

使用 Verilog 实现 PSoC® 3、PSoC 4 和 PSoC 5LP 的可编程逻辑设计

作者: Vijay Kumar Marrivagu 和 Antonio Rohit De Lima Fernandes

相关器件系列: CY8C3xxx、CY8C5xxx、CY8C42xx

相关代码示例: 无

相关应用笔记: AN81623、AN82156

要想获取本应用笔记的最新版本或相关的项目文件, 请访问以下网址: <http://www.cypress.com/AN82250>。

AN82250 介绍了 PSoC® 3、PSoC 4 和 PSoC 5LP 中 PLD 部分的可编程数字逻辑设计的实现情况。它介绍了 PSoC 通用数字模块 (UDB) 和它们的可编程逻辑器件 (PLD) 子模块。示例项目显示的是通过在 PSoC Creator™ 中创建基于 Verilog 组件来使用设计中的 PLD。

目录

1. 简介	1	关于作者	21
2. PSoC 的 UDB	2	A. 附录 A: PSoC PLD 资源与竞争对手 CPLD 的比较	22
2.1 PSoC UDB 中的 PLD 架构	2	B. 附录 B: 宏单元配置图	23
3. PSoC Creator	5	C. 附录 C: 序列检测器的 Verilog 代码	24
4. 示例项目	6	D. 附录 D: 设计构建完成后的注意事项	27
4.1 创建 Verilog 组件: Counter4Bit	9	D.1 项目报告文件	27
4.2 创建 Verilog 组件: SeqDetector	16	D.2 静态时序分析	27
5. 数据路径与基于 PLD 的设计	20	文档修订记录	28
6. 总结	20	全球销售和设计支持	29
6.1 其他信息	20		
7. 参考文档	21		

1. 简介

PSoC 3、PSoC 4 和 PSoC 5LP (以下简称为 PSoC) 不仅仅是微控制器, 它还可以使用 PSoC 灵活地集成微控制器、复杂可编程逻辑器件 (CPLD), 和高性能模拟等功能。这样可以降低成本、节省电路板空间、电能和开发时间。

注意: 本应用笔记不适用于没有 UDB 的 PSoC 41xx 器件。

本应用笔记介绍了 PSoC 通用数字模块 (UDB) 中的 PLD, 并对创建 PSoC Creator 组件来使用它们进行了相关说明。这是将复杂的可编程逻辑器件 (CPLD) 的功能集成到 PSoC 内的有效的第一步。读完本应用笔记后, 您应该基本掌握了 PSoC PLD, 并且可以使用 PSoC Creator 创建基于 Verilog 的自定义组件。

为了充分利用 PSoC 的数字性能, 下一步是读取 [AN82156 — 使用 PSoC3、PSoC 4 和 PSoC 5LP 的 UDB 数据路径设计 PSoC Creator 组件](#)。

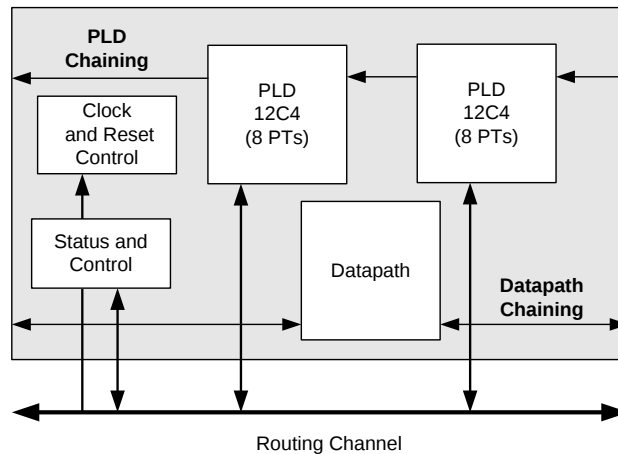
注意: 这是一本高级应用笔记 — 假设您已经熟悉了 PSoC Creator。如果您刚刚接触 PSoC, 那么可以通过参考 [AN54181 — PSoC 3 入门](#)、[AN79953 — PSoC 4 入门](#)和 [AN77759 — PSoC 5LP 入门](#), 更加对该产品的了解。如果您还对 PSoC Creator 还不熟悉, 请参考 [PSoC Creator 主页](#)。

本应用笔记还假设了您已经掌握了数字设计和 Verilog 的基本知识。如果您尚不了解这些概念, 请参考 [AN81623 — PSoC 3 和 PSoC 5LP 数字设计的最佳实践](#)以及 [KBA86336 — PSoC Verilog 基本知识](#)。参考章节列出了与 PSoC 数字设计有关的资源。

2. PSoC 的 UDB

PSoC 通过一个小型、快速、低功耗的数字模块（称为通用数字模块，即 UDB）阵列来实现可编程逻辑。PSoC 器件最多包含 24 个 UDB。如图 1 中所示，UDB 包括两个小型的可编程逻辑器件（PLD）、一个数据路径模块以及一个状态和控制逻辑。

图 1. UDB 框图



顾名思义，可编程逻辑是一个器件系列，它包含 AND（与）、OR（或）、INVERT（反转）和 FLIP-FLOP（触发）等逻辑元素阵列。一般情况下，PLD 是一个电路，可以将它配置为执行特定逻辑功能。

PSoC PLD 可用于构成寄存或组合的“乘积和”逻辑、查询表、复用器以及状态机，并且控制数据路径的操作。更多有关 UDB 数据路径的信息，请参阅 [AN82156](#)。

2.1 PSoC UDB 中的 PLD 架构

同大部分标准 PLD 相似，PSoC PLD 包含了 AND 阵列和其随后的 OR 阵列，这两个阵列均可编程。通常它指的是架构的乘积和。

在 AND 阵列中，一共有 12 个输入，能够驱动八个乘积项（PT）。可以在每个 PT 中选择输入“实值”（T）或“补码”（C）。PT 的输出就是 OR 阵列的输入。OR 门的输出均被馈送到宏单元（MC）。宏单元是带有额外组合逻辑的触发器。

每个 UDB 有两个 PLD；每个 PLD 带有 8 个 PT 和 4 个宏单元，如图 2 所示。PSoc 一共有 48 个 PLD，因此有 192 个宏单元和 384 个 PT。每个 PLD 是相互独立的，并且可以通过进位链路进行连接或被连接到数字系统互联（DSI）。

在附录 A 中，已经对 PSoc PLD 资源与竞争对手大小相似的 PLD 进行了比较。

图 2. PSoc PLD 结构

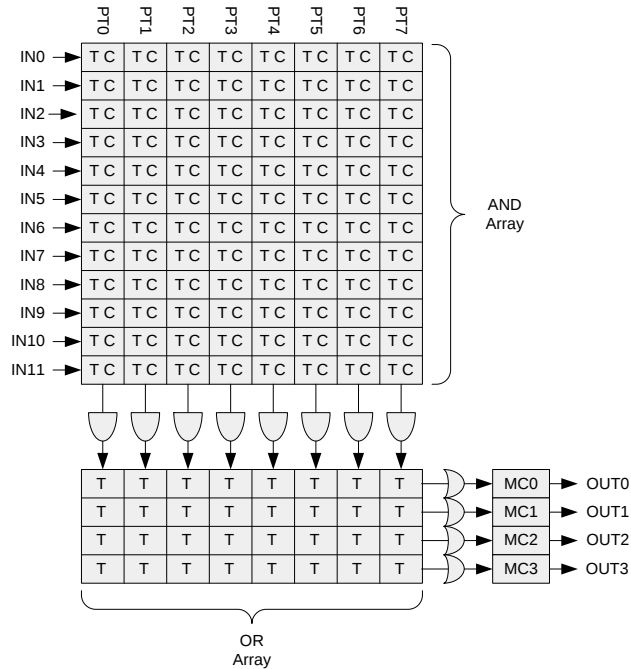
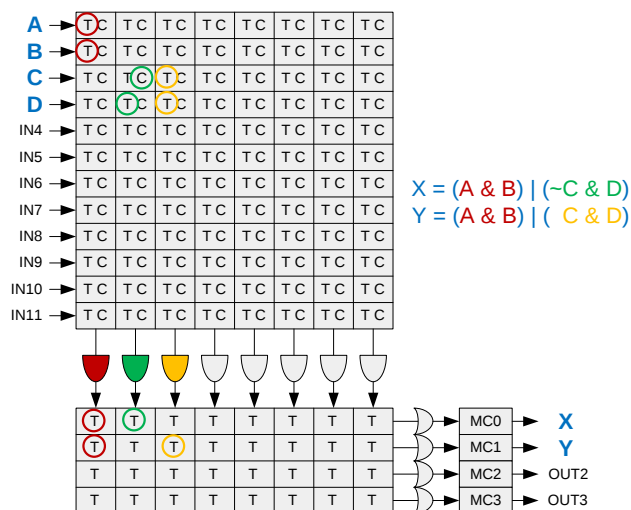


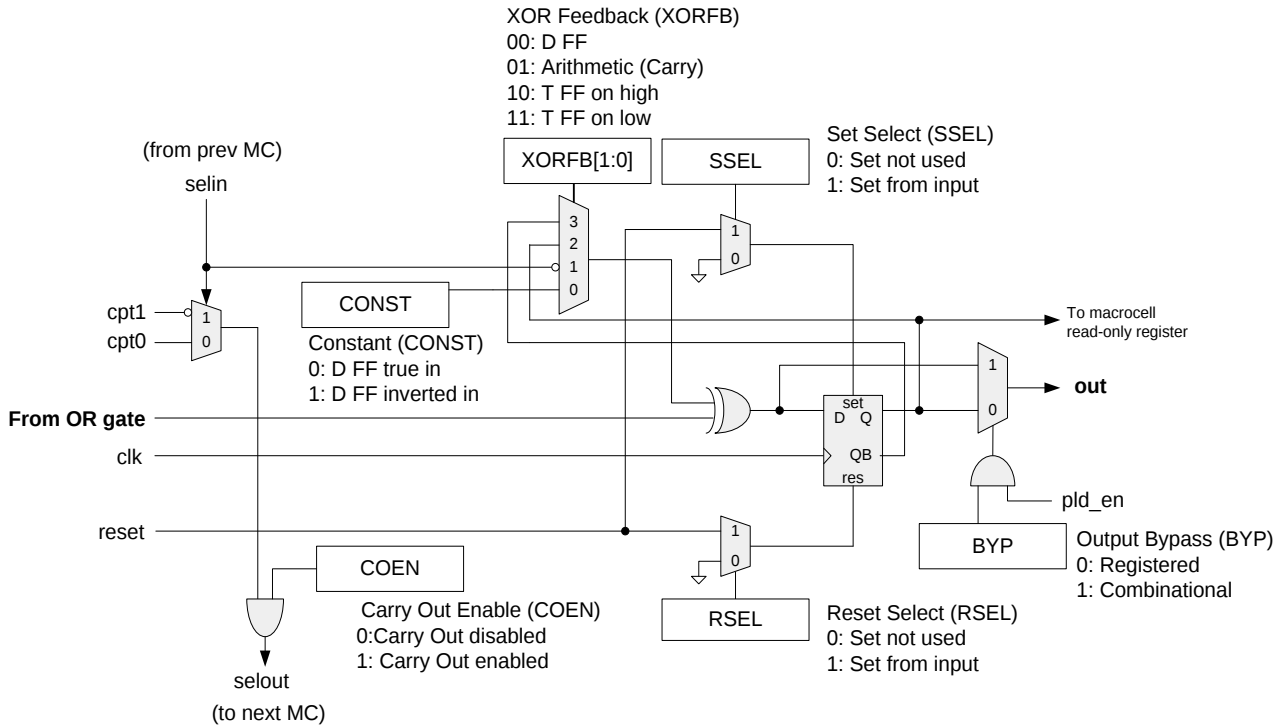
图 3 显示了逻辑方程映射到 PSoc PLD 的示例。

图 3. 映射到 PLD 的逻辑方程



在图 4 中展示了宏单元架构。可以寄存或组合宏单元输出。附录 B 通过两个示例说明了经过宏单元的数据流。更多有关您正在使用的设备的信息，请参见技术参考手册中 UDB 章节《宏单元》小节的内容。

图 4. PSoc PLD 宏单元架构



3. PSoC Creator

PSoC Creator 为硬件开发提供了一个基于原理图的环境。这样，您可以通过以下两种方法实现 UDB PLD 中的逻辑功能和状态机：

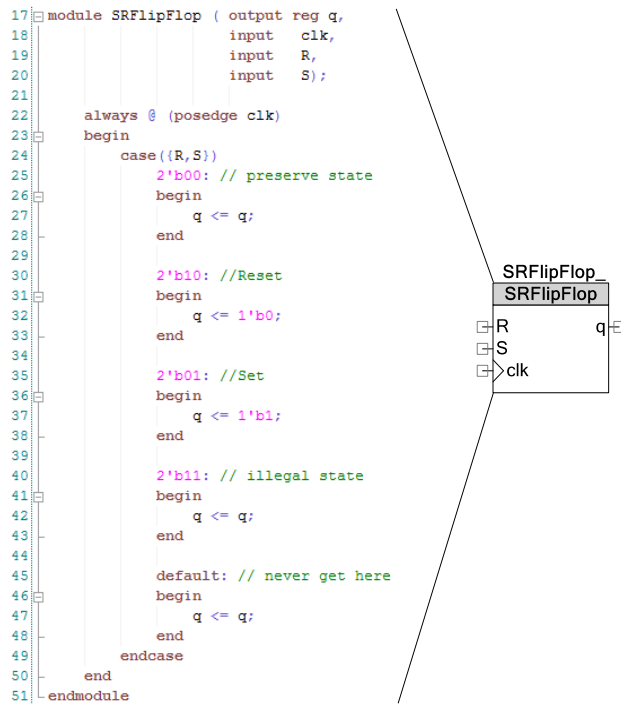
1. **Verilog**：PSoC Creator 支持硬件描述语言（HDL），既 Verilog。使用 Verilog 时，您可以实现随后映射到 PSoC UDB 的数字功能。该过程采用了 Warp™ 合成工具，即为 PSoC Creator 附带的 Verilog 编译器。

阅读本应用笔记后，您会了解如何创建基于 Verilog 的组件（请参考图 5）。

要想了解 Verilog，请参阅 [KBA86336 — PSoC 基本知识](#)。

注意： 有关 Warp 的详细信息，可以通过 PSoC Creator 中的 **Help > Documentation** 菜单查看 Warp Verilog 参考指南。

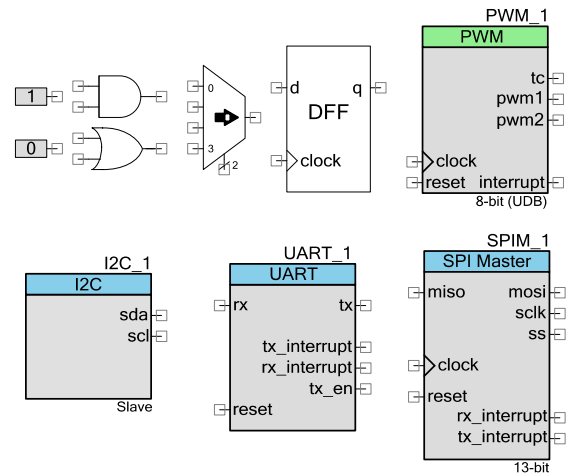
图 5. 基于 Verilog 的组件



2. **原理图**：此过程包括互连的单独门（AND、OR、XOR、NOT）、DFF 和其它数字逻辑模块，用于执行所需功能。PSoC Creator 为所有逻辑操作、复用器、查询表（LUT）和其它基于 PLD 的简单功能提供了各种门符号。

PSoC Creator 还提供了一个预建且测试的标准外设组件库。这些组件被映射到包含 PLD 和数据路径的 UDB 阵列中。图 6 中显示的是这些组件中的一部分。使用这些组件是使用 PSoC 的 PLD 功能（而不采用 Verilog）最快最简单的方法。

图 6. PSoC Creator 中的数字组件



4. 示例项目

了解 PSoc 的最好的的一种方法便是使用它。该示例项目详细介绍了创建基于 PLD 的简单 Verilog 组件的各步骤。

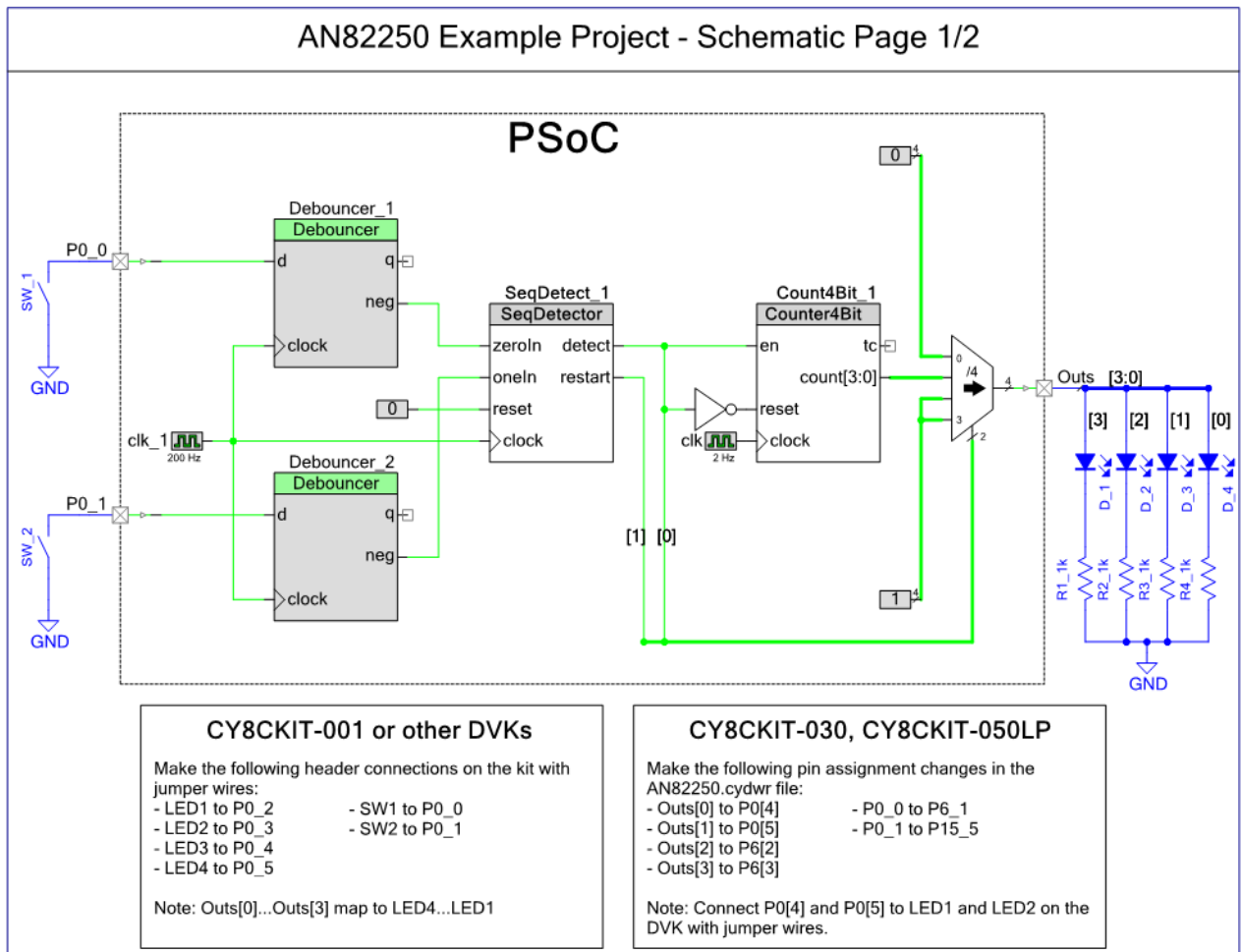
请先从应用笔记的[登陆页面](#)上下载 AN82250.zip 文件，然后阅读。要想查看该项目，请解压该文件夹，然后打开 PSoc Creator 中的 AN82250.cywrk 文件。按照原理图上的指导，可以设计该项目，使之能够在 CY8CKIT-001 PSoc 开发套件（DVK）的 PSoc 3 中运行。只要进行了轻微的修改，它即可在其他开发平台上运行。编译该项目，并将其编程到 PSoc DVK 中。

该示例项目在硬件中完全实现了 5 位序列检测器 — 无需在任何固件中进行。有关原理图的详细信息，请参考图 7。该项目的一个重要特性就是，除时钟和引脚外，原理图上显示的所有组件都是在 UDB PLD 中实现的。

该项目将二进制模型作为它的输入使用。该模型是由 PSoc DVK 上的两个按键开关组成的。按下开关 ‘SW_1’ 便表示逻辑 0，而按下开关 ‘SW_2’ 则表示逻辑 1。四个输出在 DVK 上驱动 LED，以指出检测器的状态。

重新设置 PSoc 时，LED 会闪烁发光，以指出 PSoc 已经准备好进行一次输出（即按下开关）。然后，PSoc 会遵循图 8 中所显示的状态图。如果您键入的是一个正确但不完整的序列，则 LED 会停止发光，以指出您已经进入部分序列。错误按下开关会打开四个 LED。如果您键入的 5 位序列全部正确，则 LED 会以二进制格式开始计数。

图 7. 示例项目的顶层设计原理图



信号流从输入到输出的情况如下所示：

- 使用去抖动器组件对两个按钮开关的输入进行去抖动和边沿检测。请将 PSoC Creator 更新为 [Component Pack 4](#) 或更高版本，以便访问该组件。
- 将输入引脚配置为电阻上拉。因此在按下开关时，按钮的输入会从高电平转为低电平。所以去抖动器的‘下降沿检测’输出用于指出按下开关有效。
- 然后，这些信号会被传送到 SeqDetector 组件，该组件被配置为检测 10110 序列。将 0（00000）到 31（11111）间的任意值输入到组件定制器中，即可更改此模型（图 9）。
- 如果正确输入了剩余的序列因子，可激活‘检测’输出。即使只有一个值输入存在错误，也会激活重启输出。
- 如果‘检测’信号被激活，则 4 位计数器开始计数，否则它会保持为复位状态。通过将 1 到 15 之间所需的 4 位周期值输入到组件定制器内，可以调整计数器的周期值（图 12）。
- ‘重启’和‘检测’信号会控制输出复用器，这样可以根据图 8 中的状态图驱动 4 个 LED。

图 8. 示例项目的状态图

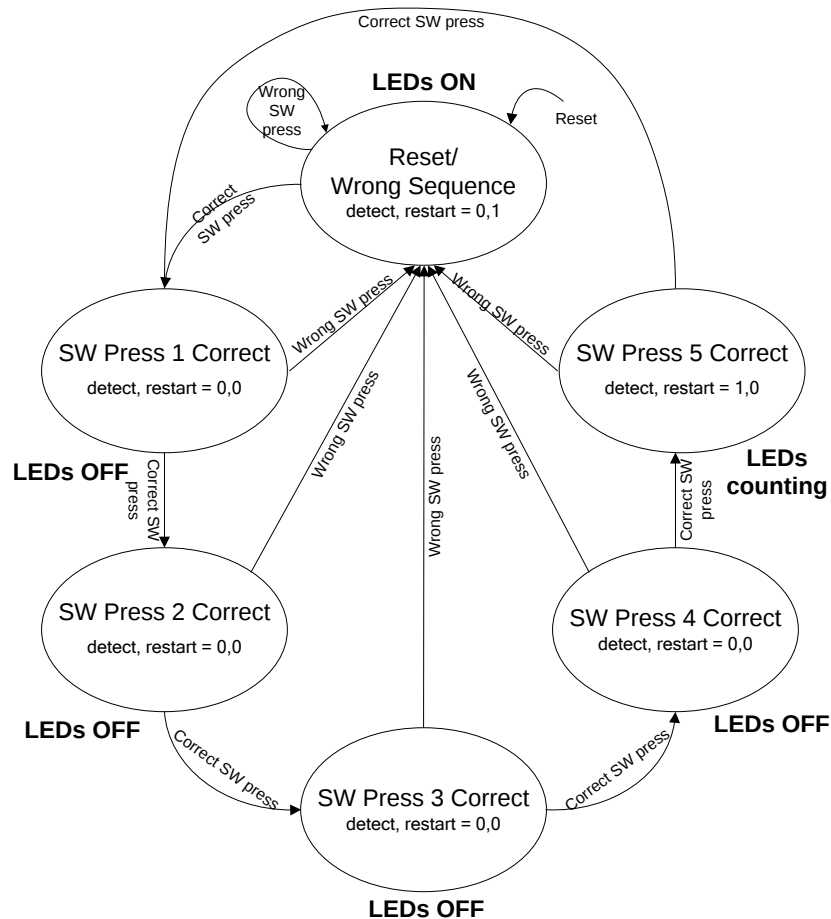


图 9. SeqDetector 组件定制器

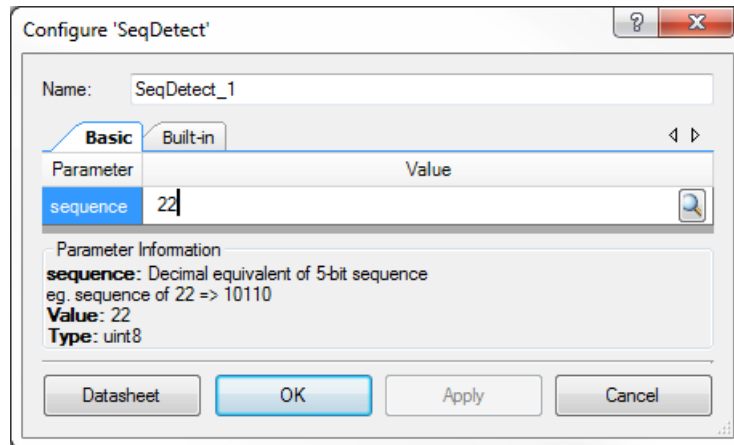
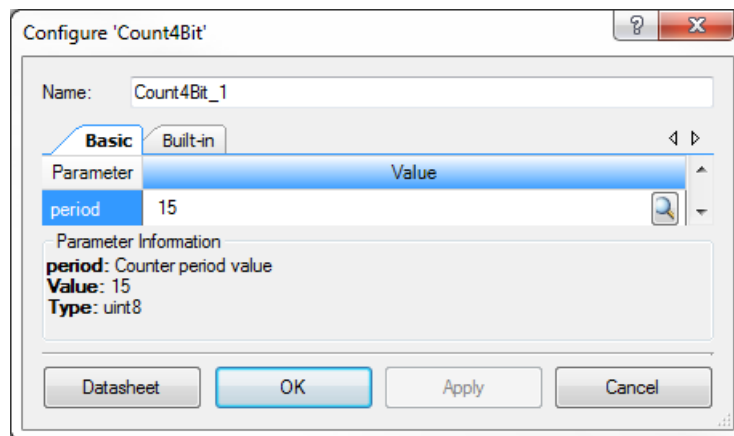


图 10. 计数器组件定制器



注意： 要查看 4 位计数器和序列检测器组件的 Verilog 文件，请导航到 **Workspace Explorer**（工作区浏览器）的 **Components**（组件）选项卡。

为有效地使用 PSoC PLD，需要在 PSoC Creator 中创建基于 Verilog 的组件。[KBA86338 — 创建基于 Verilog 的组件](#) 部分总结了基于 Verilog 的组件的创建过程。通过使用创建 SeqDetector 和 Counter4Bit 组件的示例，您可以熟悉该过程。

4.1 创建 Verilog 组件: Counter4Bit

带有同步复位和使能功能的 4 位递增计数器是一种基于 Verilog 的最简单自定义组件。

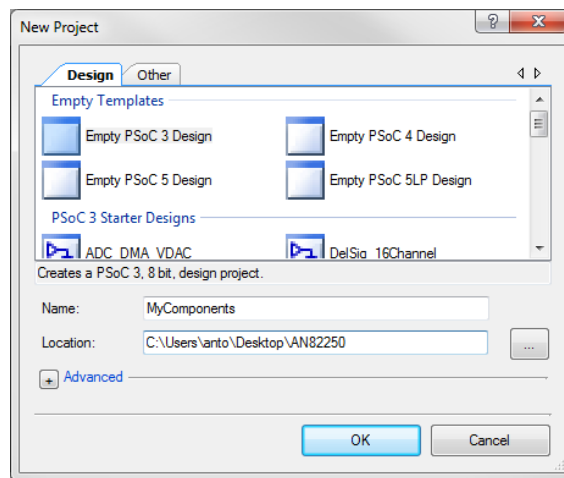
4.1.1 4 位计数器组件的创建步骤

您可以使用一个现有项目并将新组件添加到该项目内，但在该示例中，应从空白项目开始。

1. 启动 **PSoC Creator**，以开始创建一个新项目。在此示例中，将 ‘MyComponents’（我的组件）作为项目名称，如图 11 所示。

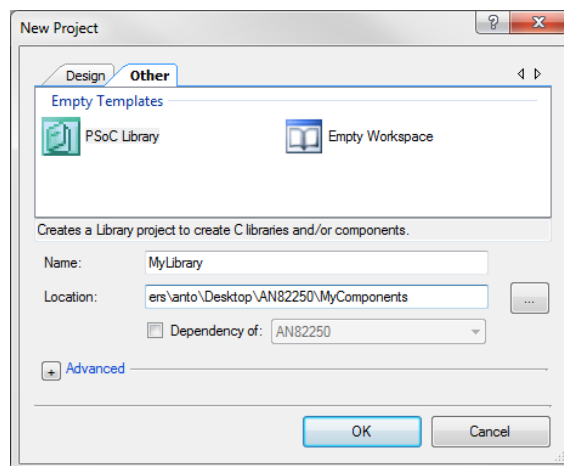
请注意，对于 Creator 3.3 或更高版本，新建项目的对话框会有所改变的。请在 “**Design Project**” 选项卡下面的对话框中选择 “**Target Device**” 并选择您正在使用的器件，然后点击 “**Next**”。在接下来的显示屏中选择 “**Empty Schematic**” 项并点击 “**Next**”，将会弹出最后一步的显示屏。请在工作区和项目下面的 “**Name**” 框中选择 “MyComponents” 并点击 “**Finish**”。

图 11. 新项目的对话框



2. 在 **Workspace Explorer** 的 **Source** 选项卡中，请右键点击 **MyComponents** 工作区，然后依次点击 **Add > New Project**。
3. 为将此新项目设置为组件库，请在 **New Project** 对话框中，点击 **Other**（其它）选项卡，并选择 **PSoC Library**（图 12）。将该示例命名为 ‘MyLibrary’，并保留位置上的默认值。这是在各个库项目中创建自定义组件的最好方法，这样可以简化组件的管理和重复使用。

图 12. 添加库项目



请注意，对于 Creator 3.3 和更高版本，新建项目的对话框会有所改变的。请在对话框中选择“**Library**”项目并点击“**Next**”。继续在“**Name**”框中选择“**MyLibrary**”，然后点击“**Finish**”。

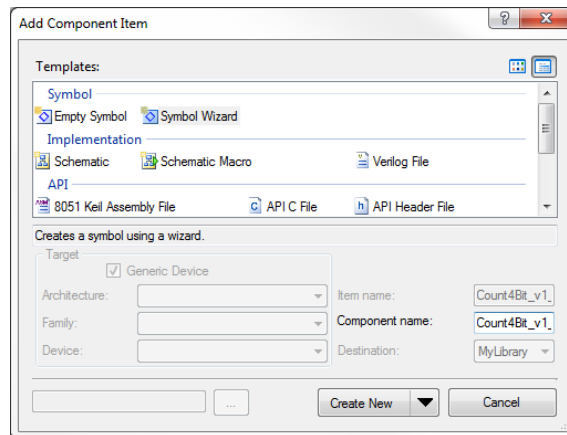
现在将一个新组件添加到刚刚创建的库中：

4. 在 **Components**（组件）选项卡上，右键点击‘**MyLibrary**’项目，然后点击上下文菜单中的 **Add Component Item**（添加组件项目）（图 13）项。

组件名称应该包含版本编号。将‘_vX_Y’标签附加到组件名称内，其中，‘X’表示主要版本、‘Y’指的是次要版本。PSoC Creator 具有版本控制功能，并有助于跟踪组件的多种版本。

5. 选择 **Symbol Wizard**（符号向导）组件模板，并将该组件命名为‘**Count4Bit_v1_00**’。

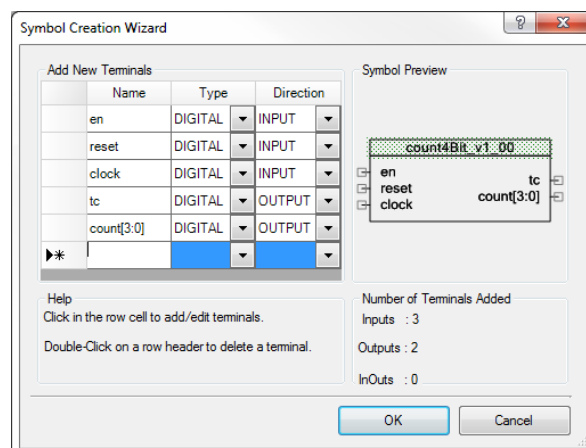
图 13. 创建自定义组件



您可以选择空白符号，但在该示例中使用向导是为了节省时间。更多有关信息，请点击 **Help > Documentation** 查看组件作者指南。

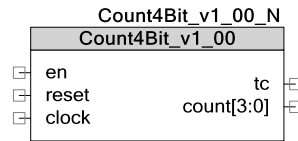
6. 要启动组件符号向导，请点击 **Create New** 按钮。
此向导要求定义输入和输出，然后使用这些信息进行创建组件符号。
7. 为原理图符号定义三个输入端和两个输出端，如图 14 所示。

图 14. Count4Bit 的符号创建向导



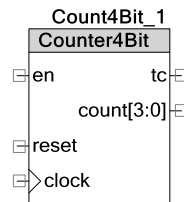
点击 **OK** 以生成符号原理图中的各个符号，如图 15 所示。

图 15. 4 位计数器的初始符号



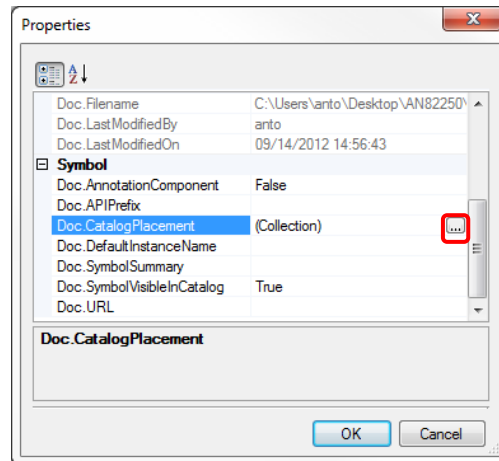
您可以调整组件大小，并修改组件的外观，如图 16 中所示。

图 16. 四位计数器的最终符号



8. 右键点击符号原理图中的空白位置，然后点击 **Properties**（属性）。在属性字段的 **Symbol**（符号）部分，请点击 **Doc. CatalogPlacement** 上的省略符号（...），如图 17 中所示。

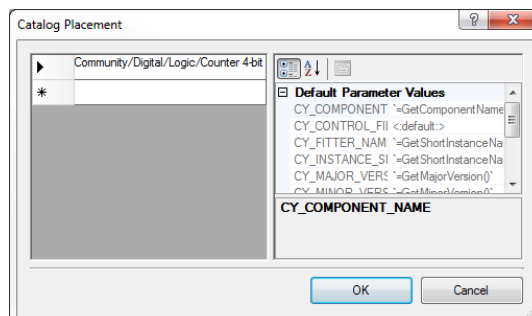
图 17. 符号 Properties 对话框



9. 在 **Catalog Placement**（目录放置）对话框中，请输入 **Community/Digital/Logic/Counter 4-bit**，如图 18 中所示。

该操作会将计数器放置在 **Component Catalog**（组件目录）窗口的 **Community**（社区）选项卡下 ‘Digital’（数字）文件夹的 ‘Logic’（逻辑）子文件夹内，目录名称为 ‘Counter 4-bit’（4 位计数器）。

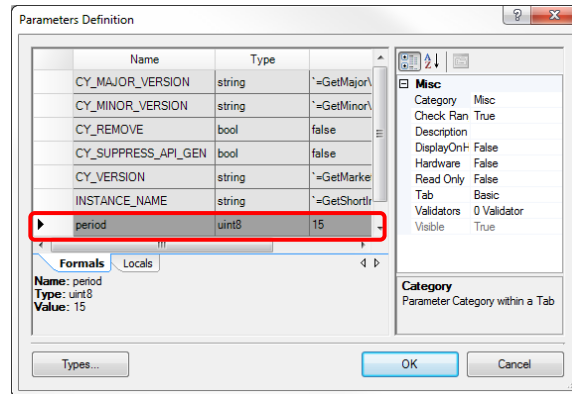
图 18. 设置目录放置



要想获得计数器的可配置周期值，则必须添加组件参数。

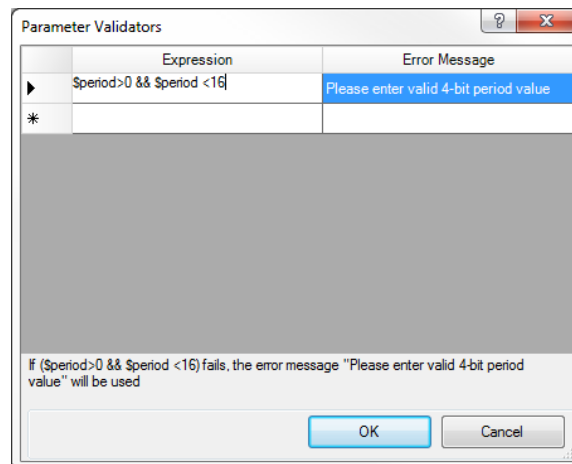
10. 右键点击符号原理图中的空白位置，然后点击 **Symbol Parameters**（符号参数）（图 19）。将参数的 **name**（名称）、**type**（类型）和 **default value**（默认值）分别指定为 ‘period’（周期）、‘uint8’ 和 ‘15’。此参数允许用户在计数器的定制器中指定其周期值（图 10）。

图 19. Count4Bit 的符号参数



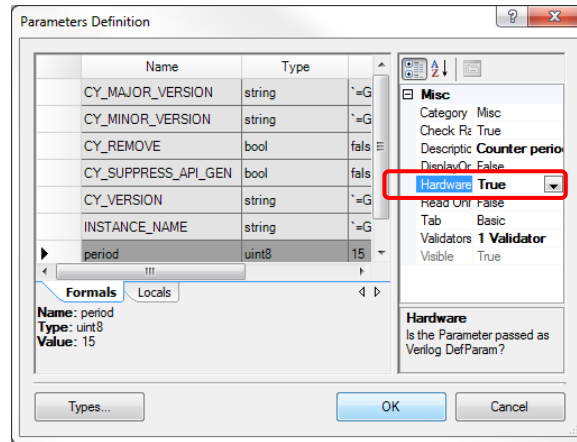
11. 通过点击 **Parameter Definition** 对话框的 **Misc** 部分中的 **Description** 字段，输入此参数的说明内容。点击 **Validators** 字段进行设置参数的验证器。该验证器会检查参数是否位于可接受的输入范围内。设置验证器以确保周期值属于 1 到 15 的范围内，如图 20 中所示。点击 **OK** 以进行更改。

图 20. 添加 Count4Bit 的验证器



12. 在 **Parameter Definition** 对话框中将 **Hardware** 字段设置为 **True**，如图 21 所示。必须将参数传送到 Verilog 文件中。

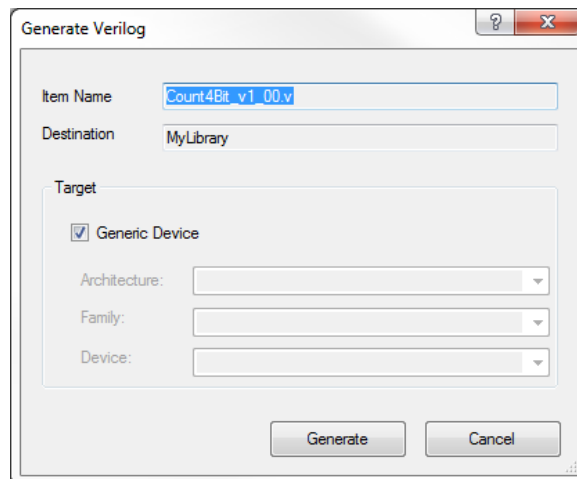
图 21. 将参数传送给硬件



下一步是将原理图符号连接到 Verilog 文件。PSoc Creator 根据组件符号生成 Verilog 外壳。

13. 若要执行该操作，请右键点击符号原理图中空白的位置，然后点击 **Generate Verilog**（生成 Verilog）。保持 **Generate Verilog** 对话框中的所有默认设置，然后点击 **Generate**（生成），如图 22 中所示。

图 22. 生成带符号的 Verilog 文件



Target（目标）值可用于限制指定器件的配置，但在该示例中，使用的是指定器件的默认设置。

刚刚创建的符号 Verilog 文件会出现。

注意：在 Verilog 文件中一共有三对 `#start header - #end`。当编辑该文件时，请将所有代码放置在这些章节内。如果您重新生成了 Verilog 文件，那么对 Verilog 文件中这三个部分外的内容进行的更改将被丢失。

现在，您可以描述 Verilog 中的计数器。若要参考，请查看代码 1 中所示的完整代码。

4.2.1 Verilog 设计：4 位计数器

首先，要寄存各个输出。将 Count4Bit_v1_00 模块的输入/输出修改为：

```
output reg [3:0] count,  
output reg tc,
```

注意：如果重新生成 Verilog 文件，需要再次进行上述修改。此外，不能在该文件中的任何其他位置进行这些定义。然后，由于这是一个同步设计，因此将（带有时钟沿的）*always* 模块添加到 Verilog 中的 #start body 和 #end 注释内：

```
always @ (posedge clock)  
begin  
.  
.  
.  
end
```

注意：为了降低时序和同步失败的可能性，在 PSoC 设计中优先采用上升沿时钟。计数器具有同步复位功能，当它被激活时，会清除 ‘tc’ 和 ‘count’。

```
if(reset)  
begin  
count <= 4'b0000;  
tc <= 1'b0;  
end
```

注意：它也支持异步复位/预设信号。请参考 *Warp Verilog 参考指南* 中第 3.3.2 节的内容，了解更多有关合成异步触发器的信息。

en 输入信号是一个使能硬件信号。如果该输入处于低电平状态，则输出仍会处于活动状态，但该组件不会改变其状态。

```
if(en)          /* start counting */  
begin  
.  
.  
.  
end  
else           /* preserve state */  
begin  
count <= count;  
tc <= tc;  
end
```

当 count 值达到周期值时，只要 count 等于 period，终端计数输出 tc 便为逻辑 1。

```
if(count == period)  
begin  
tc <= 1'b1;  
count <= 4'b0000;  
end
```

否则，4 位计数器必须从 0 开始计数到 period，并在每个正向时钟沿到来时递增 count 输出。

```
else
begin
    count <= count + 1;
    tc <= 1'b0;
end
```

结束对 Verilog 文件的更改操作后，请保存更改的内容。现在，您已经完成了对 4 位递增计数器的 Verilog 描述过程。完整代码显示在代码 1 中。

代码 1. 完成 4 位计数器的 Verilog 设计

```
module Count4Bit_v1_00 (
    output reg [3:0] count,
    output reg tc,
    input    clock,
    input    en,
    input    reset
);

    parameter period = 0;

    //`#start body` -- edit after this line, do not edit this line

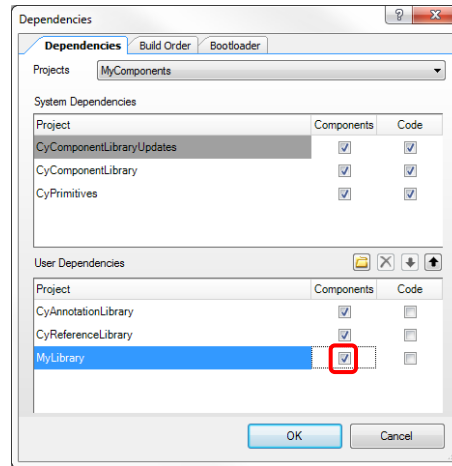
    always @ (posedge clock)
    begin
        if(reset)
            begin
                count <= 4'b0000;
                tc <= 1'b0;
            end
        else
            begin
                if(en)
                    begin
                        if(count == period)
                            begin
                                tc <= 1'b1;
                                count <= 4'b0000;
                            end
                        else
                            begin
                                count <= count + 1;
                                tc <= 1'b0;
                            end
                        end
                    end
                else
                    begin
                        count <= count;
                        tc <= tc;
                    end
                end
            end
        end

    //`#end` -- edit above this line, do not edit this line
endmodule
```

要在 ‘MyComponents’ 项目中使用此组件，必须将 ‘MyLibrary’ 设置为相关库。

为了执行此操作，请右键点击 **Source** 选项卡中的 **MyComponents**，然后选择 **Dependencies**（相关库）。请确保已经勾选了 **User Dependencies**（用户依赖库）下面 ‘MyLibrary’ 的复选框，如图 23 显示。

图 23. 添加组件的依赖库



要在设计中使用该库，请转到 MyComponents 项目中的 TopDesign.cysch 文件，然后导航到 **Component Catalog** 中。**4** 位计数器位于 **Community** 选项卡中。将其拖放到原理图上，然后进行所需连接。

注意：请参考 PSoC Creator 帮助文章 ‘组件库项目’ 和 ‘层级的基本设计教程’，以加深对创建和使用库项目内容的了解。

接下来，请在 Verilog 中创建序列检测器组件。

4.2 创建 Verilog 组件：SeqDetector

SeqDetector 组件是此示例项目的关键部分。它是一个可配置的 5 位二进制序列检测器，并且是在 PSoC PLD 中实现的。

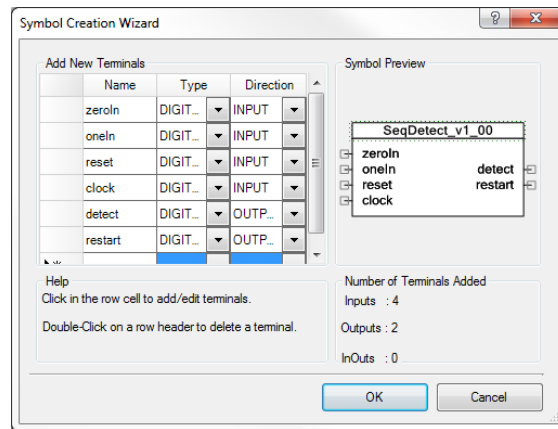
4.2.1 SeqDetector 组件的创建步骤

SeqDetector 的创建流程同计数器的很相似。

1. 选择 **Workplace Explorer** 中的 **Components** 选项卡。右键点击 **MyLibrary** 项目，然后点击 **Add Component Item**。
2. 选择 **Symbol Wizard** 组件模板，并将该组件命名为 **SeqDetect_v1_00**。
3. 单击 **Create New** 按钮，以启动组件符号向导。

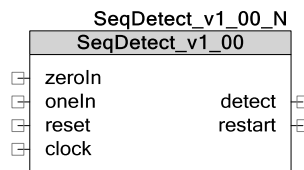
- 为原理图符号定义四个输入端和两个输出端，如图 24 中所示。

图 24. SeqDetector 的符号创建向导



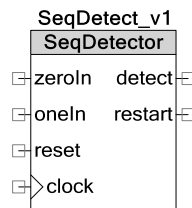
- 点击 **OK** 以生成符号原理图中的符号，如图 25 中所示。

图 25. 序列检测器的初始符号



您可以重新调整该组件的大小，如图 26 所示：

图 26. 序列检测器的最终符号



- 右键点击符号原理图中空白的位置，然后点击 **Properties**。

在属性字段中的 **Symbol** 部分，请点击 **Catalog**

Placement 上的省略符号 (...)。将 **Community/Digital/Logic/Sequence Detector 5-bit** 输入到 **CatalogPlacement** 内。

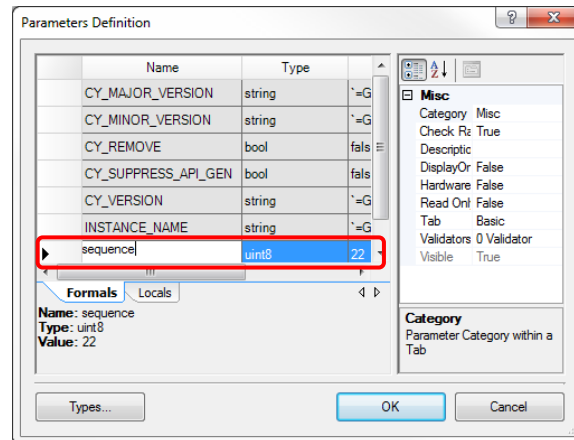
该操作会将 SeqDetector 放置在 Component Catalog 的 **Community** 选项卡下 ‘Digital’ 文件夹的 ‘Logic’ 子文件夹中，其目录名称为 ‘Sequence Detector 5-bit’。

- 要想获得 SeqDetector 的可配置序列值，必须添加组件参数。

右键点击符号原理图中空白的位置，然后点击 **Symbol Parameters**。

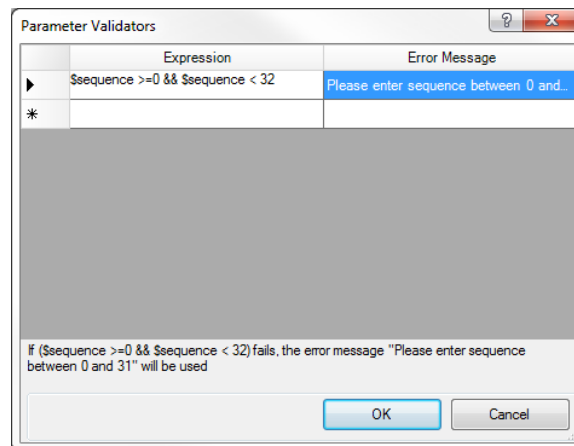
将参数的 **name**（名称）、**type**（类型）和 **default value**（默认值）分别指定为 ‘sequence’（序列）、‘uint8’ 和 ‘22’，如图 27 所示。

图 27. 为 SeqDetect 定义一个参数



- 通过点击 **Parameter Definition** 对话框中 **Misc** 部分的 **Description** 字段，输入此参数的说明内容。
- 点击 **Validators** 字段以设置序列的验证器。设置验证器确保序列值的范围为 0 到 31，如图 28 中所示。点击 **OK** 以进行更改。

图 28. SeqDetect 验证器



- 一旦返回到 **Parameter Definition** 对话框，请将 **Hardware** 字段设置为 **True**。
- 若要执行该操作，请右键点击符号原理图中空白的位置，然后点击 **Generate Verilog**（生成 Verilog）。
下一步是将原理图符号连接到 Verilog 文件。
- 保持 **Generate Verilog** 对话框中的所有默认设置，然后点击 **Generate**。

序列检测器模块的完整 Verilog 代码显示在附录 C 中。下一章将介绍代码的主要组成部分。

4.2.2 Verilog 设计: SeqDetector

同计数器一样，第一步需要寄存 SeqDetect 模块的终端列表的输出：

```
output reg detect,  
output reg restart,
```

序列检测器的主干是一个拥有六种状态的状态机（图8）。将下面的代码添加到#start body注释后面，这样可以给每一种状态创建本地参数：

```
localparam START    = 3'd0;  
  
localparam STATE_1  = 3'd1;  
  
localparam STATE_2  = 3'd2;  
  
localparam STATE_3  = 3'd3;  
  
localparam STATE_4  = 3'd4;  
  
localparam DETECT   = 3'd5;
```

请注意，通过使用关键词localparam，可以声明状态定义常量。这样可以防止这些常量与其它模块中相同名称的常量发生冲突。

将状态变量和模型变量分别声明为‘寄存器’类型和‘线路’类型。

```
reg [2:0] state_curr, state_next;  
wire [4:0] pattern = sequence;
```

序列检测器模块具有两个 always 子模块，即一个连续子模块和一个组合子模块。

always 连续子模块模仿了上升沿所触发的触发器。

always 组合子模块需要检测的信号列表定义为：

```
always @ (oneIn or zeroIn or state_curr or pattern)
```

注意：当为 PSoC Creator 编写 Verilog 时，always 语句必须包含需要检测的信号列表。

always 组合子模块对输入进行解码，并且它会根据这些输入和其当前的状态分配下一个状态。如果检测到某个输入 1 或输入 0，则会检查该输入，否则保持当前的状态。

例如，启动状态如下所示：

```
if((oneIn & pattern[4]) ||  
(zeroIn & !pattern[4]))  
begin  
    state_next <= STATE_1;  
end  
else  
begin  
    state_next <= START;  
end
```

注意：pattern[4] 保留了序列中的第一个正确值。

其他状态也同启动状态相类似 — 将组件输入与 pattern[3]、...、pattern[0] 进行比较，以便对下一个状态进行解码。

完成 Verilog 文件的更改操作后，请保存更改的内容。现在已经完成了序列检测器的设计。要想在设计中使用它，请按照计数器部分描述的步骤进行操作。

5. 数据路径与基于 PLD 的设计

通信、时序和控制应用对基础功能的逻辑结构具有不同的要求。

根据经验，使用 UDB 资源的最好方法如下：

- PLD（随机逻辑）：具有控制函数、CPLD 集成和胶合逻辑等功能。
- 数据路径（结构逻辑）：具有通信、定时和计算等功能。

例如，请考虑分别在 PLD 和数据路径中实现下面 8 位算术和逻辑操作。

功能	PLD 中的资源消耗		数据路径中的资源消耗	
	PLD	使用的比例 (%)	数据路径	使用的百分比 (%)
ADD8	5	10.4%	1	4.2%
SUB8	5	10.4%	1	4.2%
CMP8	3	6.3%	1	4.2%
SHIFT8	3	6.3%	1	4.2%

注意：表中的比例和百分比是根据包含 24 个 UDB 的器件计算的。

可以在 PSoC PLD 中实现多种功能，但如果没有很好地利用数据路径模块的优势，则很容易耗尽资源。

6. 总结

本应用笔记介绍了 UDB PLD，并说明了在 PSoC Creator 中创建基于 Verilog 的组件的设计过程。阅读本应用笔记后，您应该基本掌握了 PLD 架构，并且可以创建基于 Verilog 的自定义组件。

PSoC UDB 给数字设计提供了灵活有效的架构。您可以将从简单到中等复杂程度的广大逻辑设计范围传输到 PSoC PLD。同时使用 PLD 和数据路径是实现高度复杂设计的最佳方法。有关 UDB 数据路径的更多信息，请参阅 [AN82156](#)。

6.1 其他信息

[附录 A](#) 已经对 PSoC PLD 和竞争对手的 CPLD 之间的相应资源数量进行了比较。

[附录 B](#) 通过两个示例说明了通过宏单元的数据流。

[附录 C](#) 包含序列检测器模块的完整 Verilog 代码。

[附录 D](#) 分别对项目报告文件和静态时序分析进行了简单介绍。

7. 参考文档

应用笔记

[AN82156 — PSoC 3、PSoC 4 和 PSoC 5LP 使用 UDB 数据路径进行设计 PSoC Creator 组件](#)

[AN81623 — PSoC 3 和 PSoC 5LP 数字设计的最佳实践](#)

[AN62510 — 使用 PSoC 3 和 PSoC 5LP 实现状态机功能](#)

[AN61290 — PSoC 3 和 PSoC 5LP 硬件设计中的注意事项](#)

[AN72382 — 使用 PSoC 3 和 PSoC 5LP GPIO 引脚](#)

[AN60580 — PSoC 3 和 PSoC 5LP 中的 SIO 提示和技巧](#)

[AN54181 — PSoC 3 入门](#)

[AN79953 — PSoC 4 入门](#)

[AN77759 — PSoC 5LP 入门](#)

知识库文章

[KBA86336 — PSoC 的 Verilog 基本知识](#)

[KBA86338 — 创建基于 Verilog 的组件](#)

[KBA81772 — 向项目中添加组件要素/Verilog 组件](#)

[用于创建组件的 Verilog 和数据路径配置工具的入门信息](#)

视频

以下视频介绍了 PSoC Creator 和 Verilog 组件的创建过程：

基本步骤

[创建新项目](#)

[使用起始页](#)

组件创建

[PSoC Creator 113: 基于 Verilog 的 PLD 组件](#)

[创建新组件符号](#)

[创建 Verilog 实现](#)

[创建原理图实现](#)

关于作者

姓名:	Vijay Kumar Marrivagu
职务:	系统工程师负责人
背景信息:	具有数年的数字设计和验证方面经验。
姓名:	Antonio Rohit De Lima Fernandes
职务:	应用工程师
背景信息:	获得印度拉贾斯坦邦博拉理 BITS 学院电气工程技术学士学位。

A. 附录 A: PSoC PLD 资源与竞争对手 CPLD 的比较

表 1 将 PSoC PLD 资源和大小相似的 CPLD 进行比较。请记住，此表没有考虑 UDB 数据路径中的可编程逻辑。同时使用 PSoC PLD 和数据路径时，PSoC 比 CPLD 更有竞争力。

表 1. PSoC PLD 宏单元与竞争对手的 PLD 的比较

器件	宏单元 (MC) 数量	模块数量	每个模块的 MC 数量	每个模块的 输入数量	乘积项 (PT) 数量	每个模块的 乘积项数量
赛普拉斯 PSoC						
超集 PSoC 3、 PSoC 5LP	192	48	4	12	384	8
CY8C42	32	8	4	12	64	8
Altera MAX-II						
EPM240	128 到 240*	24	10	36	*	*
Lattice ispMACH						
4032ZE	32	2	16	36	160	80
4064ZE	64	4	16	36	320	80
40128ZE	128	8	16	36	640	80
Xilinx Coolrunner-II						
XC2C32A	32	2	16	56	112	56
XC2C64A	64	4	16	56	224	56
XC2C128	128	8	16	56	448	56

* Altera MAX-II 架构不是一个传统的乘积项架构。

图 29 和图 30 显示的是分别针对 D 型触发器 (D-FF) 和 T 型触发器 (T-FF) 功能经过宏单元的数据流。

C. 附录 C: 序列检测器的 Verilog 代码

```

module SeqDetect_v1_20 (
    output reg detect,
    output reg restart,
    input  clock,
    input  oneIn,
    input  reset,
    input  zeroIn
);

    /* Note that the value assigned to the parameter in this line
     * has no effect. The actual parameter value is taken from
     * the component customizer.
     */
    parameter sequence = 0;

    /*`#start body` -- edit after this line, do not edit this line
     * Six states are required.
     * The states follow START -> STATE_1 -> ... -> DETECT if the
     * correct inputs are entered. As soon as a wrong input is entered
     * the design jumps to the START state. The states are defined as
     * localparams (instead of `defines) to limit their scope to this
     * module only.
     */
    localparam START    = 3'd0;          /* detect, restart = 0, 1 */
    localparam STATE_1  = 3'd1;          /* detect, restart = 0, 0 */
    localparam STATE_2  = 3'd2;          /* detect, restart = 0, 0 */
    localparam STATE_3  = 3'd3;          /* detect, restart = 0, 0 */
    localparam STATE_4  = 3'd4;          /* detect, restart = 0, 0 */
    localparam DETECT   = 3'd5;          /* detect, restart = 1, 0 */

    /* registered value to hold 3-bit state */
    reg [2:0] state_curr, state_next;

    /* pattern[4:0] holds the user-supplied sequence value
     * suppose sequence = 22 then pattern[4:0] = 5'b10110
     * Note that pattern[4] is the first-entered user input
     */
    wire [4:0] pattern = sequence;

    /* Sequential block of the state machine - outputs are assigned here */
    always @ (posedge clock)
    begin
        /* reset causes the component to enter the START state */
        if(reset)
        begin
            state_curr <= START;
            /* Immediately assign detect and restart values */
            detect <= 1'b0;
            restart <= 1'b1;
        end
        else /* reset is not asserted - go through states */
        begin
            state_curr <= state_next;

            /* Assign 'detect' value - 1 only in DETECT state, 0 otherwise */
            if (state_next == DETECT)
            begin
                detect <= 1'b1;
            end
            else
        end
    end

```

```
begin
    detect <= 1'b0;
end

/* Assign 'restart' value - 1 only in RESTART state, 0 otherwise */
if (state_next == START)
begin
    restart <= 1'b1;
end
else
begin
    restart <= 1'b0;
end
end

end

/* Finite State Machine combinatorial block - contains most of the
 * combinatorial logic.
 */
always @ (oneIn or zeroIn or state_curr or pattern)
begin
    /* If either a one or zero has been entered, take action */
    if(oneIn | zeroIn)
    begin
        case(state_curr)
            START:      /* Initial state */
            begin
                /* check whether the first bit entered is correct */
                if((oneIn & pattern[4]) || (zeroIn & !pattern[4]))
                begin
                    state_next <= STATE_1; /* advance to the next state */
                end
                else
                begin
                    /* revert to the initial state */
                    state_next <= START;
                end
            end

            STATE_1:    /* First input is correct */
            begin
                if((oneIn & pattern[3]) || (zeroIn & !pattern[3]))
                begin
                    state_next <= STATE_2;
                end
                else
                begin
                    state_next <= START;
                end
            end

            STATE_2:    /* Two inputs are correct */
            begin
                if((oneIn & pattern[2]) || (zeroIn & !pattern[2]))
                begin
                    state_next <= STATE_3;
                end
                else
                begin
                    state_next <= START;
                end
            end
        end
    end
end
```

```
STATE_3:      /* Three inputs are correct */
begin
    if((oneIn & pattern[1]) || (zeroIn & !pattern[1]))
    begin
        state_next <= STATE_4;
    end
    else
    begin
        state_next <= START;
    end
end

STATE_4:      /* Four inputs are correct */
begin
    if((oneIn & pattern[0]) || (zeroIn & !pattern[0]))
    begin
        state_next <= DETECT;
    end
    else
    begin
        state_next <= START;
    end
end

DETECT:      /* All five inputs are correct! */
begin
    /* When in the detect state, if an input is given, show same behavior as START */
    /* check whether the bit entered is the correct beginning to a new sequence*/
    if((oneIn & pattern[4]) || (zeroIn & !pattern[4]))
    begin
        state_next <= STATE_1;
    end
    else /* revert to the initial state */
    begin
        state_next <= START;
    end
end

default:/* we should never get here - reset the component */
begin
    state_next <= START;
end

endcase

end
else /* if neither 1 or 0 have been entered, stay in same state */
begin
    state_next <= state_curr;
end
end

end

//`#end` -- edit above this line, do not edit this line
endmodule
```

D. 附录 D：设计构建完成后的注意事项

D.1 项目报告文件

从 **Workspace Explorer** 窗口的 **Results** 选项卡访问项目编译报告文件（<project_name>.rpt）。成功编译后，会创建该报告文件。下面是报告文件的主要章节。

- 技术映射总结部分（包括宏单元、乘积项、数据路径、引脚、时钟分频器等各项资源）的使用情况如图 31 中介绍的内容。

图 31. PSoC Creator 项目编译的报告文件

TopDesign.cysch	Design09.cydwr	Mycounter.v	Design09.rpt		
660	-----				
661	Technology mapping summary				
662	-----				
663					
664	Resource Type	: Used : Free : Max : % Used			
665	=====				
666	Digital domain clock dividers	: 1 : 7 : 8 : 12.50%			
667	Analog domain clock dividers	: 0 : 4 : 4 : 0.00%			
668	Pins	: 9 : 63 : 72 : 12.50%			
669	Macrocells	: 5 : 187 : 192 : 2.60%			
670	Unique Pterms	: 5 : 379 : 384 : 1.30%			
671	Total Pterms	: 6 : : : :			
672	Datapath Cells	: 0 : 24 : 24 : 0.00%			
673	Status Cells	: 0 : 24 : 24 : 0.00%			
674	Control Cells	: 0 : 24 : 24 : 0.00%			

- 合成结果 — 列出了在每个合成阶段所生成的错误和警告：Verilog 的编译、解析、高级合成、优化等信息。本部分包含了使用合成器进行优化的逻辑的详细信息，并且这些内容有助于进行调试和故障排除。
- 数字放置：PLD 包装总结。图 32 展示了 PLD 使用的示例。

图 32. PLD 包装总结报告示例

PLD Packing Summary

Resource Type : Used : Free : Max : % Used

PLDs : 2 : 46 : 48 : 4.17%

- 数字放置：PLD 包装总结：PLD 统计 — 图 33 展示了每个逻辑阵列模块（LAB）中 PLD PT 和宏单元平均使用情况的示例。

图 33. PLD 使用情况报告示例

PLD Resource Type :	Average/LAB
-----	-----
Inputs :	2.00
Pterms :	2.50
Macrocells :	2.50

- 最终放置总结 — 提供了有关组件的详细信息。报告中该部分显示的是 UDB 使用、占用、统计和（坐标）放置的详细信息。

D.2 静态时序分析

调试数字设计的一个重要部分是静态时序分析（STA）。STA 用于评估数字设计，并计算信号输出和输入之间的延迟。根据这些延迟，它能够计算在设计中所使用的每个时钟的最大有效频率。

当您编译某个项目时，PSoC Creator 会自动创建静态时序分析报告。该报告介绍了设计中限制每个时钟频率的重要路径。如果计算得到的最大时钟频率低于所需的时钟频率，则会发出一个警告，指出设计中存在时序冲突。

更多有关如何避免时序冲突和处理 PSoC Creator STA 警告的信息，请查看 [AN81623 — PSoC 3 和 PSoC 5LP 数字设计的最佳实践](#)。

文档修订记录

文档标题: AN82250 — 使用 Verilog 实现 PSoC® 3、PSoC 4 和 PSoC 5LP 的可编程逻辑设计

文档编号: 001-93058

版本	ECN	变更者	提交日期	变更说明
**	4521565	MSON	10/14/2014	本文档版本号为 Rev**, 译自英文版 001-82250 Rev*E。
*A	4718362	HENG	09/04/2015	本文档版本号为 Rev*A, 译自英文版 001-82250 Rev*F。
*B	5184195	HENG	03/25/2016	本文档版本号为 Rev*B, 译自英文版 001-82250 Rev*H。
*C	5824344	AESATMP9	07/19/2017	更新徽标和版权。
*D	5956638	TDU	09/27/2017	无变更。

全球销售和 design 支持

赛普拉斯公司拥有一个由办事处、解决方案中心、工厂代表和经销商组成的全球性网络。要找到离您最近的办事处，请访问[赛普拉斯所在地](#)。

产品

ARM® Cortex® 微控制器	cypress.com/arm
汽车级产品	cypress.com/automotive
时钟与缓冲器	cypress.com/clocks
接口	cypress.com/interface
物联网	cypress.com/iot
存储器	cypress.com/memory
微控制器	cypress.com/mcu
PSoc	cypress.com/psoc
电源管理 IC	cypress.com/pmic
触摸感应	cypress.com/touch
USB 控制器	cypress.com/usb
无线连接	cypress.com/wireless

PSoc® 解决方案

[PSoc 1](#) | [PSoc 3](#) | [PSoc 4](#) | [PSoc 5LP](#) | [PSoc 6](#)

赛普拉斯开发者社区

[论坛](#) | [WICED IoT 论坛](#) | [项目](#) | [视频](#) | [博客](#) | [培训](#) | [组件](#)

技术支持

cypress.com/support

此处引用的所有其他商标或注册商标归其各自所有者所有。

 **CYPRESS**
EMBEDDED IN TOMORROW™

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

赛普拉斯半导体公司，2012-2017 年。本文件是赛普拉斯半导体公司及其子公司，包括 Spansion LLC（“赛普拉斯”）的财产。本文件，包括其包含或引用的任何软件或固件（“软件”），根据全球范围内的知识产权法律以及美国与其他国家签署条约由赛普拉斯所有。除非在本款中另有明确规定，赛普拉斯保留在该等法律和条约下的所有权利，且未就其专利、版权、商标或其他知识产权授予任何许可。如果软件并不附随有一份许可协议且贵方未以其他方式与赛普拉斯签署关于使用软件的书面协议，赛普拉斯特此授予贵方属人性质的、非独家且不可转让的如下许可（无再许可权）（1）在赛普拉斯特软件著作权项下的下列许可权（一）对以源代码形式提供的软件，仅出于在赛普拉斯硬件产品上使用之目的且仅在贵方集团内部修改和复制软件，和（二）仅限于在有关赛普拉斯硬件产品上使用之目的将软件以二进制代码形式的向外部最终用户提供（无论直接提供或通过经销商和分销商间接提供），和（2）在被软件（由赛普拉斯公司提供，且未经修改）侵犯的赛普拉斯专利的权利主张项下，仅出于在赛普拉斯硬件产品上使用之目的制造、使用、提供和进口软件的许可。禁止对软件的任何其他使用、复制、修改、翻译或汇编。

在适用法律允许的限度内，赛普拉斯未对本文件或任何软件作出任何明示或暗示的担保，包括但不限于关于适销性和特定用途的默示保证。赛普拉斯保留更改本文件的权利，届时将不另行通知。在适用法律允许的限度内，赛普拉斯不对因应用或使用本文件所述任何产品或电路引起的任何后果负责。本文件，包括任何样式设计信息或程序代码信息，仅为供参考之目的提供。文件使用人应负责正确设计、计划和测试信息应用和由此生产的任何产品的功能和安全性。赛普拉斯产品不应被设计为、设定为或授权用作武器操作、武器系统、核设施、生命支持设备或系统、其他医疗设备或系统（包括急救设备和手术植入物）、污染控制或有害物质管理系统中的关键部件，或产品植入之设备或系统故障可能导致人身伤害、死亡或财产损失其他用途（“非预期用途”）。关键部件指，若该部件发生故障，经合理预期会导致设备或系统故障或会影响设备或系统安全性和有效性的部件。针对由赛普拉斯产品非预期用途产生或相关的任何主张、费用、损失和其他责任，赛普拉斯不承担全部或部分责任且贵方不应追究赛普拉斯之责任。贵方应赔偿赛普拉斯因赛普拉斯产品任何非预期用途产生或相关的所有索赔、费用、损失和其他责任，包括因人身伤害或死亡引起的主张，并使之免受损失。

赛普拉斯、赛普拉斯徽标、Spansion、Spansion 徽标，及上述项目的组合，WICED，及 PSoc、CapSense、EZ-USB、F-RAM 和 Traveo 应视为赛普拉斯在美国和其他国家的商标或注册商标。请访问 cypress.com 获取赛普拉斯商标的完整列表。其他名称和品牌可能由其各自所有者主张为该方财产。