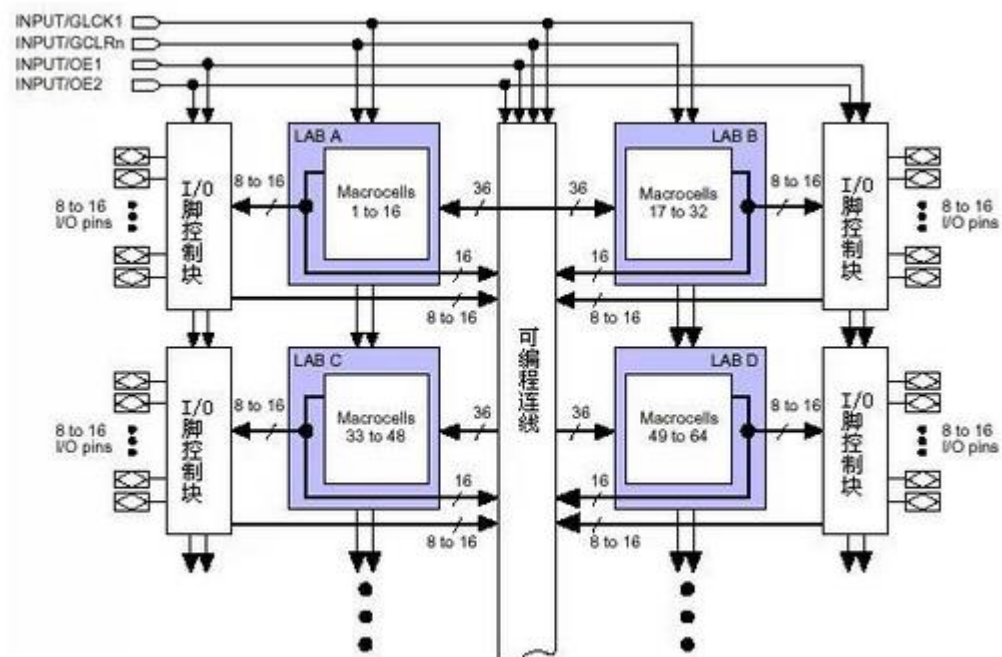


图 1 基于乘积项的 PLD 内部结构

这种 PLD 可分为三块结构：宏单元 (Macrocell)，可编程连线 (PIA) 和 I/O 控制块。宏单元是 PLD 的基本结构，由它来实现基本的逻辑功能。图 1 中蓝色部分是多个宏单元的集合（因为宏单元较多，没有一一画出）。可编程连线负责信号传递，连接所有的宏单元。I/O 控制块负责输入输出的电气特性控制，比如可以设定集电极开路输出，摆率控制，三态输出等。图 1 左上的 INPUT/GCLK1, INPUT/GCLRn, INPUT/OE1, INPUT/OE2 是全局时钟，清零和输出使能信号，这几个信号有专用连线与 PLD 中每个宏单元相连，信号到每个宏单元的延时相同并且延时最短。

宏单元的具体结构见下图：

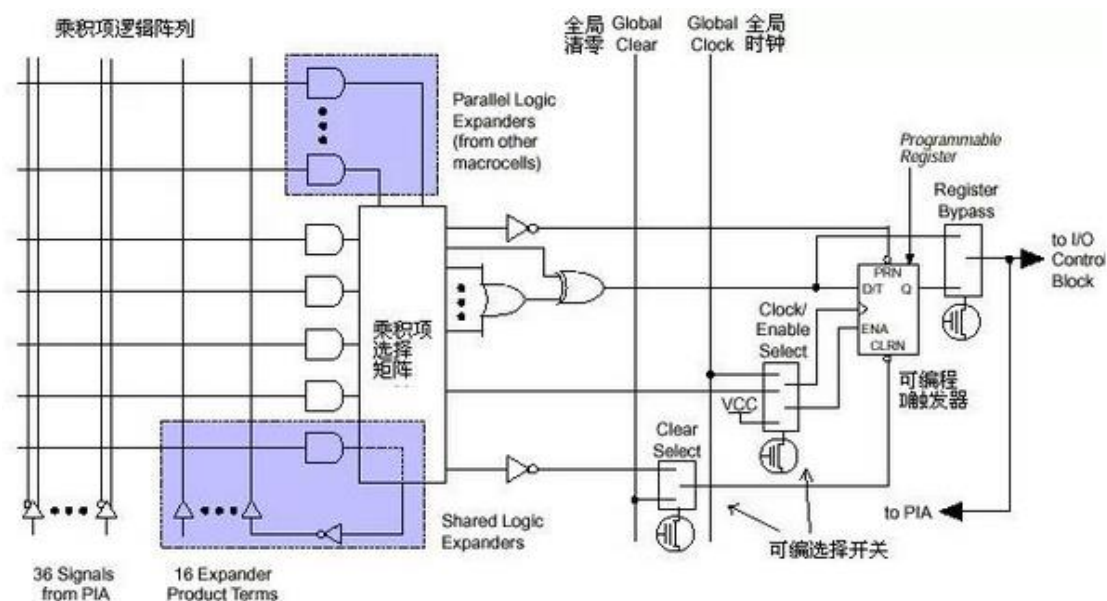


图 2 宏单元结构

左侧是乘积项阵列，实际就是一个与或阵列，每一个交叉点都是一个可编程 熔丝，如果导通就是实现“与”逻辑。后面的乘积项选择矩阵是一个“或”阵列。两者一起完成组合逻辑。图右侧是一个可编程 D 触发器，它的时钟，清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项阵列）产生的时钟和清零。如果不需要触发器，也可以将此触发器旁路，信号直接 输给 PIA 或输出到 I/O 脚。

## 二. 乘积项结构 PLD 的逻辑实现原理

下面我们以一个简单的电路为例, 具体说明 PLD 是如何利用以上结构实现逻辑的, 电路如下图:

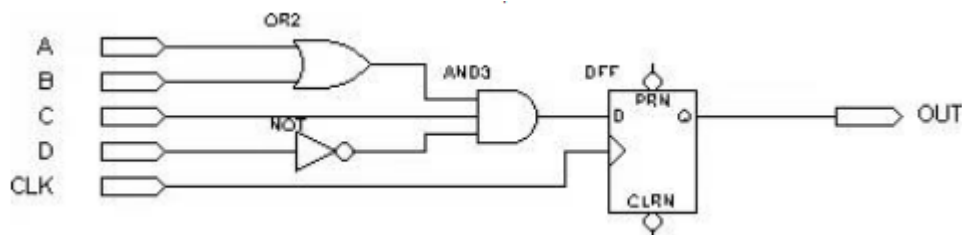


图 3

假设组合逻辑的输出 (AND3 的输出) 为  $f$ , 则  $f = (A+B)C(!D) = A*C*!D + B*C*!D$  (我们以  $!D$  表示  $D$  的“非”)

PLD 将以下面的方式来实现组合逻辑  $f$ :

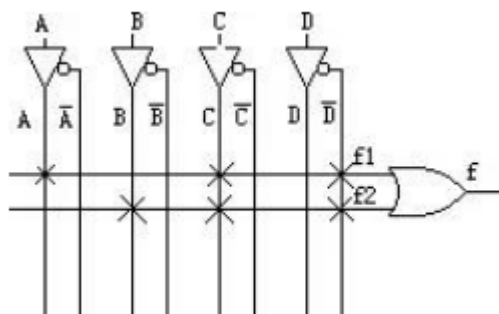


图 4

A, B, C, D 由 PLD 芯片的管脚输入后进入可编程连线阵列 (PIA)，在内部会产生 A, A 反, B, B 反, C, C 反, D, D 反 8 个输出。图中每一个叉表示相连 (可编程熔丝导通)，所以得到： $f = f1 + f2 = (A * C * !D) + (B * C * !D)$ 。这样组合逻辑就实现了。图 3 电路中 D 触发器的实现比较简单，直接利用宏单元中的可编程 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的全局时钟专用通道，直接连接到可编程触发器的时钟端。可编程触发器的输出与 I/O 脚相连，把结果输出到芯片管脚。这样 PLD 就完成了图 3 所示电路的功能。(以上这些步骤都是由软件自动完成的，不需要人为干预)

图 3 的电路是一个很简单的例子，只需要一个宏单元就可以完成。但对于一个复杂的电路，一个宏单元是不能实现的，这时就需要通过并联扩展项和共享扩展项将多个宏单元相连，宏单元的输出也可以连接到可编程连线阵列，再做为另一个宏单元的输入。这样 PLD 就可以实现更复杂逻辑。

这种基于乘积项的 PLD 基本都是由 EEPROM 和 Flash 工艺制造的，一上电就可以工作，无需其他芯片配合。

## FPGA

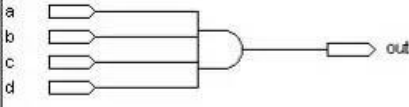
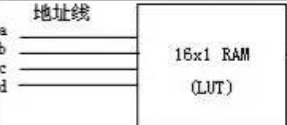
FPGA 作为 ASIC 领域中的一种半定制电路而出现的，即解决了定制电路的不足，又克服了原有可编程器件门电路有限的缺点。由于 FPGA 需要被反复烧写，它实现组合逻辑的基本结构不可能像 ASIC 那样通过固定的与非门来完成，而只能采用一种易于反复配置的结构。查找表可以很好地满足这一要求。通过烧写文件去配置查找表的内容，从而在相同的电路情况下实现了不同的逻辑功能。

### 查找表的原理与结构

查找表(Look-Up-Table)简称为 LUT，LUT 本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成是一个有 4 位地址线的 RAM。当用户通过原理图或 HDL 语言描述了一个逻辑电路以后，PLD/FPGA 开发软件会自动计算逻辑电路的所有可能结果，并把真值表(即结果)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。

下面给出一个四输入与非门电路的例子来说明 LUT 实现逻辑功能的原理。

表给出一个使用 LUT 实现四输入与门电路的真值表。

实际逻辑电路			LUT的实现方式	
				
a,b,c,d 输入	逻辑输出	地址	RAM中存储的内容	
0000	0	0000	0	
0001	0	0001	0	
....	0	...	0	
1111	1	1111	1	

从中可以看到，LUT 具有和逻辑电路相同的功能。实际上，LUT 具有更快的执行速度和更大的规模。

查找表结构的 FPGA 逻辑实现原理

我们还是以这个电路的为例：

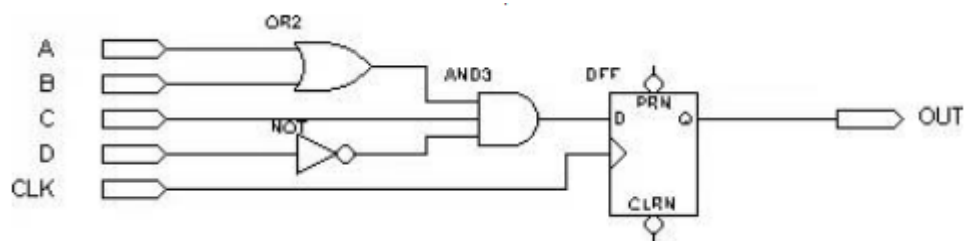


图 四输入与门电路图

A，B，C，D 由 FPGA 芯片的管脚输入后进入可编程连线，然后作为地址线连到 LUT，LUT 中已经事先写入了所有可能的逻辑结果，通过地址查找到相应的数据然后输出，这样组合逻辑就实现了。该电路中 D 触发器是直接利用 LUT 后面 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的时钟专用通道，直接连接到触发器的时钟端。触发器的输出与 I/O 脚相连，把结果输出到芯片管脚。这样 PLD 就完成了图所示电路的功能。（以上这些步骤都是由软件自动完成的，不需要人为干预）

这个电路是一个很简单的例子，只需要一个 LUT 加上一个触发器就可以完成。对于一个 LUT 无法完成的的电路，就需要通过进位逻辑将多个单元相连，这样 FPGA 就可以实现复杂的逻辑。

因为基于 LUT 的 FPGA 具有很高的集成度,其器件密度从数万门到数千万门不等,可以完成极其复杂的时序与逻辑组合逻辑电路功能,所以适用于高速、高密度的高端数字逻辑电路设计领域。其组成部分主要有可编程输入/输出单元、基本可编程逻辑单元、内嵌 SRAM、丰富的布线资源、底层嵌入功能单元、内嵌专用单元等,主要设计和生产厂家有 Xilinx、Altera、Lattice、Actel、Atmel 和 QuickLogic 等公司,其中最大的是 Xilinx、Altera、Lattice 三家。

4、比较 Xilinx 和 Altera

要比较 Xilinx 和 Altera 的 FPGA,就要清楚两个大厂 FPGA 的结构,由于各自设计的不同,两家的 FPGA 结构各不相同,参数也各不相同,但可以统一到 LUT (Look-Up-Table) 查找表上。

下图就是 A 家的 Cyclone IV 系列片子的参数:

Table 1-1. Resources for the Cyclone IV E Device Family

Resources	EP4CE6	EP4CE10	EP4CE15	EP4CE22	EP4CE30	EP4CE40	EP4CE55	EP4CE75	EP4CE115
Logic elements (LEs)	6,272	10,320	15,408	22,320	28,848	39,600	55,856	75,408	114,480
Embedded memory (Kbits)	270	414	504	594	594	1,134	2,340	2,745	3,888
Embedded 18 × 18 multipliers	15	23	56	66	66	116	154	200	266
General-purpose PLLs	2	2	4	4	4	4	4	4	4
Global Clock Networks	10	10	20	20	20	20	20	20	20
User I/O Banks	8	8	8	8	8	8	8	8	8
Maximum user I/O <sup>(1)</sup>	179	179	343	153	532	532	374	426	528

可以看到, A 家的片子,用的是 LE 这个术语。

而下图是 X 家的 Spartan-6 片子资料:

Spartan-6 FPGA Feature Summary

Table 1: Spartan-6 FPGA Feature Summary by Device

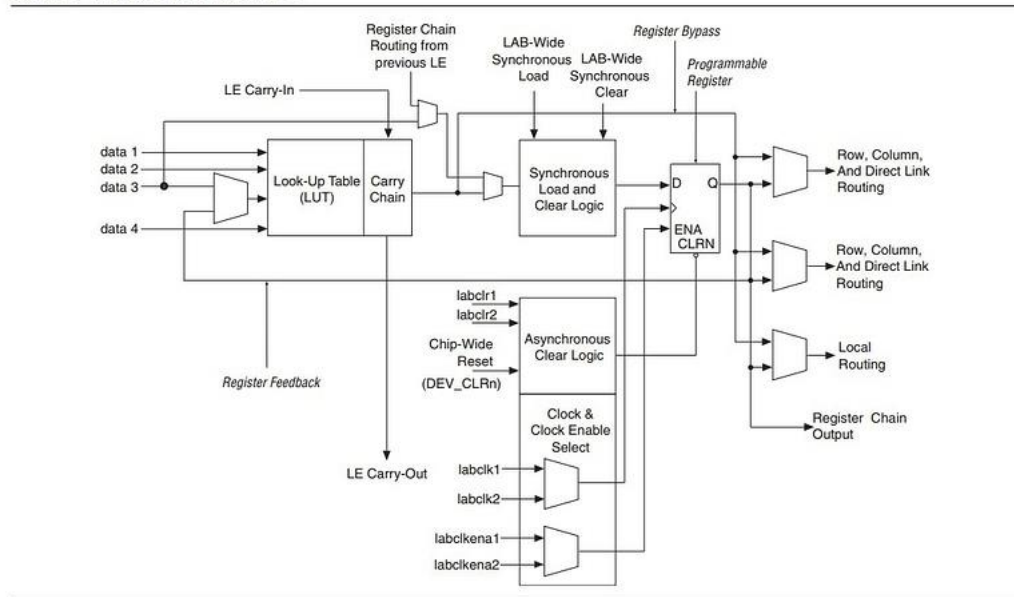
Device	Logic Cells <sup>(1)</sup>	Configurable Logic Blocks (CLBs)			DSP48A1 Slices <sup>(3)</sup>	Block RAM Blocks		CMTs <sup>(5)</sup>	Memory Controller Blocks (Max) <sup>(6)</sup>	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices <sup>(2)</sup>	Flip-Flops	Max Distributed RAM (Kb)		18 Kb <sup>(4)</sup>	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232
XC6SLX25	24,051	3,758	30,064	229	38	52	936	2	2	0	0	4	266
XC6SLX45	43,661	6,822	54,576	401	58	116	2,088	4	2	0	0	4	358
XC6SLX75	74,637	11,662	93,296	692	132	172	3,096	6	4	0	0	6	408
XC6SLX100	101,261	15,822	126,576	976	180	268	4,824	6	4	0	0	6	480
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,824	6	4	0	0	6	576
XC6SLX25T	24,051	3,758	30,064	229	38	52	936	2	2	1	2	4	250
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,088	4	2	1	4	4	296
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,096	6	4	1	8	6	348
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,824	6	4	1	8	6	498
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,824	6	4	1	8	6	540

X 家用的是 CLB 这个术语作为基本单元。

再看看两家的基本单元有何不同：

A 家的 LE 如下图:

**Figure 2–1. Cyclone IV Device LEs**



就是一个 4 输入 LUT+FF 构成

而 X 家的 CLB 如下：

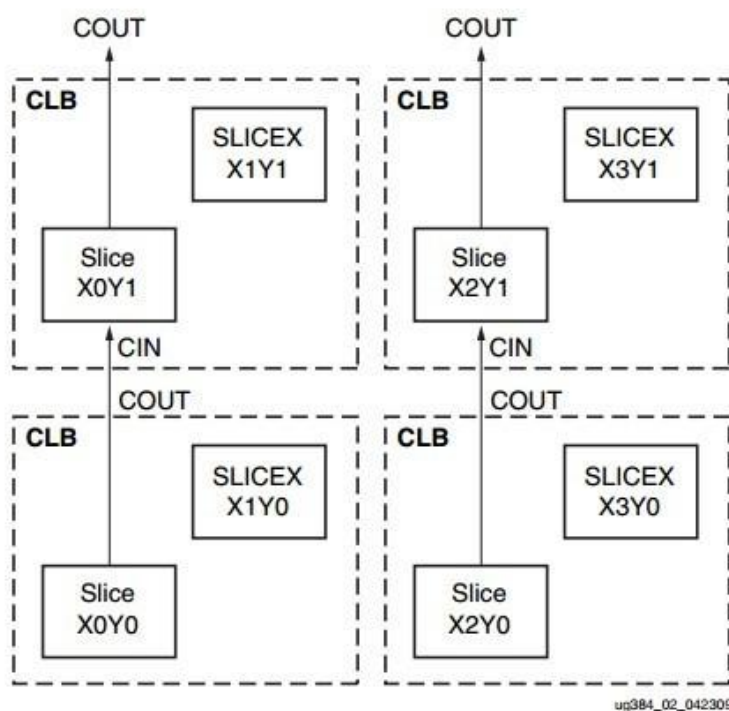


Figure 2: Row and Column Relationship between CLBs and Slices

一个 CLB 由 2 个 SLICE 构成,一个 SLICE 含有 4 个 6 输入 LUT,所以  $LUT=8*CLB$ 。

Table 2: Logic Resources in One CLB

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains <sup>(2)</sup>	Distributed RAM <sup>(1)</sup>	Shift Registers <sup>(1)</sup>
2	8	16	1	256 bits	128 bits

这样的话,可以较比一下。EP4CE6 基本就和 XC6SLX9 一个级别。。。当然 A 家的片子是 4 输入 LUT 远比不上 X 家的 6 输入 LUT。而 X 家的 S-6 片子,一个 Slice 内部有 4 个 lut, 8 个 FF。简而言之,一个 Slice=四个 LE。要注意的是 A 家 C5 以下的片子是 4 输入 LUT 而 X 家的是 6 输入 LUT, 差别也较大。如果不考虑 FF, 那么一个 X 家的 slice=4 个 A 家的 LE。例如 XC6SLX16 含有 2278 个 slices=EP4CE10 (9000LE) 的样子。当然, S-6 的 FF 多一倍, 达到了 18224 个。

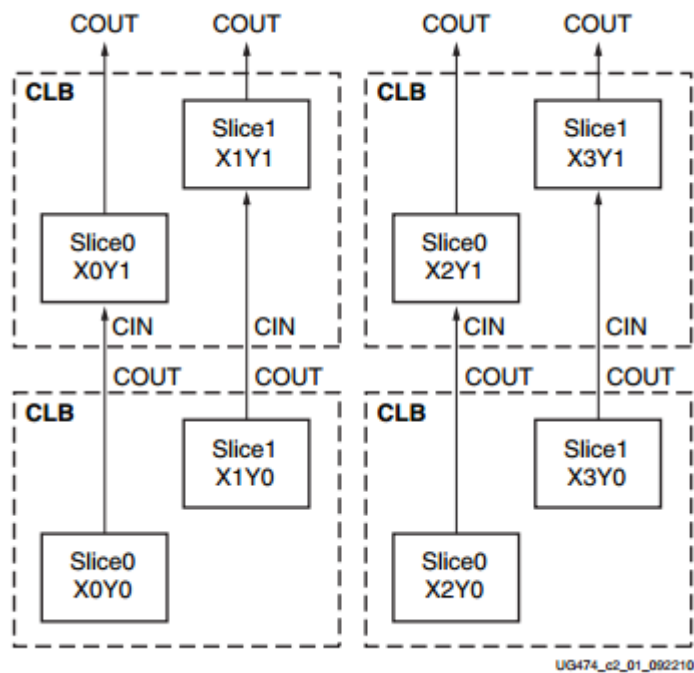
在 Virtex-5 中(我们的设计大部分是 Virtex, V5V6V7), 一个 Slice 包含了 4 个 LUT 和 4 个 FF。所以单纯从逻辑资源来看, S-6 一个 Slice 比 V-5 的 Slice 强。当然 V5 的 GTPGTX 等等还有 IO 数量是 S-6 赶不上的。当然, A 家的 Cyclone V 系列的片子, 内部和前几代完全不同, 采用了从高端的 Stratix 系列下放的技术。

## 5、分布式 RAM 和 Block ram

以下分析基于 xilinx 7 系列

CLB 是 xilinx 基本逻辑单元，每个 CLB 包含两个 slices，每个 slices 由 4 个 (A,B,C,D) 6 输入 LUT 和 8 个寄存器组成。

同一 CLB 中的两片 slices 没有直接的线路连接，分属于两个不同的列。每列拥有独立的快速进位链资源。



slice 分为两种类型 SLICEL, SLICEM. SLICEL 可用于产生逻辑, 算术, ROM. SLICEM 除以上作用外还可配置成分布式 RAM 或 32 位的移位寄存器。每个 CLB 可包含两个 SLICEL 或者一个 SLICEL 与一个 SLICEM.

7 系列的 LUT 包含 6 个输入 A1 -A6 , 两个输出 O5 , O6 .

可配置成 6 输入查找表, O6 此时作为输出。或者两个 5 输入的查找表, A1-A5 作为输入 A6 拉高, O5, O6 作为输出。

一个 LUT 包含 6 个输入, 逻辑容量为  $2^6\text{bit}$ , 为实现 7 输入逻辑需要  $2^7$  容量, 对于更多输入也一样。每个 SLICES 有 4 个 LUT, 256bit 容量能够实现最多 8bit 输入的逻辑。为了实现此功能, 每个 SLICES 还包括 3 个 MUX(多路选择器)

F7AMUX 用于产生 7 输入的逻辑功能, 用于连接 A,B 两个 LUT

F7BMUX 用于产生 7 输入的逻辑功能, 用于连接 C,D 两个 LUT

F8MUX 用于产生 8 输入的逻辑功能, 用于连接 4 个 LUT

对于大于 8 输入的逻辑需要使用多个 SLICES，会增加逻辑实现的延时。

一个 SLICES 中的 4 个寄存器可以连接 LUT 或者 MUX 的输出，或者被直接旁路不连接任何逻辑资源。寄存器的置位/复位端为高电平有效。只有 CLK 端能被设置为两个极性，其他输入若要改变电平需要插入逻辑资源。例如低电平复位需要额外的逻辑资源将 rst 端输入取反。但设为上升/下降沿触发寄存器不会带来额外消耗。

分布式 RAM

SLICEM 可以配置成分布式 RAM，一个 SLICEM 可以配置成以下容量的 RAM

Table 2-3. Distributed RAM Configuration

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

多 bit 的情况需要增加相应倍数的 LUT 进行并联。

分布式 RAM 和 BLOCK RAM 的选择遵循以下方法：

1. 小于或等于 64bit 容量的的都用分布式实现
2. 深度在 64~128 之间的，若无额外的 block 可用分布式 RAM。 要求异步读取就使用分布式 RAM。数据宽度大于 16 时用 block ram.
3. 分布式 RAM 有比 block ram 更好的时序性能。分布式 RAM 在逻辑资源 CLB 中。而 BLOCK RAM 则在专门的存储器列中，会产生较大的布线延迟，布局也受制约。

移位寄存器（SLICEM）

SLICEM 中的 LUT 能在不使用触发器的情况下设置成 32bit 的移位寄存器，4 个 LUT 可级联成 128bit 的移位寄存器。并且能够进行 SLICEM 间的级联形成更大规模的移位寄存器。

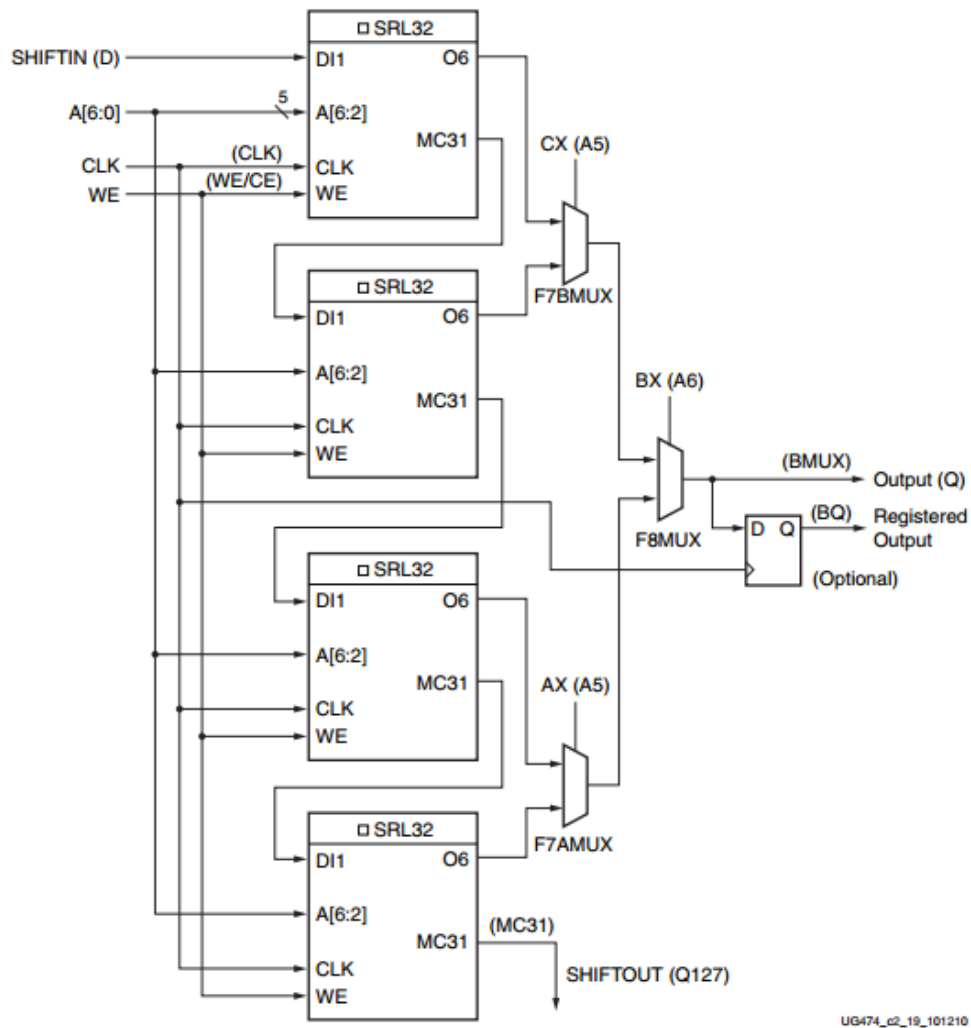


Figure 2-20: 128-Bit Shift Register Configuration

MUX

一个 LUT 可配置成 4:1MUX.

两个 LUT 可配置成最多 8:1 MUX

四个 LUT 可配置成 16 个 MUX

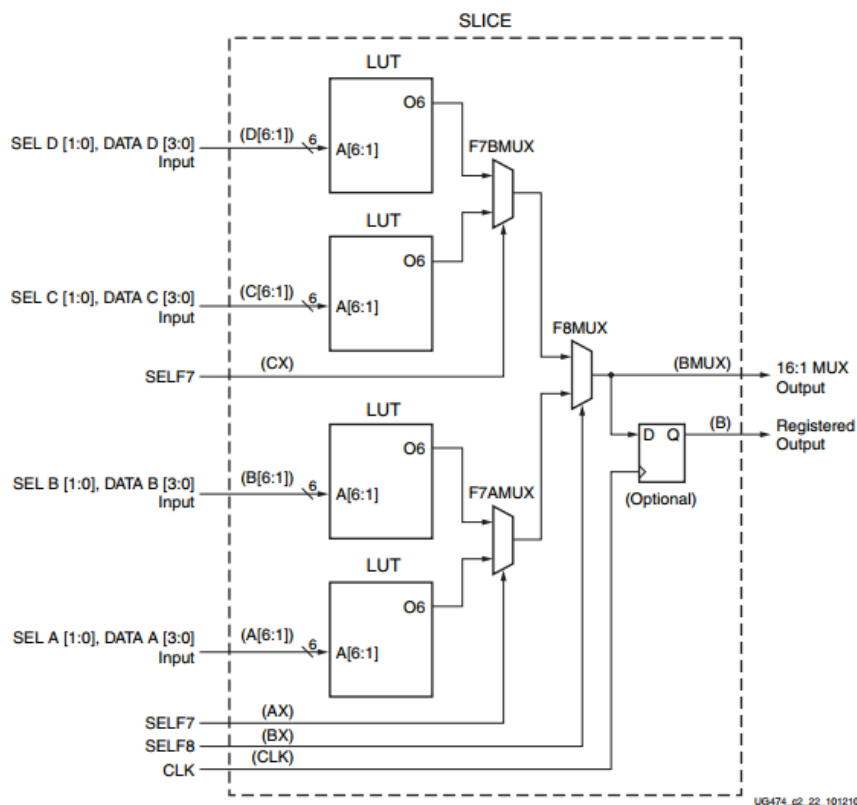


Figure 2-23: 16:1 Multiplexer in a Slice

同样可以通过连接多个 SLICES 达成更大规模设计，但是由于 SLICE 没有直接连线，需要使用布线资源，会增加较大延迟。

## 进位链

每个 SLICE 有 4bit 的进位链。每 bit 都由一个进位 MUX(MUXCY)和一个异或门组成，可在实现加法/减法器时生成进位逻辑。该 MUXCY 与 XOR 也可用于产生一般逻辑。

## 6、FPGA 设计方法概论

FPGA 是可编程芯片，因此 FPGA 的设计方法包括硬件设计和软件设计两部分。硬件包括 FPGA 芯片电路、存储器、输入输出接口电路以及其他设备，软件即是相应的 HDL 程序以及最新才流行的嵌入式 C 程序。硬件设计是基础，但其方法比较固定，本书将在第 4 节对其进行详细介绍，本节主要介绍软件的设计方法。

目前微电子技术已经发展到 SOC 阶段，即集成系统 (Integrated System) 阶段，相对于集成电路 (IC) 的设计思想有着革命性的变化。SOC 是一个复杂的系统，它将一个完整产品的功能集成在一个芯片上，包括核心处理器、存储单元、硬件加速单元以及众多的外部设备接口等，具有设计周期长、实现成本高等特点，因此其设计方法必然是自顶向下的从系统级到功能模块的软、硬件协同设计，达到软、硬件的无缝结合。

这么庞大的工作量显然超出了单个工程师的能力，因此需要按照层次化、结构化的设计方法来实施。首先由总设计师将整个软件开发任务划分为若干个可操作的模块，并对其接口和资源进行评估，编制出相应的行为或结构模型，再将其分配给下一层的设计师。这就允许多个设计者同时设计一个硬件系统中的不同模块，并为自己所设计的模块负责；然后由上层设计师对下层模块进行功能验证。

自顶向下的设计流程从系统级设计开始，划分为若干个二级单元，然后再把各个二级单元划分为下一层次的基本单元，一直下去，直到能够使用基本模块或者 IP 核直接实现为止，如图 1-6 所示。流行的 FPGA 开发工具都提供了层次化管理，可以有效地梳理错综复杂的层次，能够方便地查看某一层次模块的源代码以修改错误。



图 1-6 自顶向下的 FPGA 设计开发流程

在工程实践中，还存在软件编译时长的问题。由于大型设计包含多个复杂的功能模块，其时序收敛与仿真验证复杂度很高，为了满足时序指标的要求，往往需要反复修改源文件，再对所修改的新版本进行重新编译，直到满足要求为止。这里面存在两个问题：首先，软件编译一次需要长达数小时甚至数周的时间，这是开发所不能容忍的；其次，重新编译和布局布线后结果差异很大，会将已满足时序的电路破坏。因此必须提出一种有效提高设计性能，继承已有结果，便于团队化设计的软件工具。FPGA 厂商意识到这类需求，由此开发出了相应的逻辑锁定和增量设计的软件工具。例如，Xilinx 公司的解决方案就是 PlanAhead。

PlanAhead 允许高层设计者为不同的模块划分相应 FPGA 芯片区域，并允许底层设计者在所给定的区域内独立地进行设计、实现和优化，等各个模块都正确后，再进行设计整合。如果在设计整合中出现错误，单独修改即可，不会影响到其它模块。PlanAhead 将结构化设计方法、团队化合作设计方法以及重用继承设计方法三者完美地结合在一起，有效地提高了设计效率，缩短了设计周期。

不过从其描述可以看出，新型的设计方法对系统顶层设计师有很高的要求。在设计初期，他们不仅要评估每个子模块所消耗的资源，还需要给出相应的时序关系；在设计后期，需要根据底层模块的实现情况完成相应的修订。

## 典型 FPGA 开发流程

FPGA 的设计流程就是利用 EDA 开发软件和编程工具对 FPGA 芯片进行开发的过程。FPGA 的开发流程一般如图 1-7 所示，包括电路设计、设计输入、功能仿真、综合优化、综合后仿真、实现、布线后仿真、板级仿真以及芯片编程与调试等主要步骤。

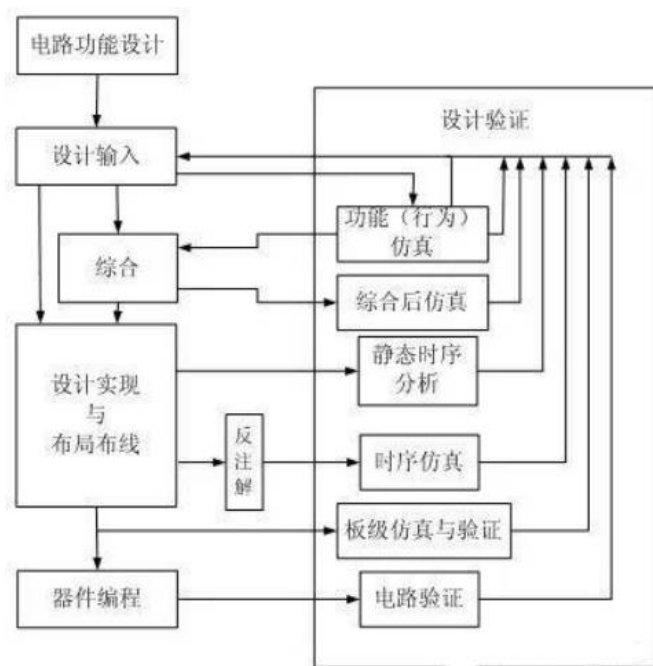


图 1-7 FPGA 开发的一般流程

### 1. 电路设计

在系统设计之前，首先要进行的是方案论证、系统设计和 FPGA 芯片选择等准备工作。系统工程师根据任务要求，如系统的指标和复杂度，对工作速度和芯片本身的各种资源、成本等方面进行权衡，选择合理的设计方案和合适的器件类型。一般都采用自顶向下的设计方法，把系统分成若干个基本单元，然后再把每个基本单元划分为下一层次的基本单元，一直这样做下去，直到可以直接使用 EDA 元件库为止。

### 2. 设计输入

设计输入是将所设计的系统或电路以开发软件要求的某种形式表示出来，并输入给 EDA 工具的过程。常用的方法有硬件描述语言（HDL）和原理图输入方法等。原理图输入方式是一种最直接的描述方式，在可编程芯片发展的早期应用比较广泛，它将所需的器件从元件库中调出来，画出原理图。这种方法虽然直观并易于仿真，但效率很低，且不易维护，不利于模块构造和重用。更主要的缺点是可移植性差，当芯片升级后，所有的原理图都需要作一定的改动。目前，在实际开发中应用最广的就是 HDL 语言输入法，利用文本描述设计，可以分为普通 HDL 和行为 HDL。普通 HDL 有 ABEL、CUR 等，支持逻辑方程、真值表和状态机等表达方式，主要用于简单的小型设计。而在中大型工程中，主要使用行为 HDL，其主流语言是 Verilog HDL 和 VHDL。这两种语言都是美国电气与电子工程师协会（IEEE）

的标准，其共同的突出特点有：语言与芯片工艺无关，利于自顶向下设计，便于模块的划分与移植，可移植性好，具有很强的逻辑描述和仿真功能，而且输入效率很高。

### 3. 功能仿真

功能仿真，也称为前仿真，是在编译之前对用户所设计的电路进行逻辑功能验证，此时的仿真没有延迟信息，仅对初步的功能进行检测。仿真前，要先利用波形编辑器和 HDL 等建立波形文件和测试向量（即将所关心的输入信号组合成序列），仿真结果将会生成报告文件和输出信号波形，从中便可以观察各个节点信号的变化。如果发现错误，则返回设计修改逻辑设计。常用的工具有 Model Tech 公司的 ModelSim、Synopsys 公司的 VCS 和 Cadence 公司的 NC-Verilog 以及 NC-VHDL 等软件。

### 4. 综合优化

所谓综合就是将较高级抽象层次的描述转化成较低层次的描述。综合优化根据目标与要求优化所生成的逻辑连接，使层次设计平面化，供 FPGA 布局布线软件进行实现。就目前的层次来看，综合优化（Synthesis）是指将设计输入编译成由与门、或门、非门、RAM、触发器等基本逻辑单元组成的逻辑连接网表，而并非真实的门级电路。真实具体的门级电路需要利用 FPGA 制造商的布局布线功能，根据综合后生成的标准门级结构网表来产生。为了能转换成标准的门级结构网表，HDL 程序的编写必须符合特定综合器所要求的风格。由于门级结构、RTL 级的 HDL 程序的综合是很成熟的技术，所有的综合器都可以支持到这一级别的综合。常用的综合工具有 Synplicity 公司的 Synplify/Synplify Pro 软件以及各个 FPGA 厂家自己推出的综合开发工具。

### 5. 综合后仿真

综合后仿真检查综合结果是否和原设计一致。在仿真时，把综合生成的标准延时文件反标注到综合仿真模型中去，可估计门延时带来的影响。但这一步骤不能估计线延时，因此和布线后的实际情况还有一定的差距，并不十分准确。目前的综合工具较为成熟，对于一般的设计可以省略这一步，但如果在布局布线后发现电路结构和设计意图不符，则需要回溯到综合后仿真来确认问题之所在。在功能仿真中介绍的软件工具一般都支持综合后仿真。

### 6. 实现与布局布线

实现是将综合生成的逻辑网表配置到具体的 FPGA 芯片上，布局布线是其中最重要的过程。布局将逻辑网表中的硬件原语和底层单元合理地配置到芯片内部的固有硬件结构上，并且往往需要在速度最优和面积最优之间作出选择。布线根据布局的拓扑结构，利用芯片内部的各种连线资源，合理正确地连接各个元件。目前，FPGA 的结构非常复杂，特别是在有时序约束条件时，需要利用时序驱动的引擎进行布局布线。布线结束后，软件工具会自动生成报告，提供有关设计中各部分资源的使用情况。由于只有 FPGA 芯片生产商对芯片结构最为了解，所以布局布线必须选择芯片开发商提供的工具。

### 7. 实现与布局布线

时序仿真，也称为后仿真，是指将布局布线的延时信息反标注到设计网表中来检

测有无时序违规（即不满足时序约束条件或器件固有的时序规则，如建立时间、保持时间等）现象。时序仿真包含的延迟信息最全，也最精确，能较好地反映芯片的实际工作情况。由于不同芯片的内部延时不一样，不同的布局布线方案也给延时带来不同的影响。因此在布局布线后，通过对系统和各个模块进行时序仿真，分析其时序关系，估计系统性能，以及检查和消除竞争冒险是非常有必要的。在功能仿真中介绍的软件工具一般都支持综合后仿真。

## 8. 板级仿真与验证

板级仿真主要应用于高速电路设计中，对高速系统的信号完整性、电磁干扰等特征进行分析，一般都以第三方工具进行仿真和验证。

## 9. 芯片编程与调试

设计的最后一步就是芯片编程与调试。芯片编程是指产生使用的数据文件（位数据流文件，Bitstream Generation），然后将编程数据下载到 FPGA 芯片中。其中，芯片编程需要满足一定的条件，如编程电压、编程时序和编程算法等方面。逻辑分析仪（Logic Analyzer, LA）是 FPGA 设计的主要调试工具，但需要引出大量的测试管脚，且 LA 价格昂贵。目前，主流的 FPGA 芯片生产商都提供了内嵌的在线逻辑分析仪（如 Xilinx ISE 中的 ChipScope、Altera QuartusII 中的 SignalTapII 以及 SignalProb）来解决上述矛盾，它们只需要占用芯片少量的逻辑资源，具有很高的实用价值。

## 基于 FPGA 的 SOC 设计方法

基于 FPGA 的 SOC 设计理念将 FPGA 可编程的优点带到了 SOC 领域，其系统由嵌入式处理器内核、DSP 单元、大容量处理器、吉比特收发器、混合逻辑、IP 以及原有的设计部分组成。相应的 FPGA 规模大都在百万门以上，适合于许多领域，如电信、计算机等行业。

系统设计方法是 SOC 常用的方法学，其优势在于，可进行反复修改并对系统架构实现进行验证，??? 包括 SOC 集成硬件和软件组件之间的接口。不过，目前仍存在很多问题，最大的问题就是没有通用的系统描述语言和系统级综合工具。随着 FPGA 平台的融入，将 SOC 逐步地推向了实用。SOC 平台的核心部分是内嵌的处理内核，其硬件是固定的，软件则是可编程的；外围电路则由 FPGA 的逻辑资源组成，大都以 IP 的形式提供，例如存储器接口、USB 接口以及以太网 MAC 层接口等，用户根据自己需要在内核总线上添加，并能自己订制相应的接口 IP 和外围设备。

## 基于 FPGA 的典型 SOC 开发流程为：

### 1. 芯片内的考虑

从设计生成开始，设计人员需要从硬件/软件协同验证的思路入手，以找出只能在系统集成阶段才会被发现的软、硬件缺陷。然后选择合适的芯片以及开发工具，在综合过程得到优化，随后进行精确的实现，以满足实际需求。由于设计规模越来越大，工作频率也到了数百兆赫兹，布局布线的延迟将变得非常重要。为了确保满足时序，需要在布局布线后进行静态时序分析，对设计进行验证。

## 2. 板级验证

在芯片设计完毕后，需要再进行板级验证，以便在印刷电路板（PCB）上保证与最初设计功能一致。因此，PCB 布局以及信号完整性测试应被纳入设计流程。由于芯片内设计所做的任何改变都将反映在下游的设计流程中，各个过程之间的数据接口和管理也必须是无误的。预计 SOC 系统以及所必须的额外过程将使数据的大小成指数增长，因此，管理各种数据集本身是急剧挑战性的任务

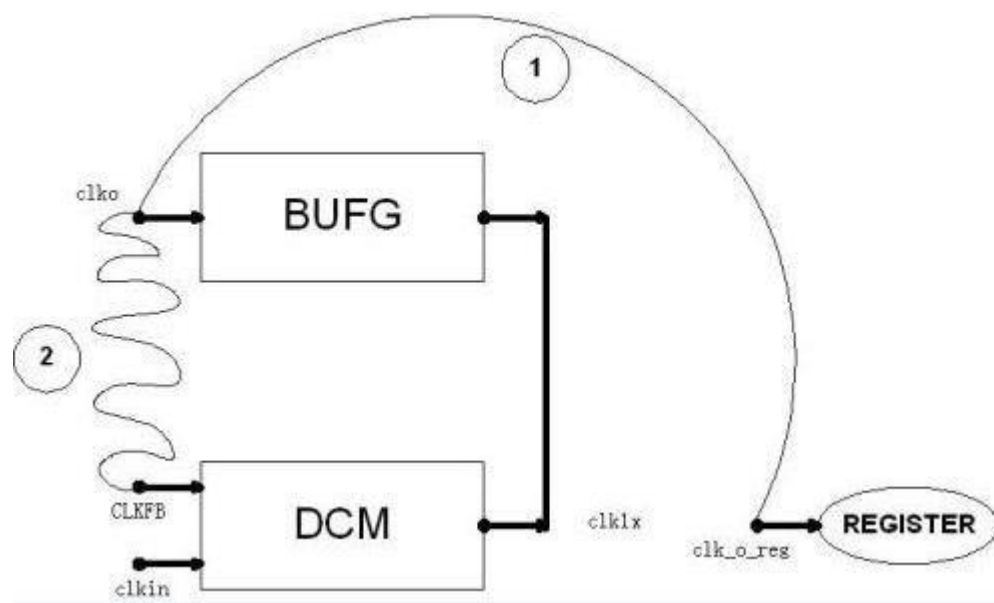
## 7、DCM 时钟管理单元

看 Xilinx 的 Datasheet 会注意到 Xilinx 的 FPGA 没有 PLL，其实 DCM 就是时钟管理单元。

### 1、DCM 概述

DCM 内部是 DLL (Delay Lock Loop 结构, 对时钟偏移量的调节是通过长的延时线形成的。DCM 的参数里有一个 PHASESHIFT (相移), 可以从 0 变到 255。所以我们可以假设内部结构里从输入引脚 clkin 到输出引脚 clk\_1x 之间应该有 256 根延时线 (实际上, 由于对不同频率的时钟都可以从 0 变到 255, 延时线的真正数目应该比这个大得多)。DCM 总会把输入时钟 clkin 和反馈时钟 CLKFB 相比较, 如果它们的延时差不等于所设置的 PHASESHIFT, DCM 就会改变在 clkin 和 clk\_1x 之间的延时线数目, 直到相等为止, 输出和输入形成闭环, 动态调整到设定值再退出。这个从不等到相等所花的时间, 就是输出时钟锁定的时间, 相等以后, lock\_flag 标识才会升高。

当 DCM 发现 clkin 和 clkfb 位相差不等于 PHASESHIFT 的时候, 就去调节 clk\_1x 和 clkin 之间延时, 所以如果 clk\_1x 和 clkfb 不相关的话, 那就永远也不能锁定了。



图一、DCM 和 BUFG 配合使用示意图

## 2、如何使用 DCM

DCM 一般和 BUFG 配合使用, 要加上 BUFG, 应该是为了增强时钟的驱动能力。DCM 的一般使用方法是, 将其输出 `clk_1x` 接在 BUFG 的输入引脚上, BUFG 的输出引脚反馈回来接在 DCM 的反馈时钟脚 `CLKFB` 上。另外, 在 FPGA 里, 只有 BUFG 的输出引脚接在时钟网络上, 所以一般来说你可以不使用 DCM, 但你一定会使用 BUFG。有些兄弟总喜欢直接将外部输入的时钟驱动内部的寄存器, 其实这个时候虽然你没有明显地例化 BUFG, 但工具会自动给你加上的。

## 3、使用 DCM 可以消除时钟 skew

使用 DCM 可以消除时钟 skew。这个东西一直是我以前没有想清楚的, 时钟从 DCM 输出开始走线到寄存器, 这段 skew 的时间总是存在的, 为什么用 DCM 就可以消除呢? 直到有一天忽然豁然开朗, 才明白其原委。对高手来说, 也许是极为 easy 的事情, 但也许有些朋友并不一定了解, 所以写出来和大家共享。

为说明方便起见, 我们将 BUFG 的输出引脚叫做 `clk_o`, 从 `clk_o` 走全局时钟布线到寄存器时叫做 `clk_o_reg`, 从 `clk_o` 走线到 DCM 的反馈引脚 `CLKFB` 上时叫 `clkfb`, 如图所示。实际上 `clk_o`, `clk_o_reg`, `clkfb` 全部是用导线连在一起的。

所谓时钟 skew, 指的就是 `clk_o` 到 `clk_o_reg` 之间的延时。如果打开 `FPGA_Editor` 看底层的结构, 就可以发现虽然 DCM 和 BUFG 离得很近, 但是从 `clk_o` 到 `clkfb` 却绕了很长一段才走回来, 从而导致从 `clk_o` 到 `clk_o_reg` 和 `clkfb` 的延时大致相等。

总之就是 `clk_o_reg` 和 `clkfb` 的相位应该相等。所以当 DCM 调节 `clkin` 和 `clkfb` 的相位相等时, 实际上就调节了 `clkin` 和 `clk_o_reg` 相等。而至于 `clk_1x` 和 `clk_o` 的相位必然是超前于 `clkin`, `clkfb`, `clk_o_reg` 的, 而 `clk_1x` 和 `clk_o` 之间的延时就很明显, 就是经过那个 BUFG 的延迟时间。

## 4、对时钟 skew 的进一步讨论

最后, 说一说时钟 skew 的概念。时钟 skew 实际上指的是时钟驱动不同的寄存器时, 由于寄存器之间可能会隔得比较远, 所以时钟到达不同的寄存器的时间可能会不一样, 这个时间差称为时钟 skew。这种时钟 skew 可以通过时钟树来解决, 也就是使时钟布线形成一种树状结构, 使得时钟到每一个寄存器的距离是一样的。很多 FPGA 芯片里就布了这样的时钟树结构。也就是说, 在这种芯片里, 时钟 skew 基本上是不存在的。

说到这里, 似乎有了一个矛盾, 既然时钟 skew 的问题用时钟树就解决了, 那么为什么还需要 DCM+BUFG 来解决这个问题? 另外, 既然时钟 skew 指的是时钟驱动不同寄存器之间的延时, 那么上面所说的 `clk_o` 到 `clk_o_reg` 岂非不能称为时钟 skew?

先说后一个问题。在一块 FPGA 内部, 时钟 skew 问题确实已经被 FPGA 的时钟方案树解决, 在这个前提下 clk\_o 到 clk\_o\_reg 充其量只能叫做时钟延时, 而不能称之为时钟 skew。可惜的是 FPGA 的设计不可能永远只在内部做事情, 它必然和外部交换数据。例如从外部传过来一个 32 位的数据以及随路时钟, 数据和随路时钟之间满足建立保持时间关系 (Setup Hold time), 你如何将这 32 位的数据接收进来? 如果你不使用 DCM, 直接将 clkin 接在 BUFG 的输入引脚上, 那么从你的 clk\_o\_reg 就必然和 clkin 之间有个延时, 那么你的 clk\_o\_reg 还能保持和进来的数据之间的建立保持关系吗? 显然不能。相反, 如果你采用了 DCM, 接上反馈时钟, 那么 clk\_o\_reg 和 clkin 同相, 就可以利用它去锁存进来的数据。可见, DCM+BUFG 的方案就是为了解决这个问题。而这个时候 clk\_o 到 clk\_o\_reg 的延时, 我们可以看到做内部寄存器和其他芯片传过来的数据之间的时钟 skew。

由此, 我们可以得出一个推论, 从晶振出来的时钟作为 FPGA 的系统时钟时, 我们可以不经过 DCM, 而直接接到 BUFG 上就可以, 因为我们并不在意从 clkin 到 clk\_o\_reg 的这段延时。

## 8、FPGA 时钟系统

### 1, FPGA 的全局时钟是什么?

FPGA 的全局时钟应该是从晶振分出来的, 最原始的频率。其他需要的各种频率都是在这个基础上利用 PLL 或者其他分频手段得到的。

### 2, 全局时钟和 BUFG:

BUFG, 输入为固定管脚, 输出为 H 型全铜全局高速网络, 这样抖动和到任意触发器的延时差最小, 这个也就是 FPGA 做同步设计可以不需要做后仿真的原因。

全局时钟: 今天我们从另一个角度来看一下时钟的概念: 时钟是 D 触发器的重要组成部分, 一个有效边沿使得 D 触发器进行一次工作。而更多的时候, D 触发器保持住上次的值。对于 D 触发器来说, 可以将输入信号和时钟作比较。也许你会问, 这么比较有什么意义。首先看我们比较得出什么东西:

翻转率:  $R = Dr / Cr \times 100\%$

就是 D 触发器改变一次值与时钟有效沿个数的比值。

举例: 你写了一个来一个时钟有效沿就取一次反的电路, 那么他的翻转率就是 100%, 翻转率和你的 FPGA 的功率有很大关系, 翻转率越高, FPGA 功率越高。

### 3, 全局时钟不够用是什么意思?

因为全局时钟需要驱动很多模块, 所以全局时钟引脚需要有很大的驱动能力, FPGA 一般都有有一些专门的引脚用于作为全局时钟用, 他们的驱动能力比较强。但是如果这些引脚用完了, 就只能用一般的引脚了, 而他们的驱动能力不强, 有可能不能满足你的时序要求。(驱动能力小的, 产生的延迟会大一些)

理论上, FPGA 的任意一个管脚都可以作为时钟输入端口, 但是 FPGA 专门设计了全局时钟, 全局时钟总线是一条专用总线, 到达片内各部分触发器的时间最短,

所以用全局时钟芯片工作最可靠，但是如果你设计的时候时钟太多，FPGA 上的全局时钟管脚用完了就出现不够用的情况。

#### 4，什么是第二全局时钟？

比如我有一个同步使能信号，连接到 FPGA 内部 80%的资源(但不是时钟),这个时候，你的信号走线到达各个 D 触发器的延迟差很大，或者翻转率比较大的时候(>40%)，这个时候你就需要使用第二全局时钟资源。

第二全局时钟资源的驱动能力和时钟抖动延迟等指标仅次于全局时钟信号。第二全局时钟资源其实是通过片内的高速行列总线来实现的，而不像全局时钟总线是一条专用总线。第二全局时钟总线是通过软件布线得到的，所以硬指标肯定是拼不过全局时钟总线。特别是当你在已经有 80%以上的布线率的情况下，可能会出现约束第二全局时钟资源失败的情况。

#### 5，CCLK:

CCLK: FPGA 同步配置时钟。如果配置模式为主模式，则该时钟由 FPGA 器件生成，并输出；如果配置模式为从模式，则该时钟由外部提供；  
当所配置的数据存放在 PROM 中，即通过 PROM 来配置器件时，必须选择 CCLK 时钟；

USER CLOCK: 用户定义的配置时钟信号，该配置时钟目前很少采用；

JTAG CLOCK: JTAG 模式的配置时钟，该时钟提供给内部的 JTAG 控制逻辑。

默认值为: CCLK

#### 6，CCLK 是怎么产生的:

CCLK 的产生根据配置模式不同而不同，如果设置为 Master 模式，则由内部的震荡电路产生，作为外部 ROM 的工作时钟，默认为 6MHZ，可通过配置选项设置；如果设置为 Slave 模式，则由计算机(或其他下载设备)提供，作为芯片内部下载电路的工作时钟；在 JTAG 模式情况下，CCLK 不输出，此时芯片内部下载电路时钟由内部震荡电路提供，TCK 仅用作边界扫描相关电路时钟。

补充: FPGA 的主配置模式中，CCLK 信号是如何产生的？

CCLK 是由 FPGA 内部一个晶振电路产生的，同时 ISE 的软件在生成 BIT 流文件时，有个 CCLK CONFIG 选项，这个选项只有在时钟为 CCLK 时才可以起作用，可以在 4-60MHz 选择，可以控制 CCLK 的频率。

在主从模式配置，配置数据的前 60 个字节导入 FPGA 之前，CCLK 一直是 2.5MHz，接下来由于前 60 个配置字节的作用，CCLK 改为 CONFIG 设定的频率，直到结束，一般 CONFIG 默认的频率是 4MHz。

#### 7，FPGA 中全局时钟怎么用啊？是把时钟接到 FPGA 的全局时钟输入引脚后，就起到全局时钟的作用了，还是在编译时需要制定某个时钟为全局时钟阿？

其实全局时钟的使用关键在你的代码... 如果你的代码中只用了一个时钟作为所有的或者大部分触发器的时钟，编译器自然会把它编译为全局时钟。当然硬件连

接上还是用全局时钟引脚较好，尤其是带 PLL 的，不是所有的全局时钟脚都能用 PLL。

无论是用离散逻辑、可编程逻辑，还是用全定制硅器件实现的任何数字设计，为了成功地操作，可靠的时钟是非常关键的。设计不良的时钟在极限的温度、电压或制造工艺的偏差情况下将导致错误的行为，并且调试困难、花销很大。在设计 PLD/FPGA 时通常采用几种时钟类型。时钟可分为如下四种类型：全局时钟、门控时钟、多级逻辑时钟和波动式时钟。多时钟系统能够包括上述四种时钟类型的任意组合。

### 1. 全局时钟

对于一个设计项目来说，全局时钟(或同步时钟)是最简单和最可预测的时钟。在 PLD/FPGA 设计中最好的时钟方案是：由专用的全局时钟输入引脚驱动的单个或多个主时钟去钟控设计项目中的每一个触发器。只要可能就应尽量在设计项目中采用全局时钟。PLD/FPGA 都具有专门的全局时钟引脚，它直接连到器件中的每一个寄存器。这种全局时钟提供器件中最短的时钟到输出的延时。

图 1 示出全局时钟的实例。图 1 定时波形示出触发器的数据输入 D[1..3]应遵守建立时间和保持时间的约束条件。建立和保持时间的数值在 PLD 数据手册中给出，也可用软件的定时分析器计算出来。如果在应用中不能满足建立和保持时间的要求，则必须用时钟同步输入信号(参看下一章“异步输入”)。

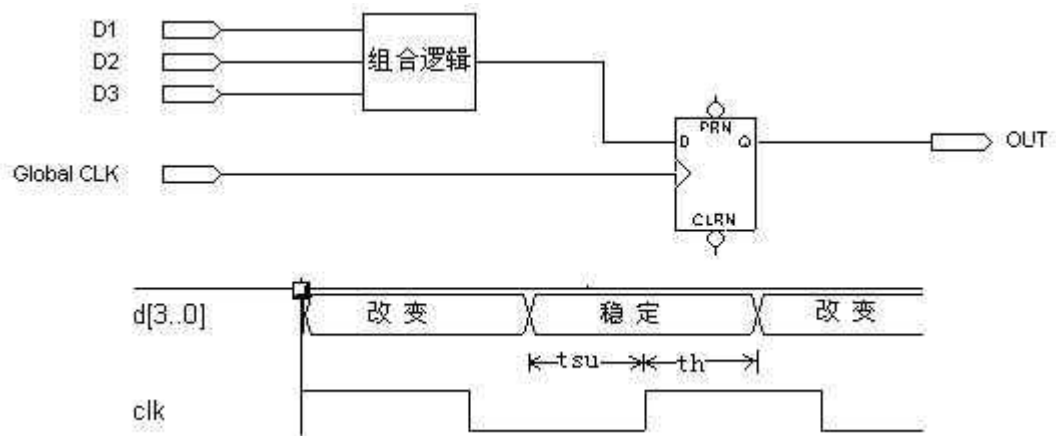


图 1 全局时钟

(最好的方法是用全局时钟引脚去钟控 PLD 内的每一个寄存器，于是数据只要遵守相对时钟的建立时间  $t_{su}$  和保持时间  $t_h$ )

### 2. 门控时钟

在许多应用中，整个设计项目都采用外部的全局时钟是不可能或不实际的。PLD 具有乘积项逻辑阵列时钟(即时钟是由逻辑产生的)，允许任意函数单独地钟控各个触发器。然而，当你用阵列时钟时，应仔细地分析时钟函数，以避免毛刺。

通常用阵列时钟构成门控时钟。门控时钟常常同微处理器接口有关，用地址线去控制写脉冲。然而，每当用组合函数钟控触发器时，通常都存在着门控时钟。如果符合下述条件，门控时钟可以象全局时钟一样可靠地工作：

- 1. 驱动时钟的逻辑必须只包含一个“与”门或一个“或”门。如果采用任何附加逻辑在某些工作状态下，会出现竞争产生的毛刺。
- 2. 逻辑门的一个输入作为实际的时钟，而该逻辑门的所有其它输入必须当成地址或控制线，它们遵守相对于时钟的建立和保持时间的约束。

图 2 和图 3 是可靠的门控时钟的实例。在 图 2 中，用一个“与”门产生门控时钟，在 图 3 中，用一个“或”门产生门控时钟。在这两个实例中，引脚 nWR 和 nWE 考虑为时钟引脚，引脚 ADD[o. . 3]是地址引脚，两个触发器的数据是信号 D[1..n]经随机逻辑产生的。

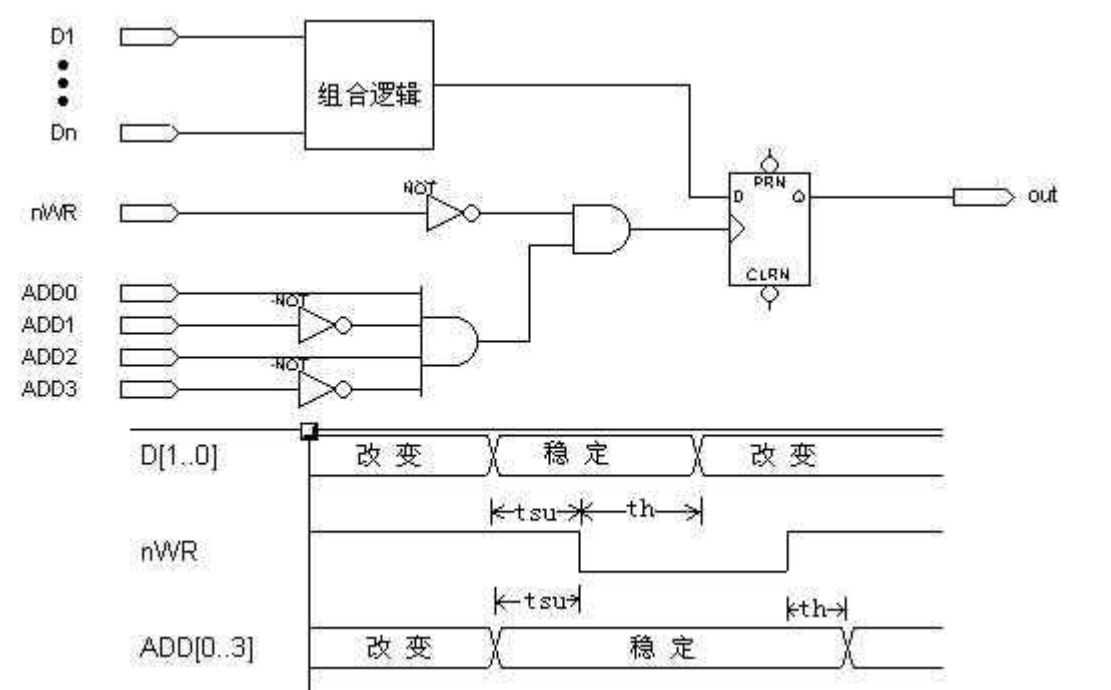


图 2 “与”门门控时钟

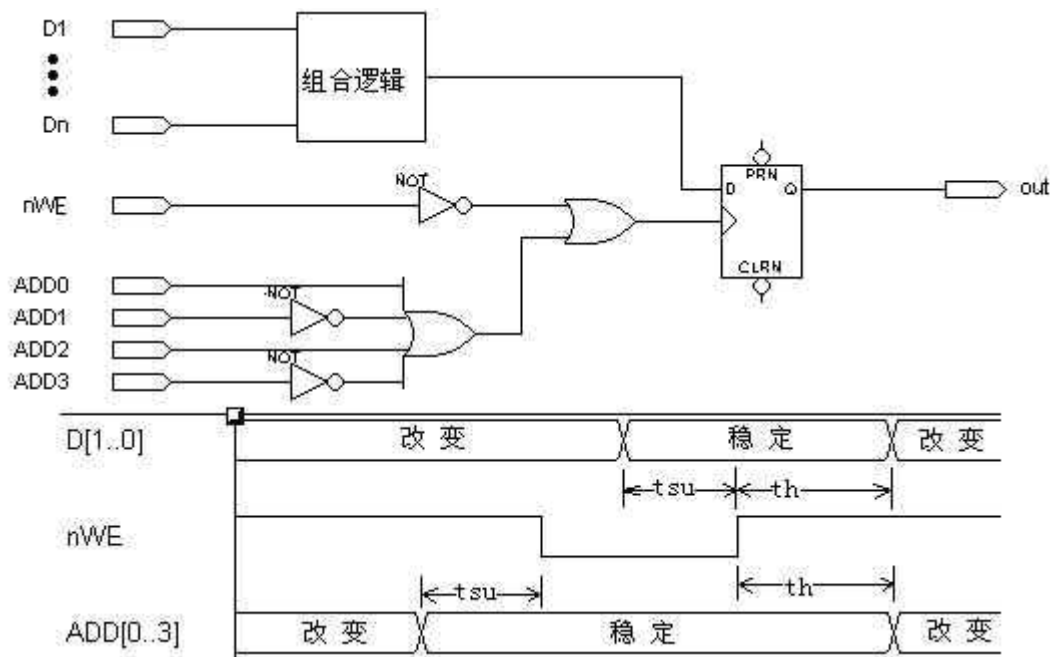


图3 “或”门门控时钟

图2和图3的波形图显示出有关的建立时间和保持时间的要求。这两个设计项目的地址线必须在时钟保持有效的整个期间内保持稳定(nWR和nWE是低电平有效)。如果地址线在规定的时间内未保持稳定,则在时钟上会出现毛刺,造成触发器发生错误的状态变化。另一方面,数据引脚D[1..n]只要求在nWR和nWE的有效边沿处满足标准的建立和保持时间的规定。

我们往往可以将门控时钟转换成全局时钟以改善设计项目的可靠性。图4示出如何用全局时钟重新设计图2的电路。地址线在控制D触发器的使能输入,许多PLD设计软件,如MAX+PLUSII软件都提供这种带使能端的D触发器。当ENA为高电平时,D输入端的值被钟控到触发器中:当ENA为低电平时,维持现在的状态。

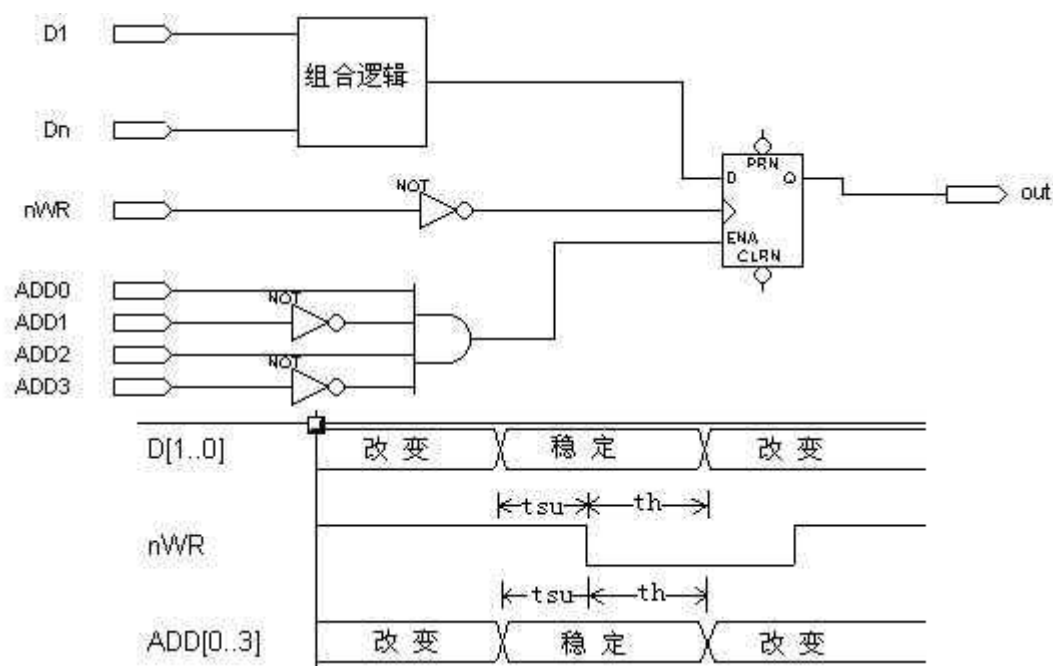


图 4 “与”门门控时钟转化成全局时钟

图 4 中重新设计的电路的定时波形表明地址线不需要在 nWR 有效的整个期间内保持稳定；而只要求它们和数据引脚一样符合同样的建立和保持时间，这样对地址线的要求就少很多。

图 给出一个不可靠的门控时钟的例子。3 位同步加法计数器的 RCO 输出用来钟控触发器。然而，计数器给出的多个输入起到时钟的作用，这违反了可靠门控时钟所需的条件之一。在产生 RCO 信号的触发器中，没有一个能考虑为实际的时钟线，这是因为所有触发器在几乎相同的时刻发生翻转。而我们并不能保证在 PLD/FPGA 内部 QA, QB, QC 到 D 触发器的布线长短一致，因此，如图 5 的时间波形所示，在器从 3 计到 4 时，RCO 线上会出现毛刺（假设 QC 到 D 触发器的路径较短，即 QC 的输出先翻转）。

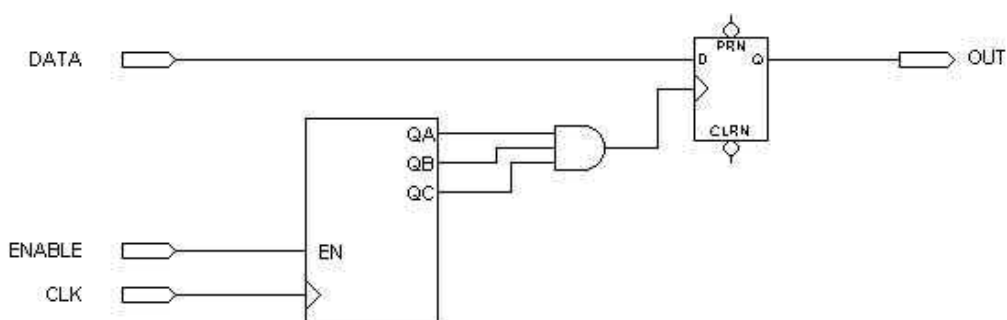


图 5 不可靠的门控时钟

(定时波形示出在计数器从 3 到 4 改变时，RCO 信号如何出现毛刺的)

图 6 给出一种可靠的全局钟控的电路，它是图 5 不可靠计数器电路的改进，RCO 控制 D 触发器的使能输入。这个改进不需要增加 PLD 的逻辑单元。

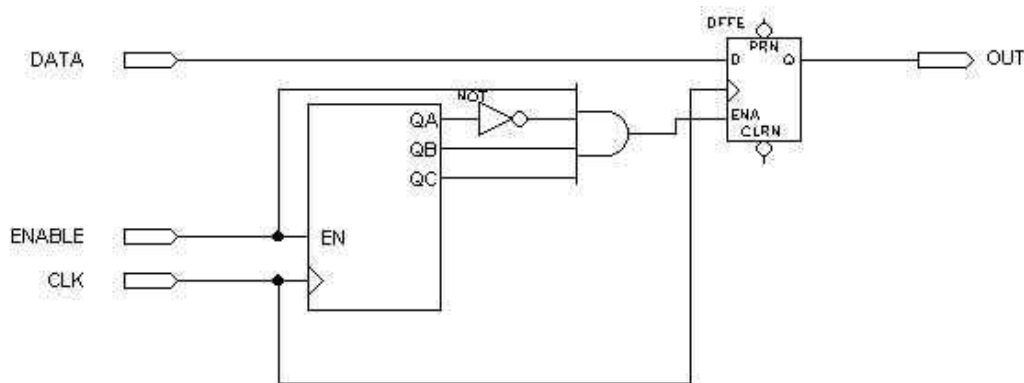


图 6 不可靠的门控时钟转换为全局时钟

(这个电路等效于图 5 电路，但却可靠的多)

### 3. 多级逻辑时钟

当产生门控时钟的组合逻辑超过一级(即超过单个的“与”门或“或”门)时，证设计项目的可靠性变得很困难。即使样机或仿真结果没有显示出静态险象，但实际上仍然可能存在着危险。通常，我们不应该用多级组合逻辑去钟控 PLD 设计中的触发器。

图 7 给出一个含有险象的多级时钟的例子。时钟是由 SEL 引脚控制的多路选择器输出的。多路选择器的输入是时钟(CLK)和该时钟的 2 分频(DIV2)。由图 7 的定时波形图看出，在两个时钟均为逻辑 1 的情况下，当 SEL 线的状态改变时，存在静态险象。险象的程度取决于工作的条件。多级逻辑的险象是可以去除的。例如，你可以插入“冗余逻辑”到设计项目中。然而，PLD/FPGA 编译器在逻辑综合时会去掉这些冗余逻辑，使得验证险象是否真正被去除变得困难了。为此，必须应寻求其它方法来实现电路的功能。

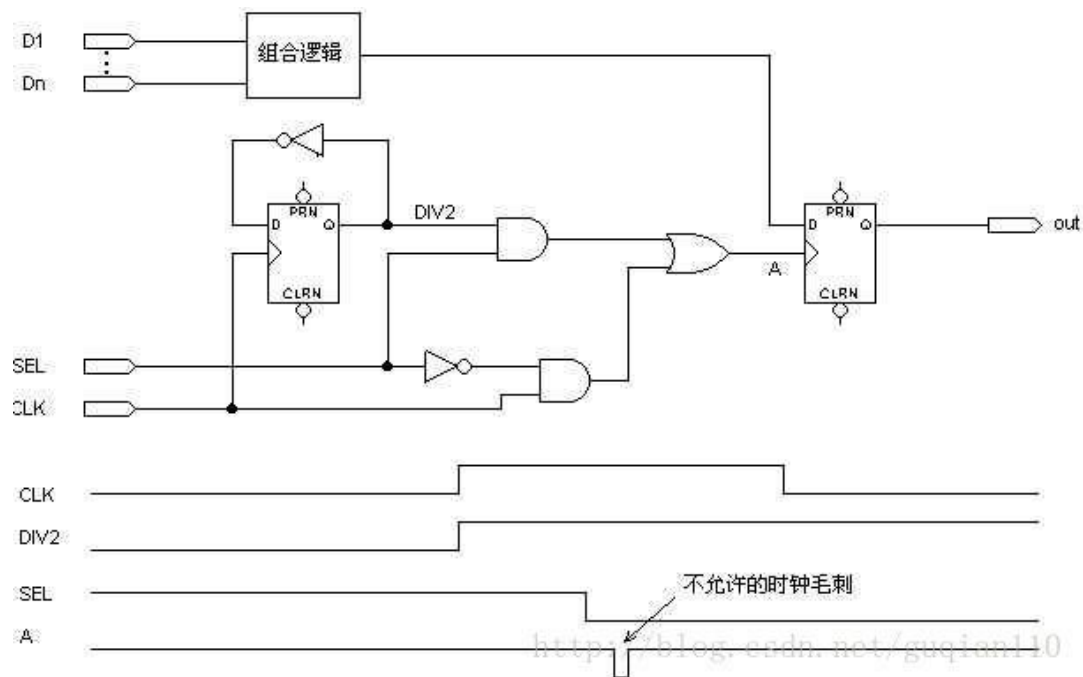


图 7 有静态险象的多级时钟

图 8 给出 图 7 电路的一种单级时钟的替代方案。图中 SEL 引脚和 DIV2 信号用于使能 D 触发器的使能输入端，而不是用于该触发器的时钟引脚。采用这个电路并不需要附加 PLD 的逻辑单元，工作却可靠多了。不同的系统需要采用不同的方法去除多级时钟，并没有固定的模式。

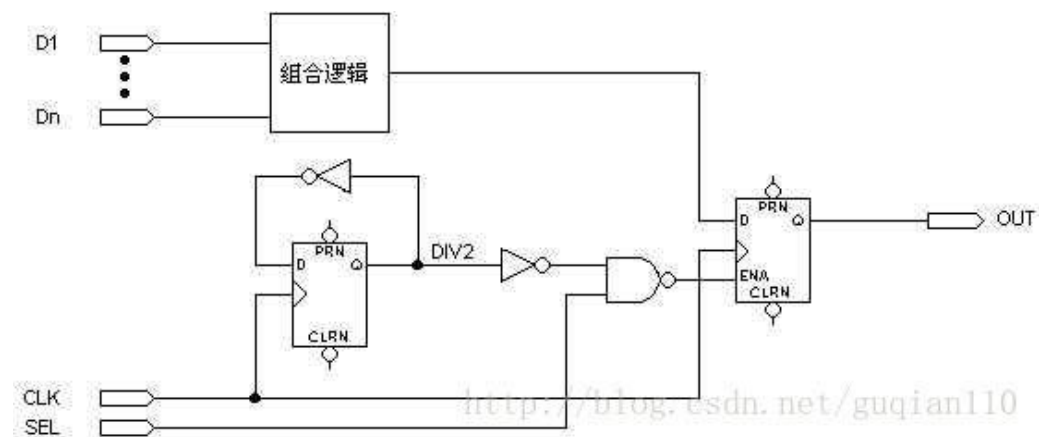


图 8 无静态险象的多级时钟

(这个电路逻辑上等效于图 7，但却可靠的多)

#### 4. 行波时钟

另一种流行的时钟电路是采用行波时钟，即一个触发器的输出用作另一个触发器的时钟输入。如果仔细地设计，行波时钟可以象全局时钟一样地可靠工作。然而，行波时钟使得与电路有关的定时计算变得很复杂。行波时钟在行波链上各触发器

的时钟之间产生较大的时间偏移，并且会超出最坏情况下的建立时间、保持时间和电路中时钟到输出的延时，使系统的实际速度下降。

用计数翻转型触发器构成异步计数器时常采用行波时钟，一个触发器的输出钟控下一个触发器的输入，参看图 9 同步计数器通常是代替异步计数器的更好方案，这是因为两者需要同样多的宏单元而同步计数器有较快的时钟到输出的时间。图 10 给出具有全局时钟的同步计数器，它和 图 9 功能相同，用了同样多的逻辑单元实现，却有较快的时钟到输出的时间。几乎所有 PLD 开发软件都提供多种多样的同步计数器。

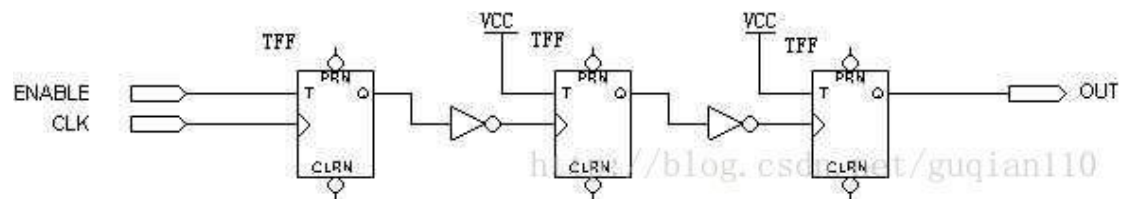


图 9 行波时钟

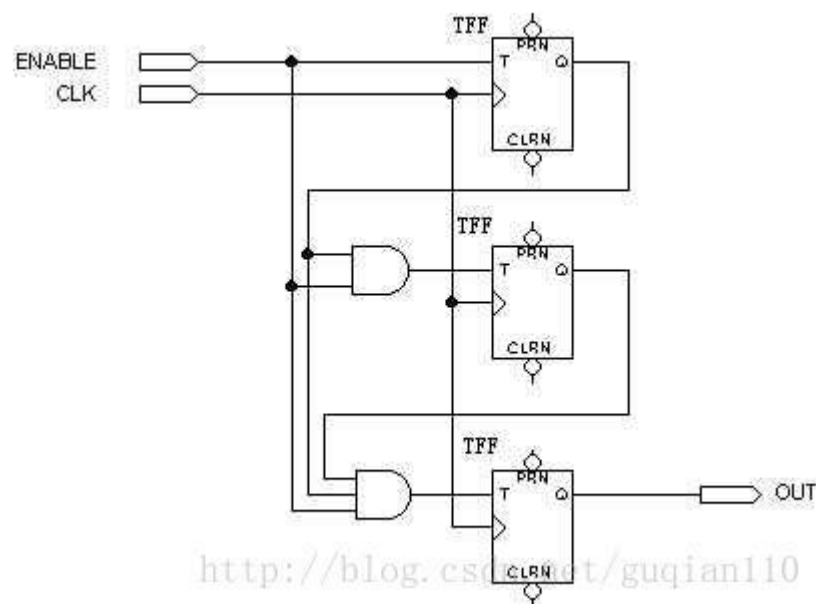


图 10 行波时钟转换成全局时钟

(这个 3 位计数器是图 9 异步计数器的替代电路，它用了同样的 3 个宏单元，但有更短的时钟到输出的延时)

## 5, 多时钟系统

许多系统要求在同一个 PLD 内采用多时钟。最常见的例子是两个异步微处理器之间的接口，或微处理器和异步通信通道的接口。由于两个时钟信号之间要求一定的建立和保持时间，所以，上述应用引进了附加的定时约束条件。它们也会要求将某些异步信号同步化。

图 11 给出一个多时钟系统的实例。CLK\_A 用以钟控 REG\_A, CLK\_B 用于钟控 REG\_B, 由于 REG\_A 驱动着进入 REG\_B 的组合逻辑, 故 CLK\_A 的上升沿相对于 CLK\_B 的上升沿有建立时间和保持时间的要求。由于 REG\_B 不驱动馈到 REG\_A 的逻辑, CLK\_B 的上升沿相对于 CLK\_A 没有建立时间的要求。此外, 由于时钟的下降沿不影响触发器的状态, 所以 CLK\_A 和 CLK\_B 的下降沿之间没有时间上的要求。如图 4, 2. II 所示, 电路中有两个独立的时钟, 可是, 在它们之间的建立时间和保持时间的要求是不能保证的。在这种情况下, 必须将电路同步化。图 12 给出 REG\_A 的值(如何在使用前)同 CLK\_B 同步化。新的触发器 REG\_C 由 CLK\_B 触控, 保证 REG\_G 的输出符合 REG\_B 的建立时间。然而, 这个方法使输出延时了一个时钟周期。

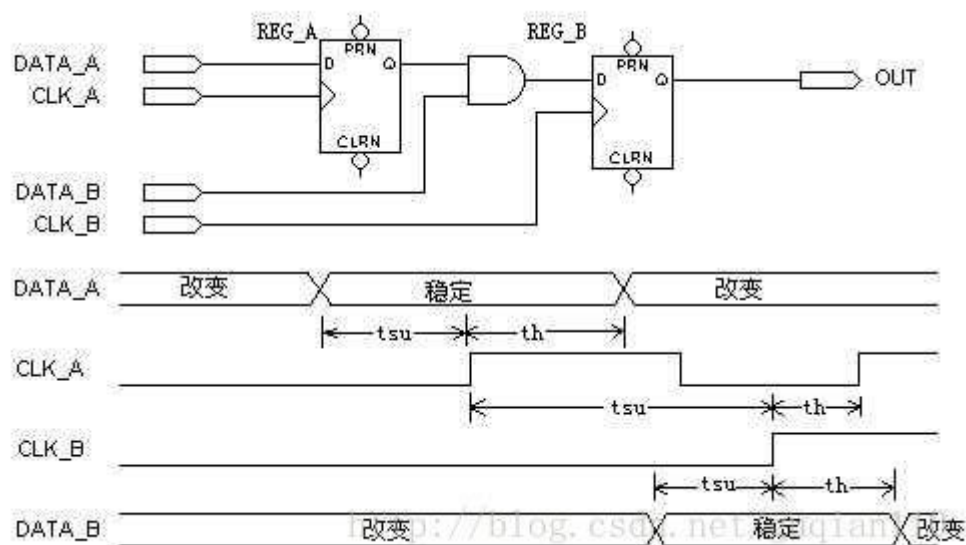


图 11 多时钟系统

(定时波形示出 CLK\_A 的上升沿相对于 CLK\_B 的上升沿有建立时间和保持时间的约束条件)

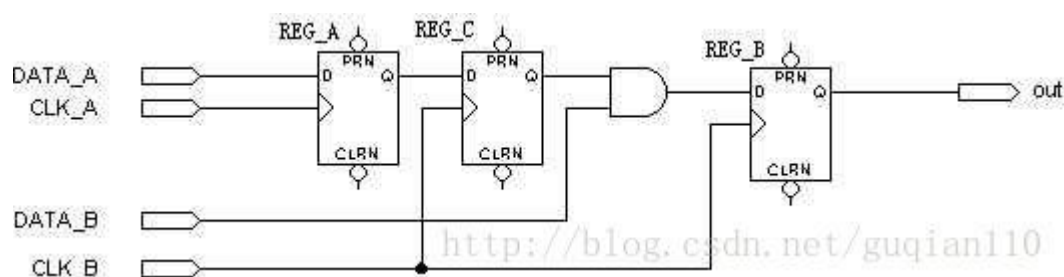


图 12 具有同步寄存器输出的多时钟系统

(如果 CLK\_A 和 CLK\_B 是相互独立的, 则 REG\_A 的输出必须在它馈送到 REG\_B 之前, 用 REG\_C 同步化)

在许多应用中只将异步信号同步化还是不够的, 当系统中有两个或两个以上非同源时钟的时候, 数据的建立和保持时间很难得到保证, 我们将面临复杂的时间问题。最好的方法是将所有非同源时钟同步化。使用 PLD 内部的锁项环(PLL 或 DLL)是一个效果很好的方法, 但不是所有 PLD 都带有 PLL、DLL, 而且带有 PLL 功能

的芯片大多价格昂贵,所以除非有特殊要求,一般场合可以不使用带 PLL 的 PLD。这时我们需要使用带使能端的 D 触发器,并引入一个高频时钟。

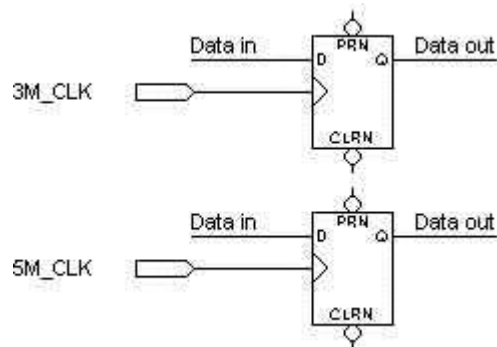
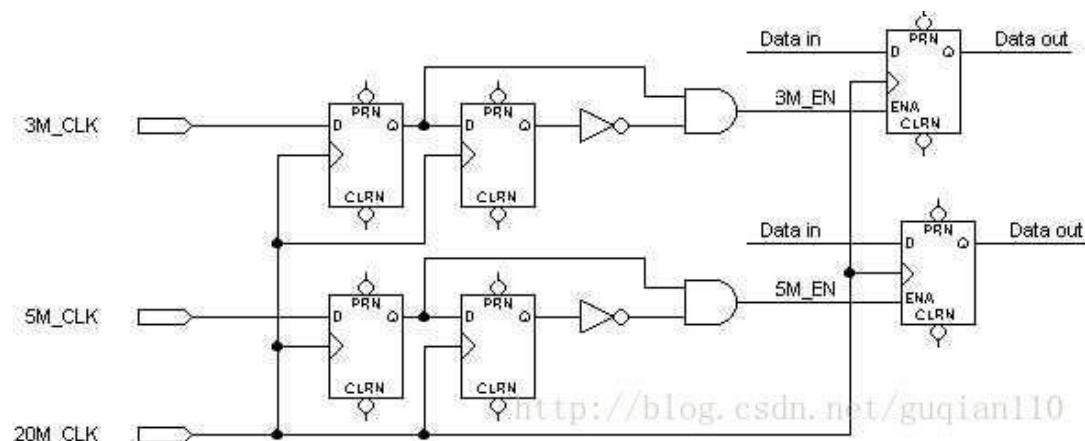


图 13 不同源时钟

如图 13 所示,系统有两个不同源时钟,一个为 3MHz,一个为 5MHz,不同的触发器使用不同的时钟。为了系统稳定,我们引入一个 20MHz 时钟,将 3M 和 5M 时钟同步化。20M 的高频时钟将作为系统时钟,输入到所有触发器的时钟端。3M\_EN 和 5M\_EN 将控制所有触发器的使能端。即原来接 3M 时钟的触发器,接 20M 时钟,同时 3M\_EN 将控制该触发器使能,原接 5M 时钟的触发器,也接 20M 时钟,同时 5M\_EN 将控制该触发器使能。这样我们就可以将任何非同源时钟同步化。



同步化任意非同源时钟

(一个 DFF 和后面非门,与门构成时钟上升沿检测电路)

另外,异步信号输入总是无法满足数据的建立保持时间,容易使系统进入亚稳态,所以也建议设计者把所有异步输入都先经过双触发器进行同步化。

### 异步时钟同步化

通过双触发器接口,异步信号输入总是无法满足数据的建立保持时间,所以建议大家把所有异步输入都先经过双触发器进行同步化。如图所示,时钟域 clk\_s 传给时钟域 clk\_d 的数据经过了双触发器的同步处理,相同的,时钟域 clk\_d 经双触发器传给时钟域 clk\_s 的数据

通过高频时钟同步化，当在单个系统中有两个或两个以上非同源时钟的时候，数据的建立和保持时间很难得到保证，我们将面临复杂的时间问题，最好的方法是将所有非同源时钟同步化：选用一个频率是它们的时钟频率公倍数的高频主时钟将他们进行同步。

假设系统有两个不同源时钟，一个为 3MHz，一个为 5MHz，不同的触发器使用不同的时钟。为了系统稳定，假设我们引入一个 20MHz 时钟。

```
module original(clk,clk5m,clk3m,clk3men,clk5men);
```

```
input clk;
```

```
input clk3m;
```

```
input clk5m;
```

```
output clk3men;
```

```
output clk5men;
```

```
reg clk3mreg1;
```

```
reg clk5mreg1;
```

```
reg clk3mreg2;
```

```
reg clk5mreg2;
```

```
always @(posedge clk)
```

```
begin
```

```
    clk3mreg1<=clk3m;
```

```
    clk3mreg2<=clk3mreg1;
```

```
    clk5mreg1<=clk5m;
```

```
    clk5mreg2<=clk5mreg1;
```

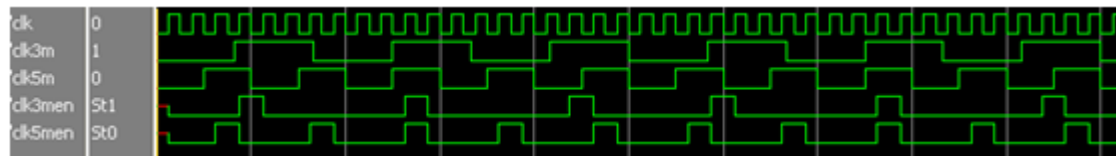
end

```
assign clk3men=clk3mreg1&(~clk3mreg2);
```

```
assign clk5men=clk5mreg1&(~clk5mreg2);
```

endmodule

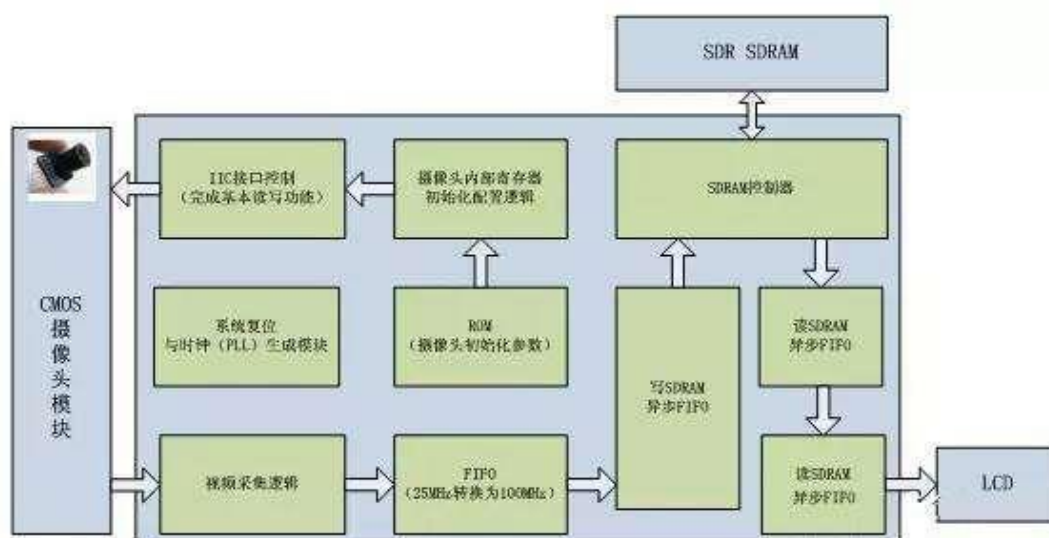
用 modelsim 仿真后得到的时序图如图所示



## 9、如何确定时序约束数值

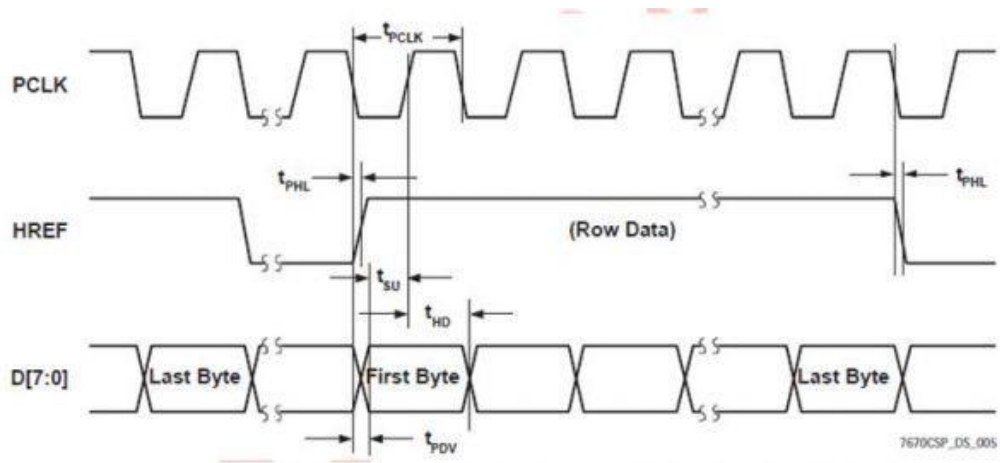
FPGA 工程的功能框图如图所示。上电初始，FPGA 需要通过 IIC 接口协议对摄像头模块进行寄存器初始化配置。这个初始化的基本参数，如初始化地址和数据存储在预先配置好的 FPGA 内嵌 ROM 中。在初始化配置完成后，摄像头就能够持续输出 RGB 标准的视频数据流，FPGA 通过对其相应的时钟、行频和场频进行检测，从而一帧一帧的实时采集图像数据。

采集到的视频数据先通过一个 FIFO，将原本 25MHz 频率下同步的数据流转换到 100MHz 频率下。接着讲这个数据再送入写 SDRAM 缓存的 FIFO 中，最终这个 FIFO 每满 160 个数据就会将其写入 SDRAM 的相应地址中。在另一侧，使用另一个异步 FIFO 将 SDRAM 缓存的图像数据送个 LCD 驱动模块。LCD 驱动模块不断的读出新的现实图像，并且驱动 3.5 寸液晶屏工作。



由于这个工程是移植过来的，SDRAM 的时序约束已经添加好并且很好的收敛了。但是，新增加的 CMOS sensor 的接口也需要做相应的时序约束。下面我们就来探讨下它的时序该如何做约束。

先看看 CMOS Sensor 的 datasheet 中提供的时序波形和相应的建立、保持时间要求。波形如图所示。

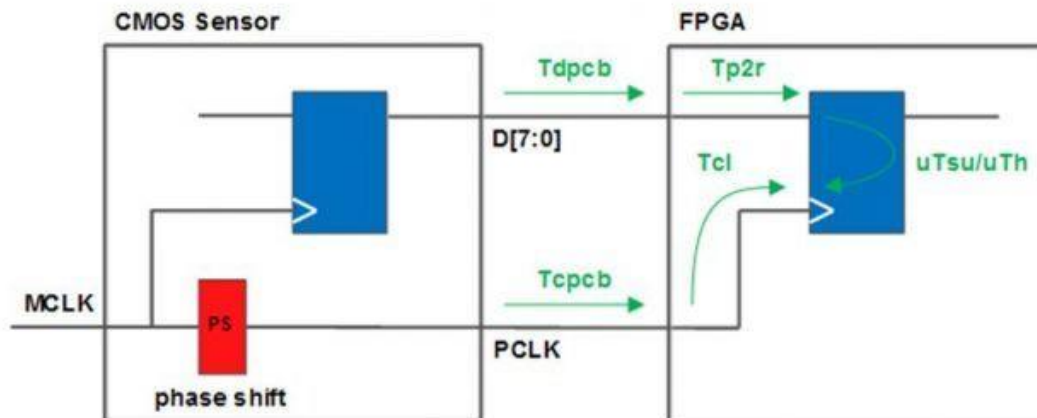


波形中出现的时间参数定义如下表所示。

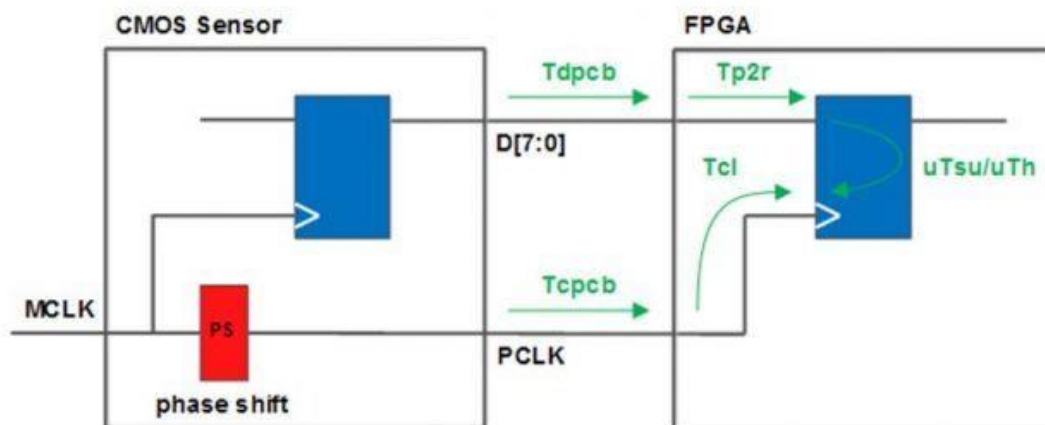
名称	定义	最小值	标准值	最大值	单位
tPDV	PCLK下降沿到数据有效时间。	5			ns
tSU	D[7:0]建立时间。	15			ns
tHD	D[7:0]保持时间。	8			ns
tPHH	PCLK下降沿到HREF上升沿时间。	0			ns
tPHL	PCLK下降沿到HREF下降沿时间。	5			ns

我们可以简单分析下这个 datasheet 中提供的时序波形和参数提供了一些什么样的有用信息。我们重点关注 PCLK 和 D[7:0] 的关系，HREF 其实也可以归类到 D[7:0] 中一起分析，他们的时序关系基本是一致的（如果存在偏差，也可以忽略不计）。这个波形实际上表达的是从 Sensor 的芯片封装管脚上输出的 PCLK 和 D[7:0] 的关系。而在理想状况下，经过 PCB 走线将这组信号连接到其他的芯片上（如 CPU 或 FPGA），若尽可能保持走线长度，在其他芯片的管脚上，PCLK 和 D[7:0] 的关系基本还是不变的。那么，对于采集端来说，用 PCLK 的上升沿去锁存 D[7:0] 就变得理所当然了。而对于 FPGA 而言，从它的管脚到寄存器传输路径上总归是有延时存在的，那么 PCLK 和 D[7:0] 之间肯定不会是理想的对齐关系。而我们现在关心的是，相对于理想的对齐关系，PCLK 和 D[7:0] 之间可以存在多大的相位偏差（最终可能会以一个延时时间范围来表示）。在时序图中，T<sub>su</sub> 和 T<sub>h</sub> 虽然是 PCLK 和 D[7:0] 在 Sensor 内部必须保证的建立时间和保持时间关系，但它同

样是 Sensor 的输出管脚上，必须得到保证的基本时序关系。因此，我们可以认为：理想相位关系情况下，PCLK 上升沿之前的  $T_{su}$  时间（即 15ns）到上升沿后的  $T_h$  时间（即 8ns）内，D[7:0] 是稳定不变的。同样的，理想情况下，PCLK 的上升沿处于 D[7:0] 两次数据变化的中央。换句话说，在 D[7:0] 保持当前状态的情况下，PCLK 上升沿实际上在理想位置的  $T_{su}$  时间和  $T_h$  时间内都是允许的。请大家记住这一点，下面我们需要利用这个信息对在 FPGA 内部的 PCLK 和 D[7:0] 信号进行时序约束。



OK，明确了 PCLK 和 D[7:0] 之间应该保持的关系后，我们再来看看他们从 CMOS Sensor 的管脚输出后，到最终在 FPGA 内部的寄存器进行采样锁存，这个整个路径上的各种“艰难险阻”（延时）。



在这个路径分析中，我们不去考虑 CMOS Sensor 内部的时序关系，我们只关心它的输出管脚上的信号。先看时钟 PCLK 的路径延时，在 PCB 上的走线延时为  $T_{cpcb}$ ，在 FPGA 内部，从进入 FPGA 的管脚到寄存器的时钟输入端口的延时为  $T_{c1}$ 。再看数据 D[7:0] 的延时，在 PCB 上的走线延时为  $T_{dpcb}$ ，在 FPGA 内部的管脚到寄存器输入端口延时为  $T_{p2r}$ 。而 FPGA 的寄存器同样有建立时间  $T_{su}$  和保持时间  $T_h$  要求，也必须在整个路径的传输时序中予以考虑。

另外，从前面的分析，我们得到了 PCLK 和 D[7:0] 之间应该满足的关系。那么，为了保证 PCLK 和 D[7:0] 稳定考虑得到传输，我们可以得到这样一个基本的关系必须满足：

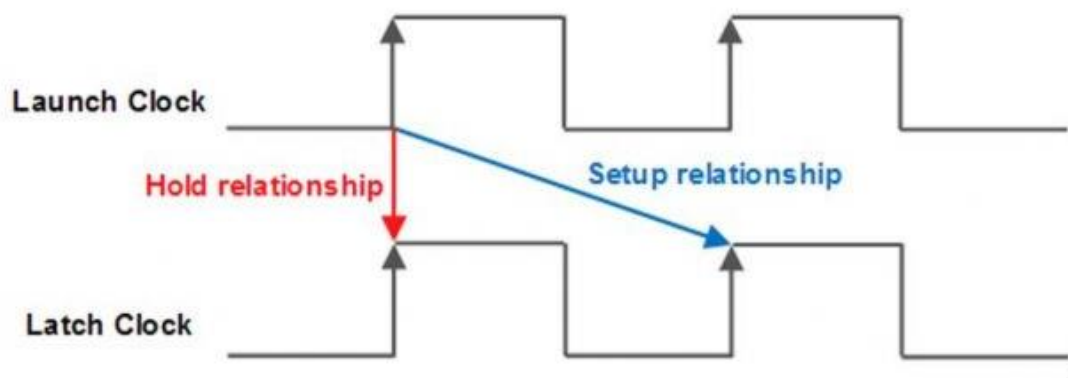
对于建立时间，有：

$$\text{Launch edge} + T_{\text{dpcb}} + T_{\text{p2r}} + T_{\text{tsu}} < \text{latch edge} + T_{\text{tpcb}} + T_{\text{tcl}}$$

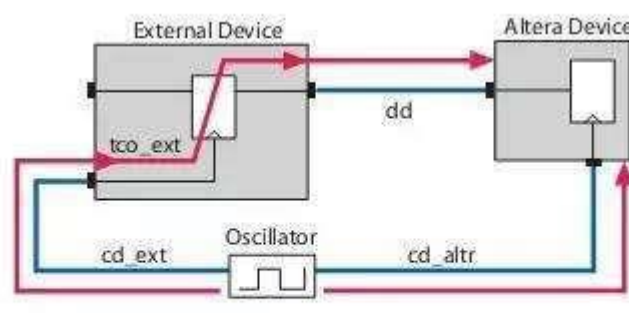
对于保持时间，有：

$$\text{Launch edge} + T_{\text{dpcb}} + T_{\text{r2p}} < \text{latch edge} + T_{\text{tpcb}} + T_{\text{tcl}} - T_{\text{th}}$$

关于 launch edge 和 latch edge，对于我们当前的设计，如下图所示。



在对这个 FPGA 的 input 接口的时序进行分析和约束之前，我们先来看看 Altera 官方是如何分析此类管脚的时序。



$$\text{input delay}_{\text{MAX}} = (\text{cd\_ext}_{\text{MAX}} - \text{cd\_altr}_{\text{MIN}}) + \text{tco\_ext}_{\text{MAX}} + \text{dd}_{\text{MAX}}$$

$$\text{input delay}_{\text{MIN}} = (\text{cd\_ext}_{\text{MIN}} - \text{cd\_altr}_{\text{MAX}}) + \text{tco\_ext}_{\text{MIN}} + \text{dd}_{\text{MIN}}$$

具体问题具体分析，我们当前的工程，状况和理想模型略有区别。实际上在上面这个模型的源寄存器端的很多信息都不用详细分析，因为我们获得的波形是来自于 Sensor 芯片的管脚。同理，我们可以得到 input delay 的计算公式如下。

$$\text{Input max delay} = (0 - T_{\text{pcb\_min}}) + T_{\text{co\_max}} + T_{\text{dpcb\_max}}$$

$$\text{Input min delay} = (0 - T_{\text{pcb\_max}}) + T_{\text{co\_min}} + T_{\text{dpcb\_min}}$$

在这两个公式中，参数  $T_{\text{co}}$  是前面我们还未曾提到的，下面我们就要分析下如何得到这个参数。 $T_{\text{co}}$  指的是理想情况下数据在源寄存器被源时钟锁存后，经过多长时间输入到管脚上。前面我们已经得到了 PCLK 和 D[7:0] 之间的关系，其实从已知的关系中，我们不难推断出  $T_{\text{co\_max}}$  和  $T_{\text{co\_min}}$ ，如图所示。若 PCLK 的时钟周期为  $T_{\text{pclk}}$ ，则：

$$T_{\text{co\_max}} = T_{\text{pclk}} - T_{\text{su}}$$

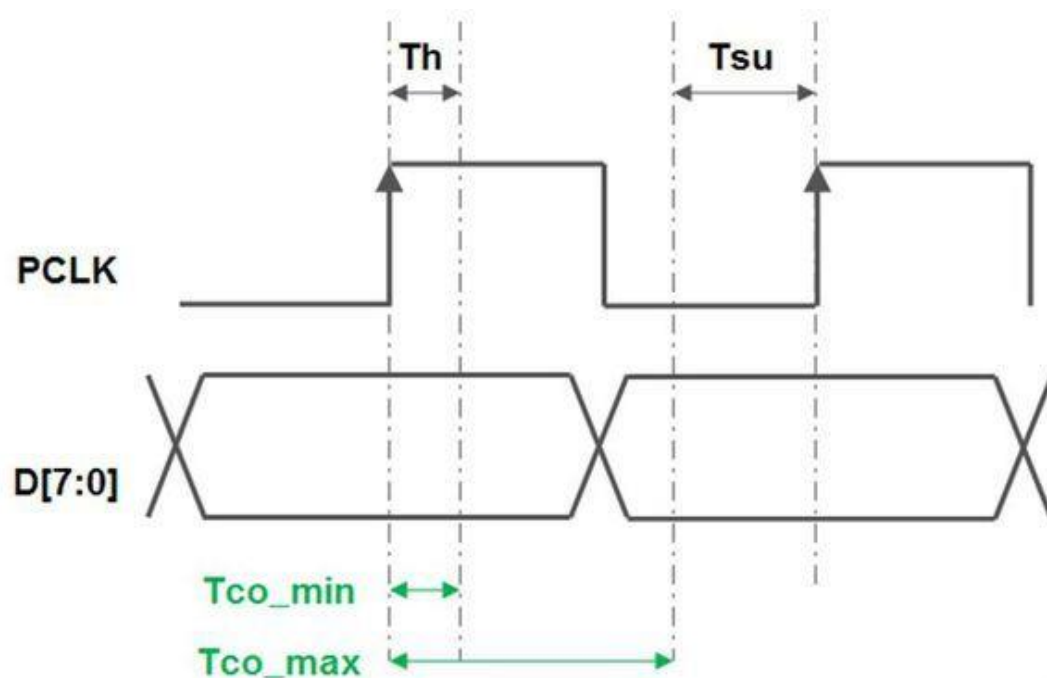
$$T_{\text{co\_min}} = T_{\text{h}}$$

在我们采样的 CMOS Sensor 图像中，PCLK 频率为 12.5MHz，即 80ns。因此，我们可以计算到：

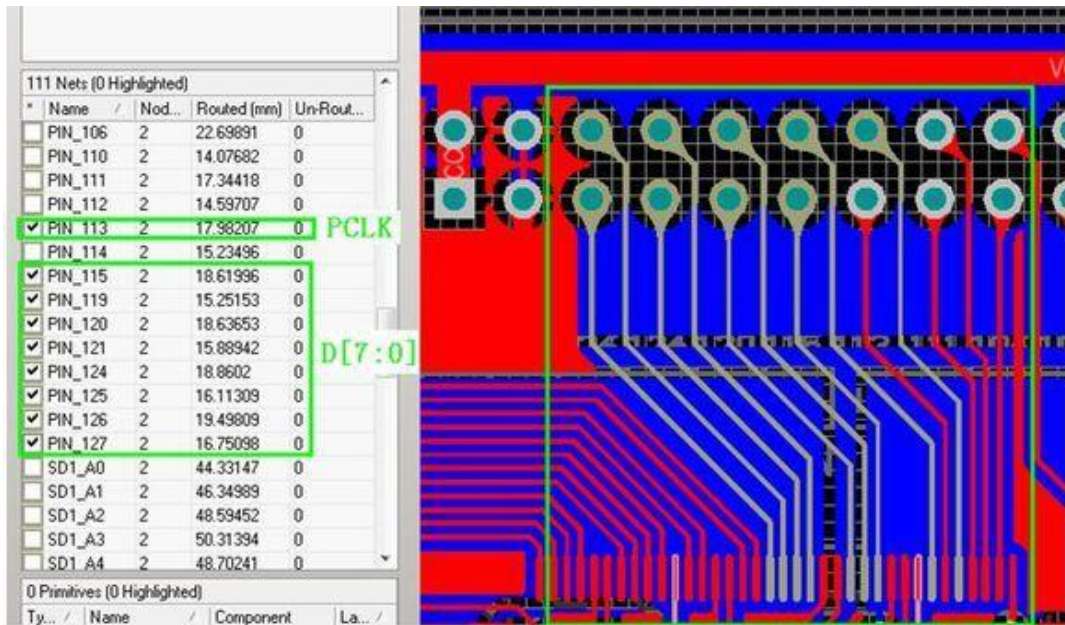
$$T_{\text{co\_max}} = 80\text{ns} - 15\text{ns} = 65\text{ns}$$

$$T_{\text{co\_min}} = 8\text{ns}$$

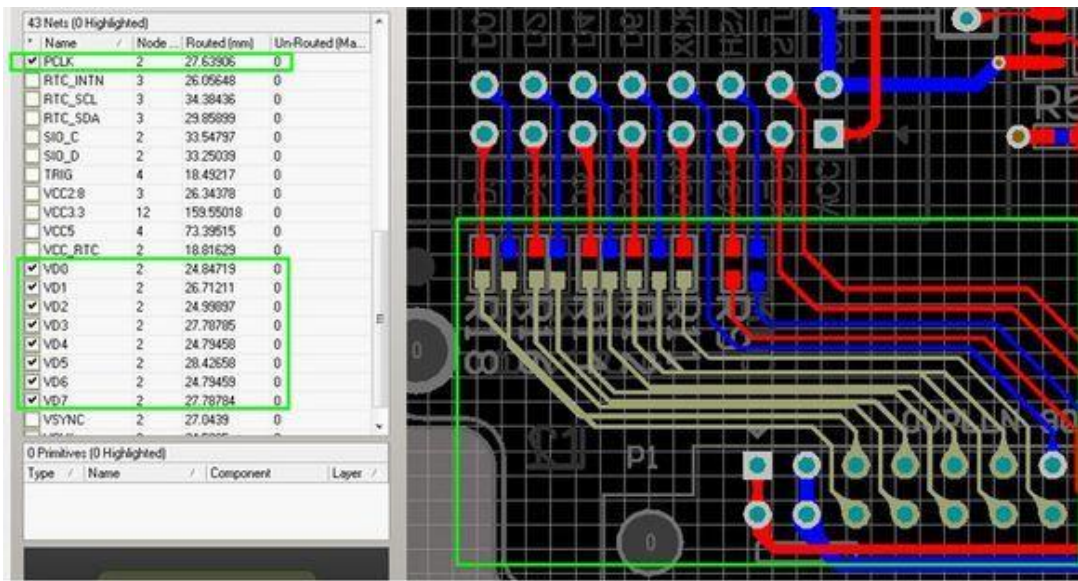
我们再看看 PCB 的走线情况，算算余下和 PCB 走线有关的延时。



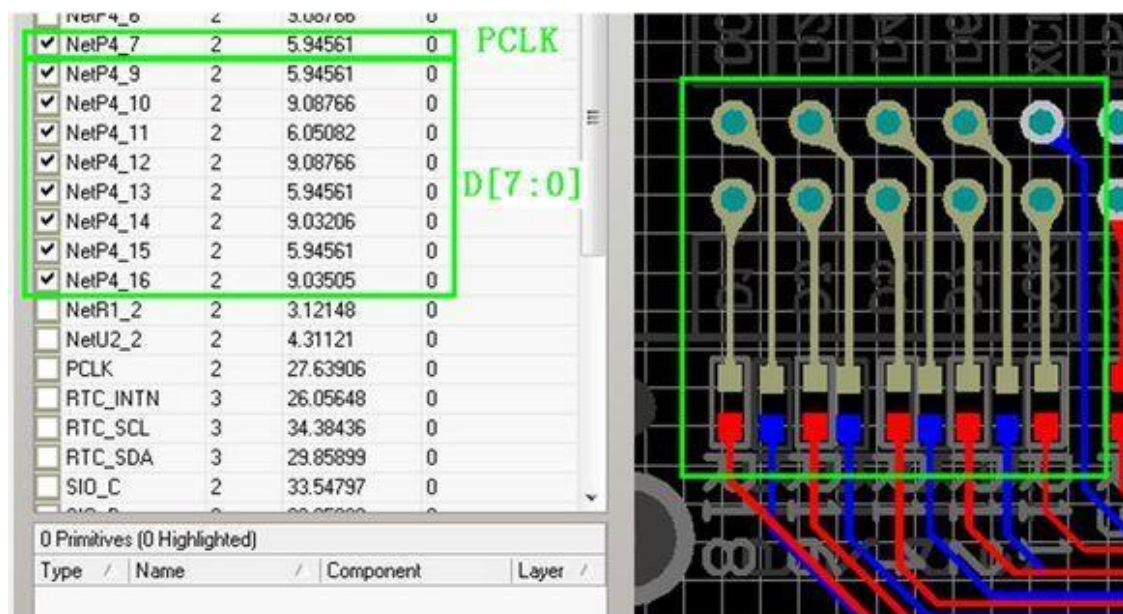
如图所示，这是 PCLK 和 D[7:0] 在 SF-CY3 核心板上的走线。



如图所示，这是 PCLK 和 D[7:0] 在 SF-SENSOR 子板上的走线，在这个板子上的走线由匹配电阻分两个部分。



根据前面的走线长度，我们可以换算一下相应的走线延时，如下表所示。因此，我们可以得到， $T_{pcb\_max} = 0.35ns$ ， $T_{pcb\_min} = 0.35ns$ ， $T_{dpcb\_max} = 0.36ns$ ， $T_{dpcb\_min} = 0.31ns$ 。



信号名	SF-CY3走线长度	SF-SENSOR走线长度1	SF-SENSOR走线长度2	总长度(m)	延时(ns)
PCLK	18	27.7	6	51.7	0.346023622
VD0	16.8	24.9	9.1	50.8	0.34
VD1	19.5	26.8	6	52.3	0.35003937
VD2	16.2	25	9.1	50.3	0.336653543
VD3	18.9	27.8	6	52.7	0.352716535
VD4	15.9	24.8	9.1	49.8	0.333307087
VD5	18.7	28.5	6	53.2	0.356062992
VD6	15.3	24.8	9.1	49.2	0.329291339
VD7	18.7	27.8	6	52.5	0.351377953
HREF	14.6	23.8	9.1	47.5	0.317913386

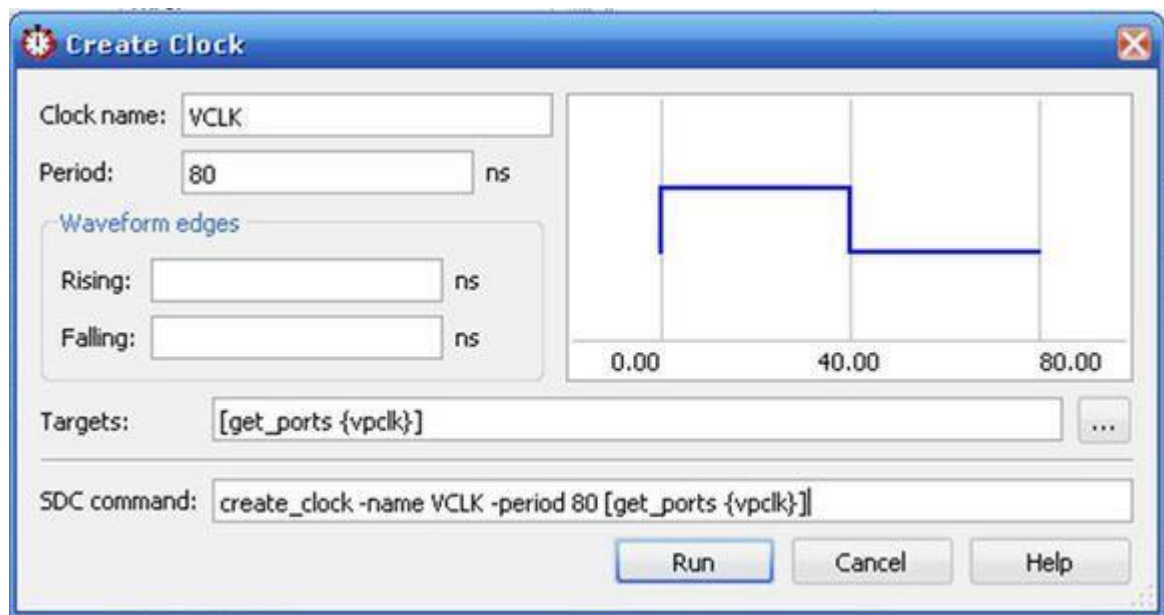
将上面得到的具体数值都代入公式，得到：

Input max delay =  $(0 - 0.35\text{ns}) + 65\text{ns} + 0.36\text{ns} = 65.01\text{ns}$

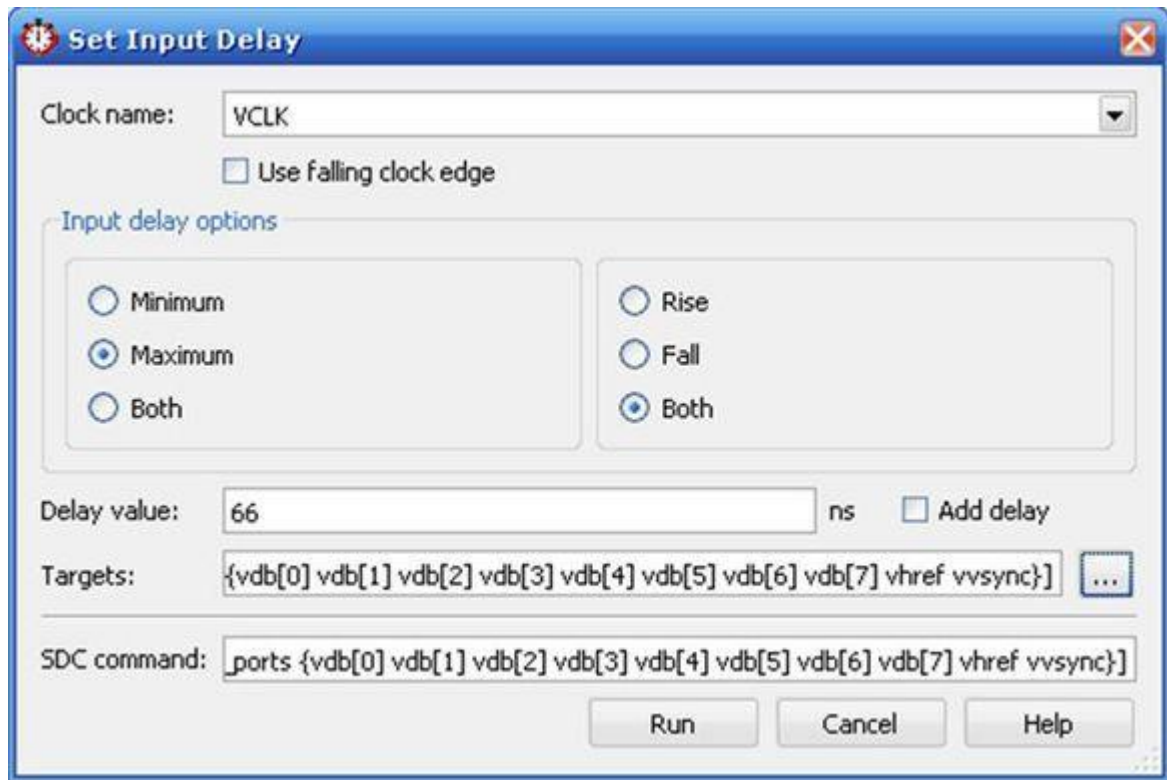
Input min delay =  $(0 - 0.35\text{ns}) + 8\text{ns} + 0.31\text{ns} = 7.96\text{ns}$

加上一些余量，我们可以去 input max delay = 66ns, input min delay = 7ns。

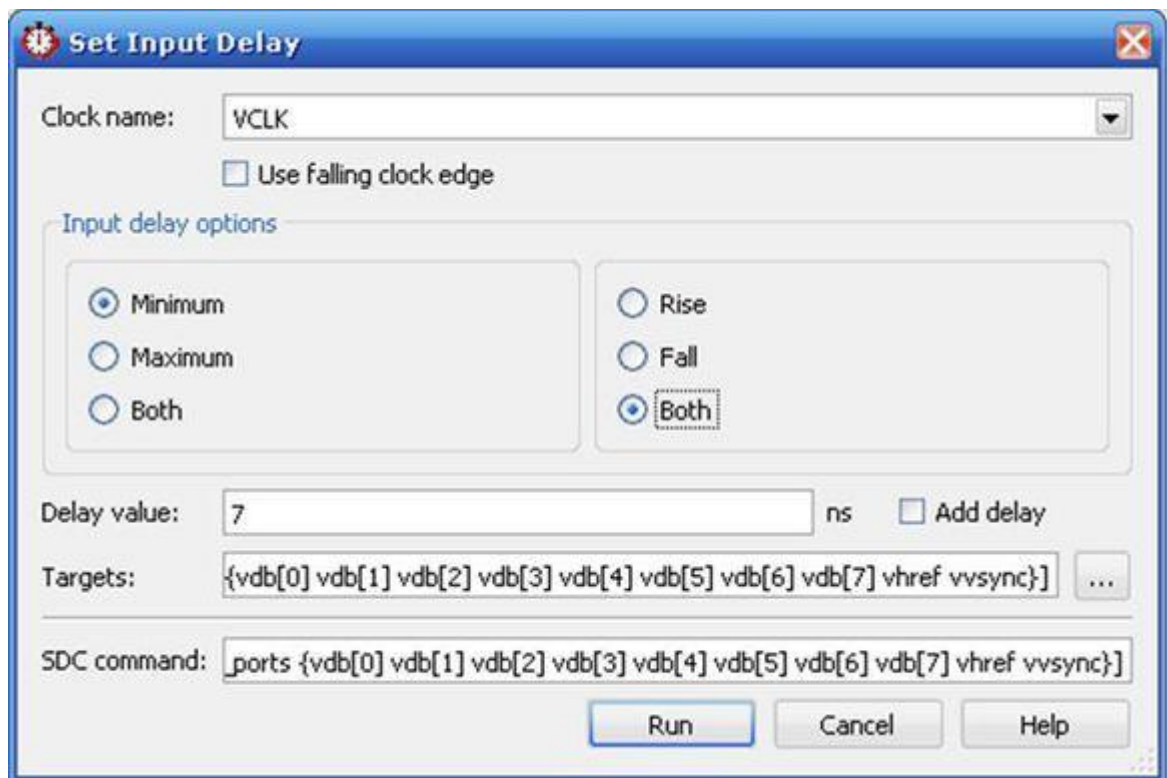
下面我们来添加时序约束，打开 TimeQuest，点击菜单栏的 Constraints→Creat Clock，做如下设置。



点击 Constraints→Set Maximum Delay，对 vdb[0] vdb[1] vdb[2] vdb[3] vdb[4] vdb[5] vdb[6] vdb[7] vhref 的 set\_max\_delay 做如下设置。



点击 Constraints\Set Minimum Delay, 对 vdb[0] vdb[1] vdb[2] vdb[3] vdb[4] vdb[5] vdb[6] vdb[7] vhref 的 set\_min\_delay 做如下设置。



约束完成后，参照前面章节 Update Timing Netlist 并且 Write SDC File...，接着就可以重新编译整个工程，再来看看这个时序分析报告。在报告中，数据的建立时间有 9-13ns 的余量，而保持时间也都有 7-11ns 的余量，可谓余量充足。

Inputs to Registers (Setup)					
Slack	From Node	To Node	Launch Clock	Latch Clock	
17 9.192	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[8]	VCLK	VCLK	
18 9.192	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[7]	VCLK	VCLK	
19 9.194	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[6]	VCLK	VCLK	
20 9.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a2	VCLK	VCLK	
21 9.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a0	VCLK	VCLK	
22 9.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a1	VCLK	VCLK	
23 9.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...to_generated[a_graycounter_fic:wrptr_glp]parity8	VCLK	VCLK	
24 9.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[0]	VCLK	VCLK	
25 9.614	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[4]	VCLK	VCLK	
26 9.769	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[5]	VCLK	VCLK	
27 10.022	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[1]	VCLK	VCLK	
28 10.218	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[5]	VCLK	VCLK	
29 10.218	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[1]	VCLK	VCLK	
30 10.218	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...auto_generated[cntr_s2e:cntr_b]counter_reg_bit[0]	VCLK	VCLK	
31 10.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[7]	VCLK	VCLK	
32 10.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[5]	VCLK	VCLK	
33 10.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[6]	VCLK	VCLK	
34 10.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[0]	VCLK	VCLK	
35 10.278	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[2]	VCLK	VCLK	
36 10.351	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[9]	VCLK	VCLK	
37 10.351	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[8]	VCLK	VCLK	
38 10.351	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[3]	VCLK	VCLK	
39 10.360	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[2]	VCLK	VCLK	
40 10.373	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc..._Sh31:ffio_ram[ram_block11a0-porta_address_reg0]	VCLK	VCLK	
41 10.373	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...yncram_Sh31:ffio_ram[ram_block11a0-porta_we_reg]	VCLK	VCLK	
42 10.375	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
43 10.615	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[3]	VCLK	VCLK	
44 11.087	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...yncram_Sh31:ffio_ram[ram_block11a0-porta_we_reg]	VCLK	VCLK	
45 11.874	vdb[1]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
46 12.012	vdb[0]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
47 12.191	vdb[3]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
48 12.319	vdb[2]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
49 12.369	vdb[5]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
50 12.393	vdb[7]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
51 12.440	vdb[6]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
52 12.566	vdb[4]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	

Inputs to Registers (Hold)					
Slack	From Node	To Node	Launch Clock	Latch Clock	
17 7.760	vdb[4]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
18 7.885	vdb[6]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
19 7.928	vdb[7]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
20 7.949	vdb[5]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
21 8.046	vdb[2]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
22 8.166	vdb[3]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
23 8.259	vdb[0]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
24 8.345	vdb[1]	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
25 9.154	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...yncram_Sh31:ffio_ram[ram_block11a0-porta_we_reg]	VCLK	VCLK	
26 9.523	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[3]	VCLK	VCLK	
27 9.685	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc..._m_Sh31:ffio_ram[ram_block11a0-porta_datain_reg0]	VCLK	VCLK	
28 9.686	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc..._Sh31:ffio_ram[ram_block11a0-porta_address_reg0]	VCLK	VCLK	
29 9.686	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...yncram_Sh31:ffio_ram[ram_block11a0-porta_we_reg]	VCLK	VCLK	
30 9.821	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[2]	VCLK	VCLK	
31 9.997	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[9]	VCLK	VCLK	
32 9.997	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[8]	VCLK	VCLK	
33 9.997	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[3]	VCLK	VCLK	
34 10.023	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[1]	VCLK	VCLK	
35 10.063	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[7]	VCLK	VCLK	
36 10.063	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[5]	VCLK	VCLK	
37 10.063	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[6]	VCLK	VCLK	
38 10.063	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[0]	VCLK	VCLK	
39 10.063	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[2]	VCLK	VCLK	
40 10.116	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[4]	VCLK	VCLK	
41 10.116	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...component[dcfifo_jd1:auto_generated]wrptr_g[1]	VCLK	VCLK	
42 10.116	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...auto_generated[cntr_s2e:cntr_b]counter_reg_bit[0]	VCLK	VCLK	
43 10.433	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[4]	VCLK	VCLK	
44 10.457	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[5]	VCLK	VCLK	
45 10.847	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[6]	VCLK	VCLK	
46 10.848	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[7]	VCLK	VCLK	
47 10.850	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[8]	VCLK	VCLK	
48 10.993	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a2	VCLK	VCLK	
49 10.993	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a0	VCLK	VCLK	
50 10.993	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]sub_parity9a1	VCLK	VCLK	
51 10.993	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...to_generated[a_graycounter_fic:wrptr_glp]parity8	VCLK	VCLK	
52 10.993	vhref	video_input:uut_videoinput/video_ctrl:uut_videooc...erated[a_graycounter_fic:wrptr_glp]counter10a[0]	VCLK	VCLK	

另外，我们也可以专门找一条路径出来，看看它的具体时序路径的分析。vd[0] 这条数据线的建立时间报告中，66ns 的 input max delay 出现在了 Data Arrival Path 中。

Path #1: Setup slack is 12.012							
Path Summary		Statistics	Data Path	Waveform	Extra Filter Information		
Data Arrival Path							
	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	0.000	0.000					clock path
1	0.000	0.000	R				clock network delay
3	66.000	66.000	F	iExt	1	PIN_127	vdb[0]
4	71.283	5.283					data path
1	66.000	0.000	FF	IC	1	IOIBUF_X16_Y24_N8	vdb[0]~input i
2	66.927	0.927	FF	CELL	1	IOIBUF_X16_Y24_N8	vdb[0]~input o
3	71.196	4.269	FF	IC	1	M9K_X27_Y20_N0	uut_videoinput uut_videoctrl uut_videofifo d
4	71.283	0.087	FF	CELL	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic
Data Required Path							
	Total	Incr	RF	Type	Fanout	Location	Element
1	80.000	80.000					latch edge time
2	83.227	3.227					clock path
1	83.227	3.227	R				clock network delay
3	83.295	0.068		uTsu	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

而在 vd[0] 的保持时间报告中，7ns 的 input min delay 则出现在了 Data Arrival Path 中。

Path #1: Hold slack is 8.259							
Path Summary		Statistics	Data Path	Waveform	Extra Filter Information		
Data Arrival Path							
	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	0.000	0.000					clock path
1	0.000	0.000	R				clock network delay
3	7.000	7.000	R	iExt	1	PIN_127	vdb[0]
4	11.850	4.850					data path
1	7.000	0.000	RR	IC	1	IOIBUF_X16_Y24_N8	vdb[0]~input i
2	7.877	0.877	RR	CELL	1	IOIBUF_X16_Y24_N8	vdb[0]~input o
3	11.772	3.895	RR	IC	1	M9K_X27_Y20_N0	uut_videoinput uut_videoctrl uut_videofifo d
4	11.850	0.078	RR	CELL	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic
<div><div></div><div>III</div><div></div></div>							
Data Required Path							
	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					latch edge time
2	3.337	3.337					clock path
1	3.337	3.337	R				clock network delay
3	3.591	0.254		uTh	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

10、FPGA 的时序基础理论

我们的分析从下图开始，下图是常用的静态分析结构图，一开始看不懂公式不要紧，因为我会在后面给以非常简单的解释：

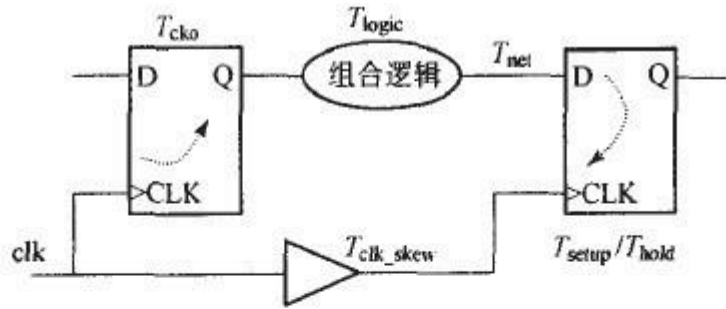


图3 寄存器的建立/保持时间分析

<http://blog.csdn.net/stephenkung1>

寄存器的建立时间余量  $T_{\text{slack, setup}}$  为:

$$T_{\text{slack, setup}} = T_{\text{period}} - (T_{\text{cko}} + T_{\text{logic}} + T_{\text{net}} + T_{\text{setup}} - T_{\text{clk\_skew}}) \quad (1)$$

寄存器的保持时间余量  $T_{\text{hold}}$  为:

$$T_{\text{hold}} = T_{\text{cko}} + T_{\text{logic}} + T_{\text{net}} - T_{\text{hold}} - T_{\text{clk\_skew}} \quad (2)$$

这两个公式是一个非常全面的，准确的关于建立时间和保持时间的公式。其中  $T_{\text{period}}$  为时钟周期;  $T_{\text{cko}}$  为 D 触发器开始采样瞬间到 D 触发器采样的数据开始输出的时间;  $T_{\text{logic}}$  为中间的组合逻辑的延时;  $T_{\text{net}}$  为走线的延时;  $T_{\text{setup}}$  为 D 触发器的建立时间;  $T_{\text{clk\_skew}}$  为时钟偏移，偏移的原因是因为时钟到达前后两个 D 触发器的路线不是一样长。

这里我们来做如下转化:

因为对于有意义的时序约束, 建立时间余量  $T_{\text{slack, setup}}$  和保持时间余量  $T_{\text{hold}}$  都要大于 0 才行, 所以对于时序约束的要求其实等价于:

$$T_{\text{period}} > T_{\text{cko}} + T_{\text{logic}} + T_{\text{net}} + T_{\text{setup}} - T_{\text{clk\_skew}} \quad (1)$$

$$T_{\text{cko}} + T_{\text{logic}} + T_{\text{net}} > T_{\text{hold}} + T_{\text{clk\_skew}} \quad (2)$$

之前说了, 这两个公式是最全面的, 而实际上, 大部分教材没讲这么深, 他们对于一些不那么重要的延时没有考虑, 所以就导致不同的教材说法不一。这里, 为了得到更加简单的理解, 我们按照常规, 忽略两项  $T_{\text{net}}$  和  $T_{\text{clk\_skew}}$ 。原因在于  $T_{\text{net}}$  通常太小, 而  $T_{\text{clk\_skew}}$  比较不那么初级。简化后如下:

$$T_{\text{period}} > T_{\text{cko}} + T_{\text{logic}} + T_{\text{setup}} \quad (3)$$

$$T_{\text{cko}} + T_{\text{logic}} > T_{\text{hold}} \quad (4)$$

简单多了吧！但是你能看出这两个公式的含义吗？其实（3）式比较好理解，意思是数据从第一个触发器采样时刻传到第二个触发器采样时刻，不能超过一个时钟周期啊！假如数据传输超过一个时钟周期，那么就会导致第二个触发器开始采样的时候，想要的数据还没有传过来呢！那么（4）式又如何理解呢？老实说，一般人一眼看不出来。

我们对于（4）式两边同时加上  $T_{setup}$ ，得到（5）：

$$T_{cko} + T_{logic} + T_{setup} > T_{hold} + T_{setup} \quad (5)$$

结合（3）式和（5）式，我们得到如下的式子：

$$T_{hold} + T_{setup} < T_{cko} + T_{logic} + T_{setup} < T_{period} \quad (6)$$

这个式子就是那个可以让我们看出规律的式子。也是可以看出静态时序分析本质的式子。

$T_{cko} + T_{logic} + T_{setup}$  是指数据从第一级触发器采样瞬间开始，传输到第二级触发器并被采样的传输延时。我们简称为数据传输延时。下面讲述（6）式两端的含义。

$T_{cko} + T_{logic} + T_{setup} < T_{period}$ ：约定数据传输延时不能太大，如果太大（超过一个时钟周期），那么第二级触发器就会在采样的时刻发现数据还没有到来。

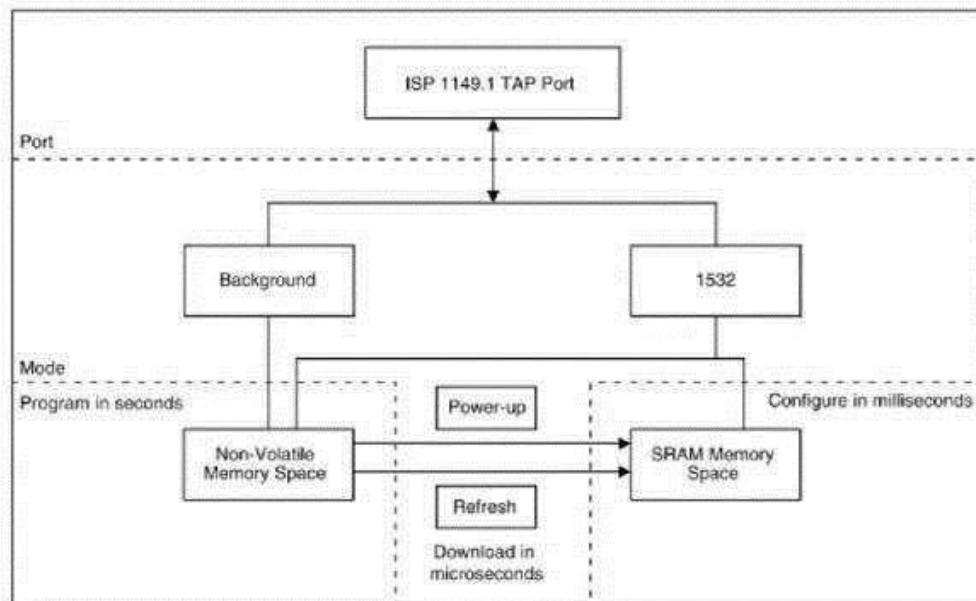
$T_{hold} + T_{setup} < T_{cko} + T_{logic} + T_{setup}$ ：约定数据传输延时不能太小。这就奇怪了，数据传得太慢大家都知道不好，难道传得太快也不行吗？是的，不行！

$T_{hold} + T_{setup}$  是一个触发器的采样窗口时间，我们知道，D 触发器并不是绝对的瞬间采样，它不可能那么理想。在 D 触发器采样的瞬间，在这瞬间之前  $T_{setup}$  时间之内，或者这瞬间之后  $T_{hold}$  时间之内，如果输入端口发生变化，那么 D 触发器就会处于亚稳态。所以采样是有窗口的，我们把  $T_{hold} + T_{setup}$  的时间宽度叫做触发器的采样窗口，在窗口期内，D 触发器是脆弱的，对毛刺没有免疫力的。假如数据传输延时特别小，那么就会发现，当第二级触发器开始采样的时候，第一级触发器的窗口期还没有结束！也就是说，如果这个时候输入端数据有变化，那么不仅第一级触发器处于亚稳态，第二级触发器也将处于亚稳态！

综上，我们就可以知道，数据传输延时既不能太大以至于超过一个时钟周期，也不能太小以至于小于触发器采样窗口的宽度。这就是静态时序分析的终极内涵。有了这个，就不需要再记任何公式了。

## 11、CPLD、FPGA 加载原理

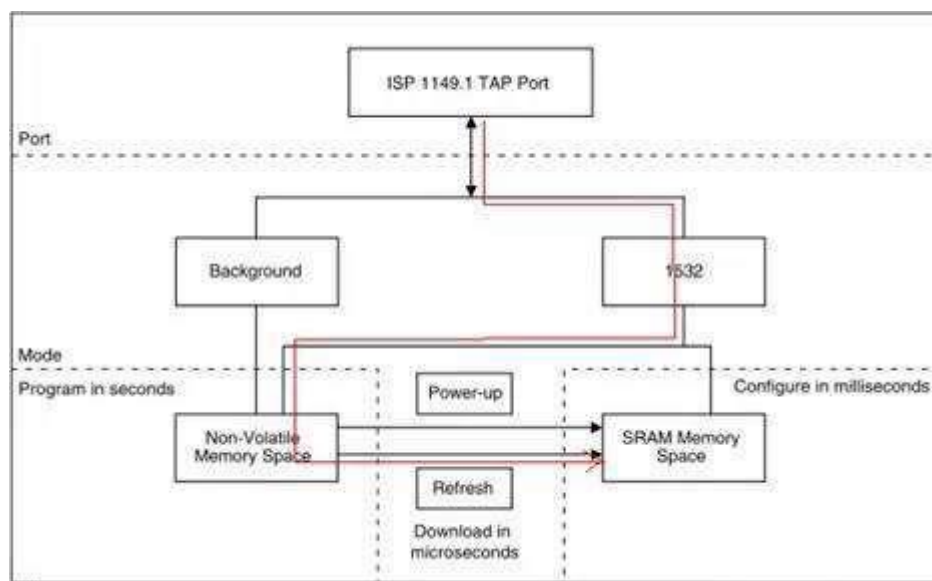
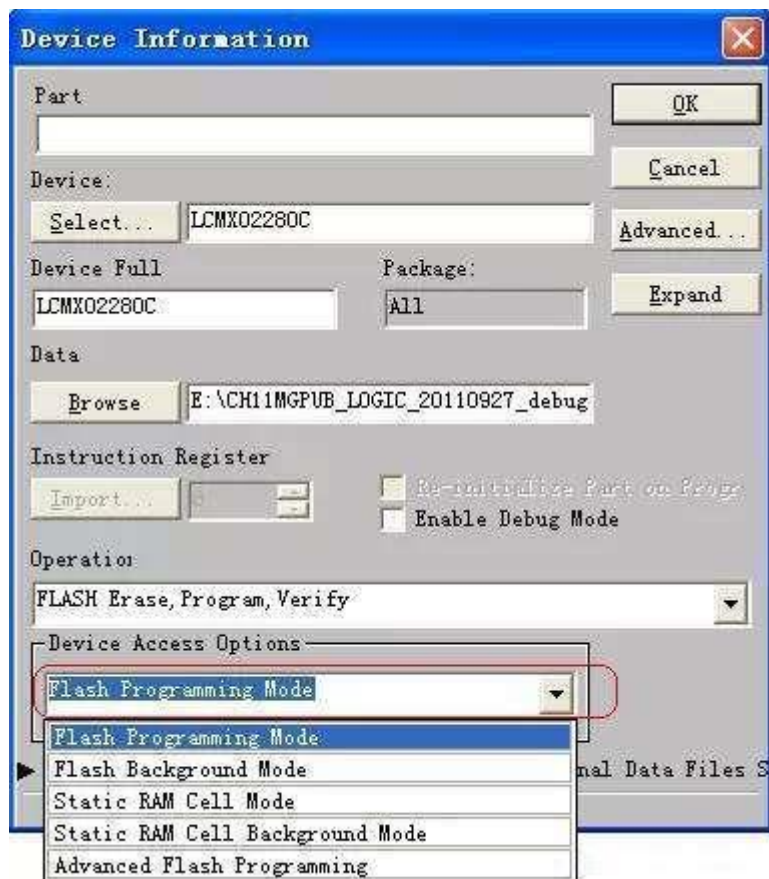
LD 一般用 JTAG 接口进行加载，内部有 FLASH 和 SRAM，CPLD 的配置文件可存在内置的 FLASH 中，因此下电不会丢失，不需要每次上电的时候，额外对 CPLD 进行配置结构如下：



方式一：当 SRAM 为空时（CPLD 一次都未加载过或者 CPLD 内部 FLASH 存储的配置文件有问题，不能加载到 SRAM 中），Flash 编程进入直接模式，此时 CPLD 的 IO 管脚状态由 BSCAN registers（边界扫描寄存器）决定，BSCAN registers 可以将 IO 设置成 high, low, tristate (default), or current value 四种。

方式 2:

方式二：当 SRAM 不为空的时候，Flash 可进行 background 编程模式. 在此模式下，在加载 on-chipFlash 时，允许 CPLD 器件仍然维持在用户操作模式下（即 CPLD 可以正常工作）。



IEEE 1532 标准简介 IEEE 1532 标准是一个基于 IEEE 1149.1 的在板编程的新标准，标准的名字为 IEEE Standard for In-System Configuration of Programmable Devices。在 1993 年，出现 ISP（In-System Programming）的概念和应用。随之产生了应用 IEEE1149.1 进行 ISP 的需求。各个厂商提供了类似的不相同的基于 JTAG 的 ISP 工具。1996 年 4 月，半导体厂商、ISP 工具开发者、ATE 开发商正式提出了 IEEE 1532 标准，旨在为 JTAG 器件的在板编程提供一系列标准的专门的寄存器和操作指令从而使得在板编程更为容易和高效。IEEE1532

完全建立在 IEEE1149.1 标准之上，在 IEEE 1532 标准上可以开发通用的编程工具，为测试、编程和系统开发提供规范的接口和器件支持、促进了编程革新，开辟了边界扫描技术新的应用领域。IEEE1532 主要应用在 CPLD、FPGA、PROM 以及任意的支持 IEEE 1532 的可编程器件的在板编程。

早期的可编程逻辑器件只有可编程只读存储器 (PROM)、紫外线可擦除只读存储器 (EPROM) 和电可擦除只读存储器 (EEPROM) 三种。由于结构的限制，它们只能完成简单的数字逻辑功能。

其后，出现了一类结构上稍复杂的可编程芯片，即可编程逻辑器件 (PLD)，它能够完成各种数字逻辑功能。典型的 PLD 由一个“与”门和一个“或”门阵列组成，而任意一个组合逻辑都可以用“与-或”表达式来描述，所以，PLD 能以乘积和的形式完成大量的组合逻辑功能，可以实现速度特性较好的逻辑功能，但其过于简单的结构也使它们只能实现规模较小的电路。

为了弥补这一缺陷，20 世纪 80 年代中期。Altera 和 Xilinx 分别推出了类似于 PAL(可编程阵列逻辑)结构的扩展型 CPLD(Complex Programmable Logic Device) 和与标准门阵列类似的 FPGA(Field Programmable Gate Array)，它们都具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。这两种器件兼容了 PLD 和通用门阵列 GAL(Generic Array Logic)的优点，可实现较大规模的电路，编程也很灵活。与门阵列等其它 ASIC(Application Specific IC)相比，它们又具有设计开发周期短、设计制造成本低、开发工具先进、标准产品无需测试、质量稳定以及可实时在线检验等优点，因此被广泛应用于产品的原型设计和产品生产(一般在 10,000 件以下)之中。几乎所有应用门阵列、PLD 和中小规模通用数字集成电路的场合均可应用 FPGA 和 CPLD 器件。

## 12、锁存器、触发器、寄存器和缓冲器的区别

### 一、锁存器

锁存器 (latch) ——对脉冲电平敏感，在时钟脉冲的电平作用下改变状态

锁存器是电平触发的存储单元，数据存储的动作取决于输入时钟（或者使能）信号的电平值，仅当锁存器处于使能状态时，输出才会随着数据输入发生变化。

锁存器不同于触发器，它不在锁存数据时，输出端的信号随输入信号变化，就像信号通过一个缓冲器一样；一旦锁存信号起锁存作用，则数据被锁住，输入信号不起作用。锁存器也称为透明锁存器，指的是不锁存时输出对于输入是透明的。

锁存器 (latch)：我听过的最多的就是它是电平触发的，呵呵。锁存器是电平触发的存储单元，数据存储的动作取决于输入时钟（或者使能）信号的电平值，当锁存器处于使能状态时，输出才会随着数据输入发生变化。（简单地说，它有两个输入，分别是一个有效信号 EN, 一个输入数据信号 DATA\_IN，它有一个输出 Q，它的功能就是在 EN 有效的时候把 DATA\_IN 的值传给 Q，也就是锁存的过程）。

应用场合：数据有效迟后于时钟信号有效。这意味着时钟信号先到，数据信号后到。在某些运算器电路中有时采用锁存器作为数据暂存器。

缺点：时序分析较困难。

不要锁存器的原因有二：1、锁存器容易产生毛刺，2、锁存器在 ASIC 设计中应该说比 ff 要简单，但是在 FPGA 的资源中，大部分器件没有锁存器这个东西，所以需要用一个逻辑门和 ff 来组成锁存器，这样就浪费了资源。

优点：面积小。锁存器比 FF 快，所以用在地址锁存是很合适的，不过一定要保证所有的 latch 信号源的质量，锁存器在 CPU 设计中很常见，正是由于它的应用使得 CPU 的速度比外部 IO 部件逻辑快许多。latch 完成同一个功能所需要的门较触发器要少，所以在 asic 中用的较多。

## 二、触发器

触发器 (Flip-Flop, 简称为 FF)，也叫双稳态门，又称双稳态触发器。是一种可以在两种状态下运行的数字逻辑电路。触发器一直保持它们的状态，直到它们收到输入脉冲，又称为触发。当收到输入脉冲时，触发器输出就会根据规则改变状态，然后保持这种状态直到收到另一个触发。

触发器 (flip-flops) 电路相互关联，从而为使用内存芯片和微处理器的数字集成电路 (IC) 形成逻辑门。它们可用来存储一比特的数据。该数据可表示音序器的状态、计数器的价值、在计算机内存的 ASCII 字符或任何其他的信息。

有几种不同类型的触发器 (flip-flops) 电路具有指示器，如 T (切换)、S-R (设置/重置) J-K (也可能称为 Jack Kilby) 和 D (延迟)。典型的触发器包括零个、一个或两个输入信号，以及时钟信号和输出信号。一些触发器还包括一个重置当前输出的明确输入信号。第一个电子触发器是在 1919 年由 W. H. Eccles 和 F. W. Jordan 发明的。

触发器 (flip-flop) 对脉冲边沿敏感，其状态只在时钟脉冲的上升沿或下降沿的瞬间改变。

T 触发器 (Toggle Flip-Flop, or Trigger Flip-Flop) 设有一个输入和输出，当时钟频率由 0 转为 1 时，如果 T 和 Q 不相同，其输出值会是 1。输入端 T 为 1 的时候，输出端的状态 Q 发生反转；输入端 T 为 0 的时候，输出端的状态 Q 保持不变。把 JK 触发器的 J 和 K 输入点连接在一起，即构成一个 T 触发器。

应用场合：时钟有效迟后于数据有效。这意味着数据信号先建立，时钟信号后建立。在 CP 上升沿时刻打入到寄存器。

## 三、寄存器

寄存器 (register)：用来存放数据的一些小型存储区域，用来暂时存放参与运算的数据和运算结果，它被广泛的用于各类数字系统和计算机中。其实寄存器就是一种常用的时序逻辑电路，但这种时序逻辑电路只包含存储电路。寄存器的存储电路是由锁存器或触发器构成的，因为一个锁存器或触发器能存储 1 位二进制数，所以由 N 个锁存器或触发器可以构成 N 位寄存器。工程中的寄存器一般按计算机中字节的位数设计，所以一般有 8 位寄存器、16 位寄存器等。

对寄存器中的触发器只要求它们具有置 1、置 0 的功能即可，因而无论是用同步 RS 结构触发器，还是用主从结构或边沿触发结构的触发器，都可以组成寄存器。一般由 D 触发器组成，有公共输入/输出使能控制端和时钟，一般把使能控制端作为寄存器电路的选择信号，把时钟控制端作为数据输入控制信号。

### 寄存器的应用

1. 可以完成数据的并串、串并转换；
2. 可以用做显示数据锁存器：许多设备需要显示计数器的记数值，以 8421BCD 码记数，以七段显示器显示，如果记数速度较高，人眼则无法辨认迅速变化的显示字符。在计数器和译码器之间加入一个锁存器，控制数据的显示时间是常用的方法。
3. 用作缓冲器；
  1. 组成计数器：移位寄存器可以组成移位型计数器，如环形或扭环形计数器。

### 四、移位寄存器

移位寄存器：具有移位功能的寄存器称为移位寄存器。

寄存器只有寄存数据或代码的功能。有时为了处理数据，需要将寄存器中的各位数据在移位控制信号作用下，依次向高位或向低位移动 1 位。移位寄存器按数码移动方向分类有左移，右移，可控制双向（可逆）移位寄存器；按数据输入端、输出方式分类有串行和并行之分。除了 D 边沿触发器构成移位寄存器外，还可以用诸如 JK 等触发器构成移位寄存器。

### 五、总线收发器/缓冲器

缓冲寄存器：又称缓冲器缓冲器(buffer)：多用在总线上，提高驱动能力、隔离前后级，缓冲器多半有三态输出功能。当负载不具有非选通输出为高阻特性时，将起到隔离作用；当总线的驱动能力不够驱动负载时，将起到驱动作用。由于缓冲器接在数据总线上，故必须具有三态输出功能。

它分输入缓冲器和输出缓冲器两种。前者的作用是将外设送来的数据暂时存放，以便处理器将它取走；后者的作用是用来暂时存放处理器送往外设的数据。有了

数控缓冲器，就可以使高速工作的 CPU 与慢速工作的外设起协调和缓冲作用，实现数据传送的同步。

Buffer:缓冲区，一个用于在初速度不同步的设备或者优先级不同的设备之间传输数据的区域。通过缓冲区，可以使进程之间的相互等待变少，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。

缓冲器主要是计算机领域的称呼。具体实现上，缓冲器有用锁存器结构的电路来实现，也有用不带锁存结构的电路来实现。一般来说，当收发数据双方的工作速度匹配时，这里的缓冲器可以用不带锁存结构的电路来实现；而当收发数据双方的工作速度不匹配时，就要用带锁存结构的电路来实现了（否则会出现数据丢失）。

缓冲器在数字系统中用途很多：

- （1）如果器件带负载能力有限，可加一级带驱动器的缓冲器；
- （2）前后级间逻辑电平不同，可用电平转换器加以匹配；
- （3）逻辑极性不同或需要将单性变量转换为互补变量时，加带反相缓冲器；（4）需要将缓变信号变为边沿陡峭信号时，加带施密特电路的缓冲器
- （5）数据传输和处理中不同装置间温度和时间不同时，加一级缓冲器进行弥补等等。

## 六、锁存器与触发器的区别

锁存器和触发器是具有记忆功能的二进制存贮器件，是组成各种时序逻辑电路的基本器件之一。区别为：latch 同其所有的输入信号相关，当输入信号变化时 latch 就变化，没有时钟端；flip-flop 受时钟控制，只有在时钟触发时才采样当前的输入，产生输出。当然因为 latch 和 flip-flop 二者都是时序逻辑，所以输出不但同当前的输入相关还同上一时间的输出相关。

1、latch 由电平触发，非同步控制。在使能信号有效时 latch 相当于通路，在使能信号无效时 latch 保持输出状态。DFF 由时钟沿触发，同步控制。

2、latch 对输入电平敏感，受布线延迟影响较大，很难保证输出没有毛刺产生；DFF 则不易产生毛刺。

3、如果使用门电路来搭建 latch 和 DFF，则 latch 消耗的门资源比 DFF 要少，这是 latch 比 DFF 优越的地方。所以，在 ASIC 中使用 latch 的集成度比 DFF 高，但在 FPGA 中正好相反，因为 FPGA 中没有标准的 latch 单元，但有 DFF 单元，一个 LATCH 需要多个 LE 才能实现。latch 是电平触发，相当于有一个使能端，且在激活之后（在使能电平的时候）相当于导线了，随输出而变化。在非使能状态下是保持原来的信号，这就可以看出和 flip-flop 的差别，其实很多时候 latch 是不能代替 ff 的。

4、latch 将静态时序分析变得极为复杂。

5、目前 latch 只在极高端的电路中使用,如 intel 的 P4 等 CPU。FPGA 中有 latch 单元,寄存器单元就可以配置成 latch 单元,在 xilinx v2p 的手册将该单元配置成为 register/latch 单元,附件是 xilinx 半个 slice 的结构图。其它型号和厂家的 FPGA 没有去查证。——一个人认为 xilinx 是能直接配的而 altera 或许比较麻烦,要几个 LE 才行,然而也非 xilinx 的器件每个 slice 都可以这样配置,altera 的只有 DDR 接口中有专门的 latch 单元,一般也只有高速电路中会采用 latch 的设计。altera 的 LE 是没有 latch 的结构,又查了 sp3 和 sp2e,别的不查了,手册上说支持这种配置。有关 altera 的表述 wangdian 说的对,altera 的 ff 不能配置成 latch,它使用查找表来实现 latch。

一般的设计规则是:在绝大多数设计中避免产生 latch。它会让您设计的时序完蛋,并且它的隐蔽性很强,非老手不能查出。latch 最大的危害在于不能过滤毛刺。这对于下一级电路是极其危险的。所以,只要能用 D 触发器的地方,就不用 latch。

有些地方没有时钟,也只能用 latch 了。比如现在用一个 clk 接到 latch 的使能端(假设是高电平使能),这样需要的 setup 时间,就是数据在时钟的下降沿之前需要的时间,但是如果是一个 DFF,那么 setup 时间就是在时钟的上升沿需要的时间。这就说明如果数据晚于控制信号的情况下,只能用 latch,这种情况就是,前面所提到的 latch timing borrow。基本上相当于借了一个高电平时间。也就是说,latch 借的时间也是有限的。

在 if 语句和 case 不全很容易产生 latch,需要注意。VIA 题目这两个代码哪个综合更容易产生 latch:

代码 1

```
always@(enable or ina or inb)
```

```
if(enable) begin
```

```
    data_out = ina;
```

```
end
```

```
else begin
```

```
    data_out = inb;
```

```
end
```

代码 2

```

input[3:0] data_in;

always@(data_in)

case(data_in)

    0 :          out1 = 1'b1;

    1, 3 :       out2 = 1'b1;

    2, 4, 5, 6, 7 : out3 = 1'b1;

    default:     out4 = 1'b1;

endcase

```

答案是代码 2 在综合时更容易产生 latch。

对 latch 进行 STA 的分析其实也是可以,但是要对工具相当熟悉才行,不过很容易出错。当前 PrimeTime 是支持进行 latch 分析的,现在一些综合工具内置的 STA 分析功能也支持,比如 RTL compiler, Design Compiler。除了 ASIC 里可以节省资源以外, latch 在同步设计里出现的可能还是挺小的,现在处理过程中大都放在 ff 里打一下。

锁存器电平触发会把输入端的毛刺带入输出;而触发器由于边沿作用可以有效抑制输入端干扰。

在 CMOS 芯片内部经常使用锁存器,但是在 PCB 板级结构上,建议用触发器在时钟边沿上锁存数据。这是因为在锁存器闸门开启期间数据的变化会直接反映到输出端,所以要注意控制闸门信号的脉冲宽度,而对于触发器,只考虑时钟的边沿。

门电路是构建组合逻辑电路的基础,而锁存器和触发器是构建时序逻辑电路的基础。门电路是由晶体管构成的,锁存器是由门电路构成的,而触发器是由锁存器构成的。也就是晶体管->门电路->锁存器->触发器,前一级是后一级的基础。锁存器和触发器它们的输出都不仅仅取决于目前的输入,而且和之前的输入和输出都有关系。

它们之间的不同在于:锁存器没有时钟信号,而触发器常常有时钟触发信号。

锁存器是异步的,就是说在输入信号改变后,输出信号也随之很快做出改变非常快。而另外一方面,今天许多计算机是同步的,这就意味着所有的时序电路的输出信号随着全局的时钟信号同时做出改变。触发器是一个同步版锁存器。

触发器泛指一类电路结构，它可以由触发信号（如：时钟、置位、复位等）改变输出状态，并保持这个状态直到下一个或另一个触发信号来到时。触发信号可以用电平或边沿操作，锁存器是触发器的一种应用类型。

## 七、D 触发器和 D 锁存器的区别

钟控 D 触发器其实就是 D 锁存器，边沿 D 触发器才是真正的 D 触发器，钟控 D 触发器在使能情况下输出随输入变化，边沿触发器只有在边沿跳变的情况下输出才变化。

两个锁存器可以构成一个触发器，归根到底还是 dff 是边沿触发的，而 latch 是电平触发的。锁存器的输出对输入透明的，输入是什么，输出就是什么，这就是锁存器不稳定的原因，而触发器是由两个锁存器构成的一个主从触发器，输出对输入是不透明的，必须在时钟的上升/下降沿才会将输入体现到输出，所以能够消除输入的毛刺信号。

## 八、寄存器与锁存器的区别

寄存器与锁存器的功能是提供数据寄存和锁存。

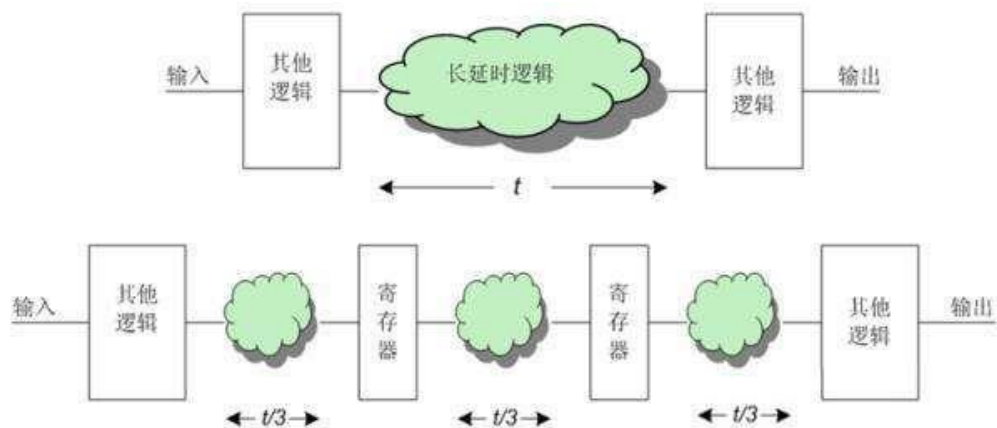
寄存功能是指把数据暂时保存，需要时取出。锁存功能是指总线电路中，锁定数据输出，使输出端不随输入端变化。

## 13、流水线

流水线设计是经常用于提高所设计系统运行速度的一种有效的方法。

为了保障数据的快速传输，必须使系统运行在尽可能高的频率上，但如果某些复杂逻辑功能的完成需要较长的延时，就会使系统难以运行在高的频率上，在这种情况下，可使用流水线技术，即在长延时的逻辑功能块中插入触发器，使复杂的逻辑操作分步完成，减小每个部分的延时，从而使系统的运行频率得以提高。流水线设计的代价是增加了寄存器逻辑，增加了芯片资源的耗用。

如某个复杂逻辑功能的实现需较长的延时，可将其分解为几个（如 3 个）步骤来实现，每一步的延时变小，在各步间加入寄存器，以暂存中间结果，这样可大大提高整个系统的最高工作频率。



设计综合到不同器件的最高工作频率

器件	非流水线设计 (MHz)	2级流水线设计 (MHz)	4级流水线设计 (MHz)
EPIK10TC100-3 (FPGA)	138.89	158.73	166.67
EPM3064ALC44-4 (CPLD)	105.26	149.25	158.73

流水线设计的关键在于整个设计时序的合理安排，前后级接口间数据流速的匹配。如果前后级流量相等，前级输出直接可作为后级输入，前级流量大于后级时，则需要增加缓存，前级流量小于后级时，则需要通过逻辑复制，串并转换等方式将数据分流，或者前级存储后输出到后级进行处理。

流水线的思想：复制了处理模块，用面积换速度。

来源：硬件十万个为什么