

Altera FPGA 全速漂移 开发指南

LED 闪烁与 PLL 配置实例

欢迎加入 FPGA/CPLD 助学小组一同学习交流：

EDN:

http://group.ednchina.com/GROUP_GRO_14596_1375.HTM

ChinaAET: <http://group.chinaaet.com/273>

淘宝店链接: <http://myfpga.taobao.com/>

技术咨询: orand_support@sina.com

特权 HSC 最新资料例程下载地址:

<http://pan.baidu.com/s/1pLmZaFx>

版本信息		
时间	版本	状态
2016-06-16	V1.00	创建。



目录

Altera FPGA 全速漂移 开发指南	1
LED 闪烁与 PLL 配置实例	1
1 功能概述.....	3
2 新建 Quartus II 工程.....	4
3 IP 核配置——PLL.....	10
4 引脚分配.....	18
5 闲置引脚设置.....	25
6 Verilog 代码解析	27
7 板级调试.....	40

1 功能概述

本实例使用 Quartus II 中用于例化 IP 核的 Megafuction 配置一个 PLL 模块，PLL 模块产生的 25MHz 时钟进行 24 位循环计数，24 位计数器的最高位赋值给连接到 LED 指示灯的引脚上，由此实现了 LED 以固定频率闪烁的效果。

该实例的功能框图如图 1 所示。FPGA 外部引脚的复位信号进入 FPGA 后，首先做了一次“异步复位，同步释放”的处理，然后这个复位信号输入到 PLL 模块，在 PLL 模块输出时钟有效后，它的锁定信号 locked 就会拉高，我们以此信号作为系统的复位信号，因此也做了“异步复位，同步释放”的处理。PLL 的输入时钟为 FPGA 外部晶振产生的时钟信号，它经过 PLL 处理后产生一个 25MHz 的时钟信号，24 位计数器在这个 PLL 时钟的“节拍”下不停的计数。

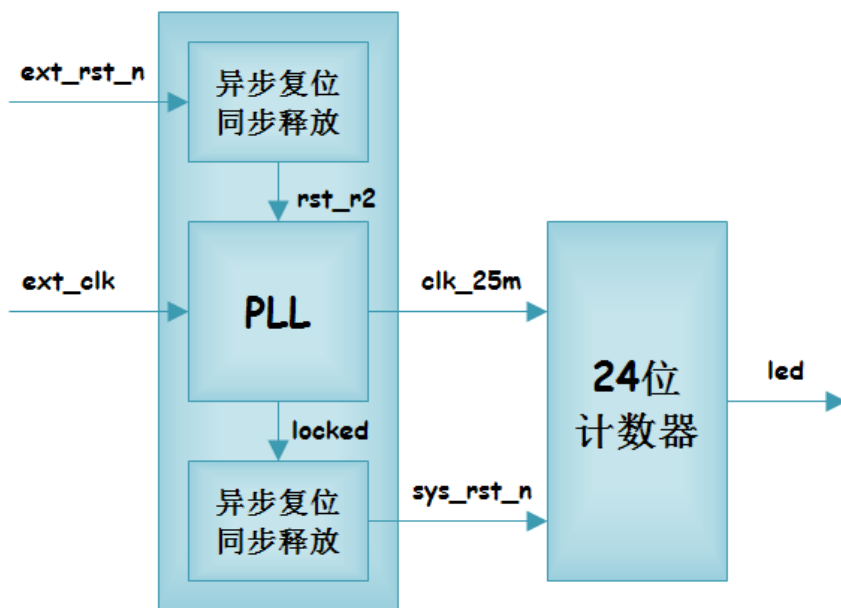


图 1 工程实例 1 功能框图

2 新建 Quartus II 工程

本书中的所有例程均基于 Quartus II 13.1 版本软件。

双击电脑桌面上的“Quartus II 13.1 (64-bit)”图标，或者单击“开始→程序→Altera 13.1.0.162 →Quartus II 13.1.0.162”，打开 Quartus II 软件。Quartus II 软件主界面如图 2 所示，第一次打开软件，通常默认由菜单栏、工具栏、工程文件导航窗口、编译流程窗口、主编辑窗口以及各种输出打印窗口组成。



图 2 Quartus II 软件主界面

下面我们要新建一个工程，在这之前建议大家硬盘中专门建立一个文件夹用于存储我们的 Quartus II 工程，这个工程目录的路径名应该只有

字母、数字和下划线，以字母为首字符，且不要包含中文和其他符号。在 Quartus II 的菜单栏上点击“File→New Project Wizard...”，首先弹出了 Introduction 页面，点击 Next 进入 Directory,Name,Top-Level Entity 页面，如图 3 所示。

- 在“**What is the working directory for this project?**”下输入新建工程所在的路径。如本实例工程的存放路径为
“D:\myfpga\DK_SF_HSC\prj\hsc_ex1”。
- 在“**What is the name of this project?**”下输入工程名，如本实例的工程名为“hsc”。
- “**What is the name of the top-level design entity for this project?**”下输入工程顶层设计文件的名称。通常我们建议工程名和工程顶层文件保持一致，如这里统一命名“hsc”。

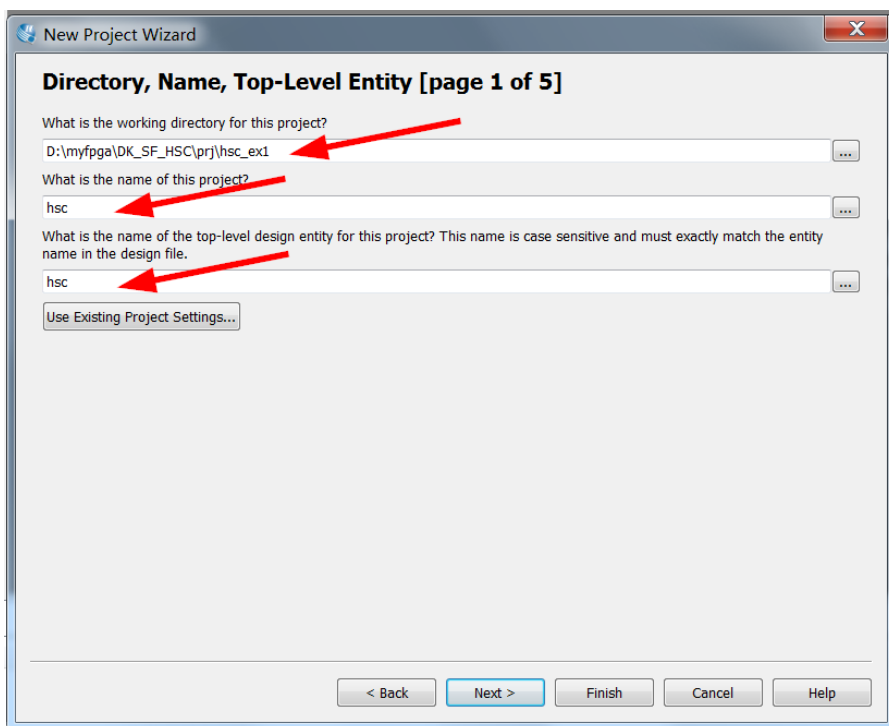


图 3 新建工程向导

设置完毕，点击“Next”。新出现的页面中可以“Add Files”添加已有的工程设计文件（Verilog 或 VHDL 文件），因为我们是完全新建的工程，没有任何预先可用的设计文件，所以不用选择。接着点击“Next”，进入“Family & Device Setting”页面 如图 4 所示。该页面主要是选择器件，我们在“Family”中选择“Cyclone IV E”系列，“Available device”中选择具体型号“E4CE22F17C8”。接着再点击 Next 进入下一个页面。

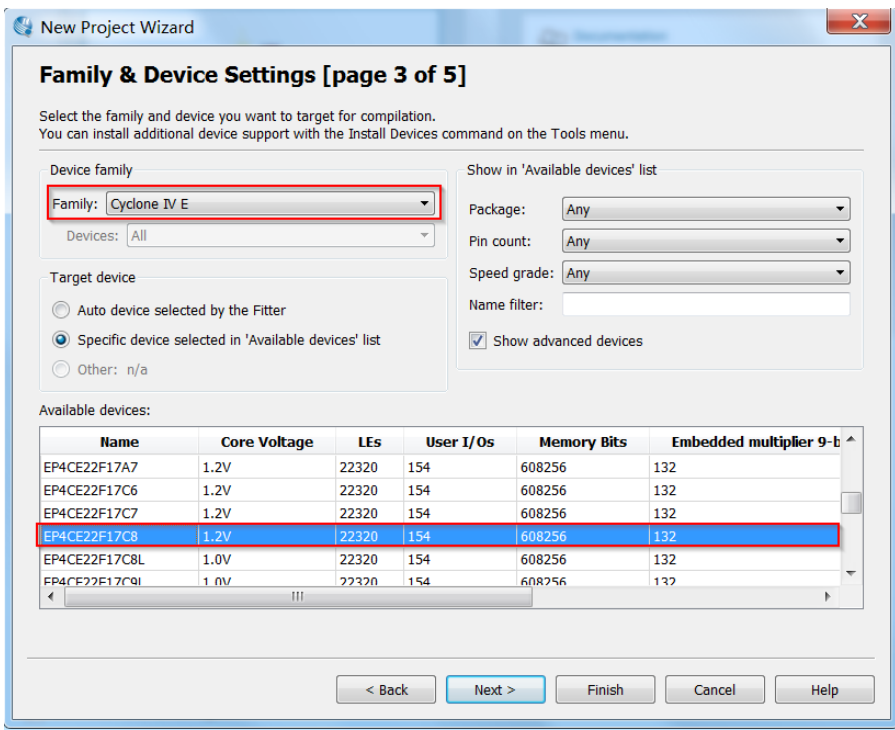


图 4 系列&器件设置页面

如图 5 所示，在 EDA Tool Settings 页面中，可以设置工程各个开发环节中需要用到的第三方（Altera 公司以外）EDA 工具，我们只需要设置“Simulation”工具为“ModelSim-Altera”，Format 为“Verilog HDL”即可，其他工具不涉及，因此都默认为<None>。

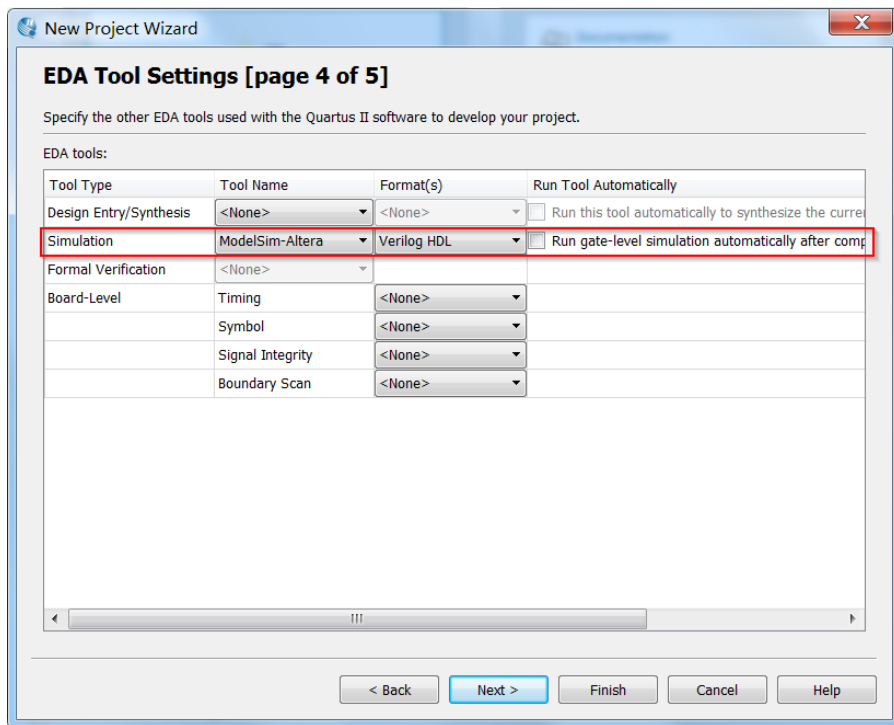


图 5 EDA 工具设置页面

完成这个页面的配置后，我们可以点击“Next”继续进入下一页面查看并核对前面设置的结果，也可以直接点击“Finish”完成工程创建。

工程创建完成后，如图 6 所示，在“Project Navigator”窗口中出现了我们所选择的器件以及顶层文件名，但是实际上此时我们并未创建工程的顶层设计文件，只不过给他命名为了 hsc。我们若双击试图打开 hsc 文件，系统马上会弹出“Can’t find design entity “hsc””的错误提示。

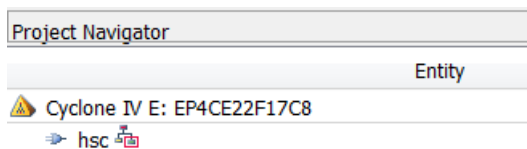


图 6 工程导航窗口

下面我们就来创建工程顶层文件，我们可以点击菜单栏的“File→New...”，然后弹出如图 7 所示的新建文件窗口，在这里我们可以选

择各种需要的设计文件格式。可以作为工程顶层设计文件的格式主要在 Design Files 类别下，我们选择 Verilog HDL File（或者 VHDL File）并单击 OK 完成文件创建。

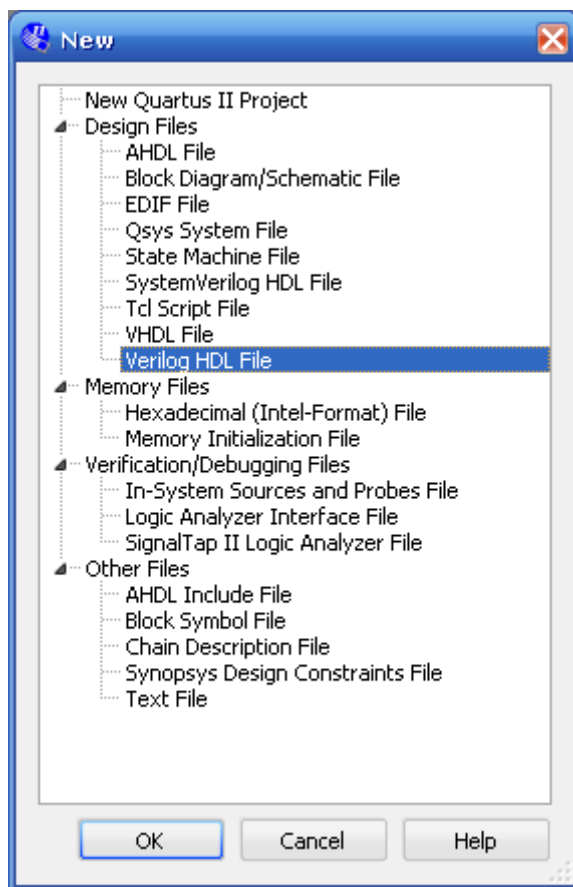


图 7 新建文件窗口

在主编辑窗口中，出现了一个新建的空白可编辑文件，我们接着在该文件中输入实现实验功能的一段 Verilog 代码。Verilog 代码极其详细说明请参看“源码解析”部分。

在这个刚创建的 Verilog 文件中输入代码后，快捷键 Ctrl+S 或点击 File→Save 后则会弹出一个对话框提示输入文件名和保存路径，默认文件名会和我们所命名的 module 名相一致，默认路径也会是当前的工程文件

夹。我们通常也都采用默认设置进行保存即可。

自此，我们的工程创建和设计输入工作已经完成。但是为了验证一下设计输入的代码的基本语法是否正确，可以点击 **Flow → Compilation** 下的“**Analysis & Elaboration**”按钮，如图 8 所示。同时我们可以查看打印窗口的 **Processing** 里的信息，包括各种 **warning** 和 **Error**。**Error** 是不得不关注的，因为 **Error** 意味着我们的代码有语法错误，后续的编译将无法继续；而 **warning** 则不一定是致命的，但很多时候 **warning** 中暗藏玄机，很多潜在的问题都可以从这些条目中找到蛛丝马迹。当然了，也并不是说一个设计编译下来就不可以有 **warning**，如果设计者确认这些 **warning** 符合我们的设计要求，那么可以忽略它。

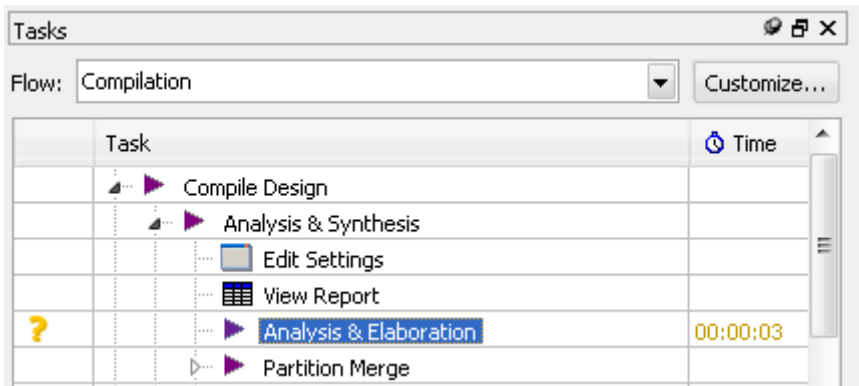


图 8 编译窗口

最后，在 **Analysis & Elaboration** 完成后，通常前面的问号会变成勾号，表示编译通过。

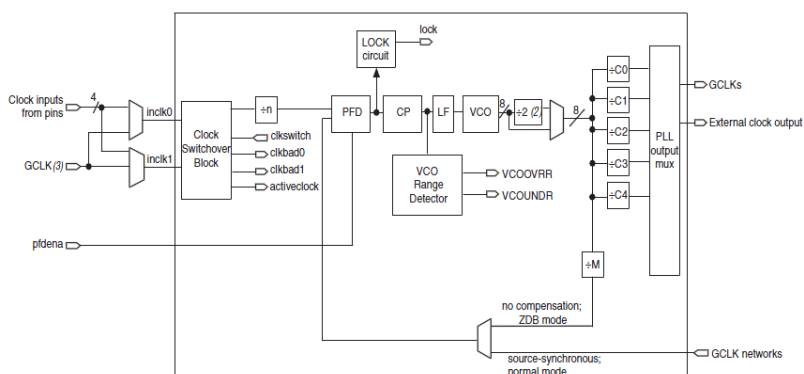
3 IP 核配置——PLL

本实例使用了一个 PLL 的硬核 IP 模块。关于 PLL，这里简单的做些基础扫盲。

PLL(Phase Locked Loop)，即锁相回路或锁相环。PLL 用于振荡器中的反馈技术。许多电子设备要正常工作，通常需要外部的输入信号与内部的振荡信号同步，利用锁相环路就可以实现这个目的。

时钟就是 FPGA 运行的心脏，它的每次跳动必须精准而毫无偏差（当然现实世界中不存在所谓的毫无偏差，但是我们希望它的偏差越小越好）。一个 FPGA 工程中，不同的外设通常工作在不同的时钟频率，所以一个时钟肯定满足不了需求；此外，有时候可能两个不同的模块共用一个时钟频率，但是由于他们运行在不同的工作环境和时序下，所以他们常常是同频不同相(相位)，怎么办？用 PLL 呗。当然了，我们的 FPGA 里面定义的 PLL，可不是仅仅只有一个反馈调整功能，它还有倍频和分频等功能集成其中。严格一点讲，我觉得这个 PLL 实际上应该算是一个 FPGA 内部的时钟管理模块了。不多说，如图 9 所示，大家看看 PLL 内部的功能框图自己体味体味。

Figure 5-10. Cyclone IV E PLL Block Diagram (Note 1)



Notes to Figure 5-10:

- (1) Each clock source can come from any of the four clock pins located on the same side of the device as the PLL.
- (2) This is the VCO post-scale counter K.
- (3) This input port is fed by a pin-driven dedicated GCLK, or through a clock control block if the clock control block is fed by an output from another PLL or a pin-driven dedicated GCLK. An internally generated global signal cannot drive the PLL.

图 9 PLL 内部功能结构

详细的工作机理请大家参考 Cyclone IV Device Handbook, Volume1 的 Chapter 5 的内容。

Cyclone IV 的 PLL 输入一个时钟信号，最多可以产生 5 个输出时钟信号，输出的频率和相位都可以在一定的范围内调整。

下面我们来看本实例如何配置一个 PLL 硬核 IP，并将其集成到工程中。如图 10 所示，在新建的工程中，点击菜单 “Tools→MegaWizard Plug-In Manager”。

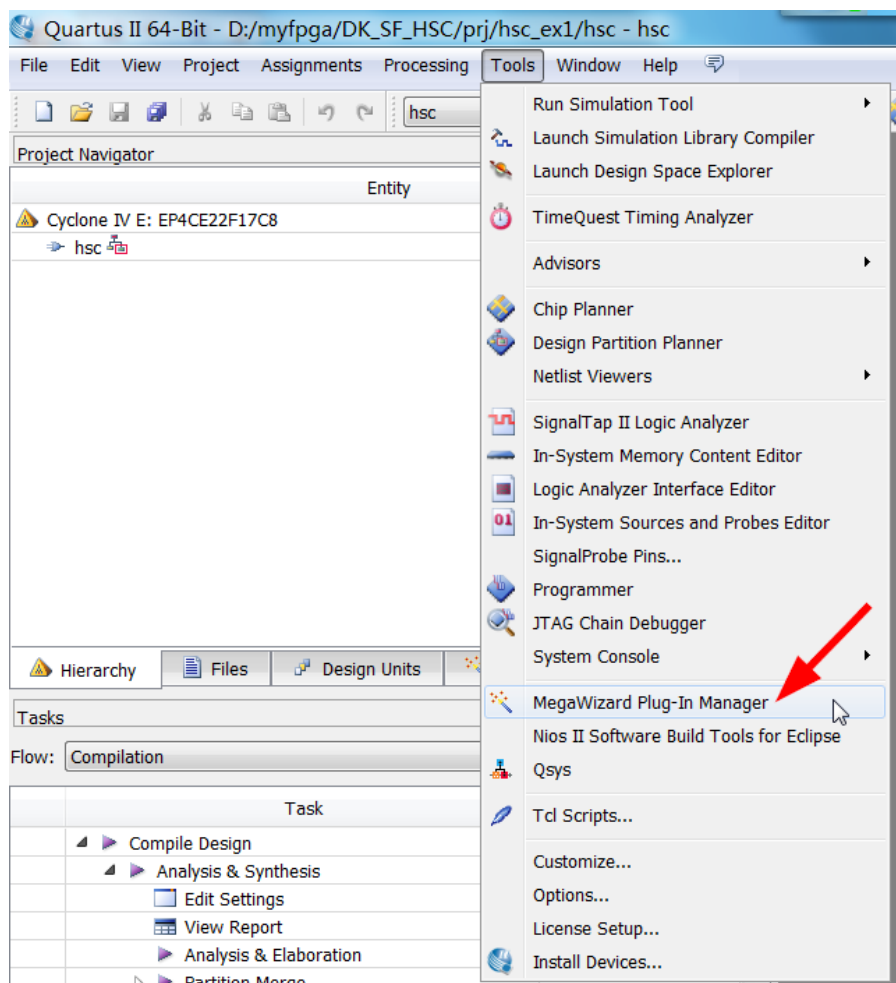


图 10 菜单选择 MegaWizard

如图 11 所示，选择“Creat a new custom megafunction variation”，然后 Next。

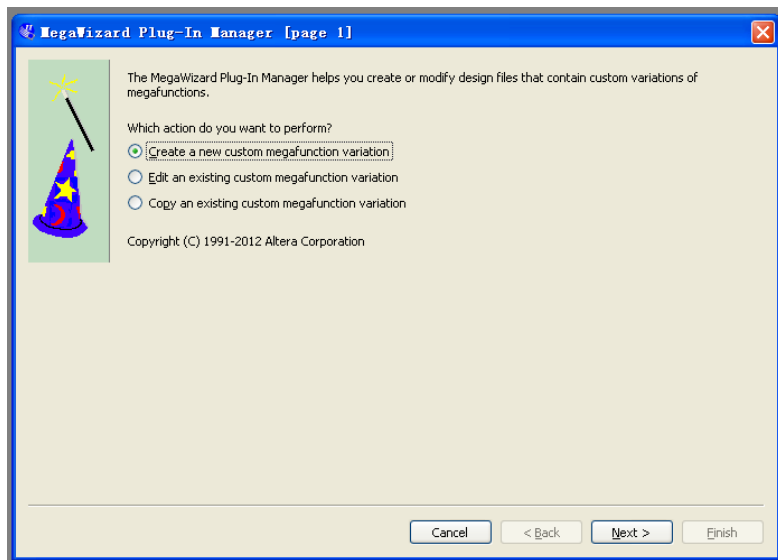


图 11 新建 MegaWizard

接着选择我们所需要的 IP 核，如图 12 所示进行设置。

- 在“Select a megafunction from the list below”下面选择 IP 核为“I/O → ALTPLL”。
- 在“What device family will you be using”后面的下拉栏中选择我们使用的器件系列为“Cyclone IV E”。
- 在“What type of output file do you want to create?”下面选择语言为“Verilog”。
- 在“What name do you want for the output file?”下面输入工程所在的路径，并且在最后面加上一个名称，这个名称是我们现在正在例化的 PLL 模块的名称，我们可以给他起名叫 `pll_controller`，然后点击 **Next** 进入下一个页面。

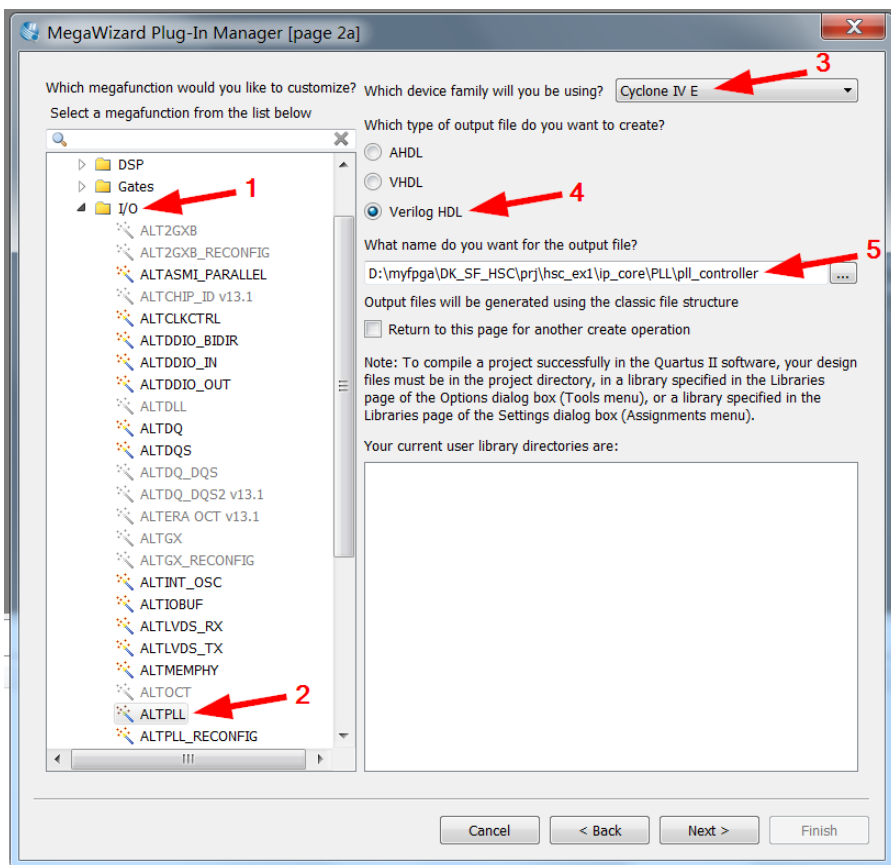


图 12 选择 ALTPLL IP 核

接着来到了 PLL 的参数配置页面，做如图 13 所示的设置。然后点击 Next 进入下一个页面。

- 在 “What device speed grade will you be using?” 后面选择 “8”，即我们使用的器件的速度等级。
- 在 “What is the frequency of the inclk0 input?” 后面选择 “25MHz”，即我们输入到该 PLL 的基准时钟频率。

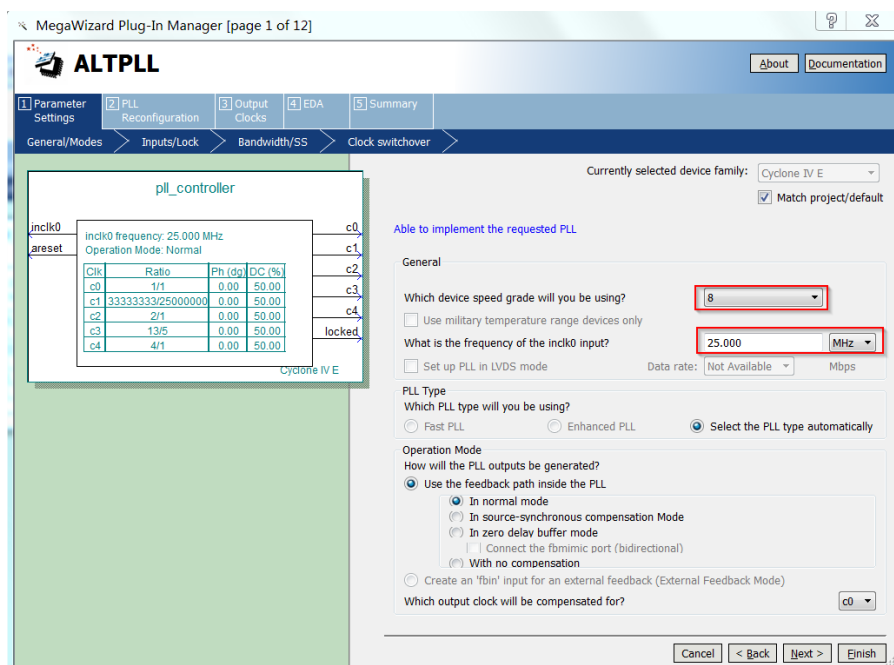


图 13 PLL 参数设置

Input/lock 页面中，如图 14 所示进行设置，接着点击 Next 进入下一个页面。

- 勾选 “Create an ‘areset’ input to asynchronously reset the PLL”，即引出该 PLL 硬核的‘areset’信号，这是该 PLL 硬核的异步复位信号，高电平有效。
- 勾选 “Create ‘locked’ output”，即引出该 PLL 硬核的‘locked’信号，该信号用于指示 PLL 是否完成内部初始化，已经可以正常输出了高电平有效。

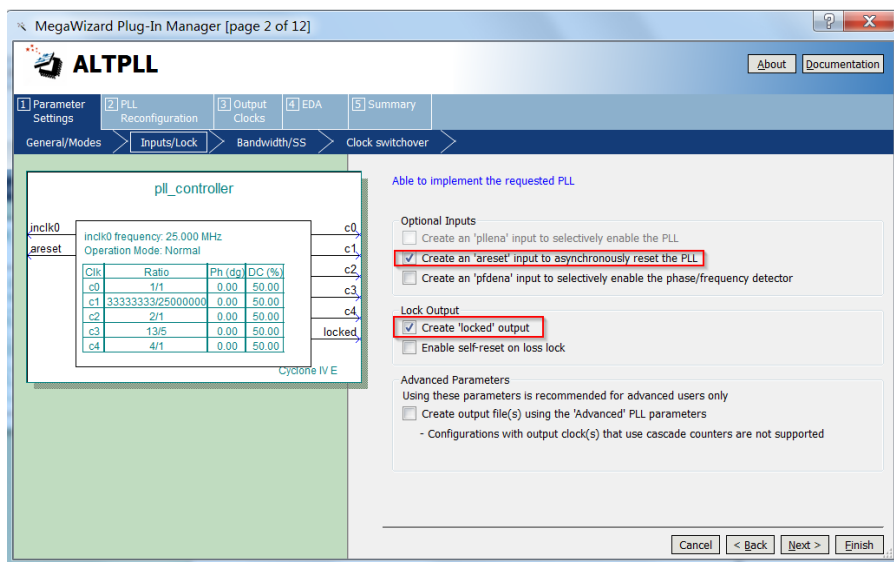


图 14 PLL 复位和锁定信号设置

Bandwidth/SS、Clock Switchover 和 PLL Reconfiguration 页面不用设置，默认即可。直接进入 Output Clocks 页面，如图 15 所示，这里有 5 个可选的时钟输出通道，通过勾选对应通道下方的 Use this clock 选项开启对应的时钟输出通道。可以在配置页面中设置输出时钟的频率、相位和占空比。这里是 C0 通道的设置。

- 勾选 “Use this clock”，表示使用该时钟输出信号。
- 输入 “Enter output clock frequency” 为 “25MHz”，表示该通道输出的时钟频率为 25MHz。
- 输入 “Clock phase shift” 为 “0 deg”，表示该通道输出的时钟相位为 0 deg。
- 输入 “Clock duty cycle(%)” 为 “50.00%”，表示该通道输出的时钟占空比为 50%。

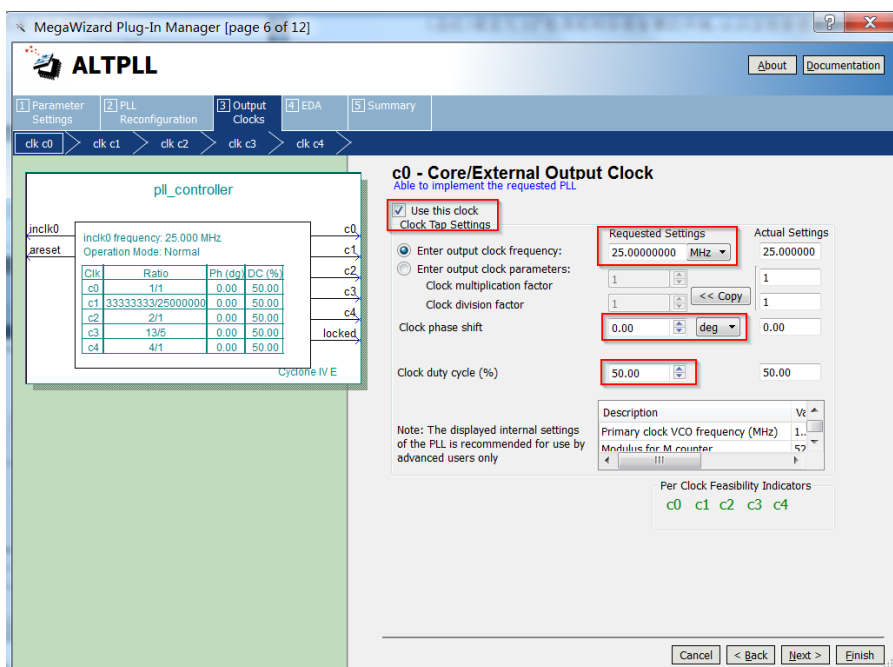


图 15 PLL 输出时钟 C0 设置

和 C0 的配置一样，我们可以分别开启并且配置 C1、C2、C3 和 C4，这些时钟虽然这个例程暂时用不上，但是后续的例程将会使用到。

- C1 的时钟频率为 33.3333MHz，相位为 0deg，占空比为 50%。
- C2 的时钟频率为 50MHz，相位为 0deg，占空比为 50%。
- C3 的时钟频率为 65MHz，相位为 0deg，占空比为 50%。
- C4 的时钟频率为 100MHz，相位为 0deg，占空比为 50%。

配置完成后，最后在 Summary 页面，如图 16 所示，勾选上*_inst.v 文件，这是一个 PLL 例化的模板文件，一会我们可以在工程目录下找到这个文件，然后打开它，将它的代码复制到工程中，修改对应接口即可完成这个 IP 核的集成。

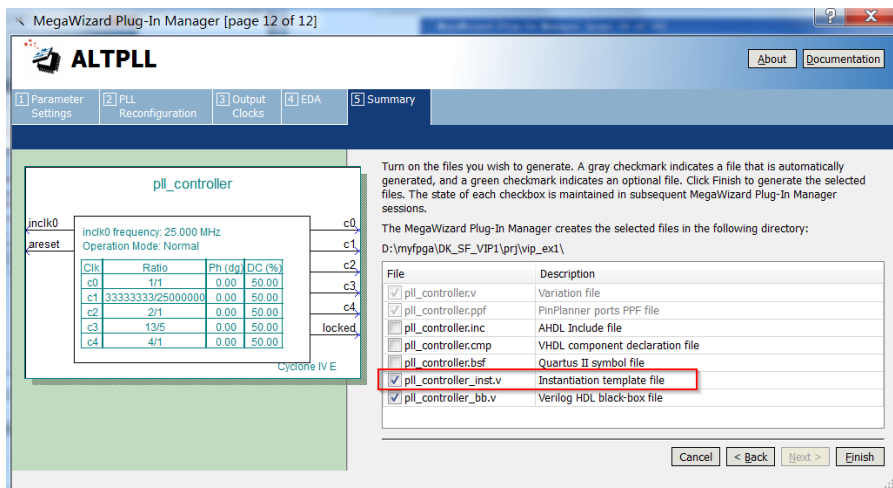


图 16 Summary 配置页面

点击 Finish 完成 PLL 的配置。工程中若弹出如图 17 所示的对话框，勾选 “Automatically add Quartus II IP Files to all projects” 选项后，点击 Yes。

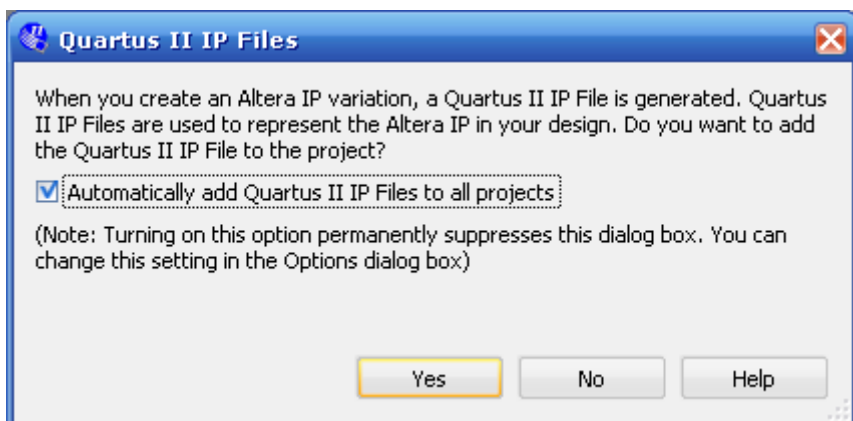


图 17 自动添加文件窗口

4 引脚分配

对于 FPGA 信号与引脚的映射分配，Quartus II 中提供了多种方式，话

说“白猫黑猫，抓到老鼠都是好猫”，所以，也无所谓哪种方式更好，大家习惯就都好。

使用 Pin Planner 进行引脚分配

如图 18 所示，点击菜单栏 “Assignments→Pin Planner”。

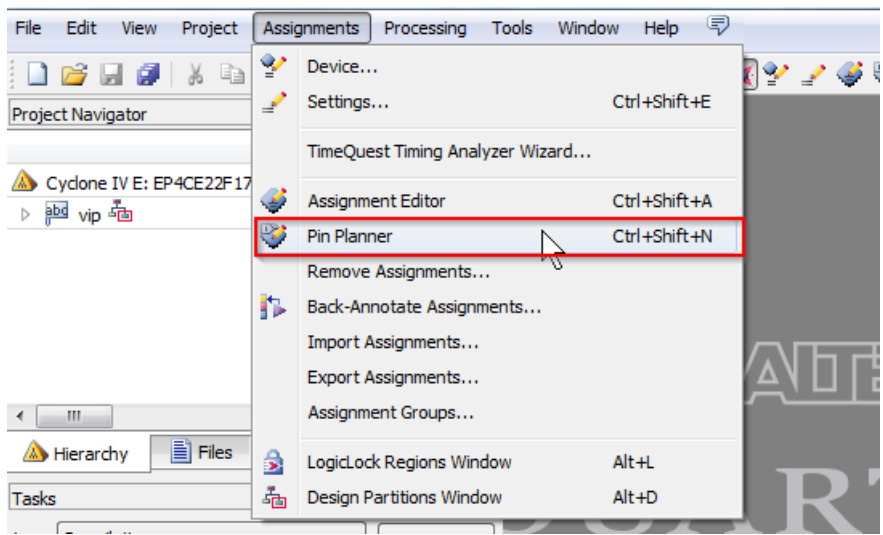


图 18 菜单选择 Pin Planner

在弹出的“Pin Planner”中，找到如图 19 所示的“All Pins”选项卡。
“Node Name”罗列了所有在设计工程的顶层代码中申明过的引脚信号，我们找到它们所对应的“Location”列，这里可以输入或者查找相应的引脚进行分配。



图 19 Pin Planner 界面

如图 20 所示，我们给 ext_clk 对应的 Location 列输入了引脚号“E15”。
这就完成了 ext_clk 信号的引脚分配。

Node Name	Direction	Location	I/O Bank
in ext_clk	Input	PIN_E15	6
in ext_rst_n	Input		
out led	Output		
<<new node>>			

图 20 Pin Planner 引脚分配

引脚分配是基于什么？拍拍脑袋，随心所欲？非也，引脚的分配一定是有理有据，那就是我们已经设计好的硬件原理图。如图 21 所示，这里 FPGA 所使用的时钟信号 ext_clk 对应原理图上的网络名 CLK_25M，它的 FPGA 引脚号是 E15，那么我们就给它分配 E15。

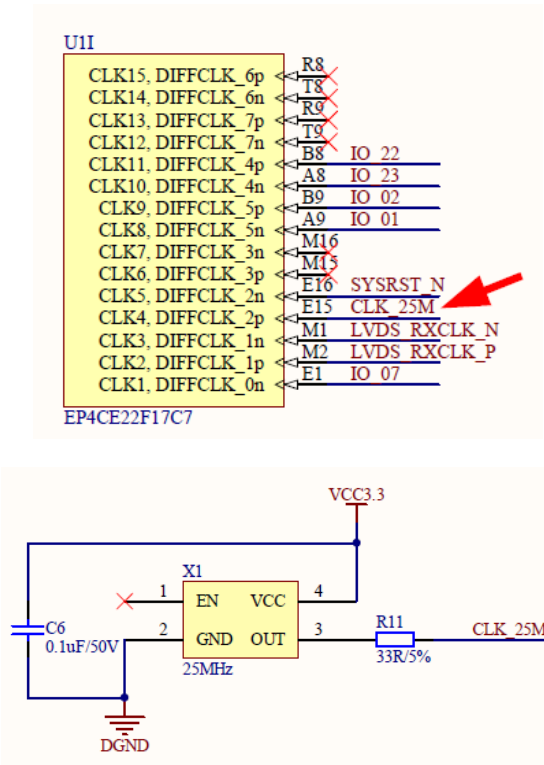


图 21 时钟信号的引脚连接原理图

使用 Tcl Console 进行引脚分配

Tcl (Tool Command Language)，即工具命令语言。是一种好用易学的

编程语言。在 EDA 工具中广泛使用，几乎所有 FPGA 开发工具都支持这种语言进行辅助设计。例如这里我们就要尝试用 tcl 脚本进行 FPGA 的引脚分配。

前面对 ext_clk 的引脚分配，我们可以用如下语句实现。

```
set_location_assignment PIN_E15 -to ext_clk
```

语法 “set_location_assignment PIN_A -to B” 是固定格式，A 代表 FPGA 引脚号，B 代表 FPGA 内部的信号名称。就这么简单。这个脚本要写到哪里？

如图 22 所示，点击菜单栏 “View→Utility Windows→Tcl Console”。

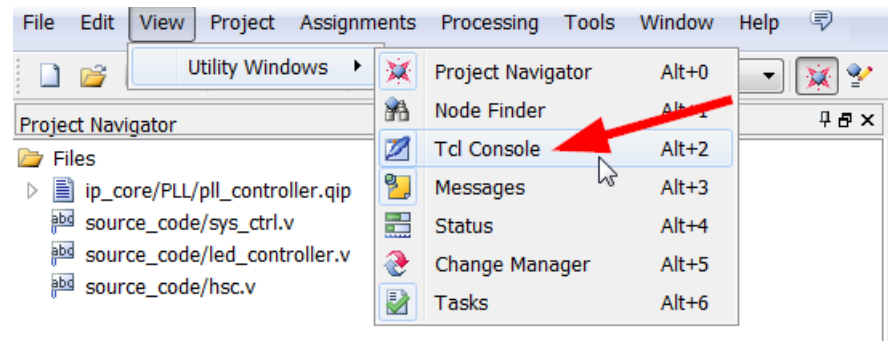


图 22 菜单选择 Tcl Console

接着如图 23 所示，我们在 “Tcl Console” 中输入以下的 3 条引脚分配脚本，最后点击 “回车”。

```
tcl> set_location_assignment PIN_E15 -to ext_clk
tcl> set_location_assignment PIN_E16 -to ext_rst_n
tcl> set_location_assignment PIN_D3 -to led
tcl>
```

图 23 Tcl Console 中进行引脚分配

此时，我们回到 “Pin Planner” 中，如图 24 所示，也可以看到所有引脚自动完成分配。







Named: *  Edit:  				
Node Name	Direction	Location	I/O Bank	VREF Group
 ext_clk	Input	PIN_E15	6	B6_N0
 ext_rst_n	Input	PIN_E16	6	B6_N0
 led	Output	PIN_D3	8	B8_N0
<<new node>>				

图 24 Pin Planner 中引脚分配

使用 TCL Scripts 进行引脚分配

还有一种快速的批量引脚分配方式，首先我们新建一个“*.tcl”为后缀的文件，并且把该文件存放在对应的 Quartus II 工程目录下。然后输入引脚分配的 tcl 脚本，如图 25 所示。

```
hsc_pin_assignment.tcl
1
2 #####
3 #时钟和复位接口
4 set_location_assignment PIN_E15 -to ext_clk
5 set_location_assignment PIN_E16 -to ext_rst_n
6
7 #####
8 #LED指示灯接口
9 set_location_assignment PIN_D3 -to led
10
```

图 25 批量引脚分配脚本

接着如图 26 所示，点击菜单栏“Tools→Tcl Scripts...”。

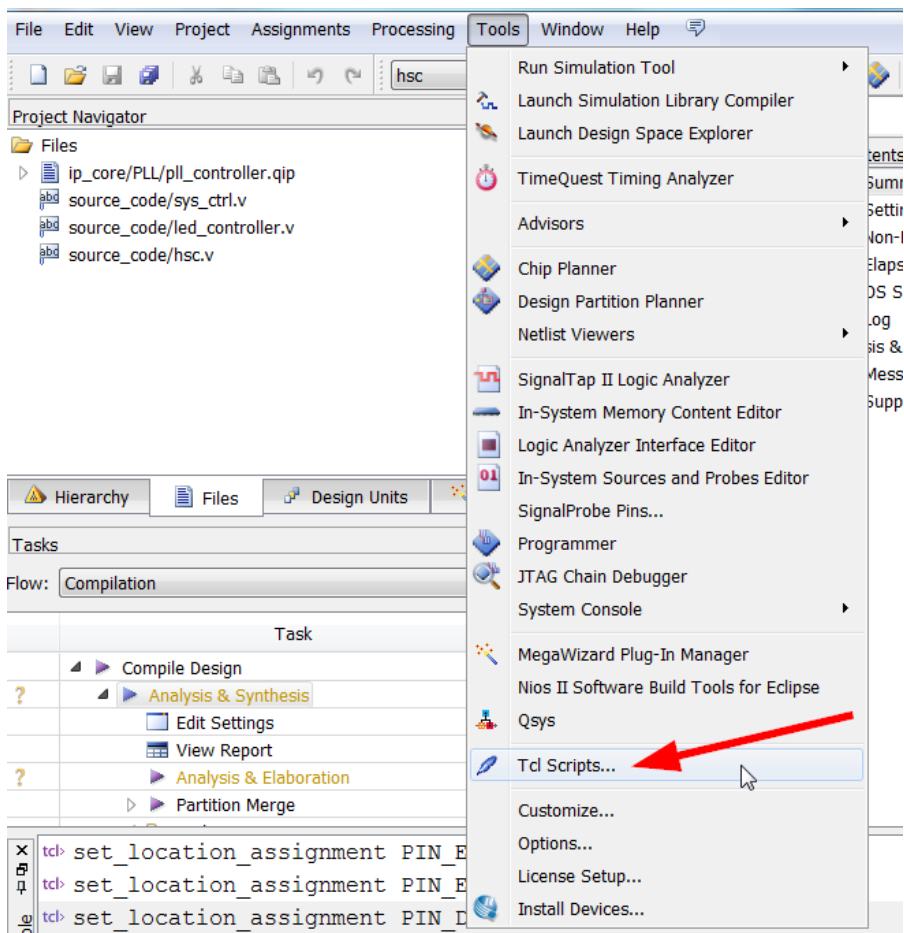


图 26 菜单选择 Tcl Scripts

弹出的“TCL Scripts”界面如图 27 所示。我们可以选择“Librarys→Project”下面所对应的我们刚刚创建的“hsc_pin_assignment.tcl”文件，对应的“Preview”下面就出现了文件中所有脚本的预览。

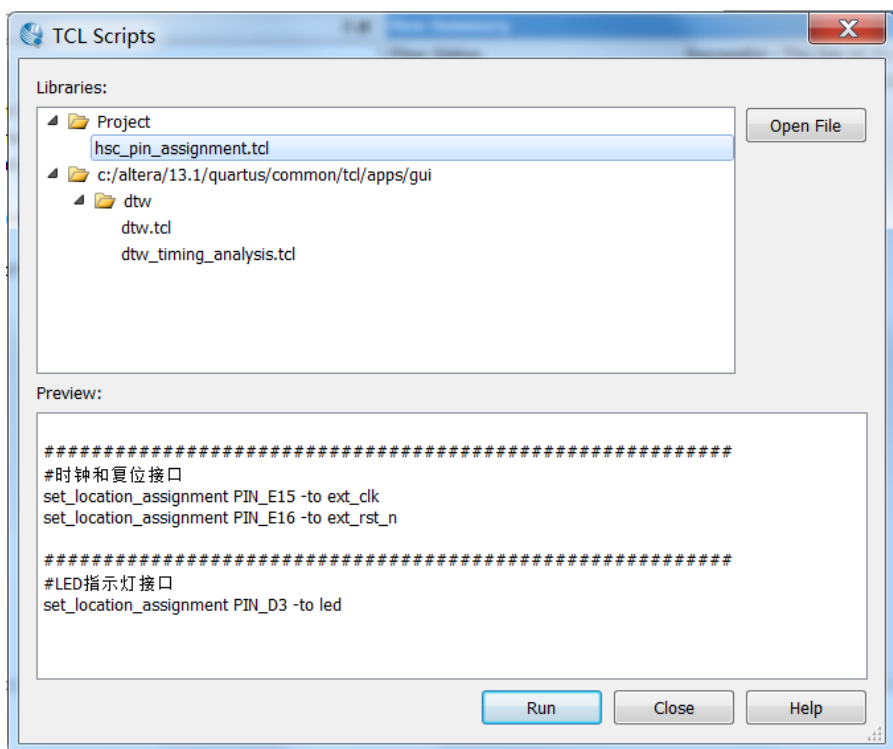


图 27 Tcl Scripts 界面

点击“TCL Scripts”界面右下角的“Run”按钮，则完成引脚分配，弹出如图 28 所示的提示窗口。

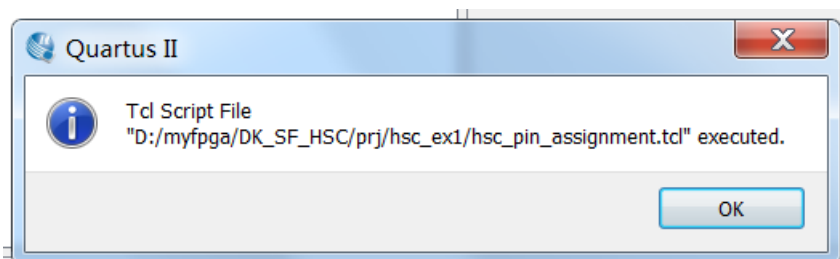


图 28 tcl 脚本执行完成提示

回到“Pin Planner”中，如图 29 所示，同样可以看到所有引脚自动完成分配。

	Node Name	Direction	Location
in	ext_clk	Input	PIN_E15
in	ext_rst_n	Input	PIN_E16
out	led	Output	PIN_J13
	<<new node>>		

图 29 Pin Planner 中引脚分配

话说“条条大路通罗马”，这三种引脚分配方式都很实用，大家可以根据自己的喜好或习惯选择。我们的建议是，第一次对工程引脚做分配，不妨选择“Pin Planner”进行分配；而今后需要重复的对一样信号名的引脚做分配时，可以考虑用后面的 tcl 方式，只要简单的复制前面做过的引脚分配脚本即可。

5 闲置引脚设置

在 FPGA 使用中，我们常常会遇到一些闲置不使用引脚带来的麻烦。对于这些闲置不使用的引脚，最好在工具软件上将他们设置为“输入三态”，这样可以避免很多稀奇古怪的问题。下面简单的说一下如何在 Quartus II 工程中做这个设置。

如图 30 所示，点击菜单栏的“Assignment→Device...”。

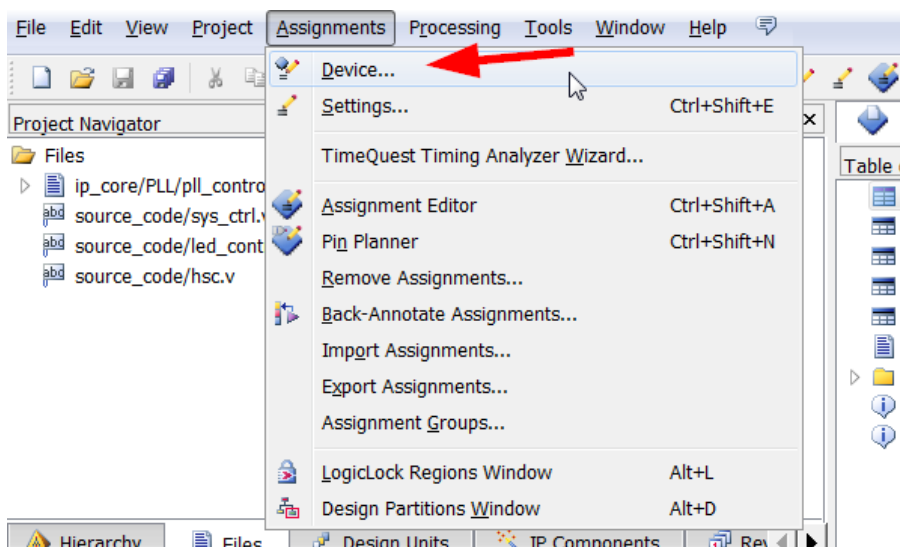


图 30 菜单选择 Device

如图 31 所示，在打开的 Device 页面中，点击“Device and Pin Options...”选项。

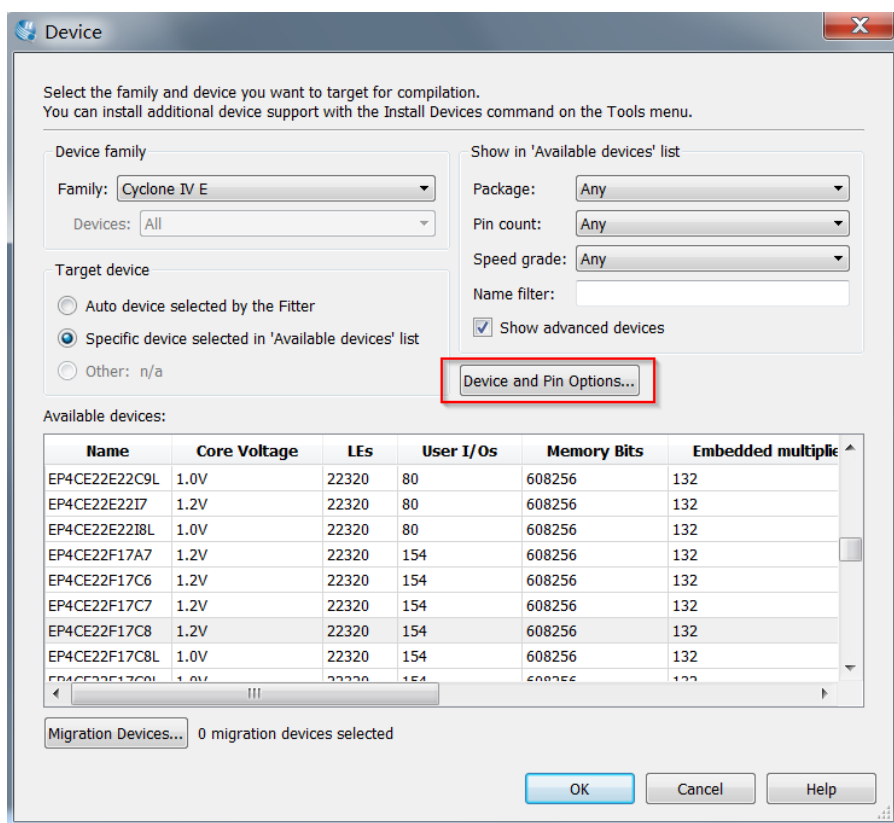


图 31 Device 选项卡

如图 32 所示，在弹出的窗口中，选择“Unused Pins”选项卡，然后在“Reserve all unused pins”后面的下拉框中选择“As input tri-stated”即可。

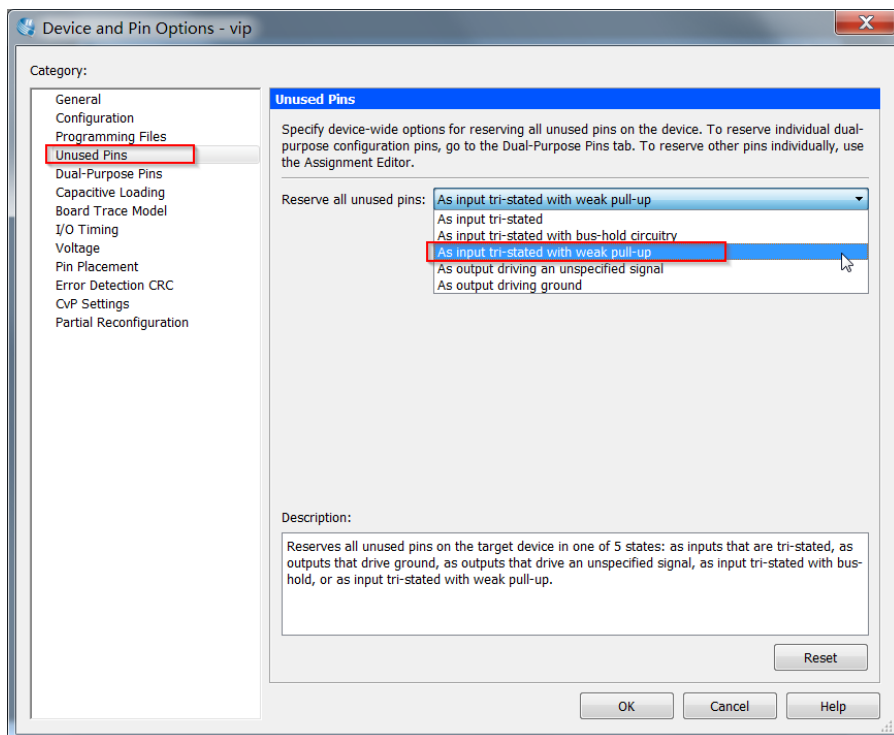


图 32 Device and Pin Options 选项卡

完成设定后，重新编译 Quartus II 的工程，则设置生效。

6 Verilog 代码解析

本实例有 4 个模块，3 个层级。其层次结构如图 33 所示。

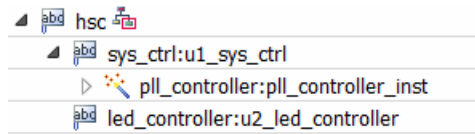


图 33 工程实例 1 代码层次图

- hsc.v 是顶层模块，其下例化了两个子模块，即 sys_ctrl.v 模块和 led_controller.v 模块。该模块仅仅用于子模块间的接口连接，以及和 FPGA 外部的接口定义，该模块中未作任何的逻辑处理。
- sys_ctrl.v 二级子模块中例化了 PLL 模块，并且对输入 PLL 的复位信号以及 PLL 锁定后的复位信号进行“异步复位，同步释放”的处理，确保系统的复位信号稳定可靠。
- pll_controller.v 三级子模块为 FPGA 器件特有的 PLL IP 硬核模块，其主要功能是产生多个特定输入时钟的分频、倍频、相位调整后的输出时钟信号。
- led_controller.v 二级子模块进行 24 位计数器的循环计数，产生分频信号用于实现 LED 指示灯的闪烁。

vip.v 模块代码解析

顶层模块的代码如下。

```
module vip(
    ext_clk, ext_rst_n,
    led
);

input ext_clk;        //外部 25MHz 输入时钟
input ext_rst_n;      //外部低电平复位信号输入
                      //LED 指示灯接口
output led;           //用于测试的 LED 指示灯
```

```
////////////////////////////////////
```

```
//系统内部时钟和复位产生模块例化
```

```
    //PLL 输出复位和时钟，用于FPGA 内部系统
```

```
wire sys_rst_n; //系统复位信号，低电平有效
```

```
wire clk_25m;      //PLL 输出 25MHz
```

```
wire clk_33m;      //PLL 输出 33MHz
```

```
wire clk_50m;      //PLL 输出 50MHz
```

```
wire clk_65m;      //PLL 输出 65MHz
```

```
wire clk_100m;     //PLL 输出 100MHz
```

```
sys_ctrl    uut_sys_ctrl(  
    .ext_clk(ext_clk),  
    .ext_rst_n(ext_rst_n),  
    .sys_rst_n(sys_rst_n),  
    .clk_25m(clk_25m),  
    .clk_33m(clk_33m),  
    .clk_50m(clk_50m),  
    .clk_65m(clk_65m),  
    .clk_100m(clk_100m)  
);
```

```
////////////////////////////////////
```

```
//LED 闪烁逻辑产生模块例化
```

```
led_controller    uut_led_controller(  
    .clk(clk_25m),  
    .rst_n(sys_rst_n),  
    .led(led)  
);
```

```
Endmodule
```

① 对于一个 Verilog 语言编写的 FPGA 模块来说，下面是一个很典型的模板。

```
module vip(  
    ext_clk, ext_rst_n, led  
);  
input ext_clk;  
input ext_rst_n;  
output led;  
  
.....  
  
endmodule
```

- “module endmodule”是固定语法，表示这是一个 FPGA 模块。
- 紧接着“module”之后的“vip”则是 FPGA 模块的名称，通常和文件名一致。
- 名称“vip”之后的“()”内则将罗列出所有该模块需要引出的输入、输出或双向信号。对于 FPGA 工程的顶层模块而言，这些罗列的信号就是 FPGA 和外部芯片之间的接口信号；而对于非顶层模块的子模块而言，这些罗列的信号则是该模块和其他 FPGA 子模块或者将要连接到 FPGA 外部信号接口的信号。
- 对于“()”内的信号，随后都需要申明它们的信号类型，是输入“input”、输出“output”还是双向口“inout”。在本实例中，ext_clk 是 FPGA 外部引脚连接的 25MHz 时钟信号；ext_rst_n 是外部阻容复位电路连接的低电平有效复位信号；led 则是输出到 FPGA 引脚连接着的 LED 指示灯开关信号。

- “……”部分则是 **FPGA** 内部的具体功能代码，若这个顶层模块，这部分是接口信号的申明和模块的例化。

② 如下的一段代码是对 **sys_ctrl.v** 模块的例化，所谓“例化”，就有点像 **C** 语言设计中的“函数调用”，“函数”内外的接口信号必须完全“对口”。

```
sys_ctrl    uut_sys_ctrl(  
    .ext_clk(ext_clk),  
    .ext_rst_n(ext_rst_n),  
    .sys_rst_n(sys_rst_n),  
    .clk_25m(clk_25m),  
    .clk_33m(clk_33m),  
    .clk_50m(clk_50m),  
    .clk_65m(clk_65m),  
    .clk_100m(clk_100m)  
);
```

- **sys_ctrl** 表示我们需要例化的模块名称，必须是已经实际存在的模块。
- **uut_sys_ctrl** 则是我们给这个例化模块取的名称，这个名称可以随意取。它的用意在于，如果一个设计中多次用到同样的模块，那么为了区分它们，就在这个名称上做点“文章”，取个不同的名称即可。
- “()”内则是接口的映射。例如“**.ext_clk(ext_clk)**,”是一个固定的格式，第一个“**ext_clk**”表示例化模块内部的信号名称，“()”内的“**ext_clk**”则表示例化模块外部，对于该设计，就是当前顶层模块中需要连接的信号名称。

③ 前面提到过，对于模块例化的接口，一类是直接连接到 **FPGA** 外部引脚上的信号，另一类是 **FPGA** 内部各个模块之间需要相互连接的信号。

- 该顶层模块中，对于连接到 **FPGA** 外部引脚上的信号，我们直接在

顶层模块的引脚列表中声明了，如 `ext_clk`、`ext_rst_n` 和 `led` 信号。

- 而如下所示的 **FPGA** 内部各个模块之间需要相互连接的信号，我们则需要进行申明，它们必须被定义为 “**wire**” 类型，而绝对不可以是 “**reg**” 类型。在该工程中，其实只用到了下面定义的 “`sys_rst_n`” 和 “`clk_25m`” 两个信号，其他信号虽然定义引出，但保留不用。

```
wire sys_rst_n; //系统复位信号，低电平有效
wire clk_25m;   //PLL 输出 25MHz
wire clk_33m;   //PLL 输出 33MHz
wire clk_50m;   //PLL 输出 50MHz
wire clk_65m;   //PLL 输出 65MHz
wire clk_100m;  //PLL 输出 100MHz
```

sys_ctrl.v 模块代码解析

`sys_ctrl.v` 模块的代码如下。该模块例化了 IP 核 PLL，对复位信号做 “异步复位、同步释放” 的处理。

```
module sys_ctrl(
    ext_clk, ext_rst_n,
    sys_rst_n, clk_25m, clk_33m, clk_50m, clk_65m, clk_100m
);
    //FPGA 外部输入时钟和复位
input ext_clk;           //外部 25MHz 输入时钟
input ext_rst_n;         //外部低电平复位信号输入
    //PLL 输出复位和时钟，用于 FPGA 内部系统
output reg sys_rst_n;    //系统复位信号，低电平有效
output clk_25m;          //PLL 输出 25MHz
output clk_33m;          //PLL 输出 33MHz
output clk_50m;          //PLL 输出 50MHz
output clk_65m;          //PLL 输出 65MHz
output clk_100m;         //PLL 输出 100MHz
```



```

////////////////////////////////////
//PLL 复位信号产生，高有效
//异步复位，同步释放

reg rst_r1,rst_r2;

always @(posedge ext_clk or negedge ext_rst_n)
    if(!ext_rst_n) rst_r1 <= 1'b0;
    else rst_r1 <= 1'b1;

always @(posedge ext_clk or negedge ext_rst_n)
    if(!ext_rst_n) rst_r2 <= 1'b0;
    else rst_r2 <= rst_r1;

////////////////////////////////////
//PLL 模块例化
wire locked;    //PLL 输出锁定状态，高电平有效

pll_controller  pll_controller_inst (
    .areset ( !rst_r2 ),
    .inclk0 ( ext_clk ),
    .c0 ( clk_25m ),
    .c1 ( clk_33m ),
    .c2 ( clk_50m ),
    .c3 ( clk_65m ),
    .c4 ( clk_100m ),
    .locked ( locked )
);

//-----
//系统复位处理逻辑

```

```

reg sys_rst_nr;

always @(posedge clk_100m)
    if(!locked) sys_rst_nr <= 1'b0;
    else sys_rst_nr <= 1'b1;

always @(posedge clk_100m or negedge sys_rst_nr)
    if(!sys_rst_nr) sys_rst_n <= 1'b0;
    else sys_rst_n <= sys_rst_nr;

endmodule

```

至于为什么需要做“异步复位，同步释放”的处理，这和异步信号容易出现“亚稳态”很有关系。那么，我们不妨先从复位信号出现“亚稳态”的问题分析入手来讨论这个问题。

如图 34 所示，在带有复位端的 D 触发器中，当 **clr** 信号“复位”有效时，它可以直接驱动最后一级的与非门，令输出 **q** 信号“异步”置位为“1”或“0”。这就是异步复位。当这个复位信号释放时，**q** 的输出由前一级的内部输出决定。然而，由于复位信号不仅直接作用于最后一级门电路，而且也会作为前级电路的一个输入信号，因此这个前一级的内部输出也受到复位信号的影响。前一级的内部电路实际上是实现了一个“保持”的功能，即在时钟沿跳变附近锁住当时的输入值，使得在时钟变为高电平时不再受输入信号的影响。

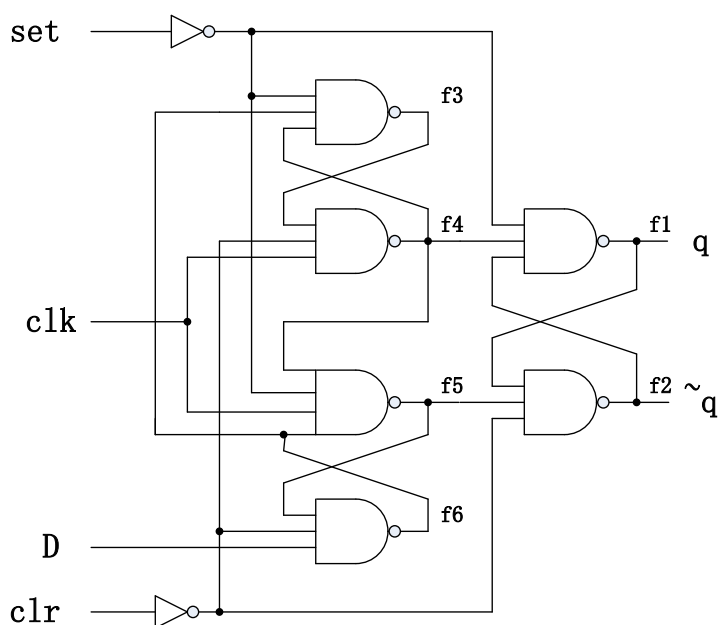


图 34 寄存器内部结构

对于这一个“维持”电路，在时钟沿变化附近，如果“clr”信号有效，那么，就会锁存住“clr”的值；如果 clr 信号释放，那么这个“维持”电路会去锁当时的 D 输入端的数据。因此，如果 clr 信号的“释放”发生在靠时钟沿很近的时间点，那么这个“维持”电路就可能既没有足够时间“维持”住 clr 值，也没有足够时间“维持”住 D 输入端的值，因此造成亚稳态，并通过最后一级与非门传到 q 端输出。

对于一般的复位信号，如果不做任何处理，如图 35 所示，FPGA 外部输入的复位信号 ext_rst_n 将直接连接到寄存器的 clr 端口，由于 ext_rst_n 和 FPGA 内部寄存器之间的“异步”性，就导致了前面提到的“亚稳态”极容易出现，而一个 FPGA 设计中复位信号几乎“无处不在”，这种高扇出信号的大面积“异步”很可能引起系统复位过程中出现各种不确定信号状态。对于某些应用而言，这是很危险的，甚至有让系统奔溃的可能。

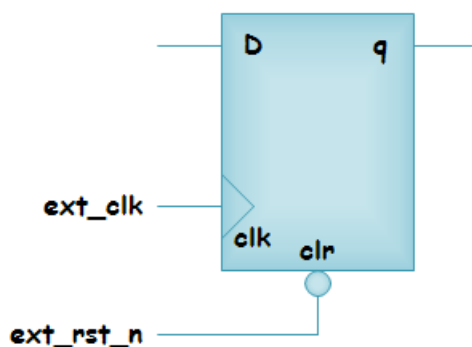


图 35 寄存器时钟和复位连接

而为了避免“亚稳态”的出现，需要做“异步复位，同步释放”的处理。具体如何实现呢？看我们的代码。

```

reg rst_r1,rst_r2;

always @(posedge ext_clk or negedge ext_rst_n)
    if(!ext_rst_n) rst_r1 <= 1'b0;
    else rst_r1 <= 1'b1;

always @(posedge ext_clk or negedge ext_rst_n)
    if(!ext_rst_n) rst_r2 <= 1'b0;
    else rst_r2 <= rst_r1;
  
```

这段代码实现的功能如图 37 所示。最后输出的复位信号 `rst_r2` 就是和时钟信号 `ext_clk` 完全“同步”的复位信号了。当然了，可能大家仍然会有疑问，`ext_rst_n` 和这两个寄存器之间仍然是异步的，那么后级寄存器的输出 `rst_r2` 不是仍然有可能受异步复位的影响吗？很对，按照前面的分析，这前后两级寄存器在 `ext_rst_n` 进入复位的那一刻，确实存在“异步”的问题。但是，在 `ext_rst_n` 退出复位的那一刻，后级寄存器的输出是不受它影响的，它的输出只取决于上一级寄存器的输出 `rst_r1`。这就是我们所说的“同步释放”。

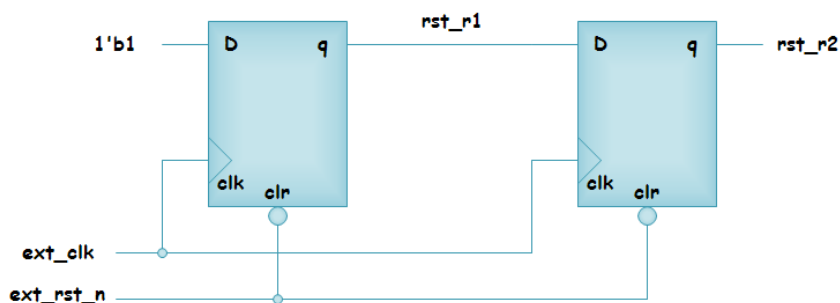


图 36 “同步复位，异步释放”寄存器电路

有喜欢追根问底的同学一定也注意了，在后级寄存器的代码中，不是还有 `ext_rst_n` 信号是否低电平的判断吗？是的，但是可以这样来看，在复位后，`rst_r2` 就处于低电平状态了，而在 `ext_rst_n` 拉高的一瞬间，如果它不“踩”在 `ext_clk` 的建立和保持时间范围内，问题不大，但是我们说它是“异步”的，所以“踩”的概率还是存在的，使用我们的这个后级寄存器设计，就算不幸“踩”中了，那么也无非两种情况，要么“`rst_r2 <= 1'b0`”，要么“`rst_r2 <= rst_r1`”，因为 `rst_r1` 此时也已经拉低了，所以无论哪种情况，`rst_r2` 的结果都会保持低电平。基于此，我们的后端寄存器输出 `rst_r2` 就完全可以不用顾忌 `ext_rst_n` 的“异步”撤销了。

```
if(!ext_rst_n) rst_r2 <= 1'b0;
```

前面的“异步复位，同步释放”是针对刚刚送入 FPGA 的外部时钟和复位信号，而它们都需要经过 PLL 硬核模块的“洗礼”，对于 PLL 工作后输出的信号，其实存在同样的“异步”问题，因此也同样需要做同样的处理。

```
reg sys_rst_nr;

always @(posedge clk_100m)
    if(!locked) sys_rst_nr <= 1'b0;
    else sys_rst_nr <= 1'b1;
```

```
always @(posedge clk_100m or negedge sys_rst_nr)
    if(!sys_rst_nr) sys_rst_n <= 1'b0;
    else sys_rst_n <= sys_rst_nr;
```

单纯从逻辑上看，似乎这段代码和前面的处理不太一样，至少格式不一样，但是大家可以自己仔细分析一下，是“换汤不换药”的。Locked 这个信号是 PLL 正常工作的指示信号，高电平表示 PLL 开始正常工作且输出可用时钟了，低电平则不然，因此它可以作为我们的复位信号。

再来看 PLL 复位这部分的代码，其实这是一个模块的例化，但是在这之前，有很多 IP 核的配置工作需要完成，这在“IP 核配置”部分我们再做详细解析。

```
pll_controller pll_controller_inst (
    .areset ( !rst_r2 ),
    .inclk0 ( ext_clk ),
    .c0 ( clk_25m ),
    .c1 ( clk_33m ),
    .c2 ( clk_50m ),
    .c3 ( clk_65m ),
    .c4 ( clk_100m ),
    .locked ( locked )
);
```

led_controller.v 模块代码解析

led_controller.v 模块的代码如下。

```
module led_controller(
    clk,rst_n,
    led
);
    //时钟和复位接口
input clk;        //25MHz 输入时钟
```

```

input rst_n;    //低电平系统复位信号输入
               //LED 指示灯接口
output led;     //用于测试的 LED 指示灯

//////////////////////////////////////
//计数产生 LED 闪烁频率
reg[23:0] cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 24'd0;
    else cnt <= cnt+1'b1;

assign led = cnt[23];

endmodule

```

这里定义了 24 位的计数器 `cnt`，该计数器正常工作时随着时钟节拍不停的递增计数。有人说，它会不会溢出？当然会，当它计数到最大值 24'hffffff 时，下一个值就“溢出”了，怎么“溢出”，清零呗，就这么简单。不过这也是我们设计期望的逻辑，所以就这么让这个 24 位计数器不停的循环工作着吧。

```

reg[23:0] cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 24'd0;
    else cnt <= cnt+1'b1;

```

再看，我们给 `led` 信号的赋值。计数器的最高位，它其实就是几个 50% 占空比的方波信号，频率则是系统时钟的“2 的 23 次方”分频。

```

assign led = cnt[23];

```

pll_controller.v 模块代码解析

略。关于 PLL 的配置请参考“IP 核配置——PLL”部分内容。

7 板级调试

① 打开“...\prj\hsc_ex1”文件夹下的工程。

② 使用 Programmer 将“...\prj\hsc_ex1\output_files”文件夹下的 vip.sof 文件下载到 HSC 开发板中。这是 JTAG 在线调试模式，下载完成，马上可以看到 HSC 开发板上的 LED 指示灯 D1 不停的闪烁。若将 HSC 核心板掉电，重新上电后 D1 将不再闪烁。

③ 使用 Programmer 将“...\prj\vip_ex1\output_files”文件夹下的 hsc.jic 文件下载到 HSC 开发板中，此时设计工程文件将固化到配置芯片 M25P16 中。重新给 HSC 开发板上电，此时可以看到 HSC 开发板的指示灯 D1 仍然闪烁。