AN82156

PSoC® 3、PSoC 4 和 PSoC 5LP — 使用 UDB 数据路径对 PSoC Creator™组件进行设计

作者：Todd Dust 和 Greg Reynolds

相关项目：有

相关器件系列：CY8C3xxx、CY8C5xxx、CY8C42xx

软件版本：PSoC Creator 3.2

要获取相关材料的完整列表，请单击此处。

AN82156 介绍了如何使用 PSoC 3、PSoC 4 和 PSoC 5LP 通用数字模块（UDB）数据路径对 PSoC Creator 组件进行设计。基于数据路径的组件可以实现常用的功能，如计数器、脉宽调制（PWM）、移位器、UART 和 SPI。使用这些组件可以创建自定义的数字外设和执行数据管理任务，从而能够减轻 CPU 的负担。本文档也说明了如何使用 PSoC Creator UDB 编辑器工具来创建、查看和修改数据路径示例。

目录

1 简介 2
2 传统 PLD 与 PSoC UDB 2
3 数据路径与基于 PLD 的设计 3
4 数据路径的架构与特性 4
4.1 动态配置 RAM （CFGRAM） 4
4.2 ALU 5
4.3 寄存器 5
4.4 条件运算符 5
4.5 输入和输出 5
4.6 链路 6
5 基于数据路径的组件 6
5.1 数据路径配置工具 6
5.2 UDB 编辑器 7
5.3 选择正确的工具 8
6 项目 1 — 8 位递减计数器 8
6.1 8 位计数器组件的详情 8
6.2 8 位计数器组件的创建步骤 11
6.3 将计数器改为 PWM 20
6.4 添加参数 24
6.5 添加头文件 27
6.6 将 PWM 扩展到 16 位 28
6.7 PSoC 3 的 16 位组件头文件 30
7 项目 2 — 递增/递减计数器 31
7.1 更多的详细信息 31
7.2 创建示例项目的步骤 31
8 项目 3 – 简单 UART 40
8.1 TX UART 组件的详细信息 40
9 从 UDB 编辑器移植到数据路径配置工具 47
10 汇总 47
11 相关资源 48
11.1. 应用笔记 48
11.2. 知识库文章 48
11.3. 技术参考手册《TRM》 48
11.4. 视频 48
12 关于作者 49
13 附录 A — 使用数据路径配置工具示例 50
13.1 项目 1 — 8 位递减计数器 50
13.2 项目 2 – 16 位 PWM 67
13.3 项目 3 — 递增/递减计数器 72
13.4 项目 4 — 简单的 UART 79
13.5 项目 5 — 并行输入和并行输出示例 82
14 附录 B — 数据路径配置工具说明书 88
14.1 动态配置 RAM（CFGRAM）部分 90
14.2 静态配置部分 93
14.3 设置初始寄存器值 100
14.4 数据路径链接 100
14.5 数据路径寄存器的固件控制 102
14.6 其他信息 102
15 附录 C — 自动生成的 Verilog 代码 103
15.1 PSoC Creator 生成的新 Verilog 文件 103
15.2 带有新数据路径实例的 Verilog 文件 103
15.3 进行 SimpleCntr8 修改后的 Verilog 文件 106
16 附录 D — 带有 PI 和 PO 的 24 位数据路径示例 109

1 简介

PSoC® 3、PSoC 4 和 PSoC 5LP （以下简称为 PSoC）不仅仅是微控制器。通过使用 PSoC 您还可以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。

将独立 CPLD 中的 CPLD 设计移植到 PSoC PLD 同复制和粘贴 Verilog 代码一样简单。AN82250 提供了实现该操作的详细说明。

然而，PSoC 的 PLD 规模比一般的 CPLD 或 FPGA 小，许多大型设计不能直接被移植到 PSoC 上。为了解决该问题，可以使用通用数字模块（UDB）数据路径。数据路径用于实现某些复杂的功能，释放 PLD 作为状态机和胶连逻辑使用。因此，数据路径大大降低了您的 Verilog 代码大小。

数据路径的核心是一个 8 位的算术逻辑单元（ALU）。该 ALU 执行加法、减法、OR、XOR、AND、递增、递减和移位等运算函数。与该 ALU 相关的各项还有某些寄存器和条件比较模块。可以将各数据路径连接起来用于执行宽度为从 1 到 32 位的操作。

将数据路径与 PLD 结合起来使用，您可以创建复杂的自定义数字外设。这些外设被捕获在 PSoC Creator 组件内。现在，PSoC Creator 拥有一大批使用 UDB 数据路径的数字组件。但也可能在标准组件中没有提供您正在寻找的功能。本应用笔记向您介绍如何创建你自己的使用数据路径专属组件。

该笔记是高级应用笔记，假定您已经熟悉使用 PSoC Creator 来开发应用。

如果您未了解 PSoC，请查看以下文档中的指导内容：

- AN54181 — PSoC 3 入门
- AN79953 — PSoC 4 入门
- AN77759 — PSoC 5LP 入门

如果您未了解 PSoC Creator，请参考：

- PSoC Creator 主页

此外，本应用笔记假定您掌握了数字设计和 Verilog 的基本知识。如果您尚未了解这些概念，请参考：

- AN81623 — PSoC 数字设计的最佳实践
- KBA86336 — PSoC 的 Verilog 基本知识
- AN82250 — 使用 Verilog 实现可编程逻辑设计。

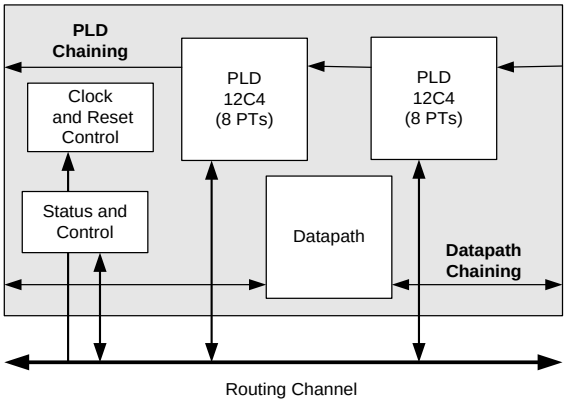
要获取相关数据路径设计资源的列表，请查看相关资源一节。

2 传统 PLD 与 PSoC UDB

PSoC 为优化的可编程器件，可以匹配或超过巨大可编程逻辑产品的功能。PSoC 的设计不适用于直接集成大型 FPGA 或 CPLD 实现。相反，PSoC 包含一组小型、快速、低功耗的可编程数字模块或 UDB。

每个 UDB 由两个小型 PLD 组成，即数据通路模块和状态与控制逻辑，如图 1 所示。PSoC 可包含最多 24 个 UDB。

图 1. 简化的 UDB 框图



数据通路模块可以实现简单的功能，如递增、递减、加法、减法、按位逻辑计算以及移位等。与 PLD 结合使用时，数据路径可作为更复杂的功能。这样的结合可以帮您轻松实现常用的功能，如计数器、PWM、移位器、UART 或 I^2C 接口。

这种结合可实现不适合 PSoC PLD 的 CPLD 或 FPGA 部分设计。使用针对数据路径优化的 Verilog 复杂功能可以提高 PSoC 数字资源的使用率。这些功能包括加法、减法、移位等运算函数。与 PLD 相比， 数据路径能够更有效地实现这些函数。

3 数据路径与基于 PLD 的设计

如果使用数据路径（而不是 PLD）执行算术操作，在 UDB 中实现的函数通常要求占用较少的资源。例如，考虑分别在 PLD 和数据路径中实现下面的 8 位算术和逻辑操作，如表 1 所示。

表 1. PLD 和数据路径资源的使用情况

功能	PLD 中的资源消耗		数据路径中的资源消耗	
	PLD	使用的百分比（%）	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%

实现这些数字功能时，应该将数据路径和 PLD 结合使用，以获得最高效果。

根据经验，下面是使用 UDB 资源的最好方法：

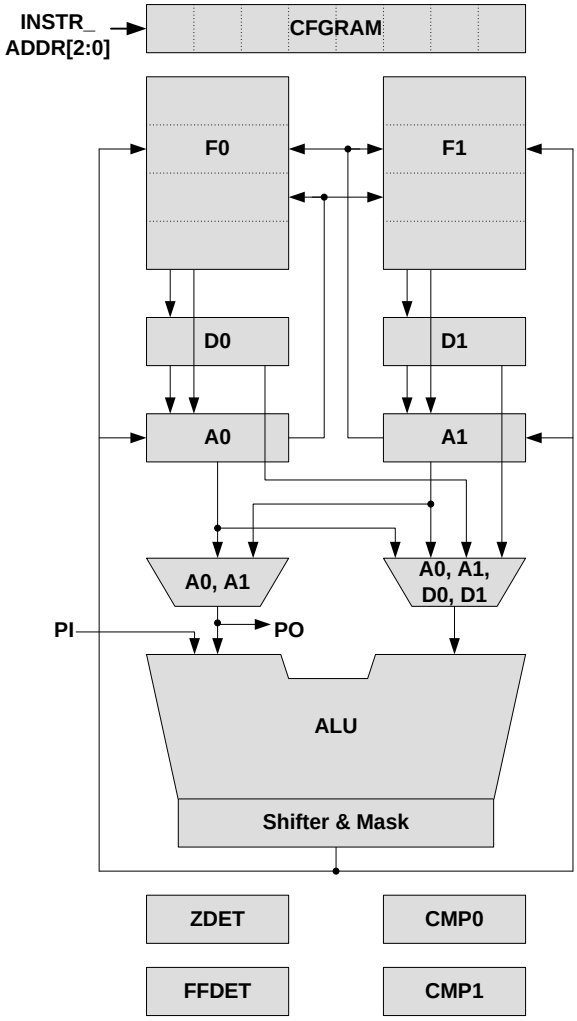
- PLD：组合逻辑、胶合逻辑和状态机。（更多有关 PLD 实现的信息，请查看 AN82250 — 设计 PSoC 3、PSoC 4 和 PSoC 5 的可编程逻辑和 AN81623 — PSoC 3、PSoC 4 和 PSoC 5 数字设计的最佳实践。）
- 数据路径：具有 CPU、计算、时序、通信以及宽度为字节或字的比较等功能的 FIFO 接口。

您可以在 PSoC PLD 中实现复杂的功能，但是您可能会很快就耗尽资源。

4 数据路径的架构与特性

数据路径包含一个可配置的 8 位 ALU、相关的比较和条件生成电路以及用于 ALU 操作和 CPU 交互的各种寄存器，如图 2 所示。此外，还有专门的移位和屏蔽模块。

图 2. 简化的数据路径框图



更多有关框图的详细信息，请参考附录 B。

4.1 动态配置 RAM （CFGRAM）

动态配置 RAM（CFGRAM）中可存储 8 种数据路径指令（或配置）。这些指令定义了数据路径的功能和连接，包括 ALU 函数、ALU 输入、寄存器写指令、移位操作、比较操作等。执行每条指令占用一个时钟周期。

由于存在八个独立指令，因此需要三条指令地址行（INSTR_ADDR[2:0]）。地址信号在数据路径时钟的每一个上升沿上确定使用哪条指令。

这三个地址行可以由 PLD 逻辑或外部信号驱动。一般使用 PLD 创建状态机。然后，该状态机中的逻辑被路由到这些地址输入端，用于控制数据路径指令。

这三个地址行有多个名称，因此会引起混淆。请注意，cs_addr[2:0]、RAD[2:0]和 INSTR_ADDR[2:0]指的是同一个地址。本应用笔记使用的名称为 INSTR_ADDR。

4.2 ALU

ALU 能够执行八种通用功能：

- 递增
- 递减
- 加法
- 减法
- 按位与
- 按位或
- 按位异或
- 通过

功能选择操作是由存储在 CFGRAM 中的指令按周期控制的。可以在 ALU 的输出上使用独立移位（左移、右移）和掩码操作。

4.3 寄存器

每个数据路径模块都有四个 8 位工作寄存器和两个 FIFO：

- A0 和 A1 — 累加器寄存器通常保存 ALU 操作需要使用的数据，也可用于存储来自 Dx 寄存器或 FIFO 的数据。
- D0 和 D1 — 数据寄存器通常保存静态数据，如计数器的起始值或重载值。
- F0 和 F1 — 这些寄存器为 4 字节 FIFO，可作为源或目标缓冲器使用。这些寄存器主要用于将 CPU 或 DMA 连接到数据路径。

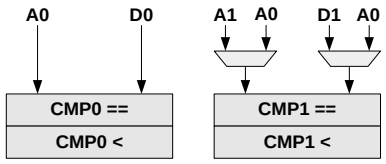
存储在 CFGRAM 中的指令确定了在调用每条指令期间如何使用这些寄存器。

通过使用 CPU（以及 PSoC 3 和 PSoC 5LP 中的 DMA）您可以对所有数据路径寄存器进行读写操作，但是，请尽可能使用 FIFO 实现该操作。累加器寄存器异步运行，并且可以随时修改（甚至在 CPU 或 DMA 访问过程中）。FIFO 与 CPU/DMA 访问同步。

4.4 条件运算符

每个数据路径均有两种比较功能，即：“小于”和“等于”，如图 3 所示。比较模块可以使用位屏蔽来执行按位比较。

图 3. 数据路径比较模块



A0 和 A1 寄存器有“零”（ZDET）和“全一”（FFDET）检测功能。FIFO 具有“已满”和“空白”状态信号。

4.5 输入和输出

UDB 被数字信号互连（DSI）（可编程数字布线的扩展结构）所包围。通过 DSI，可以连接 UDB 内的信号以及 UDB 阵列和 PSoC 中的其它模块间的信号。

有三种数据路径输入类型：指令、控制和数据。指令输入（INSTR_ADDR[2:0]）从 CFGRAM 中选择当前的数据路径。控制输入从 FIFO 加载数据寄存器，并将累加器输出捕获到 FIFO 内。数据输入包括 shift in（SI）和 carry in（CI）两种。数据路径最多有六个输入。这六个输入可来自 DSI 的芯片上连接的任何位置。

数据路最多有六个输出。这些输出可以连接到各种数据路径状态信号，包括 FIFO 状态、比较状态、上溢检测、carry out 和 shift out。然后，这些输出被连接到 DSI，在这里它们被路由到其他片上资源。

4.6 链路

每个数据路径可以执行 8 位操作。可以将多个数据路径串联起来，以创建宽度为 1 到 32 位的功能。

可以链接移位、进位、捕获和其他条件信号，以实现精度更高的算术和移位功能。这些链接信号不占用数据路径输入和输出。

请查看 PSoC 3 架构技术手册，获取 UDB 和数据路径的完整规范。

5 基于数据路径的组件

通过自定义 PSoC Creator 组件使用数据路径是最有效的方法。赛普拉斯提供了*组件创建指南（CAG）*，全面介绍了创建组件的整个过程。要想打开 PSoC Creator 中的指南，请依次选择 **Help > Documentation > Component Author Guide**。

通过两种方式可以使用PSoC Creator实现基于数据路径的组件。您可以编写Verilog文件并使用数据路径配置工具来配置该数据路径。如果不想编写Verilog，也不想使用数据路径配置工具，那么您可以使用UDB编辑器。每种方法都有其自身的优点和缺点。Verilog方法通常比UDB编辑器方法更先进，而与Verilog方法相比，UDB编辑器方法更简单。

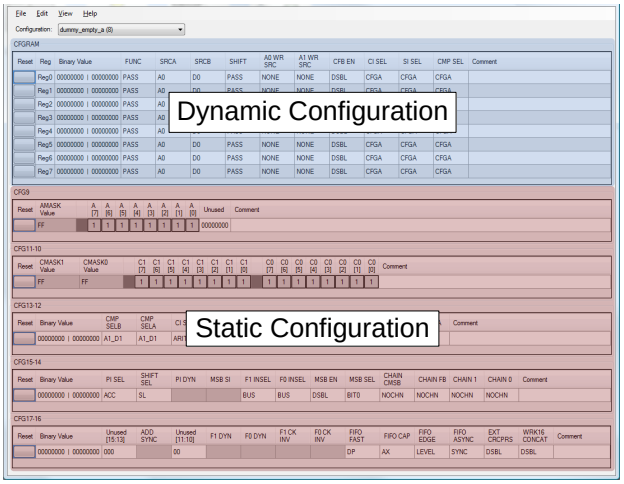
5.1 数据路径配置工具

数据路径配置工具是一个应用程序，通过它您可以创建、查看、修改并删除 Verilog 文件中的数据路径实例。该工具将解析 Verilog 文件，并将每个数据路径显示为一个实体。这个实体在工具中被称为“配置”。每个配置表示一个单一的物理数据路径。

如果 Verilog 文件中没有任何数据路径，您可以创建一个新的数据路径并将它添加到文件中。您可以将多个数据路径添加到一个文件中，并将这些路径串联起来，以创建多字节功能。

通过一个图形用户接口（GUI）显示了各种数据路径，如图 4 所示。GUI 显示为数据路径的 CFGRAM 和静态配置寄存器的内容。

图 4. 数据路径配置工具接口



附录 B — 数据路径配置工具“说明书”包含对 GUI 字段的说明。各个 GUI 字段与 PSoC 器件中的寄存器相对应。PSoC 3 架构技术参考手册、PSoC 5LP 架构技术参考手册和 PSoC 4 架构技术参考手册都详细介绍了这些寄存器。

您可以通过 PSoC Creator 启动数据路径配置工具（依次选择 **Tools > Datapath Config Tool…**）。

本应用笔记主要不是介绍数据路径配置工具，而是重点介绍 UDB 编辑器。附录 A 介绍的是如何使用数据路径配置工具来实现本应用笔记中的所有示例。该附录也提供了使用并行输入和并行输出的一个示例。

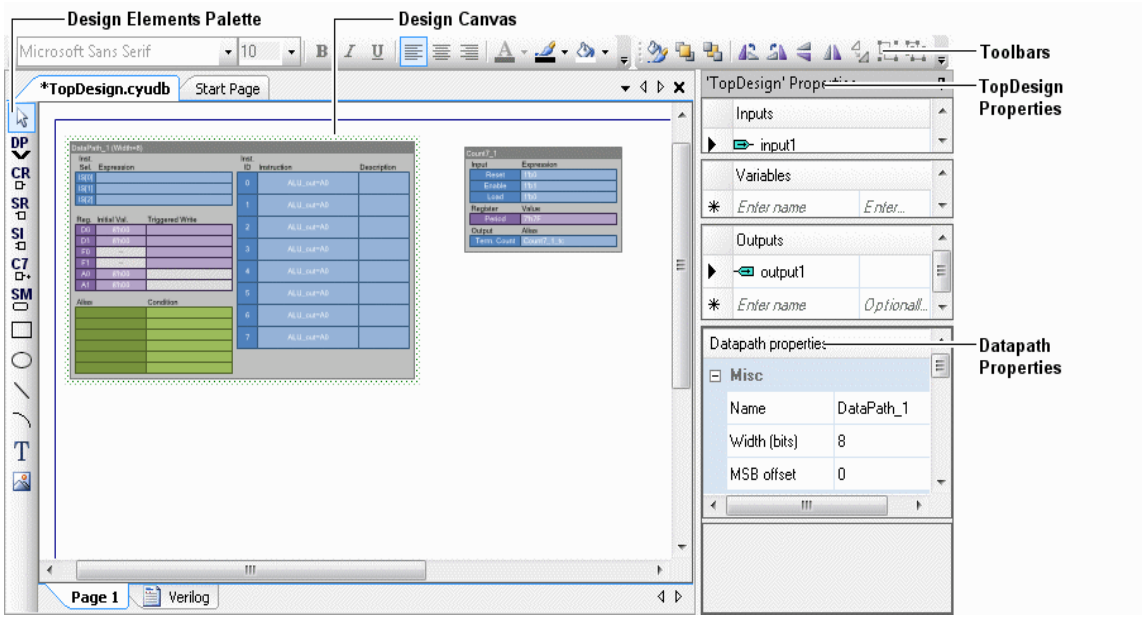
5.2 UDB 编辑器

UDB编辑器是一个图形工具，可根据设计进行构建UDB，如图5所示。使用它可以设计各种组件，而不需要编写Verilog或使用更先进的数据路径配置工具。通过UDB编辑器可以访问UDB中的各种元素，包括：数据路径、控制寄存器、状态寄存器、状态中断寄存器、count7计数器和PLD；这些元素都以图形形式表示。

UDB 编辑器允许您使用较少的数字逻辑或 Verilog 的知识来设计基于 UDB 的硬件。对 UDB 编辑器进行设计，这样您可以进行拖放，然后配置您的硬件，而无需编写代码。然后，该工具会将您的设计实时转换为 Verilog — 给您一次看到 UDB 模块如何转换为 Verilog 的机会。

因为UDB编辑器是一个图形工具，所以它对Verilog知识和UDB的复杂细节要求较少。但这样会降低硬件的灵活性和粒度控制，这是简单化抽象性的结果。它也不具有某些更高级的UDB功能，因此限制将其使用于复杂的设计中。

图5. UDB编辑器



UDB编辑器的结构如下：

Pages（页） — 打开UDB编辑器文档时，您会看到一个与图像页相同的可编辑页。这是您的设计图纸，用于放置和配置UDB元素。通过在**Page 1**选项卡中添加更多页，UDB编辑器会为您提供页数量。然后，这些页互相转换（跟进行一个设计相同）。

Verilog — 在**Page**选项卡的旁边，您可以找到**Verilog**选项卡。这是设计中被转换的硬件的只读视图。动态更新Verilog选项卡，以便在设计中发生变化时，它也会更新Verilog代码。该代码是不可编辑或删除的，所以您要在设计图纸中进行适当的改变，才能看到想要的结果。如果您想要使用Verilog来编辑设计，您可以将该代码复制并粘贴到Verilog文件内。

Design Properties（设计属性） — 在设计图纸的右侧，您可以找到各种设计属性，通过这些属性您可以配置输入、输出以及设计中所使用的变量。设计完成时，输入和输出会变成组件终端。当设计中含有数据路径时，设计属性窗口也适用于设置全局数据路径配置。

Design Elements Palette（设计元素控制板） — 设计元素控制板位于设计图纸的左侧。它是一个用于选择您设计中包含的UDB元素的菜单。它们为：

- 数据路径（DP）
- 控制寄存器（CR）
- 状态寄存器（SR）
- 状态中断寄存器（SI）
- count7 计数器（C7）
- 状态机的状态（SM）

5.3 选择正确的工具

通过使用 UDB 编辑器可以几乎实现所有设计。UDB 编辑器实现了数据路径的最通用性能。但还有一些性能它不能实现。如果您需要使用这些性能，则必须使用数据路径配置工具。不受 UDB 编辑器支持的性能包括：

- FIFO 动态控制
- FIFO 时钟反转
- 并行输入和并行输出。附录 A 中介绍了一个如何使用数据路径配置工具实现该性能的示例。
- 循环冗余校验（CRC）
- 伪随机序列（PRS）
- 可选 Carry In
- 动态 Carry In

在大多数设计中，不需要这些性能。因此，最好使用 UDB 编辑器进行操作。因为使用它更简单。在开发过程中，如果您发现 UDB 编辑器不具有您所需要的性能，那么可以复制和粘贴 UDB 编辑器生成的 Verilog 代码，然后使用数据路径配置工具修改 Verilog 文件。

注意：UDB 编辑器不能读取数据路径配置工具创建的 Verilog 文件。

如果您确定需要高级性能，或想编写自己的 Verilog，那么可以使用可选择数据路径配置工具。附录 A 显示了使用数据路径配置工具（而不是 UDB 编辑器）的所有示例。

如果 UDB 编辑器正是您需要的工具，本应用笔记其余部分将详细说明如何使用 UDB 编辑器创建基于数据路径的简单组件。

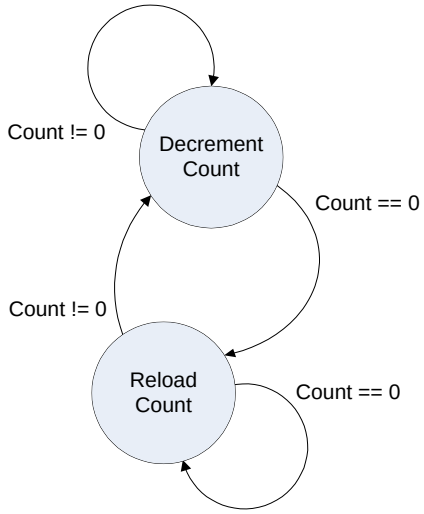
6 项目 1 — 8 位递减计数器

该项目的目的是向您介绍创建一个基于数据路径的简单组件的具体步骤。首先，需要创建一个 8 位的递减计数器组件，然后将它改为 8 位 PWM，最后将它修改为 16 位 PWM。

6.1 8 位计数器组件的详情

我们可以使用具有两个状态的状态机来表示一个简单的递减计数器，如图 6 所示。当数据路径时钟的上升沿到来时，将发生状态转换。

图 6. 简单计数器的状态图



启动后，计数器的初始值将递减。递减到零时，将触发某个事件，并重复该周期。在数据路径中可以轻松实现这种计数器。

实现该计数器时，您只需要两条 CFGRAM 中存储的数据路径指令。第一条指令用于递减存储在某个工作寄存器中的计数值。第二条指令重载存储在另一个寄存器中的初始值。不使用剩下的六条 CFGRAM 指令。

该示例中使用的两个数据路径寄存器为：

- A0 — 保存计数值，并且由 ALU 递减。
- D0 — 保存计数值为零时被重载到 A0 内的值。

您需要通过一种方法来确定数据路径在每个时钟周期内所执行的指令：加载或递减指令。图 6 显示了由计数值控制的状态转换，即判断计数值是否为零。

数据路径有一个零检测器模块（ZDET），用于监控 A0 和 A1 中的数据值。该模块有两个输出 z0 （A0 == 0）和 z1（A1 == 0），分别表示 A0 和 A1 的状态。数值为零时，相应的输出为高；该值为非零时，相应的输出为低。

在该示例中，z0 输出确定需要执行的指令。对各条指令进行寻址时将需要三位。可以将位 0 使用于 z0，并将其它位置于低电平。表 2 是转换表。

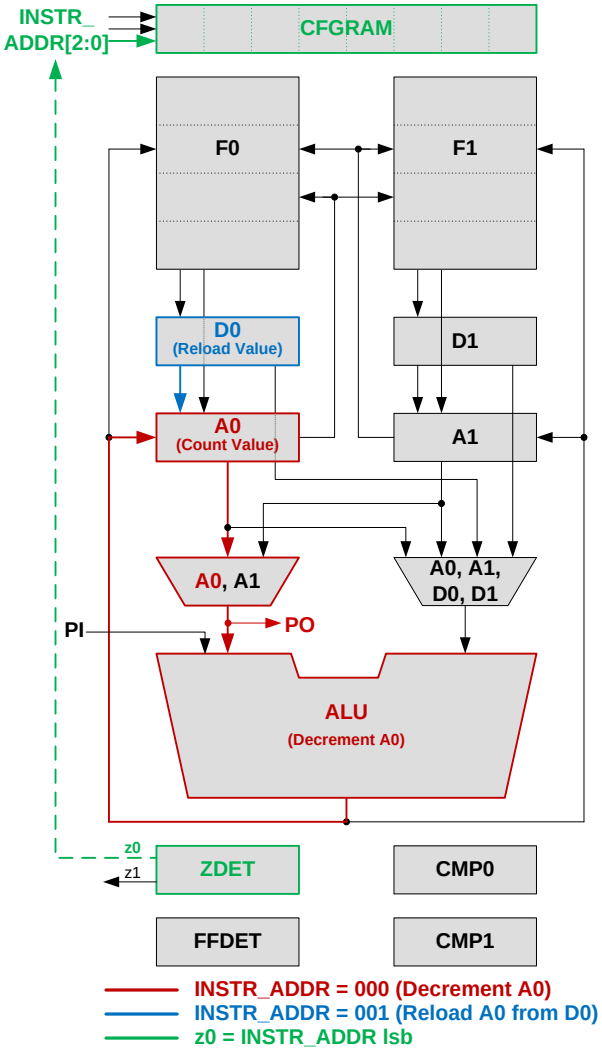
表 2. 数据路径指令

CFGRAM 指令	指令地址位（INSTR_ADDR）			操作
	2	1	0	
0	0	0	z0 = 0	递减计数
1	0	0	z0 = 1	重载计数
2-7	X	X	X	未使用

A0 中的计数值为 0 时，z0 变为‘1’。这样，下个数据路径指令便为“重载计数值”。当从 D0 将计数值重载到 A0 时，z0 变为‘0’，下一个配置将为“递减计数值”。

要想了解它的运行方式，请查看加亮显示的数据路径框图，如图 7 所示。

图 7. 加亮显示的简单计数器框图

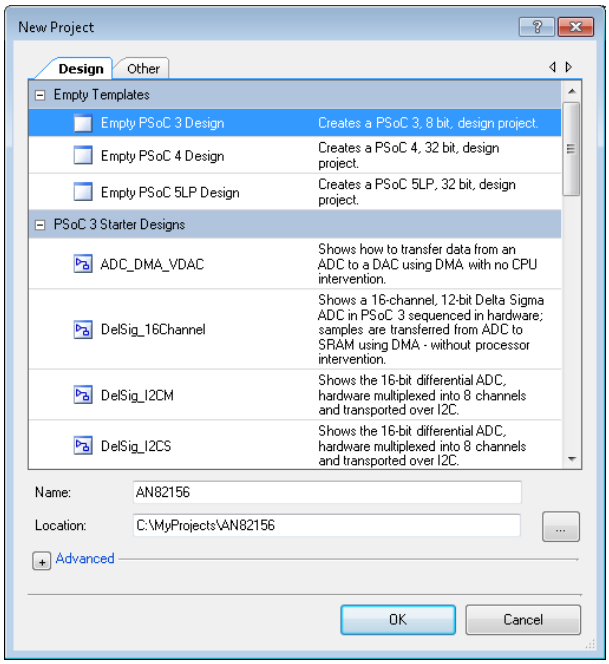


6.2 8 位计数器组件的创建步骤

然而，在这个示例中，请创建一个新的项目。

1. 启动 PSoC Creator 并创建名为 *“AN82156”* 的项目，如图 8 所示。“AN82156”工作区也被默认创建。

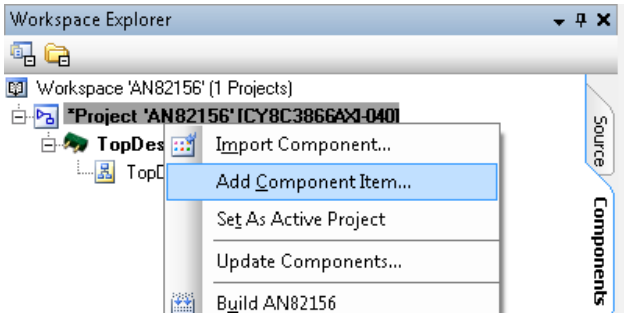
图 8. 添加新项目



这时，您可以创建一个新的库， 用于存储您的组件。PSoC Creator 3.0 和 3.0 SP1 中的 UDB 编辑器不能与库配合使用。现在，组件在一个项目中创建而不是库。该问题将在 PSoC Creator 的下个版本发布中予以修正。

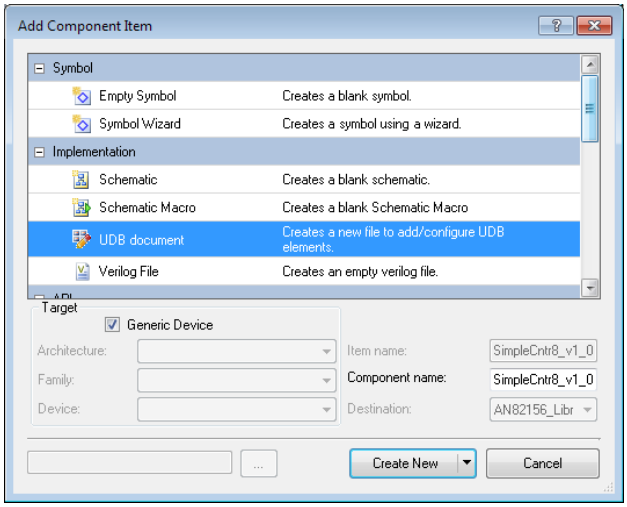
2. 切换到 **Workspace Explorer** 中的 **Components** 选项卡，然后右击 **Project ‘AN82156’**。从下拉菜单中，选择 **Add Component Item...**项，如图 9 所示。

图 9. 添加组件项



3. 选择 **UDB Document**，并将组件命名为“*SimpleCntr8_v1_0*”，如图 10 所示。

图 10. 添加 UDB 文档

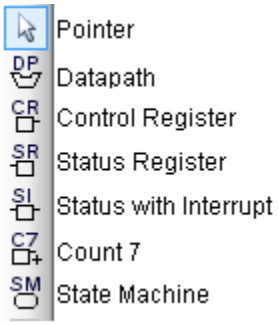


组件名称应该包含版本编号。将“_vX_Y”标签附加到组件名称，其中，‘X’表示主要版本、‘Y’指的是次要版本。PSoC Creator 具有版本控制功能，通过它您可以跟踪并使用组件的多个版本。

4. 点击 **Create New**，创建 UDB 文档原理图。

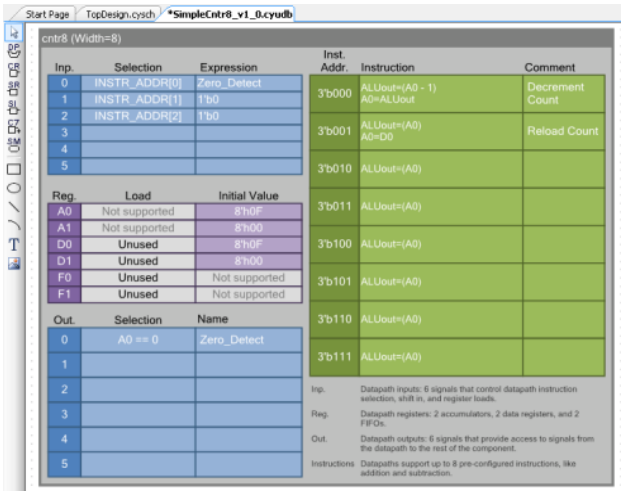
UDB 文档原理图是一张图纸，您可以在该图纸上创建基于 UDB 的组件。可以将 UDB 元素拖放到该原理图上，就像拖放到普通的 PSoC Creator 原理图内一样。您拖放到该原理图上的组件显示在该原理图的左侧，请参考图 11。

图 11. UDB 元素



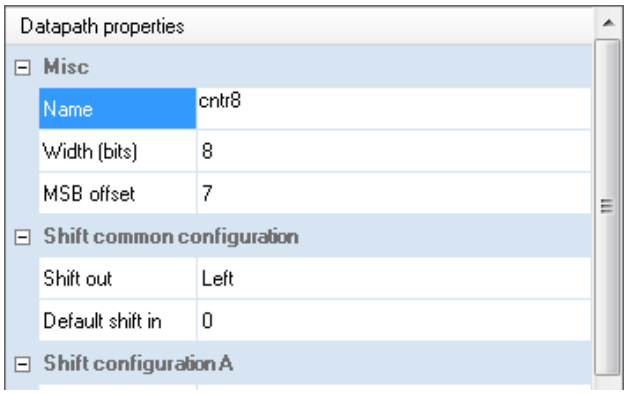
5. 将数据路径元素拖放到 UDB 文档原理图内，请参考图 12。

图 12. 数据路径元素



6. 单击数据路径实例。UDB 编辑器的右侧是属性窗口，在该窗口底部查找 **Datapath properties**（数据路径属性）。将数据路径的名称从‘*Datapath_1*’更改为‘*cntr8*’，如图 13 所示。

图 13. 数据路径属性

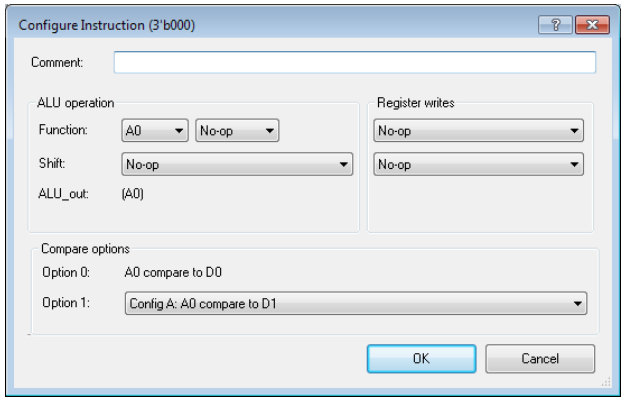


7. 在 *cntr8* 上，双击带 Inst 的绿色框。从而打开配置对话框的地址 3’b000 （参考图 14）在图 15 中显示。

图 14. 指令零

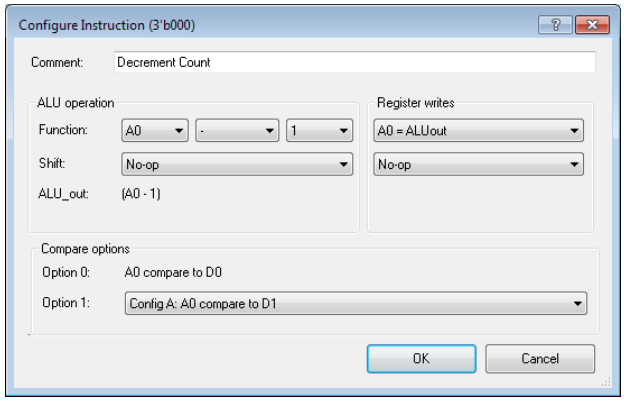


图 15. 空指令配置对话框



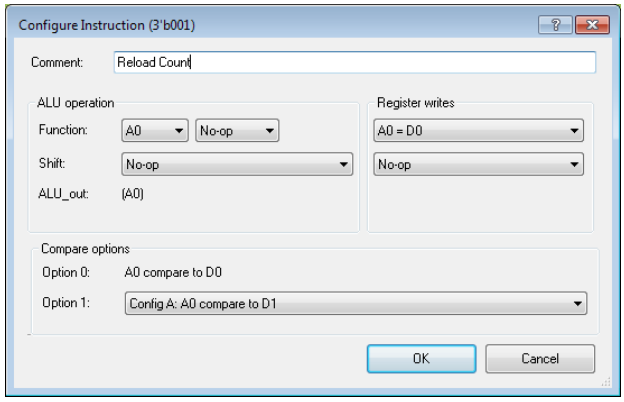
8. 在该对话框中配置第一个数据指令。返回查看表 2，并了解到对于指令零，数据路径将递减（存储在 A0 内的）计数值。
9. 将 **ALU operation Function** 设置为 A0 – 1（递减）。在 **Register Write** 字段中设置 *A0 = ALUout*。这样可配置第一条指令为递减 A0，并将其回写到 A0 内。您也可以将该指令作为一个注释进行说明：在该示例中，输入“Decrement Count.” 请参考图 16。

图 16. 指令零的配置



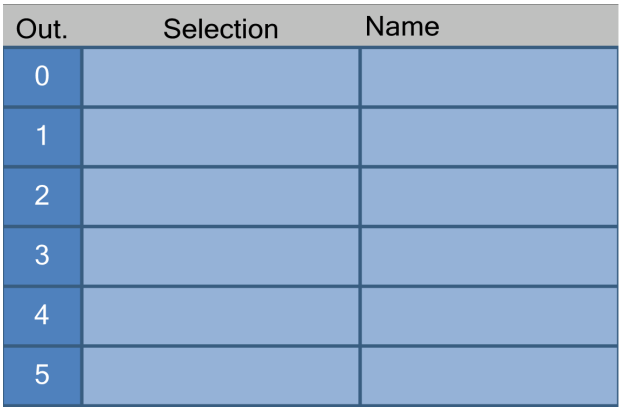
10. 接下来，配置指令地址 3’b001。双击该框。如表 2 所列，指令 1 将存储在 D0 中的值重新加载到 A0 内。在 **Register Writes** 字段中设置 *A0 = D0*。请查看图 17。

图 17. 指令一的配置



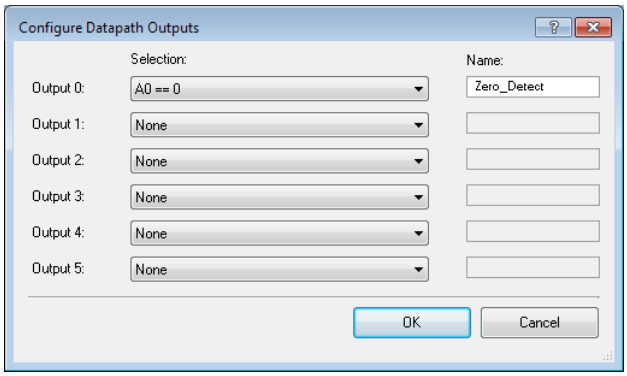
11. 接下来我们需要选择数据路径在每个时钟周期内所执行的指令。例如，我们使用集成到数据路径中的零检测器。首先，我们将零检测器信号输出给一个已被命名的标签。
12. 双击图 18 中显示的蓝色输出框。

图 18. 数据路径输出



13. 将出现配置数据路径输出对话框，如图 19 所示。
14. 将 **Output 0** 配置为 *A0 == 0*，并将 **Name** 设置为‘*Zero_Detect*’。

图 19. 数据路径输出配置



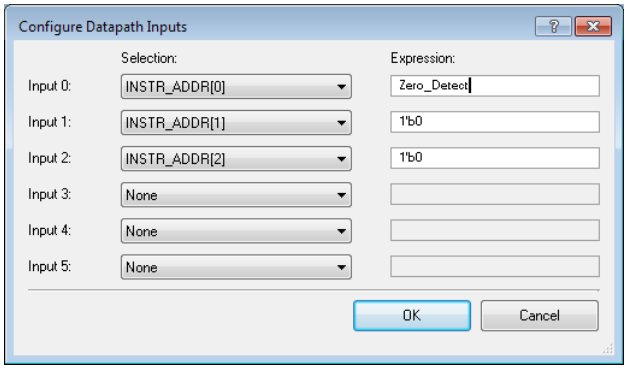
15. 信号 Zero_Detect 表示该数据路径的 z0 输出。当 A0 等于 0 时，Zero_Detect 为高电平，当 A0 不等于 0 时，Zero_Detect 为低电平。
16. 将该信号路由到数据路径的指令地址。该信号将在数据路径的输入部分得到配置。
17. 双击图 20 中显示的蓝色输入框。

图 20. 数据路径输入



18. 对于 **Input 0**，请确保 **Selection** 被设置为 *INST_ADDR[0]*，并且 **Expression** 被设置为 *'Zero_Detect'*，如图 21 所示。

图 21. 数据路径输入配置



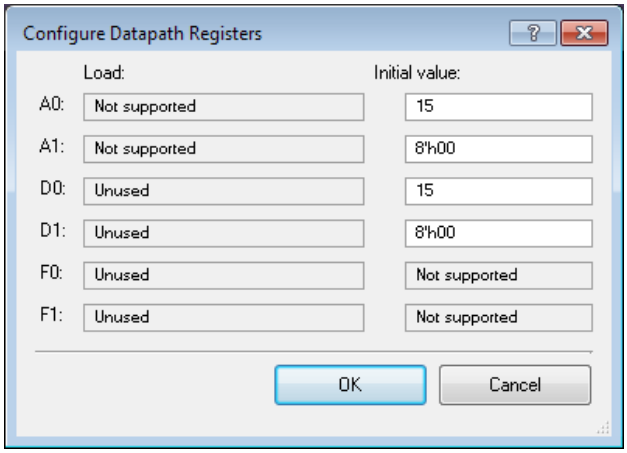
19. 通过使用 Zero_Detect 信号可以选择该数据路径将要执行的指令。Zero_Detect 为高电平时，数据路径将执行指令 1；该信号为低电平时，数据路径将执行指令 0。
20. 下面为计数器和周期寄存器定义某些初始值。
21. 双击图 22 中显示的灰色和紫色寄存器配置框。

图 22.数据路径寄存器

Reg.	Load	Initial Value
A0	Not supported	8'h00
A1	Not supported	8'h00
D0	Unused	8'h00
D1	Unused	8'h00
F0	Unused	Not supported
F1	Unused	Not supported

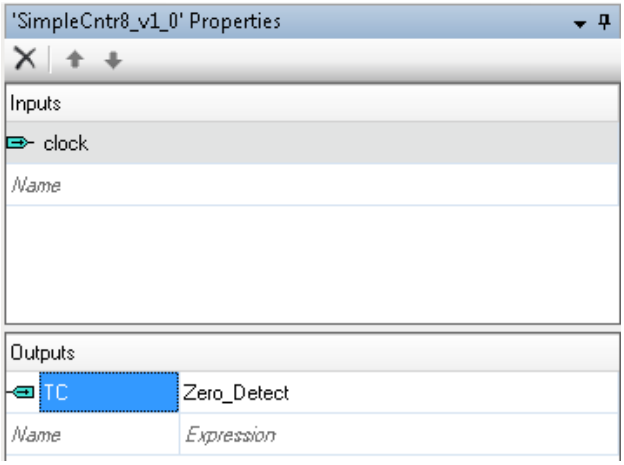
22. 将 A0 和 D0 的初始值设置为 15，请参考图 23。

图 23. 数据路径寄存器配置



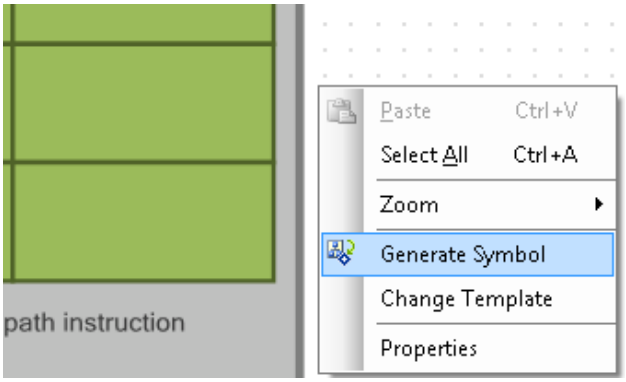
23. 我们已经对数据路径进行了配置。接下来，我们将定义组件输出。该输出是终端计数（TC）。当计数器为 0 时，该输出为高电平，其他情况下，该输出为低电平。
24. 在 **Outputs** 下的**'SimpleCntr8_v1_0' Properties** 窗口内，定义一个名为 *'TC'* 的新输出，并将 **Expression** 设置为 *'Zero_Detect'* ，如图 24 所示。

图 24. 组件输出



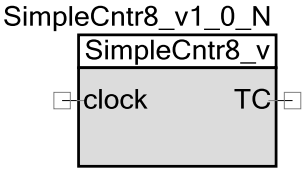
25. 为该组件生成符号。既为显示在您的顶层设计原理图中的信号。
26. 右击 UDB 文档（.cyudb）的空白区域，并选择 **Generate Symbol**，请参考图 25。

图 25. 生成符号



27. 原理图符号将如图 26 所示。

图 26. 原理图符号

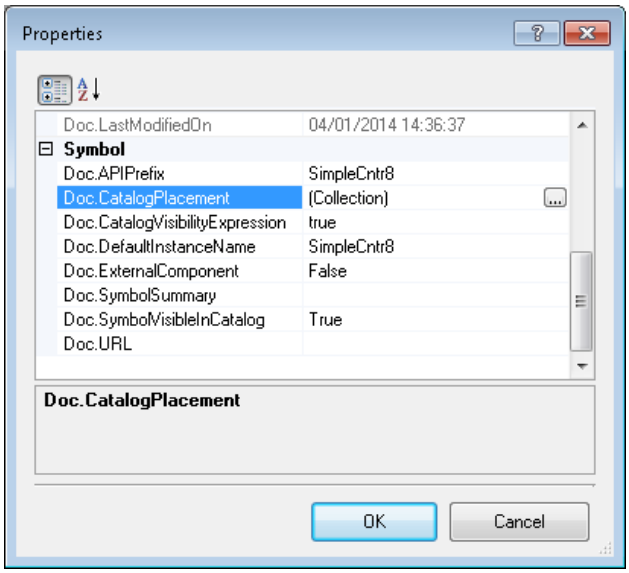


28. 在符号编辑器中，右击空白空间（而不是符号本身），然后从下拉菜单选择 **Properties**。

29. 在属性字段下的 **Symbol** 部分中输入所需的数值，如图 27 所示：

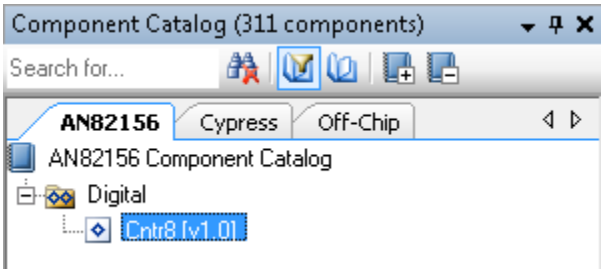
    - *Doc.APIPrefix = SimpleCntr8*
      该值作为该组件中所有 API 文件名的前缀。在该示例中不会生成任何 API，但是每当创建某个组件时，请在此处输入一个数值。
    - *Doc.CatalogPlacement = AN82156/Digital/Cntr8*
      通过点击‘...’按键打开 Catalog Placement 对话框，然后才能输入该值。PSoC Creator 使用该值定义组件目录的层次。第一项是指选项卡，在该选项卡下将显示目录中的组件。每一个后续的‘/’均表示一个子目录。组件目录必须至少包含一个子目录。上述的数值表示该组件是‘Cntr8’，它显示在 AN82156 选项卡下的‘Digital’子目录内。
    - *Doc.DefaultInstanceName = SimpleCntr8*
      这是组件被放置在原理图中时显示的默认名称。

图 27. 添加符号属性



30. 依次选择 **File > Save All**，确保保存了对项目进行的所有更改。
31. 现在可以使用该组件。在 *AN82156* 选项卡的组件目录下，可查看新的组件，如图 28 所示。

图 28. 组件目录中的新组件



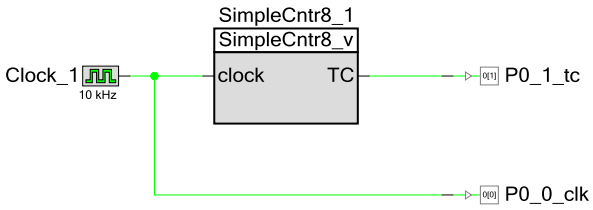
32. 在 Component Catalog 下能看到组件后，您可以将该组件放置在原理图中。该组件的使用情况与其它组件的相同。
33. 为了测试该组件，需要添加一个时钟源和计数器‘TC’输出的查看机制。
34. 将 Cntr8 组件放置在项目原理图中。
35. 将某个时钟组件连接至‘clock’终端，并将其频率设置为 10 kHz。可以使用其它工作频率，但是在示波器上易于观察 10 kHz 的频率。
36. 将一个数字输出引脚组件连接至‘clock’终端，从而在示波器上能够观察它。将该引脚命名为 P0_0_clk，并保持其他默认设置。

    注意：对于 PSoC 4，您不能直接将时钟路由到某个引脚。要想将时钟路由到某个引脚，请执行以下操作：

    a. 将数字输出引脚组件放置在您的原理图上。
    b. 在引脚定制器中选择 **Clocking** 选项卡。
    c. 将 **Out Clock:** 设置为 *External*。
    d. 返回 **Pins** 选项卡，并转到 **Output** 子选项卡。
    e. 在 **Output Mode:** 下选择 *Clock*。
    f. 点击 **OK**。
    g. 将时钟信号将连接到引脚组件上的 out_clk 终端。
37. 将一个数字输出引脚组件连接至‘TC’终端。将该引脚命名为 P0_1_tc，并保持其他默认设置。计数值为零时，该引脚将为高电平。

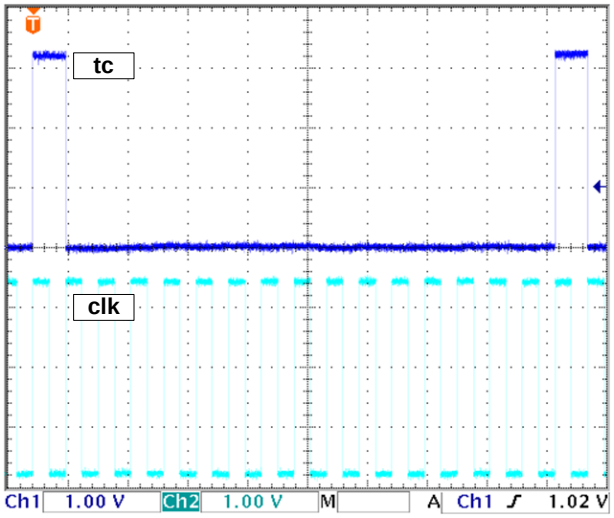
    图 29 显示的是完整的项目原理图。

图 29. 简单计数器的项目原理图



38. 在 cydwr 的 *Pins* 选项卡下，根据引脚的名称将它们分配给 P0[0]和 P0[1]。
39. 现在，您可以编译项目并编程 PSoC。您可以在引脚 P0[0]和 P0[1]上观察时钟和终端计数。
40. 保存项目，编译它并编程 PSoC。
41. 如果将示波器连接至输出引脚，您可以观察‘clock’和‘tc’输出，如图 30 所示。

图 30. 简单计数器的输出



42. 前面已将起始值 15 加载到 A0 内，因此计数周期为 16（从 15 到 0 算起）个时钟周期。A0 值为 0 时，‘TC’引脚在一个时钟周期内处于高电平，因为此时，A0 将重载 D0 的值。A0 值非零时，‘TC’为低电平，并且配置再次回转递减 A0。

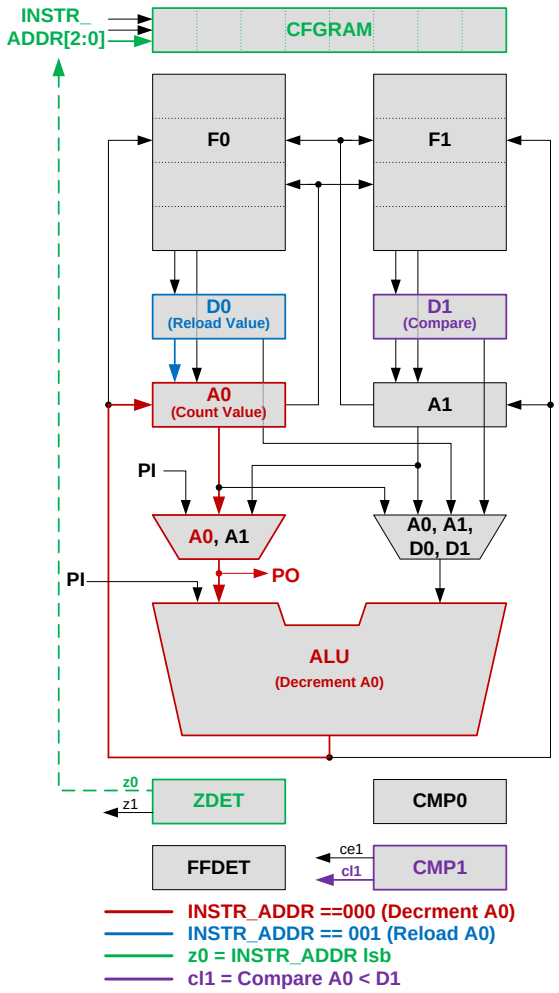
    您刚设计完成第一个基于数据路径的组件，而不需要编写任何代码。

6.3 将计数器改为 PWM

PWM 是一个具有比较功能的计数器。为了创建 PWM，您需要使用某个方法将 A0 中的值与另一个固定值进行比较。可以使用 D1 寄存器保存固定比较值，并通过设置比较模块来检查 A0 是否小于 D1。

可以通过图 31 中加亮显示的框图查看该方法：

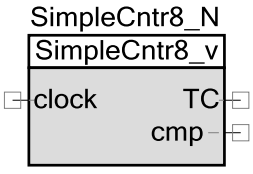
图 31. 加亮显示简单的 PWM 框图



在该示例中，对前一节中构建好的 8 位计数器组件进行修改。您可以从一个空白组件开始，也可以复制先前的计数器组件。下面各步骤假定您对现有的计数器进行修改。

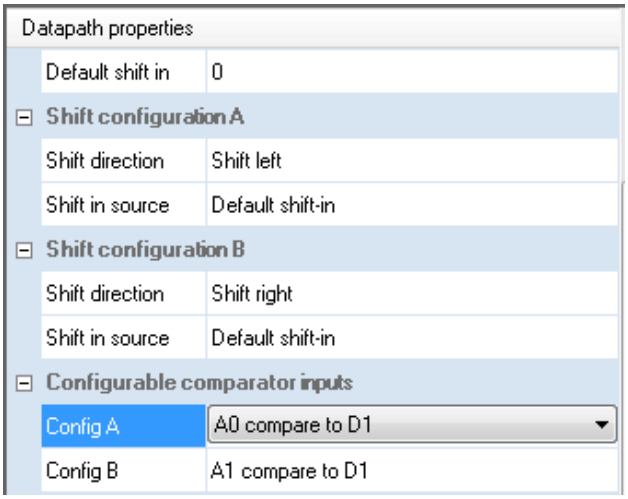
1. 打开 SimpleCntr8_v1_0 的组件符号（.cysym）文件。
2. 按住‘O’，以添加一个数字输出终端。将该终端命名为‘cmp’，并将它放置在图 32 所示的位置。

图 32. 带有‘cmp’终端的组件符号



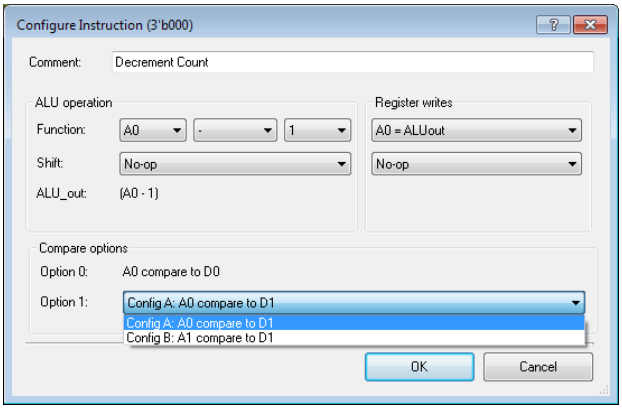
3. 依次选择 **File > Save All** 以保存所有更改内容。
4. 返回 UDB 文档（.cyudb），并点击该数据路径。在 **Datapath properties** 窗口中，请确保 **Configurable comparator inputs** 的 **Config A** 被设置为 *A0 compare to D1*；请参考图 33。

图 33. 数据路径比较配置



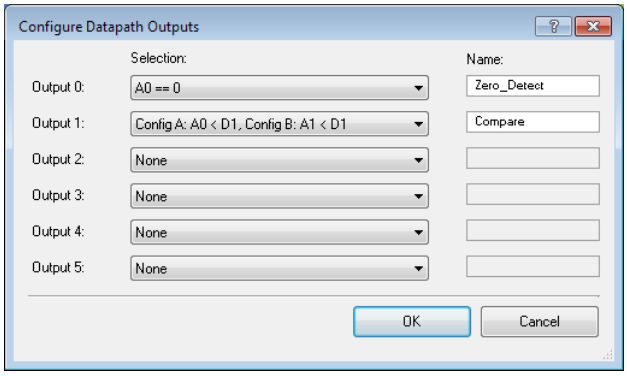
5. 请确保对于指令 0 和指令 1，已经将 **Compare Option 1** 设置为 *ConfigA: A0 compare to D1*，如图 34 所示。要想打开该对话框，请双击相应的绿色指令框。

图 34. 配置比较选项



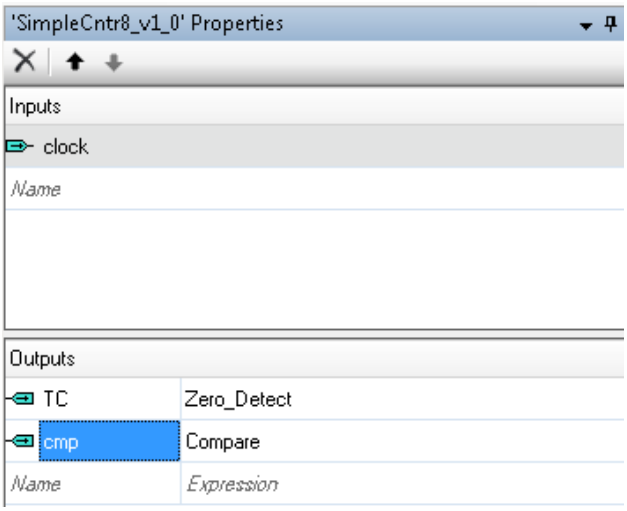
6. 双击输出框并将 **Output 1:** 配置为 *Config A: A0 <D1…*，然后将 **Name** 设置为‘*Compare*’，如图 35 所示。

图 35. 配置比较输出



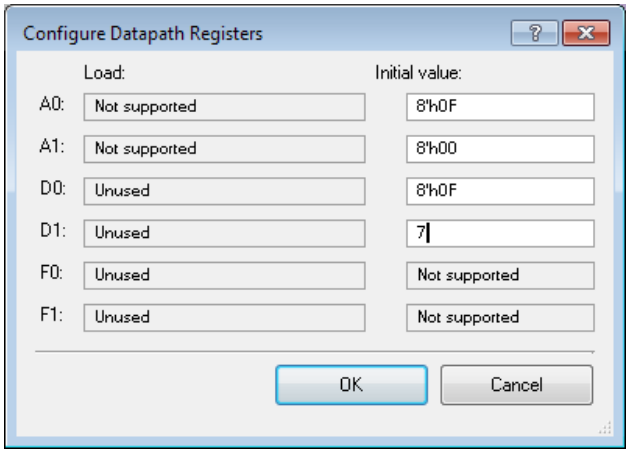
7. 定义一个名为‘*cmp*’的新组件输出，并将 *Expression* 设置为 Compare（比较）；请参考图 36。

图 36. 比较输出



8. 最后，为比较寄存器（D1）配置初始值。
9. 打开 **Configure Datapath Registers**（配置数据路径寄存器）对话框。将 D1 的初始值设置为 7，如图 37 所示。

图 37. D1 的初始值

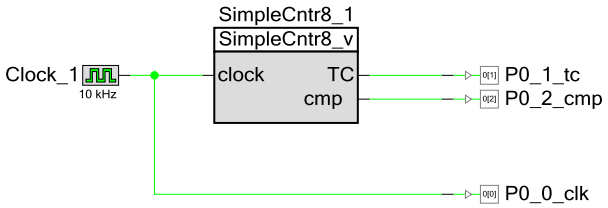


10. 依次选择 **File** > **Save All**。
11. 在该配置中，当 A0 小于 D1 时，比较模块输出将为高电平；当 A0 大于 D1 时，该输出将为低电平。

    在 Component Catalog 窗口中的 AN82156 选项卡下仍然可以查看 PWM 组件和符号。组件在项目原理图中被自动更新。

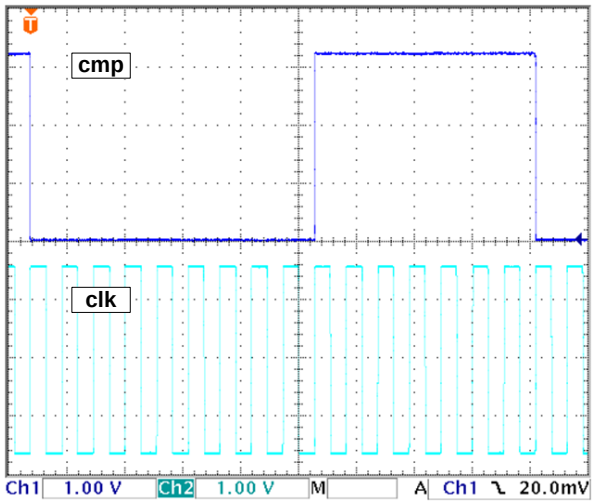
12. 添加一个输出引脚，并将它连接至‘cmp’终端。将该引脚命名为 *P0_2_cmp* ，并将它分配给引脚 P0[2]，如图 38 所示。

图 38. 简单 PWM 的项目原理图



13. 现在，您可以编译项目并编程 PSoC。可以在引脚 P0[0]和 P0[1]上观察时钟和终端计数。在 P0[2]上可观察 PWM 输出。
14. 保存项目，编译它并编程 PSoC。
15. 如果将某个示波器连接至输出引脚，您可以观察‘clock’、‘tc’和‘cmp’输出。图 39 显示的是‘clk’和‘cmp’信号。

图 39. 简单的 PWM 输出



先前已将起始值 15 加载到 A0 和 D0 内，因此计数周期的宽度为 16 个时钟周期。由于将 D1 设置为 7，所以每当 A0 小于 7 时，‘cmp’引脚将为高。您可以通过修改 D1 中的值，对 PWM 进行测试。

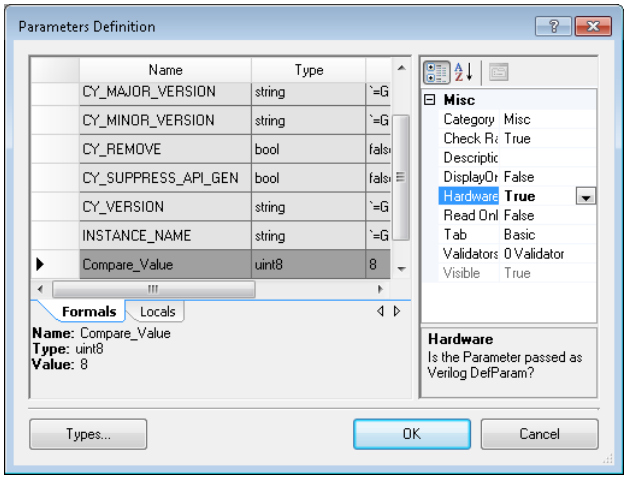
6.4 添加参数

需要修改某个组件的参数时，更改 Verilog 代码真不方便。另外，当需要使用计数周期和比较值均不相同的两个 PWM 时，您要如何处理？与几乎所有的赛普拉斯组件的工作方式相同，可以给您的组件添加用户可配置参数。

1. 打开该组件的符号编辑器页面（.cysym）并右击空白空间。
2. 从下拉菜单选择 **Symbol Parameters**。
3. 在现有参数下的空行内，输入新的参数。
   - 名称 = Compare_Value
   - 类型 = uint8
   - 值 = 8
4. 在窗口右侧的 **Misc** 设置字段中，将 **‘Hardware’** 标志设置为“*True*”，如图 40 所示。

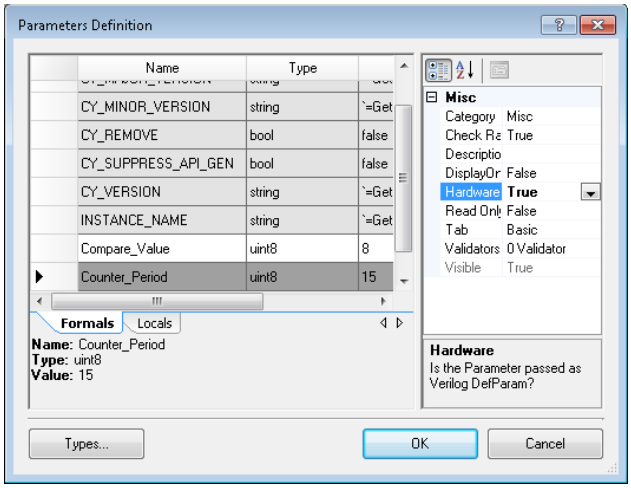
这样可以将参数开放给 Verilog 代码，因此 UDB 硬件可以使用它。

图 40. 添加一个组件参数



5. 在您刚创建的比较值定义的下一行内，输入另一个新的参数：
   - 名称 = Counter_Period
   - 类型 = uint8
   - 值 = 15
6. 在窗口右侧的 **Misc** 设置字段中，将 **Hardware** 标志设为‘True’，如图 41 所示。

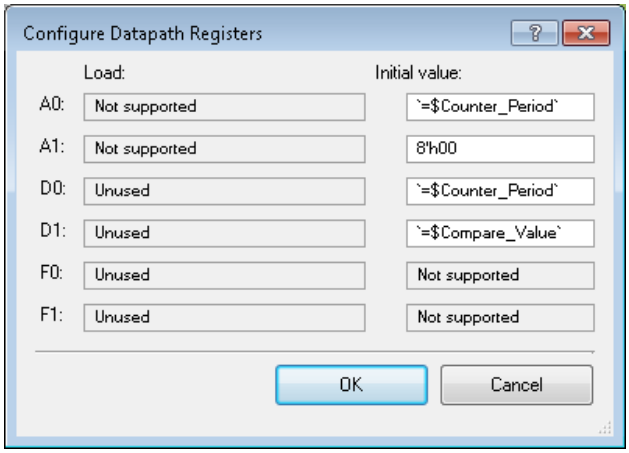
图 41. 添加另一个组件参数



7. 点击 **OK** 并依次选择 **File > Save All**，保存对该组件进行的所有更改。

   将组件符号参数链接到数据路径初始值。
8. 转到 UDB 编辑器（.cyudb），并打开 **Configure Datapath Register** 对话框。
9. 对于 **A0** 和 **D0**，请将 **Initial value:** 设置为`=$Counter_Period`。请注意，使用的是标注符号`，而不是单引号‘。
   对于 **D1**，请将 **Initial value:** 设置为`=$Compare_Value`；请参考图 42。

图 42. 根据参数设置初始值



这些代码将 A0、D0 和 D1 的初始寄存器值链接到组件参数。

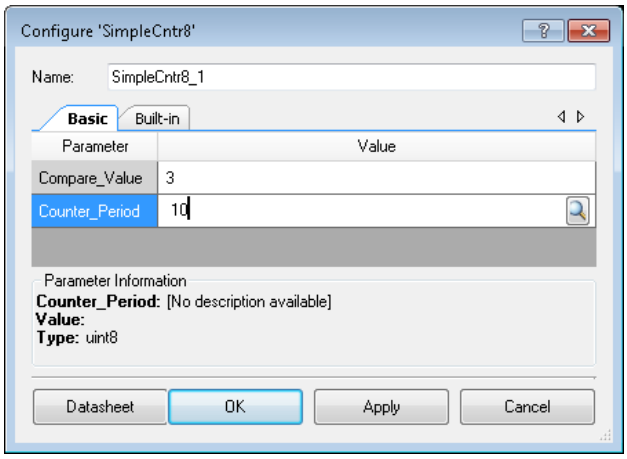
现在，您可以在构建时设置周期和比较值，而不需要在 UDB 编辑器内修改这些值。

10. 依次选择 **File > Save All**。

    返回项目的原理图，然后双击 SimpleCntr8_1 组件，以打开属性对话框。

11. 将比较值改为 3，将计数器周期改为 10，如图 43 所示。

图 43. 设置组件参数

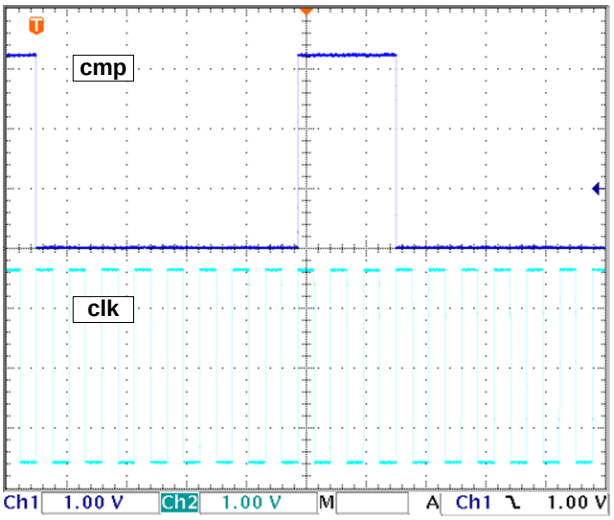


12. 单击 **OK**，应用更改内容。

13. 依次选择 **File > Save All**，编译项目，并编程 PSoC。

计数周期和比较输出已被更改，如图 44 所示。

图 44. 根据新参数设置 PWM 输出



计数周期的时长为 11 个时钟周期（周期时长等于‘10+1’，因为计数器先从 10 递减到 0，然后才进行重载），比较值为 3。得到的结果将为 8 个时钟周期的低电平输出和三个时钟周期的高电平输出。

只要所使用的数值是一个 uint8，您可以将参数修改成任何值。您甚至可以在项目中放置多个组件实例，并将它们的参数设置为不同的数值。

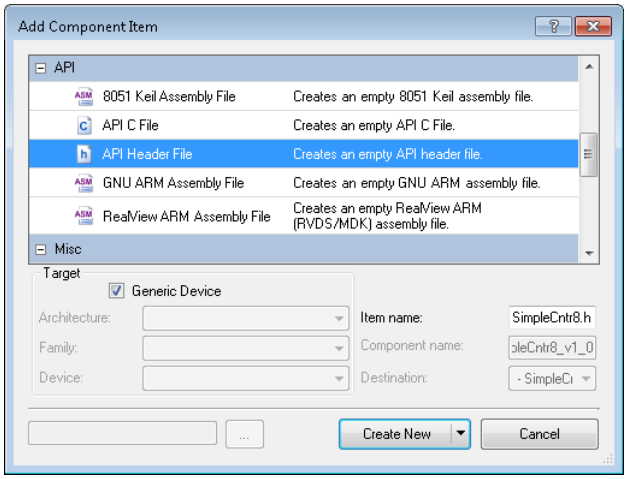
更多有关添加组件参数的信息（包括设置用户的输入值限制），请参考*组件作者指南*。

6.5 添加头文件

您可以在设计期间修改 PWM 的性能，另外，通过使用 C 代码修改 PWM 寄存器，您还可以在运行期间修改 PWM。例如，您分别使用 D0 和 D1 寄存器来保存周期值和比较值。为了便于访问这些寄存器，可以创建一个用于定义已用寄存器的头文件。

1. 在 **Components**（组件）选项卡中，右击 **SimpleCntr8_v1_0** 并选择 **Add Component Item**（添加组件项）。
2. 在 **Add Component Item** 窗口中，导航到 **API** 部分，然后点击 **API Header File**。
3. 将 **Item name** 改为 *SimpleCntr8.h*，如图 45 所示。

图 45. 添加头文件



4. 点击 **Create New**（创建新项），将头文件添加到您的组件内。

添加比较值（D1）和周期值（D0）的定义。

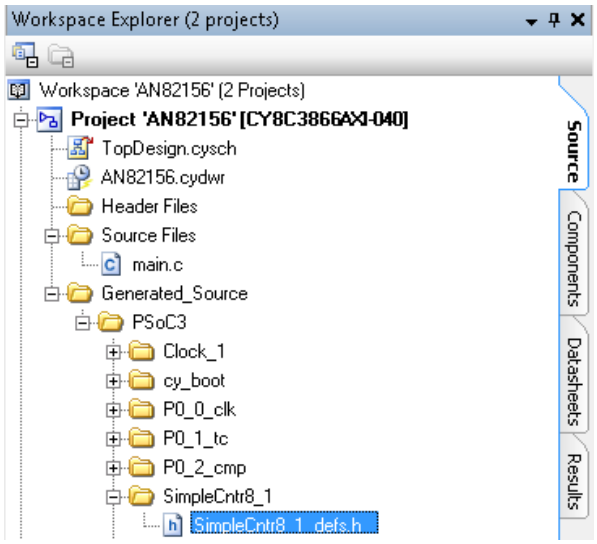
5. 在头文件的 //[]END OF FILE 前面，添加下面的定义：

```
#include "`$INSTANCE_NAME`_defs.h"
#define `$INSTANCE_NAME`_Period_Reg                    `$INSTANCE_NAME`_cntr8_D0_REG
#define `$INSTANCE_NAME`_Compare_Reg                   `$INSTANCE_NAME`_cntr8_D1_REG
```

这两个定义允许您直接写入固件中的 D0 和 D1 寄存器。*INSTANCE_NAME`_defs.h* 文件包含了数据路径中的各个寄存器的定义组。

可以在源选项卡的生成源文件夹中找到 *INSTANCE_NAME`_defs.h*，如图 46 所示。

图 46.*defs.h* 文件



如果想要在运行期间更新比较值，只要写入到您刚创建的定义即可。如果将您的组件命名为SimpleCntr8_1，那么您的C代码如下：

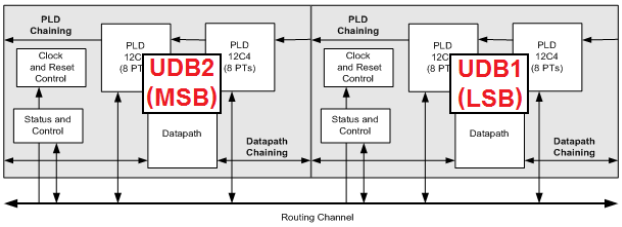
```
SimpleCntr8_1_Period_Reg = 0x08;
SimpleCntr8_1_Compare_Reg = 0x02;
```

此代码将周期值和比较值分别更新为0x08和0x02。更多有关如何使用这些定义的信息，请参考*组件作者指南*。请注意，直接写入到寄存器内的方法（如上面所述的C代码）只适用于8位寄存器。对于16位或16位以上的寄存器，您必须使用下一项目中所介绍的其他方法。

6.6 将 PWM 扩展到 16 位

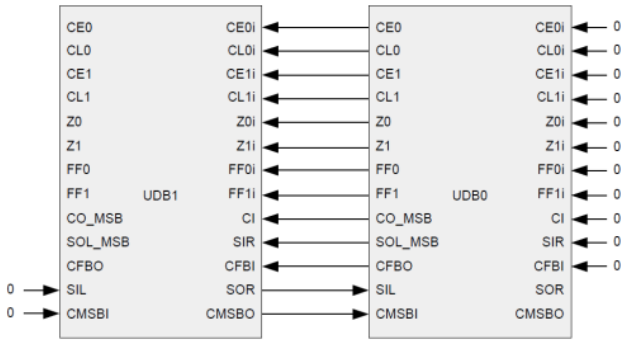
我们先了解一下数据路径链路的概念。数据路径具有连接到相邻数据路径的专用信号。通过这些信号，可以创建宽度为 1 到 32 位的功能。在该示例中，您创建的另一个 PWM 类似于第一个示例项目中的 PWM，但它的宽度为 16 位，如图 47 所示。

图 47. 带有链式 UDB 的 16 位功能



每个数据路径中的 ALU 用于将进位位、移位数据和条件信号链接到最接近的数据路径，如图 48 所示。按最低有效字节到最高有效字节的方向链接所有的条件和捕捉信号。左移是从最低有效字节链接到最高有效字节。右移是从最高有效字节链接到最低有效字节。

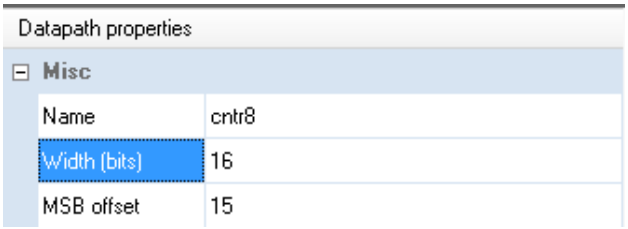
图 48. 数据路径的链接流程



通过使用 UDB 编辑器，能够轻松地将各种数据路径链接在一起。使用我们刚刚创建的 PWM，可以将其宽度设置为 16 位。

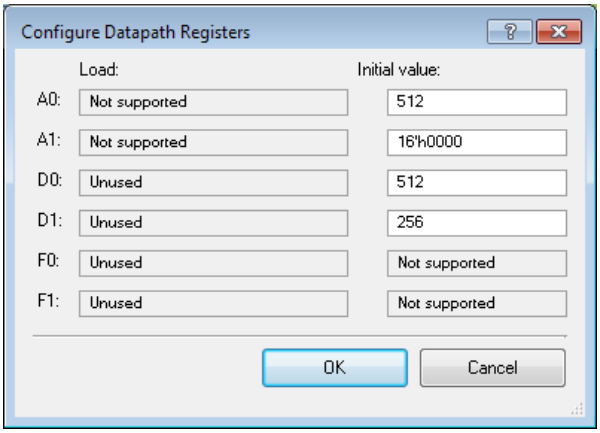
1. 转到 UDB 文档（.cyudb）。
2. 在 **Datapath properties** 面板中，将 **Width (bits)** 设置为 16，如图 49 所示。

图 49. 16 位配置



3. 将周期值（D0）和初始周期值（A0）设置为512，并将比较值（D1）设置为256，如图50所示。

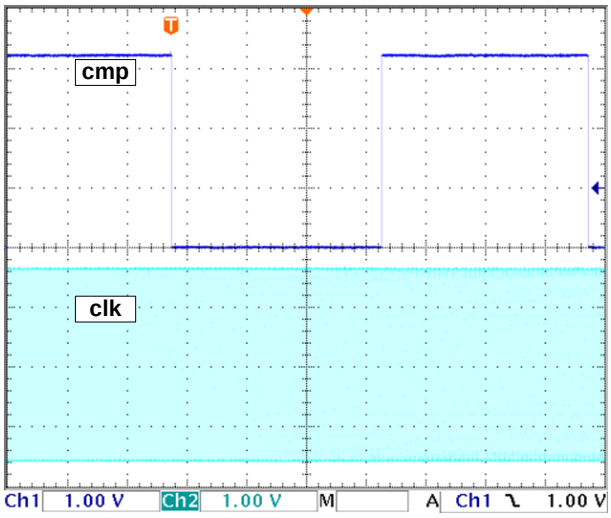
图 50. 16 位初始值



4. 在项目窗口中，依次选择**File > Save All**。
5. 返回顶层设计原理图（.cysch），并将时钟频率从10 kHz改为1 MHz（可以在示波器上观察到）。
6. 依次选择 **File > Save All**，编译项目，并编程 PSoC。

您可观察这些输出，发现该周期值和比较值比以前所创建的 8 位 PWM 的大得多，如图 51 所示。可将它们设置为任何一个 16 位数值。

图 51. 简单的 16 位 PWM 输出



您可通过使用链接方式使各函数的宽度达 32 位。将在该示例中所介绍的原理应用于较大的函数。

6.7 PSoC 3 的 16 位组件头文件

如前面所述，对 16 位寄存器进行写操作和对 8 位寄存器进行的不一样。由于可以忽略处理器和数据路径寄存器之间字节序的差异，因此您可直接写入到 8 位寄存器。当您写入到 16 位或宽度更大的寄存器中时，需要考虑到该字节序的差异。

PSoC 3 中 8051 的字节顺序不同于外设寄存器的顺序。为了便于对寄存器进行写操作，赛普拉斯提供了一些宏，如：CY_SET_REG16、CY_SET_REG24 和 CY_SET_REG32。这些宏包含两个参数：第一个是您所需要设置的寄存器地址，第二个是您想设置的值。这些宏处理所有字节顺序的交换。

因此，您必须掌握各数据路径寄存器的地址。通过使用自动生成头文件（_defs.h）中以_PTR 为结尾的定义，系统自动指出数据路径的地址。

要想更新某个值，请使用在头文件中所定义的 CY_SET_REG16 和指针。请再次查看随附的示例项目。

6.7.1 16 位参数

在前一个示例中，我们将符号参数设置为 *uint8* 类型，您也可以将参数添加到 16 位 PWM 内。您只需要将类型更改为 *uint16* 即可。

7 项目 2 — 递增/递减计数器

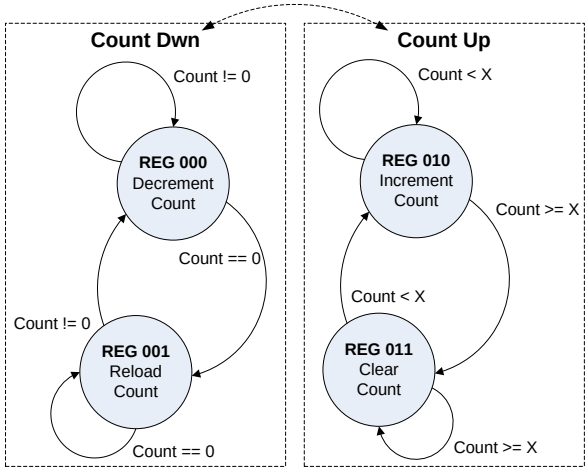
该示例介绍了将高级特性添加到数据路径组件的过程。更新了相同的 PWM 基本概念，以进行递增或递减计数。计数方向取决于您在运行时设置的参数。

该示例假设您已经熟悉了前几个示例项目中所介绍的概念。本应用笔记中提供了一个完整的递增/递减计数 PWM 项目。

7.1 更多的详细信息

这种简单的递减计数 PWM 项目使用两种状态。要想实现某个递增/递减计数器，需要使用四种状态，如图 52 所示。

图 52. 递增/递减计数器的状态图



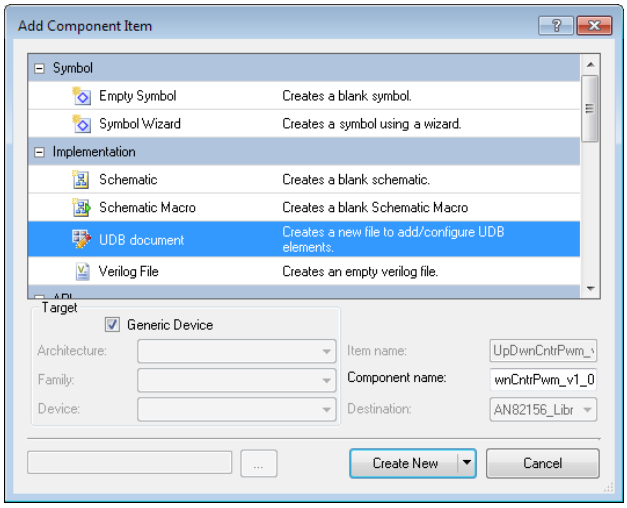
根据所设置的参数，数据路径将递减或递增 A0。您也可以设置周期值和比较值。

7.2 创建示例项目的步骤

为了避免混乱，请勿更改前几个示例项目中的组件，而应该创建一个新的。组件创建的步骤基本相同。

1. 启动 PSoC Creator，并打开简单的 8 位示例项目所使用的“AN82156”工作区。向工作区添加名称为“UpDwnCntrPwm”的新项目。
2. 在 Components （组件）选项卡中，右击 Project ‘UpDwnCntrPwm’ 并选择 Add Component Item。
3. 选择 UDB 文档并将 Component name: 更改为 *UpDwnCntrPwm_v1_0*。点击 Create New（创建新项）。

图 53. 创建新组件



4. 将数据路径元素拖放到设计图纸上。
5. 转到 Datapath properties （数据路径属性），并将 Name （名称）改为‘*UpDwn*’。

   通过配置数据路径实现图 52 中显示的功能。前两条指令与简单的 8 位 PWM 中相同，另外添加了两条指令。这两条指令能够实现递增功能，如表 3 所示。

表 3. 递增/递减计数器指令表

INSTR_ADDR	函数	寄存器写入	注释
000	ALU = A0 - 1	A0 = ALUout	递减计数值
001	No-op	A0 = D0	重载计数值
010	ALU = A0 + 1	A0 = ALUout	递增计数值
011	ALU = A0 ^ A0	A0 = ALUout	清除计数

XOR 配置可用于清除计数（A0）寄存器。计数（A0）寄存器递增计数到周期值时，它将对自己进行 XOR 操作，即清除所计数的值为零。

必须根据表 3 对每条指令进行配置。图 54 显示的是每条指令如何在数据路径元素中进行寻找。

图 54. UpDwnCntr 指令

Inst. Addr.	Instruction	Comment
3'b000	ALUout=(A0 - 1) A0=ALUout	Decrement Count
3'b001	ALUout=(A0) A0=D0	Reload Count
3'b010	ALUout=(A0 + 1) A0=ALUout	Increment Count
3'b011	ALUout=(A0 ^ A0) A0=ALUout	Clear Count

6. 在 Datapath properties （数据路径属性）面板中，请确保 Configurable comparator inputs（可配置比较器输入）下的 Config A 被设置为 *A0 compare to D1*。
7. 请确保 Compare options 下的每条指令都被设置为 *ConfigA: A0 compares to D1*。

8. 根据第一个示例配置 Zero_Detect 和 Compare 输出，另外添加了第三个输出（Period）。当计数器达到周期寄存器（D0）中的值时，第三个输出将变为高电平。这表示递增计数器已经达到最大周期值并且要进行复位；请参考图 55。

图 55. UpDwnCntr 输出

Out.	Selection	Name
0	A0 == 0	Zero_Detect
1	Config A: A0 < D1...	Compare
2	A0 == D0	Period

9. 将控制寄存器（CR）拖放到您的原理图上，如图 56 所示。

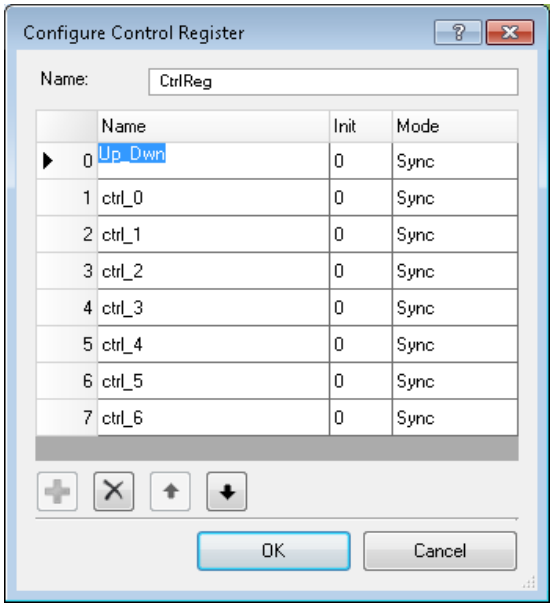
图 56. 控制寄存器

CtrlReg_1

Bit	Name	Init. Val.	Mode
0	ctrl_0	1'b0	Sync
1	ctrl_1	1'b0	Sync
2	ctrl_2	1'b0	Sync
3	ctrl_3	1'b0	Sync
4	ctrl_4	1'b0	Sync
5	ctrl_5	1'b0	Sync
6	ctrl_6	1'b0	Sync
7	ctrl_7	1'b0	Sync

10. 双击该控制寄存器。将 0 位的 Name （名称）设置为 Up_Down。将该控制寄存器重命名为 *CtrlReg*；请参考图 57。
11. CPU 可以对控制寄存器执行写操作。因此，CPU 通过对该控制寄存器位进行写操作来控制计数方向。1 表示递增计数，0 表示递减计数。

图 57. 控制寄存器配置



在第一个示例中，我们有两个数据路径指令；通过使用Zero_Detect信号很容易选择从CFGRAM执行的指令。现在，我们有四条指令：两条递减计数指令和两条递增计数指令。

仍可以将Zero_Detect用于递减计数。对于递增计数，则使用Period信号。另外，我们还有一个信号用于确定正在执行的是递增计数还是递减计数（ Up_Dwn ） 。我们有四条指令，表示它们需要两个地址位。但是，我们使用以下三个信号：Up_Dwn、Period以及Zero_Dectect，如表4所示。因此，必须通过逻辑将这三个信号缩减为两个地址线。

表 4. UpDwnPWM 指令解码

Up_Dwn	Period	Zero_Detect	INSTR_ADDR	功能
Down（递减）	N/A	0	000	递减
Down（递减）	N/A	1	001	重载
Up（递增）	0	N/A	010	递增
Up（递增）	1	N/A	011	清除

查看表4，我们会发现，始终要将一个地址位连接到Up_Dwn信号。其他地址位则被连接到Period信号或Zero_Detect信号。如果Up_Dwn信号被设置为Down，则地址位会连接到Zero_Detect；如果Up_Dwn信号被设置为Up，则地址位会连接到Period信号。

INSTR_ADDR[0] = if(up) Period else if(down) Zero_Detect

INSTR_ADDR[1] = Up_Dwn

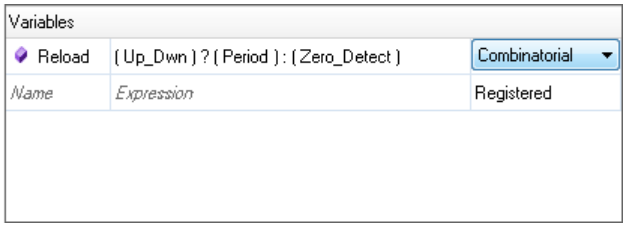
UDB编辑器允许将标准Verilog语法输入到编辑器中的各个字段。另外，我们还能创建逻辑控制的变量。在这个示例中，创建了一个新变量，用于决定INSTR_ADDR[0]使用Period还是Zero_Detect。

12. 在 **Properties** 窗口的 **Variables**（变量）下方，添加了一个名字为 *Reload*（重载）的新变量，并设置其表达式为：

    (Up_Dwn) ? (Period) : (Zero_Detect)

13. 请确保将其设置为 **Combinatorial**，请参见图 58。

图 58. 变量定义

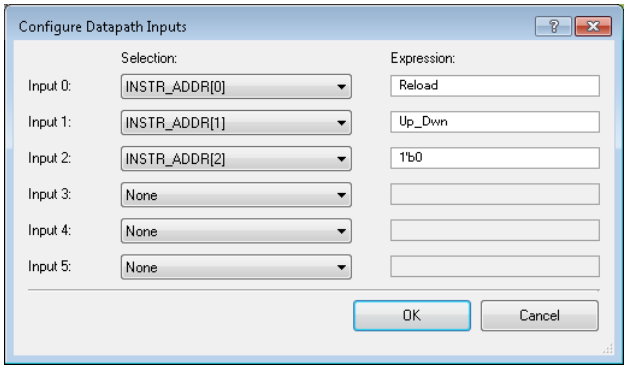


14. 注意：有时，“Combinatorial”设置被重置为“Registered”。请确保编译项目前，它始终被设置为“Combinatorial”。该问题将在 PSoC Creator 的下一个版本发布中予以修正。
15. 表达式 Reload = (Up_Dwn) ? (Period) : (Zero_Detect)是一个三元运算符。这是写一个简单 if-else 语句的更紧凑方式。
16. 具体形式如下：

    A = B ? C: D。如果 B 为 true（高），则 A = C；如果 B 为 false（低），则 A = D。

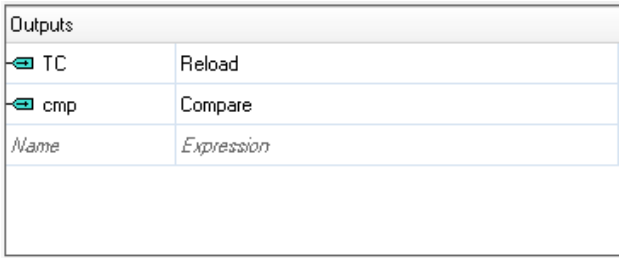
    在该示例中，如果 Up_Dwn 为高，则 Reload = Period；如果 Up_Dwn 为低，则 Reload = Zero_Detect。
17. Reload 用于选择数据路径指令。
18. 进入 Datapath Inputs（数据路径输入），并将 INSTR_ADDR[0]配置为 Reload、将 INSTR_ADDR[1]配置为 Up_Dwn，如图 59 中所示。

图 59. 指令寻址



19. 接下来，配置 PWM 输出。如前面所述，我们拥有一个终端计数（TC）和一个比较（cmp）输出。cmp 输出来自 Compare（比较）信号。TC 信号则来自新的 Reload 变量。
20. 配置输出，如图 60 所示。

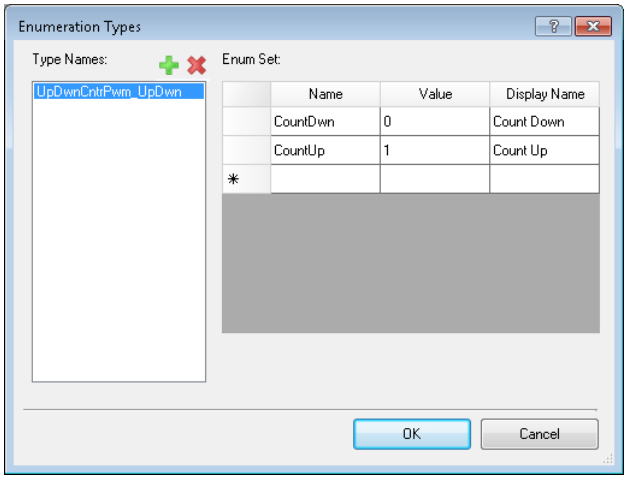
图 60. 输出



21. 右键单击.cyudb 文件中的空白处，并选择 Generate Symbol（生成符号）。

22. 右键单击符号原理图页面（.cysym），并添加符号属性：
    - Doc.APIPrefix = *UpDwnCntrPwm*
    - Doc.CatalogPlacement = *AN82156/Digital/UpDwnCntrPwm*
    - Doc.DefaultInstanceName = *UpDwnCntrPwm*
      添加几个用户可更改的参数，如：计数器周期、比较值和递增或递减计数模式。设置计数模式时，要定义新参数类型以及各组的值。
23. 右键单击符号编辑器页面，并打开符号参数对话框。
24. 单击 Types（类型）打开窗口，创建新的参数类型。
25. 单击绿色的‘+’按键来添加新类型。将它命名为 *UpDwnCntrPwm_UpDwn*。
26. 将各值输入到 Enum Set 字段，以确定‘CountDwn’和‘CountUp’定义，如图 61 所示。

图 61. 创建组件参数的新类型



27. 单击 **OK** 返回符号参数对话框。
    您可以将各参数分配给这种新类型，并将初始值设置为 0（CountDwn）或 1（CountUp）。
28. 根据前几个示例项目中的操作，为组件输入三个新参数；请参见表 5。

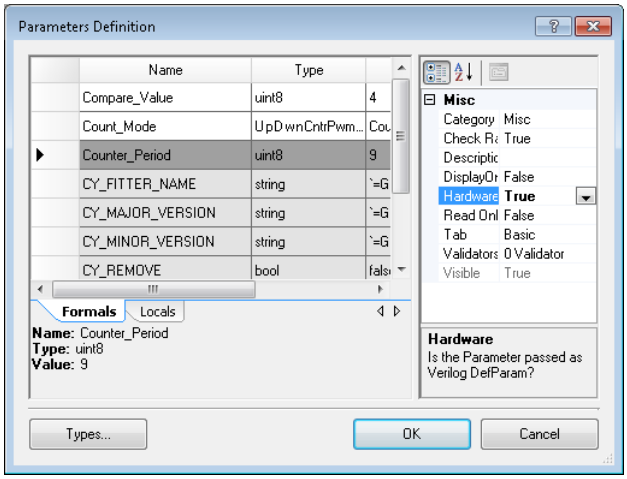
表 5. 递增/递减计数参数

名称	类型	值
Compare_Value	uint8	4
Count_Mode	UpDwnCntrPWM_UpDwn	Count Down
Counter_Period	uint8	9

Count_Mode 参数使用了新类型和值的定义。

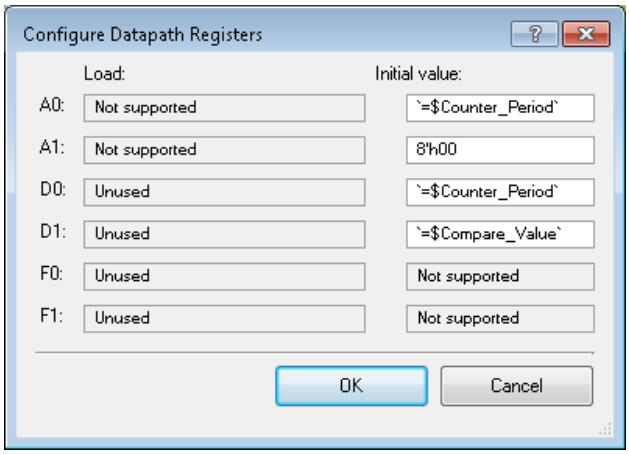
29. 将三个新参数的 **Hardware** 标志设置为‘True’，如图 62 所示。

图 62. 添加新的组件参数



30. 点击 OK，保存对组件所进行的更改。
31. 返回到.*cyudb* 文件，并根据第一个示例中的操作，修改寄存器的初始值；请参见图 63。

图 63. 根据参数设置初始值



32. 此时，我们不能通过硬件设置控制寄存器的初始值。因此，只通过固件设置它。
33. 创建一个头文件。在 Components（组件）选项卡中，右键点击 UpDwnCntrPwm_v1_0，然后选择 Add Component Item（添加组件项）。选择 API 头文件，并将其命名为 *UpDwnCntrPwm.h*。点击 Create New（创建新项）。
34. 在头文件中的/* [] END OF FILE */前面，添加以下代码行：

```
#define UP_DOWN  `$Count_Mode`
```

35. 该 define UP_DOWN 现在已链接到组件定制器中设置的 Count_Mode 参数。
36. 通过使用所包含的控制寄存器设置 *main.c* 文件中计数器的方向。请注意，所有标准控制寄存器 API 都可供使用。
37. 在 *main.c* 文件中，添加下面代码行：

```
UpDwnCntrPwm_1_CtrlReg_Write(UP_DOWN);
```

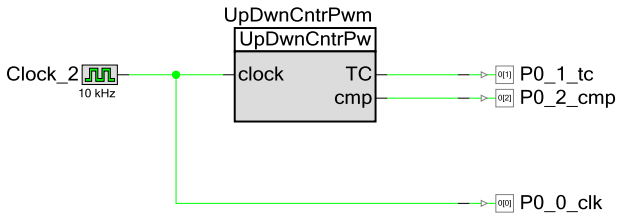
38. 如果您命名组件非为 UpDwnCntrPwm_1，则必须使用组件名称替换它。如果您命名控制寄存器非为 CtrlReg，则使用控制寄存器名称替换它。通过将控制寄存器放在.cyudb 文件中，您现在可以访问标准的控制寄存器 API，如第 28 步所示。

39. 您的组件已经就绪使用。

根据第一个示例项目添加所有相同组件。

40. 将某个 UpDwnCntrPwm 组件施放到项目原理图中。
41. 将一个时钟组件连接到该组件的‘clock’终端，并将它的频率设置为 10 kHz。
42. 将各个数字输出引脚组件分别连接至该组件的终端（即 P0_0_clk、P0_1_tc 和 P0_2_cmp），如图 64 所示。

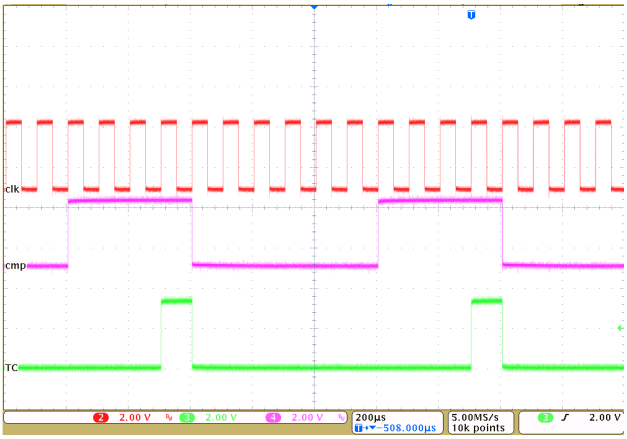
图 64. 递增/递减计数器的 PWM 项目原理图



43. 依次选择 File > Save All。
44. 编译项目并对 PSoC 进行编程。

将比较值参数、period、Count_Mode 分别设置为 4、9 和 Count_down。在一个示波器中观察 clk、cmp 和 TC 波形。您可以看见它正在递减计数，因为重载计数时，TC 变为高后，cmp 线变为低，如图 65 所示。

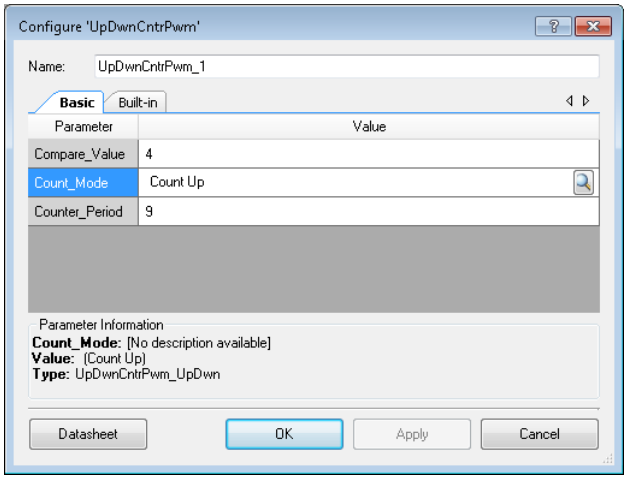
图 65. 递减计数 PWM 波形



您可以根据上述简单 PWM 示例中的方法更改周期和比较参数。也可以更改模式参数，使 PWM 进行递增计数而不是递减计数。

45. 返回项目的原理图，并双击 UpDwnCntrPwm 组件以打开属性对话框。
46. 将 Count_Mode 设置为‘Count Up’，如图 66 所示。

图 66. 将 PWM 设置为递增计数

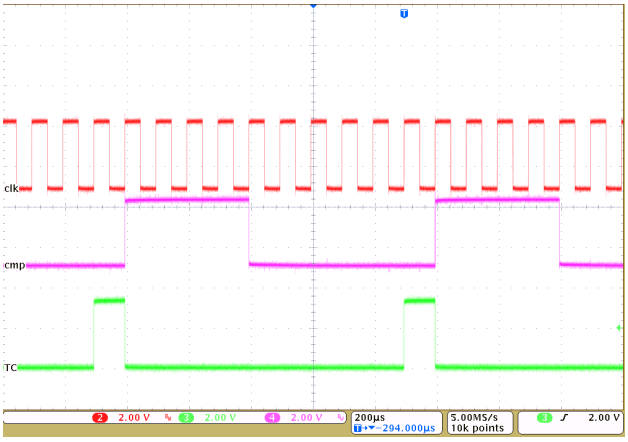


47. 单击 OK，应用更改内容。

48. 依次选择 File > Save All，编译项目，并编程 PSoC。

您可以观察，这是一个递增计数，因为将零值重载到计数器时，TC 下降沿后 cmp 值为高，如图 67 所示。

图 67. 递增计数 PWM 波形



请注意，TC 后，计数值低于比较值，并进行递增计数，因此输出为高。刚进入递减计数模式时，计数值大于比较值，并进行递减计数，因此输出为低。

8 项目 3 – 简单 UART

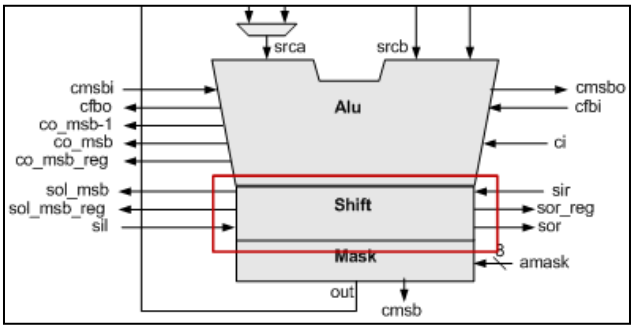
该示例项目展示了一个使用单个数据路径创建的简单 TX UART。我们将不会向您介绍创建组件的每一个步骤。但您可以在相关示例项目中查看组件和 UDB 编辑器文件。在各个已完成的示例的 *Simple_Tx* 项目中寻找“Simple_Tx”组件。如何使用该组件的一个示例被包含在相同的工作区内，并位于“*SimpleTx*”项目中。

8.1 TX UART 组件的详细信息

该组件中数据路径使用：移位并将 F0 中的值加载到 A0 内的数据路径操作。

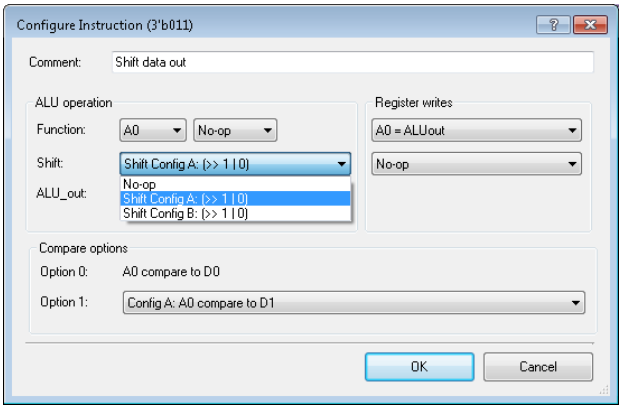
ALU 输出端上具有一个移位器，如图 68 所示。

图 68. 移位器模块



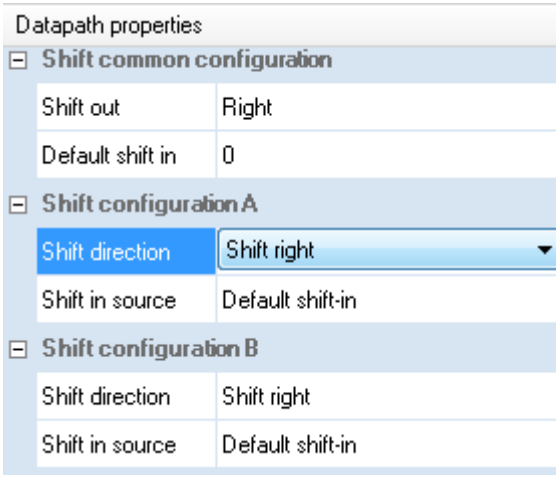
该移位器可以执行向左或向右移位。数据路径的每个指令均可独立设置移位器的操作。在每个指令的 **Configure Instruction** 对话框中设置该选项，如图 69 所示。

图 69. 移位操作设置



为了设置 Config A 和 Config B 的配置，请输入 **Datapath properties**（数据路径属性），并配置 **Shift configuration A** 和 **Shift configuration B**；请参见图 70。

图 70. 数据路径移位配置



移位的输入可以是 DSI 中的移入（SI）信号，或者它可以是输入的默认值（0、1）。

Datapath 输出复用器上仅有一个移出（SO）输出端。右移出（Shift Out Right）和左移出（Shift Out Left）共同使用该输出。您必须在 **Shift common configuration** 下（*Shift Out*）配置好该复用，如图 70 所示。

在该示例中，创建状态机用于控制数据路径的指令。在 UDB PLD 中实现该状态机。它确定了 UART 传输需要进行的下一部分：闲置、启动、数据或停止。

状态机仅有四种状态，用于控制数据路径的操作。因此，数据路径需要运行四个独特的操作：

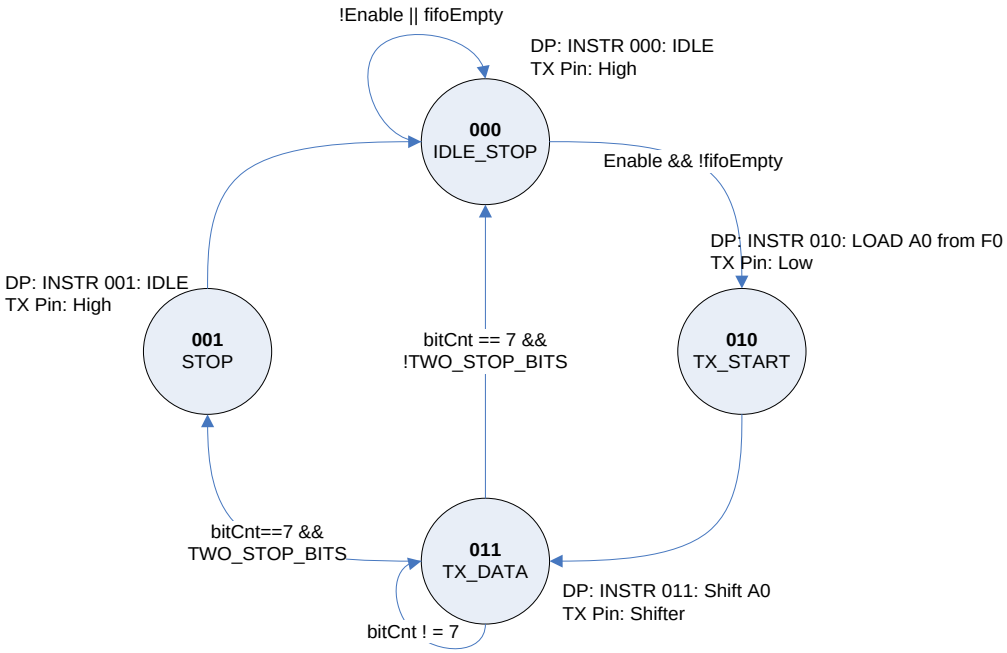
表 6. 数据路径指令示例 3

状态	INSTR_ADDR	功能	SHIFT	寄存器写入
IDLE_STOP	000	无操作	无	无
STOP	001	无操作	无	无
TX_START	010	无操作	无	A0 = F0
TX_DATA	011	无操作	SR	A0 = ALU

当代码移入到状态机内时，它改变了数据路径的指令。这种使用情况很普遍。

几乎所有复杂的数据路径组件都需要使用一个状态机来定序数据路径的操作。图 71 显示的是简单的 TX 状态机如何工作。

图 71. TX UART 状态机的流程图

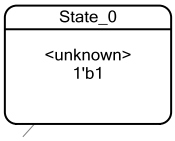


首先，通过监控 fifoEmpty 状态位，状态机将等待新数据被写入到 FIFO 内。一旦 FIFO 中存在数据，那么状态机将进入启动状态，并通过将 TX 线设置为低来发送一个 START（起始）位。在这种状态下，数据路径将 FIFO（F0）中的值加载到 A0 内。

在下一个状态中，数据路径将 A0 中的数据移出到 TX 引脚 LSB 优先（右移）。然后，状态机会发出一个或两个 STOP 位。

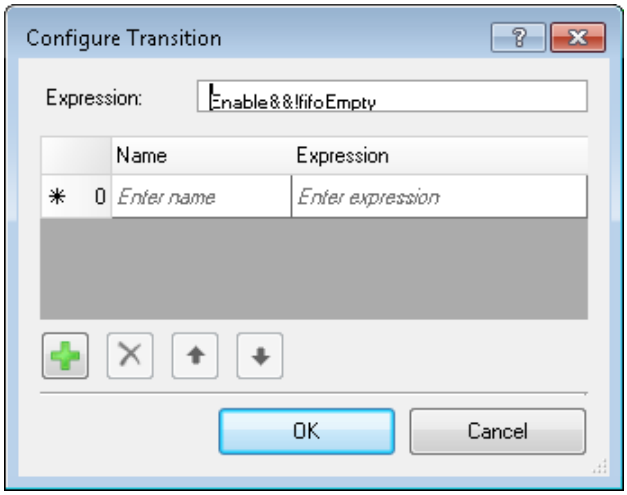
对于 UDB 编辑器，您可以复制图 71 中所显示的状态机。要实现该操作，需要将状态机元素拖放到设计图纸中；请参见图 72。

图 72. 状态机元素



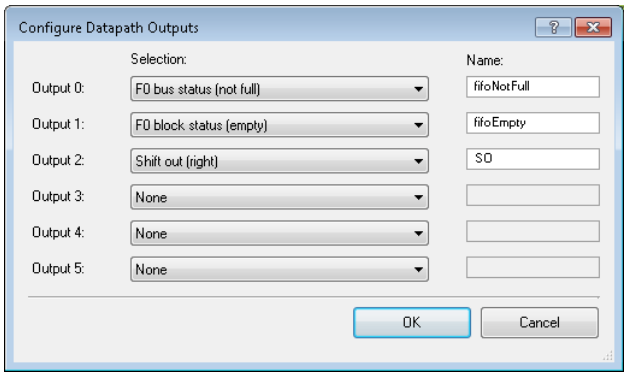
将这些元素中的四个拖放到设计图纸中，并连接它们，如图 71 所示。每次画一条线来连接各状态时，会出现一个对话框，让您写入一个表达式，用来决定转换发生的时间；请参见图 73。

图 73. 状态切换表达式



Enable 信号为高电平并且 fifoEmpty 不是高电平时，会发生图 73 中所示的切换。该切换控制着状态机如何从闲置状态进入启动状态。对于转换的发生，来自控制寄存器的 Enable 信号必须为高电平，并且数据路径 FIFO 不能为空。fifoEmpty 信号来自某个数据路径输出；请参见图 74。FIFO 中没有数据时，该信号为高电平；FIFO 中有数据时，该信号为低电平。

图 74. 数据路径输出



在每种状态中，您可以添加变量，并对这些变量进行逻辑和算法运算。例如，简单的 UART 移出八位数据。这意味着，数据通路必须保持数据（移位）状态的时长为 8 个时钟周期。

表 6 显示的是 INSTR_ADDR 为 011b 时发生的移位。图 75 显示了第三种状态（011b）的配置情况。请注意，在图 75 中的 **Variable assignment** 下面，变量 bitCnt 带有的 **Expression** bitCnt +1 表示：每次执行该状态，bitCnt = bitCnt+1。

状态机中的状态和数据路径都在输入时钟的上升沿上运行。

图 75. 状态配置

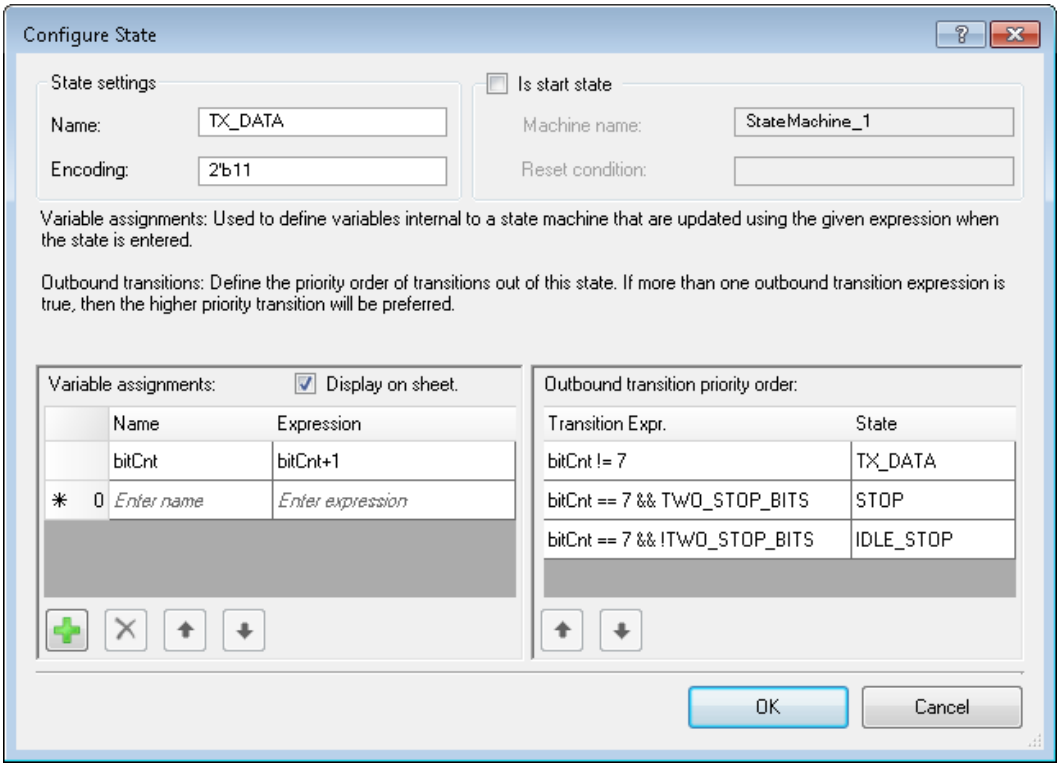
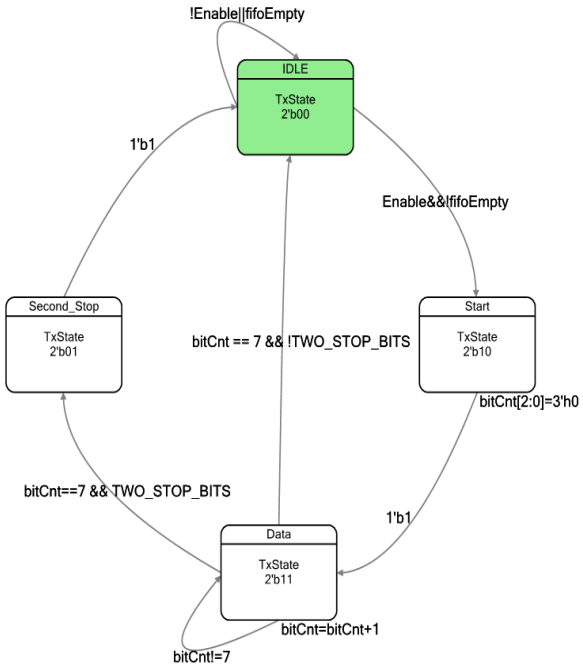


图 75 显示的是 **Outbound transition priority order**（出战转换优先顺序）对话框。该对话框用于配置状态切换。请注意，只要状态机 bitCnt 不等于 7，状态机将处于数据状态。如前面的介绍，需要移出八位，因此为什么保持这种状态的时长只有七个周期。实际上，保持这种状态的时长需要八个周期。bitCnt 被递增前会进行评价切换。第一次评价该表达式时，bitCnt 为零。

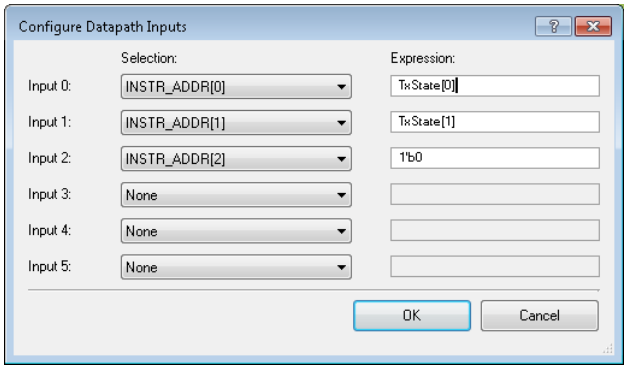
每种状态都必须采用相同的步骤，并且每种状态切换都要符合图 71 。下面的图 76 显示的是图 71 与 UDB 编辑器重复的情况。

图 76. UDB 编辑器状态机框图



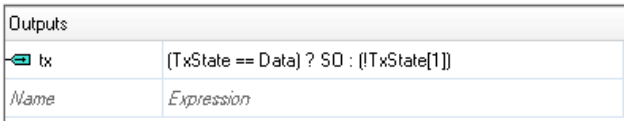
接下来，控制数据路径指令和状态机。如图 76 所示，状态机被称为 *TXState*。因为状态机中共有四种状态，所以 UDB 编辑器将为 TxState 创建一个 2 位的信号线。这个 2 位信号可以被路由到数据路径的 INSTR_ADDR 信号；请参见图 77。

图 77. 状态机控制的数据路径输入



接下来，您需要考虑如何控制 TX 输出线。如前面的介绍，在数据状态下数据被移出。在启动状态中，TX 线被置为低电平，那么在停止状态和闲置状态时会将该线设置为高电平。图 78 所示的输出表达式显示了这一点。

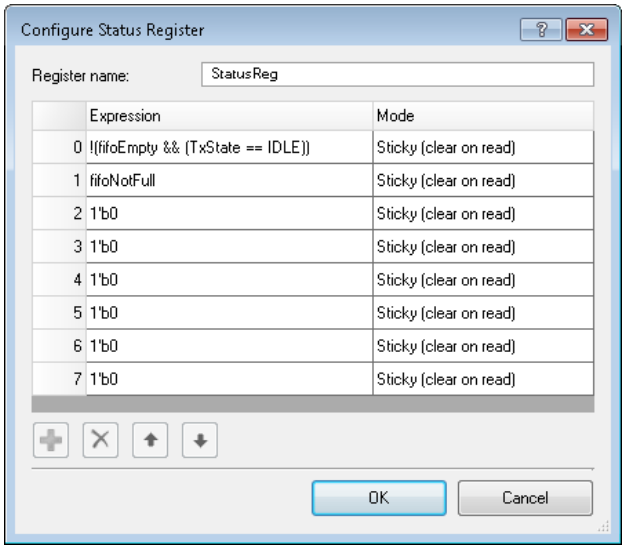
图 78. Tx 输出表达式



如果并非处于数据状态，它会反转输出 TX 线上‘TxState’的 msb 位。处于启动状态下，msb 为 1，因此其输出将为‘0’。而在停止和闲置状态下，msb 为 0，所以其输出为‘1’。在数据状态下，它将输出 SO（移出）信号。

该设计的唯一变化是添加了一个状态寄存器。该状态寄存器主要是为了向 CPU 报告状态机和数据路径的状态。

图 79. 状态寄存器配置



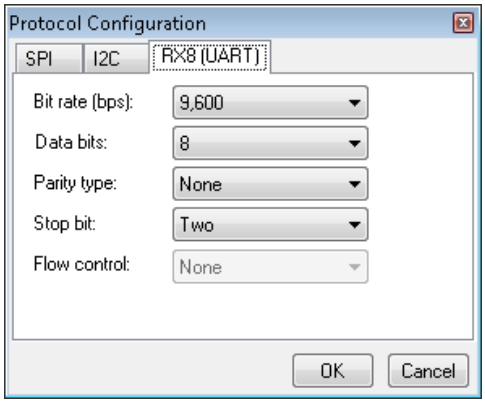
该状态寄存器的第一位将向 CPU 报告 UART 处于繁忙状态的时间。可以通过这种状态进行等待，直到 UART 完成发送数据为止。第二位将报告 FIFO 是否未满。CPU 可以使用该信号，以便将数据加载到 FIFO 中，直到 FIFO 满为止。

已经将一个头文件添加到了该组件内。头文件用于定义使能组件的某些常量；在一个停止位模式或两个停止位模式中设置该组件，并其状态寄存器会定义常量。在头文件中要注意：定义要与状态寄存器和控制寄存器中所选的位匹配。

该项目的主代码使能了带两个停止位的组件，然后连续传输 0 到 10 数值。波特率为 9600 波特时可以实现该操作。将接收器配置为 9600 波特和两个停止位。

与 PSoC Creator 一同安装的是一个被称为桥接控制屏幕（BCP）的程序。您可以使用 BCP 来接收 RX 字符。在 BCP 中，连接到 COM 端口（TX 输出已经连接到该端口）。在 **Tools**（工具） **> Protocol Configuration**（协议配置）中，配置 RX8（UART），如图 80 所示。

图 80. BCP RX8 配置



在 BCP 的编辑器中，添加了以下文本：rx8 x x x x x x x x x x。从选定的 COM 端口中读取 11 个字节。然后，您可以通过点击 Repeat（重复）按键来连续接收数据。

9 从 UDB 编辑器移植到数据路径配置工具

可能存在一个问题：您不能使用 UDB 编辑器来实现所需要的功能。这时，您只能采用 Verilog 文件和数据路径配置工具。这一步比较简单。按照附录 A 中的指导创建 Verilog 文件，然后复制由 UDB 编辑器生成的 Verilog 代码并粘贴到新的 Verilog 文件内。现在，您可以通过数据路径配置工具将 Verilog 文件修改为重要的内容。创建组件和使用数据路径配置工具的示例，请参考附录 A。

10 汇总

在 PSoC 可编程逻辑中构建组件时，UDB 数据路径增加了设计的灵活性。理解和有效地使用 UDB 数据路径能够使 PSoC 3、PSoC 4 和 PSoC 5LP 实现超过传统微处理器所提供的功能。

本应用笔记中所介绍的各示例项目仅是创建自定义解决方案的开始。欲了解更多有关将各特性和复杂性添加到自己的组件的信息，请参阅 PSoC 架构的《技术参考手册》和《组件创建指南》中的内容。

11 相关资源

11.1. 应用笔记

- AN54181 — PSoC 3 入门
- AN79953 — PSoC 4 入门
- AN77759 — PSoC 5 入门
- AN81623 — PSoC 3 和 PSoC 5 的数字设计最佳实践
- AN82250 — 实现 PSoC 3、PSoC 4 和 PSoC 5 的可编程逻辑设计

11.2. 知识库文章

- KBA86838 — 数据路径配置工具说明书
- KBA86336 — 面向 PSoC 器件的 Verilog 说明书
- KBA86338 — 创建基于 Verilog 的组件
- KBA81772 — 向项目中添加组件基元/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

11.3. 技术参考手册《TRM》

- PSoC 3 架构 TRM
- PSoC 5LP 架构 TRM
- PSoC 4 架构 TRM

11.4. 视频

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

11.4.1. 基本步骤

- 创建新组件符号
- 创建 Verilog Implementation（Verilog 实现）

11.4.2. 组件创建

- PSoC Creator 113：基于 PLD 的 Verilog 组件
- PSoC Creator 210：数据路径组件的介绍
- PSoC Creator 211：数据路径计算
- PSoC Creator 212：数据路径 FIFO
- PSoC Creator 213：多字节数据路径组件
- PSoC Creator 214：数据路径 API 生成
- PSoC Creator 教程：使用 UDB 编辑器 — 第一部分
- PSoC Creator 教程：使用 UDB 编辑器 — 第二部分
- PSoC Creator 教程：使用 UDB 编辑器 — 第三部分
- PSoC Creator 教程：使用 UDB 编辑器 — 第四部分
- PSoC Creator 教程：使用 UDB 编辑器 — 第五部分
- PSoC Creator 教程：使用 UDB 编辑器 — 第六部分

12 关于作者

姓名： Todd Dust

职务： 应用工程师

背景： Todd 毕业于西雅图太平洋大学，并获得电气工程学学士学位。毕业后，他一直在赛普拉斯从事应用工程师的工作。

姓名： Greg Reynolds

背景： Greg Reynolds 十多年来在赛普拉斯中一直担任多个角色。

13 附录 A — 使用数据路径配置工具示例

了解 PSoC 特性的最好方法之一是在某个设计中使用它。本应用笔记中包含了五个示例项目，用于介绍如何使用数据路径配置工具创建六个基于数据路径的简单组件：

- 8 位递减计数器
- 8 位 PWM
- 16 位 PWM
- 8 位增/减计数 PWM
- 只带 TX 的简单 UART
- 并行输入和并行输出

您可以在所有的 PSoC 3、PSoC 4 或 PSoC 5LP 器件上使用该示例项目。您可以在赛普拉斯网站中本应用笔记的登陆页面上查找完整的示例项目。

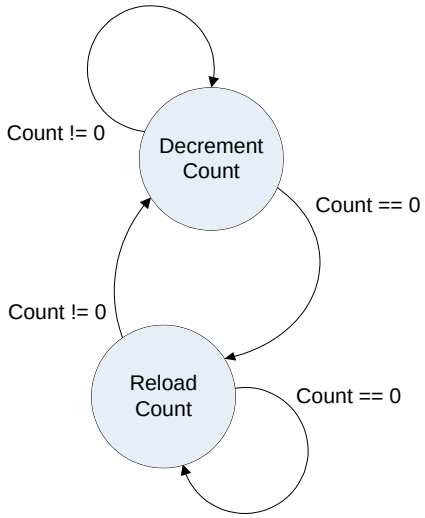
13.1 项目 1 — 8 位递减计数器

该项目的目的是向您介绍创建一个基于数据路径的简单组件的具体步骤。首先，需要创建一个 8 位的递减计数器组件，然后将它改为 8 位 PWM。

13.1.1 8 位计数器组件的详情

我们可以使用具有两个状态的状态机来表示一个简单的递减计数器，如图 81 所示。当数据路径时钟的上升沿到来时，将发生状态转换。

图 81. 简单计数器的状态图



启动后，计数器的初始值将递减。递减到零时，将触发某个事件，并重载该周期。在数据路径中可以轻松实现这种计数器。

您只需要使用两个 CFGRAM 条目（配置/指令）来实现该计数器。第一个条目用于递减存储在某个工作寄存器中的计数值。第二个条目重载初始值。剩余的六个 CFGRAM 条目不需要使用。

该示例中使用的两个数据路径寄存器为：

- A0 — 保存计数值，并且由 ALU 递减。
- D0 — 保存计数值为零时被重载到 A0 内的值。

您需要某个方法来决定数据路径的配置（指令）情况：加载或递减。图 81 显示了由计数值控制的状态转换，即判断计数值是否为零。

数据路径有一个零检测器模块（ZDET），用于监控 A0 和 A1 中的数据值。该模块有两个输出 z0 （A0 == 0）和 z1（A1 == 0），分别表示 A0 和 A1 的状态。数值为零时，相应的输出为高；该值为非零时，相应的输出为低。

在该示例中，使用 z0 输出选择配置。CFGRAM 的寻址需要 3 位。可以将位 0 使用于 z0，并将其它位保持为低。表 7 总结了配置转换：

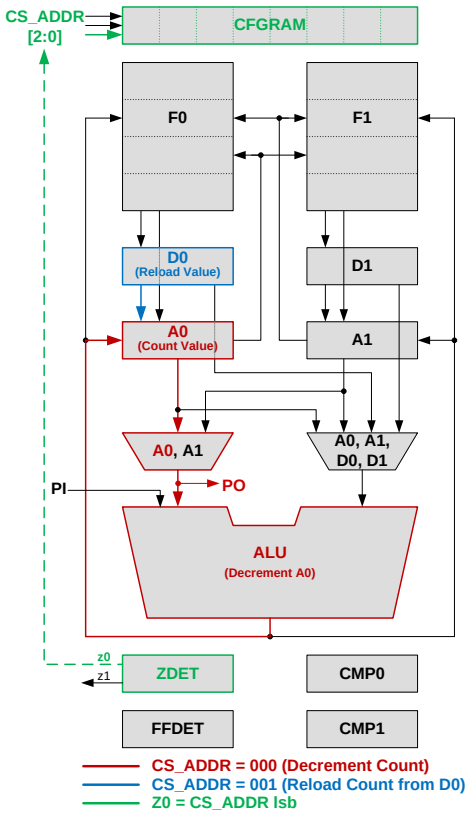
表 7. 切换表示例 1

CFGRAM 配置	CFGRAM 地址位			操作
	2	1	0	
0	0	0	z0 = 0	递减计数值
1	0	0	z0 = 1	重载计数值
2-7	X	X	X	未使用

A0 中的计数值为 0 时，z0 变为‘1’。这样，下一个数据路径配置便为“重载计数值”。当从 D0 将计数值重载到 A0 时，z0 变为‘0’，下一个配置将为“递减计数值”。

想要了解状态转换的运行方式，请查看加亮显示的数据路径框图，如图 82 所示。

图 82. 加亮显示的简单计数器框图

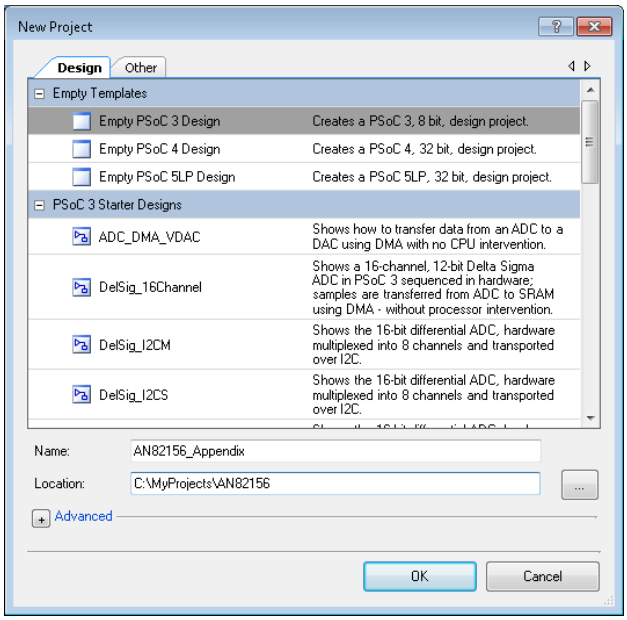


13.1.2 8 位计数器组件的创建步骤

可以使用现有的 PSoC Creator 项目，并向该项目添加新的组件库。然而，在这个示例中，请创建一个新的项目。

1. 启动 PSoC Creator 并创建名称为“*AN82156_Appendix*”的项目，如图 83 所示。“*AN82156_Appendix*”工作区也被默认创建。

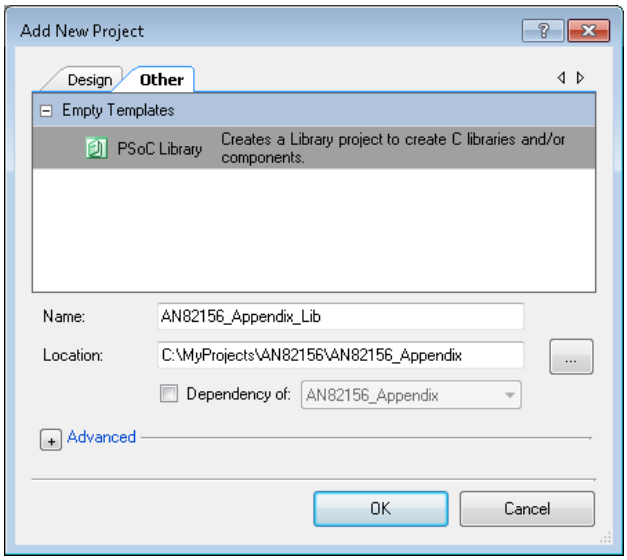
图 83. 添加新项目



各个组件被存储在 PSoC Creator 库项目中。为了区分您的库和组件与赛普拉斯的标准组件库，请为它们选择唯一的名称。

2. 在工作区浏览器的 **Source** 选项卡中，右键点击 **Workspace ‘AN82156_Appendix’**，并从显示的下拉菜单中依次选择 **Add > New Project…**。
3. 选择 **Other** 选项卡，显示 **PSoC 库**模板，如图 84 所示。将它命名为“*AN82156_Appendix_Lib*”，并使用它的默认地址。这样，该库与 *AN82156_Appendix* 工作区将位于相同的位置。

图 84. 添加新库

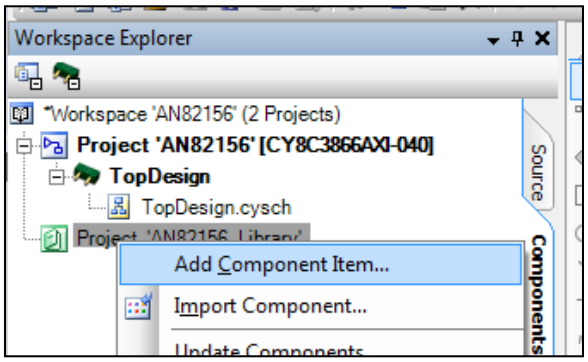


现在，新库将显示在工作区浏览器内。接下来，请将该库链接到示例项目，即可使用该库中的组件。

然后，您需要向新库添加一个新组件：

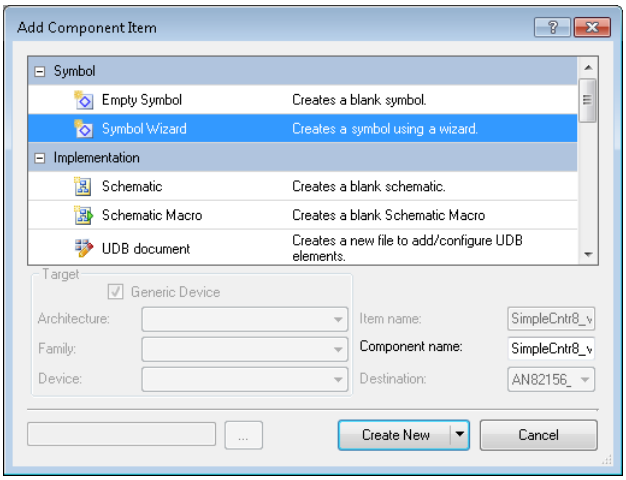
4. 切换到 Workspace Explorer 中的 **Components** 选项卡，然后右击 **AN82156_Appendix_Lib**。从下拉菜单中，选择 **Add Component Item...**项，如图 85 所示。

图 85. 添加组件项



5. 选择 **Symbol Wizard** 组件模板，并将该组件命名为 *“SimpleCntr8_v1_0”*，如图 86 所示。

图 86. 使用符号向导



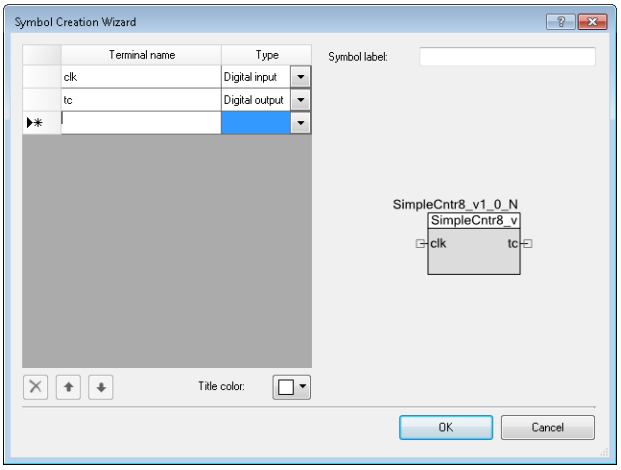
组件名称应该包含版本编号。将“_vX_Y”标签附加到组件名称，其中，‘X’表示主要版本、‘Y’指的是次要版本。PSoC Creator 具有版本控制功能，通过它您可以跟踪并使用组件的多个版本。

6. 单击 **Create New** 按键，以启动组件符号向导。

   该向导要求定义输入和输出，然后使用这些信息来创建某个组件符号。

7. 在 **Terminal Name** 字段中添加两个终端，即“*clk*”输入和“*tc*”输出，如图 87 所示。

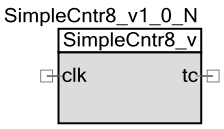
图 87. 添加输入和输出



“clk”输入为数据路径的时钟。“tc”输出将告诉您计数值为零的时间。

8. 单击**“OK”**后，将在符号编辑器页面中生成组件符号，如图 88 所示。

图 88. 所生成的组件符号

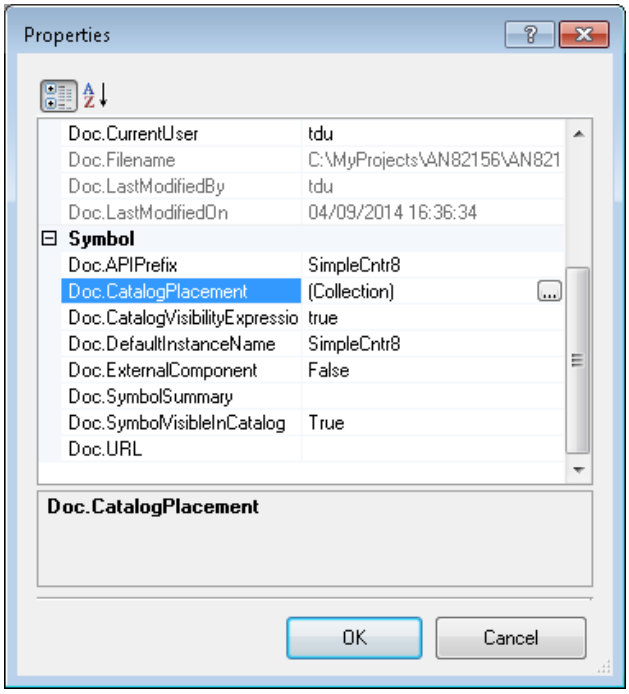


此时，‘clk’和‘tc’终端只是原理图符号的一部分；它们不参与操作。需要使用 Verilog 来定义它们的功能。

9. 在符号编辑器中，右击空白空间（而不是符号本身），然后从下拉菜单选择 **Properties**。
10. 在属性字段下的 **Symbol** 部分中输入所需的数值，如图 89 所示：

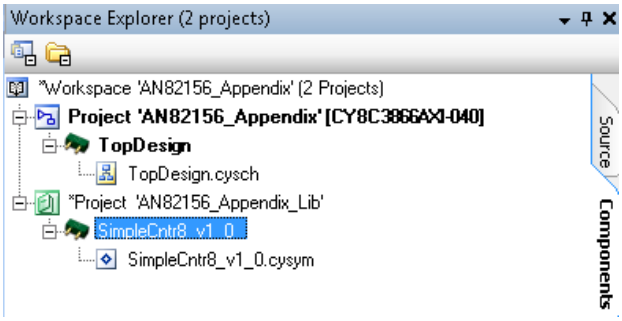
- **Doc.APIPrefix** = *SimpleCntr8*.
  该值作为该组件中所有 API 文件名的前缀。在该示例中不会生成任何 API，但是每当创建某个组件时，请在此处输入一个数值。
- **Doc.CatalogPlacement = AN82156_Appendix/Digital/Cntr8.**
  通过点击‘**...**’按键打开 Catalog Placement 对话框，然后才能输入该值。PSoC Creator 使用该值定义组件目录的层次结构。第一项是指选项卡，在该选项卡下将显示目录中的组件。每一个后续的‘/’均表示一个子目录。组件目录的层次结构必须至少包含一个子目录。这里所示的数值表示该组件是‘Cntr8’，它显示在 AN82156_Appendix 选项卡下的‘Digital’子目录内。
- **Doc.DefaultInstanceName** = *SimpleCntr8*.
  这是组件被放置在原理图中时显示的默认名称。您将组件放置在项目原理图中后可以修改该名称。

图 89. 添加符号属性



11. 执行“Save All”（Ctrl+Shift+S）操作，确保保存了对项目进行的所有更改。新的符号显示在 Workspace Explorer 中 **Components** 选项卡的 **AN82156_Appendix_Lib** 下，如图 90 所示。

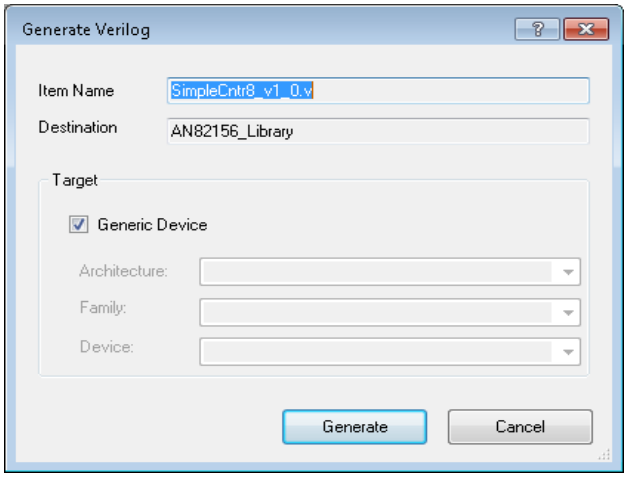
图 90. 库中的新组件



接下来，您需要将原理图符号链接到一个数据路径的实现。

12. 在符号编辑器页中，右键单击空白空间，然后选择下拉菜单中的 **Generate Verilog** 项。
13. 保持 **Generate Verilog** 对话框中的所有默认设置，然后点击 **Generate**；请参见图 91。

图 91. Generate Verilog 对话框



14. 执行“Save All”（全部保存）操作，确保保存了对项目进行的所有更改。这时，将生成一个新的 Verilog 文件，并且该文件被添加到组件中。

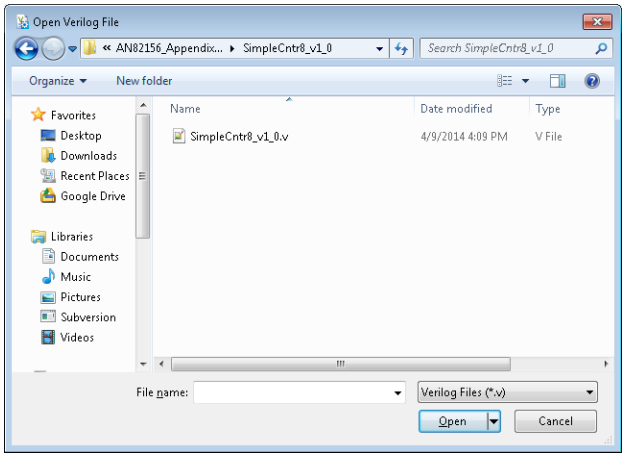
    Verilog 文件包含了 Datapath 的实现以及用于控制该 Datapath 的所有 Verilog 代码。附录 C 介绍了新组件 Verilog 文件的示例。想要添加并编辑数据路径的实例，请使用*数据路径配置工具*。

15. 启动数据路径配置工具。

16. 导航到 **Tools** > **Datapath Config Tool**...，实现上述操作。也可以通过 Start 菜单启动数据路径配置工具（依次选择 **Start** > **All Programs** > **Cypress** > **PSoC Creator 3.x** > **Component Development Kit** > **Datapath Configuration Tool**）。

17. 在数据路径配置工具中，依次选择 **File** > **Open**，然后浏览到由 PSoC Creator 生成的 *SimpleCntr8_v1_0.v* 文件所在的位置。如果您按照该示例中的各步骤进行操作，该文件位于 *AN82156_Appendix* 工作区文件夹中的 *AN82156_Appendix_Lib* 项目文件夹内，如图 92 所示。

图 92. 组件 Verilog 文件示例



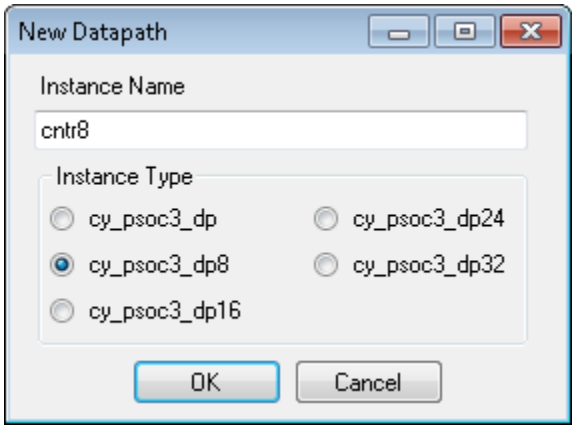
18. 选择该文件，然后通过单击 **Open** 加载 Verilog 文件到数据路径工具中。

    第一步骤是创建新的数据路径配置。

19. 从菜单中依次选择 **Edit** > **New Datapath**，打开 New Datapath 对话框窗口。

20. 在 **Instance Name** 文本框中，输入一个实例名称（使用 *“cntr8”*），然后选择“Instance Type”（实例类型）为 **cy_psoc3_dp8**，如图 93 所示。点击 **OK**。

图 93. 新的 8 位数据路径实例



这是一个 8 位计数器，所以数据路径的定义不能超过 8 位（cy_psoc3_dp8）。其它示例项目演示的是大小超过 8 位的组件。

21. 通过依次选择 **File > Save**，可以将配置保存到 Verilog 文件。

    第一次保存配置时，数据路径配置工具将实例化 Verilog 文件中的数据路径结构，如附录 C 所示。通过修改该配置，可以执行计数器功能。

22. 选择 Reg0 和 Reg1 字段中的值，以配置数据路径，如图 94 所示。其它字段可以保持其默认设置。
23. Reg0 的设置与配置 0 或指令 0 的相同，Reg1 的设置与配置 1 或指令 1 的相同。

图 94. SimpleCntr8 组件配置

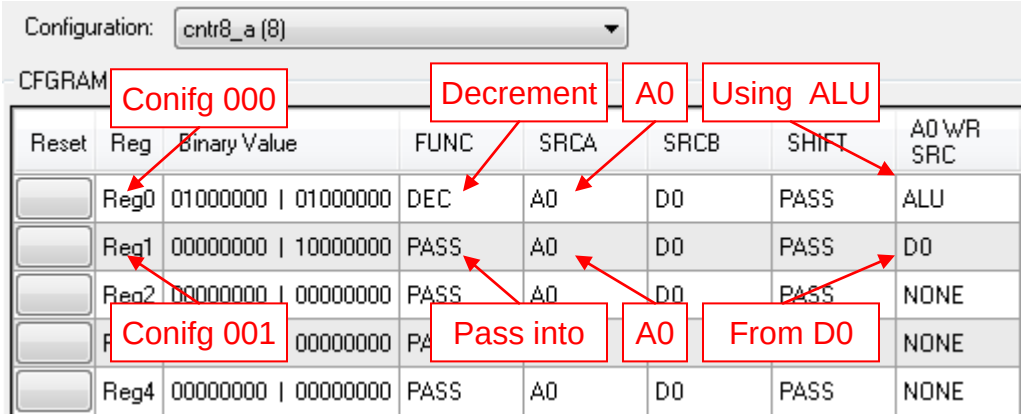


表 8. 一个数据路径配置示例

FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
DEC	A0	D0	PASS	ALU
PASS	A0	D0	PASS	D0

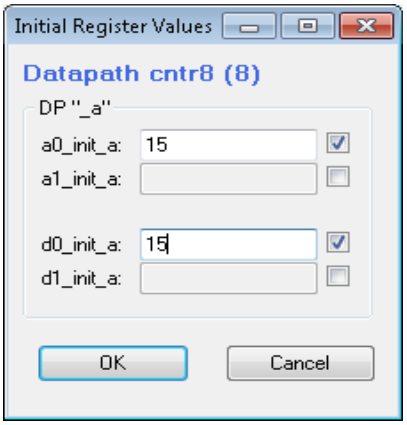
24. Reg0 和 Reg1 行表示存储在 CFGRAM 中 8 个数据路径配置中的前两个配置。配置的第一个数据用于递减计数器，第二个用于重载计数器。

    计数器的初始值为 A0 和 D0 寄存器中的值。由于通过 CPU 可以访问 Ax 和 Dx 寄存器，因此可以使用 *main.c* 文件中的代码加载这些值。通过在 Verilog 文件中定义默认的初始值可以将它加载到这些寄存器内。数据路径工具可以实现上述操作。

25. 依次选择 **View > Initial Register Values**，打开寄存器值对话框。

26. 勾选 **a0_init_a** 和 **d0_init_a** 旁边的选框。通过在该两个选框中输入 *15* 定义起始值和重载值，如图 95 所示。

图 95. SimpleCntr8 的初始寄存器值



27. 操作完成后，点击 **OK**。

    在这种情况下，D0 和 A0 中的值是相同的，所以您可以将同样的计数值重新加载到 A0 内。如果您想要第一次计数的周期与剩下的计数周期不同，请选择其他数值。在此情况下，计数周期将为 16，因为计数序列是从 15 到 0 的。

28. 从菜单中依次选择 **File** > **Save**，保存对 Verilog 文件所进行的更改。关闭数据路径配置工具，并查看 PSoC Creator 中的 *Counter_8bit_v1_0.v* 文件。

    在您保存了各项配置后，数据路径配置工具将对 Verilog 文件进行修改。当您返回时，PSoC Creator 可能会要求您重载 Verilog 文件。

    这时，您要修改 Verilog 文件，这样可以将原理图符号链接到数据路径逻辑。所提到的代码节以 Verilog 文件中的行 77 开始。

29. 将`.clk(1'b0)`改为`.clk(clk)`。可以将数据路径时钟链接到符号的‘clk’终端。 使用该终端设置数据路径的工作速度。

30. 将`.z0()`,改为`.z0(tc)`。这样可以使零检测模块（ZDET）的 z0 输出链接到符号的‘tc’终端。链接后，‘tc’终端反映 z0 输出的值。

31. 将`.cs_addr(3'b0)`文本改为 `.cs_addr({2'b00,tc})`。这样可以将 CFGRAM 地址的两个最高有效位设置为‘0’，并将位 0 设置为 `tc` 上的值。

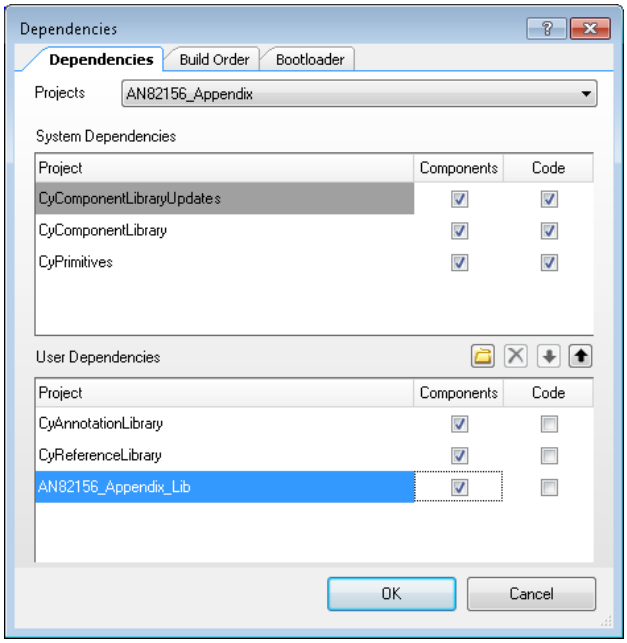
32. 请留意，`tc` 始终反映 `z0` 的值。因此，`z0` 会决定所使用的配置；请参见表 7。

33. 保存对 Verilog 文件进行的所有更改。附录 C 描述了如果所有设置无误的情况下完成的 Verilog 代码。

34. 现在，您可以使用该组件；在项目的“Component Catalog”中查看该组件前，需要设置 *AN82156_Library* 为项目依赖关系。

35. 在 **Source** 选项卡中，右键点击 **Project ‘AN82156_Appendix’**，并选择 **Dependencies…**。添加新的 *AN82156_Appendix_Lib*，并将它设置为 AN82156 项目的用户依赖关系 — 查看组件选框，如图 96 所示。

图 96. 添加 MyLibrary 并将它设置为项目依赖关系



现在，在 *AN82156_Appendix* 选项卡的组件目录下，可查看新的 *AN82156_Appendix_Lib* 组件，如图 97 所示。

图 97. 目录中的新组件



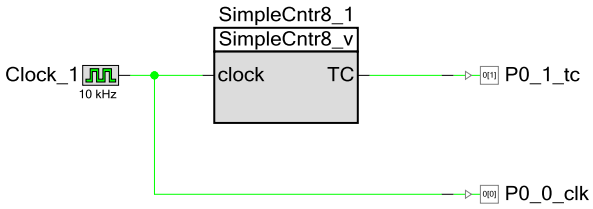
36. 在 Component Catalog 下能看到组件后，您可以将该组件放置在原理图中。该组件的使用情况与其它组件的相同。
37. 为了测试该组件，需要添加一个时钟源和计数器‘tc’输出的查看机制。
38. 将 Cntr8 组件放置在项目原理图中。
39. 将某个时钟组件连接至‘clk’终端，并将其频率设置为 10 kHz。可以使用其它工作频率，但是在示波器上易于观察 10 kHz 的频率。
40. 将一个数字输出引脚组件连接至‘clk’终端，从而在示波器上能够观察它。将它命名为 P0_0_clk，并保持其他默认设置。

    注意：对于 PSoC 4，您不能将时钟直接路由到引脚。要想将时钟路由到引脚，请进行下面操作：

    a. 将数字输出引脚组件放置在您的原理图上。
    b. 在引脚定制器中选择 **Clocking** 选项卡。
    c. 将 **Out Clock:** 设置为 *External*。
    d. 返回 **Pins** 选项卡，并转到 **Output** 子选项卡。
    e. 在 **Output Mode:** 下选择 *Clock*。
    f. 单击“OK”。
    g. 将时钟信号连接到引脚组件上的 out_clk 终端。

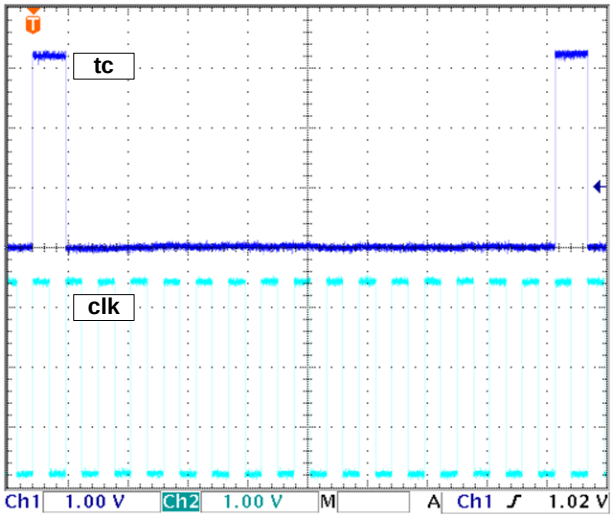
41. 将一个数字输出引脚组件连接到‘tc’终端。将它命名为 P0_1_tc，并保持其他默认设置。计数值为零时，该引脚将为高电平。图 98 显示的是完整的项目原理图。

图 98. 简单计数器的项目原理图



42. 在 cydwr 的 *Pins* 选项卡下，根据引脚的名称将它们分配给 P0[0]和 P0[1]。
43. 现在，您可以编译项目并编程 PSoC。您可以在引脚 P0[0]和 P0[1]上观察时钟和终端计数。
44. 保存项目，编译它并编程 PSoC。
45. 如果将示波器连接到输出引脚，您可以观察‘clock’和‘tc’输出，如图 99 所示。

图 99. 简单计数器的输出



46. 将起始值 15 加载到 A0 内，因此计数周期为 16（从 15 递减计数到 0）个时钟周期。A0 值为 0 时，‘tc’引脚在一个时钟周期内处于高，因为此时，A0 将重载 D0 的值。A0 值非零时，‘tc’为低电平，并且配置再次回转递减 A0。

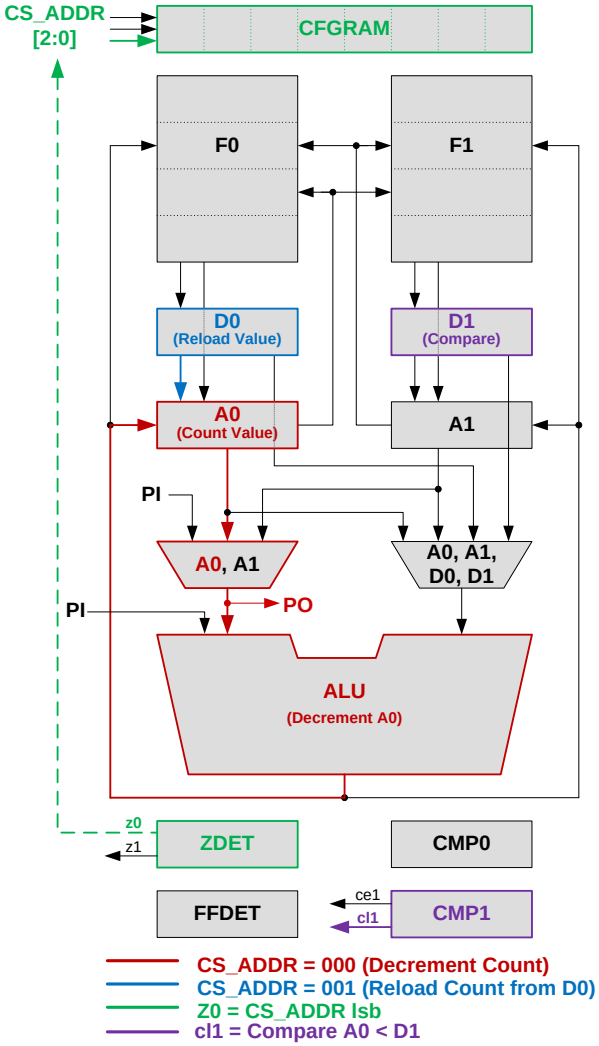
    您刚完成设计第一个基于数据路径的组件，而不需要编写任何代码。

13.1.3 将计数器改为 PWM

PWM 是一个具有比较功能的计数器。为了添加 PWM，您需要使用某个方法将 A0 中的值与另一个固定值进行比较。可以使用 D1 寄存器保存固定比较值，并通过设置比较模块来检查 A0 是否小于 D1。

可以通过图 100 中加亮显示的框图查看该方法：

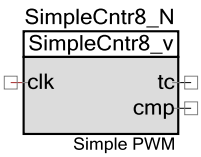
图 100. 加亮显示的简单 PWM 框图



在该示例中，对前一节中构建好的 8 位计数器组件进行修改。您可以从一个空白组件开始，也可以复制先前的计数器组件。下面各步骤假定您对现有的计数器进行修改。

1. 打开 SimpleCntr8_v1_0 的组件符号（.cysym）文件。
2. 按住‘O’，以添加一个数字输出终端。将该终端命名为‘cmp’，并将它放置在图 101 所示的位置。

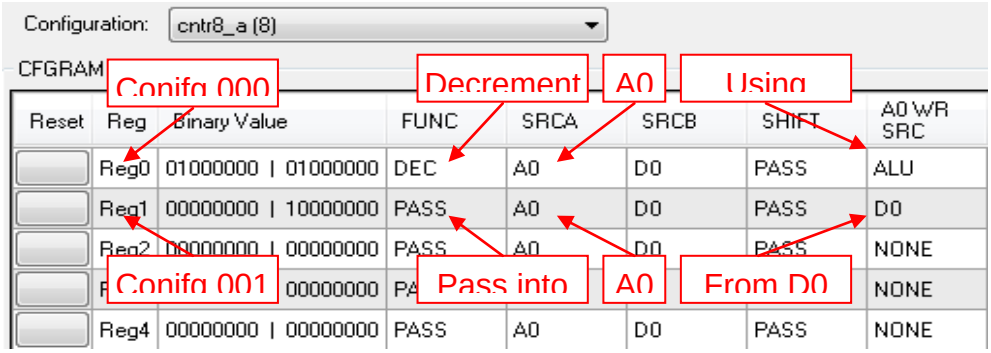
图 101. 带有‘cmp’终端的组件符号



3. 执行“Save All”（全部保存）操作，保存所有更改。
4. 启动数据路径配置工具，并打开 *SimpleCntr8_v1_0.v* Verilog 文件。数据路径配置工具将自动解析 Verilog 文件，并显示出现的第一个配置。

   如前面所述，PWM 是一个具有比较功能的计数器。因此，只需要两个配置，就能递减并重载 A0 中的计数值。这意味着，数据路径配置工具中的大部分设置保持不变，如图 102 所示。

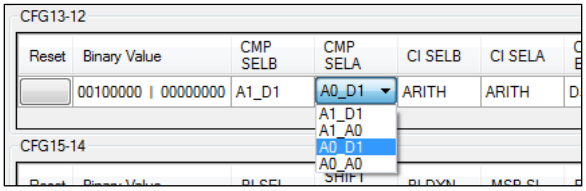
图 102. PWM 的 CFGRAM 设置



   您需要添加(A0 < D1)比较。通过设置可配置比较模块可以实现该操作。

5. 将 **CMP SELA** 设置为“*A0_D1*”，并配置比较功能，使之使用 A0 和 D1 进行比较，如图 103 所示。

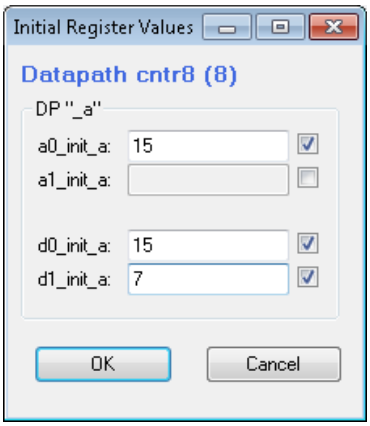
图 103. 比较模块 1 的复用设置



   请注意，A0 和 D0 存储的是计数值和重载值。您需要将默认的比较值添加到 D1 寄存器内。

6. 从菜单中依次选择 **View** > **Initial Register Values**，打开对话框窗口。
7. 选中 **d1_init_a** 的选框。
8. 保持‘a0’和‘d0’的值为 *15*。这样，计数周期将等于 16 个时钟周期。
9. 将‘d1’的值设置为 7。这样，比较值将为 7，如图 104 所示。

图 104. 初始 PWM 寄存器值



10. 点击 **OK**，以保存各寄存器中所设置的值。

11. 保存对 Verilog 文件进行的所有更改。

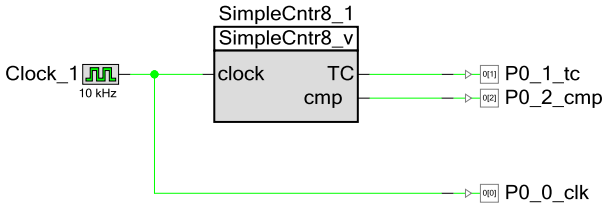
    每当 A0 值小于 D1 值时，比较模块的输出将为高。与对计数器进行的操作相同，通过使用 Verilog 文件可以将符号链接到数据路径硬件。

12. 关闭数据路径配置工具，并打开 PSoC Creator 中的 Verilog 文件。
13. 在 `output tc` 下面，添加 `output cmp`。这样可链接到组件符号中的‘cmp’终端。
14. 将`.cl1()`更改为`.cl1(cmp)`，即可将‘cmp’信号链接到 Compare1 模块中的“小于”比较的输出。
15. 保存对 Verilog 文件进行的所有更改。

    在 Component Catalog 窗口中的 AN82156_Appendix 选项卡下仍然可以查看 PWM 组件和符号。组件在项目原理图中被自动更新。

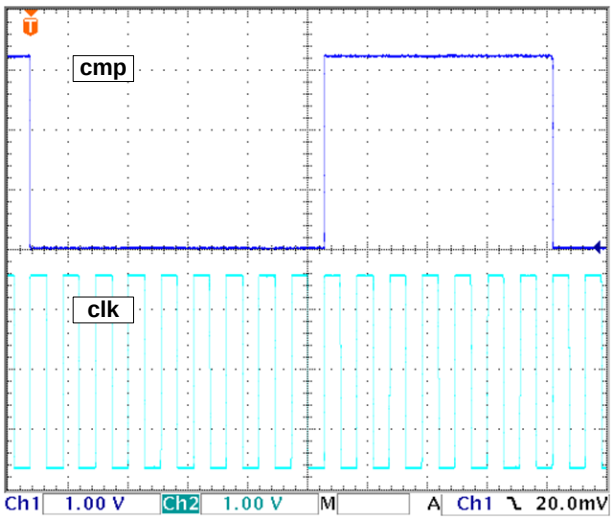
16. 添加一个输出引脚，并将它连接至‘cmp’终端。将该引脚命名为 *P0_2_cmp* ，并将它分配给引脚 P0[2]，如图 105 所示。

图 105. 简单 PWM 的项目原理图



17. 现在，您可以编译项目并编程 PSoC。可以在引脚 P0[0]和 P0[1]上观察时钟和终端计数。在 P0[2]上可观察 PWM 输出。
18. 保存项目，编译它并编程 PSoC。
19. 如果将某个示波器连接至输出引脚，您可以观察‘clock’、‘tc’和‘cmp’输出。图 106 显示的是‘clk’和‘cmp’信号。

图 106. 简单的 PWM 输出



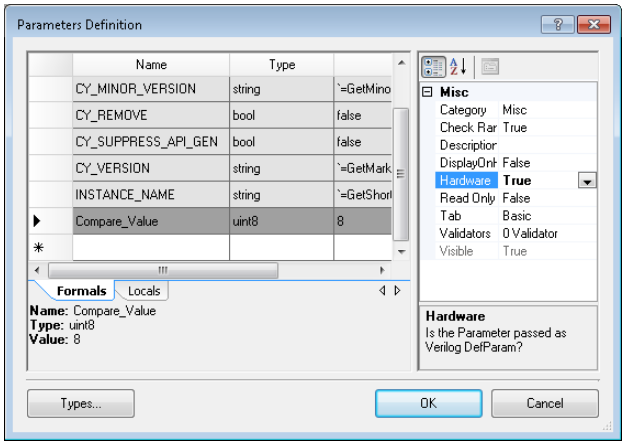
将起始值 15 加载到 A0 和 D0 内，因此计数周期的宽度为 16 个时钟周期。由于将 D1 设置为 7，所以每当 A0 值小于 7 时，‘cmp’引脚将为高。您可以通过修改 D1 中的比较值，对 PWM 进行测试。

13.1.4 添加参数

需要修改某个组件的参数时，更改 Verilog 代码真不方便。另外，当需要使用两个 PWM 的计数周期和比较值均不相同时，你要如何处理？与几乎所有的赛普拉斯组件的工作方式相同，可以给您的组件添加用户可配置参数。

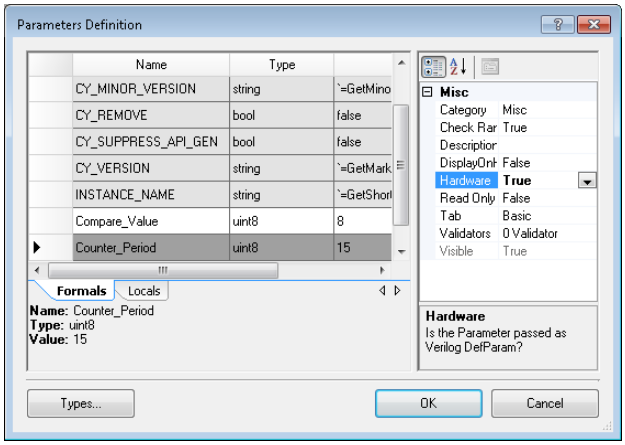
1. 打开该组件的符号编辑器页面（.cysym），并右击空白空间。
2. 从下拉菜单中选择 **Symbol Parameters** 项。
3. 在现有参数下的空行内，输入新的参数。
   - 名称 = Compare_Value
   - 类型 = uint8
   - 值 = 8
4. 在窗口右侧的 **Misc** 设置字段中，将 **Hardware** 标志设为“*True*”，如图 107 所示。
   这样可以将参数开放给 Verilog，因此 UDB 硬件可以使用它。

图 107. 添加一个新的组件参数



5. 在您刚创建的比较值定义的下一行，输入另一个新的参数：
   - 名称 = Counter_Period
   - 类型 = uint8
   - 值 = 15
6. 在窗口右侧的 **Misc** 设置字段中，将 **Hardware** 标志设为“*True*”，如图 108 所示。

图 108. 添加另一个新的组件参数



7. 点击 **OK**，并执行“Save All”操作，保存对该组件进行的所有更改。

   现在，您需要将新的组件符号参数链接到 Verilog 逻辑。

8. 打开组件的 Verilog 文件，查找第 25 行或第 25 行附近的如下文本：

```
//      Your code goes here
```

9. 使用下面的内容代替上述文本：

```
parameter [7:0]   Counter_Period = 8'd0;
parameter [7:0]   Compare_Value  = 8'd0;
```

10. 查找第 29 行或该行附近含有初始寄存器值的文本：

```
cy_psoc3_dp8 #(.a0_init_a(15), .d0_init_a(15), .d1_init_a(7),
```

11. 使用刚定义的参数代替固定值：

```
cy_psoc3_dp8 #(.a0_init_a(Counter_Period), .d0_init_a(Counter_Period),
.d1_init_a(Compare_Value),
```

    这些代码将 A0、D0 和 D1 的初始寄存器值链接到组件参数。后面的示例会介绍如何使用数据路径配置工具实现上述的链接操作。

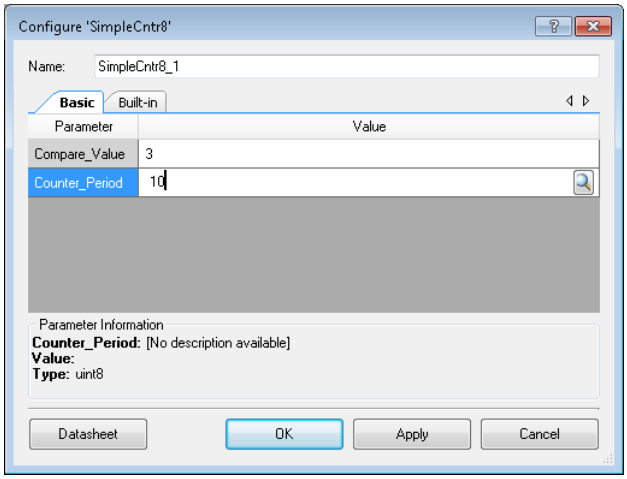
用户不用修改组件就能在编译期间设置各参数。

12. 保存所有更改的内容，编译项目并编程 PSoC。

    先前的默认比较值参数被设置为 7。现在，您可以在项目原理图范围内修改该参数。

13. 返回项目的原理图，然后双击 SimpleCntr8_1 组件，以打开属性对话框。
14. 将比较值改为 3，将计数器周期改为 10，如图 109 所示。

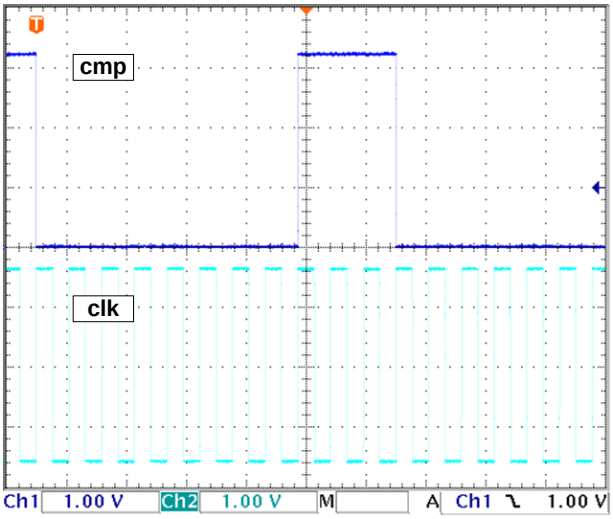
图 109. 选择组件参数



15. 单击 **OK**，应用更改内容。
16. 保存所有更改的内容，编译项目并编程 PSoC。

计数周期和比较输出已被更改，如图 110 所示。

图 110. 根据新参数设置 PWM 输出



计数周期的时长为 11 个时钟周期（周期时长等于‘10+1’，因为计数器先从 10 递减到 0，然后才进行重载），比较值为 3。得到的结果将为 8 个时钟周期的低电平输出和三个时钟周期的高电平输出。

只要所使用的数值是一个 uint8，您可以将参数修改成任何值。您甚至可以在项目中放置多个组件实例，并将它们的参数设置为不同的数值。

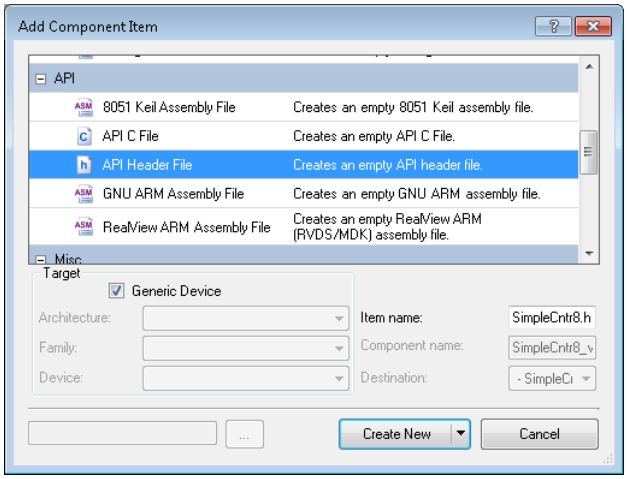
更多有关添加组件参数的信息（包括设置用户的输入值限制），请参考*组件创建指南*。

13.1.5 添加头文件

您可以在设计期间修改 PWM 的性能，另外，通过使用 C 代码修改 PWM 寄存器，您还可以在运行期间修改 PWM。例如，您分别使用 D0 和 D1 寄存器来保存周期值和比较值。为了便于访问这些寄存器，可以创建一个用于定义已用寄存器的头文件。

1. 在 **Components** 选项卡下，右键单击 **SimpleCntr8_v1_0**，然后点击 **Add Component Item**。
2. 在 **Add Component Item** 窗口中，导航到 **API** 部分，然后点击 **API Header File**。
3. 将 **Item name** 改为 *SimpleCntr8.h*，如图 111 所示。

图 111. 添加头文件



4. 点击 **Create New**（创建新项）。将头文件添加到您的组件内。

   现在，可以添加了比较值（D1）和周期值（D0）的定义。

5. 在头文件的 `//[]END OF FILE` 前面，添加下面的定义：

```
#define `$INSTANCE_NAME`_Period_Reg (*(reg8 *) `$INSTANCE_NAME`_cntr8_u0__D0_REG )

#define `$INSTANCE_NAME`_Compare_Reg (*(reg8 *) `$INSTANCE_NAME`_cntr8_u0__D1_REG )
```

   这两个定义允许您直接写入固件中的 D0 和 D1 寄存器。*cyfitter.h* 文件包含了项目中所使用的各组件的寄存器定义集。如果您所进行的操作与本应用笔记中所描述的各步骤不同，那么，您可能需要在 *cyfitter.h* 文件中添加寄存器定义，并使用它们。

   如果想要在运行期间更新比较值，只要写入到您刚创建的定义即可。如果将您的组件命名为SimpleCntr8_1，那么您的C代码如下：

```
SimpleCntr8_1_Period_Reg = 0x08;

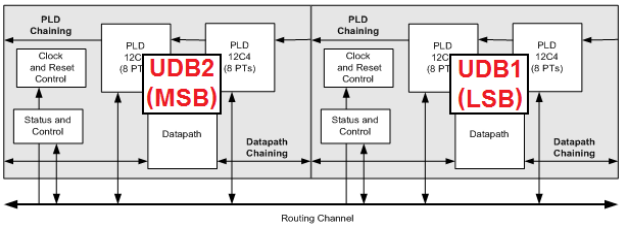
SimpleCntr8_1_Compare_Reg = 0x02;
```

   该代码分别更新了周期值和比较值为0x08和0x02。更多有关如何使用这些定义的信息，请参考*组件创建指南*。请注意，直接写入到寄存器内的方法（如上面所述的C代码）只适用于8位寄存器。对于16位或16位以上的寄存器，需要使用下一个项目中所介绍的其他方法。

13.2 项目 2 – 16 位 PWM

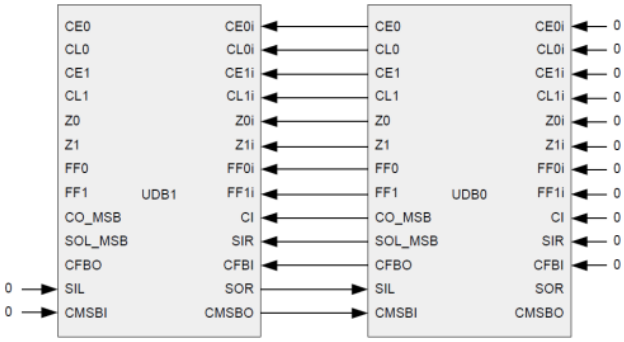
该示例项目介绍了数据路径链接的概念。数据路径具有连接到相邻数据路径的专用信号。通过这些信号，可以创建宽度多达 32 位的功能。在该示例中，您创建的另一个 PWM 类似于第一个示例项目中的 PWM，但它的宽度为 16 位，如图 112 所示。

图 112. 带有链式 UDB 的 16 位功能



每个数据路径中的 ALU 用于将进位、移位数据以及条件信号链接到最接近的数据路径，如图 113 所示。按最低有效字节到最高有效字节的方向链接所有的条件和捕捉信号。左移是从最低有效字节链接到最高有效字节。右移是从最高有效字节链接到最低有效字节。

图 113. 数据路径的链接流程



该示例假定您已经熟悉了前一个示例中所介绍的概念。本应用笔记仍介绍一个完整的 16 位 PWM，以供参考。

13.2.1 16 位 PWM 组件的详情

16 位 PWM 的基本功能与第一个示例项目中 8 位 PWM 的基本功能相同。在该两个示例中，计数值都会递减，并且，当计数值小于比较值时，输出将为高电平。不同的是，您需要使用两个数据路径才能处理 16 位的所有操作。

13.2.2 16 位 PWM 组件的创建步骤

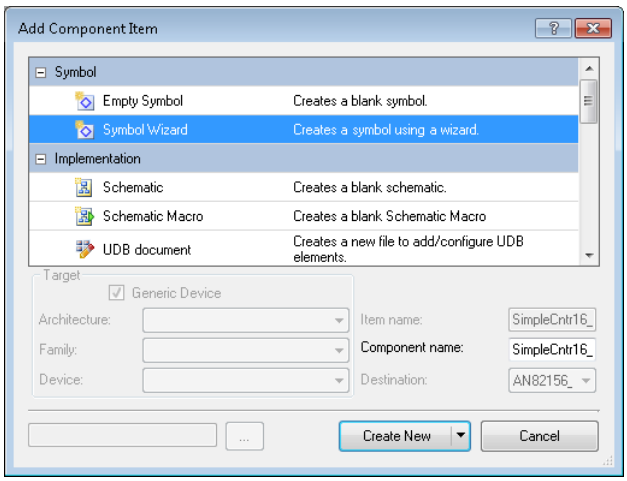
为了避免困惑，请勿对第一示例项目中的组件进行更改，而应该创建新的组件。创建组件的基本步骤与前一示例的相同。

1. 启动 PSoC Creator，并打开前一示例所使用的 **AN82156_Appendix** 工作区。向工作区内添加一个新项目，并将其命名为“*16bitCounter*”。

   您可以使用新的工作区，但是该示例假定您正在使用与前一项目相同的工作区。如果使用同样的工作区可以简化库和依赖关系的管理。

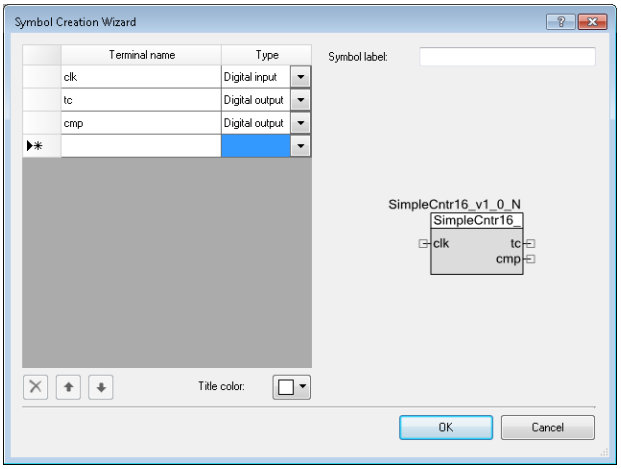
2. 使用 Symbol Wizard 将新的组件项添加到 ***AN82156_Appendix_Lib*** 内，如图 114 所示。该示例中的组件名称为“*SimpleCntr16_v1_0*”。

图 114. 创建新的符号



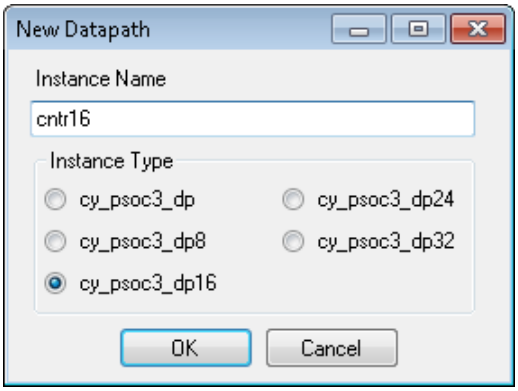
3. 与 8 位 PWM 示例中的情况相似，请添加一个 *clk* 输入、一个 *tc* 和一个 *cmp* 输出，如图 115 所示。

图 115. 为 PWM 添加终端



4. 右击符号原理图页面，并添加符号属性：
   - Doc.APIPrefix = *SimpleCntr16*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Cntr16*
   - Doc.DefaultInstanceName = *SimpleCntr16*
5. 为新的组件生成 Verilog 文件，保留所有默认设置。
6. 点击 **Save All**，保存所有更改。
7. 启动*数据路径配置工具*，并打开新创建的 Verilog 文件。
8. 添加新的数据路径配置。该示例中的配置名称为“*cntr16*”。选中“**cy_psoc3_dp16**”，如图 116 所示。

图 116. 创建新的 PWM 数据路径



9. 点击 **OK**，创建数据路径配置。

   请注意，有 *cntr16_a(16)*和 *cntr16_b(16)*两个配置。这两个独立的数据路径配置将被添加到 Verilog 文件内。‘a’配置用于最低有效位（LSB），‘b’配置用于最高有效位（MSB）。
10. 根据 8 位 PWM 示例中的内容设置“_a”配置 CFGRAM 和 CFG13-12 部分，如图 117 和图 118 所示。

图 117. SimpleCntr16 组件配置

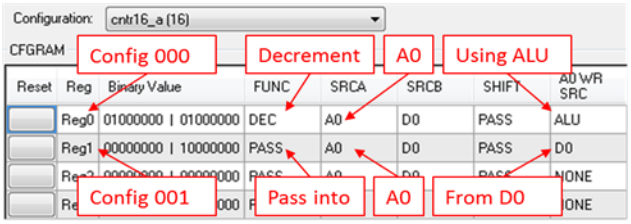
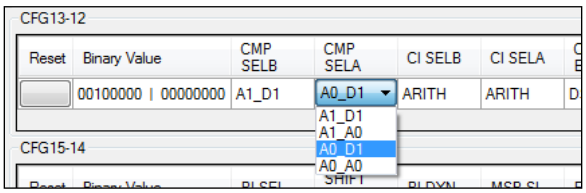


表 9. 数据路径配置示例 2

REG	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
000	DEC	A0	D0	PASS	ALU
001	PASS	A0	D0	PASS	D0

图 118. 比较模块 1 的复用设置



请注意，ALU 功能和比较模块的设置与 8 位示例的相同。因为您尚未配置链路，所以这些设置仍一样的。

11. 保存对 Verilog 文件进行的更改。

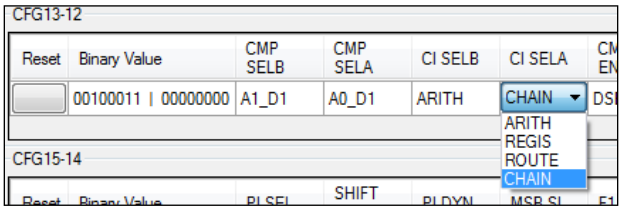
    由于“_b”配置将为 PWM 的高 8 位，所以您需要对该配置进行同样的修改。数据路径配置工具可将某个配置的设置内容复制给另一个。

12. 依次选择 **Edit** > **Copy Datapath** 来复制该配置。
13. 切换为‘cntr16_b’配置，并选择 **Edit** > **Paste Datapath** 来粘贴‘_a’配置到‘_b’配置。
14. 保存对 Verilog 文件进行的更改。

    配置该链路；仅有‘_b’配置使用这些设置。

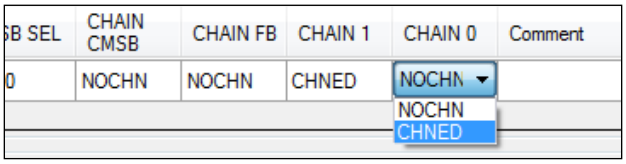
15. ‘在 cntr16_b’配置中，将 CI_SELA 设置为“CHAIN”，如图 119 所示。这样配置来自前一数据路径的进位信号。

图 119. 配置进位信号的链接



16. 将 CHAIN1 和 CHAIN0 设置为“CHNED”，如图 120 所示。这样会将各比较条件（ce0、ce1、cl0、cl1、z0、z1、ff0、ff1）链接在一起。

图 120. 配置数据路径链接

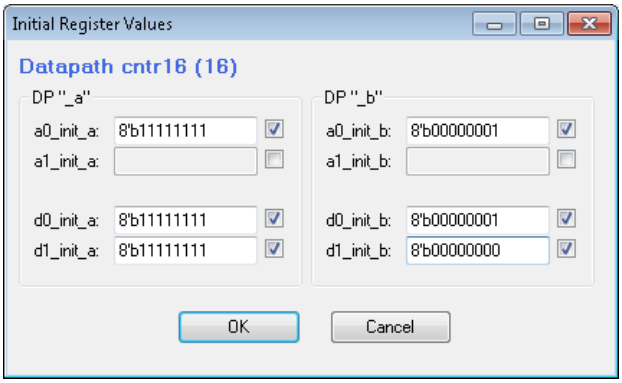


17. 保存对 Verilog 文件进行的更改。

    您还需要设置各个寄存器的初始值。前一示例已经说明了如何使用各参数来配置周期值和比较值。因为该示例着重于介绍链接方面的内容，所以使用了固定值使示例更加简单。

18. 依次选择 **View** > **Initial Register Values**。
19. 设置 DP“_a”和 DP“_b”的值，如图 121 所示。该示例显示的是二进制的值，表示每个寄存器仍是 8 位的宽度。

图 121. 16 位 PWM 的初始寄存器值



请注意，“_a”配置为 LSB（最低有效位），则“_b”配置为 MSB（最高有效位）。这些值向 PWM 提供了一个 511 个时钟周期的时长和数值为 255 的比较值。

20. 保存对 Verilog 文件进行的更改，并关闭数据路径配置工具。
21. 打开 *SimpleCntr16_v1_0.v* Verilog 文件。

    请注意，Verilog 代码中具有两个数据路径的配置。这两个数据路经链接在一起，构成 16 位 PWM。接下来，您需要添加一些额外的信号，以用于 Verilog 逻辑。
22. 在第 25 行上或附近的'#start body'后面加上以下代码：

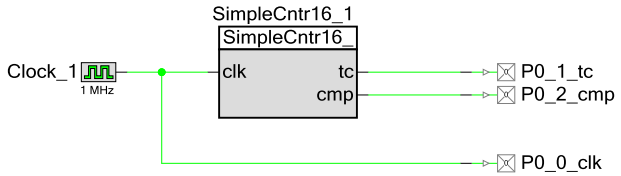
```
// Unused Datapath Connections
wire tc_lsb, cmp_lsb;
```

    该硬件链接类似于 8 位 PWM 的链接。但不同之处在于该链接的输出为 2 位宽而不是 1 位宽。因为在数据路径配置工具中您已经将 *Chain0 和 Chain1* 串连在一起，所以输出的高位包含了最终结果。例如，z0 输出的高位表示何时两个数据路径的 A0 寄存器值全部为零。因此，您需要将‘tc’和‘cmp’分配给高位，并将第 22 步骤中所定义的两条线路用于低位。这些线路将不被使用，但如果不添加它们，Verilog 合成器会生成警告。
23. 将.clk()设置为.clk(clk)。
24. 将.z0()设置为.z0({tc, tc_lsb})。
25. 将.cl1() 设置为.cl1({cmp, cmp_lsb})。
26. 将.cs_addr(3'b0)设置为.cs_addr({2'b0,tc})。
27. 点击 Save All，保存所有更改。

    目前，可将组件用于您的项目。
28. 将 *AN82156_Appendix_Lib 设置为 16bitCounter* 项目所包含的内容。

    新组件将显示在“组件目录”中，并可用于您的项目。
29. 将 Cntr16 组件施入原理图。
30. 将某个时钟组件连接到‘clk’终端，并将它的频率设置为 1 MHz。
31. 将数字输出引脚组件分别连接至‘clk’、‘tc’和‘cmp’终端，如图 122 所示。

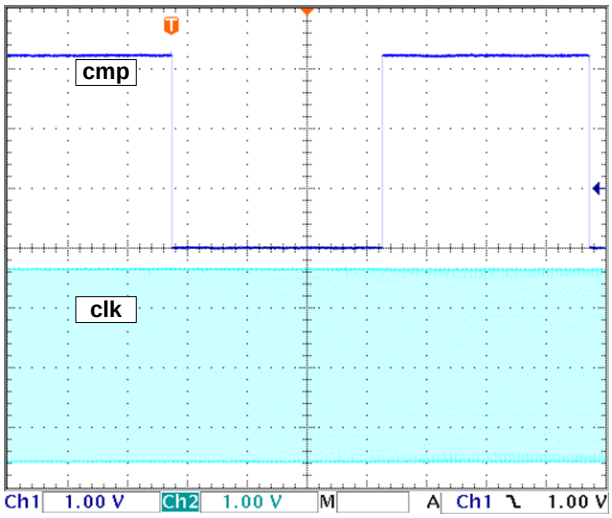
图 122. 简单的 16 位 PWM 项目原理图



32. 根据上述内容命名各引脚，并按它们的名称将它们分配给 P0[0]、P0[1]和 P0[2]。
33. 保存所有更改的内容，编译项目并编程 PSoC。

    您可观察这些输出，发现该周期值和比较值比以前所创建的 8 位 PWM 的大得多，如图 123 所示。可将它们设置为任何一个 16 位数值。

图 123. 简单的 16 位 PWM 输出



您可通过使用链接方式使各功能的宽度达 32 位。将在该示例中所介绍的原理应用于较大的功能。

13.2.3 PSoC 3 的 16 位组件头文件

如前面所述，对 16 位寄存器进行写操作和对 8 位寄存器进行的不一样。由于可以忽略处理器和数据路径寄存器之间字节顺序的差异，因此您可直接写入到 8 位寄存器。当您写入到 16 位或宽度更大的寄存器中时，需要考虑到该字节顺序的差异。

PSoC 3 中 8051 的字节顺序不同于外设寄存器的顺序。为了便于对寄存器进行写操作，赛普拉斯提供了一些宏，如：CY_SET_REG16、CY_SET_REG24、CY_SET_REG32。这些宏包含两个参数：第一个是您所需要设置的寄存器地址，第二个是您想设置的值。这些宏处理您所需要的所有字节顺序的交换。因此，您必须掌握各数据路径的寄存器地址。定义这些宏的过程与之前的过程是相同的。不同之处仅在于定义的过程，您要使用 `((reg8 *)` 而不是使用 `(*(reg8 *)`。它提供了指向数据路径寄存器的指针。应该将_PTR 添加到所定义的名称的结尾，以便区分。

更新某个值时，请使用在头文件中所定义的 CY_SET_REG16 和指针。请再次查看随附的示例项目。

13.3 项目 3 — 递增/递减计数器

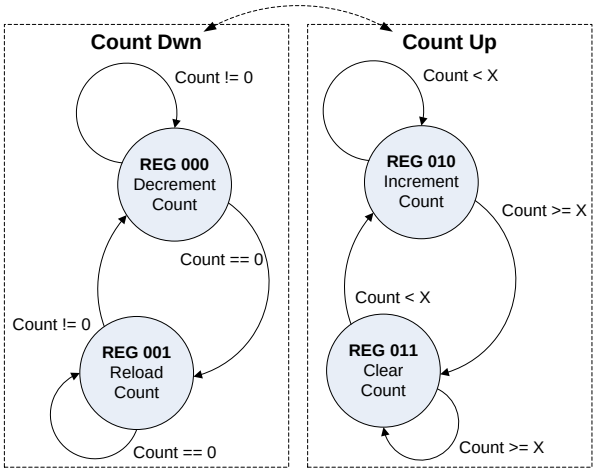
该示例介绍了将高级特性添加到数据路径组件的过程。更新了相同的 PWM 基本概念，以进行递增或递减计数。计数方向取决于在设计时或运行时用户设置的参数。

该示例假设您已经熟悉了前几个示例项目中所介绍的概念。本应用笔记中提供了一个完整的递增/递减计数 PWM 项目。

13.3.1 更多的详细信息

这种简单的递减计数 PWM 项目使用两种状态。要想实现某个递增/递减计数器，需要使用四种状态，如图 124 所示。

图 124. 递增/递减计数器的状态图



根据所设置的参数，数据路径将递减或递增 A0。

13.3.2 创建示例项目的步骤

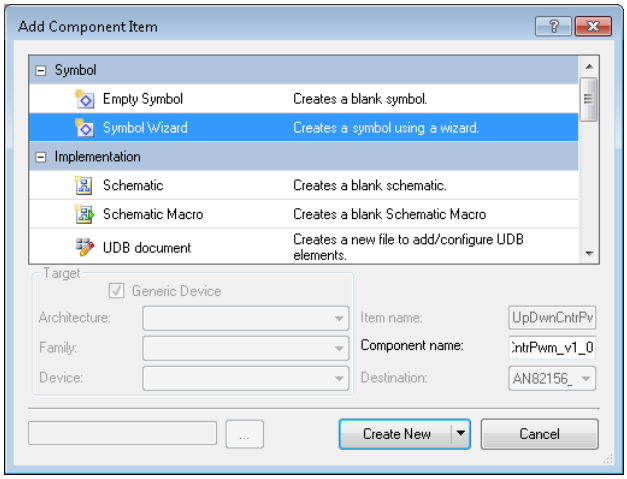
为了避免困惑，请勿更改前几个示例项目中的组件，而应该创建一个新的。创建组件的基本步骤与前一示例中的相同。

1. 启动 PSoC Creator，并打开简单 8 位示例项目所使用的“**AN82156_Appendix**”工作区。向工作区添加名称为“*UpDwnCntrPwm*”的新项目。

   您可以使用新的工作区，但是该示例假设您正在使用与前一项目相同的工作区。这样会简化库管理和依赖关系。

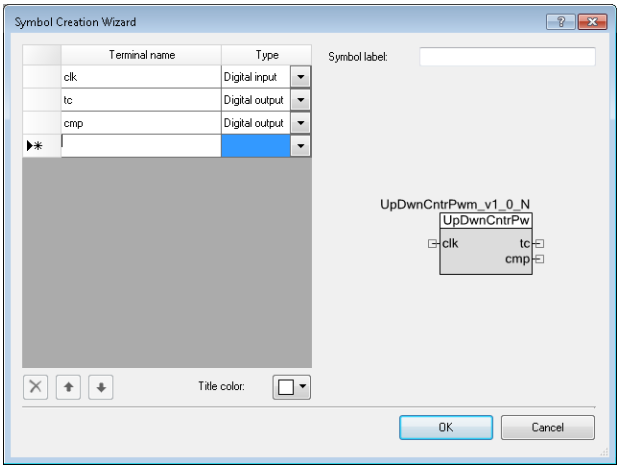
2. 使用 Symbol Wizard 将新的组件项添加到 ***AN82156_Appendix_Lib*** 内，如图 125 所示。该示例中使用的组件名称为“*UpDwnCntrPwm_v1_0*”。

图 125. 为递增/递减计数器添加一个新符号



3. 同 8 位 PWM 组件中的情况一样，添加一个 *clk* 输入、*tc* 和 *cmp* 输出，如图 126 所示。

图 126. 向递增/递减计数器添加终端

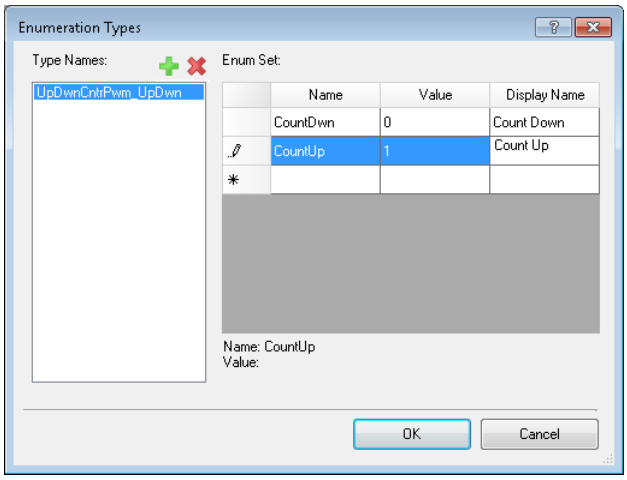


4. 右击符号原理图页面，并添加符号属性：
   - Doc.APIPrefix = *UpDwnCntrPwm*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/UpDwnCntrPwm*
   - Doc.DefaultInstanceName = *UpDwnCntrPwm*

   您需要添加几个用户可更改的参数，如：计数器周期、比较值和递增或递减计数模式。

   设置计数模式时，要定义新参数类型。
5. 右键单击符号编辑器页面，并打开符号参数对话框。
6. 单击 **‘Types’**（类型）按键打开窗口，创建新的参数类型。
7. 单击绿色的‘**+**’按键来添加新类型。将它命名为 *UpDwnCntrPwm_UpDwn*。
8. 将各值输入到 ***Enum Set*** 字段，以确定‘CountDwn’和‘CountUp’定义，如图 127 所示。

图 127. 创建组件参数的新类型



9. 单击 **OK** 返回符号参数对话框。

   您可以将各参数分配给这种新类型，并将初始值设置为 0（CountDwn）或 1（CountUp）。
10. 根据前几个示例项目中的操作，为组件输入三个新参数；请参见表 10 和图 128。

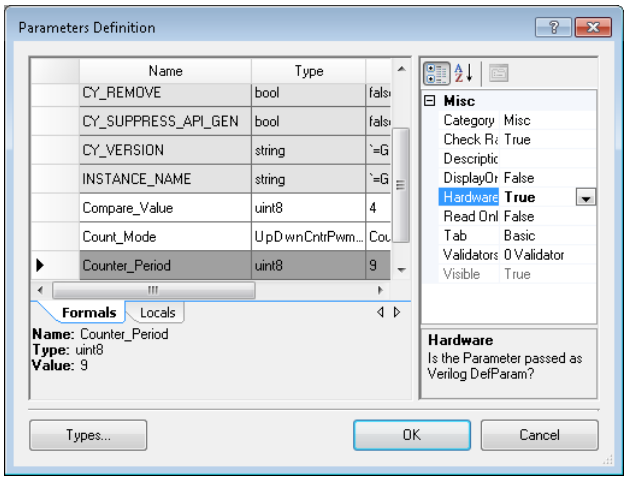
表 10. 递增/递减计数参数

名称	类型	值
Compare_Value	uint8	4
Count_Mode	UpDwnCntrPWM_UpDwn	Count Down
Counter_Period	uint8	9

Count_Mode 参数使用了新类型和值的定义。

11. 将三个新参数的 **Hardware** 标志设置为‘True’，如图 128 所示。

图 128. 添加新的组件参数



12. 点击 **OK**，并保存对组件进行的所有更改。
13. 在符号编辑器中右击空白位置，并为新组件符号生成 Verilog 文件，另外，保留所有默认设置。
14. 点击 Save All，保存所有更改。
15. 启用“数据路径配置工具”，并打开生成的 *UpDwnCntrPwm_v1_0.v* 文件。

    前两个配置类似于简单的 8 位 PWM 示例中的配置，但您还需要另外两个配置，用于递增计数：
16. 创建一个新的 Datapath 配置，称为“UpDwn”。使用‘cy_psoc3_dp8’类型。
17. 配置 CFGRAM，以符合表 11。

表 11. 三个数据路径配置示例

REG	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
000	DEC	A0	D0	PASS	ALU
001	PASS	A0	D0	PASS	D0
010	INC	A0	D0	PASS	ALU
011	XOR	A0	A0	PASS	ALU

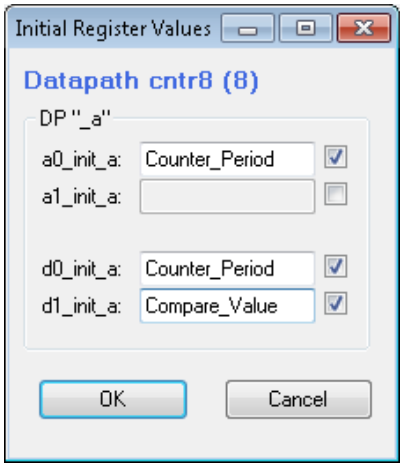
18. XOR 配置可用于清除计数寄存器。计数寄存器递增计数到周期值后，它将对自己进行 XOR 操作，即清除所计数的值为零。
19. 与简单的 8 位 PWM 示例中一样，将 **CMP SELA** 字段设置为 *‘A0_D1’* 来配置 D1 为比较值。

    在上述部分中，您已经添加了用户可调整的一些参数。请将它们的名称输入到“Initial Register Values”（初始寄存器值）字段内，这样就不需要手动编辑 Verilog 文件进行更改了。

    例如，与 8 位 PWM 示例中的情况一样，将各参数命名为“Counter_Period”和“Compare_Value”。

20. 打开 Initial Register Values 窗口，并将各参数名称输入到 A0、D0 和 D1，如图 129 所示。

图 129. 将各参数作为初始寄存器值



21. 单击 **OK**，保存对 Verilog 文件所进行的更改内容。
22. 保存更改内容，并关闭数据路径配置工具。

    接下来，您需要将一些 Verilog 代码添加到组件以实现这些性能。
23. 打开 *UpDwnCntrPwm_v1_0.v* 文件，并寻找下面文本所在的位置：

```
//  Your code goes here
```

24. 使用下面的内容代替上述文本：

```
// Control Register "pwmCntlReg" bits
localparam CNTL_CNT_UP_DWN = 0;
// Compare0 less than signal
wire up_reload;
//Zero Detect Signal
wire zero_detect;
// Up/down control
wire upDwn;
// Signal to control reload of counter
wire reload;
// Control register signals
wire [07:00] control;



// Up/down control
assign upDwn=control[CNTL_CNT_UP_DWN];
// Logic for the 'reload' signal
assign reload = ( upDwn ) ? ( up_reload ) : ( zero_detect );
//assign termnical count output
assign tc = reload;
    // Control register instance. This text is
    // found in the Component Author Guide.
    cy_psoc3_control #(.cy_init_value (Count_Mode), .cy_force_order(`TRUE))
```

```
pwmCntlReg(
/* output [07:00] */ .control(control)
);
```

25. 为数据路径的逻辑添加下列链接：
    - 将.clk()改为为.clk(clk)。
    - 将.cs_addr(3'b0)改为.cs_addr({1'b0, upDwn, reload})。
    - 将.ce0 ()改为.ce0(up_reload)。
    - 将.z0()改为.zo(tc)。
    - 将.cl1()改为.cl1(cmp)。

    现在您已经完成了所有需要的修改，以下简单回顾这些改变来帮助您更好地理解。首先，您所添加的控制寄存器（cy_psoc3_control）允许在运行时通过代码更改计数的方向。欲了解更多有关控制寄存器和不同选项的信息，请参考*组件创作指南*。

    如果想用主代码控制计数的方向，请在头文件中添加控制寄存器的定义。参见随附的示例项目。

    下面请注意，.cs_addr 有两个控制位而不是一个。在上述示例中，数据路径只有两种状态，因此使用一个控制位足够。由于这里有四种状态，因此需要使用两个控制位。

    另外，还要注意您正在使用‘reload’信号而不是‘tc’信号来控制数据路径的配置。之所以这样做，因为现在您可以递增计数，不能仅使用 ZDET 来表示某个计数周期的结束。您需要使用 ce0 比较。因此，配置该计数器进行递增计数时，只要 A0 中的值不等于 D0 中的值，它将连续进行递增计数。当该计数器值等于 D0 值时，寄存器会重新加载计数值。

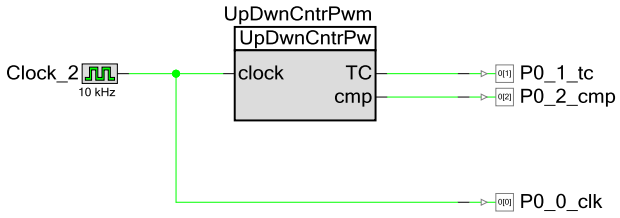
    ‘upDwn’信号确定正在进行递增计数（INC、XOR）还是递减计数（DEC、Load）。

26. 保存对 Verilog 文件进行的所有更改。

    对 Verilog 文件进行更改后，可以在您的项目中使用组件。根据第一个示例项目添加所有相同组件。

27. 使 ***AN82156_Appendix_Lib*** 成为 UpDwnCntrPwm 项目的附属部分。
28. 将某个 UpDwnCntrPwm 组件施放到项目原理图中。
29. 将一个组件时钟连接至该组件的‘clk’终端，并将它的频率设置为 10 kHz。
30. 将各个数字输出引脚组件分别连接至该组件的终端（即 P0_0_clk、P0_1_tc 和 P0_2_cmp），如图 130 所示。

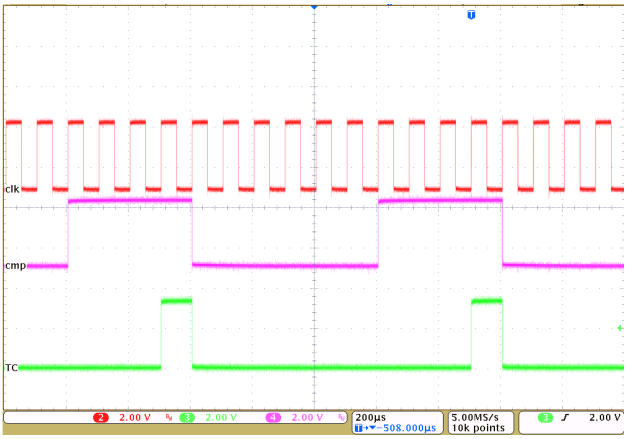
图 130. 递增/递减计数器的 PWM 项目原理图



31. 保存所有更改的内容，构建项目并编程 PSoC。

    将默认的比较值参数和周期值分别设置为 4 和 9。通过使用所添加到项目的引脚可以观察它们。

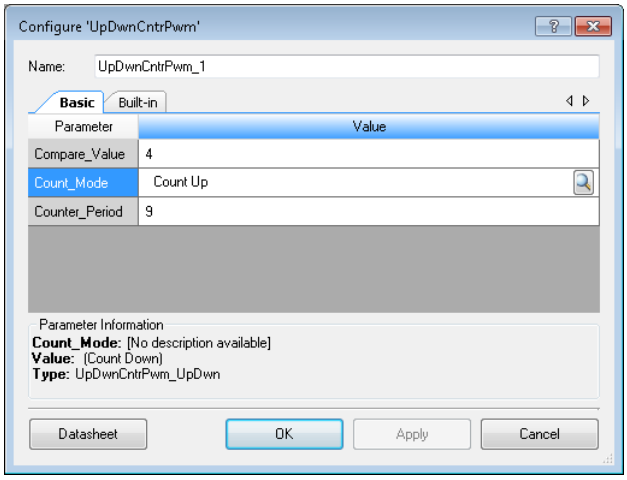
图 131. 递减计数 PWM 波形



您可以根据上述简单 PWM 示例中的方法更改周期和比较参数。也可以更改模式参数，使 PWM 进行递增计数而不是递减计数。

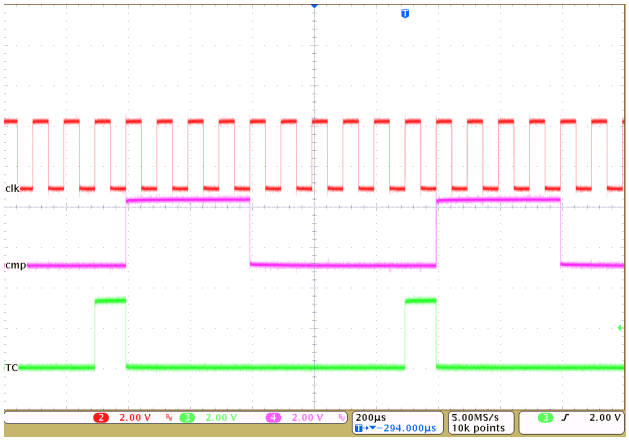
32. 返回项目的原理图，并双击 UpDwnCntrPwm 组件以打开组件属性。
33. 将 **Count_Mode**（计数模式）设置为‘Count’，如图 132 所示。

图 132. 将 PWM 设置为递增计数



34. 单击 **OK**，应用更改内容。
35. 保存所有更改的内容，构建项目并编程 PSoC。
36. 您可以观察，这是一个递增计数，因为将零值重载到计数器时，TC 下降沿后 cmp 值为高，如图 133 所示。

图 133. 递增计数 PWM 波形



37. 请注意，TC 后，计数值低于比较值，并进行递增计数，因此输出为高。刚进入递减计数模式时，计数值大于比较值，并进行递减计数，因此输出为低。

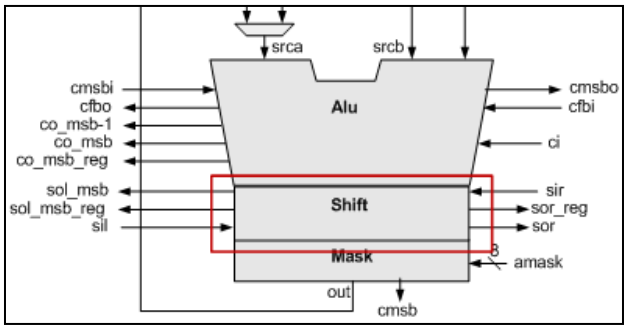
13.4 项目 4 — 简单的 UART

该示例项目展示了一个使用单个数据路径创建的简单 TX UART。我们将不会向您介绍创建组件的每一步骤。但您可以在相关示例项目中查看组件和 Verilog 文件。在各个已完成的示例的 ***AN82156_Appendix_Lib*** 项目中寻找“***Simple_Tx***”组件。如何使用该组件的一个示例被包含在相同的工作区内，并位于“***SimpleTx***”项目中。

TX UART 组件的详细信息

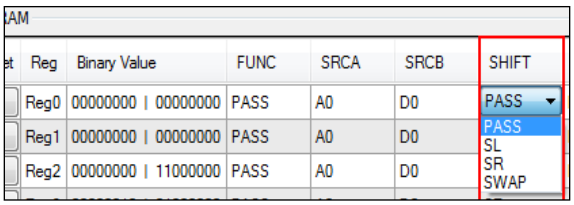
在此组件中，数据路径的用途很简单。所使用的数据路径操作仅包括：移位并将 F0 中的值加载到 A0 内。ALU 输出端上具有一个移位器，如图 134 所示。

图 134. 移位器模块



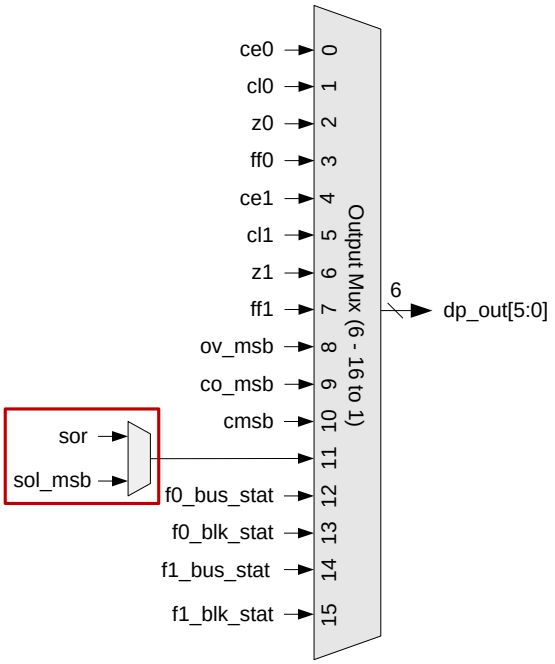
该移位器能够将各个位左移或右移，并能够交换各半字节。数据路径的每个配置均可独立设置移位器的操作。可在数据路径配置工具的动态配置区中设置该选项，如图 135 所示。

图 135. 移位操作设置



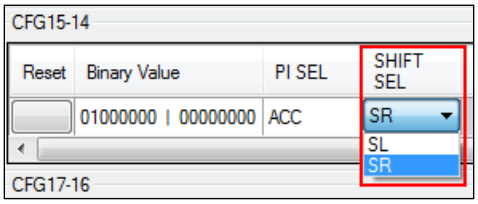
数据路径输出复用器上仅有一个移出（SO）输出端。右移出（Shift Out Right）和左移出（Shift Out Left）共同使用该输出，如图 136 所示。

图 136. 数据路径输出的复用器框图



您必须正确地配置该复用器。可在 CFG15-14 *SHIFT SEL* 中的静态配置区内进行该操作，如图 137 所示。

图 137. SHIFT SEL 的配置



在该示例中，创建 Verilog 状态机来控制数据路径的配置。在 UDB PLD 中实现该状态机。它确定了 UART 传输需要进行的下一部分，如起始、数据或停止。

如果您查看了 Verilog 代码，请注意下面代码行：

```
reg  [1:0]  state;  // Main state machine variable and datapath control
```

如果您查看数据路径的输入和输出，请注意 *state*（状态）用于控制 CS_Addr:

```
/*  input   [02:00]          */  .cs_addr({1'b0, state}),
```

通过使用 case 语句可以实现 Verilog 状态机，从而更改数据路径配置/指令值。

```
case (state)

        STATE_IDLE:…

        STATE_START:…

        STATE_DATA:…

        STATE_STOP:…

        endcase
```

请注意，上述每一个案例均有如下定义：

```
// Main State machine states

// Idle / Stop bit 1
localparam STATE_IDLE  = 2'b00;
// Stop bit 2
localparam STATE_STOP  = 2'b01;
// Start bit
localparam STATE_START = 2'b10;
// Data
localparam STATE_DATA  = 2'b11;
```

状态机仅有四种状态，并且这四种状态将用于控制数据路径的配置。因此，数据路径需要四个独特配置：

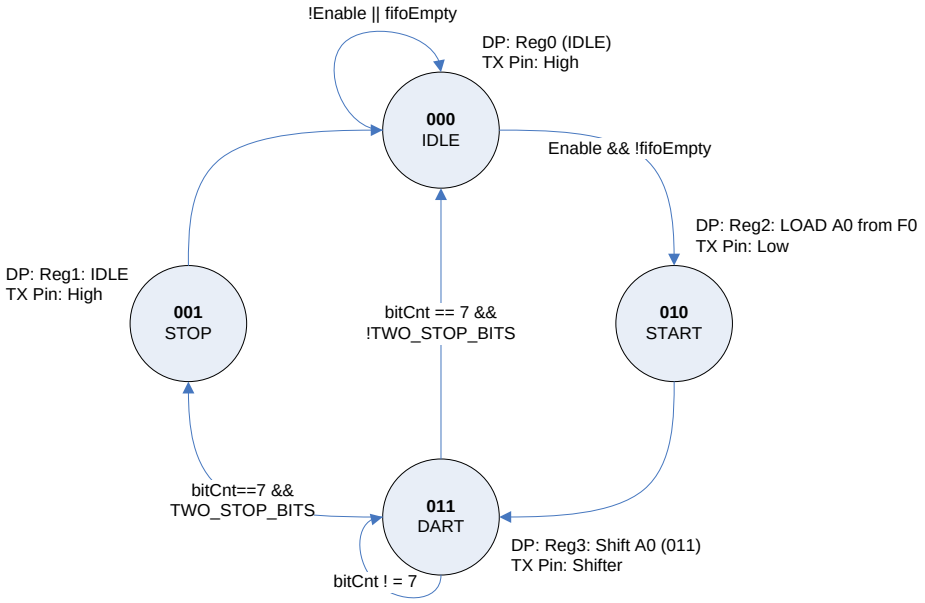
表 12. 简单的 Tx 数据路径配置

REG	FUNC	SRCA	SHIFT	A0 WR SRC
000	通过	A0	无	无
001	通过	A0	无	无
010	通过	A0	无	F0
011	通过	A0	SR	ALU

当代码移入 Verilog 状态机时，它改变了数据路径的配置。这是一个普通情况。几乎所有复杂的数据路径组件需要使用一个 Verilog 状态机来定序数据路径的配置。

图 138 显示的是简单的 TX Verilog 状态机如何工作。

图 138. TX UART Verilog 的流程图



首先，通过监控 fifoEmpty 状态位，状态机将等待新数据被写入到 FIFO 内。FIFO 存有新数据后，它会通过将 TX 线设为低电平（LOW）来发送一个 START（起始）位。在该状态下，数据路径将 F0 中的值加载到 A0 内。

在下一个状态中，数据路径将移出 A0 中的数据。该操作结束后，它会发出一个 STOP（停止）位。它可以发出一个或两个停止位。

处于数据状态时您已经移出了数据。在起始状态中将 TX 线置为低电平，则在停止状态和闲置状态时会将该线上拉到高电平。通过以下 Verilog 代码可执行该操作。

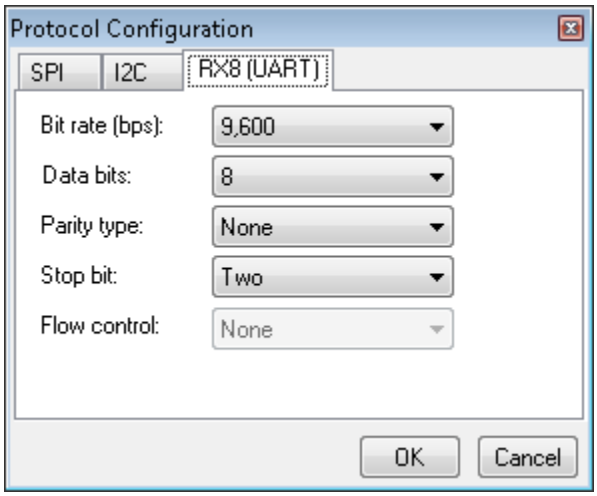
```
// This next statement determines the tx output.  if in data state, output the shift
register output "srOut", else output a low during the start state, and a high during
stop state.
assign tx = (state[1:0] == STATE_DATA ) ? srOut : ( !state[1] );
```

如果不处于数据状态，它会反转输出 TX 线上‘state’（状态）的 msb 位。处在起始状态时，msb 为‘1’，因此其输出将为‘0’。则在停止和闲置状态时，msb 为‘0’，所以其输出为‘1’。

该项目的主代码使能带两个停止位的组件，然后以 9600 波特连续传输 0 到 10 数值。将接收器配置为 9600 波特和两个停止位。

与 PSoC Creator 安装的是一个称为桥接控制屏幕（BCP）的程序。您可以使用 BCP 来接收 RX 字符。在 BCP 中，连接到 COM 端口（TX 输出已经连接到该端口）。在 **Tools**（工具） **> Protocol Configuration**（协议配置）中，配置 RX8 （UART），如图 139 所示。

图 139. BCP RX8 配置

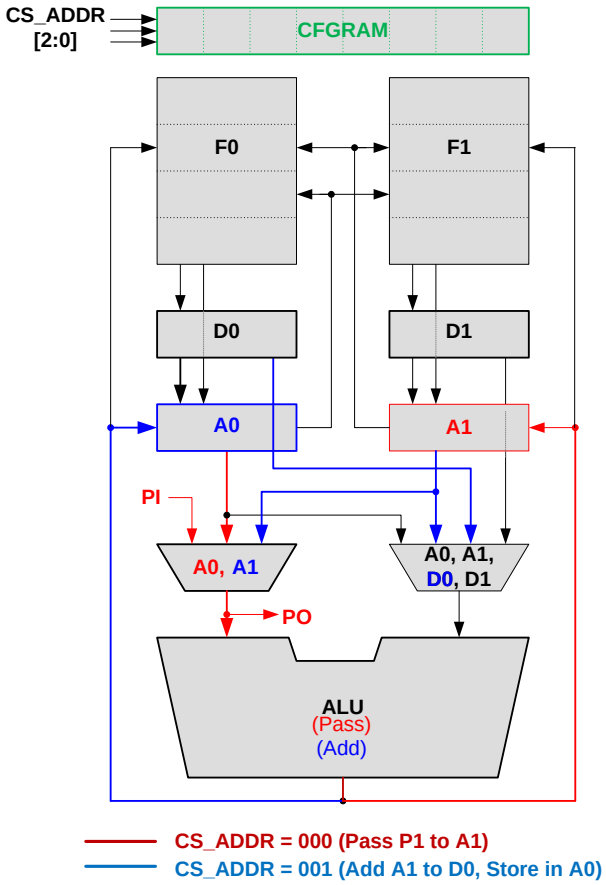


在 BCP 的编辑器中，添加以下文本：rx8 x x x x x x x x x x。从选定的 COM 端口中读取 11 个字节。然后，您可以打 **Repeat**（重复）按键来连续接收数据。

13.5 项目 5 — 并行输入和并行输出示例

该示例项目演示了如何使用数据路径的并行输入和并行输出。通过使用 8 位并行加法器可以演示这些性能。加法器从并行输入（PI）读取 8 位并行值，并将该值存储在 A1 内。然后它将存储在 A1 中的值添加为 D0 中的固定值。然后，该值将在并行输出（PO）上输出。

图 140. 并行加法器的实现



该示例假设您已经熟悉了前几个示例项目中所介绍的概念。本应用笔记中提供了一个完整的 PI_PO_Example 项目。

13.5.1 创建示例项目的步骤

为了避免混乱，请勿更改前几个示例项目中的组件，而应该创建一个新的。组件创建的步骤基本相同。

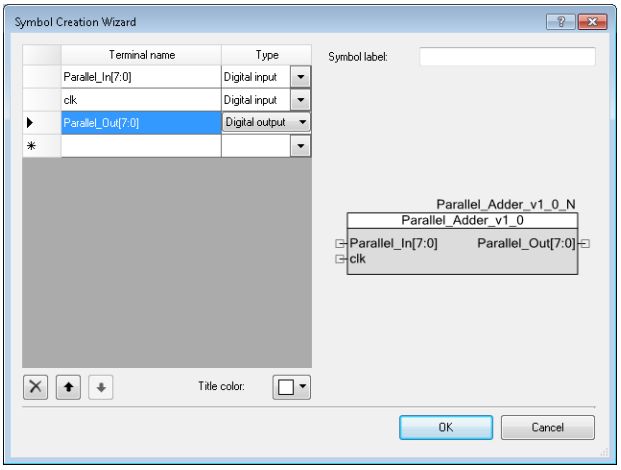
1. 启动 PSoC Creator，并打开简单 8 位示例项目所使用的“**AN82156_Appendix**”工作区。向工作区添加名称为“*PI_PO_Example*”的新项目。

   您可以使用新的工作区，但是该示例假定您正在使用与前一项目相同的工作区。这样会简化库管理和依赖关系。

2. 使用 Symbol Wizard 将新的组件项添加到 ***AN82156_Appendix_Lib*** 内。该示例使用“*Parallel_Adder_v1_0*”作为组件名称。

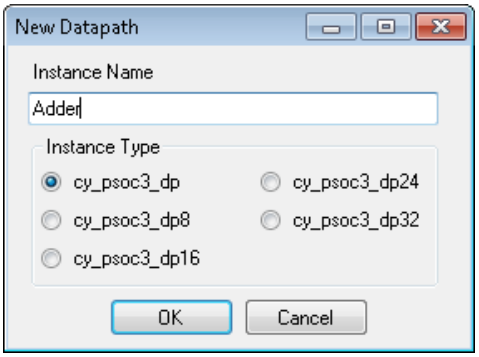
3. 添加一个 *clk* 输入和一个（定义 8 位输入的）Parallel_In[7:0]输入。另外，还定义了 Parallel_Out[7:0]输出，如图 141 所示。

图 141. 将终端添加到并行加法器内



4. 右击符号原理图页面，并添加符号属性：
   - Doc.APIPrefix = *Parallel_Adder*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Parallel_Adder*
   - Doc.DefaultInstanceName = *Parallel_Adder*
5. 添加一个名为 *Add_Value* 的新符号参数，并将其类型设置为 uint8，然后将 Hardware 设置为 True（真）。
6. 右击符号编辑器的空白位置，并为新组件符号生成 Verilog 文件，另外，保留所有默认设置。
7. 点击 Save All，保存所有更改。
8. 启动数据路径配置工具，并打开刚生成的 *Parallel_Adder_v1_0.v* 文件。
9. 创建一个名称为“adder”的新数据路径配置。使用‘cy_psoc3_dp’类型；请参考图 142。

图 142. 数据路径的选择



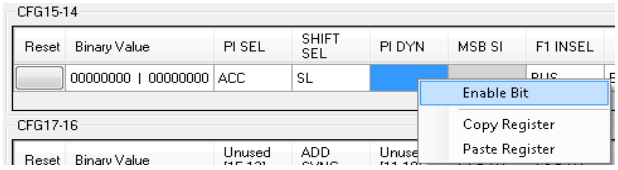
10. 通过该选择，您可以访问数据路径的 PI 和 PO。其他四个 Instance Types 不能访问 PI 和 PO。
11. cy_psoc3_dp 只是一个 8 位的实例。如果您需要的位数多于 8 位，那么需要放置多个实例，然后通过写入 Verilog 代码手动链接它们。附录 D 显示的是这些实例被链路在一起构成 24 位数据路径的一个示例。
12. 配置 CFGRAM 部分，以符合表 13。

表 13. 五个数据路径配置示例

REG	FUNC	SRCA	SRCB	A0 WR SRC	A1 WR SRC	CFB EN
000	PASS	A0	D0	NONE	ALU	ENBL
001	ADD	A1	D0	ALU	NONE	DSBL

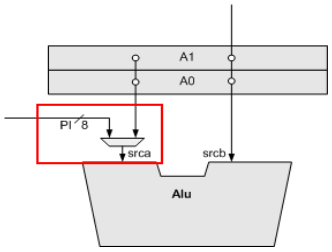
13. 第一种配置（REG 000）采用了 PI 上的值，并将其保存在 A1 内。第二种配置（REG 001）采取了 A1 中的值，并将该值加上 D0 的值，然后将得到的值保存在 A0 内。
14. 接下来，配置 PI 控制。
15. 在 CFG15-14 字段中，右击 PI DYN 下面的字段，然后选择 Enable Bit（使能位），如图 143 所示。

图 143. 使能动态 PI 控制



16. 将 PI DYN 设置为 EN。
17. 这样，可以将 SRCA 输入动态选择为 PI、A0 或 A1；请参考图 144。

图 144. scra 复用器



18. 该复用器的选择通过位 CFB EN 下的 CFGRAM 字段控制。将其设置为 DSBL 时，srca 输入来自 A0 或 A1；将其设置为 ENBL，srca 输入来自 PI。
19. 如果您查看表 13，将发现在第一个配置中，CFB EN 被设置为 ENBL。因此显示 srca 来自 PI。所以在第一个配置中，数据路径将该值从 PI 中传送到 A1 内。
20. 在第二个配置中，CFB EN 被设置为 DSBL。这便表示 scra 输入由 CFGRAM 控制，并被设置为 A1。
21. 将 D0 的初始值设置为 Add_Value。
22. 保存更改内容，并关闭数据路径配置工具。

    接下来，您需要将一些 Verilog 代码添加到组件以实现这些性能。
23. 打开 *Parallel_Adder_v1_0.v* 文件，并寻找下面文本所在的位置：

```
//  Your code goes here
```

24. 使用下面的内容代替上述文本：

```
/* Register to hold state of statemachine*/
reg state;
wire[7:0] po;
/*State loads the value from the PI into A1, latches value in A0 out to PO*/
localparam STATE_LOAD  = 2'b00;
```

```
/*State adds the value in A1 to D0 and stores in A0*/
localparam STATE_ADD  = 2'b01;

/*State machine is always run on positive edge of clock*/

always @( posedge clk )
begin
case (state)
/* Datapath loads in value from PI to A1, the value in A0 is latched to PO*/
          STATE_LOAD:
          begin
                state <= STATE_ADD;

/*we must latch the PO value here, because in the next state PO is not valid*/
                Parallel_Out <= po;
          end

          STATE_ADD:
          begin
                state <= STATE_LOAD;
          end
          endcase
```

25. 为数据路径的逻辑添加下列链接：
    - 将.clk()改为.clk(clk)。
    - 将.cs_addr(3'b0)改为.cs_addr({2'b0, state})。
    - 将.pi()改为.pi(Parallel_In)。
    - 将.p0()改为. p0(po)

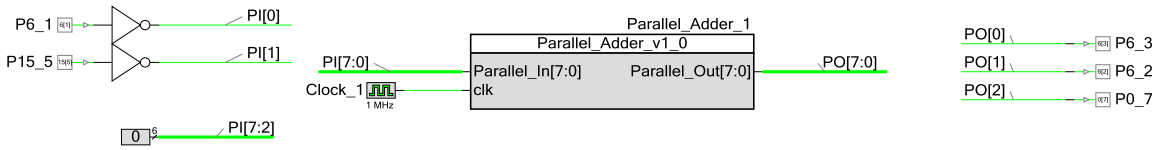
    我们刚刚创建了一个双态状态机，它在两个数据路径配置之间进行切换。第一种配置采取了 PI 上的值，并将其保存在 A1 内。第二种配置采取了 A1 中的值，并将该值加上 D0 的值，然后将得到的值保存在 A0 内。

    PO 始终连接至 A1 或 A0。只有将 srca 选择为 A0 时，该结果才有效，因此我们必须在合适时候注册 PO。下面是需要执行的操作：

    ```
    Parallel_Out <= po;
    ```

    该信号直接采取 PO 的值，并将其保存在信号 Parallel_Out（组件输出）内。在下一个状态内，将 scra 选择为 A1，因此该 PO 无效。这便是它在负载状态中被注册的原因。
26. 完成所有更改后，请保存所有更改的内容。
27. 使 AN82156_Appendix_Lib 成为 PI_PO_Example 项目的附属部分。
28. 将 Parallel_Adder 拖放到您的原理图内。
29. 根据图 145 配置原理图。

图 145. 并行加法器原理图



P6_1 和 P15_5 是数字输入引脚组件，它们被配置为 **Resistive Pull Up**（电阻上拉）。

P6_3、P6_2 和 P0_7 是数字输出引脚组件。

该原理图为 CY8CKIT-030 和 CY8CKIT-050 专门设计。如果您正在使用不同的硬件平台，那么您需要如下更改引脚分布：

表 14. PSoC 4 套件的引脚映射

套件	建议使用的输入引脚	建议使用的输出引脚
CY8CKIT-042	P0[7]、P1[0]	P0[5]、P0[6]、P0[0]
CY8CKIT-043	P0[7]、P1[0]	P1[1]、P1[2]、P1[3]
CY8CKIT-044	P0[7]、P1[0]	P1[1]、P1[2]、P1[3]
CY8CKIT-049-42xx	P0[7]、P1[0]	P0[5]、P0[6]、P0[0]

要想命名某条连线，请执行以下操作（参考图 146）：

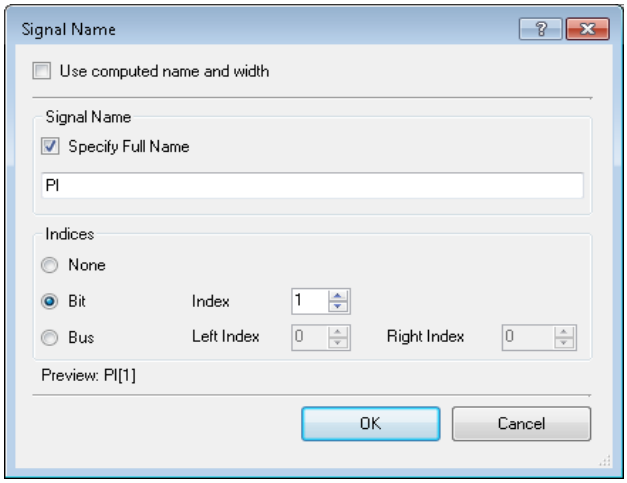
a. 双击该连线。

b. 取消勾选 **Use Computed name and width** 复选框。

c. 勾选 **Specify Full Name** 复选框。

d. 输入该网络（连线）的名称，并选择它是否存在 **Indices**。

图 146. 命名连线对话框



30. 在组件定制器中，将 Add_Value 设置为 2。
31. 保存所有更改内容，并对该项目进行构建和编程。
32. 在 CY8CKIT-050 或 CY8CKIT-030 上，在 P0_7 和 LED2 之间连接一根导线。

    各种按键和 LED 将显示简单加法器运行的情况。LED 4 表示该输出的位 0，LED 3 表示位 1，LED 2 表示位 2。

    SW2 表示该输入的位 0，SW3 表示该输入的位 1。

    如果您正在使用表 14 中所列出的一个套件，那么应该将一个外部按键连接到 P1[0]上，并将外部 LED 连接到该表中所列出的输出引脚上。

    如果没有按下任何按键，则连接到输出位 1 的 LED 3 将发光。这是因为 D0 保存了数值 2。如果您按下了连接到输入位 0 的按键， 那么连接到输出位 0 和输出位 1 的 LED 应该发光以指出数值 3。

    该示例演示了如何使用数据路径的 PI 和 PO。现在您已经知道如何使用这些信号来创建使用 PI 和 PO 更加复杂的设计。

14 附录 B — 数据路径配置工具说明书

本节显示的是数据路径配置工具和基本数据路径硬件间的关系。

GUI 可以分成两个基本部分，即动态配置部分和静态配置部分，如图 147 所示。

- 动态配置 — 您可以设置数据路径设置在各种状态下以不同的方式运行
- 静态配置 — 在各种状态下保持相同的性能

图 147. 数据路径配置工具界面部分

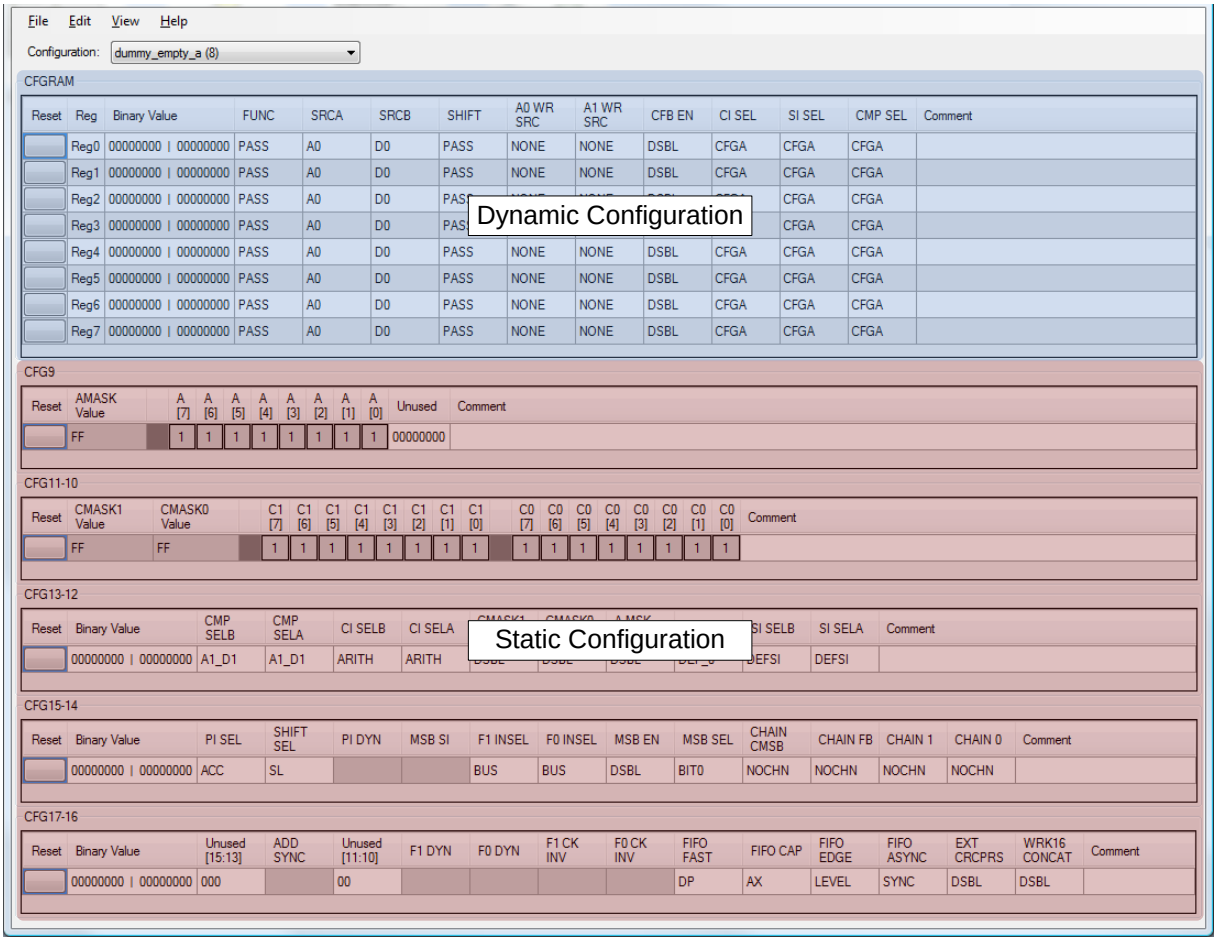
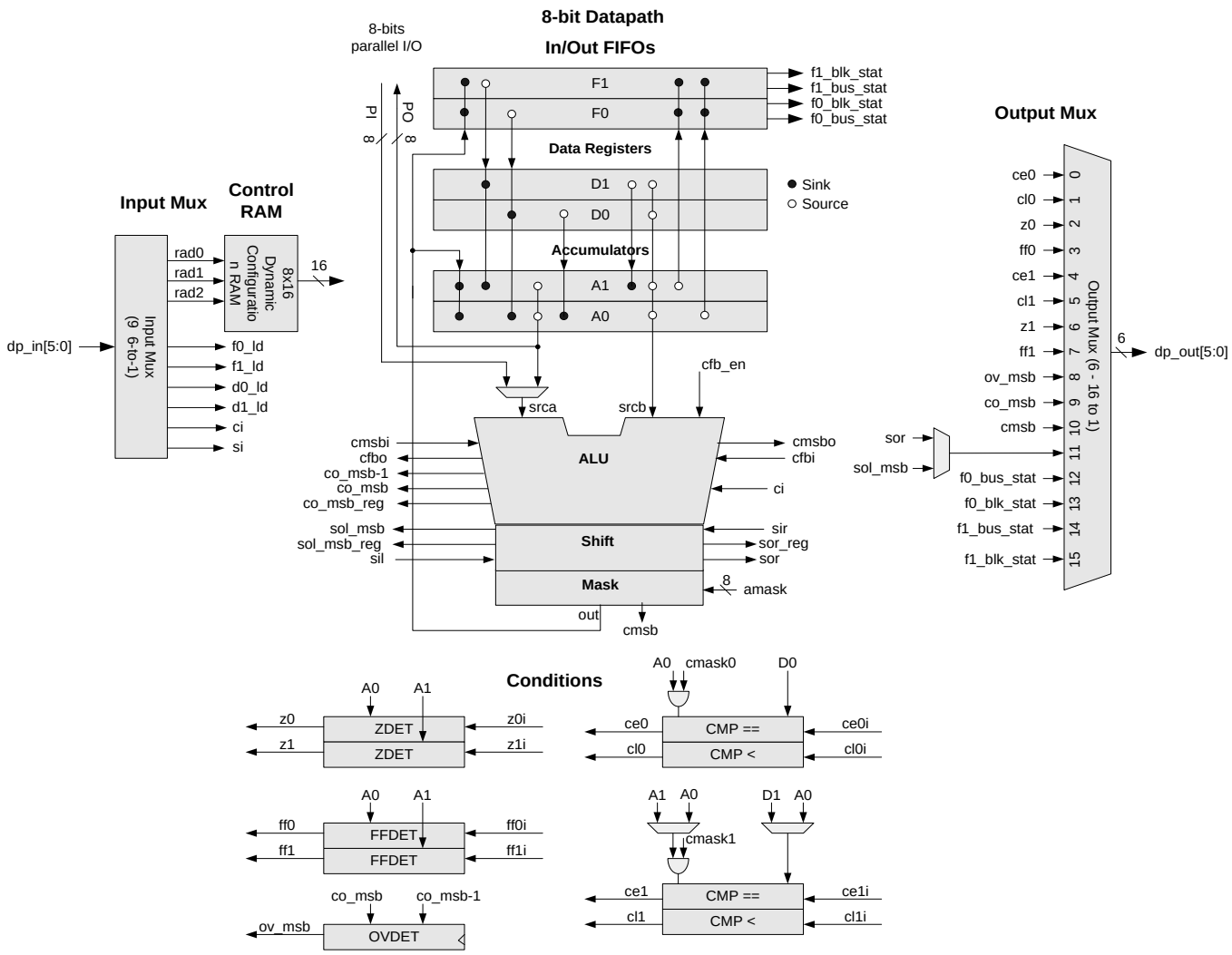


图 148. 数据路径框图


8-bit Datapath
8-bits parallel I/O
In/Out FIFOs
F1
F0
f1_blk_stat
f1_bus_stat
f0_blk_stat
f0_bus_stat
PI 8
PO 8
Data Registers
D1
D0
Sink
Source
Accumulators
A1
A0
Input Mux
Control RAM
rad0
rad1
rad2
8x16 Dynamic Configuration RAM
16
dp_in[5:0]
Input Mux (9 6-to-1)
f0_ld
f1_ld
d0_ld
d1_ld
ci
si
cfb_en
srca
srcb
cmsbi
cfbo
co_msb-1
co_msb
co_msb_reg
ALU
cmsbo
cfbi
ci
sol_msb
sol_msb_reg
sil
Shift
sir
sor_reg
sor
Mask
8
amask
out
cmsb
Output Mux
ce0
cl0
z0
ff0
ce1
cl1
z1
ff1
ov_msb
co_msb
cmsb
sor
sol_msb
f0_bus_stat
f0_blk_stat
f1_bus_stat
f1_blk_stat
Output Mux (6 - 16 to 1)
6
dp_out[5:0]
Conditions
A0
A1
z0
z1
ZDET
ZDET
z0i
z1i
A0
cmask0
D0
ce0
cl0
CMP ==
CMP <
ce0i
cl0i
A0
A1
ff0
ff1
FFDET
FFDET
ff0i
ff1i
A1
A0
D1
A0
cmask1
ce1
cl1
CMP ==
CMP <
ce1i
cl1i
co_msb
co_msb-1
ov_msb
OVDET

14.1 动态配置 RAM（CFGRAM）部分

动态配置部分表述了动态配置 RAM。它配置了八个配置/指令的数据路径性能。下表详细介绍了 GUI 中各字段的函数。

表 15. 数据路径配置工具的 CFGRAM 部分

动态配置 RAM 部分
数据路径配置工具

CFGRAM

Reset	Reg	Binary Value	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC	A1 WR SRC	CFB EN	CI SEL	SI SEL	CMP SEL
	Reg0	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg1	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg2	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg3	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg4	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg5	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg6	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg7	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA

数据路径框图

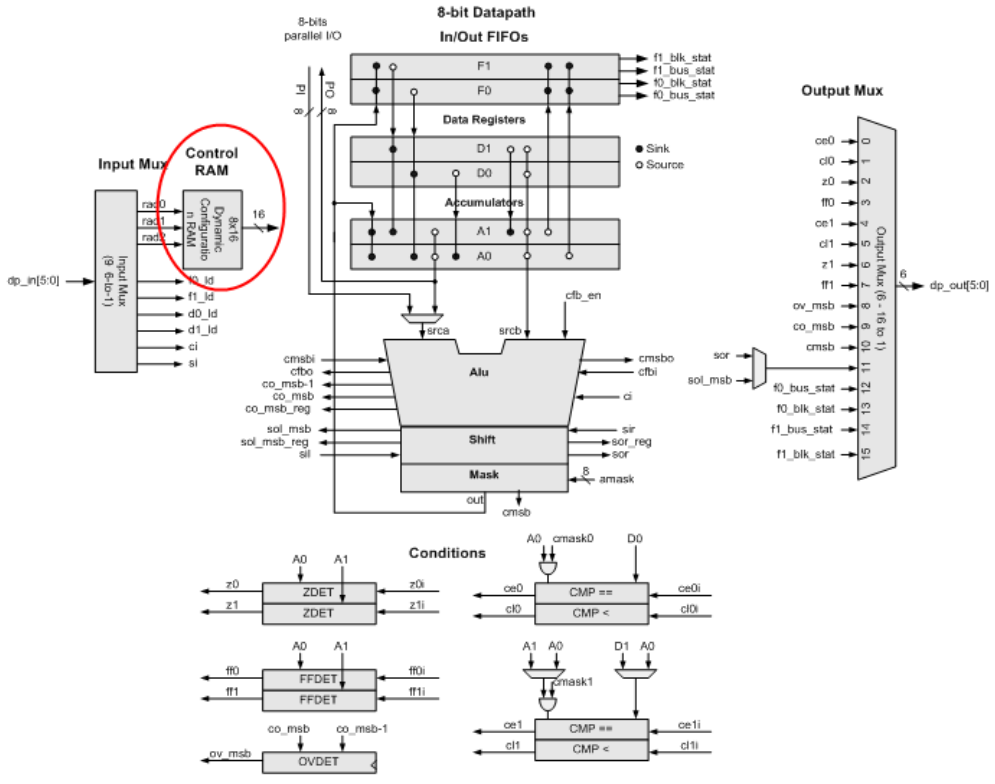


表 16. 动态配置部分的列说明

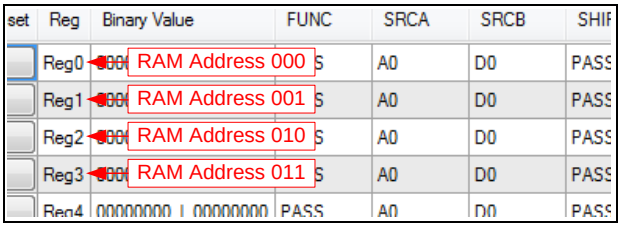
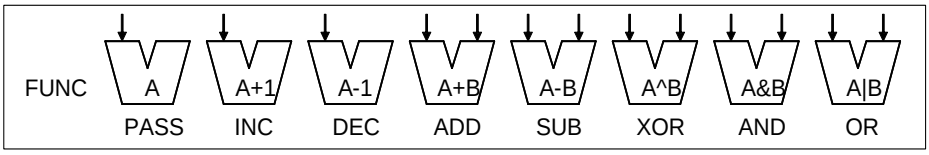
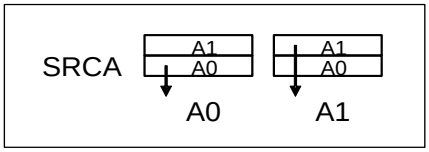
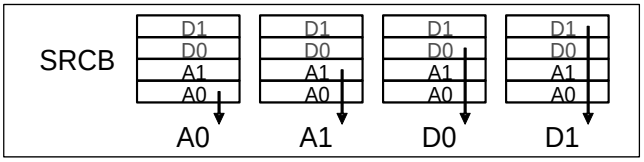
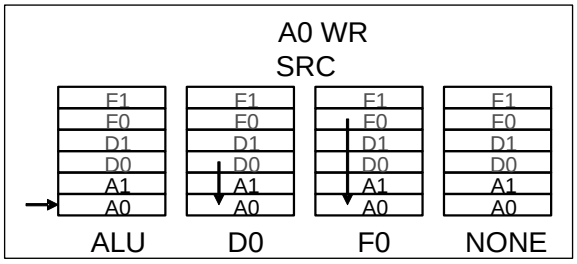
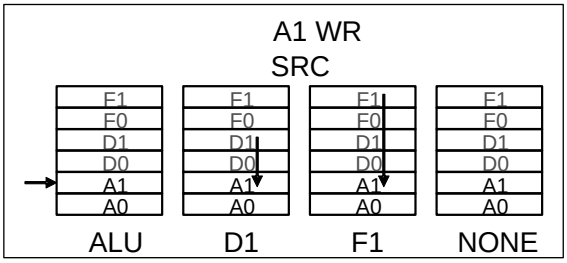
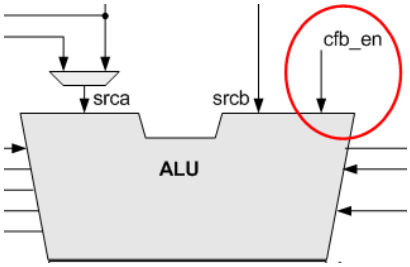
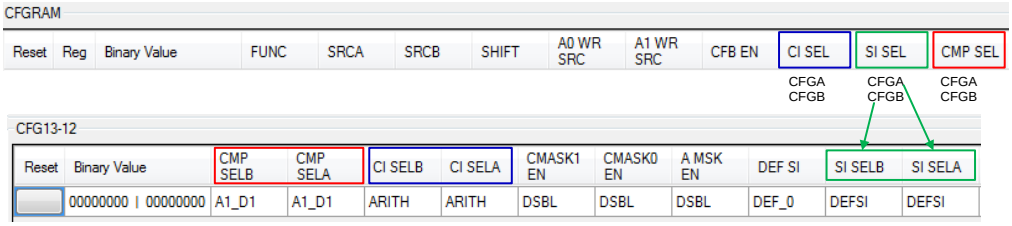
动态配置部分	
寄存器 本列显示了与每一行相对应的配置/指令信息。通过三个 CFGRAM 地址信号的值可确定要选择的配置，因此需要确保为每个配置选择正确的行。	
FUNC 本列确定在上面配置中执行八个 ALU 函数中的函数。	
SRCA 本列确定 ALU 的‘srca’输入源。 srca 也可以来自 PI — 请参考表 21（CFG 15-14）。	
SRCB 本列确定 ALU 的‘srcb’输入源。	
SHIFT 本列确定移位模块的函数。	
A0 WR SRC 本列确定完成 ALU 操作*后* A0 寄存器中的内容。	

表 2. 动态配置部分的列说明

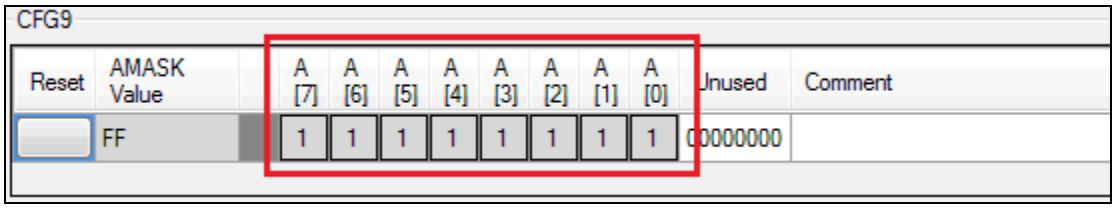
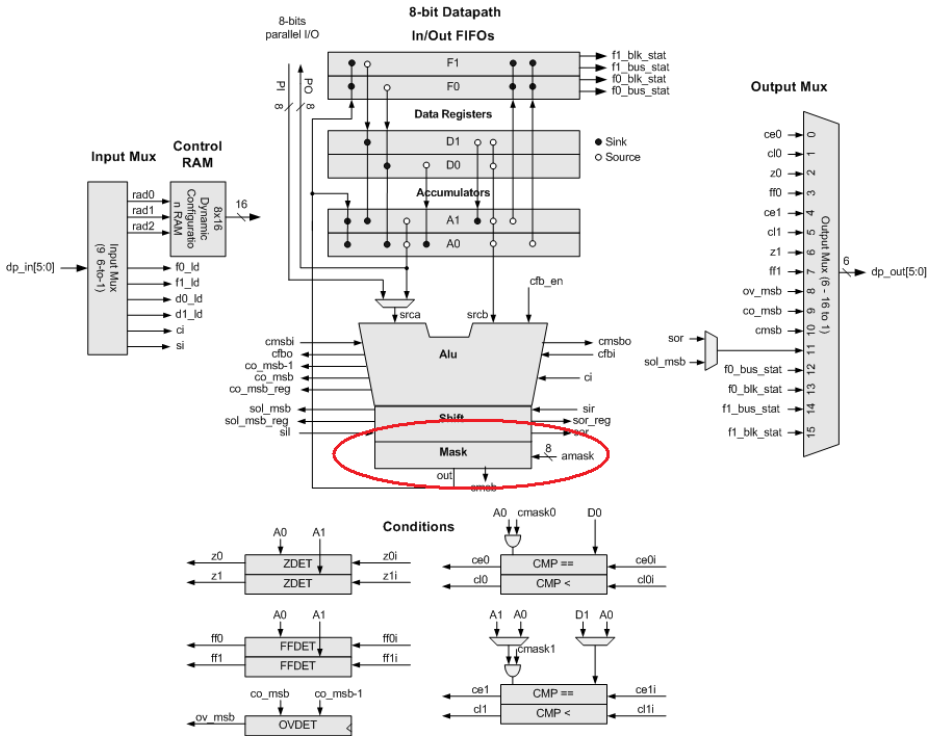
动态配置部分	
A1 WR SRC 本列确定完成 ALU 操作后 A1 寄存器中的内容。	
CFB EN CRC 配置使能 — 请参考 CFG15 和 CFG14 寄存器中的 PI DYN。	
CI SEL、SI SEL、CMP SEL 进位选择（Carry in Select）、移入选择（Shift In Select）以及比较选择（Compare Select）。每项都有两个配置选项：A 和 B，您可根据要求选择所需的配置选项。 更多有关信息，请参考表 20（CFG13-12）。	

14.2 静态配置部分

静态配置部分演示了数据路径寄存器 CFG9 到 CFG17，如表 17、表 18、表 19、表 20 和表 21 所示。这些寄存器控制着静态函数，包括移位方向、屏蔽、FIFO 配置以及链接。

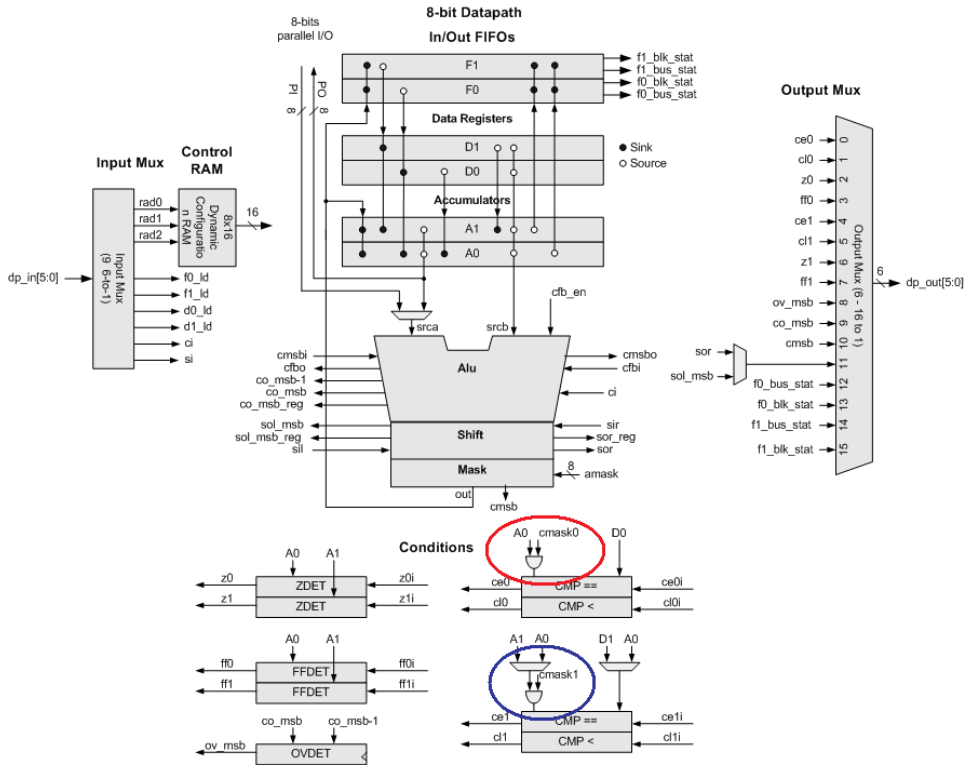
14.2.1 CFG9 寄存器

表 17. CFG9 寄存器定义

AMASK 值
数据路径配置工具

■ AMASK 值 — 该字段包含了应用于数据路径 ALU 模块输出的 8 位屏蔽值。对移位寄存器中的输出和该寄存器中的内容进行 AND 运算。默认情况下，该特性被禁用。要想使能它，必须设置 AMASK EN 的位 CFG12。
数据路径框图


14.2.2 CFG11 和 CFG10 寄存器

表 18. CFG11 和 CFG10 寄存器定义

CMASK0 值和 CMASK1 值
数据路径配置工具
CFG11-10 Reset / CMASK1 Value: FF / CMASK0 Value: FF C1 [7] C1 [6] C1 [5] C1 [4] C1 [3] C1 [2] C1 [1] C1 [0]: 1 1 1 1 1 1 1 1 C0 [7] C0 [6] C0 [5] C0 [4] C0 [3] C0 [2] C0 [1] C0 [0]: 1 1 1 1 1 1 1 1 Comment
■ CMASK0 和 CMASK1 — 这些字段将设置用于比较器模块输入的屏蔽值。进行比较前，先要将 A0 或 A1 寄存器的内容和这些寄存器的内容进行 AND 运算。要想使能它们，必须在 CFG12 中设置 CMASK0 EN 和 CMASK1 EN 位（表 19）。
数据路径框图


14.2.3 CFG13 和 CFG12 寄存器

表 19. CFG13 和 CFG12 寄存器定义

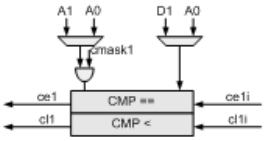
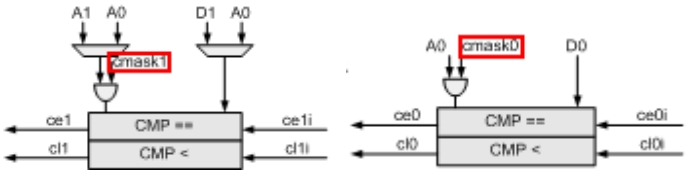
CFG13-12 的配置选项

数据路径配置工具

CFG13-12

Reset	Binary Value	CMP SELB	CMP SELA	CI SELB	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA
	00000000 | 00000000	A1_D1	A1_D1	ARITH	ARITH	DSBL	DSBL	DSBL	DEF_0	DEFSI	DEFSI

CFG13-12 的详细信息

CMP SELB & CMP SEL A 为比较模块 1 配置比较 B 和比较 A 选项。RAM 配置的 CMP SEL 字段确定在已给的周期内 A 还是 B 选项有效。 **注意：**比较模块 0 只能对 D0 和 A0 中的屏蔽值进行比较。	**A1_D1**：A1 < D1、A1 == D1 **A1_A0**：A1 < A0、A1 == A0 **A0_D1**：A0 < D1、A0 == D1 **A0_A0**：A0 < A0、A0 == A0 
CI SELB & CI SEL A 选择 carry in 的源。	**ARITH**：移动（carry）方向由 ALU 算术控制。 **REGIS**：移入（carry in）是上一周期寄存的移出（carry out）。 **ROUTE**：从数据路径输入中选择一个移入（Carry in）。 **CHAIN**：移入（Carry in）由链中前一个数据路径驱动。
CMASK0 EN、CMASK1 EN、AMASK EN 使能比较 1 和比较 0 模块（如右侧所示）上的屏蔽以及 ALU（不显示）输出上的模块的屏蔽。请参考表 17 和表 18。	
DEF SI 该选项定义默认移位的值是 0 还是 1。注意：必须将 SI SELB 或 SI SELA 设为 DEFSI。	**DEF_0**：0 **DEF_1:** 1
SI SELB & SI SEL A 选择 A 和 B 移入配置的 shift in 源。	**DEFSI**：Shifts in 可以是 0，或者是 1，如 DEF SI 所定义的。 **REGIS**：Shift in 是上一周期寄存的 shift out。 **ROUTE**：Shift in 是从数据路径输入中选择的。 **CHAIN**：Shift in 由链中前一个数据路径驱动。

14.2.4 CFG15 和 CFG14 寄存器

表 20. CFG15 和 CFG14 寄存器定义

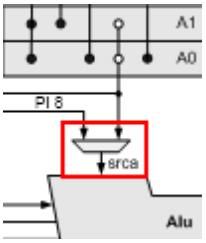
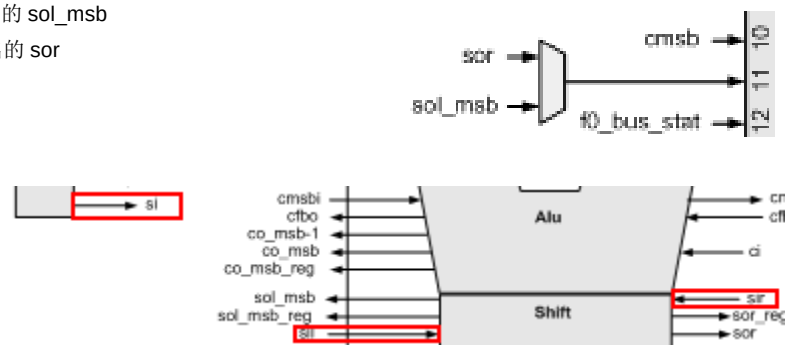
CFG15-14 寄存器选择

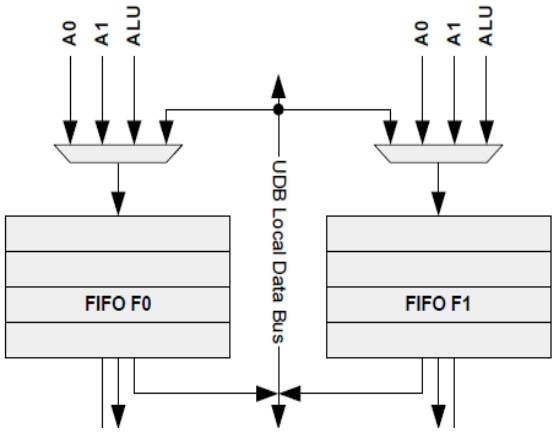
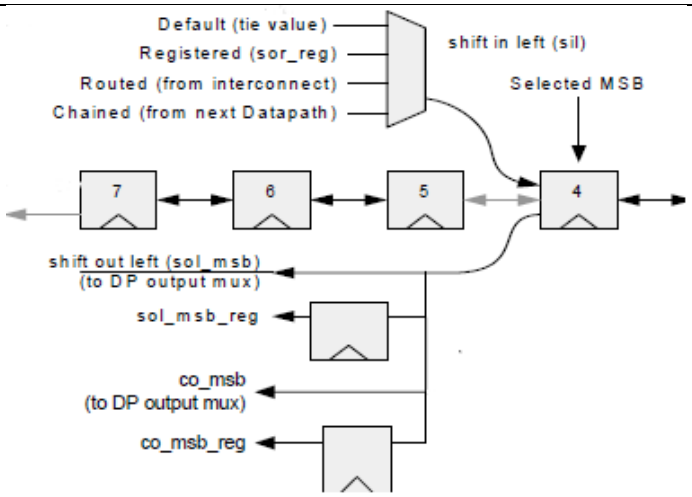
数据路径配置工具

CFG15-14

Reset	Binary Value	PI SEL	SHIFT SEL	PI DYN	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0
	00000000 | 00000000	ACC	SL			BUS	BUS	DSBL	BIT0	NOCHN	NOCHN	NOCHN	NOCHN

CFG15-14 的详细信息

PI SEL 该选项将控制 SRCA 的复用器输入。此输入可以源自动态配置选项中的 SCRA 选项，也可以源自并行输入。	**ACC：**通过在动态配置区域中进行选择可以选择源 A 的来源。 **PIN：**源 A 的来源由数据路径的并行输入选择。 
Shift SEL 选择数据路径输出复用器的 shift out 信号。	**SL：**选为输出的 sol_msb **SR：**选为输出的 sor 
PI DYN 使能并行输入（PI）复用器选择的动态控制，从而将信号传输到 ALU ASRC 输入。	**DS：**PI 复用器由该寄存器中的 PI SEL 位静态控制。 **EN：**PI 复用器由动态 RAM 中的 CFB_EN 位动态控制（假设 PI SEL 为‘0’）。当置位该位并且 CFB_EN 为‘0’时，ALU ASRC 输入将为 A0 或 A1；当 CFB_EN 为‘1’时，ALU ASRC 输入为 PI 路由得到。
MSB SI 支持算术 Shift Right（右移）操作。	**REG：**默认的 shift in 选择是由 DEF SI 值确定的（CFG12，表 19）。 **MSB：**使用当前定义的 MSB （CFG14 中的 MSB EN 和 MSB SEL 字段）覆盖掉默认的 shift in 值（在 SELA[1:0]/SELB[1:0]中，该值被定义为‘00’）。

CFG15-14 的详细内容（续表）	
F1 INSEL & F0 INSEL 定义了 FIFO 1 和 FIFO 0 的输入源。	**Bus:** FIFO 输入是 CPU 总线，FIFO 输出是 A0 或者 D0 寄存器 **A0：** FIFO 输入是 A0。FIFO 输出是 CPU 总线。 **A1：** FIFO 输入是 A1。FIFO 输出是 CPU 总线。 **ALU：** FIFO 输入是 ALU。FIFO 输出是 CPU 总线。 
MSB EN 和 MSB SEL 您可以调整 ALU 输出的 MSbit。通过这两个设置您可以禁用和使能该特性，并选择作为 MSbit 的位。	当 MSbit 更改为除位 7 以外的任何位，shift in、shift out 和 carry out 输出将相应地改变。 
链路 CMSB 使能将 CRC MSB 从链路中的下一个数据路径链接至该模块。	CRC MSB 信号流是从 MS 模块到 LS 模块的。当数据路径不包含 CRC 计算的最高有效位时，请置位该位。 **NOCHN：** CRC MSB 不被链接。 **CHNED：** 从该链路中的下个数据路径模块链接 CRC MSB。
链路 FB 使能 CRC 反馈从链路中的前一个数据路径连接至该模块。	CRC FB 信号流是从 LS 模块到 MS 模块的。当数据路径不包含 CRC 计算的最低有效位时，请置位该位。 **NOCHN：** CRC 反馈不被链接。 **CHNED：** 从链路中的前一个数据路径链接 CRC。
链路 1 和链路 0 定义 CL0、CL1、CE0、CE1、Z0、Z1、FF0 以及 FF1 的输出是否是相连的。	使用链接（CHNED）方式时，前一个数据路径的条件将被链接到此数据路径。链路 0 将影响到 CL0、CE0、Z0 以及 FF0 条件。链路 1 将影响 CL1、CE1、Z1 以及 FF1 条件。

14.2.5 CFG17 和 CFG16 寄存器

表 21. CFG17 和 CFG16 寄存器定义

FIFO 配置
数据路径配置工具

CFG17-16

Reset	Binary Value	Unused [15:13]	ADD SYNC	Unused [11:10]	F1 DYN	F0 DYN	F1 CK INV	F0 CK INV	FIFO FAST	FIFO CAP	FIFO EDGE	FIFO ASYNC	EXT CRCPRS	WRK16 CONCAT
	00010000 | 00000000	000	ADD	00					DP	AX	LEVEL	SYNC	DSBL	DSBL

- ADD SYNC — 确定是否向 FIFO 模块状态中添加额外的同步触发器。该字段控制着以总线时钟分辨率进行总线读/写操作时和在布线数据路径输出时确认新状态之间的周期时序。只有一个配置位可以控制两个 FIFO 控制该时序。请参考下一页的 FIFO 配置章节。

  NONE：不会将触发器添加到 FIFO 模块状态的输出端。
  ADD：将触发器添加到 FIFO 模块状态的输出端。

- F1 DYN — 控制 FIFO1 方向是静态的还是动态的。在静态模式下，F1_SEL[1:0]位控制着 FIFO 的读和写访问。将该位设置为动态模式时，会存在两种配置：内部访问（数据路径对 FIFO 进行读写操作）和外部访问（系统总线对 FIFO 进行读写操作）。在该模式下，F1_SEL[1:0]位控制着内部访问模式下的 FIFO 写操作源。

  OFF：静态模式。FIFO 方向为静态的，并由 F1_SEL[1:0]控制。
  ON：动态模式。FIFO 方向为动态的，通过切换 DP 布线信号 d1_load，可以控制在内部和外部访问间切换。

- F0 DYN — F1 DYN 的读取说明

- F0 CLK INV 和 F1 CLK INV — 确定 FIFO 时钟的反转是否对应数据路径时钟。

  NEG：FIFO 时钟和 DP 时钟的极性是相同的
  POS：FIFO 时钟的反转对应着 DP 时钟的反转

- FIFO FAST — 确定 FIFO 的时钟源为数据路径时钟还是 PSoC 总线时钟。在快速模式下，FIFO 的时钟源为捕获延迟得到降低的总线时钟。如果使用该模式，功耗会增加。这是因为需要使能主设备和总线时钟的象限门控。该位控制着 UDB 中的两个 FIFO 模式，但它只适合于被配置为输出模式的 FIFO。

  DP：数据路径时钟
  BUS：总线时钟

- FIFO CAP — 使能 FIFO 捕获模式。被使能时，对 A0 或 A1 的读取操作将分别被写入到 F0 或 F1 内。

  AX：读取 A0 或 A1 时将直接返回寄存器中的值。
  FX：读取 A0（或 A1）将触发将该结果写入到 F0（或 F1）内的操作。

- FIFO EDGE — 确定从低电平转换为高电平时是否对 FIFO 进行写入操作。或者，如果随时发生对 FIFO 进行写操作，F0 或 F1 的负载信号即为高电平。

  LEVEL：FIFO 写入（输出模式）是电平敏感的。
  EDGE：FIFO 写入（输出模式）是边缘敏感的。

- FIFO ASYNC — 确定 FIFO 模块状态信号的输出是否需要触发器。请参考下一页上的 FIFO 配置章节。

  SYNC：不将触发器添加到 FIFO 模块状态的输出。
  ASYNC：将触发器添加到 FIFO 模块状态的输出。只有数据路径时钟与 MASTER_CLK 同步时，才能将 ADD SYNC 设置为 *NONE* 以及将 FIFO ASYNC 设置为 *ASYNC*，或将 ADD SYNC 设置为 *ADD* 以及将 FIFO ASYNC 设置为 *SYNC* 。

- EXT CRCPRS — 覆盖掉 CRC/PRS 计算的内部配置，并允许对 CRC/PRS 信号进行外部路由。置位该位时，可以访问 CRC 操作的原始模块输入（包括 shift in 数据和反馈数据），并且必须在外部计算这些信号。（通常在 PLD 内）。

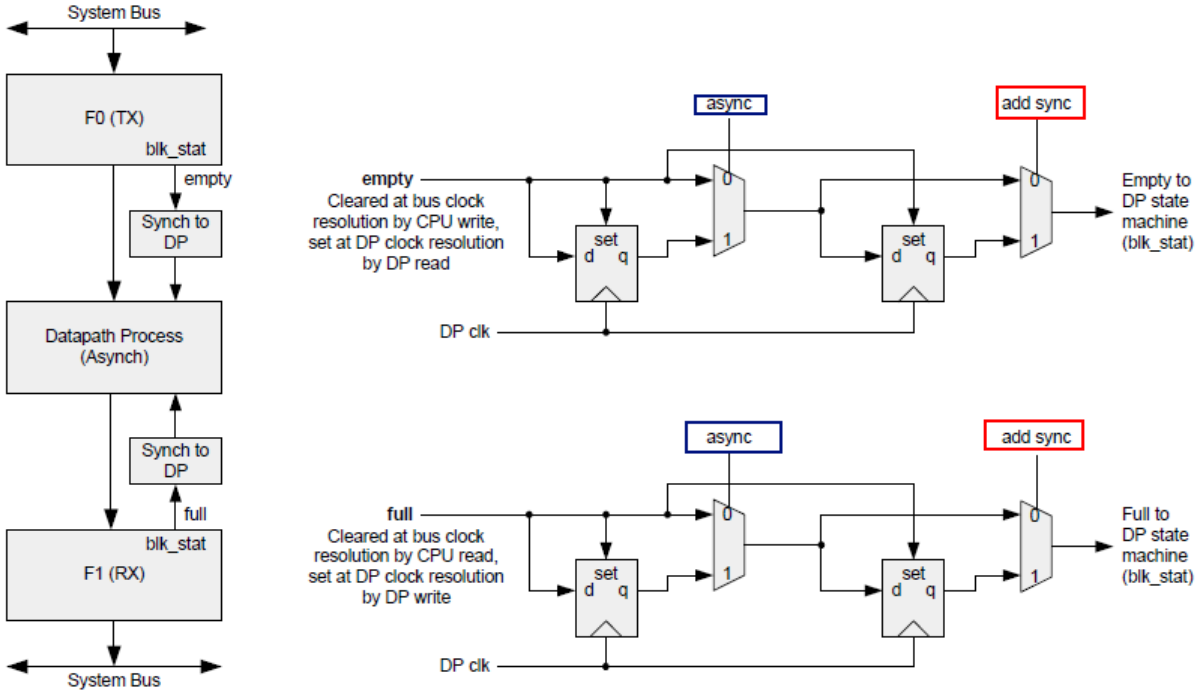
  DSBL：内部 CRC/PRS 布线
  ENBL：外部 CRC/PRS 布线。

- WRK16 CONCAT — 控制着 16 位访问空间内的工作寄存器访问模式。默认情况下，当该位为'0'时，访问将在 UDB 对的同一个寄存器上按链接顺序发生。将该位置 '1' 时，将对单个 UDB 中的串联寄存器进行 16 位的读或写访问。各种组合分别为{A1,A0}、{D1,D0}、F1,F0}、{CTL,STAT}、{MSK, ACTL}、{8'b0,MC}。

  DSBL：16 位默认访问模式。16 位访问将按照链接/地址顺序对两个连续的 UDB 中已给寄存器进行读写操作。
  ENBL：16 位串联访问模式。16 位访问将对单个 UDB 中的串联寄存器进行读写操作

FIFO 配置

FIFO 输出同步框图

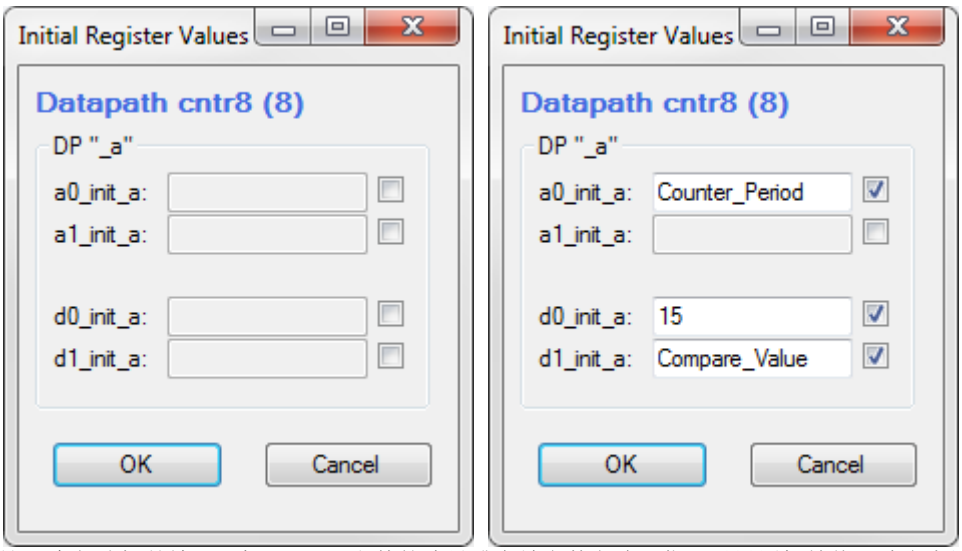


ASYNC	ADD SYNC	操作	使用模型
0	0	与总线时钟同步	改变 CPU 读/写状态是以总线时钟分辨率进行的。如果满足了总线时钟时序，那么可以用于最小的延迟。
0	1	从总线时钟上重新采样得到 DP 时钟	这是默认的同步操作模式。当使用当前选择的 DP 时钟同步进行 CPU 读/写状态更改的重新采样时，将为 UDB 逻辑提供 DP 时钟设置时间的完整周期。
1	0	（冗余）	
1	1	从总线时钟双同步到 DP 时钟	当使用一个自由运行的异步 DP 时钟时，通过该设置可以将 CPU 读和写操作双同步到 DP 时钟。

14.3 设置初始寄存器值

要想设置 DCT 中的 A0、A1、D0 和 D1 的初始值，请依次选择 **View > Initial Register Values**。例如，如果数据路径名称为 Cntr8，那么该窗口将如图 149（左边）所示。

图 149. Initial Register Values（初始寄存器值）



通过点击这两个复选框并输入目标 Verilog 文件的编号或有效参数名称，您可以设置初始值（请参考图 149 右侧中显示的内容）。

14.4 数据路径链接

通过这些专用的数据路径链接信号，可以有效地实现单周期 16、24 和 32 位功能，而不需要使用通道布线资源。

如图 150 所示，所有生成的条件和捕获信号链接的方向都是从最低有效模块到最高有效模块。左移是从最低有效字节链接到最高有效字节。右移是从最高有效字节链接到最低有效字节。CFBO（反馈）的 CRC/PRS 链接信号从最低有效模块链接到最高有效模块，而 CMSBO （MSB 输出）从最高有效模块链接到最低有效模块。

图 150. 数据路径的链接信号流

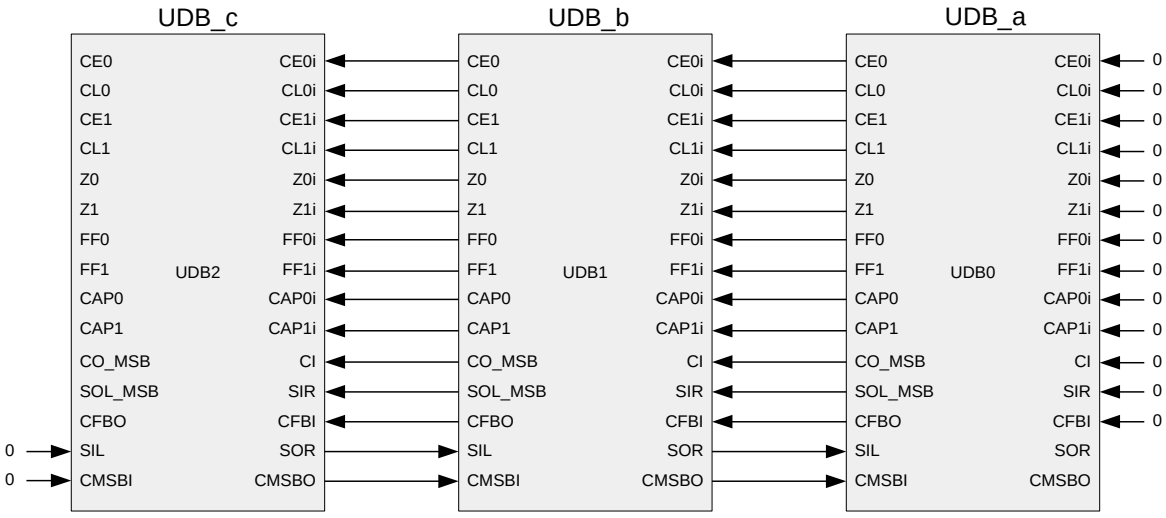
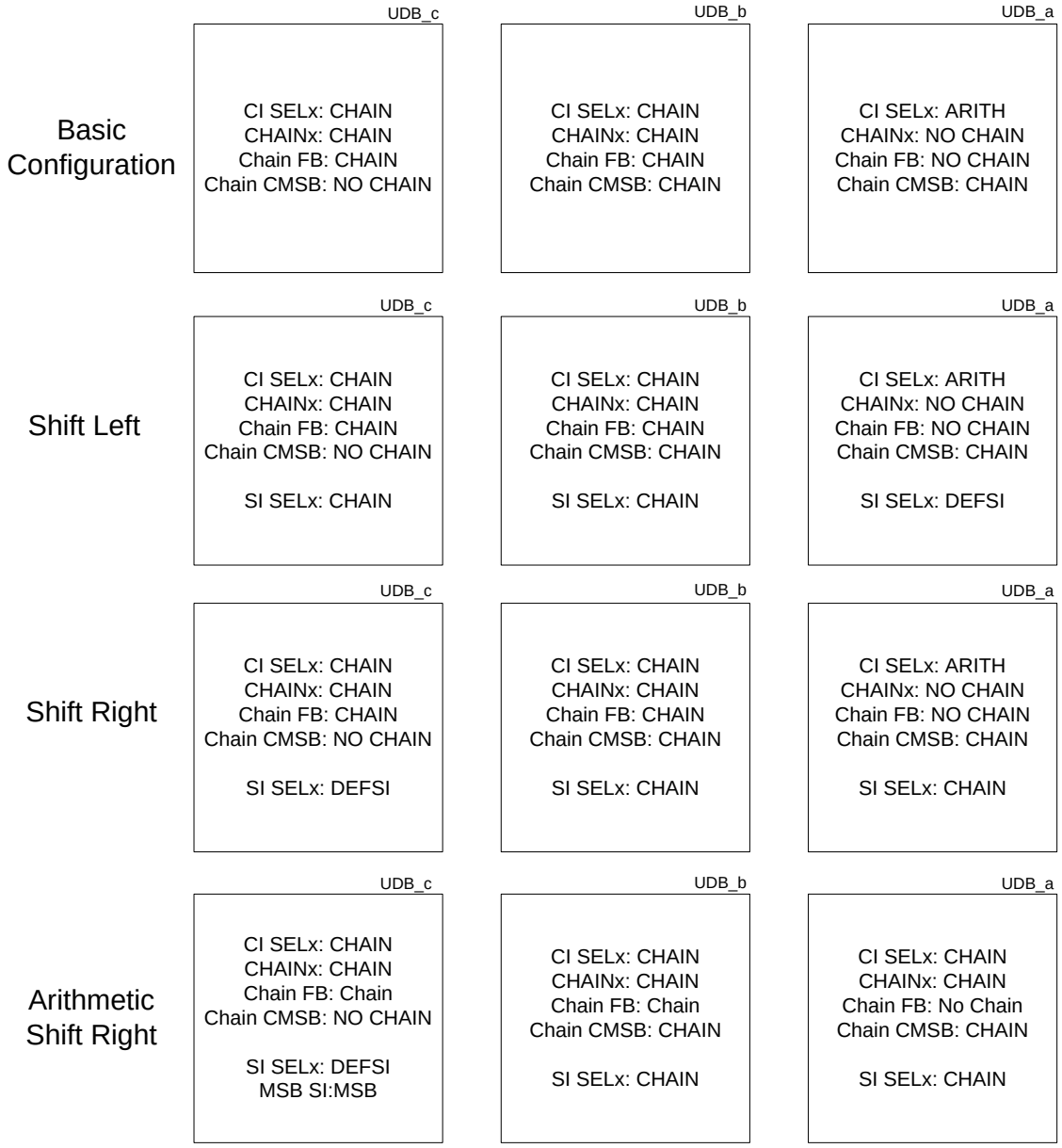


图 151 显示的是链接数据路径的设置情况。UDB_a 是最低有效模块，UDB_c 是最高有效模块。图 151 显示的是通过清除或重复中间数据路径配置，可以创建 3-UDB（多达 24 位）功能、16 位或 32 位功能。该图也显示了链路 FB 和链路 CMSB 的配置情况（即使它们未被使用）。

请记住图 150，将各个数据路径链接在一起时，多个设计（例如，简单的加法或减法）将使用图 151 中的‘基本配置’，因此要将所有信号从 LSB （UDB_a）链接至 MSB （UDB_c）（链接 CMSB 除外）。如果您执行了移位操作，那么需要根据移位方向来更改移位链接配置（图 151 中显示的 Shift-Left（左移）、Shift-Right（右移）和 Arithmetic Shift-Right（算术右移）。

图 151. 链接的 DCT 配置



14.5 数据路径寄存器的固件控制

通过使用宏 CY_SET_REG8（地址、值）和 CY_GET_REG8（地址）或这些函数的相应 16、24 或 32 位版本，可以在固件内访问数据路径寄存器。

这些寄存器的地址提供在 *cyfitter.h* 文件中（成功构建后会生成该文件）。例如，如果一个名为“cntr8”的 8 位数据路径实例在一个名为“SimpleCntr8_1”的组件中得到实例化，那么 *cyfitter.h* 文件将包含一个列出了所有数据路径寄存器地址的代码模块，请参考图 152。

图 152. 数据路径寄存器的地址

```
/* SimpleCntr8_1 */
#define SimpleCntr8_1_cntr8_u0__16BIT_A0_REG CYREG_B1_UDB05_06_A0
#define SimpleCntr8_1_cntr8_u0__16BIT_A1_REG CYREG_B1_UDB05_06_A1
#define SimpleCntr8_1_cntr8_u0__16BIT_D0_REG CYREG_B1_UDB05_06_D0
#define SimpleCntr8_1_cntr8_u0__16BIT_D1_REG CYREG_B1_UDB05_06_D1
#define SimpleCntr8_1_cntr8_u0__16BIT_DP_AUX_CTL_REG CYREG_B1_UDB05_06_ACTL
#define SimpleCntr8_1_cntr8_u0__16BIT_F0_REG CYREG_B1_UDB05_06_F0
#define SimpleCntr8_1_cntr8_u0__16BIT_F1_REG CYREG_B1_UDB05_06_F1
#define SimpleCntr8_1_cntr8_u0__A0_A1_REG CYREG_B1_UDB05_A0_A1
#define SimpleCntr8_1_cntr8_u0__A0_REG CYREG_B1_UDB05_A0
#define SimpleCntr8_1_cntr8_u0__A1_REG CYREG_B1_UDB05_A1
#define SimpleCntr8_1_cntr8_u0__D0_D1_REG CYREG_B1_UDB05_D0_D1
#define SimpleCntr8_1_cntr8_u0__D0_REG CYREG_B1_UDB05_D0
#define SimpleCntr8_1_cntr8_u0__D1_REG CYREG_B1_UDB05_D1
#define SimpleCntr8_1_cntr8_u0__DP_AUX_CTL_REG CYREG_B1_UDB05_ACTL
#define SimpleCntr8_1_cntr8_u0__F0_F1_REG CYREG_B1_UDB05_F0_F1
#define SimpleCntr8_1_cntr8_u0__F0_REG CYREG_B1_UDB05_F0
#define SimpleCntr8_1_cntr8_u0__F1_REG CYREG_B1_UDB05_F1
```

14.6 其他信息

更多有关数据路径配置工具的信息，请在依次点击 **Help** > **Documentation** 后显示的 DCT 或依次点击 **Help** > **Documentation** > **Component Author Guide** 后显示的 PSoC Creator 中查看组件创建指南的附录 B。

15 附录 C — 自动生成的 Verilog 代码

本附录提供了 PSoC Creator 和数据路径配置工具生成的 Verilog 代码示例。这里显示的代码是从示例项目#1 的组件创建过程中复制而来的。与本应用笔记相关联的 PSoC Creator 项目包含了完整的示例项目和注释。这里显示的代码演示了使用 PSoC Creator 和数据路径配置工具时 Verilog 代码的演变。

15.1 PSoC Creator 生成的新 Verilog 文件

当 PSoC Creator 首次生成 Verilog 文件时，它非常像下面的形式：

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 09/17/2012 at 11:02
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

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

//        Your code goes here

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

在这种情况下，由于‘tc’和‘clk’引脚显示在 SimpleCntr8 组件符号中，因此将它们添加到了文件中。如果组件符号中不包括任何终端，则要省略这些代码行。

15.2 带有新数据路径实例的 Verilog 文件

当数据路径配置工具为数据路径配置生成代码时，该代码将被附加到现有的组件 Verilog 文件内。一个未经修改的配置类似于以下内容：

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 09/17/2012 at 11:02
```

```
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

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

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
cy_psoc3_dp8 #(.cy_dpconfig_a(
{
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM1: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7: */
    8'hFF, 8'h00,  /*CFG9: */
    8'hFF, 8'hFF,  /*CFG11-10: */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12: */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
    1'h0, `SC_FIFO1_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /*CFG15-14: */
    10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16: */
}
)) cntr8(
```

```
            /*  input                   */  .reset(1'b0),
            /*  input                   */  .clk(1'b0),
            /*  input   [02:00]         */  .cs_addr(3'b0),
            /*  input                   */  .route_si(1'b0),
            /*  input                   */  .route_ci(1'b0),
            /*  input                   */  .f0_load(1'b0),
            /*  input                   */  .f1_load(1'b0),
            /*  input                   */  .d0_load(1'b0),
            /*  input                   */  .d1_load(1'b0),
            /*  output                  */  .ce0(),
            /*  output                  */  .cl0(),
            /*  output                  */  .z0(),
            /*  output                  */  .ff0(),
            /*  output                  */  .ce1(),
            /*  output                  */  .cl1(),
            /*  output                  */  .z1(),
            /*  output                  */  .ff1(),
            /*  output                  */  .ov_msb(),
            /*  output                  */  .co_msb(),
            /*  output                  */  .cmsb(),
            /*  output                  */  .so(),
            /*  output                  */  .f0_bus_stat(),
            /*  output                  */  .f0_blk_stat(),
            /*  output                  */  .f1_bus_stat(),
            /*  output                  */  .f1_blk_stat()
);
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

请注意，虽然已添加了配置 RAM 和硬件连接的部分，但它们不包括任何实际配置数据。因此该数据路径组件是无用的。表 22 介绍的是数据路径配置中的硬件连接。

表 22. Verilog 硬件连接的定义

Verilog 代码	定义/映射
.reset(1'b0),	数据路径的复位信号
.clk(1'b0),	数据路径的时钟输入
.cs_addr(3'b0),	配置存储地址信号
.route_si(1'b0),	布线的 Shift In
.route_ci(1'b0),	布线的 Carry In
.f0_load(1'b0),	FIFO 0 负载信号
.f1_load(1'b0),	FIFO 1 负载信号
.d0_load(1'b0),	D0 负载信号
.d1_load(1'b0),	D1 负载信号
.ce0(),	比较模块 0“等于”输出
.cl0(),	比较模块 0“小于”输出
.z0(),	零检测模块 0 的输出
.ff0(),	全 1 检测模块 0 的输出
.ce1(),	比较模块 1“等于”输出
.cl1(),	比较模块 1“小于”输出

Verilog 代码	定义/映射
.z1(),	零检测模块 1 的输出
.ff1(),	全 1 检测模块 1 的输出
.ov_msb(),	溢出检测
.co_msb(),	Carry Out
.cmsb(),	没有包含在本应用笔记中
.so(),	Shift Out
.f0_bus_stat(),	FIFO 0 总线状态标志*
.f0_blk_stat(),	FIFO 0 模块状态标志*
.f1_bus_stat(),	FIFO 1 总线状态标志*
.f1_blk_stat()	FIFO 1 模块状态标志*

*更多有关这些状态输出的信息，请参考技术参考手册

这些链接是数据路径可以工作的前提。没有它们，组件符号、Verilog 代码和数据路径硬件之间没有什么关联。

15.3 进行 SimpleCntr8 修改后的 Verilog 文件

这是简单 8 位计数器的完整 Verilog 文件：

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 09/17/2012 at 11:02
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

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

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
cy_psoc3_dp8 #(.a0_init_a(8'b00001111), .d0_init_a(8'b00001111),

.cy_dpconfig_a(
{
    `CS_ALU_OP__DEC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
```

```
    `CS_CMP_SEL_CFGA, /*CFGRAM1:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7:    */
   8'hFF, 8'h00,  /*CFG9:     */
   8'hFF, 8'hFF,  /*CFG11-10:    */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12:    */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
   1'h0, `SC_FIFO1_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /*CFG15-14:    */
   10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16:    */
}
)) cntr8(
        /*  input                   */  .reset(1'b0),
        /*  input                   */  .clk(clk), /* tie clk signal to datapath clock */
        /*  input   [02:00]         */  .cs_addr({2'b0,tc}), /* tc determines address lsb */
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0_load(1'b0),
        /*  input                   */  .f1_load(1'b0),
        /*  input                   */  .d0_load(1'b0),
        /*  input                   */  .d1_load(1'b0),
        /*  output                  */  .ce0(),
        /*  output                  */  .cl0(),
        /*  output                  */  .z0(tc), /* tc signal comes from z0 state */
        /*  output                  */  .ff0(),
        /*  output                  */  .ce1(),
        /*  output                  */  .cl1(),
        /*  output                  */  .z1(),
        /*  output                  */  .ff1(),
        /*  output                  */  .ov_msb(),
        /*  output                  */  .co_msb(),
        /*  output                  */  .cmsb(),
```

```
        /*  output                  */  .so(),
        /*  output                  */  .f0_bus_stat(),
        /*  output                  */  .f0_blk_stat(),
        /*  output                  */  .f1_bus_stat(),
        /*  output                  */  .f1_blk_stat()
);
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

已配置了递减和重新加载 A0 这两个配置选项，并定义了 A0 和 D0 的初始寄存器的值。另外，还将硬件链接至符号终端。您可以从头创建该文件，但使用数据路径配置工具创建它更为容易。

16 附录 D — 带有 PI 和 PO 的 24 位数据路径示例

```
// Requires these signals to tie the 3 datapaths together
      wire [14:0] chain0;
      wire [14:0] chain1;

// Datapath 0
cy_psoc3_dp #(.cy_dpconfig(
{
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
        8'hFF, 8'h00,      /*SC_REG4     Comment: */
        8'hFF, 8'hFF,      /*SC_REG5     Comment: */
      `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
      `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
      `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
      `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
      `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
      1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
      `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
      `SC_FB_NOCHN, `SC_CMP1_NOCHN,
      `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
       10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
      `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
      `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
})) Datapath0(
      /* input */ .clk(),              // Clock
      /* input [02:00] */ .cs_addr(),  // Control Store RAM address
      /* input */ .route_si(1'b0),     // Shift in from routing
      /* input */ .route_ci(1'b0),     // Carry in from routing
```

```
        /* input */ .f0_load(1'b0),        // Load FIFO 0
        /* input */ .f1_load(1'b0),        // Load FIFO 1
        /* input */ .d0_load(1'b0),        // Load Data Register 0
        /* input */ .d1_load(1'b0),        // Load Data Register 1
        /* output */ .ce0(),               // Accumulator 0 = Data register 0
        /* output */ .cl0(),               // Accumulator 0 < Data register 0
        /* output */ .z0(),                // Accumulator 0 = 0
        /* output */ .ff0(),               // Accumulator 0 = FF
        /* output */ .ce1(),               // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),               // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                // Accumulator 1 = 0
        /* output */ .ff1(),               // Accumulator 1 = FF
        /* output */ .ov_msb(),            // Operation over flow
        /* output */ .co_msb(),            // Carry out
        /* output */ .cmsb(),              // Carry out
        /* output */ .so(),                // Shift out
        /* output */ .f0_bus_stat(),       // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),       // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),       // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),       // FIFO 1 status to DP
        /* input */ .ci(1'b0),             // Carry in from previous stage
        /* output */ .co(chain0[12]),      // Carry out to next stage
        /* input */ .sir(1'b0),            // Shift in from right side
        /* output */ .sor(),               // Shift out to right side
        /* input */ .sil(chain0[10]),      // Shift in from left side
        /* output */ .sol(chain0[11]),     // Shift out to left side
        /* input */ .msbi(chain0[9]),      // MSB chain in
        /* output */ .msbo(),              // MSB chain out
        /* input [01:00] */ .cei(2'b0),    // Compare equal in from prev stage
        /* output [01:00] */ .ceo(chain0[1:0]), // Compare equal out to next stage
        /* input [01:00] */ .cli(2'b0),    // Compare less than in from prv stage
        /* output [01:00] */ .clo(chain0[3:2]), // Compare less than out to next stage
        /* input [01:00] */ .zi(2'b0),           // Zero detect in from previous stage
        /* output [01:00] */ .zo(chain0[5:4]),   // Zero detect out to next stage
        /* input [01:00] */ .fi(2'b0),           // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(chain0[7:6]),   // 0xFF detect out to next stage
        /* input [01:00] */ .capi(2'b0),         // Capture in from previous stage
        /* output [01:00] */ .capo(chain0[14:13]),     // Capture out to next stage
        /* input */ .cfbi(1'b0),                 // CRC Feedback in from previous stage
        /* output */ .cfbo(chain0[8]),           // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),               // Parallel data port
        /* output [07:00] */ .po()         // Parallel data port
);

// Datapath 1
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
```

```
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
          8'hFF, 8'h00,      /*SC_REG4     Comment: */
          8'hFF, 8'hFF,      /*SC_REG5     Comment: */
        `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
        `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
        `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
        `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
        `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
        1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
        `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
        `SC_FB_NOCHN, `SC_CMP1_NOCHN,
        `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
         10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
        `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
        `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
}}) Datapath1(
        /* input */ .clk(),                // Clock
        /* input [02:00] */ .cs_addr(),    // Control Store RAM address
        /* input */ .route_si(1'b0),       // Shift in from routing
        /* input */ .route_ci(1'b0),       // Carry in from routing
        /* input */ .f0_load(1'b0),        // Load FIFO 0
        /* input */ .f1_load(1'b0),        // Load FIFO 1
        /* input */ .d0_load(1'b0),        // Load Data Register 0
        /* input */ .d1_load(1'b0),        // Load Data Register 1
        /* output */ .ce0(),               // Accumulator 0 = Data register 0
        /* output */ .cl0(),               // Accumulator 0 < Data register 0
        /* output */ .z0(),                // Accumulator 0 = 0
        /* output */ .ff0(),               // Accumulator 0 = FF
        /* output */ .ce1(),               // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),               // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                // Accumulator 1 = 0
        /* output */ .ff1(),               // Accumulator 1 = FF
        /* output */ .ov_msb(),            // Operation over flow
        /* output */ .co_msb(),            // Carry out
        /* output */ .cmsb(),              // Carry out
        /* output */ .so(),                // Shift out
        /* output */ .f0_bus_stat(),       // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),       // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),       // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),       // FIFO 1 status to DP
        /* input */ .ci(chain0[12]),       // Carry in from previous stage
        /* output */ .co(chain1[12]),      // Carry out to next stage
        /* input */ .sir(chain0[11]),      // Shift in from right side
        /* output */ .sor(chain0[10]),     // Shift out to right side
```

```
        /* input */ .sil(chain1[10]),     // Shift in from left side
        /* output */ .sol(chain1[11]),    // Shift out to left side
        /* input */ .msbi(chain1[9]),     // MSB chain in
        /* output */ .msbo(chain0[9]),    // MSB chain out
        /* input [01:00] */ .cei(chain0[1:0]),  // Compare equal in from prev stage
        /* output [01:00] */ .ceo(chain1[1:0]), // Compare equal out to next stage
        /* input [01:00] */ .cli(chain0[3:2]),  // Compare less than in from prv stage
        /* output [01:00] */ .clo(chain1[3:2]), // Compare less than out to next stage
        /* input [01:00] */ .zi(chain0[5:4]),   // Zero detect in from previous stage
        /* output [01:00] */ .zo(chain1[5:4]),  // Zero detect out to next stage
        /* input [01:00] */ .fi(chain0[7:6]),   // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(chain1[7:6]),  // 0xFF detect out to next stage
        /* input [01:00] */ .capi(chain0[14:13]),// Capture in from previous stage
        /* output [01:00] */ .capo(chain1[14:13]),     // Capture out to next stage
        /* input */ .cfbi(chain0[8]),           // CRC Feedback in from previous stage
        /* output */ .cfbo(chain1[8]),          // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),              // Parallel data port
        /* output [07:00] */ .po()              // Parallel data port
);

// Datapath 2
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
          8'hFF, 8'h00,      /*SC_REG4     Comment: */
          8'hFF, 8'hFF,      /*SC_REG5     Comment: */
        `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
        `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
        `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
        `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
```

```
        `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
        1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
        `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
        `SC_FB_NOCHN, `SC_CMP1_NOCHN,
        `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
         10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
        `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
        `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
}}) Datapath2(
        /* input */ .clk(),                     // Clock
        /* input [02:00] */ .cs_addr(),         // Control Store RAM address
        /* input */ .route_si(1'b0),            // Shift in from routing
        /* input */ .route_ci(1'b0),            // Carry in from routing
        /* input */ .f0_load(1'b0),             // Load FIFO 0
        /* input */ .f1_load(1'b0),             // Load FIFO 1
        /* input */ .d0_load(1'b0),             // Load Data Register 0
        /* input */ .d1_load(1'b0),             // Load Data Register 1
        /* output */ .ce0(),                    // Accumulator 0 = Data register 0
        /* output */ .cl0(),                    // Accumulator 0 < Data register 0
        /* output */ .z0(),                     // Accumulator 0 = 0
        /* output */ .ff0(),                    // Accumulator 0 = FF
        /* output */ .ce1(),                    // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),                    // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                     // Accumulator 1 = 0
        /* output */ .ff1(),                    // Accumulator 1 = FF
        /* output */ .ov_msb(),                 // Operation over flow
        /* output */ .co_msb(),                 // Carry out
        /* output */ .cmsb(),                   // Carry out
        /* output */ .so(),                     // Shift out
        /* output */ .f0_bus_stat(),            // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),            // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),            // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),            // FIFO 1 status to DP
        /* input */ .ci(chain1[12]),            // Carry in from previous stage
        /* output */ .co(),                     // Carry out to next stage
        /* input */ .sir(chain1[11]),           // Shift in from right side
        /* output */ .sor(chain1[10]),          // Shift out to right side
        /* input */ .sil(1'b0),                 // Shift in from left side
        /* output */ .sol(),                    // Shift out to left side
        /* input */ .msbi(1'b0),                // MSB chain in
        /* output */ .msbo(chain1[9]),          // MSB chain out
        /* input [01:00] */ .cei(chain1[1:0]),  // Compare equal in from prev stage
        /* output [01:00] */ .ceo(),            // Compare equal out to next stage
        /* input [01:00] */ .cli(chain1[3:2]),  // Compare less than in from prv stage
        /* output [01:00] */ .clo(),            // Compare less than out to next stage
        /* input [01:00] */ .zi(chain1[5:4]),   // Zero detect in from previous stage
        /* output [01:00] */ .zo(),             // Zero detect out to next stage
        /* input [01:00] */ .fi(chain1[7:6]),   // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(),             // 0xFF detect out to next stage
        /* input [01:00] */ .capi(chain1[14:13]),// Capture in from previous stage
        /* output [01:00] */ .capo(),           // Capture out to next stage
        /* input */ .cfbi(chain1[8]),           // CRC Feedback in from previous stage
        /* output */ .cfbo(),                   // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),              // Parallel data port
        /* output [07:00] */ .po()              // Parallel data port
```

文档修订记录

文档标题：AN82156 — PSoC® 3、PSoC 4 和 PSoC 5LP — 使用 UDB 数据路径对 PSoC Creator™组件进行设计

文档编号：001-89207

版本	ECN	变更者	提交日期	变更说明
**	4127354	RLIU	09/18/2013	本文档版本号为 Rev. **，译自英文版 001-82156 Rev. *C。
*A	4718361	GKL	04/21/2015	本文档版本号为 Rev. *A，译自英文版 001-82156 Rev. *E。
*B	5012809	GKL	11/16/2015	本文档版本号为 Rev. *B，译自英文版 001-82156 Rev. *F。
*C	5464883	TDU	10/07/2016	本文档版本号为 Rev. *C，译自英文版 001-82156 Rev. *G。 更新到新的模板。 完成日落复审。
*D	5830736	AESATMP8	07/24/2017	更新标志和版权。

全球销售和设计支持

赛普拉斯公司拥有一个由办事处、解决方案中心、原厂代表和经销商组成的全球性网络。如欲查找离您最近的办事处，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 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，包括任何样本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。