

FPGA 入门教程

1. 数字电路设计入门
2. **FPGA 简介**
3. **FPGA 开发流程**
4. **RTL 设计**
5. **Quartus II 设计实例**
6. **ModelSim 和 Testbench**

1.数字电路设计入门

1.1 数字电路设计

数字电路设计的核心是逻辑设计。通常，数字电路的逻辑值只有‘1’和‘0’，表征的是模拟电压或电流的离散值，一般‘1’代表高电平，‘0’代表低电平。

高低电平的含义可以理解为，存在一个判决电平，当信号的电压值高于判决电平时，我们就认为该信号表征高电平，即为‘1’。反之亦然。

当前的数字电路中存在许多种电平标准，比较常见的有 TTL、CMOS、LVTTTL、LVCMOS、ECL、PECL、LVDS、HSTL、SSTL 等。这些电平的详细指标请见《补充教程 1：电平标准》。

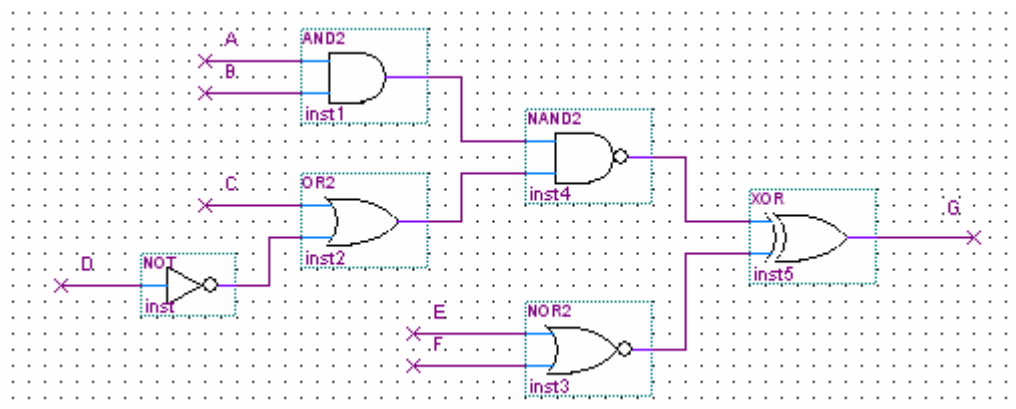
数字电路设计大致可分为组合逻辑电路和时序逻辑电路。

一般的数字设计的教材中对组合逻辑电路和时序逻辑电路的定义分别为：组合逻辑电路的输出仅与当前的输入有关，而时序逻辑电路的输出不但与输入有关，还和系统上一个状态有关。

但是在设计中，我们一般以时钟的存在与否来区分该电路的性质。由时钟沿驱动工作的电路为时序逻辑电路。大家注意，这两种电路并不是独立存在的，他们相互交错存在于整个电路系统的设计中。

1.1.1 组合逻辑电路

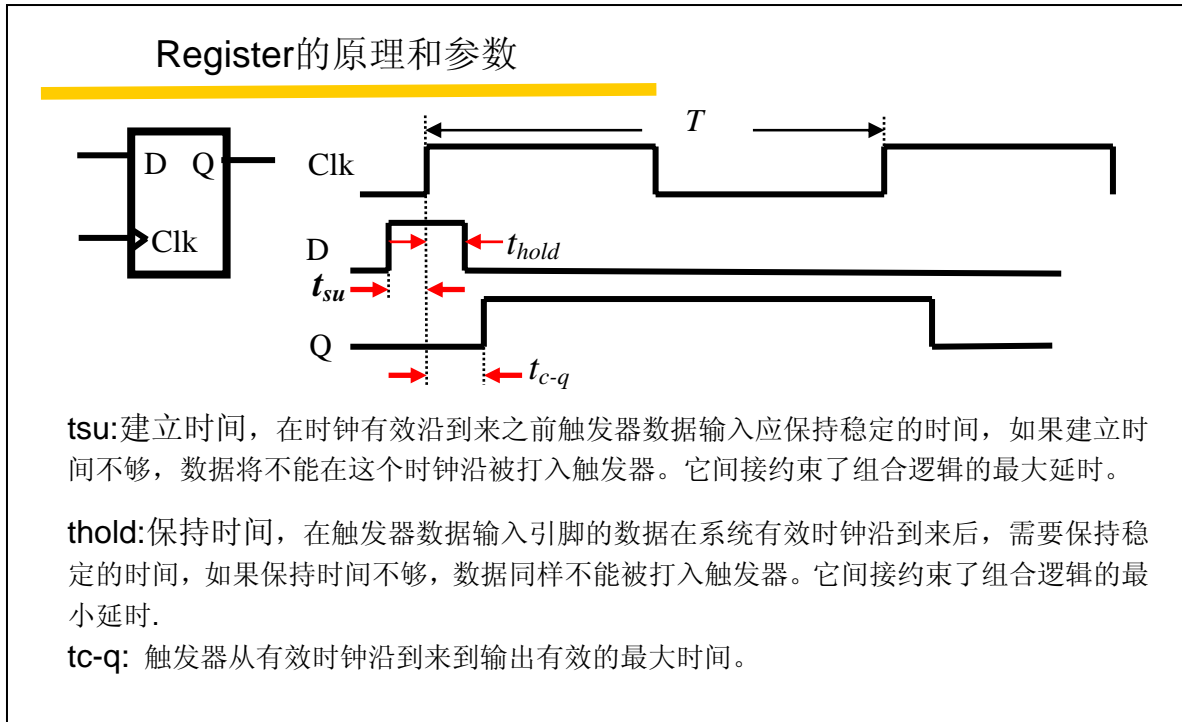
组合逻辑电路由任意数目的逻辑门电路组成，一般包括与门、或门、非门、异或门、与非门、或非门等。一般的组合逻辑电路如下图：



其中 A,B,C,D,E,F 为输入，G 为输出。

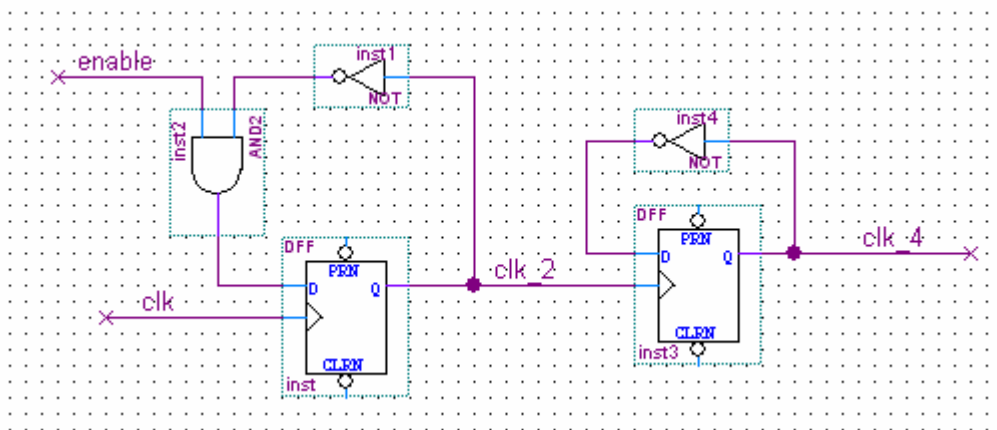
1.1.2 时序逻辑电路

时序逻辑电路由时钟的上升沿或下降沿驱动工作，其实真正被时钟沿驱动的是电路中的触发器（Register），也称为寄存器。触发器的工作原理和参数如下图：

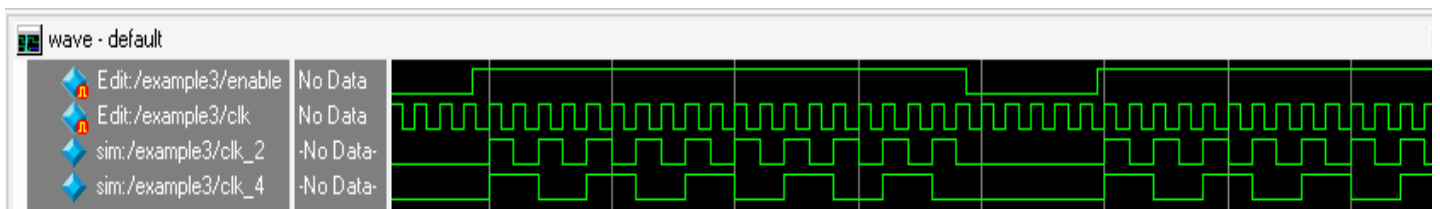


下面是两个简单的时序逻辑电路例子：

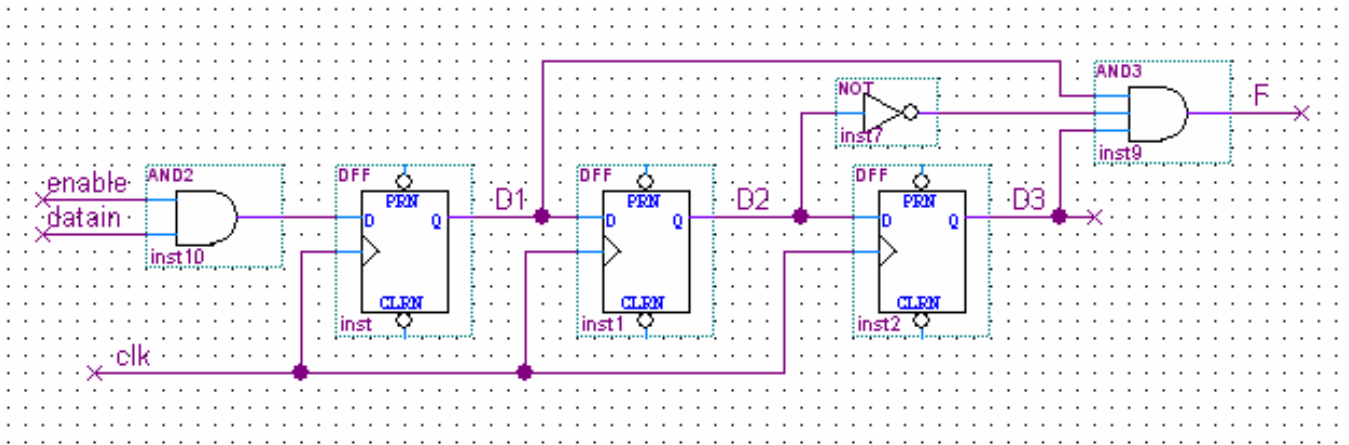
(1)、时钟分频电路



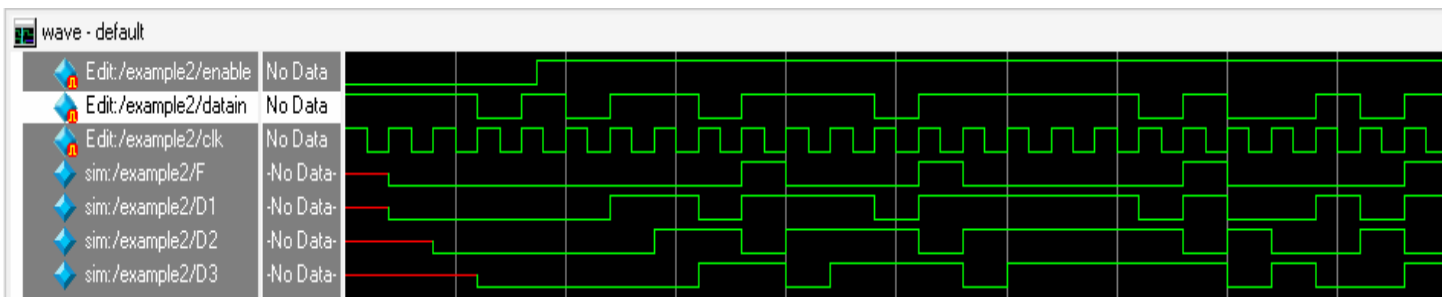
该时序电路的功能为实现对时钟'clk'的 4 分频，其中'clk_2'为 2 分频时钟，'clk_4'为 4 分频时钟，'enable'为该电路的使能信号。其功能仿真波形如下图所示：



(2)、序列检测器



该时序电路实现了一个序列检测器，当输入序列'datain'中出现'101'时，标志位 F 将输出'1'，其他时刻输出'0'。电路中'clk'为时钟信号，'D1'，'D2'，'D3'为移位寄存器的输出，'enable'为该电路的使能信号。其功能仿真波形如下图所示：



可见，时序电路设计的核心是时钟和触发器，这两者也是我们设计电路时需重点关注的。

1.2 毛刺的产生与消除

1.2.1 竞争与冒险

当一个逻辑门的输入有两个或两个以上的变量发生改变时，由于这些变量是经过不同路径产生的，使得它们状态改变的時刻有先有后，这种时差引起的现象称为竞争（Race）。竞争的结果将很可能导致冒险（Hazard）发生（例如产生毛刺），造成错误的后果，并影响系统的工作。

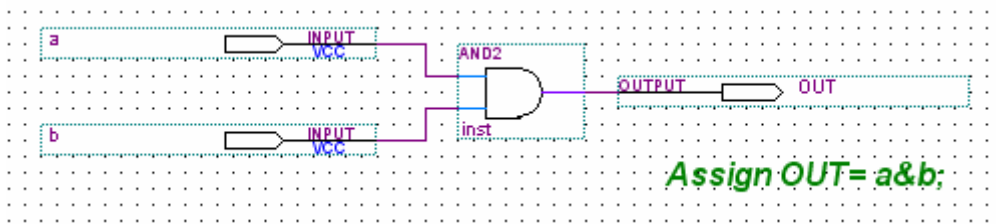
组合逻辑电路的冒险仅在信号状态改变的時刻出现毛刺，这种冒险是过渡性的，它不会使稳态值偏离正常值，但在时序电路中，冒险是本质的，可导致电路的输出值永远偏离正常值或者发生振荡。

避免冒险的最简单的方法是同一時刻只允许单个输入变量发生变化，或者使用寄存器采样的办法。

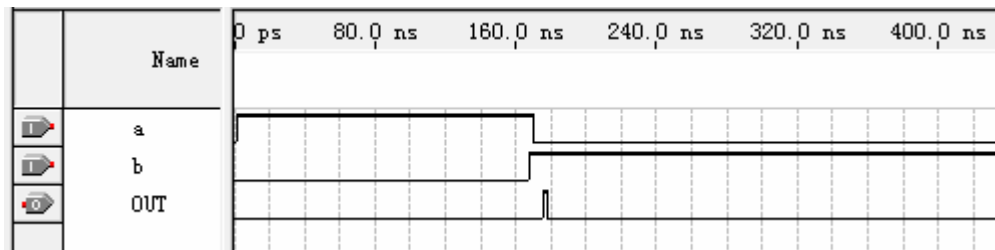
1.2.2 毛刺的产生与危害

信号在 FPGA 器件中通过逻辑单元连线时，一定存在延时。延时的大小不仅和连线的长短和逻辑单元的数目有关，而且也 and 器件的制造工艺、工作环境等有关。因此，信号在器件中传输的时候，所需要的时间是不能精确估计的，当多路信号同时发生跳变的瞬间，就产生了“竞争冒险”。这时，往往会出现一些不正确的尖峰信号，这些尖峰信号就是“毛刺”。

让我们来具体看一下毛刺是如何产生的。下图是一个与门电路，



我们期望的设计是，a 和 b 信号同时变化，这样输出 OUT 将一直为 0，但是实际中 OUT 产生了毛刺，它的仿真波形如下所示：



可见，即使是在最简单的逻辑运算中，如果出现多路信号同时跳变的情况，在通过内部走线之后，就一定会产生毛刺。而现在数字电路设计中的信号往往是由时钟控制的，如果将带有毛刺的输出信号直接连接到时钟输入端、清零或置位端口的设计，可能会导致严重的后果；此外对于多数据输入的复杂运算系统，每个数据都由相当多的位数组成。这时，每一级的毛刺都会对结果有严重的影响，如果是多级的设计，那么毛刺累加后甚至会影响整个设计的可靠性和精确性。

判断一个逻辑电路在某些输入信号发生变化时是否会产生毛刺，首先要判断信号是否会同时变化，然后判断在信号同时变化的时候，是否会产生毛刺，这可以通过逻辑函数的卡诺图或逻辑函数表达式来进行判断。

1.2.3 毛刺的消除

毛刺是数字电路设计中的棘手问题，它的出现会影响电路工作的稳定性、可靠性，严重时会导致整个数字系统的误动作和逻辑紊乱。

可以通过以下几种方法来消除毛刺：

1、输出加 D 触发器

这是一种比较传统的去除毛刺的方法。原理就是用一个 D 触发器去读带毛刺的信号，利用 D 触发器对输入信号的毛刺不敏感的特点，去除信号中的毛刺。在实际中，对于简单的逻辑电路，尤其是对信号中发生在非时钟跳变沿的毛刺信号，去除效果非常的明显。

但是如果毛刺信号发生在时钟信号的跳变沿，D 触发器的效果就没有那么明显了（加 D 触发器以后的输出 q，仍含有毛刺）。另外，D 触发器的使用还会给系统带来一定的延时，特别是在系统级数较多的情况下，延时也将变大，因此在使用 D 触发器去除毛刺的时候，一定要视情况而定，并不是所有的毛刺都可以用 D 触发器来消除。

2、信号同步法

设计数字电路的时候采用同步电路可以大大减少毛刺。由于大多数毛刺都比较短（大概几个纳秒），只要毛刺不出现在时钟跳变沿，毛刺信号就不会对系统造成危害了。因此一般认为，只要在整个系统中使用同一个时钟就可以实现系统同步。但是，时钟信号在 FPGA 器件中传递时是有延时的，我们无法预知时钟跳变沿的精确位置。也就是说我们无法保证在某个时钟的跳变沿读取的数据是一个稳定的数据，尤其是在多级设计中，这个问题就更加突出。因此，做到真正的“同步”就是去除毛刺信号的关键问题。所以同步的关键就是保证在时钟的跳变沿读取的数据是稳定的数据而不是毛刺数据。以下为两种具体的信号同步方法。

（1）信号延时同步法

信号延时法，它的原理就是在两级信号传递的过程中加一个延时环节，从而保证在下一个模块中读取到的数据是稳定后的数据，即不包含毛刺信号。这里所指的信号延时可以是数据信号的延时，也可以是时钟信号的延时。

（2）状态机控制

使用状态机也可以实现信号的同步和消除毛刺的目的。在数据传递比较复杂的多模块系统中，由状态机在特定的时刻分别发出控制特定模块的时钟信号或者模块使能信号，状态机的循环控制就可以使得整个系统协调运作，同时减少毛刺信号。那么只要我们在状态机的触发时间上加以处理，就可以避免竞争冒险，从而抑制毛刺的产生。

3、格雷码计数器

对于一般的二进制或十进制计数器，在计数时，将有多位信号同时跳变。例如一个 3bit 二进制计数器，由'111'转换为'000'时，必将产生毛刺。此时，使用格雷码计数器将避免毛

刺的出现，因为格雷码计数器的输出每次只有一位跳变。

其他关于毛刺的详细讨论，请见补充教程 2：关于毛刺问题的探讨。

1.3 同步电路设计

同步电路是指所有电路在同一个公共时钟的上升沿或下降沿的触发下同步地工作。但在实际系统中，往往存在多时钟域的情况，这时同步的概念有所延伸，不再专指整个设计同步于同一时钟沿，而是指设计应该做到局部同步，在每个时钟域内的电路要同步于同一时钟沿。

1.3.1 同步电路设计的优点：

1. 同步设计能有效地避免毛刺的影响，使得设计更可靠；
2. 同步设计易于添加异步复位reset，以使整个电路有一个确定的初始状态；
3. 同步设计可以减小环境对芯片的影响，避免器件受温度，电压，工艺的影响；
4. 同步设计可以使静态时序分析变得简单和可靠；
5. 同步设计可以很容易地组织流水线，提高芯片的运行速度。

1.3.2 同步电路的设计准则：

1. 尽可能在设计中使用同一时钟，时钟走全局时钟网络。走全局时钟网络的时钟是最简单、最可预测的时钟，它具有很强的驱动能力，可以驱动 FPGA 内部中的所有触发器，并保证 Clock skew 可以小到忽略的地步。

2. 避免使用混合时钟沿采样数据，即避免在设计中同时使用时钟的上升沿和下降沿。

3. 尽量少在模块内部使用计数器分频所产生的时钟。计数器分频时钟需完成的逻辑功能完全可由 PLL 锁相环或时钟使能电路替代。计数器分频时钟的缺点是使得系统内时钟不可控，并产生较大的 Clock skew，还使静态时序分析变得复杂。

4. 避免使用门控时钟。因为经组合逻辑产生的门控时钟极可能产生毛刺，使 D 触发器误动作。

5. 当整个电路需要多个时钟来实现，则可以将整个电路分成若干局部同步电路（尽量以同一个时钟为一个模块），局部同步电路之间接口当作异步接口考虑，而且每个时钟信号的时钟偏差（ ΔT ）要严格控制。

6. 电路的实际最高工作频率不应大于理论最高工作频率，留有设计余量，保证芯片可靠工作。

7. 电路中所有寄存器、状态机在系统被 reset 复位时应处在一个已知的状态。

关于同步电路设计中的其他问题请详见补充教程 3：华为同步电路设计规范。

1.3.3 关于时钟设计的讨论

目前的工程设计中一般使用同步时序电路来完成整个系统的设计，由上一节可见，时钟在同步电路设计中起着至关重要的作用。那么，我们在设计时首先要完成的是对时钟的设计。如今在设计中常见的时钟类型包括：全局时钟、内部逻辑时钟和门控时钟。

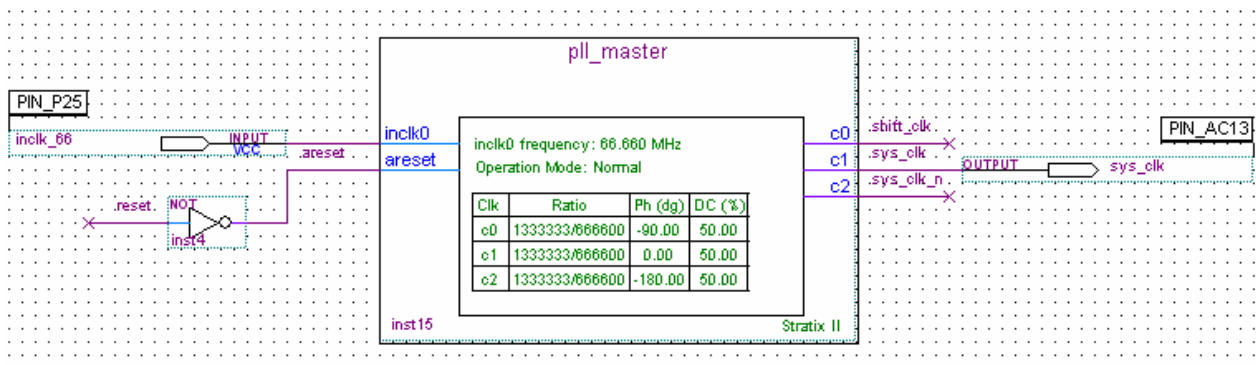
1. 全局时钟

全局时钟即同步时钟，它通过 FPGA 芯片内的全局时钟布线网络或区域时钟网络来驱动，全局时钟具有高扇出、高精度、低 Jitter 和低 Skew 的特点，它到芯片中的每一个寄存器的延迟最短，且该延迟可被认为是固定值。所以我们推荐在所有的设计中的时钟都使用全局时钟。全局时钟的设计有以下几种方法：

- (1). 由 PLL 锁相环来产生全局时钟。
- (2). 将 FPGA 芯片内部逻辑产生的时钟分配至全局时钟布线网络。
- (3). 将外部时钟通过专用的全局时钟输入引脚引入 FPGA。

在我们的设计中，一般推荐电路中的所有的时钟都由 PLL 锁相环产生。一方面，PLL 锁相环可实现倍频和移相的操作，使我们很方便地获得所需频率和相位的时钟；另一方面，PLL 锁相环默认将其驱动的时钟分配至全局时钟网络或区域时钟网络，Jitter 和 Skew 都很小。

下图取自我们项目中的一个 PLL 锁相环设计，该 PLL 用于驱动 DDR 的接口模块。因为功能所需，DDR 接口需要三个 133MHz 的时钟，相位分别是‘ -90° ’、‘ 0° ’、‘ -180° ’，图中所示即为该时钟的产生模块。我们使用 Quartus II 的 Megawizard 生成 PLL 锁相环的 IP core。其中‘inclk_66’为 PLL 锁相环的输入时钟，由外部的 66MHz 晶振提供，经过 PLL 倍频和移相后得到所需的三个全局时钟。



2. 内部逻辑时钟

内部逻辑时钟即指由芯片内部的组合逻辑或计数器分频产生的时钟。

对于组合逻辑时钟，特别是由多级组合逻辑产生的时钟，是要被严格禁止使用的，因为一方面组合逻辑极易产生毛刺，特别是对多级组合逻辑；另一方面组合逻辑电路的 **Jitter** 和 **Skew** 比较大，这将恶化时钟的质量。所以，一般组合逻辑产生的内部时钟仅仅适用于时钟频率较低、时钟精度要求不高的情况。

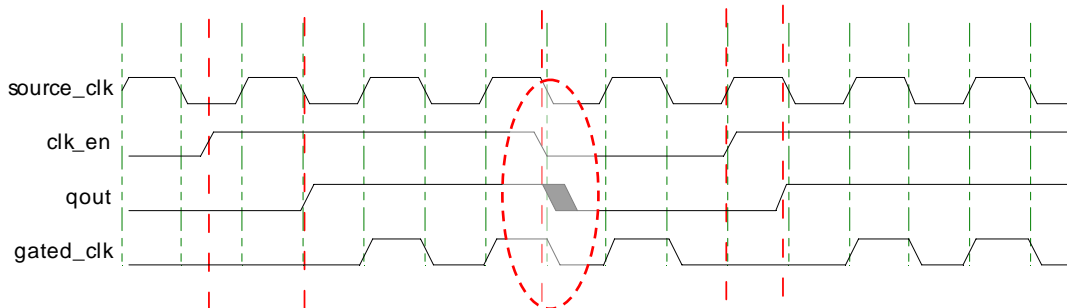
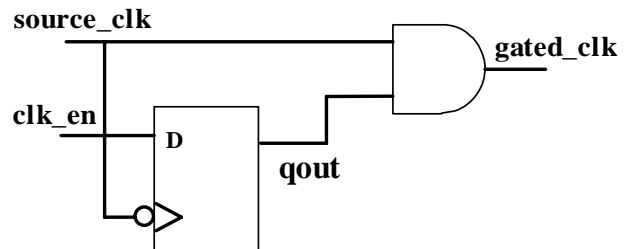
对于计数器分频产生的时钟，也应该尽量少地使用，因为这种时钟会带来比较大的延迟，降低设计的可靠性，也使得静态时序分析变得复杂。计数器分频时钟需完成的逻辑功能完全可由 **PLL** 锁相环或时钟使能电路替代。

还有一种由触发器产生的时钟—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。文中 1.1.2 节描述的时钟分频电路就是一种行波时钟。因为各触发器的时钟之间产生较大的时间偏移，很容易就会违反建立时间、保持时间的要求，导致亚稳态的发生。所以，这种行波时钟要被严格禁止使用。

3. 门控时钟

一般情况下，应该避免使用门控时钟。因为经组合逻辑产生的门控时钟极可能产生毛刺，对系统造成很大危害。但对于某些功耗很大的系统而言，需要使用门控时钟来降低功耗。

我们推荐使用右图中描述的门控时钟的设计，该设计一般不会产生毛刺和亚稳态的问题。因为触发器避免了毛刺的产生，而亚稳态只可能出现在源时钟的下降沿，但是随后它与源时钟低相位相与，最后不会产生影响。

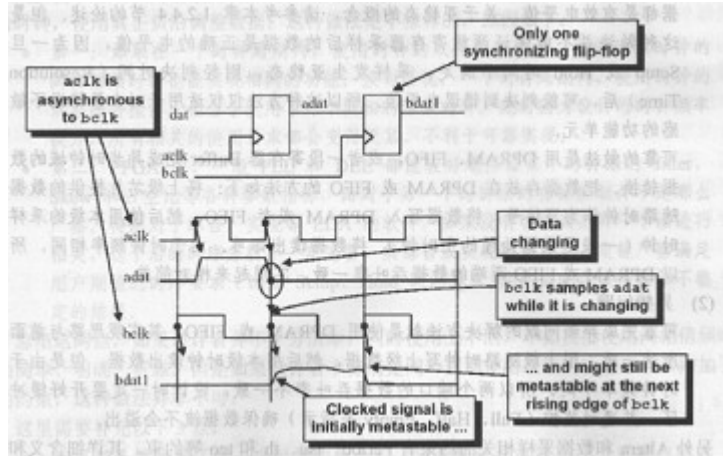


门控时钟最好只在顶层模块中出现，并将其分离到一个在顶层的独立模块中。这同时保证了底层的每个模块有单一的时钟，且在本模块中的时钟不进行门控。

在补充教程 4 和补充教程 5 中，我们对时钟和时序的设计进行了更详细的讨论。

1.3.4 亚稳态

在同步电路或异步电路中，如果触发器的 **setup** 时间或 **hold** 时间不能得到满足，就可能产生亚稳态，此时触发器输出端 **Q** 在有效时钟沿之后比较长的一段时间处于不确定的状态，在这段时间里 **Q** 端将会产生毛刺并不断振荡、最终固定在某一电压值上，此电压值并不一定等于原来数据输入端 **D** 的值。这段时间称为决断时间（**resolution time**）。经过决断时间之后，**Q** 端将稳定到 **0** 或 **1** 上，但是究竟是 **0** 还是 **1**，这是随机的，与输入没有必然的关系。



亚稳态的危害主要体

现在破坏系统得稳定性上，由于输出在稳定下来之前可能是毛刺、振荡、固定的某一电压值，因此亚稳态除了导致逻辑误判之外，严重情况下输出 **0~1** 之间的中间电压值还会使下一级产生亚稳态（即导致亚稳态的传播）。逻辑误判将导致功能性错误，而亚稳态的传播则扩大了故障面，严重时将导致系统崩溃。

在异步时序电路中更容易发生亚稳态，因为异步电路一般具有多个时钟域，数据在两个时钟域间传递时，非常容易导致 **setup** 时间或 **hold** 时间不满足而发生亚稳态。在同步时序电路中，当两个触发器间的组合逻辑延迟过大时，会导致 **setup** 时间不满足而发生亚稳态。

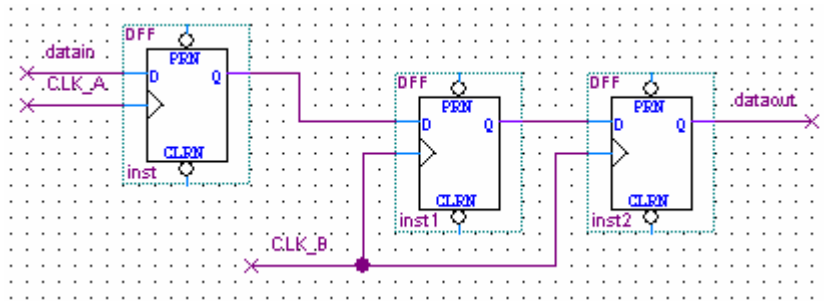
1.3.5 对跨时钟域数据的处理

对跨时钟域数据的处理的核心就是要保证下级时钟对上级数据采样的 **setup** 时间或 **hold** 时间满足要求，即尽量避免亚稳态的发生和传播。但是，我们知道，只要系统中有异步元件，亚稳态就是无法避免的，因此设计的电路首先要减少亚稳态导致错误的发生，其次要使系统对产生的错误不敏感。我们推荐使用以下方法来解决异步时钟域数据同步问题。

1. 用触发器打两拍

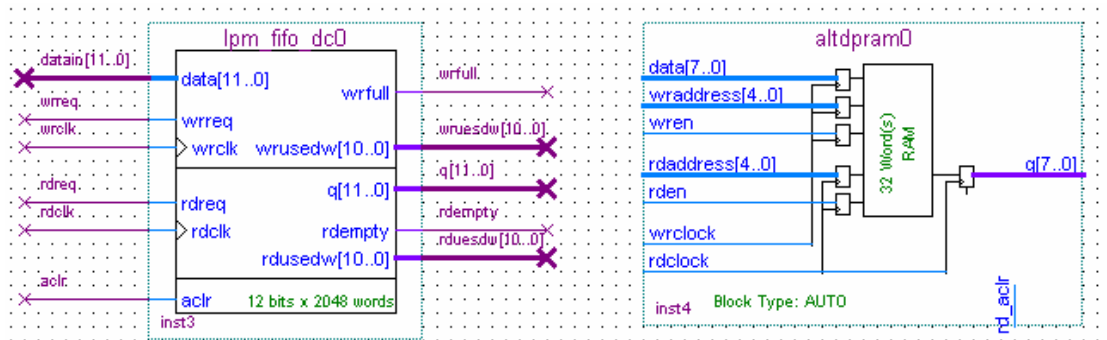
如下图，左边为异步输入端，经过两级触发器同步，在右边的输出将是同步的，而且该输出基本不存在亚稳态。其原理是即使第一个触发器的输出端存在亚稳态，经过一个 **CLK** 周期后，第二个触发器 **D** 端的电平仍未稳定的概率非常小，因此第二个触发器 **Q** 端基本不会产生亚稳态。然而，亚稳态是无法被根除的，一旦亚稳态发生，后果的严重程度依赖于你

设计系统对产生的错误是否敏感。



2. 异步 FIFO 或 DPRAM

因为异步 FIFO 或 DPRAM 使用格雷码计数器设计读写地址的指针，所以它可以很好地避免亚稳态的发生。使用方法如下，将上级芯片提供的数据随路时钟作为写信号，将数据写入异步 FIFO 或 DPRAM，然后使用本级的采样时钟将数据读出即可。唯一需要注意的是，当两级时钟频率不同时，需要设计好缓冲区，并通过监控 full、half、empty、useword 信号，保证数据不会溢出，也不会被读空。



3. 调整时钟相位

这种方法的设计难度较大，而且适用面有限。首先需对跨时钟域数据的路径进行详细的静态时序分析，然后将违反 setup 时间和 hold 时间的情况一一列出，在不影响其它设计性能的前提下，综合考虑调整两级时钟的相位关系，最终使其 setup 时间和 hold 时间满足要求。

2.FPGA 简介

2.1 什么是 FPGA

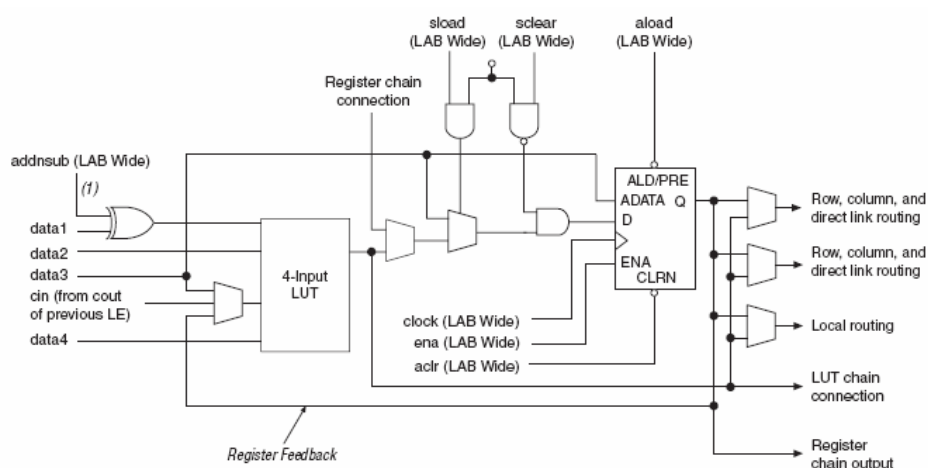
FPGA 是 Field Programmable Gate Array 的缩写,即现场可编程门阵列,是一种可编程的 IC 芯片 (集成电路芯片),以下是目前项目中使用的几种 FPGA 芯片:



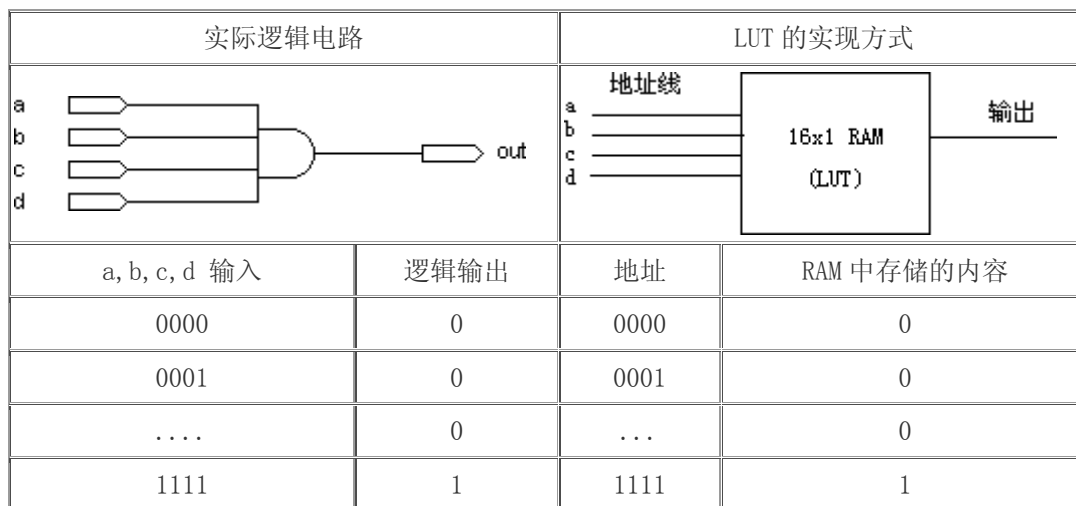
2.2 FPGA 的结构与组成

通常 FPGA 由布线资源分隔的可编程逻辑单元构成阵列,又由可编程 I/O 单元围绕阵列构成整个芯片,排成阵列的逻辑单元由布线通道中的可编程内连线连接起来实现一定的逻辑功能。

目前我们使用的 FPGA 的可编程逻辑单元一般由查找表和触发器构成。下图所示即为 Cyclone 系列 FPGA 芯片的逻辑单元 (LE) 组成。



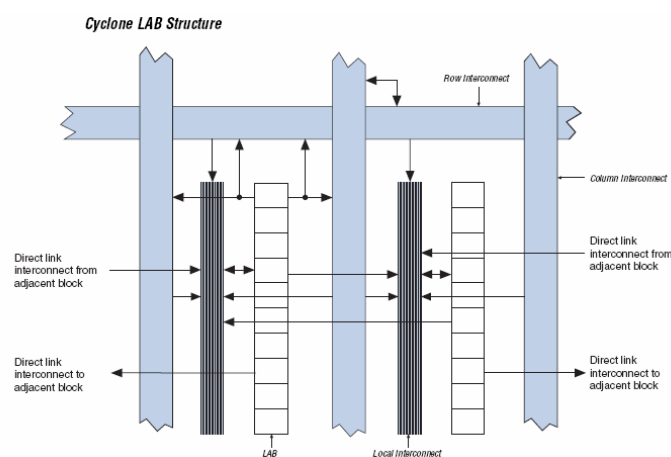
查找表 (Look-Up-Table) 简称为 LUT,其本质上就是一个静态存储器 SRAM。对于下图左边所示的电路,查找表是这样实现的:首先 FPGA 开发软件会自动计算逻辑电路的所有可能的结果,然后把结果事先写入查找表中,FPGA 工作时,输入信号所进行的逻辑运算就等于输入一个地址进行查表,找出地址对应的内容后输出,即实现了该逻辑功能。



如果所设计的是时序电路，需要触发器，则 FPGA 开发软件会自动将触发器配置在查找表的后面，实现组合逻辑时就将触发器旁路掉。

当然，对于复杂的设计，一个 LUT 是无法完成的，FPGA 可以通过进位逻辑将多个 LUT 相连起来，实现 n 输入的查找表，实现设计要求。

通俗地说，FPGA 就是由查找表、触发器和布线资源组成。下图是一个 Cyclone 系列 FPGA 芯片的内部结构，其中一对查找表和触发器构成逻辑单元 LE，若干个 LE 组成逻辑阵列块 LAB，最后再配上各种布线资源，就是一个 FPGA 芯片了。



详细的介绍请见：补充教程 6 FPGA PLD 结构与原理。

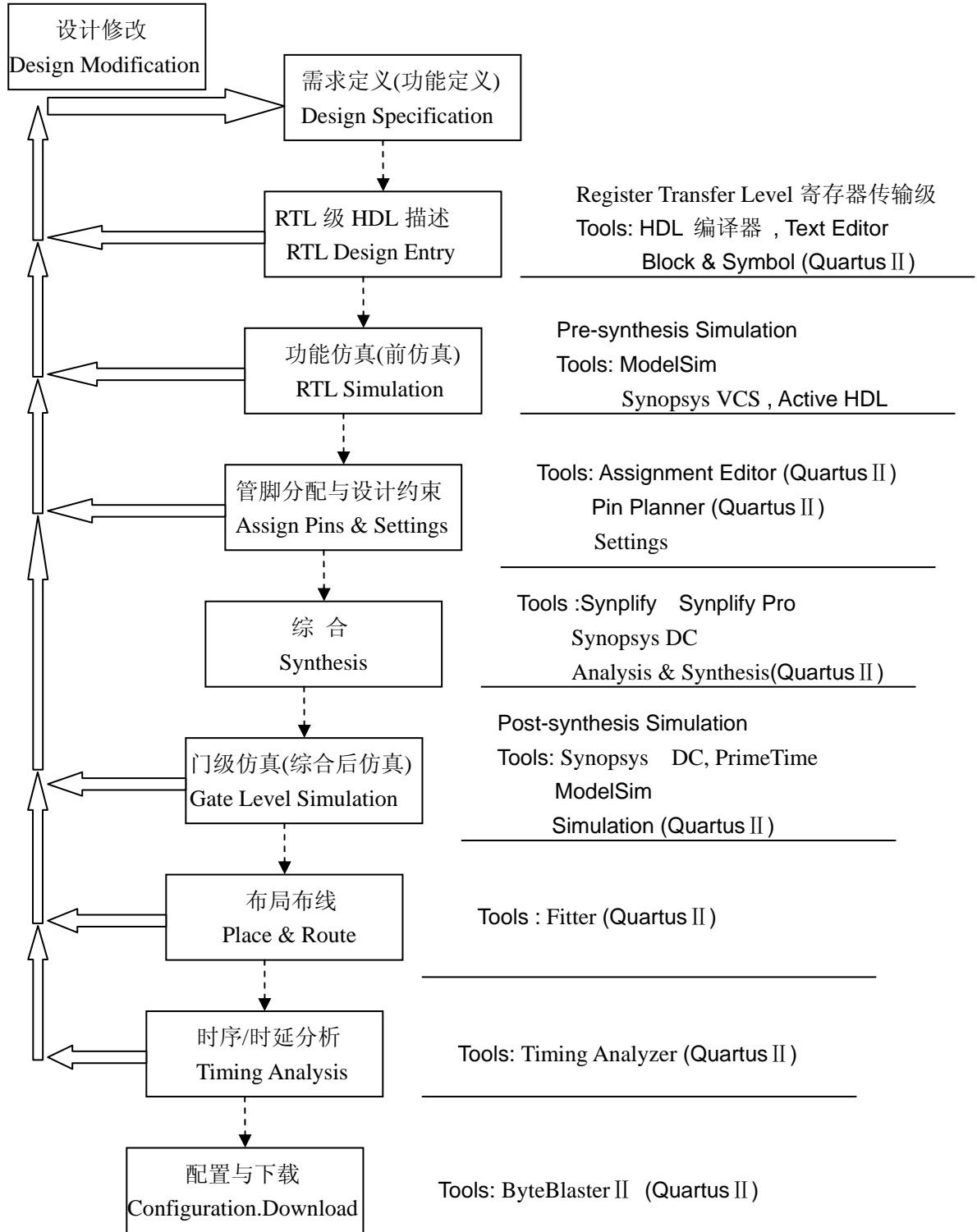
2.3 FPGA 与 ASIC 设计的区别

ASIC 是 Application Specific Integrated Circuit 的缩写，即专用集成电路。ASIC 和 FPGA 属于 SOC (System on a chip 片上系统) 的两个发展方向，两者唯一的区别在于，ASIC 的逻辑电路是固化在其芯片中的，我们可以将 ASIC 理解为不可编程的 FPGA。

由于 FPGA 设计是基于固有的硬件结构（如逻辑单元、块 RAM、PLL/DLL、时钟资源等）的；而 ASIC 设计结构灵活，目标多样，所以 ASIC 设计的代码风格和 FPGA 设计的代码风格有明显差异，特别是在功耗、速度、时序等要求上。例如 ASIC 设计中根据要求会有意识地采用某些组合逻辑、门控时钟等，以降低功耗或提高速度。

3. FPGA 开发流程

HDL (Hardware Design Language) 和原理图是两种最常用的数字硬件电路描述方法，HDL 设计法具有更好的可移植性、通用性和模块划分与重用性的特点，在目前的工程设计中被广泛使用。所以，我们在使用 FPGA 设计数字电路时，其开发流程是基于 HDL 的。



3.1 需求定义(功能定义)

设计和实现一个系统的第一步，是明确整个系统的性能指标，然后进一步将系统功能划分为可实现的具体功能模块，同时明确各模块的功能与基本时序，还可大致确定模块间的接口，如时钟、读写信号、数据流和控制信号等。

3.2 RTL 级 HDL 描述

RTL 级（寄存器传输级）指不关注寄存器和组合逻辑的细节（如使用了多少逻辑门、逻辑门的连接拓扑结构等），通过描述寄存器到寄存器之间的逻辑功能的 HDL 设计方法。RTL 级比门级更抽象，同时也更简单和高效。RTL 级的最大特点是可以直接用综合工具将其综合为门级网表。RTL 级设计直接决定着系统的功能和效率。我们使用的 HDL 语言是 verilog。

3.3 功能仿真(前仿真)

功能仿真也称综合前仿真，其目的是验证 RTL 级描述是否与设计意图一致。为了提高效率，功能仿真需要建立 testbench，其测试激励一般使用行为级 HDL 语言描述。

3.4 管脚分配与设计约束

无论是 RTL 级还是门级的 HDL 设计方法，在实现该逻辑时都需要与实际的 FPGA 芯片相匹配。管脚分配是指将设计文件的输入输出信号指定到器件的某个管脚，设置此管脚的电平标准、电流强度等。设计约束指对设计的时序约束和在综合、布局布线阶段附加的约束等。

3.5 综合

将 RTL 级 HDL 语言翻译成由与、或、非门等基本逻辑单元组成的门级连接（网表），并根据设计目标与要求（约束条件）优化所生成的逻辑连接，输出门级网表文件。

3.6 门级仿真(综合后仿真)

在综合后通过后仿真来检查综合结果是否与原设计一致。一般，综合后仿真和功能仿真的测试激励相同。由于综合工具日益完善，在目前的 FPGA 设计中，这一步骤被省略掉。

3.7 布局布线

布局布线就是使用综合后的网表文件，将工程的逻辑与时序要求与器件的可用资源相匹配。也可以简单地将布局布线理解为对 FPGA 内部查找表和寄存器资源的合理配置，那么‘布

局’可以被理解挑选可实现设计网表的最优的资源组合，‘布线’就是将这些查找表和寄存器资源以最优方式连接起来。

3.8 时序/时延分析

通过时序/时延分析获得布局布线后系统的延时信息，不仅包括门延时，而且还有实际的布线延时。时序/时延分析的时序仿真是最准确的，能较好地反映芯片的实际工作情况，同时发现时序违规（Timing Violation），即不满足时序约束条件或者器件固有时序规则（建立时间、保持时间）的情况。

3.9 配置与下载

通过编程器（programmer）将布局布线后的配置文件下载至 FPGA 中，对其硬件进行编程。配置文件一般为.pof 或.sof 文件格式，下载方式包括 AS（主动）、PS（被动）、JTAG（边界扫描）等方式。

4.RTL 设计

4.1 使用 verilog 进行 RTL 设计

使用 verilog 进行 RTL 设计一般可归纳为 3 种基本的描述方式：

- 1、数据流描述：采用 assign 连续赋值语句
- 2、行为描述：使用 always 语句或 initial 语句块的过程赋值语句。
- 3、结构化描述：实例化已有的功能模块或原语，即平常所说的元件例化和 IP core。

其中，连续赋值语句是连续驱动的，也就是说只要输入变化，都会导致该语句的重新计算。其被赋值的变量不会被电路寄存。过程赋值语句包括非阻塞过程赋值、阻塞过程赋值和连续过程赋值。对于这些抽象的概念，我们将在其后的实际例子中详细解释。

4.1.1 硬件设计意识

RTL 设计其实就是用语言的方式去描述硬件电路行为的过程。这同一般的软件设计有很大区别，因为对于很多的软件代码，硬件电路是无法实现的（即无法综合，从语言到硬件电路的解析过程称为综合）。我们只能使用可综合的代码结构来实现我们所需的硬件电路。

首先，我们需要建立硬件设计的意识，硬件意识是 RTL 级设计的基础。

- 1.电路在物理上是并行工作的。其含义是，一旦接通电源，所有电路都同时工作。
- 2.电路行为的先后顺序通过时钟节拍得顺序来体现。

4.1.2 RLT 级设计时需注意的问题：

1. 凡是在 always 或 initial 语句中赋值的变量，一定是 reg 类型变量；凡是在 assign 语句中赋值的变量，一定是 wire 类型变量；
2. 定义存储器：reg [3:0] MEMORY [0:7]；地址为 0~7，每个存储单元都是 4bit；
3. 由于硬件是并行工作的，在 Verilog 语言的 module 中，所有描述语句（包括连续赋值语句 assign、行为语句块 always 和 initial 语句块以及模块实例化）都是并发执行的。
4. 使用完备的 if...else 语句，使用条件完备的 case 语句并设置 default 操作，以防止产生锁存器 latch，因为锁存器对毛刺敏感。
5. 严禁设计组合逻辑反馈环路，它最容易引起振荡、毛刺、时序违规等问题。
6. 不要在两个或两个以上的语句块（always 或 initial）中对同一个信号赋值。

4.1.3 阻塞赋值与非阻塞赋值

在 Verilog HDL 中，有两种过程性赋值方式，即阻塞赋值（blocking assignment）和非阻塞赋值（non-blocking assignment）。这两种赋值方式有着根本的区别，如果使用不当，综合出来的结果会和设计的期望结果相去甚远。

1. 阻塞赋值的操作符为“=”。它的含义是在计算等式右侧表达式值及完成其赋值时不会被其他的 verilog 语句打断，就是说，在当前赋值没有完成之前，它阻塞了其他 verilog 语句的执行。

例如：begin

```
x = y + c; //语句 1
y = a + b; //语句 2
end
```

语句 1 和语句 2 都要求在同一仿真时刻执行，但在语句 1 没有执行完之前，是不会执行语句 2 的，它"阻塞"语句 2 的执行。这里说的"执行完"指的是在本时刻计算完 y+c 的值，并赋给 y。若该仿真时刻 y=5(旧值)，c=4,a=3,b=1，则 x=9,y=4。

还有下面的情况，过程 1 和过程 2 是同一个 module 中的两个 always 语句块。

过程 1:

过程 2:

```
always @ (posedge clk or negedge rst_n) always @ (posedge clk or negedge rst_n)
begin                                  begin
  if ( rst_n == 1'b0)                  if ( rst_n == 1'b0)
    y = 1'b0;                          x = 1'b1;
  else                                  else
    y = x;                              x = y;
end                                    end
```

仿真开始时，两个 always 语句块将并行地同时执行。如果仿真器先执行过程 2 的话，则在复位完后，x、y 的值最终都会变为 1；如果先执行过程 1，则 x、y 的值最终都会变为 0。

为什么会出现这种情况呢？因为按照 Verilog 的标准，上例中两个 always 块是并行执行的，与书写的前后顺序无关，但是，verilog 标准没有规定在这种特殊情况下的执行顺序。所以，这需要在编程时避免这样的代码。

2. 非阻塞赋值的操作符为“<= ”。它的含义是在赋值操作时刻开始时计算等式右边，赋值操作时刻结束时更新等式左边。在这期间，可以执行其它 verilog 语句，也不阻塞其它 verilog 语句的执行。

例如：begin

```

x <= y + c; //语句 1
y <= a + b; //语句 2
end

```

虽然语句 1 的在语句 2 之前，但是由于是非阻塞赋值，语句 1 的执行不影响语句 2 的执行，在该仿真时刻完了后，才更新左端的值。若该仿真时刻 $y=5$ (旧值)， $c=4, a=3, b=1$ ，则 $x=8, y=4$ 。

同样，对于下面的情况，过程 1 和过程 2 是同一个 module 中的两个 always 语句块。

<pre> 过程 1: always @ (posedge clk or negedge rst_n) begin if (rst_n == 1'b0) y = 1'b0; else y = x; end </pre>	<pre> 过程 2: always @ (posedge clk or negedge rst_n) begin if (rst_n == 1'b0) x = 1'b1; else x = y; end </pre>
--	--

由于仿真时刻完了后才会去更新左端的值，所以上面两个过程将最终形成一反馈移位寄存器。

在实际使用中，我们需要遵循以下的原则：

1. 在时序逻辑中，使用非阻塞赋值；
2. 在组合逻辑中，使用阻塞赋值；
3. 在同一个 always 块中，不要混合使用阻塞赋值和非阻塞赋值；
4. 在同一个 always 块中，如果既有组合逻辑又有时序逻辑，使用非阻塞赋值；
5. always 模块的敏感表为电平敏感信号时，使用阻塞赋值；
6. 不要使用#0 时延进行赋值。
7. 不要在阻塞赋值中使用时延语句。
8. 在行为级描述中，若语句间是顺序执行的关系，使用阻塞赋值。

另外一点需注意的就是，阻塞赋值是相对时间赋值，非阻塞赋值是绝对时间赋值。

4.2 HDL 代码的可综合性

任何符合 HDL 语法标准的代码都是对硬件行为的一种描述，但不一定是可直接对应成电路的设计信息。行为描述可以基于不同的层次，如系统级，算法级，寄存器传输级(RTL)、门级等等。以目前大部分 EDA 软件的综合能力来说，只有 RTL 或更低层次的行为描述才能保证是可综合的。绝大部分电路设计必须遵循 RTL 的模式来编写代码，而不能随心所欲得

写仅仅符合语法的 HDL 代码。

4.2.1 哪些是不可综合的代码

我们不可能将所有不可综合的代码都一一列出，我们举几个例子来说明这个问题：

1.对于一些抽象的行为描述代码是不可综合的。比如我们常见的延迟语句(如：`#delay`)、初始化语句 `initial` 以及等待语句 `wait` 都是不可综合的，他们所描述的功能对硬件而言是不可实现的。

2.对于一些抽象的运算代码也是不可综合的。比如，实现两个变量相除运算的代码：

```
always @(posedge clk)
    c<=A/B;
```

我们将发现这句代码只能在前仿真中正确执行，在 `Quartus II` 和其他 EDA 软件中都不能将其综合成硬件。试想一下，如果我们自己用笔算除法是怎么做的？从高位到低位逐次试除、求余、移位。试除和求余需要减法器，商数和余数的中间结果必须有寄存器存储；而这些运算显然不能在一个时钟延上完成，它需要一个状态机来控制时序,经过多个时钟周期才能得出结果。一句简单的 `C=A/B` 同所有这些相比显得太抽象，对于只能接受 RTL 或更低层次描述的 EDA 软件来说确实太难实现。(注：有些 FPGA 的配套软件提供乘除法的运算模块，但也只能支持直接调用，不支持把形如 `C=A/B` 的语句综合成除法模块)。

3.对于不定次数的循环运算是不可综合的，比如对于如下的代码段：

```
for (i=0; i<wordlength ; i=i+1)
    parity = parity xor data[i];
```

当 `wordlength` 为变量时，任何 EDA 软件都不能综合这个代码。这是因为硬件规模必须是有限的、固定的。当综合软件遇到循环语句时，总是将其展开成若干条顺序执行的语句，然后再综合成电路。若 `wordlength` 是常数，则展开的语句数是确定的，具有可综合性；而若它是变量时，展开的语句数不确定，对应的硬件电路数量也不能确定，无法被综合。这样的语句还包括 `while`, `repeat`, `forever` 等，他们都不可综合。

4.2.2 如何判断自己写的代码是可综合的

通常 EDA 软件对 HDL 代码的综合能力总是比人的思维差。也就是说，对于一段代码，如果你不能想象出一个较直观的硬件实现方法，那 EDA 软件肯定也不行。比如说，加法器、多路选择器是大家都熟悉的电路，所以类似 `A+B-C`, `(A>B)?C:D` 这样的运算一定可以综合。而除法、开根、对数等等较复杂的运算，必须通过一定的算法实现，没有直观简单的实现方法，则可以判断那些计算式是不能综合的，必须按它们的算法写出更具体的代码才能实现。此外，硬件无法支持的抽象行为描述，当然也不能被综合。

不过，这样的判断标准非常主观模糊，遇到具体情况还得按设计人员自己的经验来判断。如果要一个相对客观的标准，一般来说：在 RTL 级的描述中，所有逻辑运算和加减法运算、以及他们的有限次组合，基本上是可综合的，否则就有无法综合的可能性。当然，这样的标准仍然有缺陷，所以，正确的判断仍然要靠实践来积累经验。当你可以较准确判断代码的可综合性的时候，你对 HDL 的掌握就算完全入门了。

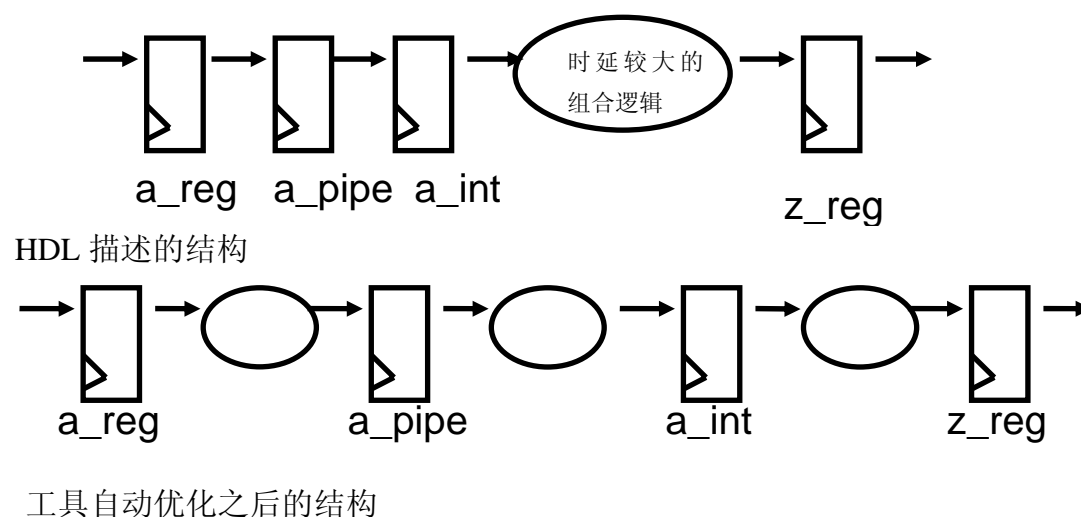
4.3 RTL 代码的优化

优化 RTL 代码追求的最终目标是面积或者速度，或者是两者的平衡。面积和速度是 FPGA 设计的两个基本标准。“面积”是指一个设计所消耗 FPGA 的逻辑资源数量，“速度”指设计在芯片上稳定运行所能够达到的最高频率。两者是对立统一的，即要求一个设计同时具备面积最小、速度最高时不现实的。我们的设计目标应该是在满足设计时序要求和最高频率要求的前提下，占用最小的芯片面积，或者在所规定的面积下，使设计的时序余量更大，频率更高。

4.3.1 Pipelining 技术

Pipelining，即流水线时序优化方法，其本质是调整一个较大的组合逻辑路径中的寄存器位置，用寄存器合理分割该组合逻辑路径，从而降低路径 Clock-To-Output 和 Setup 等时间参数的要求，达到提高设计频率的目的。

具体实现可通过下图的方法，首先对时延较大的组合逻辑，在它旁边加上若干个需优化的寄存器，然后运用优化工具自动将大组合逻辑拆分为几个小的组合逻辑，同时将寄存器放在小组合逻辑在中间。



工具自动优化之后的结构

4.3.2 模块复用与资源共享

模块复用与资源共享的目的在于节约 FPGA 的逻辑资源，即节约面积。我们通过下面这个例子来说明这个优化方法。

如果需要一个补码平方器，输入是 8bit 补码，求其平方和。因为输入是补码，所以当最高位是 1 时，表示原值是负数，需要按位取反，加 1 后再平方；当最高位是 0 时，表示原值是正数，直接求平方。下面是两种 RTL 设计实现方式：

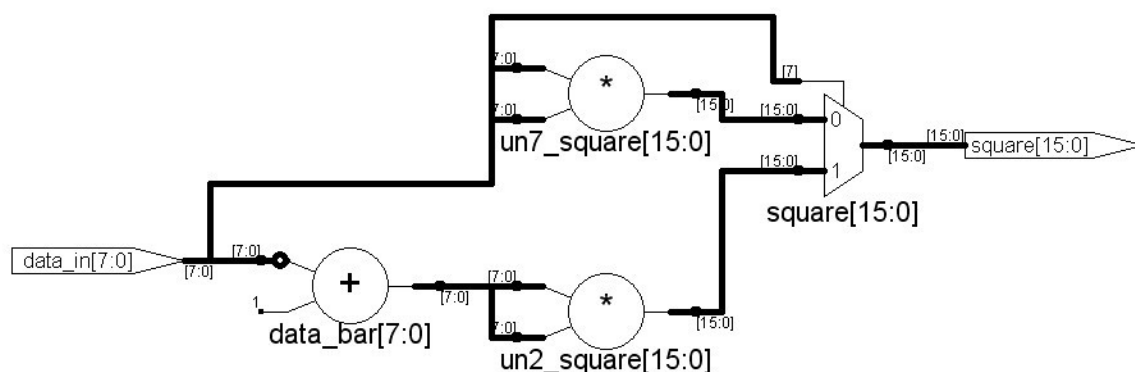
第一种 RTL 设计实现：

```
module resource_share1 (data_in,square);
    input [7:0] data_in; //输入是补码
    output [15:0] square;
    wire [7:0] data_bar;
    assign data_bar = ~data_in + 1;
    assign square=(data_in[7])? (data_bar*data_bar) : (data_in*data_in);
endmodule
```

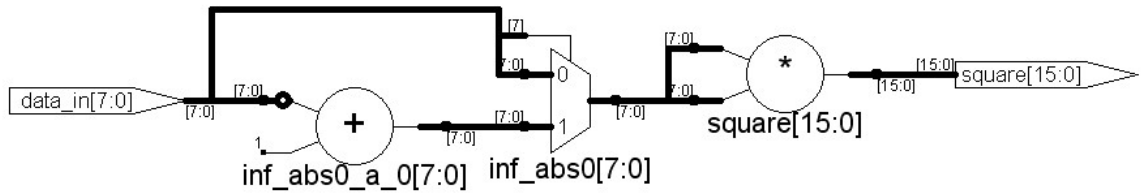
第二种 RTL 设计实现：

```
module resource_share2 (data_in,square);
    input [7:0] data_in; //输入是补码
    output [15:0] square;
    wire [7:0] data_tmp;
    assign data_tmp = (data_in[7])? (~data_in + 1) : data_in;
    assign square = data_tmp * data_tmp;
endmodule
```

可以发现，第一种 RTL 设计需要两个 16bit 乘法器，同时平方；而第二种 RTL 设计只使用了一个乘法器。我们使用 Synplify Pro 对两个 RTL 设计分别进行综合，结果第一种 RTL 设计所占用的逻辑资源为 140 个 LUTs，第二种 RTL 设计为 75 个 LUTs。下图是两个设计的实现结果：



第一种 RTL 设计，占用 140 个 LUTs

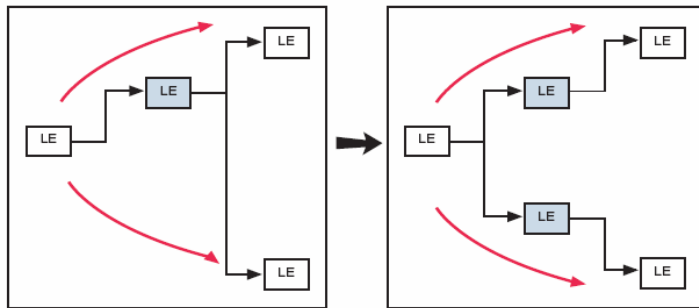


第二种 RTL 设计，占用 75 个 LUTs

4.3.3 逻辑复制

逻辑复制是一种通过增加面积而改善时序条件的优化手段。逻辑复制最常使用的场合是调整信号的扇出。如下图，FPGA 一般通过插入 buffer 来满足信号的驱动能力的要求，在复制了一条路径后，就可减轻这条路径的驱动能力，同时路径的延时也同时减小了。

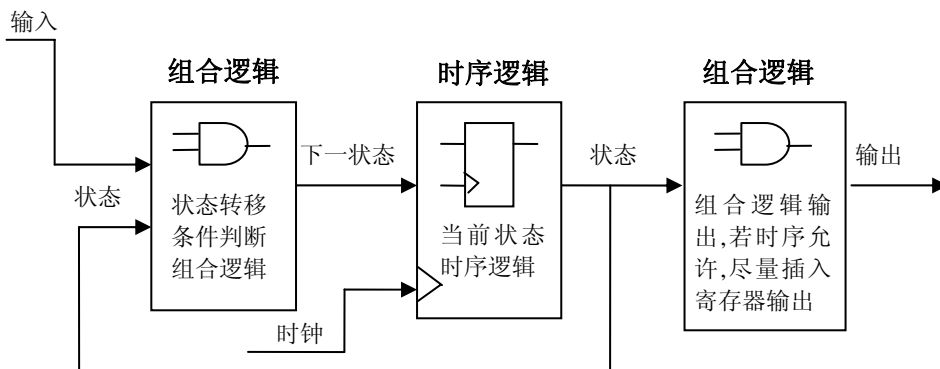
Figure 9-7. Register Duplication



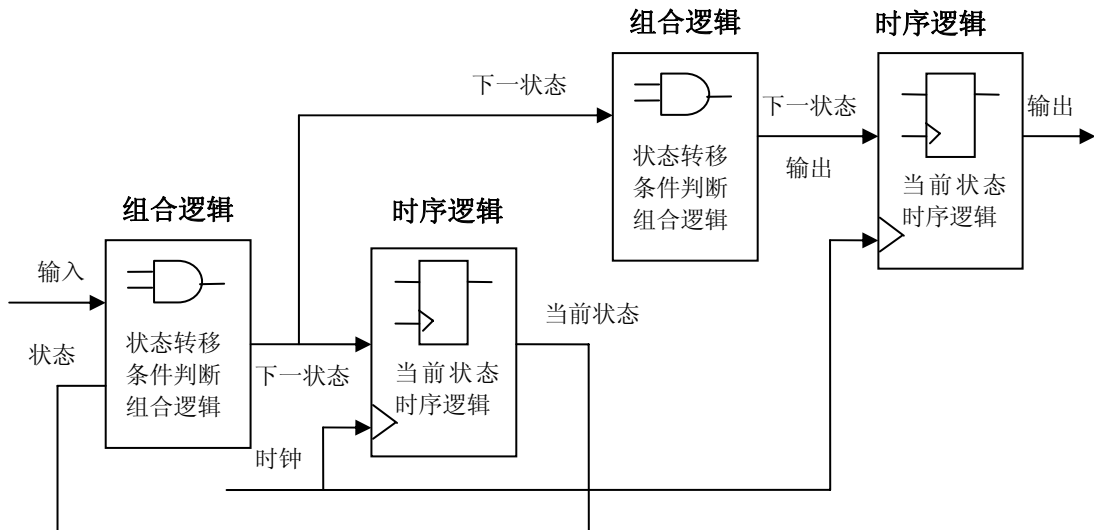
4.4 状态机的设计

我们推荐在状态机的设计中采用两段式写法（2 个 always 模块）或三段式写法（3 个 always 模块）。

两段式写法的核心思想是，一个 always 模块采用同步时序描述状态转移；另一个 always 模块采用组合逻辑方式判断状态转移条件，描述状态转移规律。其结构如下图：



三段式写法的核心思想是，一个 `always` 模块采用同步时序的方式描述状态转移，一个采用组合逻辑的方式判断状态转移条件，描述状态转移规律，第三个 `always` 模块使用同步时序电路描述每个状态的输出。其结构如下图：



我们归纳出状态机设计的其它技巧和准则：

1. 状态机的编码最好使用 `one-hot`（独热码）。
2. 一个完备的状态机应该具备初始化状态和默认状态。
3. 状态机的编码可以用 `parameter` 定义，不推荐使用 `define` 宏定义。
4. 时序逻辑 `always` 模块使用非阻塞赋值“`<=`”，组合逻辑模块使用阻塞赋值“`=`”。
5. 使用完备的 `if...else` 语句和 `case` 语句。
6. `case` 语句需具备 `full_case` 和 `parallel_case` 属性。`full_case` 定义了所有可能的输入模式，`parallel_case` 定义了 `case` 项是不重复的。

下面是一个状态机的例子，它采用了三段式写法，`one-hot` 编码，并且符合其它状态机的设计准则。

```

module state ( nrst,clk, i1,i2,o1,o2,err);
  input      nrst,clk;
  input      i1,i2;
  output     o1,o2,err;
  reg        o1,o2,err;
  reg  [2:0]  NS,CS;
  parameter [2:0]    //one hot with zero idle
    IDLE  = 3'b000,
    S1    = 3'b001,
    S2    = 3'b010,
    ERROR = 3'b100;

```



```

always @ (posedge clk or negedge nrst) //1st always block, sequential state transition
  if (!nrst)
    CS <= IDLE;
  else
    CS <=NS;

```

```

always @ (nrst or CS or i1 or i2) //2nd always block, combinational condition judgment
  begin
    NS = 3'bx;
    case (CS)
      IDLE:  begin
                if (~i1)          NS = IDLE;
                if (i1 && i2)      NS = S1;
                if (i1 && ~i2)    NS = ERROR;
              end
      S1:    begin
                if (~i2)          NS = S1;
                if (i2 && i1)      NS = S2;
                if (i2 && (~i1))  NS = ERROR;
              end
      S2:    begin
                if (i2)           NS = S2;
                if (~i2 && i1)     NS = IDLE;
                if (~i2 && (~i1)) NS = ERROR;
              end
      ERROR: begin
                if (i1)           NS = ERROR;
                if (~i1)          NS = IDLE;
              end
      default: begin
                NS = IDLE;
              end
    endcase
  end

```

```

always @ (posedge clk or negedge nrst) //3rd always block, the sequential FSM output
  if (!nrst)
    {o1,o2,err} <= 3'b000;
  else
    begin
      {o1,o2,err} <= 3'b000;
      case (NS)
        IDLE: {o1,o2,err}<=3'b000;
        S1:   {o1,o2,err}<=3'b100;
      endcase
    end

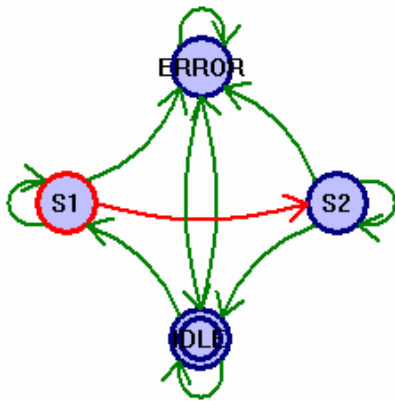
```

```

S2: {o1,o2,err}<=3'b010;
ERROR: {o1,o2,err}<=3'b111;
endcase
end
endmodule

```

我们推荐使用工具 Synplify Pro 分析该状态机，可以使用专用的 FSM 观察器（FSM Viewer）进行分析。下图为在 FSM Viewer 中得到的状态转移图和 FSM 信息显示选项卡。左图中被选中的状态和转移路径，将在右图中同时被标示出。

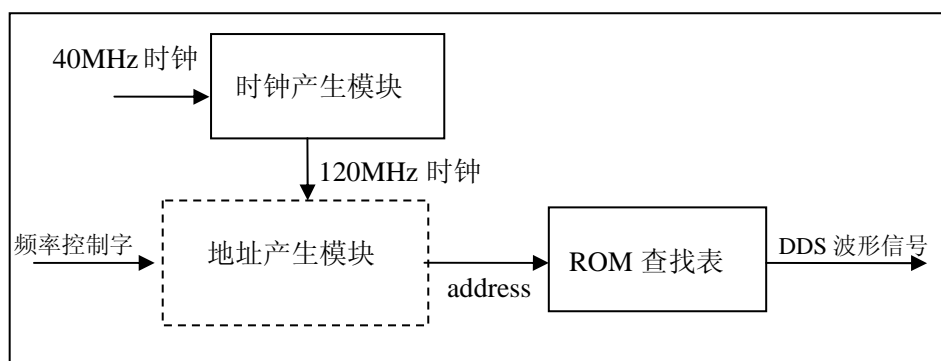


	From State	To State	Condition
1	ERROR	ERROR	i1
2	S2	ERROR	!i2&!i1
3	S1	ERROR	i2&!i1
4	IDLE	ERROR	!i2&i1
5	S2	S2	i2
6	S1	S2	i2&i1
7	S1	S1	!i2
8	IDLE	S1	i2&i1
9	ERROR	IDLE	!i1
10	S2	IDLE	!i2&i1
11	IDLE	IDLE	!i1


5. Quartus II 设计实例

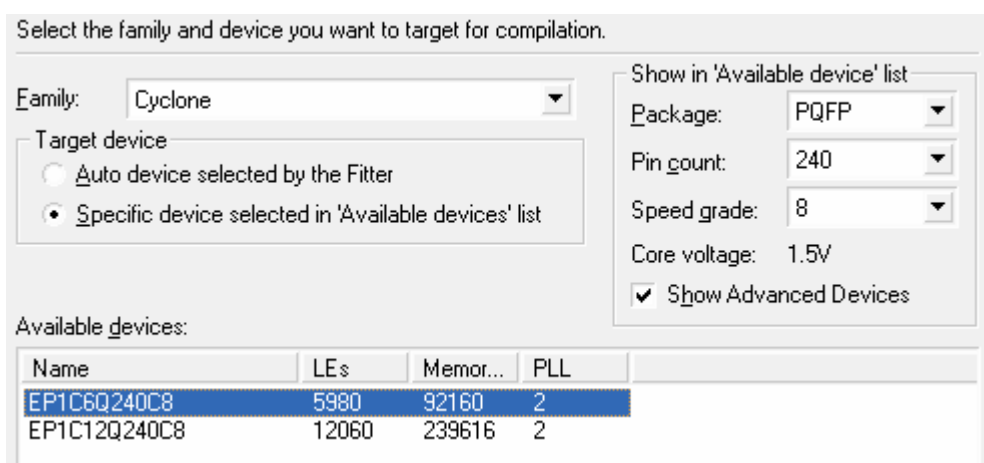
下面我们将通过一个 Quartus II 设计实例来向大家介绍如何进行实际的 FPGA 设计开发。

该实例设计实现一个简单的DDS正弦波发生器。它可分解为3个部分来设计：时钟产生模块，地址产生模块，ROM查找表模块。整个系统实现的思路是：首先，由外部晶振引入40MHz的时钟到FPGA内部，进入时钟产生模块，对时钟进行处理并3倍频后，得到一个稳定精确的120MHz的系统时钟。然后，地址产生模块在系统时钟的激励下，将频率控制字与累加寄存器输出的数据进行累加，然后把累加结果作为地址输出给ROM查找表模块。最后，ROM查找表模块在每个系统时钟的上升沿，按照地址来读取ROM查找表中相应的波形采样点数据并输出，该数据就是最终的DDS信号。那么下面我们就来开始设计这个DDS正弦波发生器。



5.1 新建 project

首先，建立一个 Quartus II 工程，点击 file →  New Project Wizard... 在弹出的对话框中输入工程目录和工程名，然后在器件选择框中选择 FPGA 芯片的型号。我们在这里假定使用 cyclone 系列的 EP1C6Q240C8。

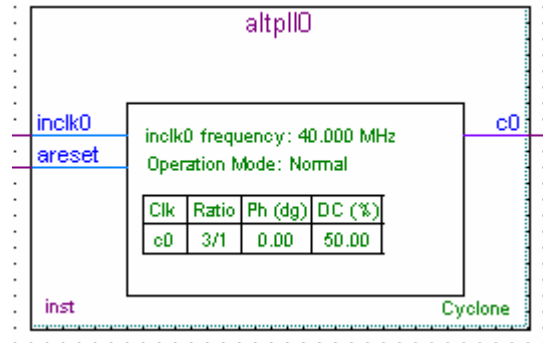


5.2 设计输入

设计输入的第一步，是建立顶层设计文件：点击 file→NEW→Block Diagram/Schematic File 这样就建立一个新的原理图文件，我们将这个原理图文件作为顶层文件。

对于时钟产生模块，我们打算用PLL锁相环来实现，所以可以使用Quartus II自带的IP core调用工具Megafunction来产生PLL锁相环的。

点击 tools→MegaWizard Plug-In Manager... 或者右键选择 Insert→Symbol；选择 create a new custom megafuction variation，然后选择 I/O 下的 ALTPLL，接着按照要求配置各项参数，最后得到PLL模块。

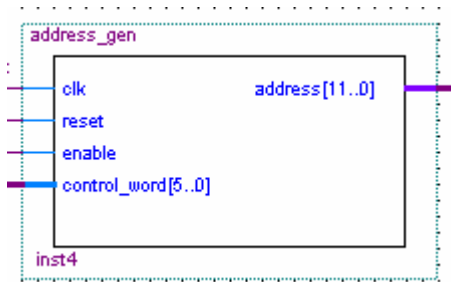


下一步是实现地址产生模块，点击 file→NEW→Verilog HDL File 建立一个新的verilog文件，然后输入RTL代码：

```
`timescale 1ns/10ps
module address_gen (clk, reset, enable, control_word, address);
    input clk, reset, enable;
    input [5:0] control_word;
    output [11:0] address;
    reg [11:0] address;
    always @(posedge clk or negedge reset)
        begin
            if (reset == 1'b0)
                begin
                    address <= 12'h000;
                end
            else if (enable == 1'b1)
                begin
                    address <= address + {6'b0, control_word};
                end
            else
                begin
                    address <= address;
                end
        end
end
```

endmodule

接着我们将这些 RTL 代码生成为一个子模块，以方便将其加入到顶层文件中。点击 file → Create/Update → Create Symbol Files for Current File 即生成了该模块，然后点击右键选择 Insert → Symbol，引用该模块至原理图中。

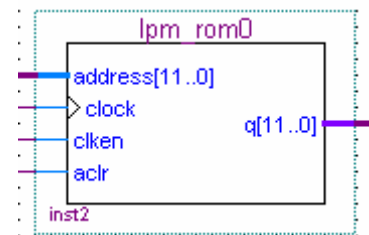
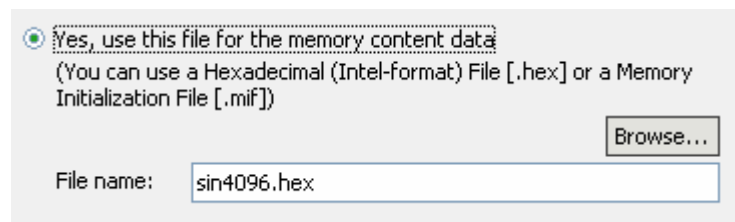


最后一步，是实现 ROM 查找表模块。我们同样可以调用 Quartus II 的 IP core 来实现，

因为在 Megafunction 中提供了专用的 ROM 模块。点击 tools →

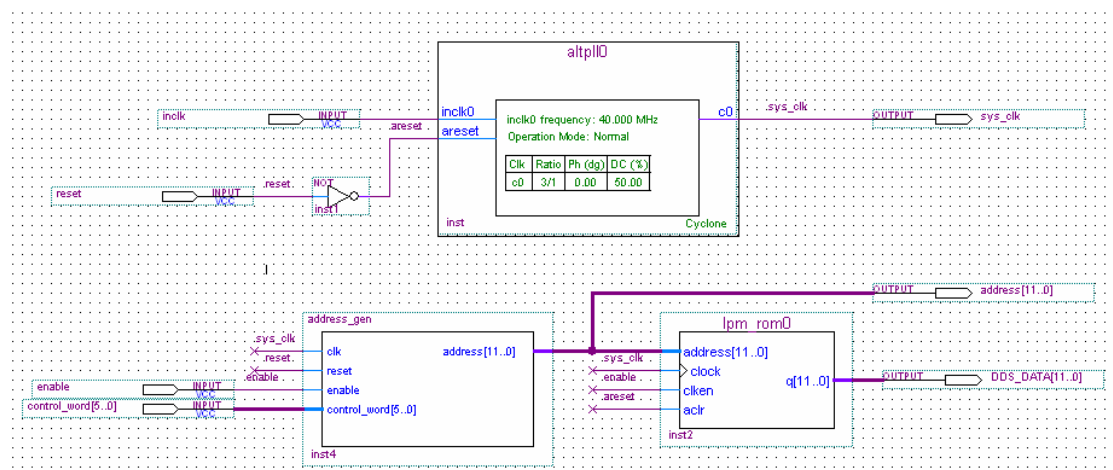


然后选择 Storage 下的 LPM_ROM，设置 ROM 容量为 4096 × 12bit，最后还需要指定 ROM 的初始值，如下左图。正弦查找表的初始值一般由 matlab 产生，然后存储成 .hex 或 .mif 文件。这样就得到了 ROM 查找表模块。




这样，整个系统就设计完成了，下图显示的就是整个设计的顶层文件。在设计的最后，我们应该确保设计中没有语法错误和设计违规，可以使用 Processing → Start →

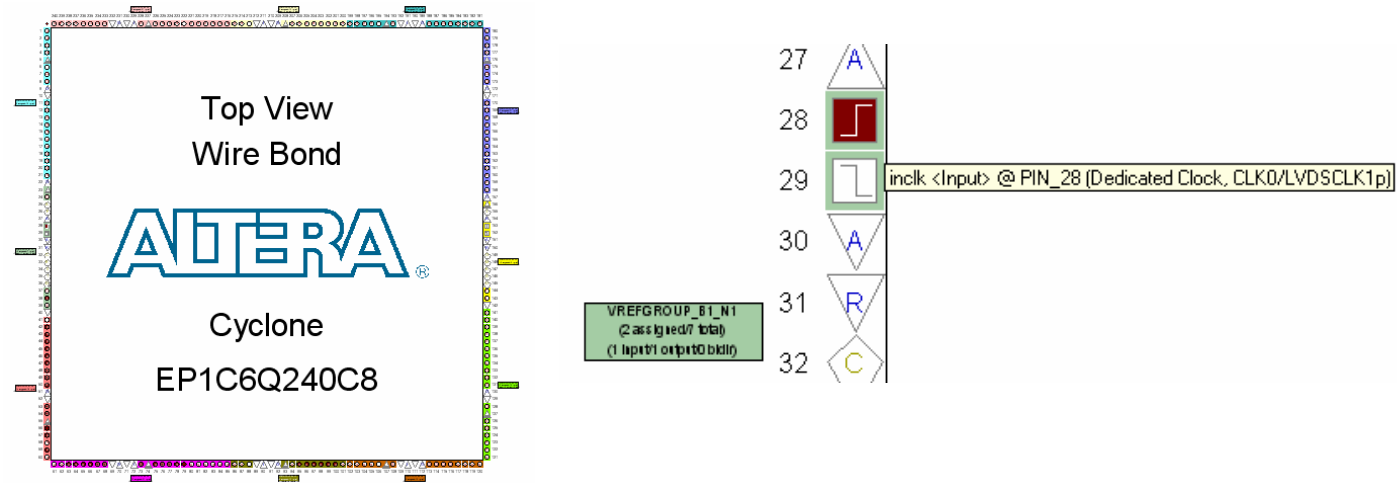
Start Analysis & Synthesis 来检查。



5.3 管脚分配与设计约束


在顶层文件设计完后，我们需要将 project 中涉及到的引脚分配至指定的 pin 上，这一步就将我们逻辑设计和实际的 FPGA 硬件联系起来，因为管脚分配同 PCB 板上的硬件设计必须是一一对应的，在某些特殊情况下，绘制 PCB 板前需验证管脚分配是否能够通过。

点击 Assignments →  Pin Planner，进入管脚分配的界面。下图是 FPGA 芯片的管脚分布图，每个管脚对应其相应的功能，如 pin28 是专用的时钟输入引脚。




在管脚分配时，需注意我们在逻辑设计中定义的引脚功能是否被 FPGA 硬件允许，其次是电平标准要选择正确，这一点也是和硬件相关的。下图显示的是这个 project 中引脚的分配情况，电平标准为 LVTTTL。

Named: *		All Pins						
	Node Name	Direction	Location /	I/O Bank	Vref Group	I/O Standard	Reserved	Group
1	inclck	Input	PIN_28	1	B1_N1	LVTTTL (default)		
2	sys_clk	Output	PIN_38	1	B1_N1	LVTTTL (default)		
3	address[11]	Output	PIN_42	1	B1_N2	LVTTTL (default)		address[11..0]
4	address[10]	Output	PIN_43	1	B1_N2	LVTTTL (default)		address[11..0]
5	address[9]	Output	PIN_44	1	B1_N2	LVTTTL (default)		address[11..0]
6	address[8]	Output	PIN_45	1	B1_N2	LVTTTL (default)		address[11..0]
7	address[7]	Output	PIN_46	1	B1_N2	LVTTTL (default)		address[11..0]
8	address[6]	Output	PIN_47	1	B1_N2	LVTTTL (default)		address[11..0]
9	address[5]	Output	PIN_48	1	B1_N2	LVTTTL (default)		address[11..0]
10	address[4]	Output	PIN_49	1	B1_N2	LVTTTL (default)		address[11..0]
11	address[3]	Output	PIN_50	1	B1_N2	LVTTTL (default)		address[11..0]
12	address[2]	Output	PIN_53	1	B1_N2	LVTTTL (default)		address[11..0]
13	address[1]	Output	PIN_54	1	B1_N2	LVTTTL (default)		address[11..0]
14	address[0]	Output	PIN_56	1	B1_N2	LVTTTL (default)		address[11..0]
15	reset	Input	PIN_57	1	B1_N2	LVTTTL (default)		
16	enable	Input	PIN_59	1	B1_N2	LVTTTL (default)		
17	DDS_DATA[...	Output	PIN_63	4	B4_N2	LVTTTL (default)		DDS_DATA[1...
18	DDS_DATA[...	Output	PIN_64	4	B4_N2	LVTTTL (default)		DDS_DATA[1...
19	DDS_DATA[9]	Output	PIN_65	4	B4_N2	LVTTTL (default)		DDS_DATA[1...

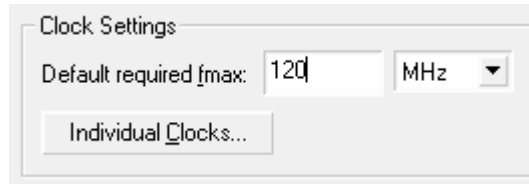
最后，对管脚分配进行验证，点击 Processing → Start →  Start I/Q Assignment Analysis。

设计约束指对设计的时序约束和在综合、布局布线阶段附加的约束等。因为对每个 project，约束是不同的，所以我们这里只举例说明。

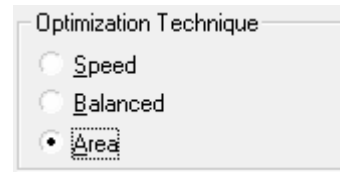
比如对这个 project，我们希望系统的最高工作频率为 120MHz，那么我们点击


Assignments →  Settings... Ctrl+Shift+E，

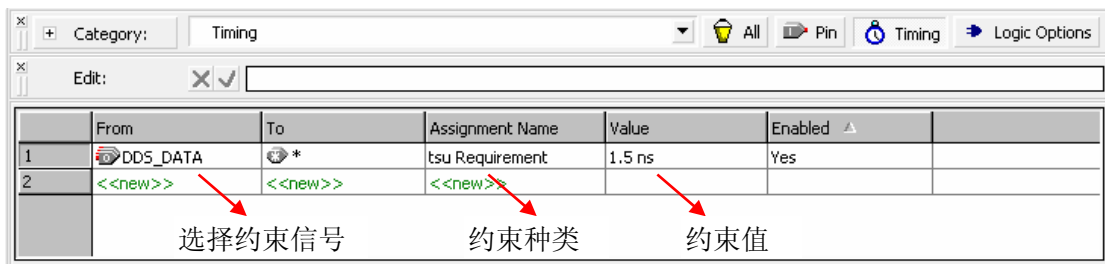
在窗口中可以设置：



又比如，我们希望节省 FPGA 芯片资源，那么就可以在综合时对面积进行优化，还是在刚才的窗口中设置：



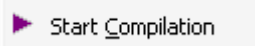
此外，如果需要对某个信号进行单独的约束时，就要进入 Assignments Editor 对话框来添加约束，点击 Assignments →  Assignment Editor Ctrl+Shift+A，按下图的显示来选择约束信号、约束种类和约束值，即设置完毕。



至于具体施加何种设计约束，就只有等待大家在实践中慢慢掌握了，可以查阅其他书籍或 Quartus II 使用手册，更多的还是依靠经验。

5.4 全编译

在以上步骤都完成后，我们将对整个 project 进行全编译（Compilation）。点击 Processing

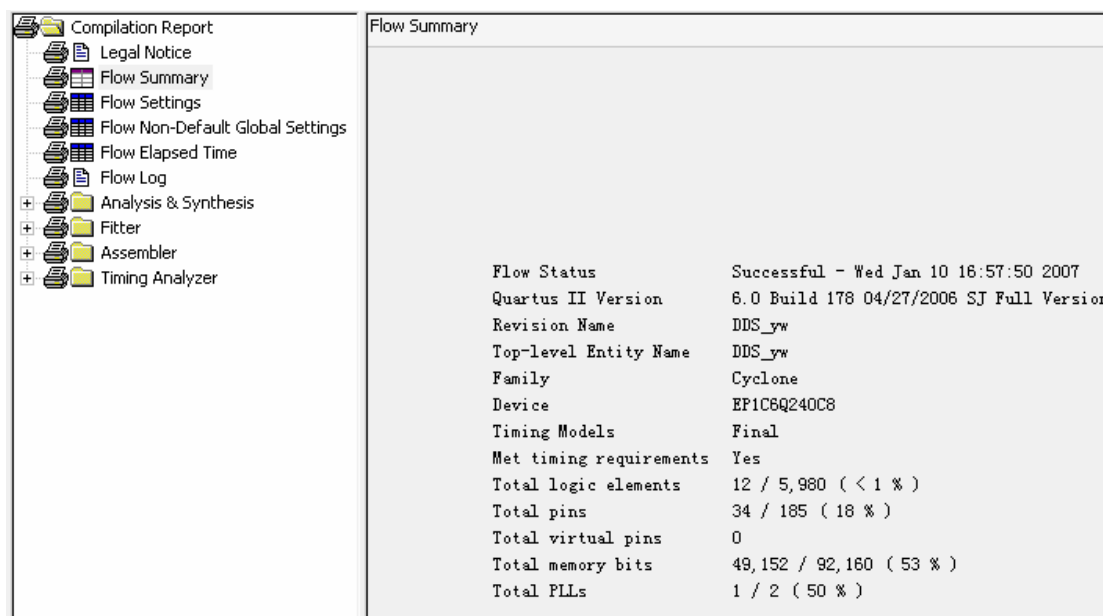
→  Start Compilation。

全编译主要包括四步操作：分析综合（Analysis & Synthesis）、布局布线（Fitter）、编程配置（Assembler）、时序时延分析（Timing Analyzer），这些操作都是一次性完成的，无需干预。

Module	Progress %	Time
Full Compilation	72 %	00:00:31
Analysis & Synthesis	100 %	00:00:12
Fitter	100 %	00:00:14
Assembler	89 %	00:00:05
Timing Analyzer	0 %	00:00:00

全编译的同时还生成了一份编译报告，这个报告中包含了许多重要信息。尤其是时序分

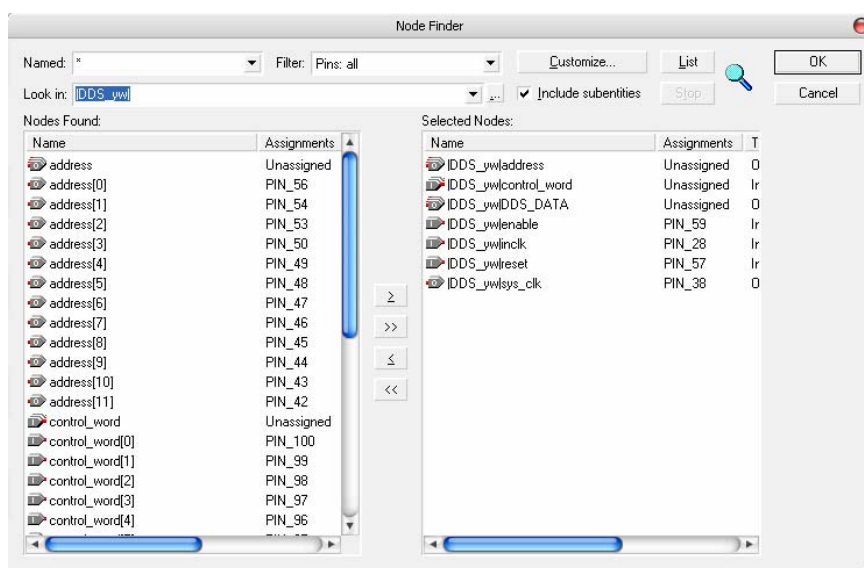
析报告，如果设计中有时序违规的情况发生，报告中的那一项将被显示为红色。一般来说，我们必须保证项目中没有任何时序违规的情况。



5.5 时序仿真

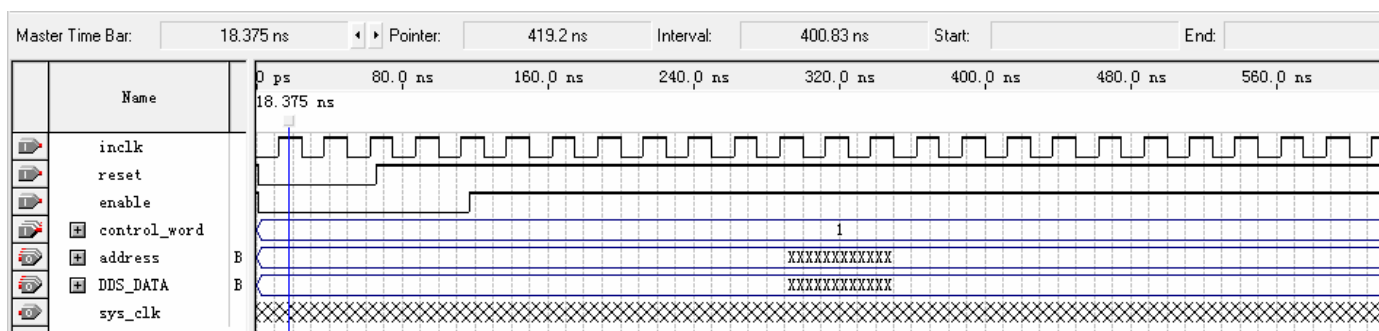
为了验证设计的正确性，必须进行时序仿真。

首先建立仿真文件，点击 **file**→**NEW**→**Vector Waveform File**，即得到仿真波形文件。然后右键→**Insert Node or Bus**→**Node Finder...** 导入需要仿真的信号。

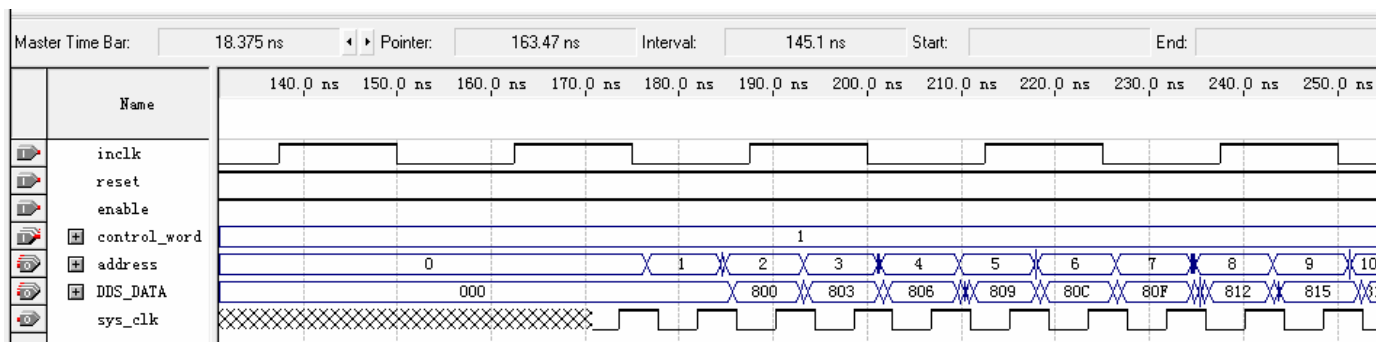


然后就是绘制仿真波形。大家要明白这一点，我们对设计进行仿真，无非是施加激励，

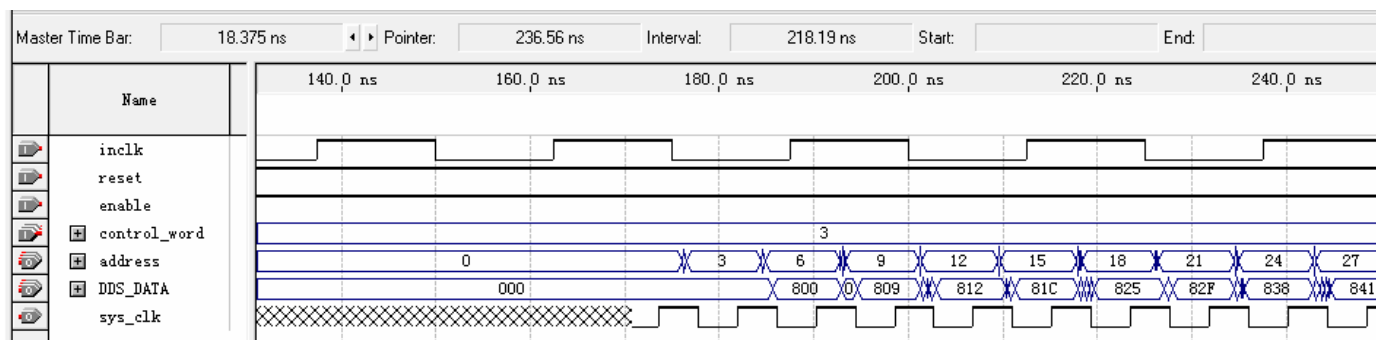
然后观察输出（即响应）是否符合我们的设计要求。绘制波形就是施加激励的一种方式。下图是仿真之前的波形图，暂时将频率控制字设置为 1。



运行仿真。点击 Processing → Start Simulation，得到下图的仿真结果：



修改频率控制字为 3，得到下图的仿真结果：



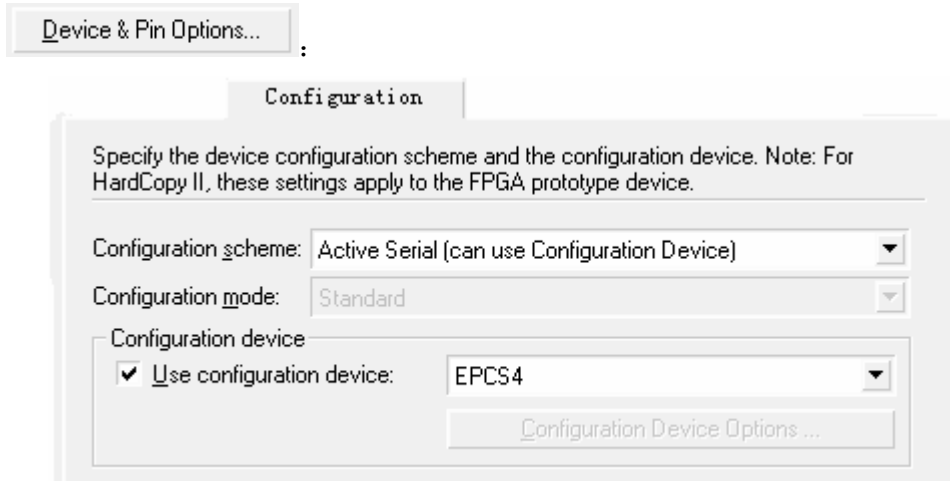
可以看出，仿真波形都是正确的，满足我们的设计要求，这样，就可以下载至 FPGA 中进行调试了。

补充一点：大家可以清晰地看到波形仿真中的毛刺，这是因为 Quartus II 的仿真是综合后仿真，就是说是带有延时信息的。毛刺产生的原因是因为组合逻辑在系统工作频率很高时产生了竞争。


5.6 下载编程

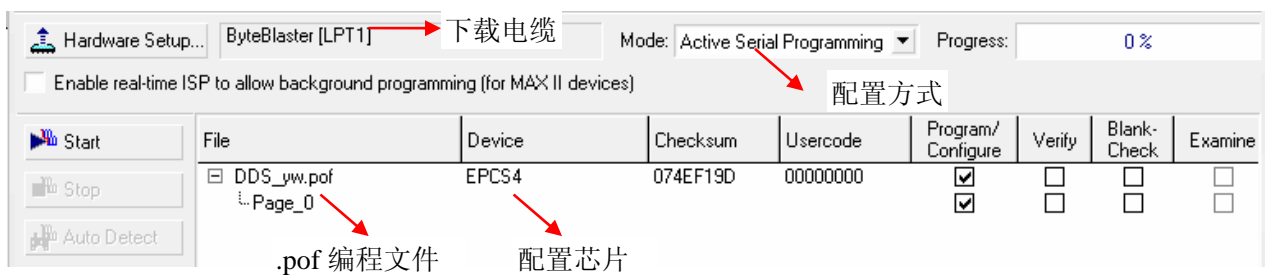
最后一步就是将全编译产生的编程文件（.pof 或 .sof 文件）下载至 FPGA 中。

首先，我们需要设置配置方式和配置芯片，点击 Assignments → Device →



在设置完成后，需要重新运行全编译。


然后，点击 Tools →  Programmer，进入下载编程配置窗口。



我们使用的下载电缆是 ByteBlaster II，如右图。

安装下载电缆的驱动请见补充教程 7：安装 QuartusII 下载电缆。



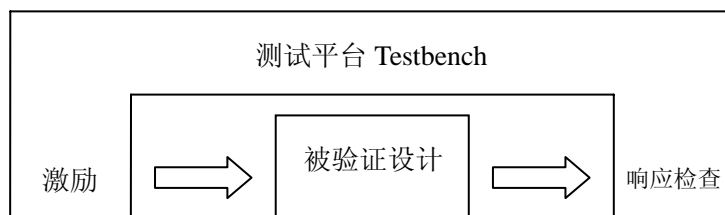
最后，在硬件连接正常的情况下，点击  Start 即完成配置。需要注意的是，在任何情况下插拔下载电缆，都必须关电。

6. ModelSim 和 Testbench

我们在进行功能仿真时，经常需要用到 ModelSim 和 Testbench。由于篇幅的限制，这里只做简要的介绍。

ModelSim 的使用请见补充教程 8：modelsim 使用教程。

Testbench，其实就是测试平台的意思，一般 Testbench 的结构如下：



Testbench 的编写一般使用行为级语法，因为它不需要被综合。而且一般使用元件例化的方法将被验证设计例化至 Testbench 文件中。下面结合 DDS 正弦波发生器的设计列举一个 testbench 的实例。

首先，我们要得到“被验证设计”文件。我们设计的 DDS 正弦波发生器的顶层文件是一个用原理图方式描述的文件（格式为.bdf），我们可以点击 file→Create/Update→

Create HDL Design File for Current File

，得到 verilog 描述方式的顶层文件，如下：

```
module DDS_generator (inclk, reset, enable, control_word, DDS_DATA);
input   inclk;
input   reset;
input   enable;
input   [5:0] control_word;
output  [11:0] DDS_DATA;

wire sys_clk;
wire [11:0] SYNTHESIZED_WIRE_0;
wire SYNTHESIZED_WIRE_1;

altpll0 Ualtpll0 (.inclk0(inclk),
                 .areset(reset),
                 .c0(sys_clk));

lpm_rom0 Ulpm_rom0 (.clock(sys_clk),
```

```

        .clken(enable),
        .aclr(reset),
        .address(SYNTHESIZED_WIRE_0),
        .q(DDS_DATA));
assign  SYNTHESIZED_WIRE_1 = ~reset;
address_gen  Uaddress_gen (.clk(sys_clk),
        .reset(SYNTHESIZED_WIRE_1),
        .enable(enable),.control_word(control_word),
        .address(SYNTHESIZED_WIRE_0));

endmodule

```

然后设计 testbench，testbench 包括两部分，一是激励，二是被验证设计的元件例化。对于 DDS 正弦波发生器，我们共需要时钟 inclk，复位 reset，使能 enable 和频率控制字 control_word 这四个激励源。那么具体的 testbench 设计如下：

```

`timescale 1ns/10ps
module Testbench ;

parameter CLK_PERIOD =20 ;
reg clk ;
reg rst ;
reg enable;
reg word;
reg dds_dataout;

initial //时钟激励 （产生周期为 20ns，即频率为 50MHz 的时钟）
begin
    clk = 0;
    forever begin
        # (CLK_PERIOD/2) clk = ~ clk ;
    end
end

initial //Reset 激励 （产生 5 个周期长度低电平的 reset 信号 ）
begin
    rst = 1;
    # CLK_PERIOD rst = 0;
    # (5*CLK_PERIOD) rst = 1;
end

initial //enable 激励 （等待复位操作结束后，对 enable 信号置位）

```

```

begin
    enable=0;
    # (3*CLK_PERIOD) enable=0;
    @(rst) enable=1;
end

initial //频率控制字激励 (先将频率控制字设置为 1, 在 1000 个周期后变为 3)
begin
    word = 6'b1;
    # (1000*CLK_PERIOD) word = 6'3;
end

$display ("Simulation Finished!"); // 显示"Simulation Finished!"
$stop; //停止仿真
end

DDS_generator UDDS_generator (.inclk(clk), // 被验证设计的元件例化
                                .reset(rst),
                                .enable(enable),
                                .control_word(word),
                                .DDS_DATA(dds_dataout));

endmodule

```

同时, 我们也可以将这两个文件 DDS_generator 和 Testbench 添加至 ModelSim 中进行功能仿真。

至此, 本教程就全部书写完毕了, 希望这个教程能给大家提供帮助, 也希望大家在实际的项目中不断体会、总结、提高, 最终成为一位 FPGA 高手。

(本教程作者: 电子科技大学 416 教研室 杨威)

如有问题, 请发 email: lovsnowy@163.com

参考书籍:

- 1、Altera FPGA/CPLD 设计(基础篇) EDA 先锋工作室
- 2、Altera FPGA/CPLD 设计(高级篇) EDA 先锋工作室
- 3、基于 Verilog HDL 的数字系统应用设计 王钊 卓兴旺 编著
- 4、设计与验证 Verilog HDL EDA 先锋工作室