AN82156

PSoC® 3 and PSoC 5 – Designing PSoC Creator™ Components with UDB Datapaths

Authors: Greg Reynolds, Todd Dust, and Mark Hastings
Associated Project: Yes
Associated Part Family: CY8C3xxxx, CY8C5xxxx
Software Version: PSoC Creator 2.1
For a complete list of related application notes, click here.
If you have a question, or need help with this application note, contact the authors at tdu@cypress.com, gir@cypress.com or meh@cypress.com.

AN82156 explains how to design PSoC® Creator™ components that use PSoC 3 and PSoC 5 Universal Digital Block (UDB) datapath modules. Datapath-based designs can offload CPU tasks to increase the efficiency of components. This application note describes the development process step by step. You also will learn how to use the PSoC Creator Datapath Configuration Tool to create, view, and modify datapath instances in Verilog files.

Contents

Introduction 1
Traditional PLDs vs. PSoC UDBs 2
  Datapath vs. PLD-based Designs 2
Datapath Architecture and Features 3
  Datapath Working Registers.....3
  Comparison Operators 3
  Inputs and Outputs 3
Datapath-based Components 4
  Creating a New Component Symbol 4
  Datapath Components and Verilog 4
  The Datapath Configuration Tool 4
Datapath Example Projects.....5
  Project #1 – Eight-Bit Down Counter.....5
  Project #2 – 16-bit PWM.....16
  Project #3 – Up / Down Counter.....19
  Project #4 – Simple UART.....23
Summary.....26
Related Application Notes.....26
Appendix A – Datapath Configuration Tool Details 27
  Dynamic Configuration RAM Section 29
  Static Configuration Section 32
Appendix B – Auto-Generated Verilog Code.....38
  A New Verilog File Generated by PSoC Creator.....38
  The Verilog File with a New Datapath Instance.....38
  The Verilog File after All Modifications 41

Introduction

Have you maxed out your CPU bandwidth? PSoC 3 and PSoC 5 UDBs can lighten the load on your CPU by creating intelligent custom peripherals. The datapaths in those UDBs can be used to create sophisticated multiprocessor-based designs.

Many PSoC customers have experience writing HDL code for CPLDs or FPGAs. PSoC 3 and PSoC 5 support Verilog for PLDs. However, PSoC 3 and PSoC 5 PLDs are smaller than full-fledged CPLDs or FPGAs, and many designs are too large for PSoC 3 and PSoC 5. PSoC's unique datapath modules remove this obstacle by integrating one or more small 8-bit processors into PLD-based designs.

This application note introduces you to the UDB datapath and teaches you how to use it in PSoC Creator components. It is an advanced application note—it assumes that you are familiar with developing applications using PSoC Creator for PSoC 3 or PSoC 5. If you are new to PSoC 3 or PSoC 5, see the introductions in AN54181, Getting Started with PSoC 3, and AN77759, Getting Started with PSoC 5. If you are new to PSoC Creator, see the PSoC Creator home page.

This application note also assumes a basic understanding of Verilog and its usage with PSoC 3 and PSoC 5 UDBs. If you are new to these concepts, see AN81623, PSoC 3 and PSoC 5 Digital Design Best Practices, and AN82250, PSoC 3 and PSoC 5 Implementing Programmable Logic Designs.

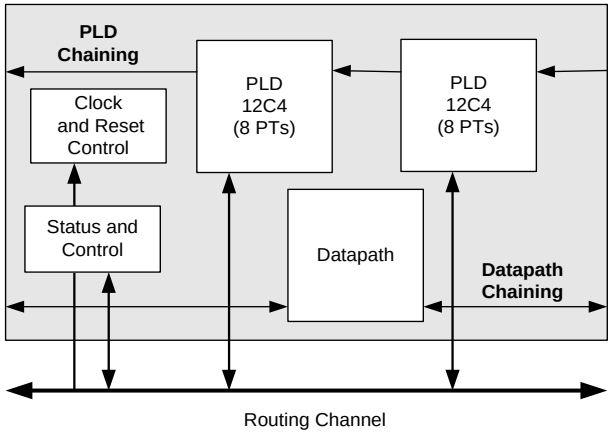
For a list of related PSoC design resources, see the Related Application Notes section.

Traditional PLDs vs. PSoC UDBs

PSoC 3 and PSoC 5 are highly optimized programmable devices that can match or exceed the functionality of much bigger programmable logic products. They are not designed to directly integrate a large FPGA or CPLD implementation. Instead, PSoC 3 and PSoC 5 contain an array of small, fast, low-power programmable digital blocks, or UDBs.

The UDBs are made up of two small PLDs, a datapath module, and status and control logic, as Figure 1 shows.

Figure 1. Simplified UDB Block Diagram



PSoC 3 and PSoC 5 devices can have as many as 24 UDBs. You can chain UDBs to create peripheral functions that are greater than 8 bits.

A datapath module is a configurable 8-bit ALU with as many as six registers that can perform simple functions, such as increment, decrement, add, subtract, bitwise logic, and shift.

The datapath modules create smart custom peripherals that can offload CPU tasks. They can easily implement often-used functions, such as counters, PWMs, and shifters. To implement complex functions, use the datapath with PLD-based glue logic and state machines.

Datapath vs. PLD-based Designs

Datapath modules can be used to conserve limited UDB PLD resources. For example, if a simple 8-bit down counter is implemented solely in the PLDs, it uses the following resources:

Resource	#Used	%Used (24 UDBs)
Macrocells	8	4.17%
Total PTerms	8	2.08%
Datapath Cells	0	0.00%

If the same counter is implemented in the datapath, it uses the following resources:

Resource	#Used	%Used (24 UDBs)
Macrocells	1	0.52%
Total PTerms	2	0.52%
Datapath Cells	1	4.17%

If you use an up counter in a PLD-based design, you will consume even more resources:

Resource	#Used	%Used (24 UDBs)
Macrocells	8	4.17%
Total PTerms	15	3.91%
Datapath Cells	0	0.00%

However, this is not the case with the datapath because it can perform both increment and decrement operations:

Resource	#Used	%Used (24 UDBs)
Macrocells	1	0.52%
Total PTerms	2	0.52%
Datapath Cells	1	4.17%

The best way to use UDB resources is as follows:

- PLDs: Combinatorial logic, glue logic, state machines. (See AN82250, PSoC 3 and PSoC 5 Implementing Programmable Logic Designs and AN81623, PSoC 3 and PSoC 5 Digital Design Best Practices, for more information on PLD implementations.)
- Datapaths: Calculations, data management, and byte-wide comparisons.

You can implement complex functions in PSoC 3 and PSoC 5 PLDs, but it is easy to run out of resources if you do not take advantage of the datapath modules.

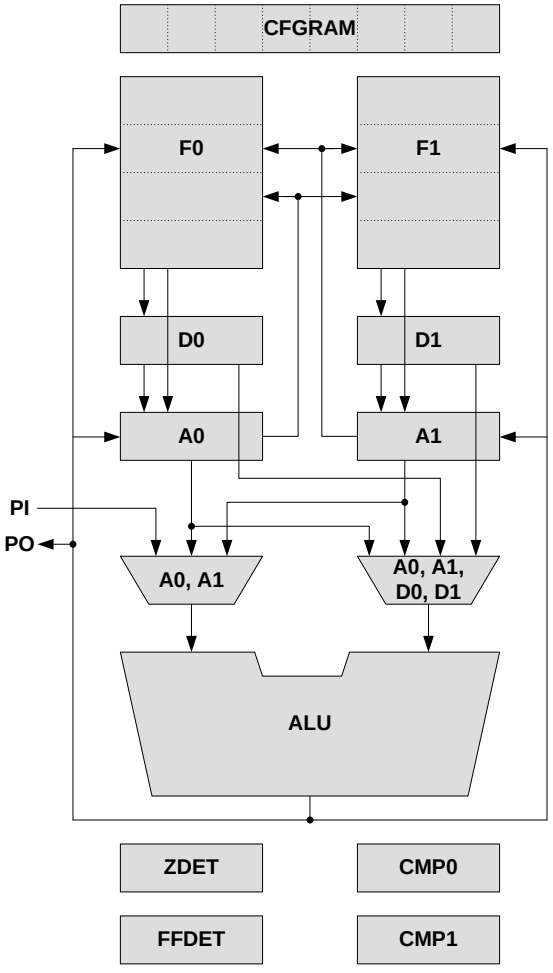
Datapath Architecture and Features

The datapath module is a small 8-bit processor that can support as many as eight program steps, or configurations, as Figure 2 shows. It has an ALU, configuration RAM, four working registers, two FIFOs, two comparators, five condition detectors, and input and output data ports. You can use the FIFOs and ports to route data to other digital blocks or to the CPU.

The ALU can perform eight functions: pass-through, increment, decrement, add, subtract, XOR, AND, and OR. Independent blocks also exist for shifting and masking.

Eight unique datapath configurations can be stored in the configuration RAM (CFGRAM). The configurations control the datapath functions and the interconnects. The CFGRAM address inputs determine, on each rising edge of the datapath clock, which configuration is used.

Figure 2. Simplified Datapath Block Diagram



See the PSoC 3 Architecture TRM (Technical Reference Manual) for complete details on the UDBs and datapaths.

Datapath Working Registers

Each datapath module has six 8-bit working registers:

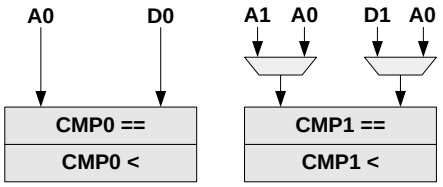
- A0 and A1 – The accumulator registers typically hold data used in an ALU operation. You can also use them to store data from the Dx registers or FIFOs.
- D0 and D1 – The data registers typically contain static data, such as the starting value to load into a counter.
- F0 and F1 – These registers are 4-byte FIFOs that you can use as both source and destination buffers.

You can read and write any datapath register using the CPU or DMA, but use the FIFOs whenever possible. The working registers operate asynchronously and may change at any time, including during CPU or DMA accesses.

Comparison Operators

Each datapath has two compare functions: “less than” and “equal to”, as Figure 3 shows. The compare blocks can also use masks to perform bitwise comparisons.

Figure 3. CMP0 and CMP1 Inputs



The A0 and A1 registers have zero (ZDET), all ones (FFDET), and overflow detect functions. The FIFOs have full and empty status signals.

Inputs and Outputs

Within and surrounding the UDBs is the DSI, an extensive fabric of programmable switches that connect signals within a UDB, between pairs of UDBs, throughout the array of UDBs, and between the UDB array and many other blocks in PSoC 3 and PSoC 5.

Datapath control and status signals can be routed as follows:

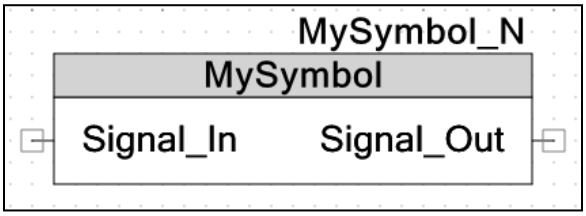
- Within the UDB to for desired functions.
- Through the DSI for UDB chaining to implement larger (more than 8-bit) functions.
- Through the DSI to other blocks in PSoC 3 and PSoC 5 (DMA, analog, I/Os) for system-level functions.

As noted, you can access the CPU and DMA through the FIFOs, as well as through the UDB control and status registers.

Datapath-based Components

A PSoC Creator component, as Figure 4 shows, is the simplest way to use a datapath. Cypress has developed a *Component Author Guide (CAG)* to help you create new components. Open the guide from within PSoC Creator by selecting *Help→Documentation→Component Author Guide*. The *CAG* tells you how to create a component step by step and describes how to add features that are not discussed in this application note.

Figure 4. Example Component Symbol



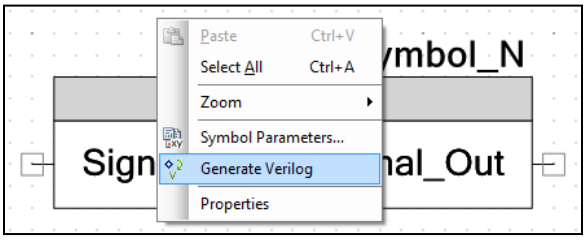
Creating a New Component Symbol

The first step in creating a component is to create a component symbol, using the Symbol Wizard. The example projects in this application note guide you through the steps to create a component symbol. You can also create a symbol manually, as described in the CAG. After you create the symbol, the datapath can be instantiated in Verilog.

Datapath Components and Verilog

Components that include a datapath module must be based on Verilog. PSoC Creator can auto-generate a Verilog file for your component symbol, as Figure 5 shows.

Figure 5. Verilog Generation by PSoC Creator



The auto-generated Verilog file does not contain a datapath instance when it is first created. You must modify the Verilog file to instantiate a datapath configuration. PSoC Creator includes a Datapath Configuration Tool to help with this task.

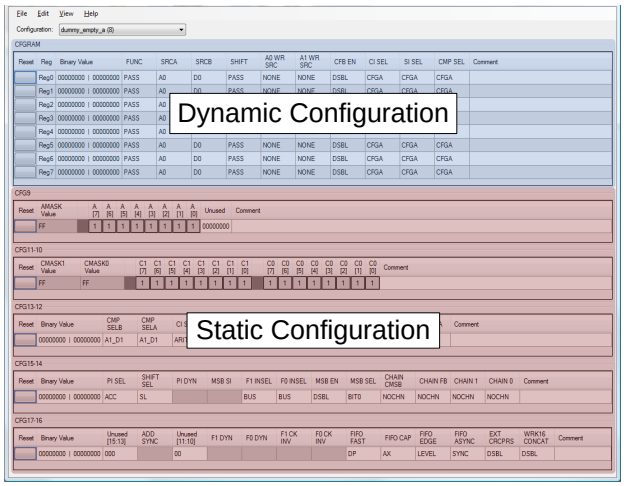
The Datapath Configuration Tool

The Datapath Configuration Tool is an application that allows you to create, view, modify, and delete datapath instances within component Verilog files. It parses a Verilog file and displays each datapath instance as an entity, called a “configuration”. One configuration represents a single physical datapath module.

If no configuration is present in the Verilog file, you can create a new one and add it to the file. You can add multiple configurations to a single file. Configurations can be linked to represent connections between datapaths.

The configurations are presented to the user through a graphical interface (GUI), as Figure 6 shows. The GUI display is just a representation of the datapath’s dynamic configuration RAM and static configuration registers.

Figure 6. Example Datapath Configuration Tool Interface



There are two major sections in the GUI. Appendix A contains descriptions of the GUI fields in each section. The GUI fields correspond to registers in the PSoC device. The registers are described in detail in the PSoC 3 Architecture TRM.

The upper section, “CFGRAM”, is the dynamic configuration. It determines the functions of the ALU core. Each row represents one of the eight configuration states stored in the configuration RAM.

The lower section is the static configuration, which represents the datapath registers CFG9 to CFG17. Those registers control static functions, including shift direction, masking, FIFO configuration, and chaining.

When you save a configuration in the Datapath Configuration Tool, you modify the Verilog file. Any changes made to the fields in the Datapath Configuration Tool result in changes to the Verilog code.

Datapath Example Projects

One of the best ways to learn about a PSoC feature is to use it in a design. This application note contains four example projects that guide you through the steps to create five simple datapath-based components:

- 8-bit counter
- 8-bit PWM
- 8-bit up/down counting PWM
- 16-bit PWM
- TX-only simple UART

You can use the example projects in this application note on any PSoC 3 or PSoC 5 device. Completed example projects are available on this application note's landing page.

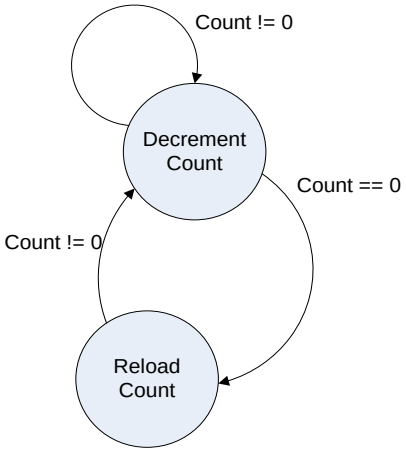
Project #1 – Eight-Bit Down Counter

The purpose of this project is to introduce you to the steps you must take to create a simple datapath-based component. First you will create an 8-bit down counter component, and then you will modify it to become an 8-bit PWM.

8-bit Counter Component Details

A simple down counter can be represented by a state machine with two states, as Figure 7 shows. A state transition takes place on the rising edge of the datapath clock.

Figure 7. Simple Counter State Diagram



The counter starts with an initial value and decrements it. When the count reaches zero, an event is triggered and the period is reloaded. This type of counter is easily implemented in a datapath.

You need only two configuration RAM entries to implement the counter. The first decrements a count value stored in one of the working registers. The second reloads the count with the initial value. The remaining six configuration RAM entries are unused.

You need a method to decide the datapath's state: loading or decrementing. Figure 7 shows that state transitions are controlled by the value of the count, that is, whether the count is zero or not.

The datapath has a built-in zero detector block (ZDET); its output is high when the count is zero and low when the count is non-zero. You can use the ZDET output 'z0' to drive which configuration is used. The addressing of CFGRAM requires 3 bits. You can tie bit 0 to z0 and tie the other bits LOW. The transition table looks like the following:

Configuration State	Address Bits			State Description
	2	1	0	
0	0	0	z0 = 0	Decrement Count
1	0	0	z0 = 1	Reload Count
2-7	X	X	X	Not Used

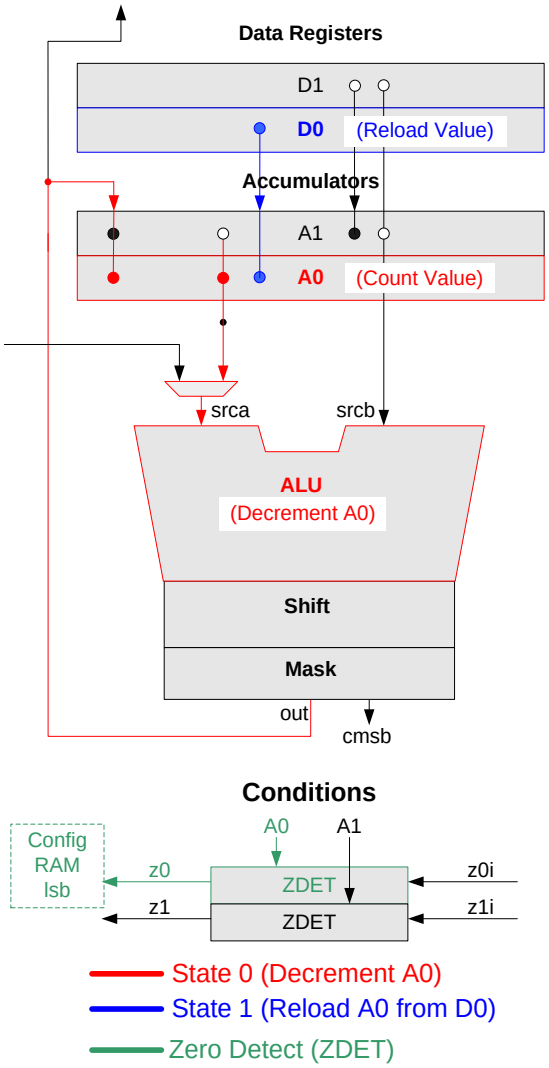
When the count reaches zero, z0 becomes '1'. This causes the next state to be "Reload Count". When the count is reloaded, z0 becomes '0' and the next state is "Decrement Count".

You need to use only two datapath registers to implement the counter:

- A0 – holds the count value and is decremented by the ALU. The ZDET block continually monitors the value of A0.
- D0 – holds the value that is reloaded into A0 when the count reaches zero.

This may be easier to visualize by looking at a highlighted datapath block diagram, as Figure 8 shows.

Figure 8. Simple Counter Block Diagram Detail

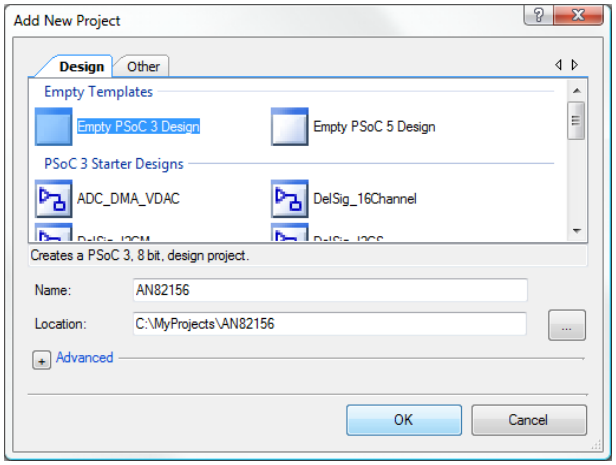


8-bit Counter Component Creation Steps

You can use an existing PSoC Creator project and add a new component library to it. But, for this example, you will use an empty project as a starting point.

1. Launch PSoC Creator and start a project called "AN82156", as Figure 9 shows. An "AN82156" workspace is also created by default.

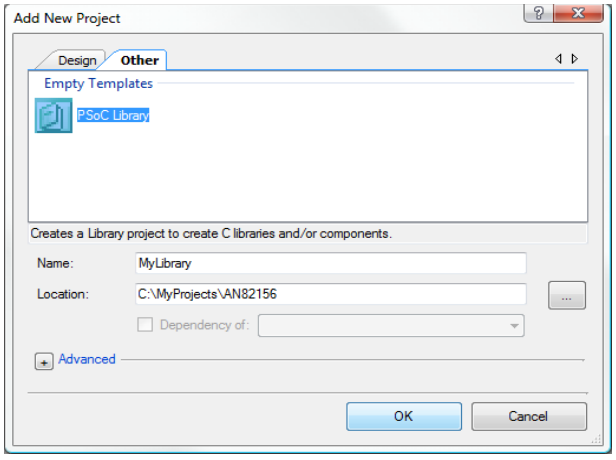
Figure 9. Add New Project



Components are stored in a PSoC Creator library project. It is best to create a library project for your components so that they are not overwritten by updates to the standard Cypress libraries.

2. Highlight the *AN82156* workspace in the Source tab of the Workspace Explorer. Right-click on it and select *Add→New Project* from the drop-down menu that appears.
3. Select the Other tab to reveal the PSoC Library template, as Figure 10 shows. Name it "*MyLibrary*" for this example, and leave the location as its default value. This creates the library in the same location as the *AN82156* workspace.

Figure 10. Add New Library



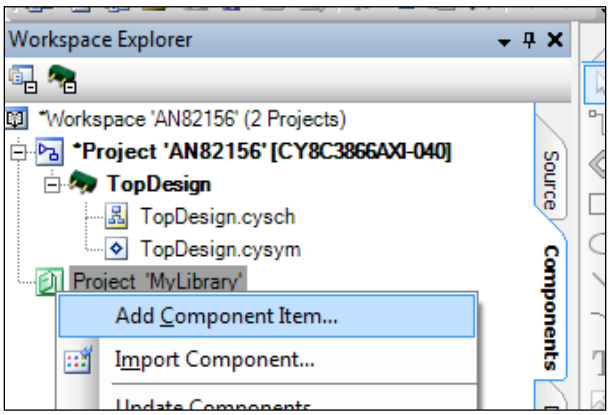
The new library now shows up in the Workspace Explorer. Later, you will link this library to the example project so that you can use the components stored in it.

Now you need to add a new component to the library just created:

4. Switch to the Components tab of the Workplace Explorer and right-click on the *MyLibrary* project.

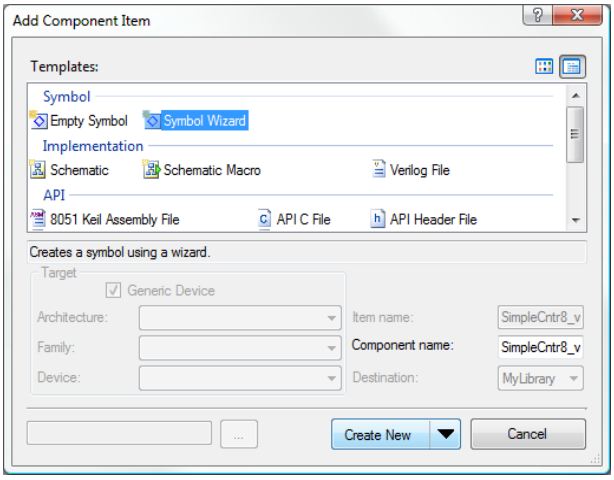
Select *Add Component Item* from the pop-up menu, as Figure 11 shows.

Figure 11. Add Component Item



5. Select the *Symbol Wizard* component template and name the component "SimpleCntr8_v1_0", as Figure 12 shows.

Figure 12. Use Symbol Wizard



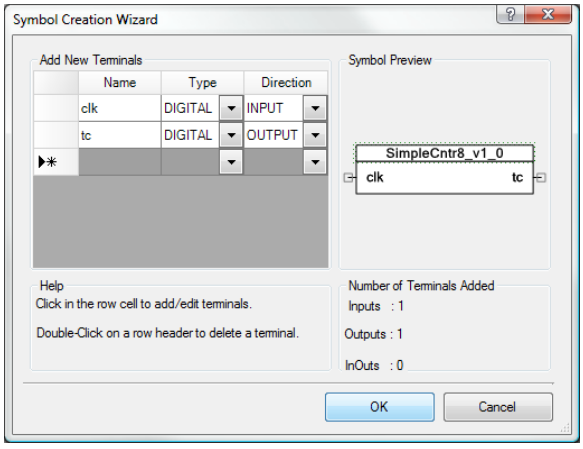
It is a good idea to include a version number in the component name. Append to the component name the tag "_vX_Y", where 'X' is the major version and 'Y' is the minor version. PSoC Creator has versioning capabilities and can help you track and use multiple versions of your components.

6. Click the Create New button to launch the component symbol wizard.

This wizard asks you to define the inputs and outputs, and it uses this information to create a component symbol. You will define one input terminal and one output terminal in the schematic symbol.

7. Add two terminals in the *Add New Terminals* fields – a "*clk*" input and a "*tc*" output, as Figure 13 shows.

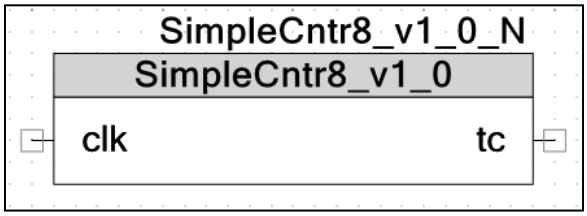
Figure 13. Add Inputs and Outputs



The "clk" input will be the datapath clock. The "tc" output will be used to tell you when the count value is zero.

8. Click OK to generate the symbol in the symbol schematic page, as Figure 14 shows.

Figure 14. Generated Component Symbol



Right now, the 'clk and 'tc' terminals are just part of the schematic symbol; they don't do anything. You will define their function later, using Verilog.

9. Right-click on an empty space in the symbol schematic – not the symbol itself – and select *Properties* from the drop-down menu that appears.
10. Enter values in the *Symbol* section of the property fields, as Figure 15 shows:

- *Doc.APIPrefix* = SimpleCntr8.

  This value is appended to any API functions generated for the component. You will not be generating an API in this example, but it is good practice to enter a value here whenever you create a new component.

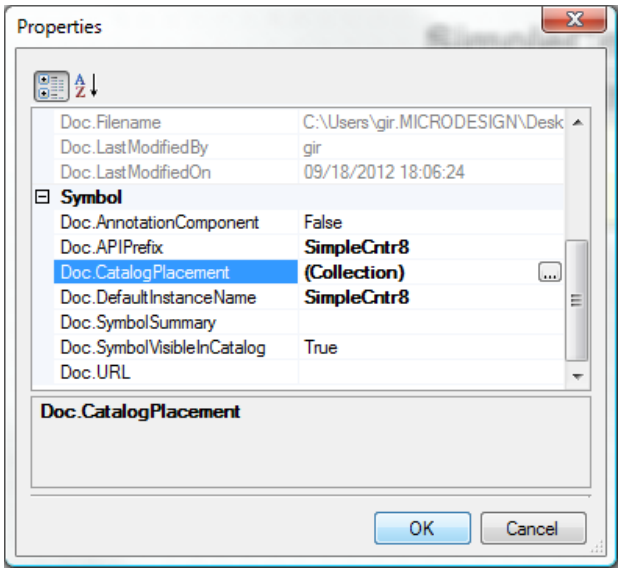
- *Doc.CatalogPlacement* = MyLibrary/Digital/Cntr8.

  You must click on the '…' to open the Catalog Placement dialog to enter this value. This value is the hierarchy for the component placement within a library. The first term must be the name of the library. Each subsequent '/' represents a subfolder. The value you are using means that the component will be visible as 'Cntr8' in the 'Digital' folder of MyLibrary.

- *Doc.DefaultInstanceName* = SimpleCntr8.

This is the default name that appears when the component is placed in a schematic. You can change it after you place the component in the project schematic.

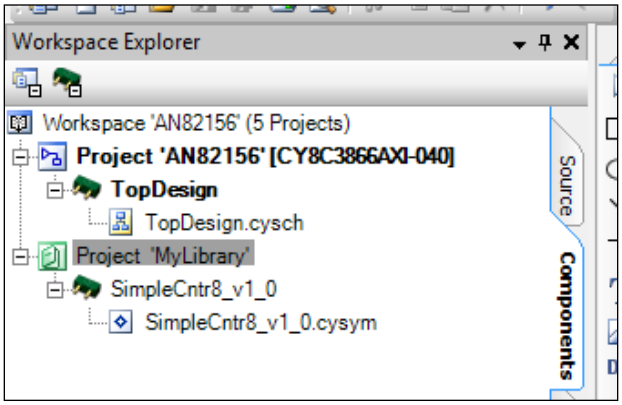
Figure 15. Add Symbol Properties



Use the same name for the Doc.*APIPrefix* and Doc.*DefaultInstanceName* to avoid confusion between the API and schematic symbol.

11. Perform a “Save All” to ensure that all changes have been applied to the project. The new symbol shows up under ‘MyLibrary’ in the Components tab of the Workspace Explorer, as Figure 16 shows.

Figure 16. New Component in the Library

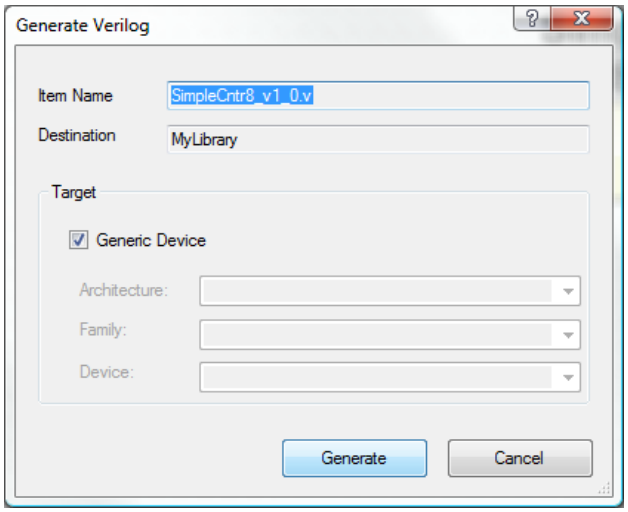


Now you need to link the schematic symbol to a datapath implementation. PSoC Creator can generate a Verilog file that does this for you.

12. Right-click on an empty space in the symbol schematic and select *Generate Verilog* from the drop-down menu that appears.

13. Leave all settings in the *Generate Verilog* dialog box at the defaults and click Generate, as Figure 17 shows.

Figure 17. Generate Verilog Dialog



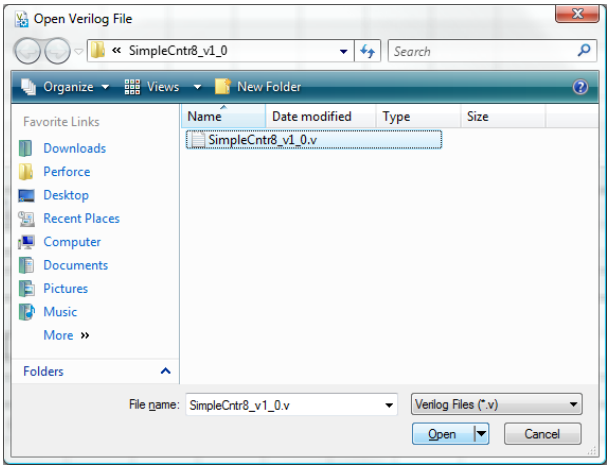
14. Perform a “Save All” to ensure that all changes have been applied to the project. A new Verilog file is generated and added to the component.

The Verilog file contains the datapath implementation, as well as all Verilog code needed to control the datapath. To add and edit datapath instances, use the *Datapath Configuration Tool*.

15. Launch the Datapath Configuration Tool. It is separate from PSoC Creator, but it can be found in the *Component Development Kit* folder in the PSoC Creator application files on your PC.

16. In the datapath configuration tool, Select *File→Open* and browse to the location of the *SimpleCntr8_v1_0.v* file generated by Creator. If you followed the steps in this example, the file is located in the *MyLibrary* project folder inside the *AN82156* workspace folder, as Figure 18 shows.

Figure 18. Example Component Verilog File

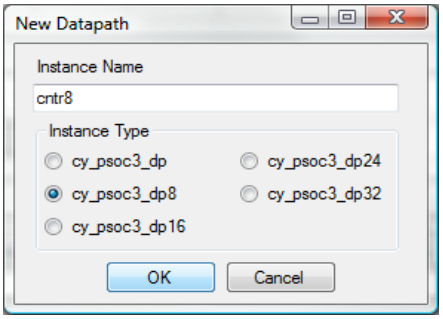


17. Select the file and click Open to load the Verilog file into the datapath tool.

The first step is to create a new datapath configuration.

18. Select *Edit→New Datapath* from the menu to open the New Datapath dialog window.
19. Enter an Instance Name (use "cntr8") and select the cy_psoc3_dp8 Instance Type, as Figure 19 shows.

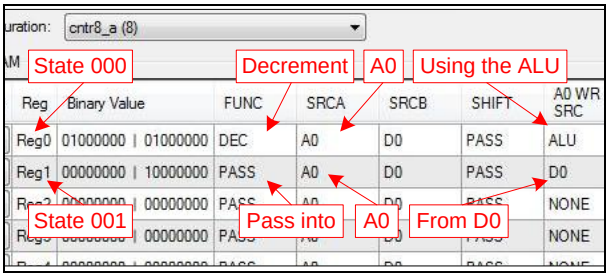
Figure 19. New 8-bit Datapath Instance



This will be an 8-bit counter, so you do not need to define the datapath to be anything larger than 8 bits (cy_psoc3_dp8). The other example projects demonstrate components wider than 8 bits.

20. Select values in the Reg0 and Reg1 fields to configure the datapath, as Figure 20 shows. The other fields can remain at their default settings.

Figure 20. Configuration for the SimpleCntr8 Component



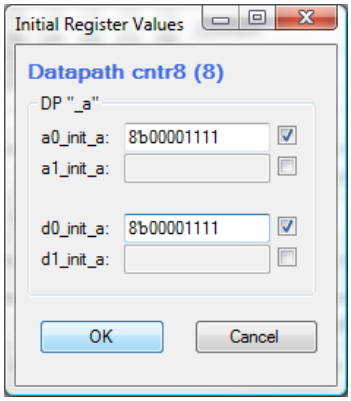
FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
DEC	A0	D0	PASS	ALU
PASS	A0	D0	PASS	D0

The Reg0 and Reg1 rows represent the first two of the eight datapath configuration states. You configured the first one to decrement the counter and the second one to reload the counter.

You need to start with a count value loaded into the A0 and D0 registers. The Ax and Dx registers are accessible by the CPU, so you could use code in *main.c* to load them. You also can load a default initial value into them by defining it in the Verilog file. The datapath tool can do this for you.

21. Select *View→Initial Register Values* to open the register value dialog.
22. Check the boxes for *a0_init_a* and *d0_init_a*. Set the values to *8'b00001111* to define a period of 16 counts, as Figure 21 shows.

Figure 21. Initial Register Values for the SimpleCntr8



23. Click OK when done.

In this case, the D0 and A0 values are identical so that you reload the same count value into A0. You can choose different values if you want the first period to be different from the rest.

24. Select *File→Save* from the menu to apply the changes to the Verilog file. Close the Datapath Configuration Tool and look at the *Counter_8bit_v1_0.v* file in Creator.

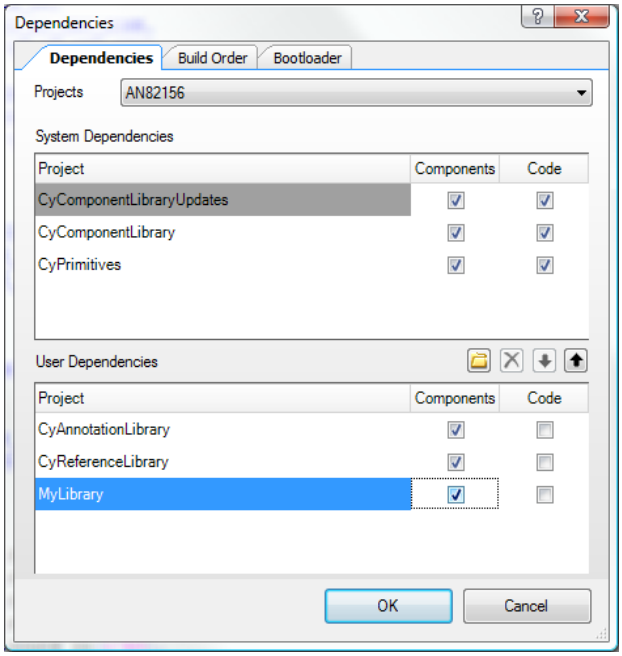
The Datapath Configuration Tool makes changes to the Verilog file when you save the configuration. PSoC Creator may ask to reload the file when you switch back. At that point, you need to make some changes to the Verilog file to link the schematic symbol to the datapath logic.

25. Change `.clk(1'b0)` to `.clk(clk)`. This links the datapath clock to the symbol's 'clk' terminal. You can use this terminal to set the speed at which the datapath will operate.
26. Change `.z0(),` to `.z0(tc)`. This links the z0 output of the zero detect block (ZDET) to the symbol's 'tc' terminal. If linked, the 'tc' terminal reflects the value of the z0 output.
27. Change the `.cs_addr(3'b0)` text to `.cs_addr({2'b00,tc})`. This sets the two most significant bits of the control store address to '0' and sets bit 0 to the value of `tc`.
28. Save all changes to the Verilog file.

Now that the component is ready for use, you need to set *MyLibrary* as a project dependency.

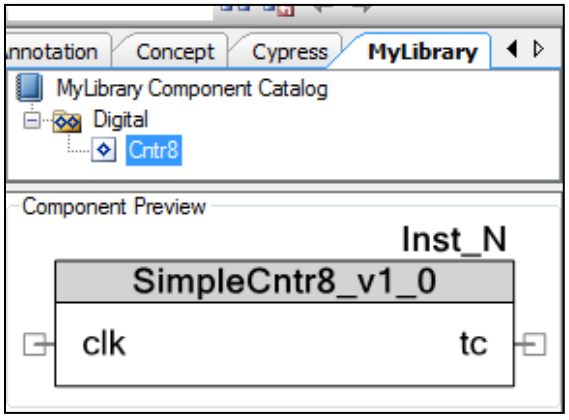
29. Add the new *MyLibrary* as a user dependency of the AN82156 project – be certain that the Components box is checked. To Add a dependency, right-click on *'Project AN82156'* in the Source tab and select *Dependency*, as Figure 22 shows.

Figure 22. Adding MyLibrary as a Project Dependency



The new *MyLibrary* and its components are now visible in the Component Catalog, as Figure 23 shows.

Figure 23. New Component in the Catalog



After the component is visible in the Component Catalog, you can place it in the schematic and use it like any other component.

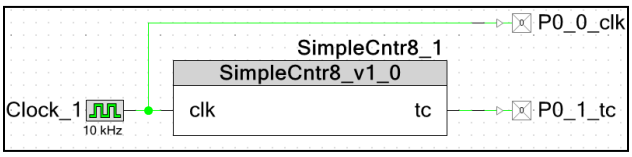
To test it, you need to add a clock source and a way to view the counter's 'tc' output.

30. Place the Cntr8 component in the project schematic.
31. Connect a clock component to the 'clk' terminal. Set it to 10 kHz. Any other value would work, but 10 kHz makes it easy to view.
32. Connect a digital output pin to the 'clk' terminal so that you can observe it on a scope. Name it P0_0_clk and leave all other settings at their default.

33. Connect a digital output pin to the 'tc' terminal. Name it P0_1_tc and leave all other settings at their default. This pin will be used to observe when the count reaches zero.

Figure 24 shows the completed project schematic.

Figure 24. Simple Counter Project Schematic

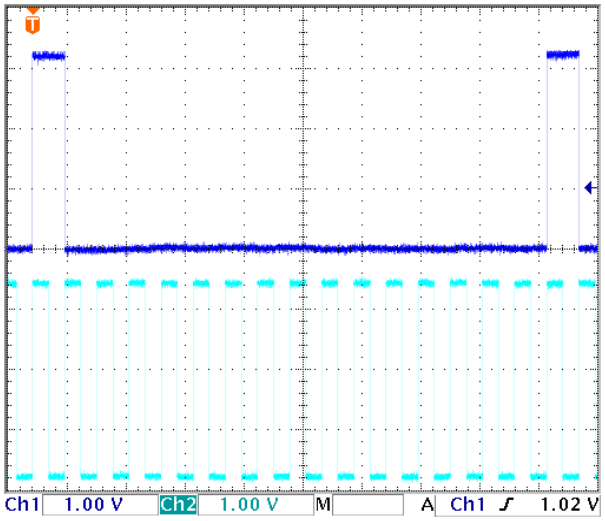


34. Assign the pins to P0[0] and P0[1] according to their names.

Now you are ready to compile the firmware and to program the PSoC. You can observe the clock and the terminal count on pins P0[0] and P0[1].

35. Save the project, compile it, and program the PSoC.
36. If you connect a scope to the output pins, you can observe the 'clock' and 'tc' outputs, as Figure 25 shows.

Figure 25. Simple Counter Outputs



You loaded A0 with a starting value of 0x0F, so you can see that the period is 16 clock cycles wide. The 'tc' pin pulses HIGH for only one clock cycle, when A0 reaches zero, because A0 is reloaded with the contents of D0 at that time. When A0 is no longer zero, 'tc' is set LOW and the state machine transitions back to begin decrementing A0 again.

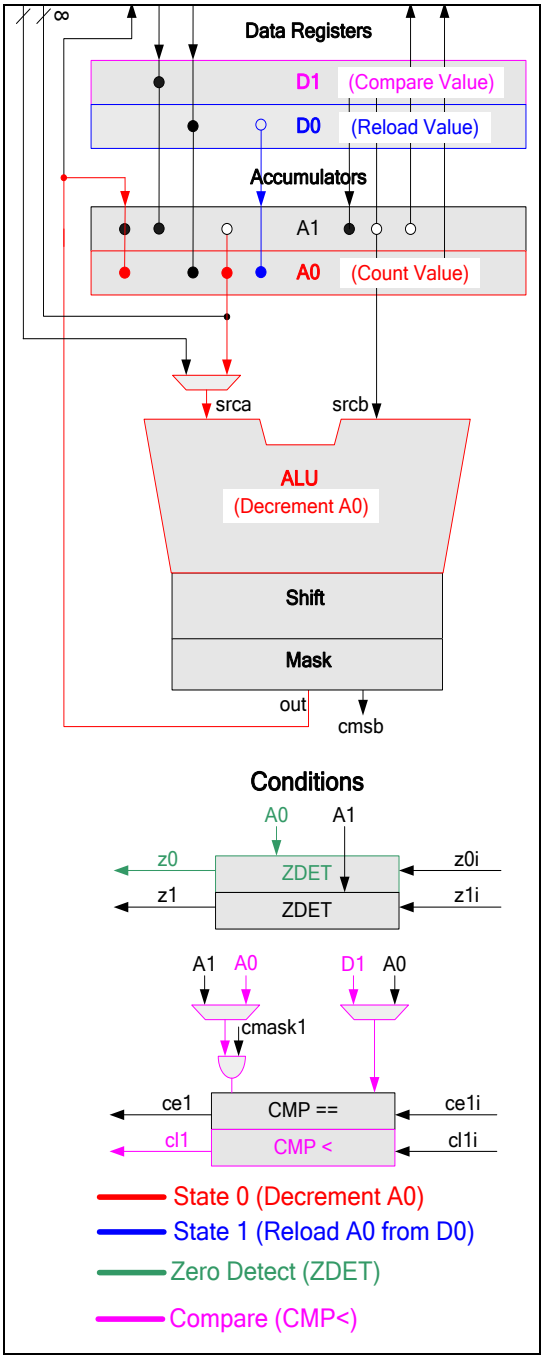
You have just designed your first datapath-based component, and you didn't write any firmware to do it!

Modify the Counter to be a PWM

A PWM is just a counter with a compare. To add PWM, you need a way to compare the value in A0 with another fixed value. You can have the D1 register hold the fixed compare value, and set the compare block to check if A0 is less than D1.

You can visualize it using a highlighted block diagram, as Figure 26 shows:

Figure 26. Simple PWM Block Diagram Highlights

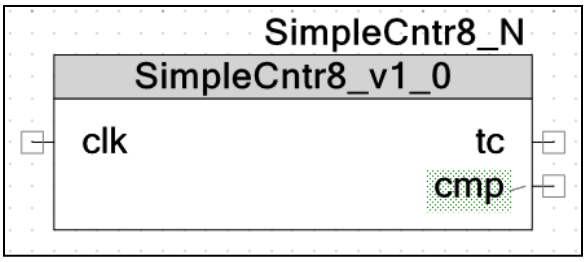


This example modifies the 8-bit counter component that you built in the previous section. You can start with an

empty component or create a copy of the previous counter component, but these steps assume you are modifying the existing counter.

1. Open the component symbol (cysym) file for the SimpleCntr8_v1_0.
2. Stretch the symbol body down two ticks to create space for another terminal.
3. Press 'O' to add a digital output terminal. Name it 'cmp' and place it as Figure 27 shows.

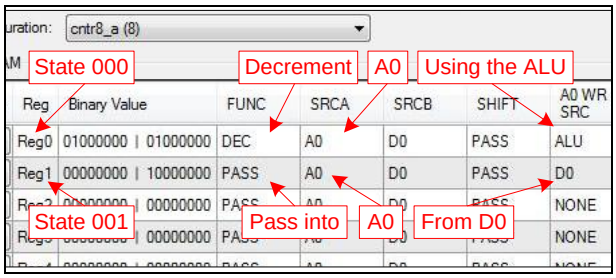
Figure 27. Component Symbol with 'cmp' Terminal



4. Do a "Save All" to capture the changes.
5. Launch the Datapath Configuration Tool and open the *SimpleCntr8_v1_0.v* Verilog file. The Datapath Configuration Tool automatically parses the Verilog file and displays the configuration that is present.

We said earlier that a PWM is just a counter with a compare. You still need only two states, and the count value in A0 is still decremented and reloaded. This means that most of the settings in the Datapath Configuration Tool remain unchanged, as Figure 28 shows.

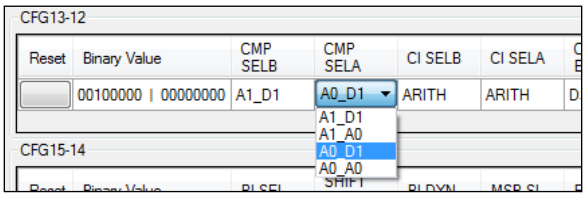
Figure 28. CFGRAM Settings for PWM



You need to add the (A0 < D1) compare. To do that, set the configurable compare block.

6. Set *CMP SELA* to "A0_D1" to configure the compare register to use A0 and D1 for the compare, as Figure 29 shows.

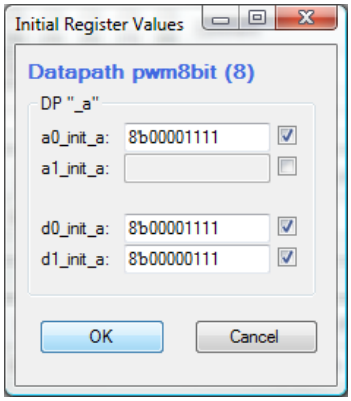
Figure 29. Compare Block 1 Mux Settings



Remember that A0 and D0 hold the count and reload values. You need to add a default compare value in the D1 register.

7. Select *View→Initial Register Values* from the menu and open the dialog window.
8. Check the box for *d1_init_a*.
9. Leave the 'a0' and 'd0' values at *8'b00001111*. This sets the period to 16 clock cycles.
10. Set the 'd1' value to *8'b00000111*. This sets the compare value to 7, as Figure 30 shows.

Figure 30. Initial PWM Register Values



11. Click 'OK' to apply the values to the registers.
12. Do a Save to capture the changes in the Verilog file.

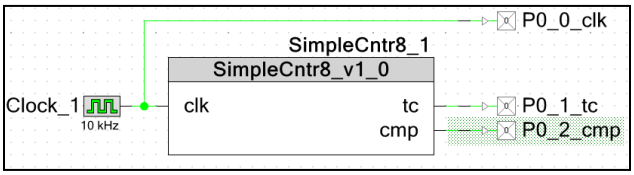
Now the compare block will output HIGH whenever A0 is less than D1. You need to link the symbol to the datapath hardware using the Verilog file, just as you did with the counter.

13. Add `output cmp,` below `output tc,`. This creates the link to the 'cmp' terminal in the component symbol.
14. Change the `.cl1()` line to `.cl1(cmp)`. This links the 'cmp' signal to the "less-than" output of the Compare1 block.
15. Save all changes to the Verilog file.

The PWM component and symbol are still visible in the Component Catalog in the MyLibrary tab. The component is automatically updated in the project schematic.

16. Add an output pin and connect it to the 'cmp' terminal. Name it *P0_2_cmp* and assign it to pin P0[2], as Figure 31 shows.

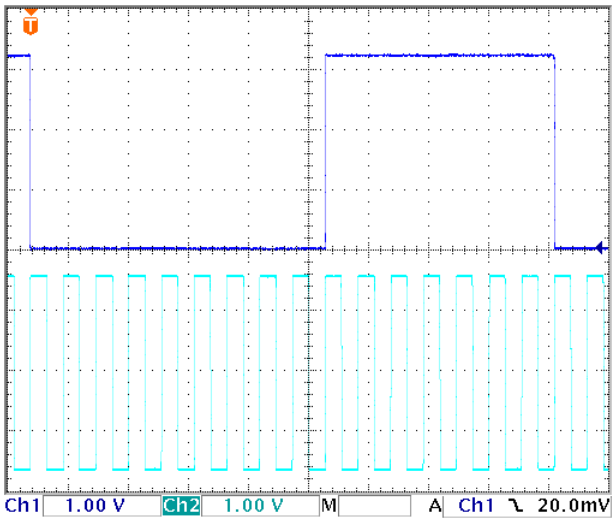
Figure 31. Simple PWM Project Schematic



Now you are ready to compile the firmware and program the PSoC. The clock and the terminal count can still be observed on pins P0[0] and P0[1]. The PWM output can be observed on P0[2].

17. Save the project, compile it, and program the PSoC.
18. If you connect a scope to the output pins, you can observe the 'clock', 'tc', and 'cmp' outputs. Figure 32 shows the 'clk' and 'cmp' signals.

Figure 32. Simple PWM Outputs



You loaded A0 and D0 with a starting value of 0x0F, so you know that the period is 16 clock cycles wide. You set D1 to be 7, so the 'cmp' pin is HIGH whenever A0 is less than 7. You can change the compare value in D1 to test your PWM.

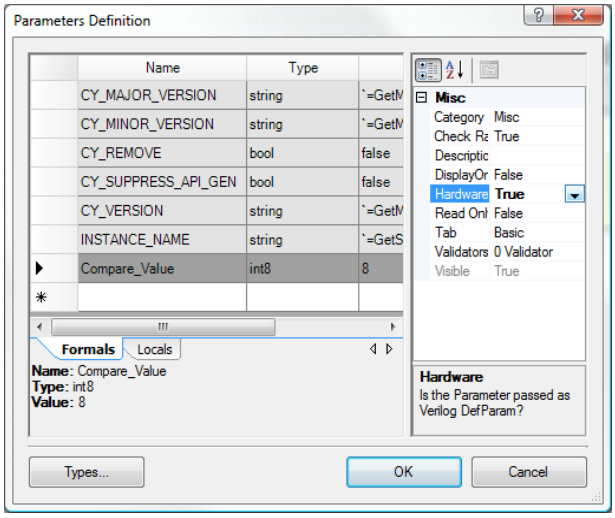
Adding Parameters

It is inconvenient to have to modify Verilog code whenever you need to make a change to one of the component's parameters. And, what if you needed two PWMs with different periods and compare values? You can make minor modifications to the component to change parameters from within the project schematic, similar to the way most Cypress components work.

1. Open the component's symbol page and right-click in an empty space.
2. Select *Symbol Parameters* from the drop-down menu that appears.
3. Enter a new parameter in the empty row below the existing parameters:
   - Name = Compare_Value
   - Type = uint8
   - Value = 8
4. Set the 'Hardware' flag to "True" in the *Misc* settings field on the right of the window, as Figure 33 shows.

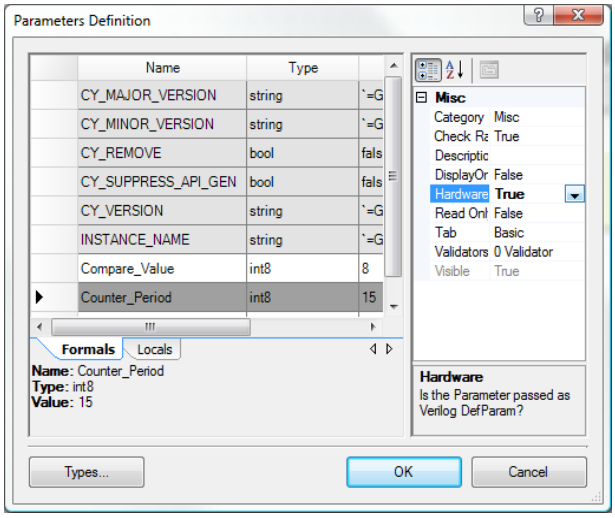
This exposes the parameter to the Verilog so that the UDB hardware can use it.

Figure 33. Adding a New Component Parameter



5. Enter another new parameter in the row below the compare value definition you just created:
   - Name = Counter_Period
   - Type = uint8
   - Value = 15
6. Set the Hardware flag to "True" in the Misc settings field on the right of the window, as Figure 34 shows.

Figure 34. Adding Another New Component Parameter



7. Click OK and do a “Save All” to apply the changes to the component.

Now, you need to link the component symbol parameters you just defined to the Verilog logic.

8. Open the component’s Verilog file and locate the text, on or near line 23, that says:

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

9. Replace that text with:

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

10. Locate the text on or near line 26 that holds the initial register values:

```
cy_psoc3_dp8 #(.a0_init_a(8'b00001111),
.d0_init_a(8'b00001111),
.d1_init_a(8'b00000111),
```

11. Replace the fixed values with the parameters you just defined:

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

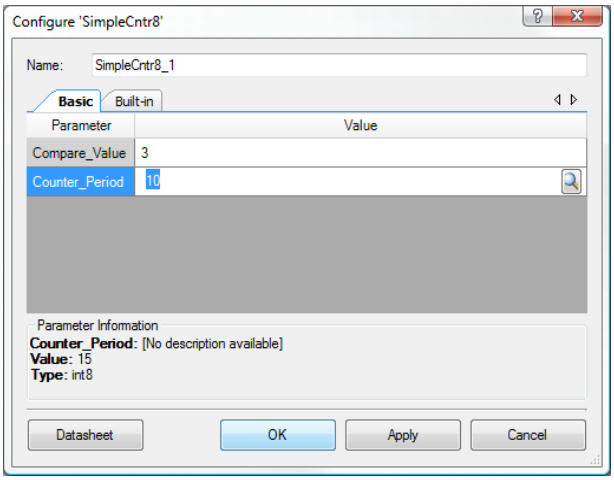
This code links the initial register values for A0, D0, and D1 to the component’s parameters. Now the user can set these at compile time without modifying the component.

12. Do a “Save All,” compile the code, and program the PSoC.

Previously, you set the default compare value parameter to 8. Now you can change the parameters from within the project schematic.

13. Go back to the project schematic and double-click on the SimpleCntr8_1 component to open the properties.
14. Change the compare value to 3 and the counter period to 10, as Figure 35 shows.

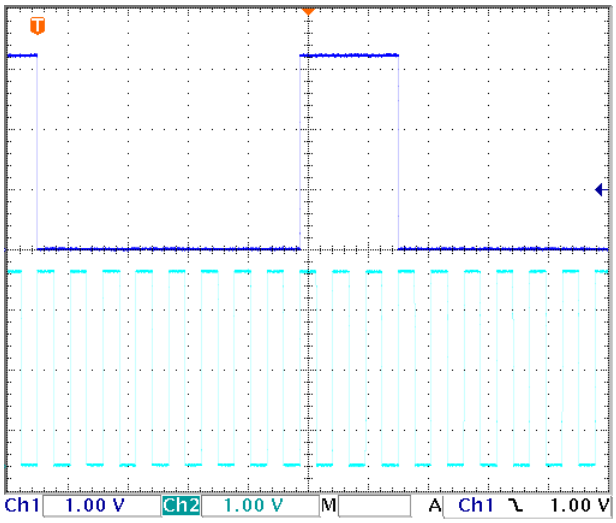
Figure 35. Selecting Component Parameters



15. Click OK to apply the changes.
16. Do a “Save All,” compile the code, and program the PSoC.

You can see in Figure 36 that the period and compare output are different than they were before.

Figure 36. PWM Output with New Parameters



You set the period to 11 cycles (the period is 10+1 because the counter will go from 10 to 0 before it reloads), and the compare to 3. This results in eight clock cycles of LOW output and three cycles of HIGH output.

You can change the parameters to almost anything you want, as long as the value is an 8-bit integer. You can even place multiple instances of the component in your project and set them to different values.

For more information on adding component parameters, including how to set limits on what value the user can enter, see The *Component Author Guide*.

Adding Header Files

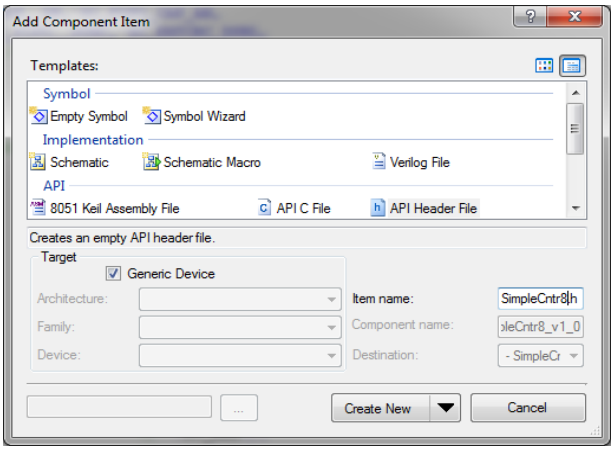
In addition to being able to change the behavior of the PWM at design time, it is often desirable to change the PWM during run-time. This can be done by modifying the PWM registers via C code. To make these registers easy to access, create a header file that defines the various registers used.

For instance, with the PWM, you used register D0 to hold the period value and register D1 to hold the compare value.

17. In the components tab, right-click on SimpleCntr8_v1_0, click *add component item*
18. In the add component Item window, navigate down to the API section and click on *API Header File.*

19. Change the *Item name* to SimpleCntr8.h, as Figure 37 shows.

Figure 37. Add Header File



20. Click the *Create New* button. This will add a header file to your component.

Now it is time to add defines for the compare value (D1) and the period value (D0). First, you need to find out how these registers are defined by PSoC Creator. To do this, you must build a project with the PWM and look in the cyfitter.h file.

21. Build the project you previously created with the PWM.
22. Open cyfitter.h
23. Find the defines for SimpleCntr8_1

You should notice in this file a set of defines for all of the registers associated with SimpleCntr8_1. For this example, you will use the defines for D0 and D1. Copy the following two lines:

```
#define        SimpleCntr8_1_cntr8_u0__D0_REG
CYREG_B0_UDB02_D0

#define        SimpleCntr8_1_cntr8_u0__D1_REG
CYREG_B0_UDB02_D1
```

24. Copy and paste these defines into your header file.
25. Remove the CYREG and all text after it for each define
26. Create a unique name for each register. For example, call D0 Period_Reg, and D1 Compare_1.
27. Prepend each of these unique names with `` `$INSTANCE_NAME`_ ``. This will allow for unique defines for each instance of the component placed on the schematic.
28. Next, after each unique name, you need to tell Creator that you are accessing a register. Do this by adding `(*(reg8 *)` in front of what you are defining. If you wanted to define a pointer, you could use `((reg8 *)`.

Your defines should look like the following:

```
#define          `$INSTANCE_NAME`_Period_Reg
(*(reg8 *) SimpleCntr8_1_cntr8_u0__D0_REG )

#define          `$INSTANCE_NAME`_Compare_Reg
(*(reg8*) SimpleCntr8_1_cntr8_u0__D1_REG )
```

Now, if you want to update the compare value during run-time, simply write to the define you just created. If your component was named SimpleCntr8_1, then your C code would look like the following:

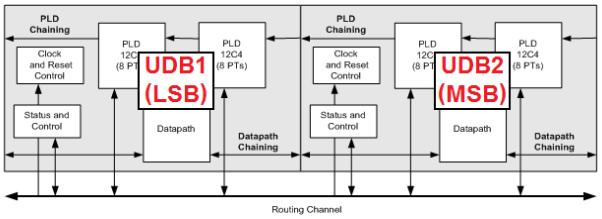
```
SimpleCntr8_1_Compare_Reg = 0x02;
```

This would update the compare value to hex 0x02. For more information on this and how to use these defines, consult the attached example projects. Note the method of directly writing to the register, as in the C code above, works only for 8-bit registers. For 16 bits or higher, you need to use a different method, which is discussed in the next project.

Project #2 – 16-bit PWM

The purpose of this example project is to introduce you to the concept of datapath chaining. You will create a PWM that is similar to the first example project, but it will be 16 bits wide, as Figure 38 shows.

Figure 38. Two Chained UDBs



This example assumes that you are already familiar with the concepts introduced in the previous example projects. A completed 16-bit PWM project is included with this application note if you need to refer to it.

16-bit PWM Component Details

The basic functionality of a 16-bit PWM is the same as that of the 8-bit PWM from the first example project. In both cases, a count is decremented and an output goes HIGH when the count is less than a compare value. The difference is that you need to use two datapaths to manipulate all 16 bits.

16-bit PWM Component Creation Steps

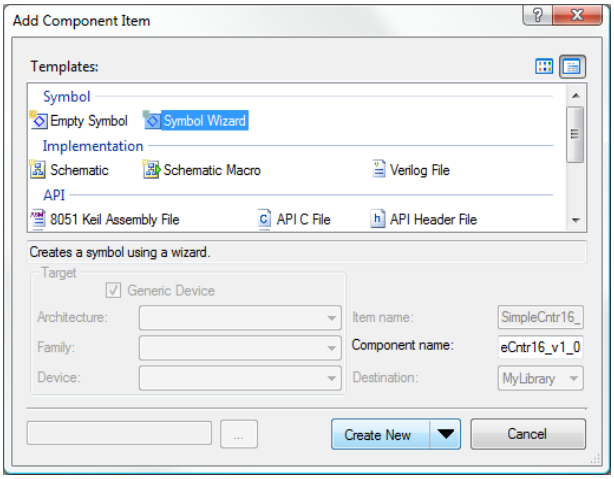
To avoid confusion, create a new component, instead of modifying the one from the first example project. The basic component creation steps are the same.

1. Launch PSoC Creator and open the "AN82156" workspace that you used for the previous example. Add a new project, called "*16bitCounter*", to the workspace.

You can start in a new workspace, but this example assumes that you are using the same one as before. Using the same one simplifies library and dependency management.

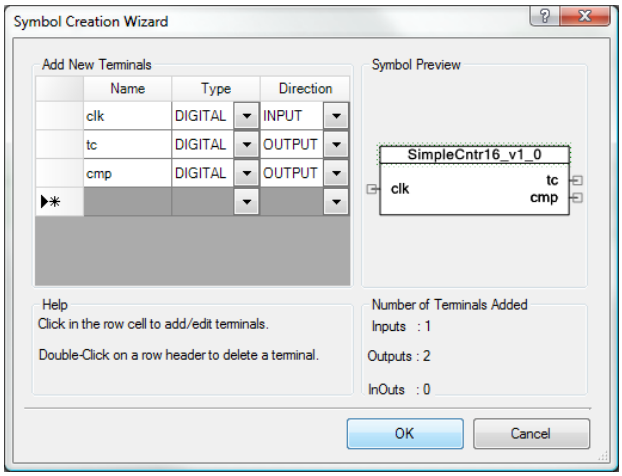
2. Add a new component item to *MyLibrary* (this example will use "*SimpleCntr16_v1_0*" as the component name) using the Symbol Wizard, as Figure 39 shows.

Figure 39. New Symbol Creation



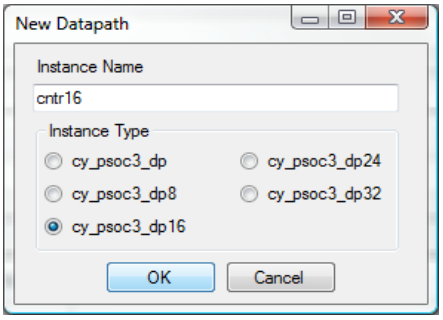
3. Add a *clk* input and *tc* and *cmp* outputs, just as in the 8-bit PWM example, as Figure 40 shows.

Figure 40. Adding Terminals to the PWM



4. Right-click on the symbol schematic page and add symbol properties:
   - Doc.APIPrefix = *SimpleCntr16*
   - Doc.CatalogPlacement = *MyLibrary/Digital/Cntr16*
   - Doc.DefaultInstanceName = *SimpleCntr16*
5. Generate the Verilog file for the new component symbol – leave everything at defaults.
6. Do a "Save All" to apply the changes.
7. Launch the Datapath Configuration Tool and open the Verilog file you just created.
8. Add a new datapath configuration (this example will use "*cntr16*" as the configuration name). Select "cy_psoc3_dp16", as Figure 41 shows.

Figure 41. Creating a New PWM Datapath

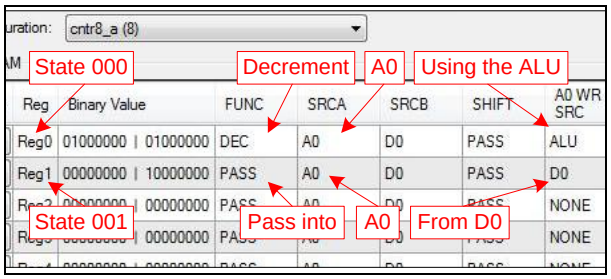


9. Click OK to create the datapath configuration.

Notice that there are now *cntr16_a(16)* and *cntr16_b(16)* configurations. Two separate datapaths instances will be added to the Verilog file. The 'a' configuration is for the LSB, and the 'b' configuration is for the MSB.

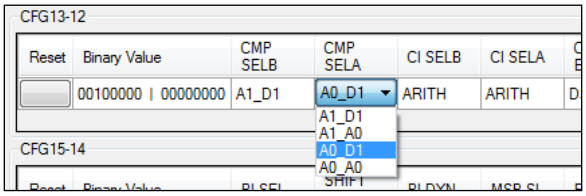
10. Set the "_a" configuration CFGRAM and CFG13-12 sections the same as in the 8-bit PWM example, as Figure 42 and Figure 43 show.

Figure 42. Configuration for the SimpleCntr16 Component



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

Figure 43. Compare Block 1 Mux Settings



Remember to set the ALU function and the Compare block the same as in the 8-bit example. You haven't configured chaining yet, so the settings are the same.

11. Do a Save to apply the changes to the Verilog file.

You need to make the same changes to the "_b" configuration because it will be the upper 8 bits of our PWM. The Datapath Configuration Tool can copy settings from one configuration to another.

12. Select *Edit→Copy Datapath* to copy the configuration.

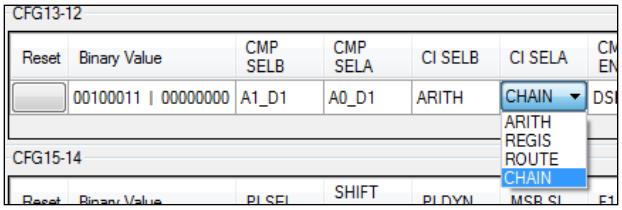
13. Switch to the 'cntr16_b' configuration and select *Edit→Paste Datapath* to paste the '_a' configuration into the '_b' configuration.

14. Do a Save to apply the changes to the Verilog file.

Now you will configure the chaining; only the '_b' configuration will use these settings.

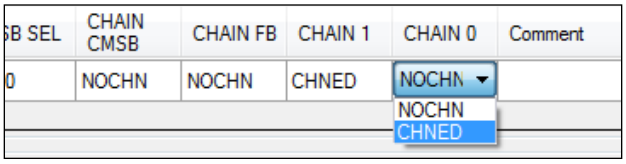
15. In the 'cntr16_b' configuration, set CI_SELA to "CHAIN", as Figure 44 shows. This configures the carry-in signal to come from the previous datapath.

Figure 44. Configuring Carry-in Chaining



16. Set CHAIN1 and CHAIN0 to "CHNED", as Figure 45 shows. These configure the compare conditions to be chained (ce0, ce1, cl0, cl1, z0, z1, ff0, ff1).

Figure 45. Configuring Datapath Chaining



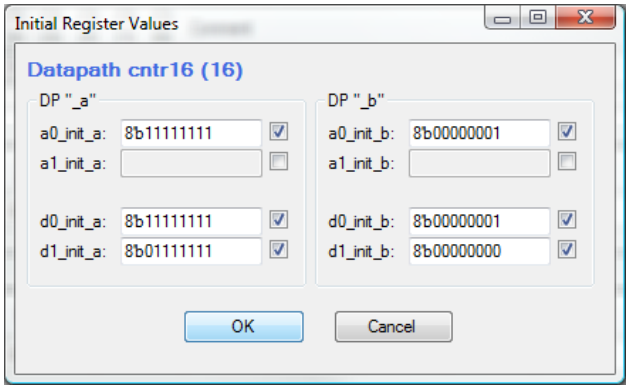
17. Do a Save to apply the changes to the Verilog file.

You also need to set initial values for the registers. The previous example demonstrated how to use parameters to configure the period and compare values. Because this example is focused on chaining, use fixed values to keep things simple.

18. Select *View→Initial Register Values*.

19. Set the DP"_a" and DP"_b" values, as Figure 46 shows.

Figure 46. Initial Register Values for the 16-bit PWM



Remember that the "_a" configuration is the LSB and the "_b" configuration is the MSB. These values give the PWM a period of 511 clock cycles and a compare value of 127.

20. Do a Save to apply the changes to the Verilog file and close the Datapath Configuration Tool.
21. Open the *SimpleCntr16_v1_0.v* Verilog file.

Notice that there are two datapath instances in the Verilog code. These two datapaths are chained together to form the 16-bit PWM.

22. Add the following Verilog code after `'#start body'` on or near line 23:

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

The hardware links are similar to those of the 8-bit PWM. The difference is that now the outputs are 2 bits wide, instead of 1. Because you chained *Chain0 and Chain1* in the Datapath Configuration Tool, the upper bit of the outputs contains the final result. For example, the upper bit of the Z0 output will tell us when both datapaths A0 registers are all zero. So you assign 'tc' and 'cmp' to the upper bit, and use the two wires you defined in step 22 for the lower bits. These wires aren't used, but if they are not added, the Verilog synthesizer will generate a warning.

23. Assign the 'clk' input to .clk().
24. Assign the 'tc' output to .z0({`tc, tc_lsb}`).
25. Assign the 'cmp' output to .cl1(`{cmp, cmp_lsb}`).
26. Set `.cs_addr(3'b0)` to `.cs_addr({2'b0,tc})`.
27. Do a "Save All" to apply the changes.

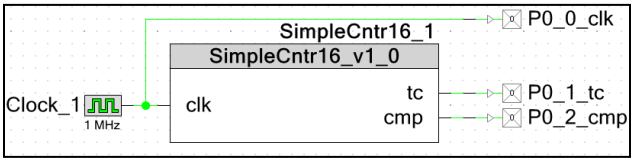
The component is now ready for use in your project.

28. Set your *MyLibrary* library as a dependency for the *16bitCounter* project.

The new component should be present in the Component Catalog along with the UpDwnCntrPwm component. It is ready for use in your project.

29. Drag a Cntr16 component onto the schematic.
30. Connect a clock to the 'clk' terminal and set it to 1 MHz.
31. Connect output pins to the 'clk', 'tc', and 'cmp' terminals, as Figure 47 shows.

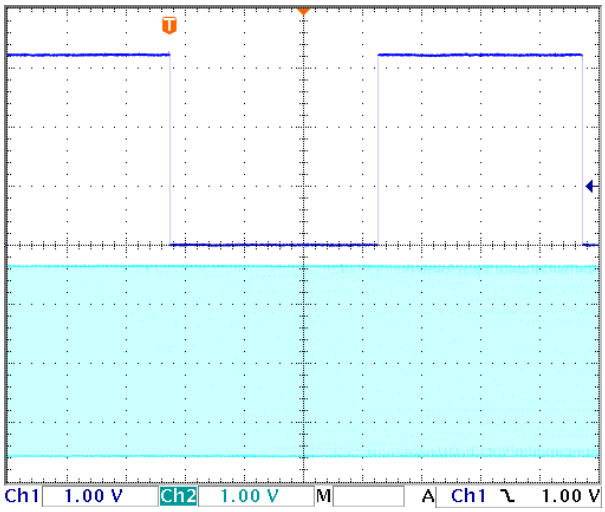
Figure 47. Simple 16-bit PWM Project Schematic



32. Name the pins as shown above and assign them to P0[0], P0[1], and P0[2] according to their names.
33. Do a "Save All", compile the code, and program the PSoC.

You can observe the outputs to see that the period and compare values are much larger than the 8-bit PWM you previously made, as Figure 48 shows. You can set them to any 16-bit value.

Figure 48. Simple 16-bit PWM Output



Congratulations on making a 16-bit PWM. The ability to chain multiple UDBs creates many possibilities for your components.

Header Files for 16-bit Components

As noted, writing to 16-bit registers is different from 8-bit writes. You can directly write to 8-bit registers because you don't have to worry about endian differences between the processor and the datapath registers. When you move up to 16 bits and higher, you need to worry about endian differences.

The PSoC 3's 8051 endian-ness is different from the peripheral registers. To make writing to registers simple, Cypress provides a few macros: CY_SET_REG16, CY_SET_REG24, CY_SET_REG32. These macros take the register address that you want to set as the first parameter, followed by the value you want to set. These macros handle any endian swapping that you need.

Therefore, you need to know the address of the datapath registers. The process to create defines for these is exactly the same as before. The only difference is you use `((reg8 *)` instead of `(*(reg8 *)` in the define. This gives us a pointer to the datapath register. Often, it is best to add _PTR to the end of the define name so you can differentiate.

Then, when you want to update a value, use CY_SET_REG16 and the pointer you defined in your header file. Again, there are examples in the attached example project.

Project #3 – Up / Down Counter

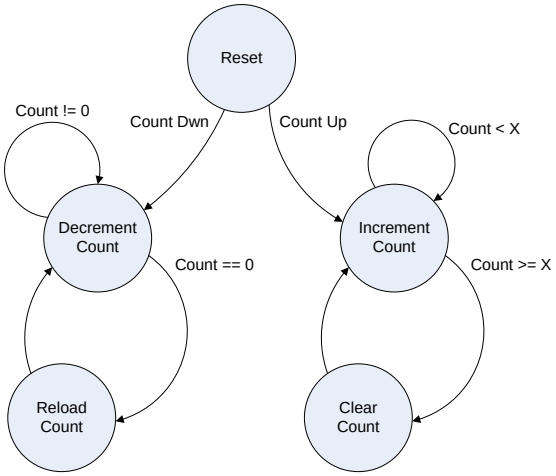
The purpose of this example is to introduce you to the steps for adding more advanced features to a datapath component. The same basic PWM concept is updated to add the ability to count up or down based on a parameter that the user can set at design time or during run-time.

This example assumes that you are already familiar with the concepts introduced in the previous example projects. A completed Up/Down PWM project is included with this application note.

Additional Details

The simple down-counting PWM used two states. To implement an up/down counter, four states are needed, as Figure 49 shows.

Figure 49. Up/Down Counter State Diagram



The datapath will decrement or increment A0 depending on the parameter you set. You also can see the period and compare.

Example Project Steps

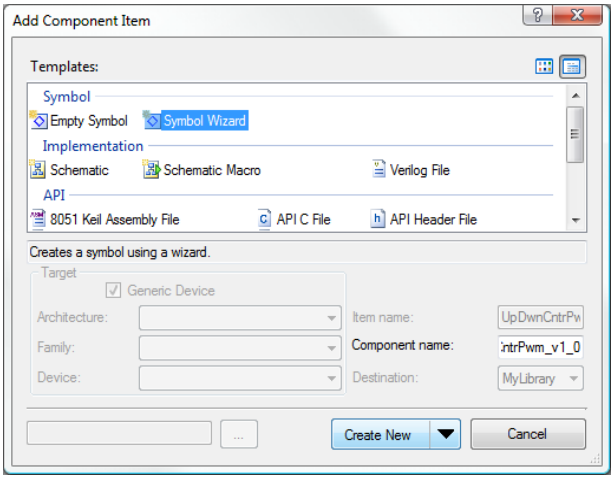
You will create a new component, instead of modifying the ones from the previous example projects, to avoid any confusion. The basic component creation steps are the same.

1. Launch PSoC Creator and open the "AN82156" workspace that you used for the simple 8-bit example. Add a new project, named "*UpDwnCntrPwm*", to the workspace.

You can start a new workspace, but this example assumes that you are using the same one as before. This simplifies library management and dependencies.

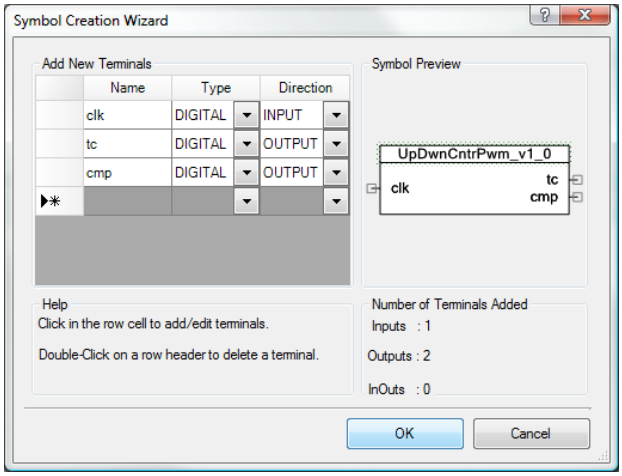
2. Add a new component item to *MyLibrary* (this example will use "*UpDwnCntrPwm_v1_0*" as the component name) using the Symbol Wizard, as Figure 50 shows.

Figure 50. Creating a New Symbol for the Up/Down Counter



3. Add a *clk* input and *tc* and *cmp* outputs, just like the 8-bit PWM component, as Figure 51 shows.

Figure 51. Adding Terminals for the Up/Down Counter



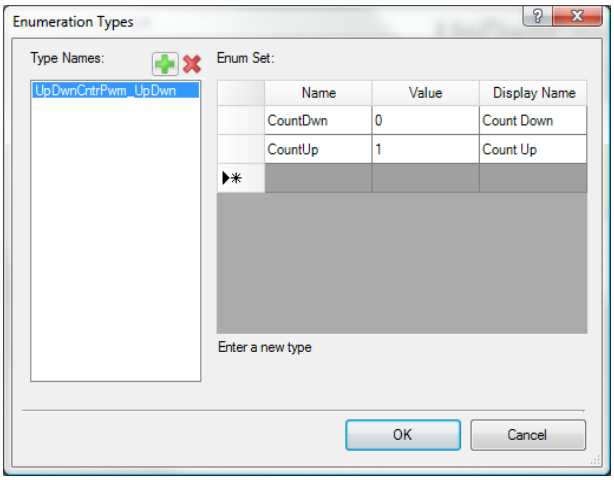
4. Right-click on the symbol schematic page and add symbol properties:
   - Doc.APIPrefix = *UpDwnCntrPwm*
   - Doc.CatalogPlacement = *MyLibrary/Digital/UpDwnCntrPwm*
   - Doc.DefaultInstanceName = *UpDwnCntrPwm*

You need to add some parameters that the user can modify – period, compare value, and Up/Down mode. For the Up/Down mode setting, define a new type of parameter and its values.

5. Right-click on the symbol schematic page and open the symbol parameters dialog.
6. Click on the 'Types' button to open the window to create two new parameter types.

7. Click the green '+' button to add a new type. Name it *UpDwnCntrPwm_UpDwn*.
8. Enter values into the *Enum Set* fields to define 'CountDwn' and 'CountUp' definitions, as Figure 52 shows.

Figure 52. Creating New Component Parameter Types



9. Click 'OK' to return to the symbol parameters dialog.

You can now set parameters to this new type and assign an initial value of 0 (CountDwn) or 1 (CountUp).

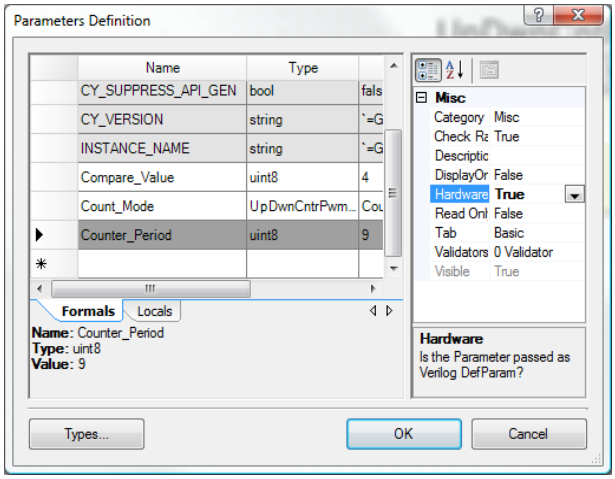
10. Enter three new parameters for the component, just as you did in the previous examples:

Name	Type	Value
Compare_Value	uint8	4
Count_Mode	UpDwnCntrPWM_UpDwn	Count Down
Counter_Period	uint8	9

The Count_Mode parameter uses the new type and value definitions. You can add your own values and Boolean logic to the parameters this way.

11. Set the Hardware flag to 'True' for all three new parameters, as Figure 53 shows.

Figure 53. Adding New Component Parameters



12. Click OK and save the changes to the component.
13. Right-click on an empty spot in the symbol schematic and generate the Verilog file for the new component symbol – leave all settings at the defaults.
14. Do a "Save All" to apply the changes.
15. Launch the Datapath Configuration Tool and open the *UpDwnCntrPwm_v1_0.v* file that you just generated.

Use the 8-bit PWM from the first example as a starting point for this new component, but you need two additional configuration states. The first two states are the same as in the simple 8-bit PWM. The second two states implement the up-count feature.

An XOR state is used to clear the count register. After the count counts up to the period value, it will XOR itself, an act that clears the count back to zero.

16. Create a new datapath configuration named "UpDwn". Use the 'cy_psoc3_dp8' type.
17. Configure the CFGRAM section to match the state diagram described earlier:

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

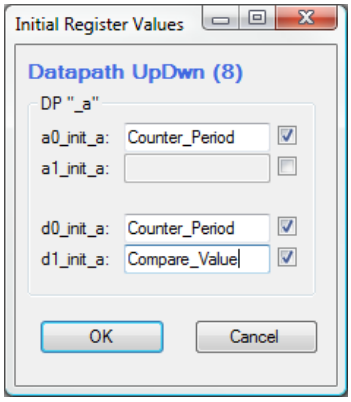
18. Set the CMP SELA field to 'A0_D1' to configure D1 as the compare value, just as in the simple 8-bit PWM.

Earlier, you added some parameters that the user could adjust. Enter the names of these parameters into the Initial Register Values fields so that you do not need to manually edit the Verilog to make changes.

For this example, you will use “Counter_Period” and “Compare_Value” for the parameter names, just as you did in the 8-bit PWM.

19. Open the *Initial Register Values* window and add parameter names into A0, D0, and D1, as Figure 54 shows.

Figure 54. Using Parameters for Initial Register Values



20. Click ‘OK’ and save changes to apply them to the Verilog file.

Now you need to add some Verilog code to the component. The previous examples required very little code, but implementing these features requires more.

21. Open the *UpDwnCntrPwm_v1_0.v* file and delete the text that says:

```
parameter Compare_Value = 4;
parameter Count_Mode = 0;
parameter Counter_Period = 9;
```

22. Locate the text that says:

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

23. Replace that text with:

```
// Placeholders for the parameter values
// passed from the component interface.
parameter [7:0]  Counter_Period = 8'd0;
parameter [7:0]  Compare_Value  = 8'd0;
parameter [7:0]  Count_Mode     = 8'd0;

// Control Register "pwmCntlReg" bits
localparam CNTL_CNT_UP_DWN = 0;

// Compare0 less than signal
wire up_reload;

// 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 )
: ( tc );

// 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)
);
```

24. Make the following links in the datapath logic:

- Change `.clk()` to `.clk(clk)`.
- Change `.cs_addr(3'b0)` to `.cs_addr({1'b0, upDwn, reload})`.
- Change `.cl0()` to `.cl0(up_reload)`.
- Change `.z0()` to `.zo(tc)`.
- Change `.cl1()` to `.cl1(cmp)`.

Now that you have made these changes, let’s discuss them briefly so you understand what is going on. First, you added a control register (cy_psoc3_control) that allows us to change the count direction via code during run-time. For more information on this control register and the different options, consult the *CAG*.

If you want to control the direction from main code, add a define for the control register in your header file. See the attached projects as an example.

Next, you will notice that the .cs_addr now has 2 bits of control instead of 1. In the previous examples, the datapath had only two states, so 1 bit of control was sufficient. Now that you have four states, you need 2 bits of control.

In addition, notice that instead of using the ‘tc’ signal to control the datapath state, you are using a signal ‘reload’. You need to do this, because now that you can count up, you can’t just use the ZDET to indicate the end of a period. You need to use the cl0 comparison. Therefore, when the counter is configured as an Up counter, it will continue to count up as long as the value in A0 is less than the period value stored in D0; when the counter is no longer less, it will trigger the reload state of the datapath.

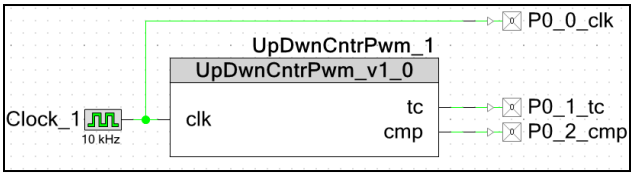
The ‘upDwn’ signal controls whether you are doing Up counting operations (INC, XOR) or Down counting operations (DEC, Load)

25. Save all changes to the Verilog file.

After you make these changes to the Verilog file, the component is ready to be used in your project. You will add all the same components as in the first example project.

26. Add the *MyLibrary* library as a dependency of the UpDwnCntrPwm project.
27. Drag an UpDwnCntrPwm component to the project schematic.
28. Connect a clock to the 'clk' terminal of the component and set it to 10 kHz.
29. Connect digital output pins to the component's terminals – P0_0_clk, P0_1_tc, and P0_2_cmp – as Figure 55 shows.

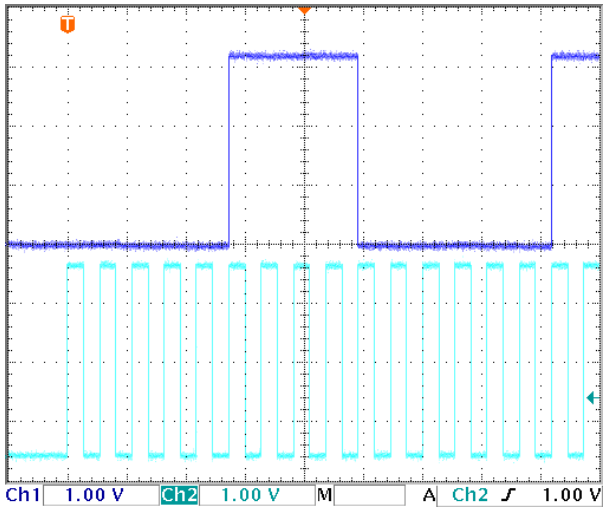
Figure 55. Up/Down Counter PWM Project Schematic



30. Do a "Save All", compile the code, and program the PSoC.

You set the default compare value parameter to be 4 and the period to be 9. This can be observed using the pins you added to the project. The first period is one cycle short because the count was decremented at the first clock tick, as Figure 56 shows.

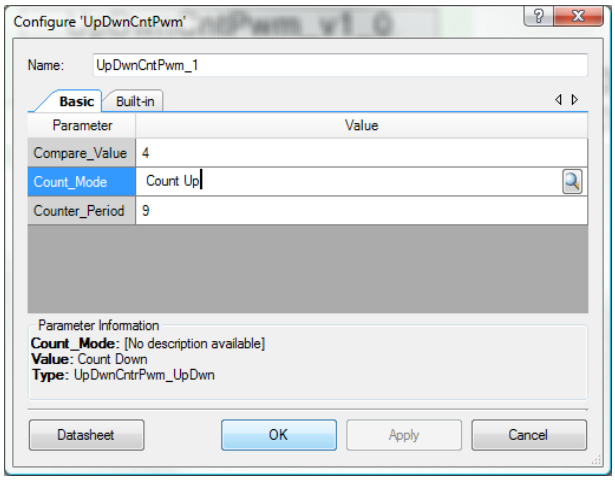
Figure 56. Down Counting PWM Waveforms



You can change the period and compare parameters just as you did in the simple PWM example. You can also change the mode parameter so that the PWM counts up instead of down.

31. Go back to the project schematic and double-click on the UpDwnCntrPwm component to open the properties.
32. Change the count mode to 'Count Up', as Figure 57 shows.

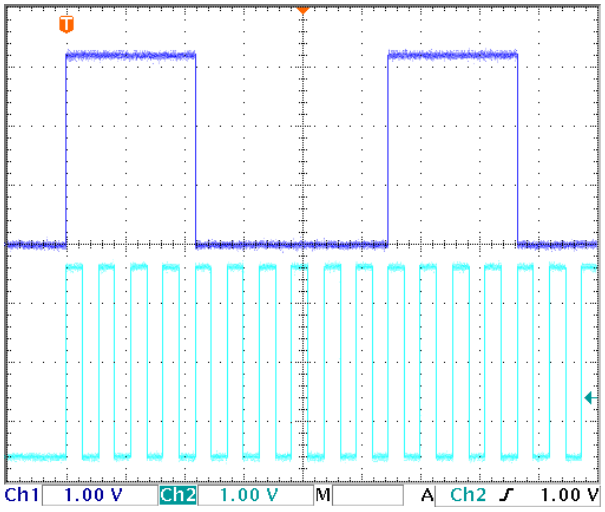
Figure 57. Setting the PWM to Count Up



33. Click OK to apply the changes.
34. Do a "Save All", compile the code, and program the PSoC.

The only way to observe the difference between the Up and Down count is to look at the first period after the PWM starts, as Figure 58 shows.

Figure 58. Up Counting PWM Waveforms



Notice that the output was HIGH at the beginning of the Up mode because the count value started out less than the compare value and was incremented. The output was LOW at the beginning of the Down mode because the count value started off greater than the compare value and was decremented.

Project #4 – Simple UART

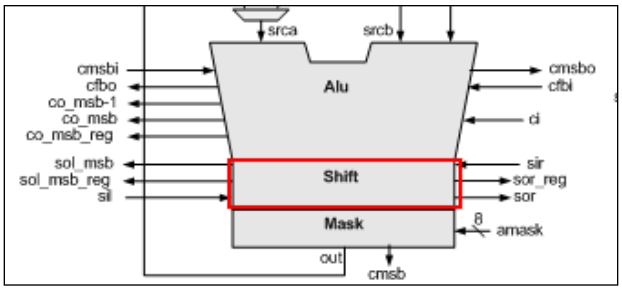
This example project demonstrates a simple TX UART created with a single datapath. We will not walk you through each step of creating the Component. Instead, you can review the component and Verilog file found in the associated example projects. Find the component, called "*Simple_Tx*", in the *MyLibrary* project of the completed examples. An example of how this component is used is included in the same workspace, in the project "*SimpleTx*".

TX UART Component Details

The datapath usage in this component is simple. The only datapath operations used are shifting, and loading a value into A0 from F0.

So far, we have not discussed shifting in any detail. At the output of the ALU is a shifter, as Figure 59 shows.

Figure 59. Shifter Block



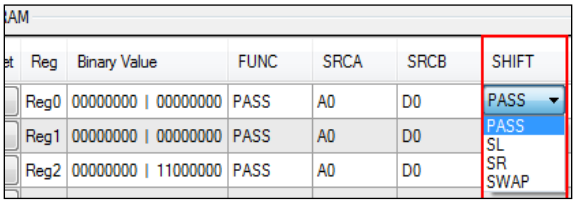
This shifter can shift bits either left or right, and it has the ability to swap nibbles, as Figure 60 shows.

Figure 60. Shift Functions



Each configuration of the datapath can independently set the operation of the Shifter. This option is set in the dynamic configuration area of the datapath configuration tool, as Figure 61 shows.

Figure 61. Shift Operation Settings



Also, you must take into account that there is only one Shift Out (SO) output on the datapath output mux. This output is shared between the Shift Out Right and the Shift Out Left. Therefore, you must configure this mux properly. This is done in the Static Configuration area under CFG15-14 *SHIFT SEL*, as Figure 62 and Figure 63 show.

Figure 62. SHIFT SEL Configuration

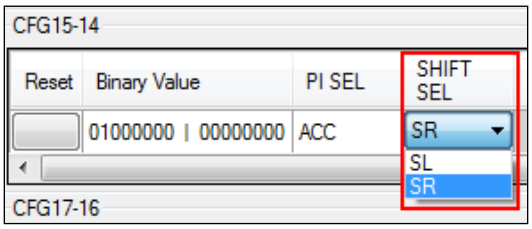
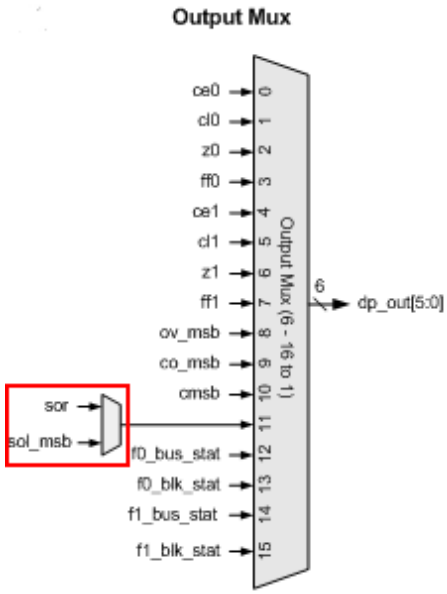


Figure 63. Shift Out Mux Diagram



In this example you do something you have not done in the previous examples: create a Verilog state machine that controls the operation of the datapath. This state machine, which is implemented in the UDB PLDs, determines which part of the UART transmission needs to occur next, such as Start, Data, or Stop.

If you look at the Verilog code, you will notice a line of code that looks like the following:

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

If you look at the Datapath inputs and outputs, you will notice that *state* is used to control the CS_Addr:

```
/*    input      [02:00]                */
.cs_addr({1'b0, state[1:0]}),
```

The value of state is changed by the Verilog state machine implemented with a switch case statement.

```
case (state)

        STATE_IDLE:…

        STATE_START:…
```

```
        STATE_DATA:…
        STATE_STOP:…
endcase
```

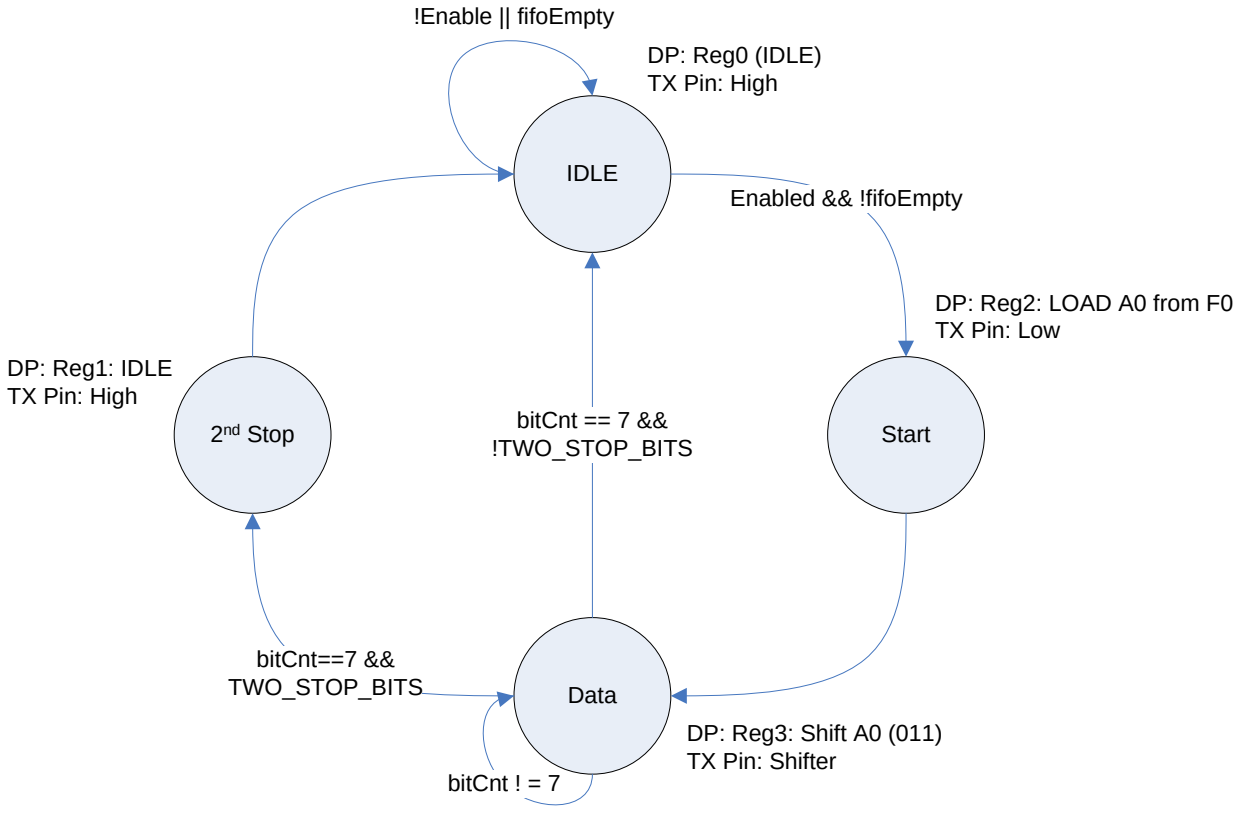
Notice that each of these cases is defined as the following:

```
// Main State machine states

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

Figure 64 shows how the simple TX Verilog state machine works.

Figure 64. TX UART Verilog Flowchart



First, the state machine waits for new data to be written to the FIFO, by monitoring the fifoEmpty status bit. After it has new data, it sends a START bit by setting the TX line LOW. During this state, the datapath loads the value in F0 into A0.

The state machine has only four states. Again, these four states are used to control the operation of the datapath. Thus the datapath needs four unique operations:

Reg	FUNC	SRCA	SHIFT	A0 WR SRC
Reg0	Pass	A0	None	None
Reg1	Pass	A0	None	None
Reg2	Pass	A0	None	F0
Reg3	Pass	A0	SR	ALU

As the code moves through the Verilog state machine, it changes the datapath operation. This is a common use case.

Most complex datapath components need a Verilog state machine to sequence the datapath operations.

In the next state the datapath shifts out the data in A0. After it is done, it sends out a STOP bit. It can either send 1 or 2 stop bits.

You may be wondering how you control the TX output line. As stated above, when you are in the data state, you shift data out. The Start state pulls the TX line LOW, and the stop and IDLE state set it HIGH. This is done through the following Verilog code.

```
// 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] );
```

If it is not in the data state, it drives the inverse of the MSBit of ‘state’ on the TX line. For Start, the MSBit is 1, so it outputs a ‘0’. For Stop and Idle, the MSBit is 0, so it outputs a ‘1’.

Summary

Any embedded system designer looking to shave a few precious microseconds off code execution time will appreciate the efficiency that datapath-based components can bring to a project. Effectively using the UDB datapaths will allow you to extend the capabilities of PSoC 3 and PSoC 5 beyond even what traditional microprocessors can offer.

The example projects described in this application note are just the starting point for you to create your own custom solutions. The PSoC 3 Architecture TRM and the CAG have more information on how to add features and complexity to your components.

Related Application Notes

AN54181 – Getting Started with PSoC 3

AN77759 – Getting Started with PSoC 5

AN62083 – PSoC 3 to PSoC 5 Migration Guide

AN81623 – PSoC 3 and PSoC 5 Digital Design Best Practices

AN82250 – PSoC 3 and PSoC 5 Implementing Programmable Logic Designs

Related Training Material

PSoC Creator 210: Intro to Datapath Components

PSoC Creator 211: Datapath Computation

PSoC Creator 212: Datapath FIFOs

PSoC Creator 213: Multi-Byte Datapath Components

PSoC Creator 214: Datapath API Generation

About the Authors

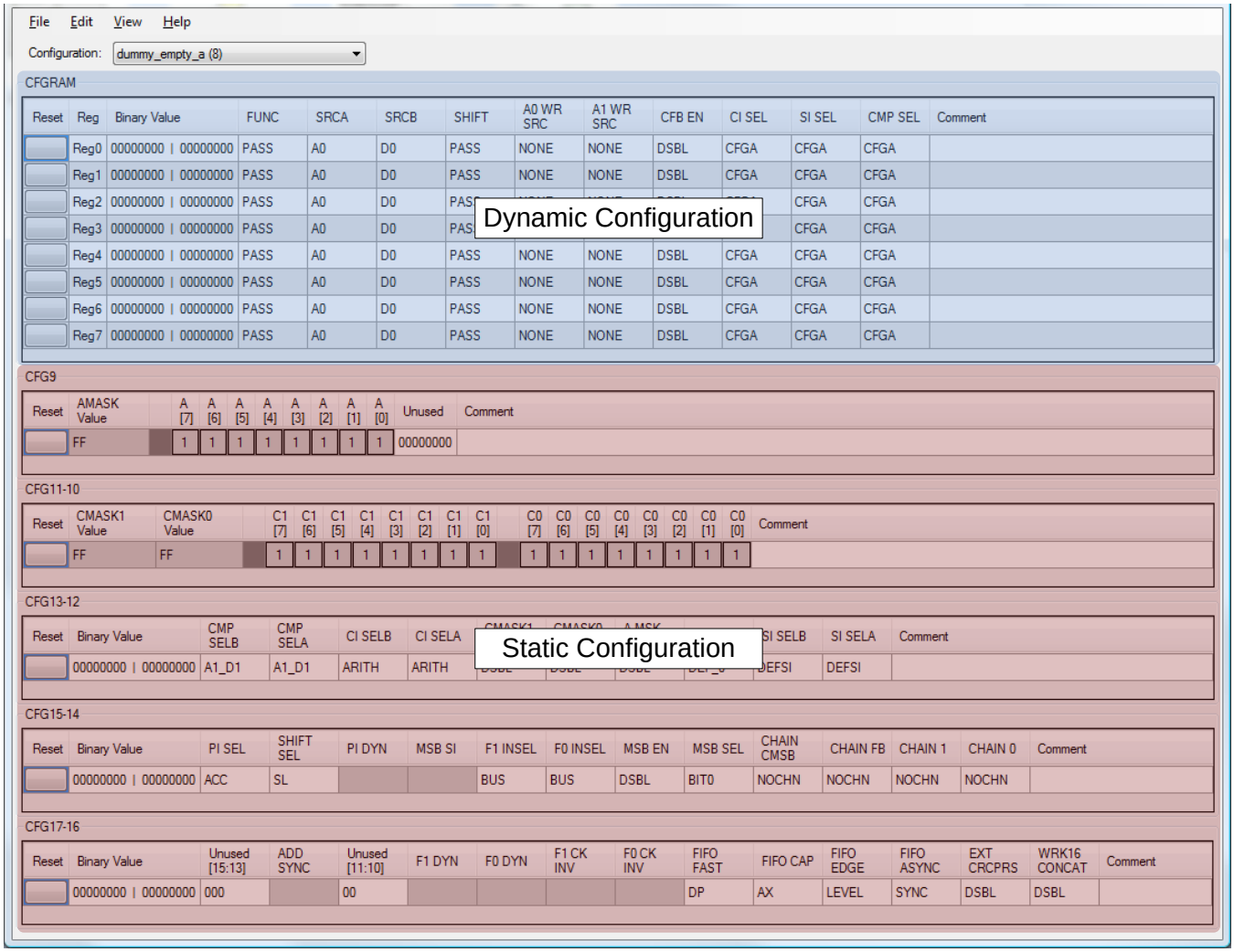
Name:	Greg Reynolds
Title:	Applications Engineer Staff
Background:	Greg Reynolds has been with Cypress in several roles for more than a decade, but he's sometimes convinced that it's been much, much longer. He always wanted to be a Chemical Engineer, but it wasn't until his second year of college that he realized how much he hated chemistry and loved electronics. He thinks it's for the best that he changed his major.
Name:	Mark Hastings
Title:	Applications Engineer MTS
Background:	Mark Hastings graduated from Washington State University in 1984 with a BSEE degree. For most of the past 28 years, he has been involved in embedded and mixed-signal designs. His free time is spent hiking and taking pictures in the Washington Cascades.
Name:	Todd Dust
Title:	Applications Engineer Staff
Background:	Todd graduated from Seattle Pacific University with a degree in Electrical Engineering. He has been with Cypress as an Applications Engineer ever since.

Appendix A – Datapath Configuration Tool Details

This appendix shows the relationship between the Datapath Configuration Tool GUI, the datapath block diagram, and the auto-generated Verilog code. The PSoC 3 TRM gives a complete description of the UDBs and their registers.

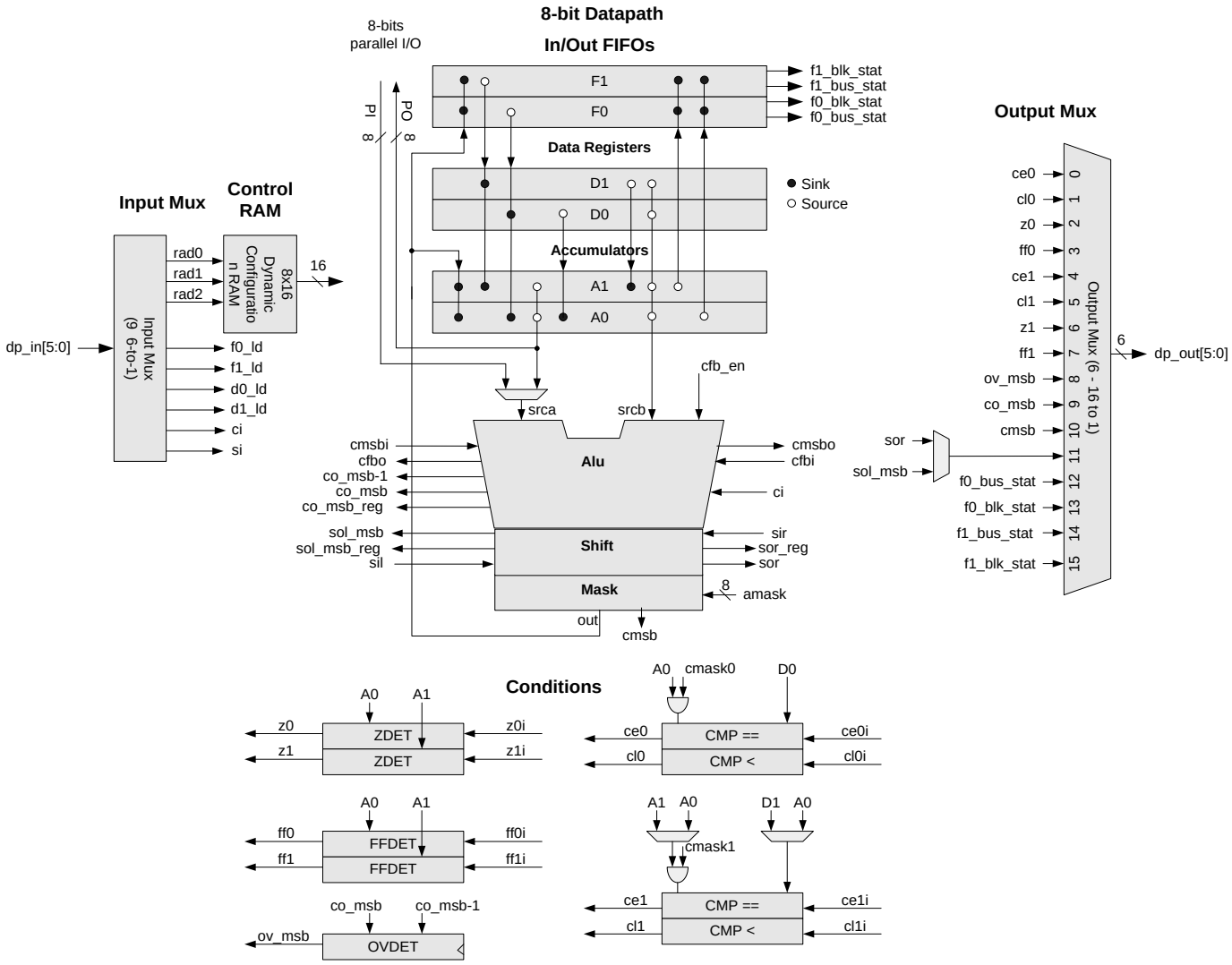
The Datapath Configuration Tool presents the datapath registers in a GUI. The GUI can be divided into two general sections – the Dynamic Configuration and Static Configuration sections, as Figure 65 shows.

Figure 65. Datapath Configuration Tool Interface Sections



The complete set of datapath features can be represented by a block diagram, as Figure 66 shows:

Figure 66. Detailed Datapath Block Diagram



Dynamic Configuration RAM Section

The Dynamic Configuration section is a representation of the configuration RAM. The tables below explain the function of each of the fields in the GUI.

Table 1. The CFGRAM Section of the Datapath Configuration Tool

Dynamic Configuration RAM Section
Datapath Configuration Tool

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

Datapath Block Diagram

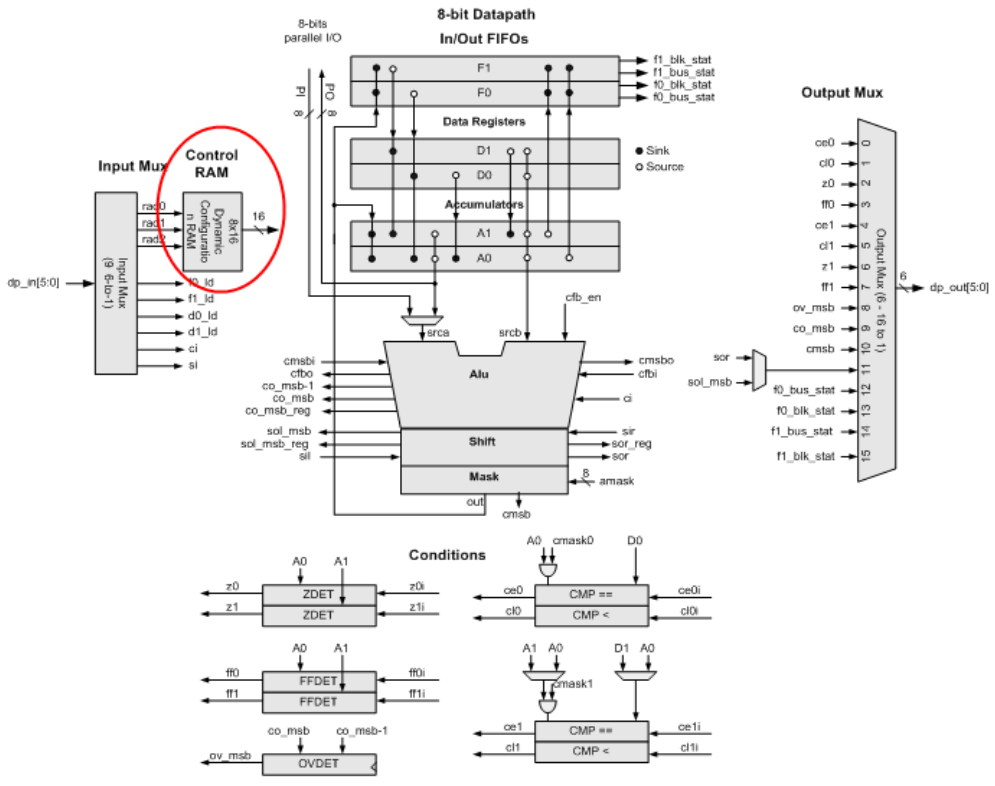
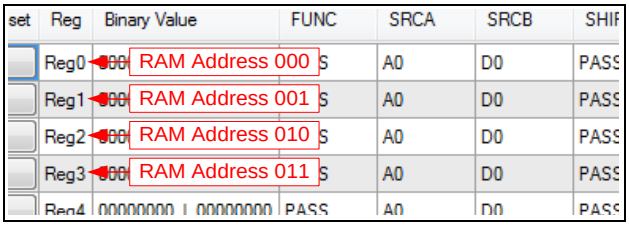
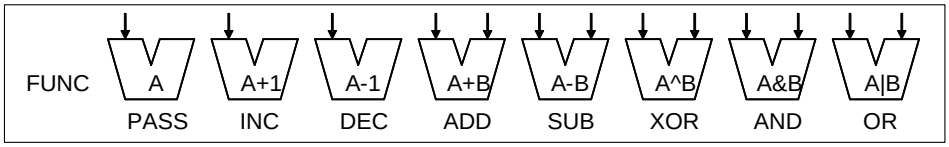
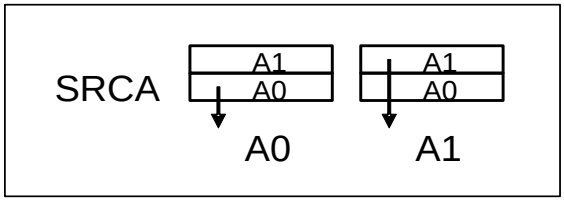
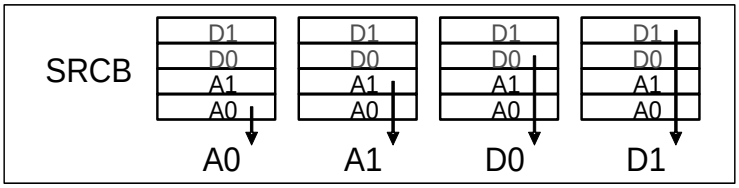
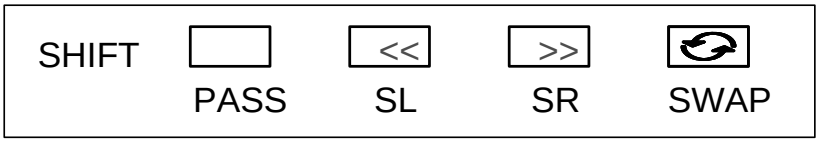
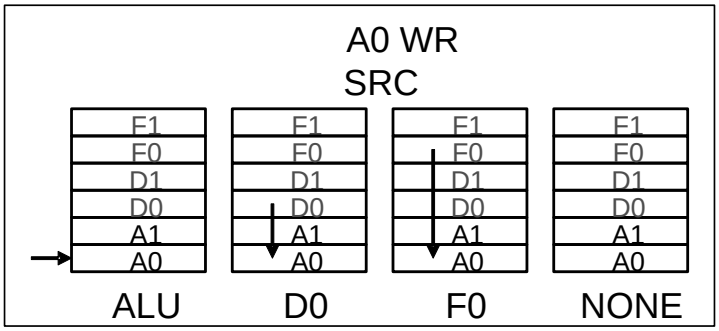
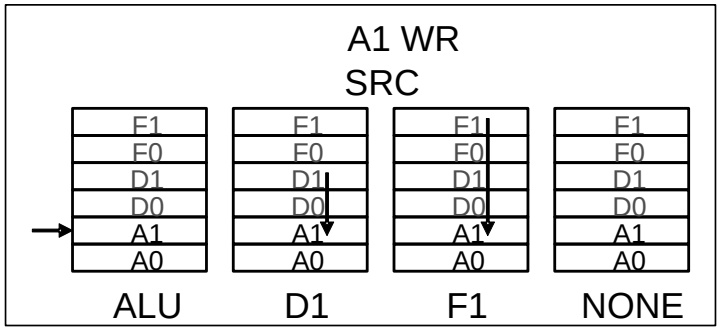
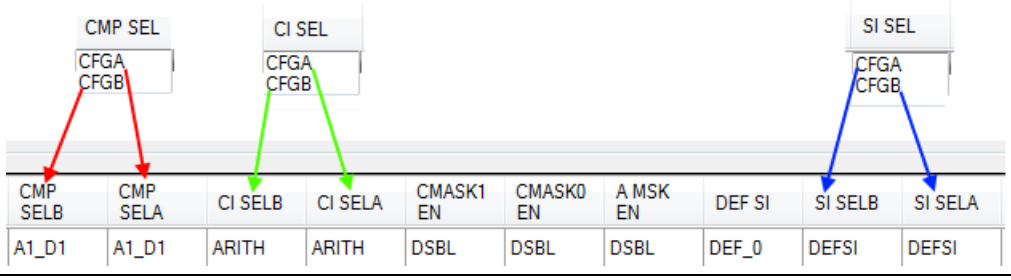


Table 2. Dynamic Configuration Section Column Descriptions

Dynamic Configuration Section	
Reg This column shows the configuration state to which the row corresponds. The value of the three CFGRAM address signals determines which state is selected, so take care to select the correct row for each configuration.	
FUNC This column determines which of the eight ALU functions will be performed in that state.	
SRCA This column determines the source for the ALU's 'srca' input.	
SRCB This column determines the source for the ALU's 'srcb' input.	
SHIFT This column determines the function of the shift block.	
A0 WR SRC This column determines the contents of the A0 register *after* the ALU operation is complete.	

<table>
<tr><th colspan="2">Dynamic Configuration Section</th></tr>
<tr><td>A1 WR SRC

This column determines the contents of the A1 register after the ALU operation is complete.</td><td>
</td></tr>
<tr><td>CI SEL, SI SEL, CMP SEL

Carry in Select, Shift In Select, and Compare Select. Each of these has two configuration options: A and B. You can choose which the one you want for each configuration</td><td>
</td></tr>
</table>

* CFB_EN is outside the scope of the application note and will not be discussed.

Static Configuration Section

The Static Configuration section represents the datapath registers CFG9 to CFG17, as Table 3, Table 4, Table 5, Table 6, and Table 7 show. They control static functions, including shift direction, masking, FIFO configuration, and chaining.

Table 3. CFG9 Register Definition

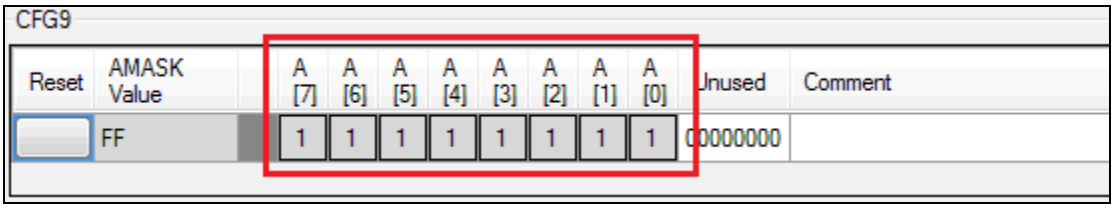
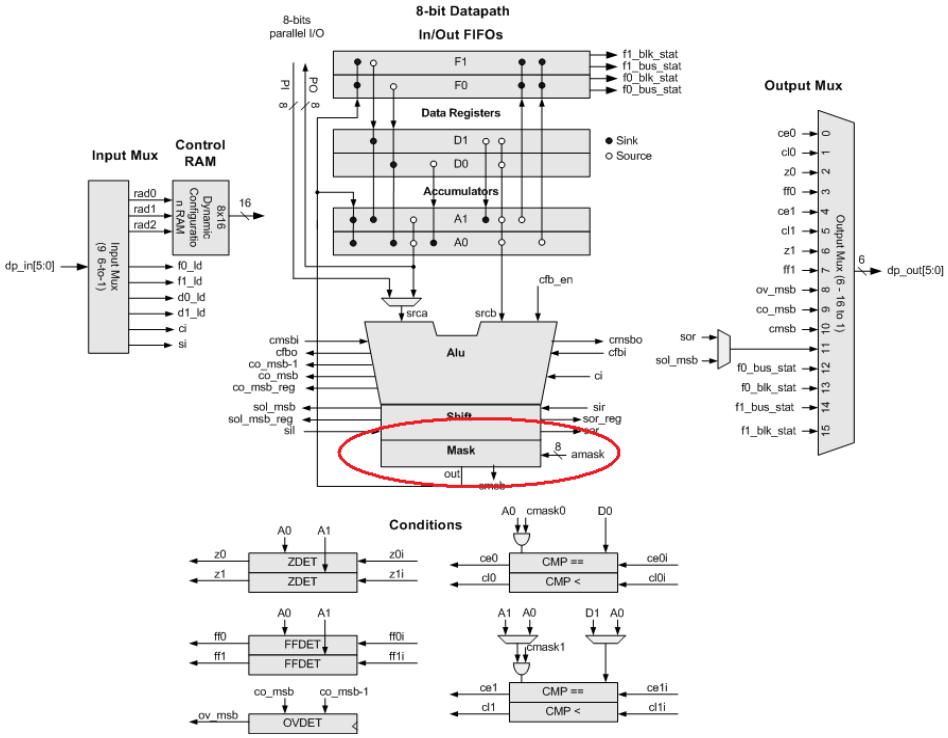
AMASK Value
Datapath Configuration Tool

■ AMASK Value – This field sets the value used in the MASK block of the ALU. The output of the shift register is ANDed with the contents of this register.
Datapath Block Diagram


Table 4. CFG11 and CFG10 Register Definitions

CMASK0 and CMASK1 Values
Datapath Configuration Tool

CFG11-10

Reset	CMASK1 Value	CMASK0 Value	C1 [7]	C1 [6]	C1 [5]	C1 [4]	C1 [3]	C1 [2]	C1 [1]	C1 [0]	C0 [7]	C0 [6]	C0 [5]	C0 [4]	C0 [3]	C0 [2]	C0 [1]	C0 [0]	Comment
	FF	FF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

- CMASK0 and CMASK1 – These fields set the mask values used with the comparator block inputs. The contents of the A0 or A1 register are ANDed with the contents of these registers before comparison.

Datapath Block Diagram

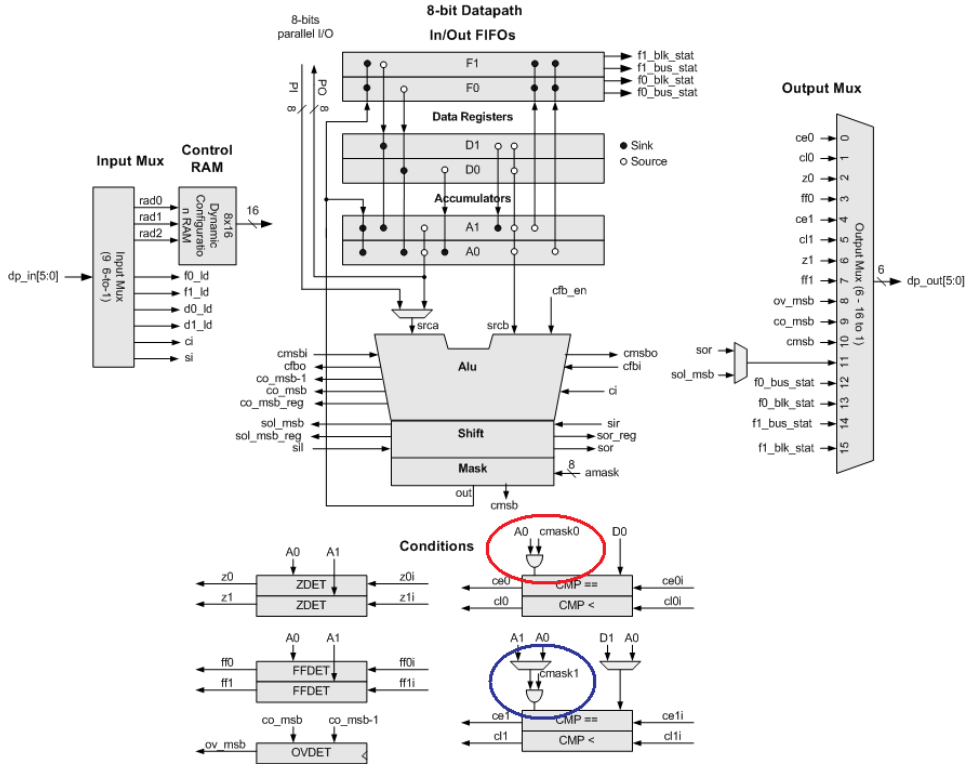


Table 5. CFG13 and CFG12 Register Definitions

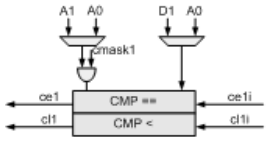
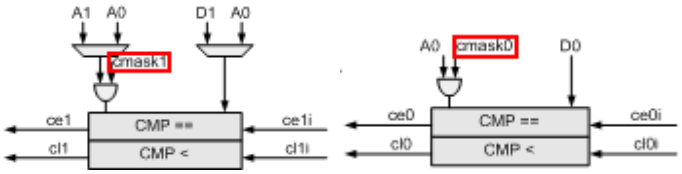
<table>
<tr><th colspan="2">CFG13-12 Configuration Selections</th></tr>
<tr><td colspan="2">Datapath Configuration Tool</td></tr>
<tr><td colspan="2">
CFG13-12
<table>
<tr><th>Reset</th><th>Binary Value</th><th>CMP SELB</th><th>CMP SELA</th><th>CI SELB</th><th>CI SELA</th><th>CMASK1 EN</th><th>CMASK0 EN</th><th>A MSK EN</th><th>DEF SI</th><th>SI SELB</th><th>SI SELA</th></tr>
<tr><td></td><td>00000000 | 00000000</td><td>A1_D1</td><td>A1_D1</td><td>ARITH</td><td>ARITH</td><td>DSBL</td><td>DSBL</td><td>DSBL</td><td>DEF_0</td><td>DEFSI</td><td>DEFSI</td></tr>
</table>
</td></tr>
<tr><th colspan="2">CFG13-12 Details</th></tr>
<tr><td>CMP SELB & CMP SEL A
Configures the Comparison B and Comparison A Options for compare block 1</td><td>A1_D1: A1 < D1 A1 == D1
A1_A0: A1 < A0 A1 == A0
A0_D1: A0 < D1 A0 == D1
A0_A0: A0 < A0 A0 == A0

</td></tr>
<tr><td>CI SELB & CI SEL A
Selects the source of the carry in</td><td>ARITH: The carry is controlled by ALU arithmetic
REGIS: The carry in is carry out registered from previous cycle
ROUTE: Carry in is selected from one of the datapath inputs
CHAIN: Carry in is driven from previous datapath in chain</td></tr>
<tr><td>CMASK0 EN & CMASK1 EN
Enables the Masks on the Compare 1 and Compare 0 Blocks</td><td>
</td></tr>
<tr><td>DEF SI
Defines whether the default shift in value is a 0 or a 1. Note SI SELB or SI SELA must be set to DEFSI for this to matter</td><td>DEF_0: Zero
DEF_1: One</td></tr>
<tr><td>SI SELB & SI SEL A
Selects the source of the shift in for the A and B shift in configuration</td><td>DEFSI: Shifts in either a zero or a one, as defined by DEF SI
REGIS: Shift in is shift out register from previous cycle
ROUTE: Shift in is selected from a datapath input
CHAIN: Shift in is driven by previous datapath in chain</td></tr>
</table>

Table 6. CFG15 and CFG14 Register Definitions

CFG15-14 Configuration Selections

Datapath Configuration Tool

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

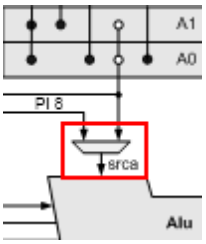
CFG15-14 Details

PI SEL

Controls the Mux Input for SRCA. The input can either be sourced by the SRCA option in Dynamic Config or from the parallel input

ACC: Source A is sourced by the selection in the dynamic configuration area

PIN: Source A is sourced by the parrell input to the datapath



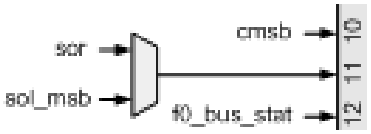
Shift SEL

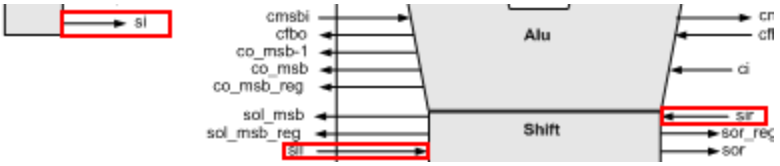
Controls whether the *si* signal is connected to *sil* or *sir* on the datapath

Also controls the selection of the shift out signal at the datapath output mux

SL: *si* connected to *sir, sol_msb chosen for output*

SR: *si* connected to *sil, sol_msb chosen for output*





F1 INSEL & F0 INSEL

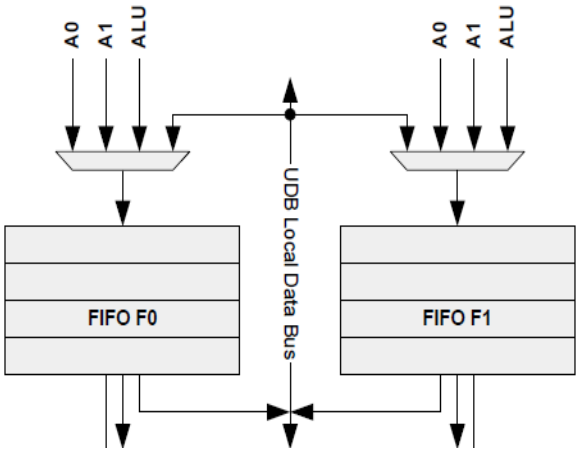
Defines the input source of FIFO 1 and FIFO 0.

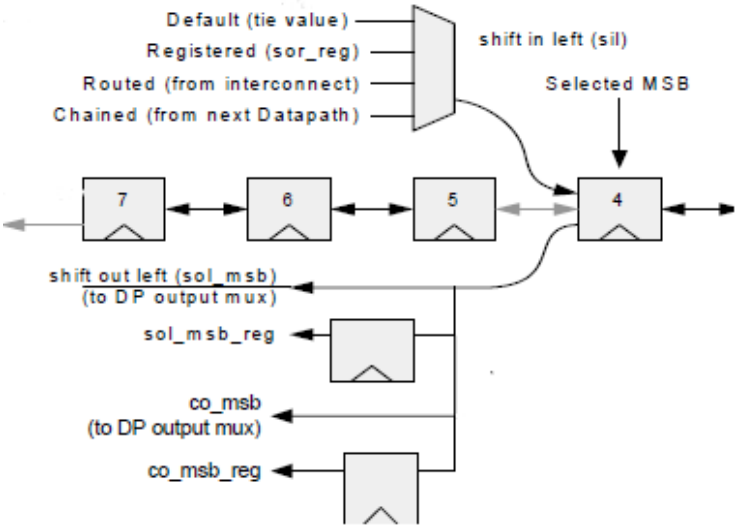
Bus: FIFO input is the CPU bus, FIFO output is A0 or D0 Registers

A0: FIFO input is A0. FIFO Output is CPU BUS.

A1: FIFO input is A1. FIFO Output is CPU BUS.

ALU: FIFO input is the ALU. FIFO Output is CPU BUS.



CFG15-14 Configuration Selections	
MSB EN and MSB SEL The MSbit of the ALU output can be adjusted. These two settings allow you to disable and enable this feature and choose which bit is the MSbit.	When the MSbit is changed to anything other than bit 7, the shift in, shift out, and carry out outputs all change accordingly 
Chain 1 & Chain 0 Defines whether the outputs of CL0, CL1, CE0, CE1, Z0, Z1, FF0, and FF1 are chained	When set to chained, the conditions from the previous datapath are chained to this datapath.

*Note PI DYN, MSB SI, CHAIN CMSB, and CHAIN FB are beyond the scope of this application note. Information on these settings can be found in the TRM.

Table 7. CFG17 and CFG16 Register Definitions

FIFO Configurations

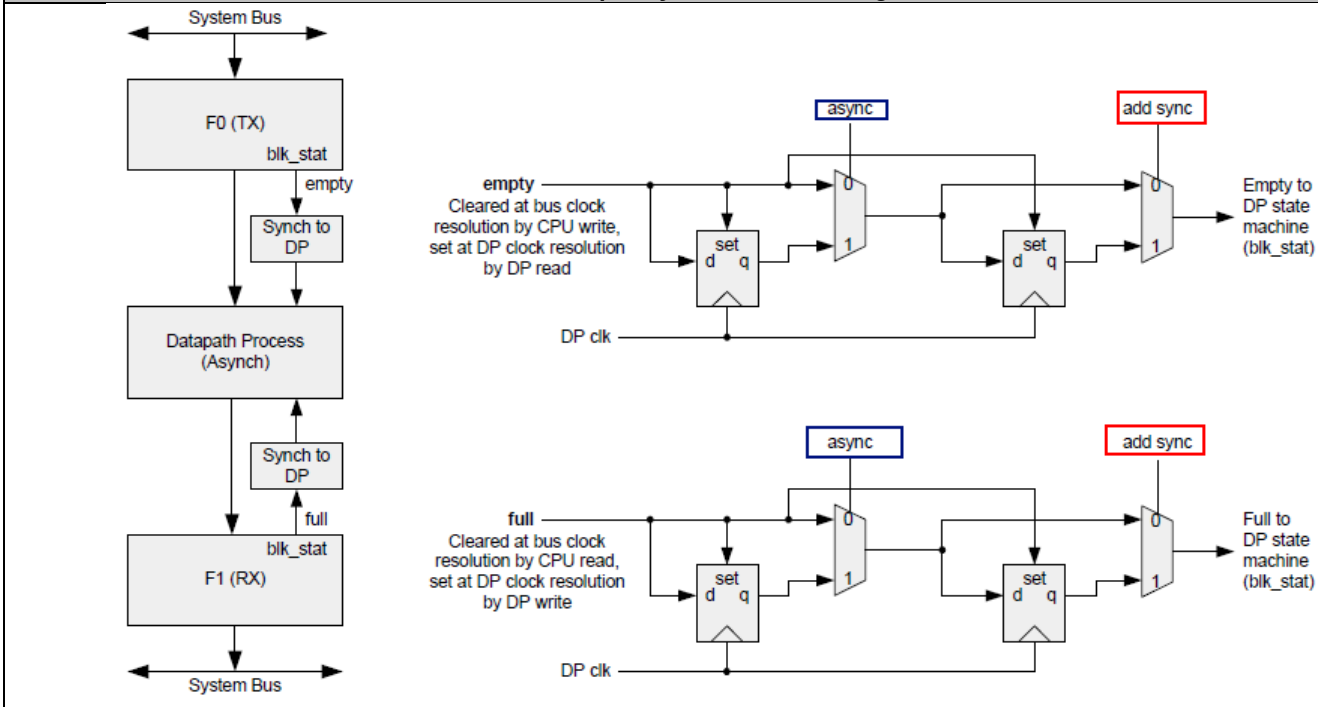
Datapath Configuration Tool

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 – This field determines whether an additional sync flip-flop is added to the FIFO block status.
- F0 CLK INV and F1 CLK INV – These fields determine whether the FIFO clock is inverted relative to the datapath clock.
- FIFO FAST – This field determines whether the FIFOs are clocked using the datapath clock or the PSoC Bus Clock.
  DP: Datapath Clock
  BUS: Bus Clock
- FIFO CAP – This field enables FIFO capture mode. If enabled, a read of A0 or A1 will write into F0 or F1, respectively. When set to AX, this is enabled.
- FIFO EDGE – This field determines whether FIFO writes occur on a LOW to HIGH transition. Or, if they can occur at any time, the F0 or F1 load signal is HIGH.
- FIFO ASYNC – This field determines whether a flip-flop is needed on the output of the FIFO block status signals.

FIFO Output Synchronization Diagram



*Note F1 DYN, F0 DYN, EXT CRCPRS, and WRK16CONCAT are outside the scope of this application note

Appendix B – Auto-Generated Verilog Code

This appendix provides examples of the Verilog code generated by PSoC Creator and the Datapath Configuration Tool. The code shown here was copied during the creation of the component from Example Project #1. The PSoC Creator projects associated with this application note contain completed example projects with full comments. The code shown here is used only to demonstrate the evolution of the Verilog code as you use PSoC Creator and the Datapath Configuration Tool.

A New Verilog File Generated by PSoC Creator

When a Verilog file is first generated by PSoC Creator, it looks a lot like this:

```
//`#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
```

In this case, the ‘tc’ and ‘clk’ pins have been added because they were present in the SimpleCntr8 component symbol. If your component symbol did not contain any terminals, then these lines of code would be omitted.

The Verilog File with a New Datapath Instance

When the Datapath Configuration Tool generates code for a datapath configuration, the code is appended to an existing component Verilog file. An unmodified configuration looks similar to this:

```
//`#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
```

Notice that the sections for the Configuration RAM and the hardware links have been added, but they do not contain any real configuration data. This datapath component would not do anything useful. Table 8 explains the hardware links found in the datapath configuration.

Table 8. Verilog Hardware Link Definitions

Verilog Code	Definition / Mapping
.reset(1'b0),	Datapath reset signal.
.clk(1'b0),	Datapath clock input.
.cs_addr(3'b0),	Configuration store address signals.
.route_si(1'b0),	Routed Shift In
.route_ci(1'b0),	Routed Carry In
.f0_load(1'b0),	FIFO 0 Load Signal
.f1_load(1'b0),	FIFO 1 Load Signal
.d0_load(1'b0),	D0 Load Signal
.d1_load(1'b0),	D1 Load Signal
.ce0(),	Compare block 0 “equals” output.
.cl0(),	Compare block 0 “less than” output.
.z0(),	Zero detect block 0 output.
.ff0(),	All ones detect block 0 output

Verilog Code	Definition / Mapping
.ce1(),	Compare block 1 “equals” output.
.cl1(),	Compare block 1 “less than” output.
.z1(),	Zero detect block 1 output.
.ff1(),	All ones detect block 1 output
.ov_msb(),	Overflow detect
.co_msb(),	Carry Out
.cmsb(),	Not Covered in this AN
.so(),	Shift Out
.f0_bus_stat(),	FIFO 0 bus status flags*.
.f0_blk_stat(),	FIFO 0 block status flags*.
.f1_bus_stat(),	FIFO 0 bus status flags*.
.f1_blk_stat()	FIFO 0 block status flags*.

*For More information on these status outputs consult the TRM

These links must be made for the datapath to function. Without them, there is no correlation between the component symbol, the Verilog code, and the datapath hardware.

The Verilog File after All Modifications

This is the finished Verilog for the simple 8-bit counter:

```
//`#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),
        /*  input   [02:00]         */  .cs_addr({2'b0,tc}),
        /*  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),
        /*  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
```

The two configuration states that decrement and reload A0 have been configured, and the initial register values for A0 and D0 are defined. The links between the hardware and the symbol terminals have also been established. You could create this file from scratch, but it is much easier to use the Datapath Configuration Tool.

Document History

Document Title: PSoC® 3 and PSoC 5 – Designing PSoC Creator™ Components with UDB Datapaths

Document Number: 001-82156

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3756721	GIR	09/27/2012	New document.
*A	3776358	GIR	10/19/2012	1. Corrected resource usage table names on page 2. 2. Replaced text with Figure 3 for improved clarity. 3. Minor clarifications to example project steps. 4. Added hyperlinks for AN82250. 5. Added hyperlinks for datapath videos. 6. Removed some related application note references. 7. Replaced Figure 1 and Figure 38 to match related documents. 8. Updated Figure 4, Figure 5, Figure 13, Figure 14, Figure 23, Figure 24, Figure 27, Figure 31, Figure 40, Figure 47, Figure 51, and Figure 55 to correct the default component color. 9. Minor formatting corrections to accommodate other changes.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

PSoC 3, PSoC 5, and PSoC Creator are registered trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.
Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.