Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN82156

Designing PSoC Creator Components with UDB Datapaths

Author: Todd Dust, and Greg Reynolds

Associated Project: Yes

Associated Part Family: CY8C3xxx, CY8C5xxx, CY8C42xx, CY8C6xxx

Software Version: PSoC® Creator™ 4.2

For a complete list of related material, click here.

AN82156 explains how to design PSoC® Creator™ Components that use PSoC Universal Digital Block (UDB) datapaths. Datapath-based Components can implement common functions such as counters, PWMs, Shifters, UARTs, and SPI. More importantly datapaths can be used to create custom digital peripherals, and to perform data management tasks to offload the CPU. This application note describes how to use the UDB Editor to create custom datapath components.

Contents

1 Introduction

PSoC® 3, PSoC 4, PSoC 5LP, and PSoC 6 MCU (hereafter referred to as PSoC) are more than just microcontrollers. With PSoC you can integrate the functions of a microcontroller, complex programmable logic device (CPLD), and high-performance analog with unmatched flexibility. This saves cost, board space, power, and development time.

Porting a CPLD design from a standalone CPLD to PSoC PLDs can be as easy as copying and pasting Verilog code. AN82250 provides excellent step-by-step instructions on how to do this.

However, PSoC's PLDs are smaller than most CPLDs or FPGAs, and many designs are too large to port directly to PSoC. To overcome this obstacle, Universal Digital Block (UDB) datapaths can be used. Datapaths are designed to implement computationally complex functions such as adding, subtracting, shifting, and so on. The theory is that pieces of the CPLD design get ported to UDBs so that the CPLD design can be ported to PSoC. This application note teaches about the datapath and how to create designs with the datapath.

At the heart of the datapath is an 8-bit arithmetic logic unit (ALU). This ALU performs functions such as add, subtract, OR, XOR, AND, increment, decrement, and shift. Associated with this ALU are several registers and several conditional comparison blocks. Datapaths can be chained to perform operations of any bit width between 1 and 32 bits.

Using the datapath in combination with PLDs allows you to create complex custom digital peripherals. These peripherals are captured in PSoC Creator Components. Today, PSoC Creator has a rich set of digital Components that use UDB datapaths. However, it may be that the functionality you are looking for is not available in a standard Component. This application note shows you how to create your own Components that use the datapath. This application note primarily fouces on using the UDB editor to create custom compoennts.

This is an advanced application note—it assumes that you are familiar with developing applications using PSoC Creator.

If you are new to PSoC, see introductions in:

- AN54181 Getting Started with PSoC 3
- AN79953 Getting Started with PSoC 4
- AN77759 Getting Started with PSoC 5LP
- AN221774 Getting Started with PSoC 6 MCUs

If you are new to PSoC Creator, see:

- PSoC Creator home page

In addition, this application note assumes a basic understanding of digital design and Verilog. If you are new to these concepts, see:

- AN81623 PSoC Digital Design Best Practices
- KBA86336 Just Enough Verilog for PSoC
- AN82250 Implementing Programmable Logic Designs with Verilog.

This application note also assumes you are familiar with the UDB editor and how it works, for more information see

- Universal Digital Block (UDB) Editor Guide

For a list of related datapath design resources, see the Related Resources section.

If you are familiar with Datapaths but want to determine if you should use the UDB Editor or the datapath configuration tool jump to Datapath-Based Components.

If you known you want to use the UDB Editor jump to Project #1 – 8-Bit Down Counter.

If you know you want to use the datapath configuration tool, or need to use the parallel in/out jump to Appendix A – Examples with Datapath Configuration Tool.

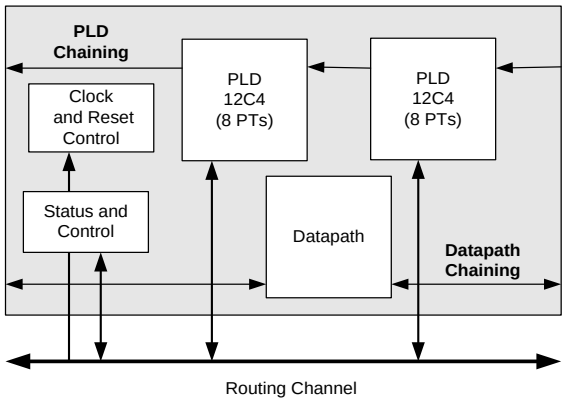
Otherwise continue reading.

2 Traditional PLDs Versus PSoC UDB

PSoC is an optimized programmable device that can match or exceed the functionality of much larger programmable logic products. PSoC is not designed to directly integrate a large FPGA or CPLD implementation. Instead, PSoC contains an array of small, fast, low-power programmable digital blocks, or UDBs.

Each UDB is made up of two small PLDs, a datapath module, and status and control logic, as Figure 1 shows.

Figure 1. Simplified UDB Block Diagram



A datapath module can perform functions such as increment, decrement, add, subtract, bitwise logic, and shift. Paired with the PLDs, datapaths can be used for more complex functions. This combination can easily implement often-used functions such as counters, PWMs, shifters, UARTs, or I^2C interfaces.

This combination can also be used to implement pieces of a CPLD or FPGA design that won't fit in PSoC PLDs. Complex functions written in Verilog can be optimized for the datapaths for a more efficient use of PSoC digital resources. This includes functions such as adders, subtractors, shifters, etc. The datapath can implement these much more efficiently than the PLDs.

3 Datapath Versus PLD-Based Designs

Functions implemented in UDBs typically require fewer resources if the datapath is used to perform arithmetic operations instead of PLDs. For example, consider the following 8-bit arithmetic and logic operations implemented in PLDs versus datapaths, as Table 1 shows.

Table 1. PLD vs Datapath Resource Usages

Function	Resource Consumption in PLDs Only		Resource Consumption in Datapaths Only	
	PLDs	% Used[1]	Datapath	% Used[1]
ADD8	5	10.4%	1	4.2%
SUB8	5	10.4%	1	4.2%
CMP8	3	6.3%	1	4.2%
SHIFT8	3	6.3%	1	4.2%

[1] This is based off usage in a PSoC 5 LP device.

Datapaths should be used along with PLDs to implement digital functions in the most efficient way.

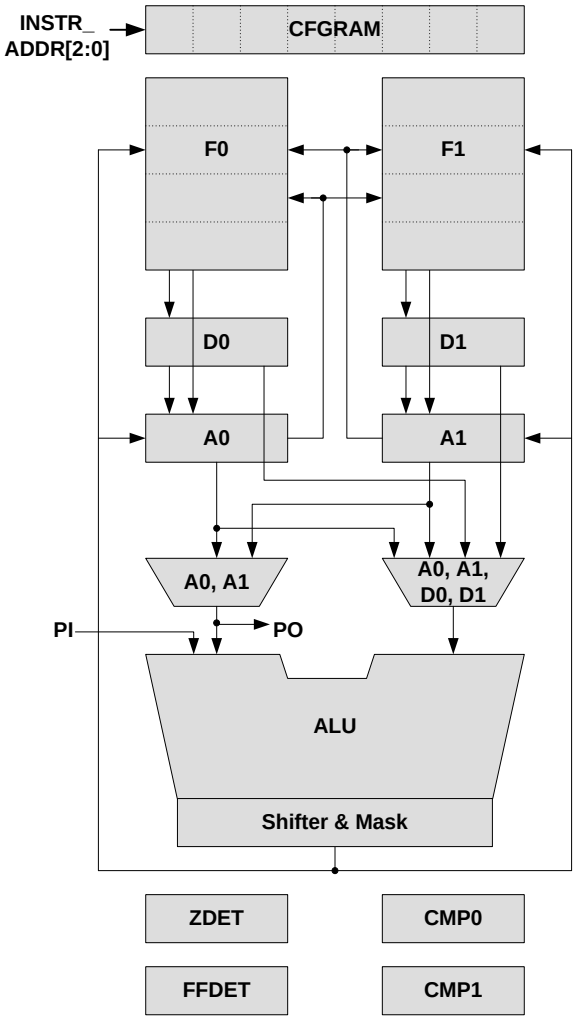
As a rule of thumb, here is the best way to utilize UDB resources:

- PLDs: Combinatorial logic, glue logic, state machines. (See AN82250 PSoC 3, PSoC 4 and PSoC 5 Implementing Programmable Logic Designs, and AN81623 PSoC 3, PSoC 4 and PSoC 5 Digital Design Best Practices, for more information on PLD implementations.)
- Datapaths: FIFO interface with the CPU, calculations, timing, communications, and byte- or word-wide comparisons.

4 Datapath Architecture and Features

The datapath contains a configurable 8-bit ALU with associated compare- and condition-generation circuits, and registers for ALU manipulation and CPU interaction, as Figure 2 shows. There are also independent blocks for shifting and masking.

Figure 2. Simplified Datapath Block Diagram



See Appendix B for a detailed block diagram.

4.1 Dynamic Configuration RAM (CFGRAM)

You can store eight unique datapath instructions (or configurations) in the dynamic configuration RAM (CFGRAM). The instructions define the ALU function, ALU inputs, register writes, shift operation, comparison operation, etc. Each instruction executes in one clock cycle.

Because there are eight unique instructions, there are three instruction address lines (INSTR_ADDR[2:0]). The address signals determine which instruction is used on each rising edge of the datapath clock.

These three address lines can be driven by PLD logic or by external signals. A common use case is to create a state machine with PLDs. Logic from the state machine is then routed to these address inputs to control datapath instructions.

These three address lines have multiple names, which can lead to confusion. Note that cs_addr[2:0], RAD[2:0], and INSTR_ADDR[2:0] are all the same thing. This Application note will use the name INSTR_ADDR.

4.2 ALU

The ALU can perform eight general-purpose functions:

- increment
- decrement
- add
- subtract
- bitwise AND
- bitwise OR
- bitwise XOR
- pass through

Function selection is controlled by the instruction stored in CFGRAM, on a cycle-by-cycle basis. Independent shift (left and right) and masking operations are available at the output of the ALU.

4.3 Registers

Each datapath module has four 8-bit working registers and two FIFOs:

- A0 and A1 – The accumulator registers typically hold data used in ALU operations. 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 or reload values for a counter.
- F0 and F1 – These registers are 4-byte FIFOs that you can use as both source and destination buffers. They are primarily used to interface between the CPU or DMA and the datapath.

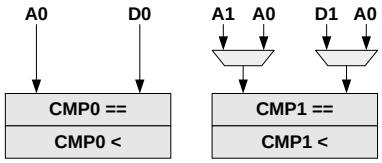
The instruction stored in the CFGRAM determines how these registers are used during each instruction.

You can read or write any datapath register using the CPU (and DMA in parts with DMA), but use the FIFOs whenever possible. The accumulator registers operate asynchronously and may change at any time, including during CPU or DMA accesses. The FIFOs are synchronized for CPU/DMA access.

4.4 Conditional 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. Datapath Comparison Blocks



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

4.5 Inputs and Outputs

UDBs are surrounded by the Digital Signal Interconnect (DSI), an extensive fabric of programmable digital routing. The DSI connects signals within a UDB, and between the UDB array and other blocks in PSoC.

There are three types of datapath inputs: instruction, control, and data. The instruction inputs (INSTR_ADDR[2:0]) select the current datapath instruction from the CFGRAM. The control inputs load the data registers from FIFOs and capture accumulator outputs into the FIFOs. Data inputs include shift in(SI) and carry in(CI). The datapath has a maximum of six inputs. These six inputs can come from anywhere on the chip that has a connection to the DSI.

The datapath has a maximum of six outputs. These outputs can connect to a variety of datapath status signals including FIFO status, comparison status, overflow detect, carry out, and shift out. These outputs then connect to the DSI where they can be routed to other on-chip resources.

4.6 Chaining

Each datapath can perform 8-bit operations. You can chain multiple datapaths to create functions that are of any bit width from 1 to 32 bits wide.

The shift, carry, capture, and other conditional signals can be chained to form higher-precision arithmetic and shift functions. These chained signals don't consume datapath inputs and outputs.

See the PSoC 3 Architecture TRM for complete UDB and datapath specifications.

5 Datapath-Based Components

A custom PSoC Creator Component is the best way to use a datapath effectively. Cypress supplies a *Component Author Guide (CAG)* to describe the entire Component creation process. To open the guide within PSoC Creator, select **Help > Documentation > Component Author Guide**.

There are two methods to implement datapath-based Components using PSoC Creator. You can write a Verilog file and use the Datapath Configuration Tool to configure the datapath. Or if you are not comfortable writing Verilog or using the Datapath Configuration Tool, you can use the UDB Editor. Each method has advantages and disadvantages; in general the Verilog method is more advanced than the UDB Editor, and the UDB Editor is easier to use than Verilog.

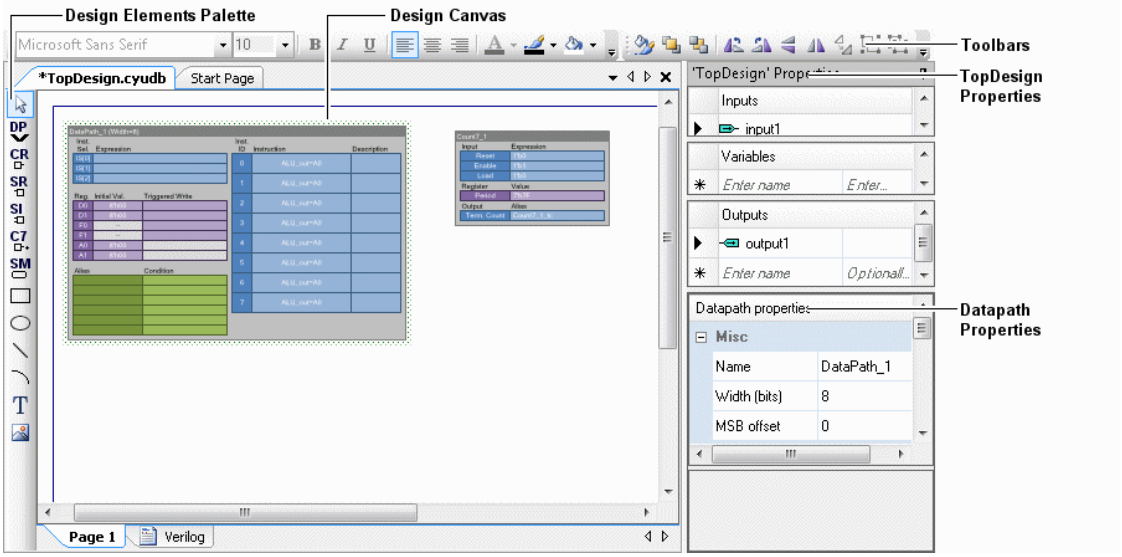
5.1 UDB Editor

The UDB Editor is a graphical tool used to construct UDB-based designs, as Figure 4 shows. It can be used to design Components without writing Verilog or using the more advanced Datapath Configuration Tool. The UDB Editor allows access to various elements in the UDB including the datapath, control register, status register, status interrupt register, count7 counter, and PLDs; all in a graphical form.

The UDB Editor allows you to design UDB-based hardware with very little knowledge of digital logic or Verilog. It is designed such that you can drag and drop and configure your hardware without writing verilog code. The tool then translates your design to Verilog in real time – giving you an opportunity to see how the UDB blocks translate to Verilog.

Since the UDB Editor is a graphical tool, it requires less knowledge of Verilog and the intricate details of the UDB. However, it sacrifices some flexibility and fine-grained control over the hardware as a result of simplifying abstractions. It also does not incorporate some of the more advanced UDB functionality, which may be limiting for complex designs.

Figure 4. UDB Editor



The structure of the UDB Editor is as follows.

Pages – When you open a UDB Editor document, you see an editable page similar to a schematic page. This is your design canvas, used to place and configure your UDB elements. You can have any number of UDB Editor pages by adding more pages from the **Page 1** tab. These pages then are translated together as one design.

Verilog – Next to the **Page** tab, you can find the **Verilog** tab. This is a read-only view of the translated hardware in your design. It is dynamically updated; whenever a change is made in your design, it also updates the Verilog code. This code is not editable or deletable, so you must make the appropriate change in the design canvas to see the results. You may also copy and paste this code to a Verilog file if you wish to edit your design using Verilog.

Design Properties – Located to the right of the design canvas, you can find the design properties, which allow you to configure the inputs, outputs, and variables used in your design. The inputs and outputs form your Component terminals when your design is complete. The design properties window is also used to set the global datapath configuration if your design contains a datapath.

Design Elements Palette – The design elements palette is located to the left of the design canvas. This is a menu used to choose the UDB elements to include in your design. These are:

- datapath (DP)
- control register (CR)
- status register (SR)
- status interrupt register (SI)
- count7 counter (C7)
- state machine state (SM)

For more detailed information on the UDB Editor consult the Universial Digital Block (UDB) Editor Guide

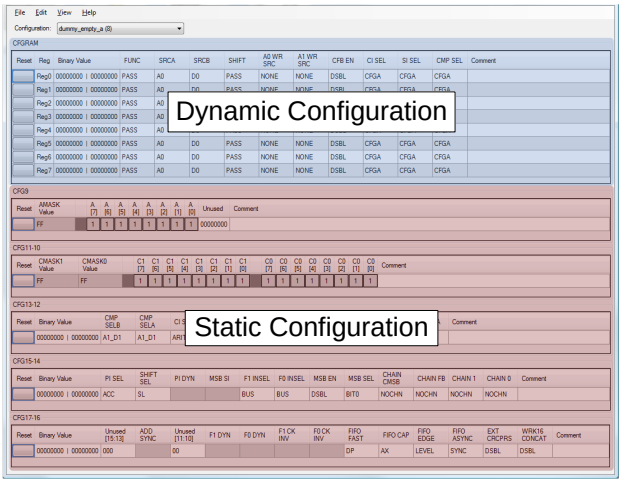
5.2 The Datapath Configuration Tool

The Datapath Configuration Tool is an application that allows you to create, view, modify, and delete datapath instances within Verilog files. The tool parses a Verilog file and displays each datapath as an entity, called a "configuration" in the tool. One configuration represents a single physical datapath.

If no datapath is present in the Verilog file you can create a new one and add it. You can add multiple datapaths to a single file, and you can link them (chain) to create multi-byte functions.

Datapaths are presented through a graphical user interface (GUI), as Figure 5 shows. The GUI display is just a representation of the datapath's CFGRAM and static configuration registers.

Figure 5. Datapath Configuration Tool Interface



Appendix B – Datapath Configuration Tool “Cheat Sheet” contains a description of the GUI fields. The GUI fields correspond to registers in the PSoC device. These registers are described in detail in the PSoC 3 Architecture TRM, PSoC 5LP Architecture TRM, and PSoC 4 Architecture TRM.

You can launch Datapath Configuration Tool from within PSoC Creator (**Tools** > **Datapath Config Tool…**).

The main body of this application note does not focus on the Datapath Configuration Tool, but instead focuses on the UDB Editor. Appendix A shows how to do all of the examples shown in this application note using the Datapath Configuration Tool. Also included in the Appendix is an example of using the parallel input and parallel output.

5.3 Choosing the Correct Tool

Most designs can be done using the UDB Editor. The UDB Editor implements the most common features of the datapath. However, there are some features that it does not implement. If you need to use those features then you must use the Datapath Configuration Tool. The features the UDB Editor doesn't support are the following:

- Dynamic FIFO Control
- FIFO Clock Inversion
- Parallel in and parallel out. There is an example on how to do this with the Datapath Configuration Tool in Appendix A.
- Cyclic Redundancy Check (CRC)
- Pseudo Random Sequence (PRS)
- Selectable Carry In
- Dynamic Carry In

These features are not needed in most designs. Thus it is best to start with the UDB Editor, as it is much simpler to use. If you find out during development that the UDB Editor doesn't have the features you need you can always copy and paste the Verilog code generated by the UDB Editor and modify the Verilog file using the Datapath Configuration Tool.

Note: The UDB Editor cannot read a Verilog file created by the Datapath Configuration Tool.

If you know you need one of these advanced features, or would rather write your own Verilog and use the Datapath Configuration Tool that is still an option. Appendix A shows all the examples using the Datapath Configuration Tool instead of the UDB Editor.

If the UDB Editor is the correct tool for you the remainder of this application note provides step by step instructions on how to create simple datapath-based Components using the UDB Editor.

Note: Attached to this application note are several completed examples. Completed examples do not exsist for all devices; instrucitons below are sufficient to create a Component on any PSoC device.

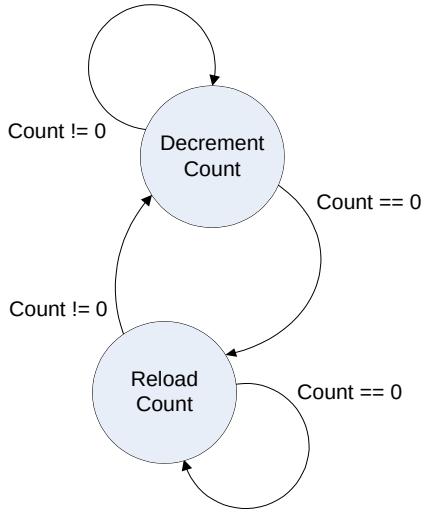
6 Project #1 – 8-Bit Down Counter

The purpose of this project is to introduce you to the steps you must take to create a simple datapath-based Component. This will be demonstrated by creating a simple 8-bit down counter.

6.1 8-Bit Counter Component Details

A simple down-counter can be represented by a state machine with two states, as Figure 6 shows.

Figure 6. 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 datapath instructions to implement this counter. The first instruction decrements the count value. The second reloads the count with the initial value.

The two datapath registers used in this example are:

- A0 – holds the count value and is decremented by the ALU.
- D0 – holds the value that is reloaded into A0 when the count reaches zero.

For this counter to work, you need a method to decide which instruction the datapath is executing: loading or decrementing. Figure 6 shows that transitions are controlled by the value of the count, that is, whether the count is zero or not.

The datapath has a zero detector block (ZDET) that monitors the value of the data in A0 and A1. The block has two outputs, z0 (A0 == 0) and z1 (A1 == 0), which indicate the condition of A0 and A1, respectively. Each output is HIGH when the value is zero and LOW when the value is non-zero.

In this example, the z0 output is used to control which instruction is executed.

Remember the datapath can have 8 uinque instructions. The instruction used by the datapath is choosen by a 3 bit address line. As stated before, this counter needs only two instructions, so you can tie bit 0 to z0 and hold the other bits LOW. Table 2 is the transition table.

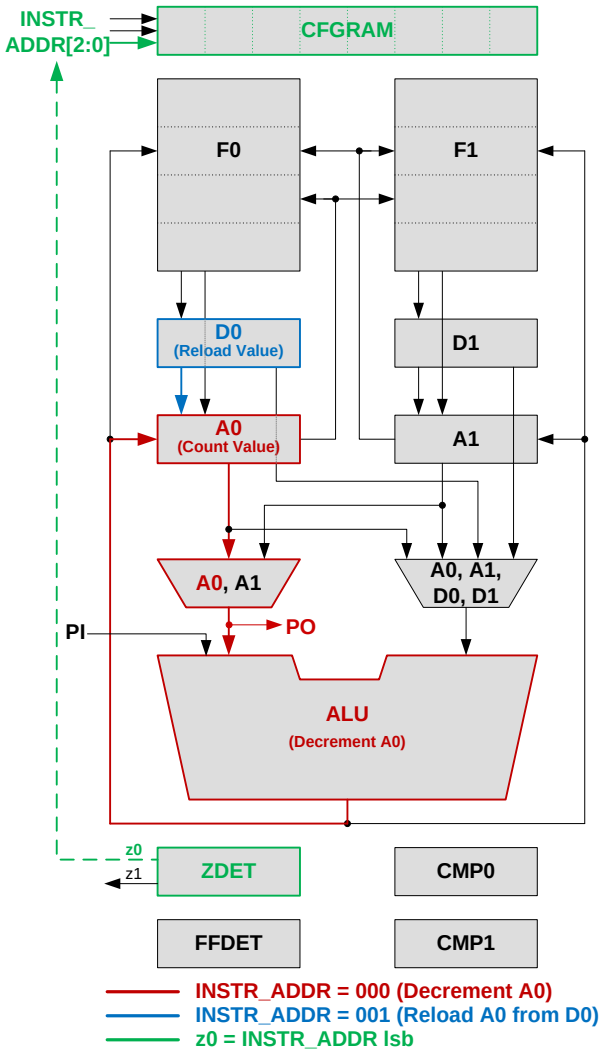
Table 2. Datapath Instructions

CFGRAM Instruction	Instruction Address Bits (INSTR_ADDR)			Operation
	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 in A0 reaches zero, z0 becomes '1'. This causes the datapath instruction to be "Reload Count". When the count is reloaded from D0 into A0, z0 becomes '0' and the instruction becomes "Decrement Count".

To visualize how this works, look at a highlighted datapath block diagram shown in Figure 7.

Figure 7. Simple Counter Block Diagram With Highlights

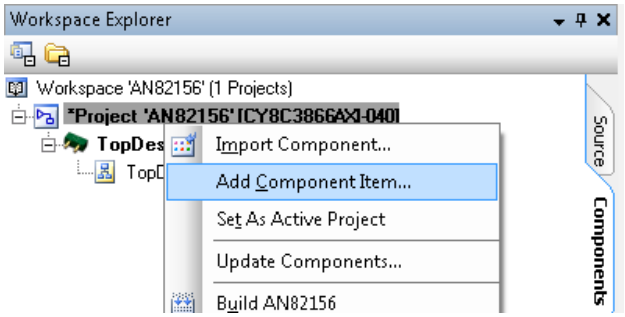


6.2 8-Bit Counter Component Creation Steps

For this example, use an empty project as a starting point.

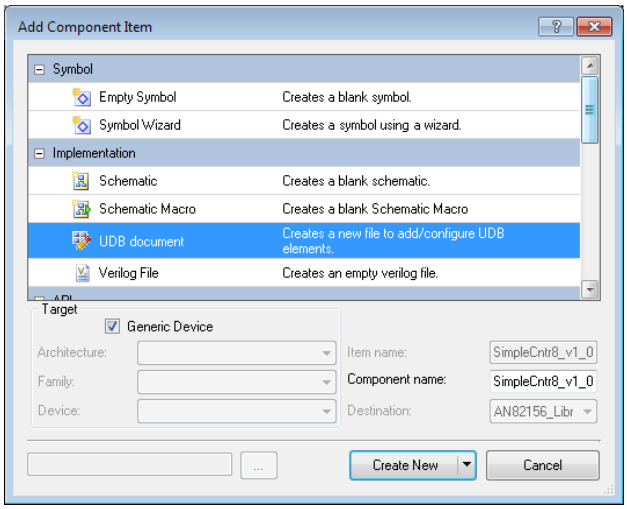
1. Launch PSoC Creator and create a project named "*AN82156*". An "AN82156" workspace is also created by default.
2. Switch to the **Components** tab of the **Workspace Explorer** and right-click **Project 'AN82156'**. Select **Add Component Item…** from the drop-down menu, as Figure 8 shows.

Figure 8. Add Component Item



3. Select **UDB Document** and name the Component "*SimpleCntr8_v1_0*", as Figure 9 shows.

Figure 9. Add UDB Document

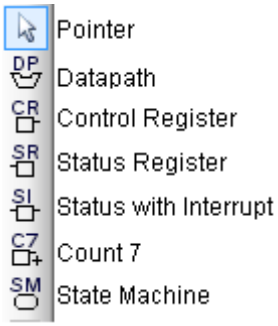


It is a good practice to include a version number in the Component name. Append to the Component name the tag "_vX_Y", where 'X' is the major version and 'Y' is the minor version. PSoC Creator has versioning capabilities that can help you track and use multiple versions of your Components.

4. Click **Create New** to create a UDB Document Schematic.

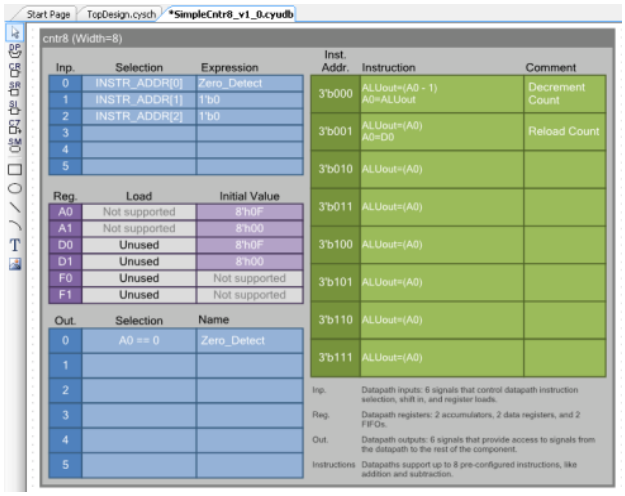
   The UDB Document Schematic is a canvas upon which you create your UDB-based Components. You can drag and drop UDB elements onto the schematic much like you would do with a regular PSoC Creator schematic. The Components that you can drag onto the schematic are shown on the side bar left of the schematic; see Figure 10.

Figure 10. UDB Elements



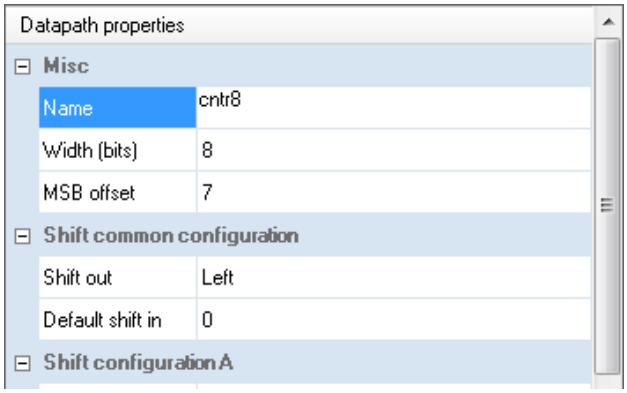
5. Drag a Datapath Element onto the UDB Document schematic; see Figure 11.

Figure 11. Datapath Element



6. Click the datapath instance. On the right-hand side of the UDB Editor is the properties window; at the bottom find the **Datapath properties**. Change the Datapath name from '*Datapath_1'* to '*cntr8'* as Figure 12 shows.

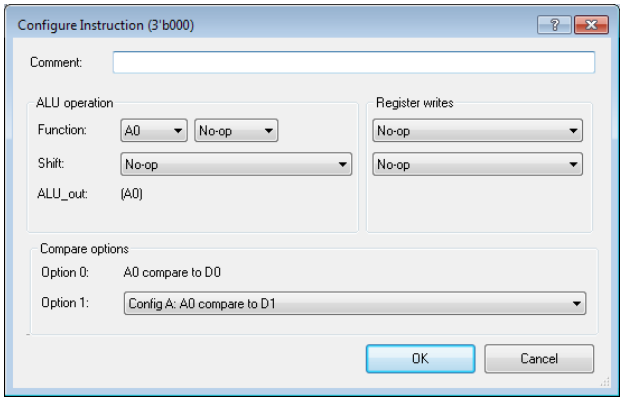
Figure 12. Datapath Properties



7. On Datapath *cntr8,* double-click the green box with the Inst. Addr of 3'b000 (see Figure 13) to open the configuration dialog shown in Figure 14.

Figure 13. Instruction Zero

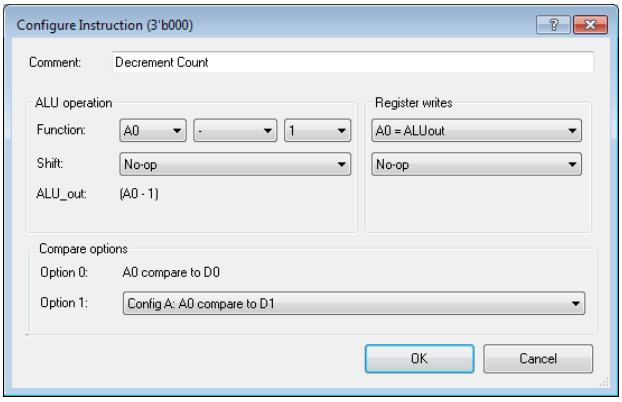
Figure 14. Blank Instruction Configuration Dialog



In this dialog box, configure the first datapath instruction. Look back at Table 2 and see that for Instruction zero the datapath decrements the count (which is stored in A0).

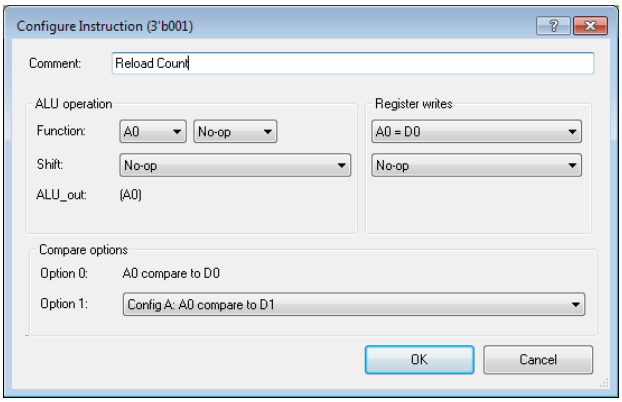
8. Set the **ALU operation Function** to A0 – 1 (Decrement). In the **Register Write** section set *A0 = ALUout*. This configures the first instruction to decrement A0 and write it back into A0. You can also describe this instruction with a comment: in this example, enter "Decrement Count." See Figure 15.

Figure 15. Instruction Zero Configuration



9. Next, configure Instruction Address 3'b001. Double-click that box. As listed in Table 2, instruction 1 reloads A0 with the value stored in D0. In the **Register writes** section set *A0 = D0*. See Figure 16.

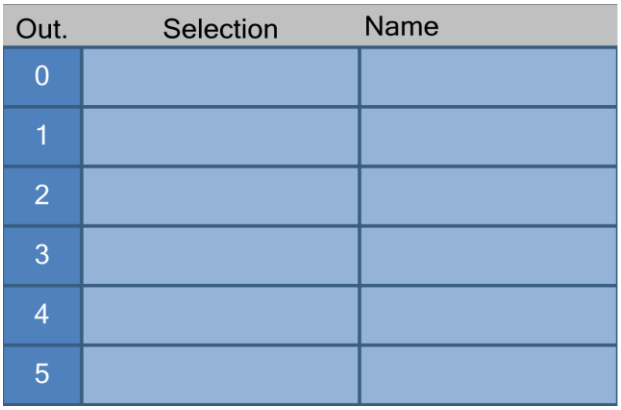
Figure 16. Instruction One Configuration



Next, we need to select the instruction the datapath executes during each clock cycle. For this example, we use the zero detector (ZDET) built into the datapath. First, we bring the zero detector signal out to a named label.

10. Double-click the blue output box shown in Figure 17.

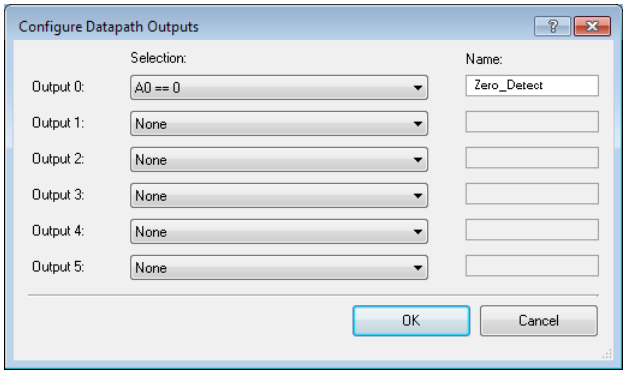
Figure 17. Datapath Outputs



The Configure Datapath Outputs dialog appears, as Figure 18 shows.

11. Configure Output 0 to A0 == 0 and set the Name to 'Zero_Detect'.

Figure 18. Datapath Output Configuration



The signal Zero_Detect represents the ZDET output of the datapath. Zero_Detect is HIGH when A0 is equal to zero and LOW when A0 is not equal to zero.

Route this signal to the instruction address of the datapath. This is configured in the Input section of the datapath.

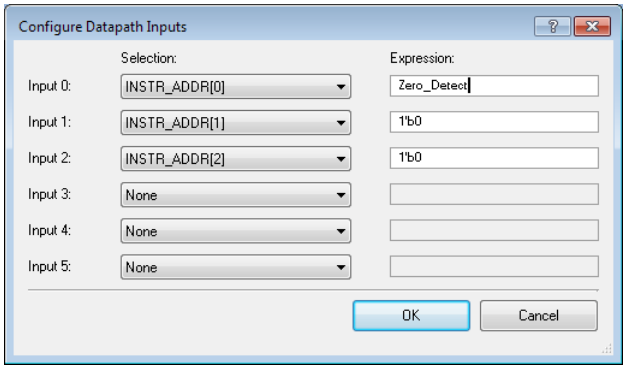
12. Double-click the blue input box shown in Figure 19.

Figure 19. Datapath Inputs

Inp.	Selection	Expression
0	INSTR_ADDR[0]	1'b0
1	INSTR_ADDR[1]	1'b0
2	INSTR_ADDR[2]	1'b0
3		
4		
5		

13. For **Input 0** ensure that **Selection** is set to *INST_ADDR[0],* and set the **Expression** to *'Zero_Detect'* as Figure 20 shows.

Figure 20. Datapath Input Configuration



The Zero_Detect signal selects the instruction the datapath executes. When Zero_Detect is HIGH, the datapath executes instruction 1 (Reload A0 with D0); when it is LOW, the datapath executes instruction 0 (Decrement A0).

Next, define some initial values for the counter and period registers.

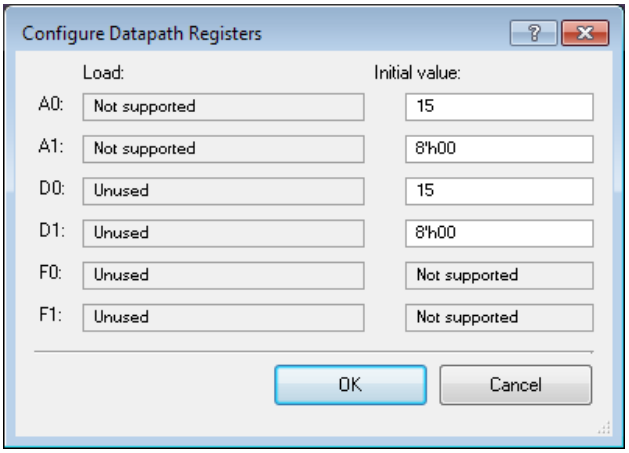
14. Double-click the gray and purple register configuration box shown in Figure 21.

Figure 21.Datapath Registers

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

15. Set the initial value for A0 and D0 to 15 see Figure 22.

Figure 22. Datapath Register Configuration

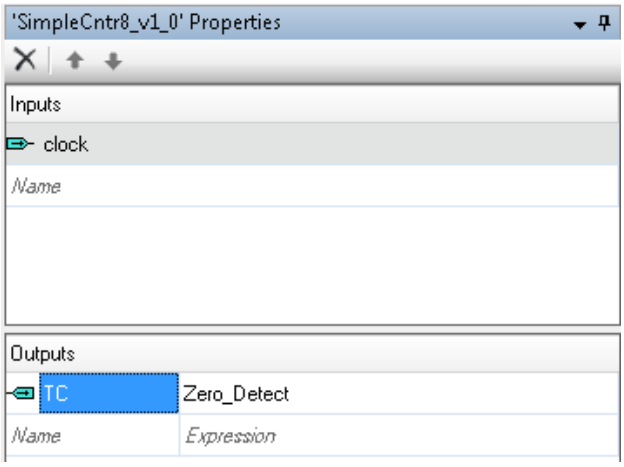


This sets the values at which A0 and D0 start. This means when the component starts A0 will have a value of 15, and thus will count down to 0, when it reaches zero it will be loaded with the value in D0, which we set to 15.

We have configured the datapath. Next, we define a Component output. This output is the terminal count (TC). It goes HIGH when the counter reaches zero, and is LOW at all other times. We use this output to test the functionality of the component.

16. In the 'SimpleCntr8_v1_0' Properties window under Outputs define a new output named 'TC' and set the Expression to 'Zero_Detect' as Figure 23 shows.

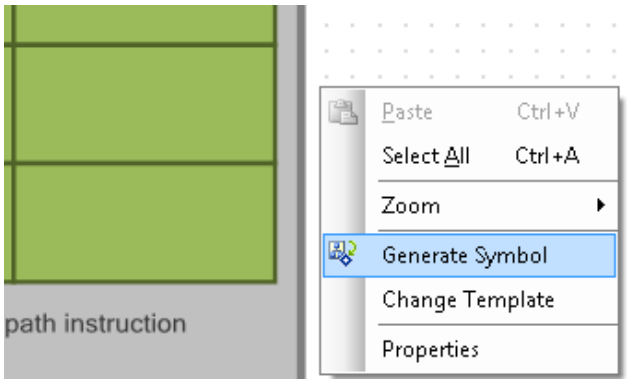
Figure 23. Component Outputs



Generate a symbol for this Component. This is the symbol that appears on your top design schematic.

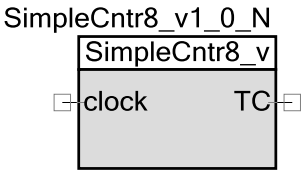
17. Right-click inside the blank area of the UDB Document (*.cyudb*) and select **Generate Symbol** see Figure 24.

Figure 24. Generate Symbol



A schematic symbol appears as Figure 25 shows.

Figure 25: Schematic Symbol

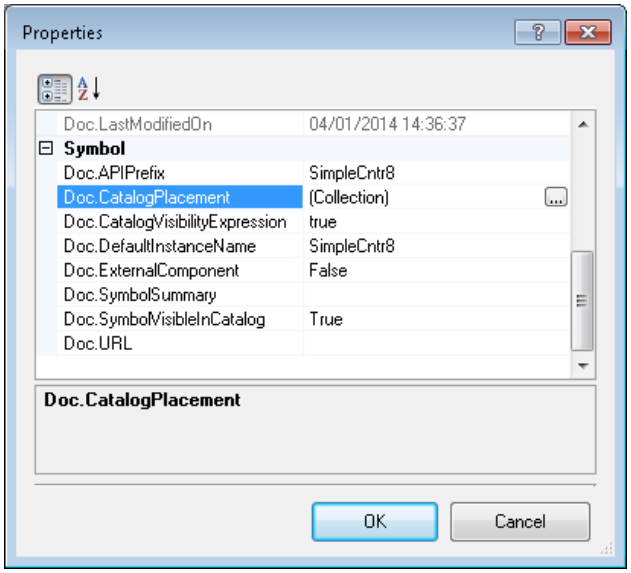


18. Right-click an empty space in the symbol Editor – not the symbol itself – and select **Properties** from the drop-down menu.

19. Enter values in the **Symbol** section of the property fields as Figure 26 shows:

- *Doc.APIPrefix* = SimpleCntr8.
  This value is prefixed to any API file names generated for the Component. You do not generate an API in this example, but enter a value here whenever you create a Component.
- *Doc.CatalogPlacement = AN82156/Digital/Cntr8.*
  Click on the '**…**' button to open the Catalog Placement dialog to enter this value. PSoC Creator uses this value to define the hierarchy of the Component catalog. The first term is the tab under which the Component appears in the catalog. Each subsequent '/' represents a sublevel. The hierarchy must contain at least one sublevel. The value written above indicates that the Component is visible as 'Cntr8' in the 'Digital' sublevel of the AN82156 tab.
- *Doc.DefaultInstanceName = SimpleCntr8.*
  This is the default name that appears when the Component is placed in a schematic.

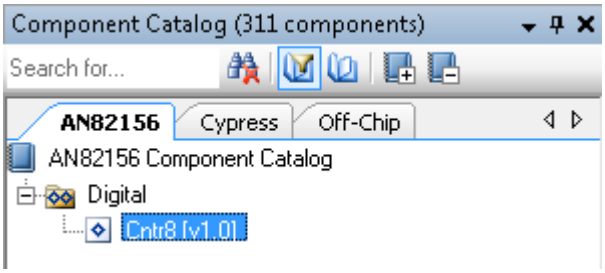
Figure 26. Add Symbol Properties



20. Select **File** > **Save All** to ensure that all changes have been applied to the project.

The Component is ready for use. The new Component is visible in the Component Catalog under the *AN82156* tab, as Figure 27 shows.

Figure 27. New Component in Component 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 must add a clock source and a way to view the counter's 'TC' output.

21. Place the Cntr8 Component in the project schematic.
22. Connect a Clock Component to the 'clock' terminal. Set the clock to 10 kHz. Any value is OK, but 10 kHz makes it easy to view on an oscilloscope.
23. Connect a Digital Output Pin Component to the 'clock' terminal so that you can observe it on a scope. Name it P0_0_clk and leave all other settings at their default values.

    Note: For PSoC 4, you cannot directly route a clock out to a pin. To route the clock to the pin, follow these instructions:

    a. Place a Digital Output Pin Component on your schematic.
    b. Select the **Clocking** tab in the pins customizer.
    c. Set the **Out Clock:** to *External.*
    d. Go back to the **Pins** tab, and go to the **Output** subtab.
    e. Under **Output Mode:** select *Clock.*
    f. Click **OK**.
    g. Connect the clock signal to the out_clk terminal on the Pins Component.

Note: For PSoC 6 MCU, you cannot directly route a clock out to a pin. To route the clock to the pin, follow these instructions:

a. Place a Digital Output Pin Component on your schematic.

b. Place a TFF Component on your schematic.

c. Connect the output of the clock Component to the clk terminal of the TFF Component.

d. Connect a Logic High ‘1” Component to the t input of the TFF Component.

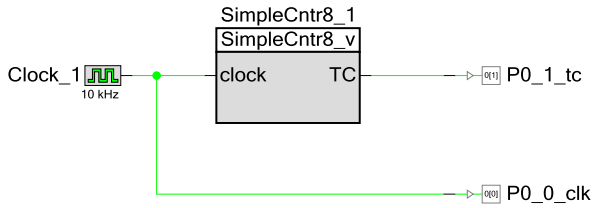
e. Connect the q output of the TFF Component to the Digital Output Pin placed in Step a.

Note: The clock frequency seen on the outside of the device will be half the actual value used by the UDB Component.

24. Connect a Digital Output Pin Component to the 'TC' terminal. Name it P0_1_tc and leave all other settings at their default values. This pin is HIGH when the count is zero.

    Figure 28 shows the completed project schematic.

Figure 28. Simple Counter Project Schematic



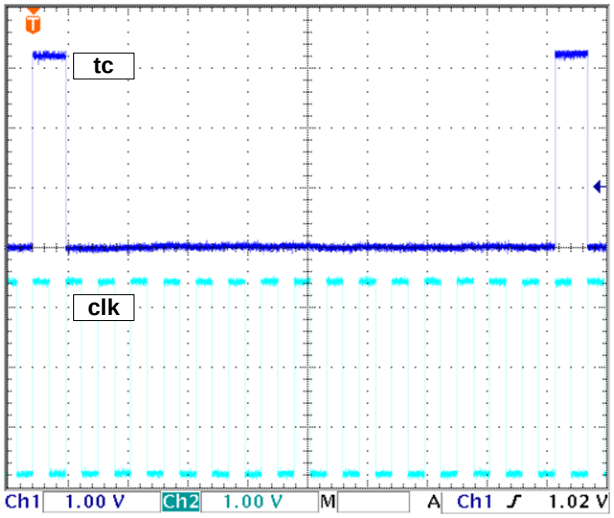
25. Assign the pins to P0[0] and P0[1], according to their names, in the *Pins* tab of the cydwr.

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

26. Save the project, build it, and program the PSoC.

If you connect a scope to the output pins, you can observe the 'clock' and 'tc' outputs, as Figure 29 shows.

Figure 29. Simple Counter Outputs



You loaded A0 with a starting value of 15, so you can see that the period is 16 (counts from 15 to 0) 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 configuration transitions back to decrementing A0 again.

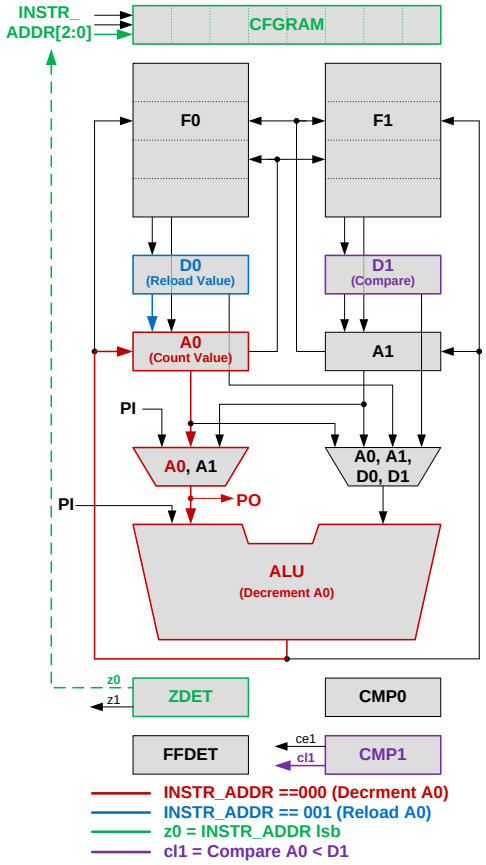
You have just designed your first datapath-based Component, and you didn't write any firmware to do it!

6.3 Modify the Counter to be a PWM

A PWM is just a counter with a compare. To create a 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 30 shows:

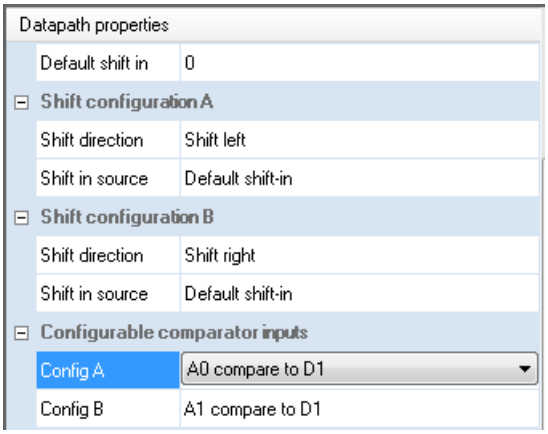
Figure 30. Simple PWM Block Diagram With Highlights



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.

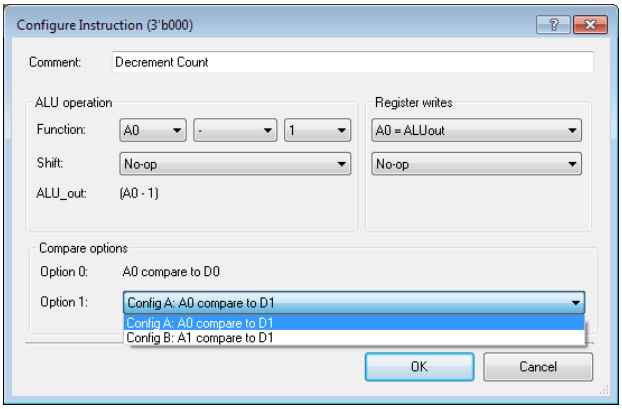
1. Go back to the UDB Document (.cyudb) and click on the datapath. In the Datapath properties window, ensure that Config A of the Configurable comparator inputs is set to A0 compare to D1; see Figure 31.

Figure 31: Datapath Comparison Configuration



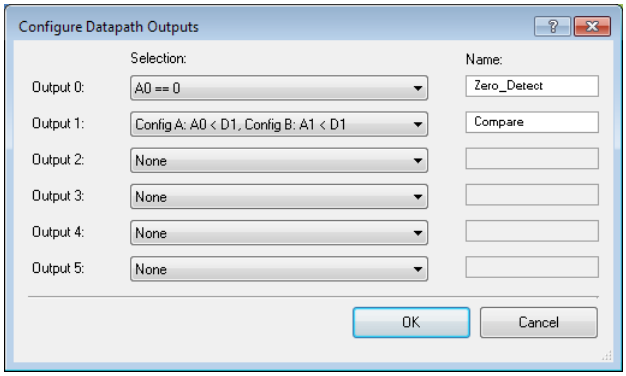
2. Ensure that for both instruction 0 and instruction 1, **Compare Option 1** is set to *ConfigA: A0 compare to D1,* as Figure 32 shows. To open this dialog, double-click the appropriate green instruction box.

Figure 32. Configure Compare Options



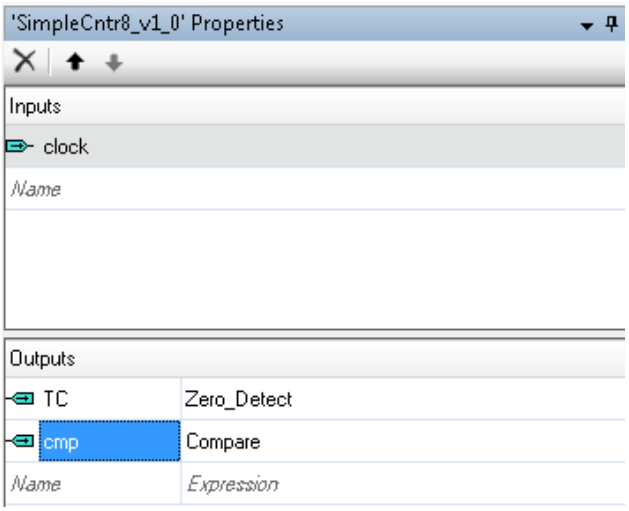
3. Double-click the Outputs box and configure **Output 1:** for *Config A: A0 <D1…* and set the **Name** to '*Compare' as* Figure 33 shows.

Figure 33. Configure Compare Output



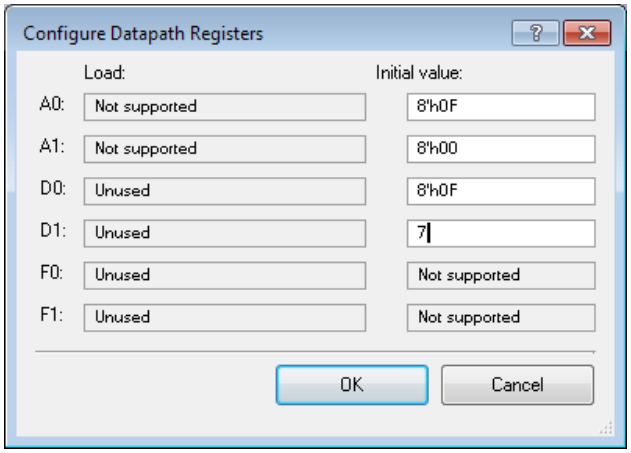
4. Define a new Component output named *'cmp'* and set the *Expression* to Compare; see Figure 34.

Figure 34. Compare Output



5. Finally, configure an initial value for the compare register (D1).
6. Open the **Configure Datapath Registers** dialog. Set the initial value of D1 to 7 as Figure 35 shows.

Figure 35. D1 Initial Value



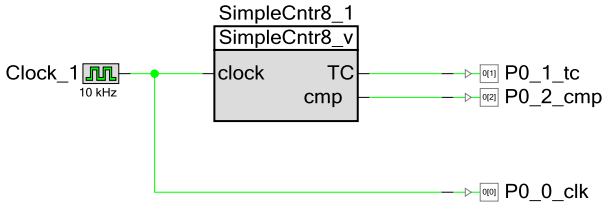
7. Right-click any white space on the schematic canvas and select Generate Symbol, this will create a new symbol with the new output cmp.
8. Select **File** > **Save All**.

With this configuration, the compare block outputs a HIGH whenever A0 is less than D1; LOW when A0 is greater than D1.

The PWM Component and symbol are still visible in the Component Catalog in the AN82156 tab. The Component is automatically updated in the project schematic.

9. 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 36 shows.

Figure 36. Simple PWM Project Schematic

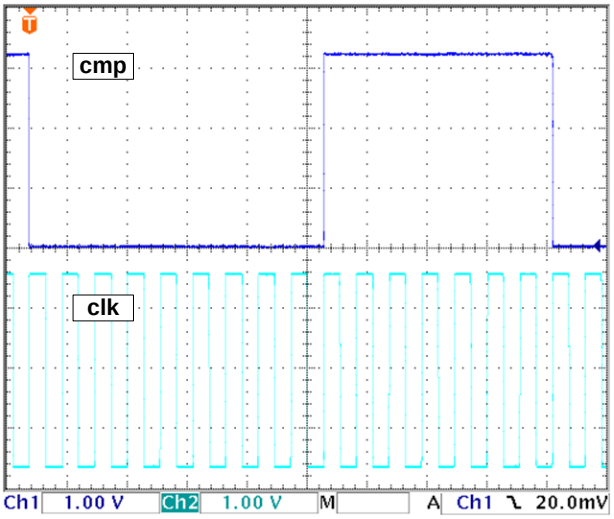


Now, you are ready to build the project and program the PSoC. The clock and the terminal count can still be observed on pins P0[0] and P0[1]. The PWM output can be observed on P0[2].

10. Save the project, build it, and program the PSoC.

If you connect a scope to the output pins, you can observe the 'clock', 'tc', and 'cmp' outputs. Figure 37 shows the 'clk' and 'cmp' signals.

Figure 37. Simple PWM Outputs



You loaded A0 and D0 with a starting value of 15, 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.

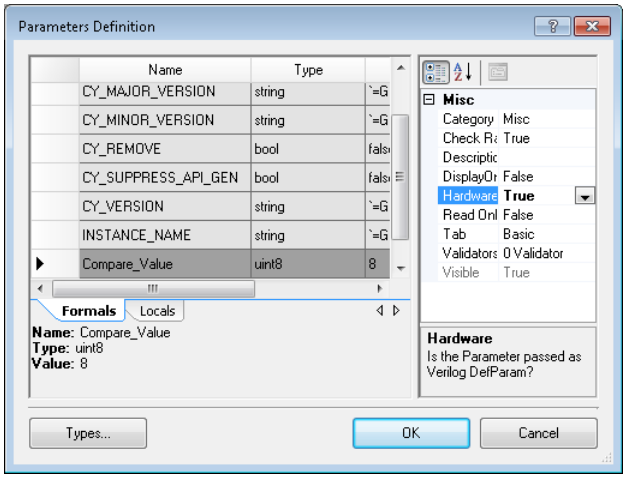
6.4 Adding Parameters

It is inconvenient to modify Verilog code whenever you need to make a change to one of the Component's parameters. Furthermore, what if you needed two PWMs with different periods and compare values? You can add user-configurable parameters to your Component, similar to the way most Cypress Components work.

1. Open the Component's Symbol Editor page (.cysym) 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 at the right-hand side of the window, as Figure 38 shows.

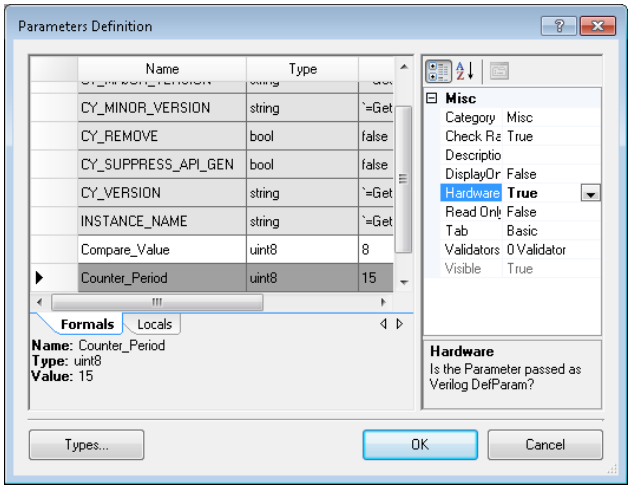
   This exposes the parameter to the Verilog code so that the UDB hardware can use it.

Figure 38. Adding a 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 39 shows.

Figure 39. Adding Another Component Parameter

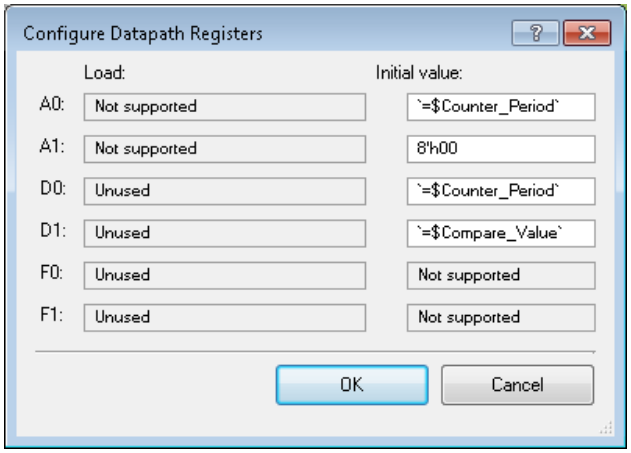


7. Click **OK** and select **File** > **Save All** to apply the changes to the Component.

The next steps show you how to link the parameters to the UDB Editor.

8. Go to the UDB Editor (.cyudb) and open the **Configure Datapath Register** dialog.
9. For **A0** and **D0** set the **Initial value:** to `=$Counter_Period`. Notice that the accent ` key is used, not the single quote ' key. For **D1**, set the **Initial value:** to `=$Compare_Value`; see Figure 40.

Figure 40. Initial Values With Parameters



This code links the initial register values for A0, D0, and D1 to the Component's parameters.

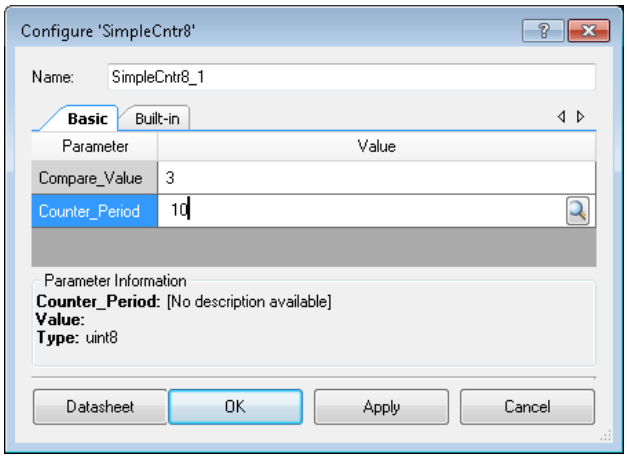
Now, you can set the period and compare values at build time without modifying these values in the UDB Editor.

10. Select **File** > **Save All**.

Go back to the project schematic and double-click the SimpleCntr8_1 Component to open the properties dialog.

11. Change the compare value to 3 and the counter period to 10, as Figure 41 shows.

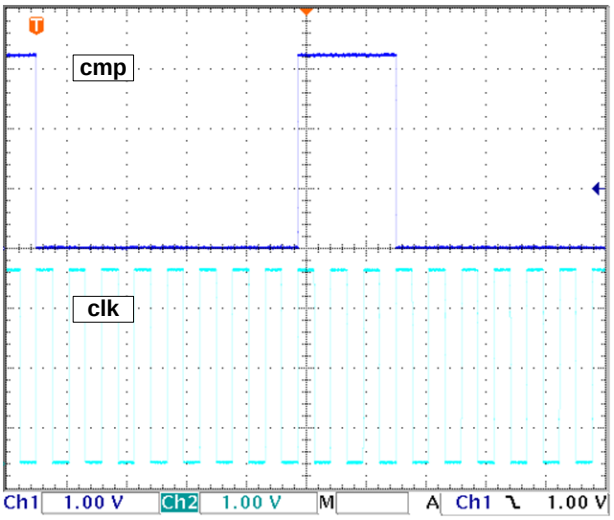
Figure 41. Setting Component Parameters



12. Click **OK** to apply the changes.
13. Select **File** > **Save All**, build the project, and program the PSoC.

As you can see in Figure 42, the period and compare output have changed.

Figure 42. PWM Output With New Parameters



You set the period to 11 cycles (the period is 10+1 because the counter goes from 10 to 0 before it reloads), and the compare to 3. The result is 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 a uint8. You can even place multiple instances of the Component in your project and set them to different values.

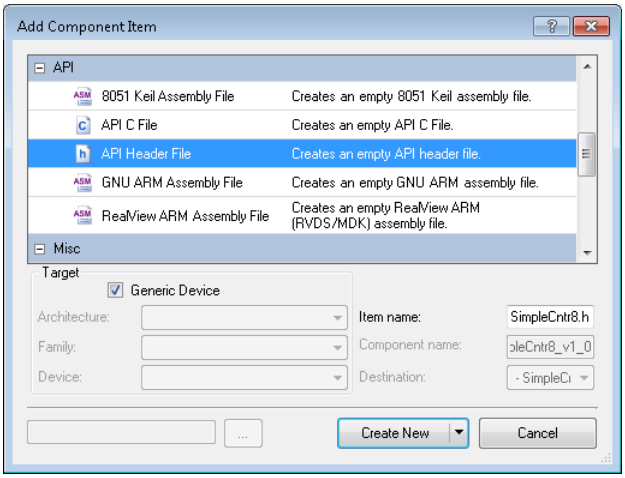
For more information on adding Component parameters, including how to set limits on what value users can enter, see the *Component Author Guide*.

6.5 Adding Header Files

In addition to being able to change the behavior of the PWM at design time, you can change the PWM during run time by modifying the PWM registers via C code. For example, you used register D0 to hold the period value and register D1 to hold the compare value. To make these registers easy to access, create a header file that defines the registers used, so they can be modified in c code.

1. In the **Components** tab, right-click **SimpleCntr8_v1_0** and select **Add Component Item**.
2. In the Add Component Item window, navigate down to the **API** section and click **API Header File**.
3. Change the **Item name** to *SimpleCntr8.h,* as Figure 43 shows.

Figure 43. Add Header File



4. Click **Create New**. This adds a header file to your Component.

Add definitions for the compare value (D1) and the period value (D0).

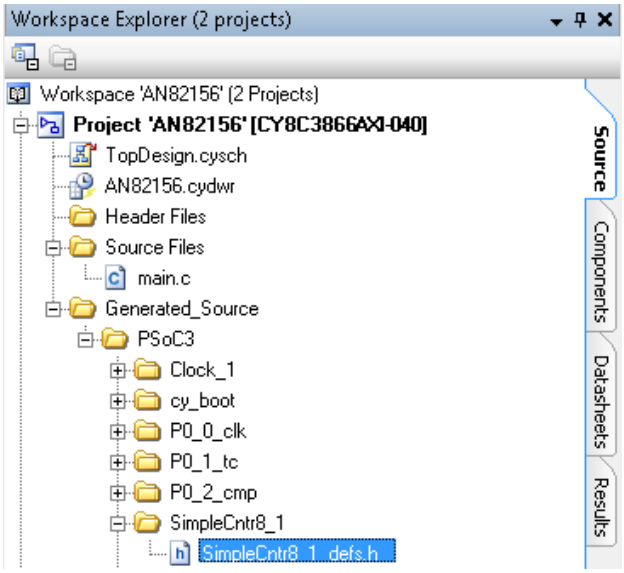
5. Add the following definitions above `//[]END OF FILE` in the header file:

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

These two definitions allow you to directly write to the D0 and D1 registers in firmware. The *INSTANCE_NAME`_defs.h* file contains a set of definitions for the registers in the datapath.

INSTANCE_NAME`_defs.h can be found in the generated source folder of the source tab as Figure 44 shows.

Figure 44. The *defs.h* File



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

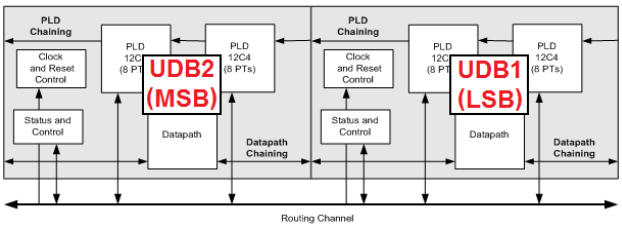
```
SimpleCntr8_1_Period_Reg = 0x08;

SimpleCntr8_1_Compare_Reg = 0x02;
```

This updates the period to 0x08 and the compare value to 0x02. For more information on how to use these defines, refer to *Component Author Guide*. Note that the method of directly writing to the register, as shown in the C code above, works only for 8-bit registers. For 16 bits or higher, you must to use a different method, which is discussed in the next project.

6.6 Expanding the PWM to 16Bits

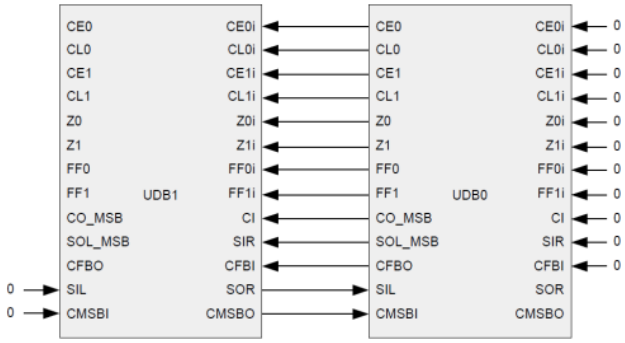
Let us look at the concept of datapath chaining. Datapaths have dedicated signals that are tied to neighboring datapaths. These signals allow you to create functions that are 1 to 32 bits wide. In this example we create a PWM that is similar to the first example project, but is 16 bits wide, as Figure 45 shows.

Figure 45. A 16-bit Function With Chained UDBs



The ALU in each datapath is designed to chain carries, shifted data, and conditional signals to its nearest neighbor, as Figure 46 shows. All conditional and capture signals chain in the direction of the least significant byte to the most significant byte. Shift-left also chains from the least to the most significant byte. Shift-right chains from the most significant to the least significant.

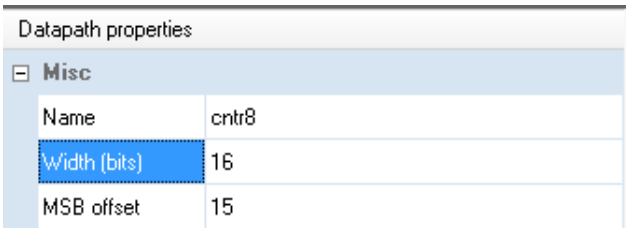
Figure 46. Datapath Chaining Flow



The UDB Editor makes it easy to chain datapaths together. Use the PWM that we just created and make it 16 bits wide.

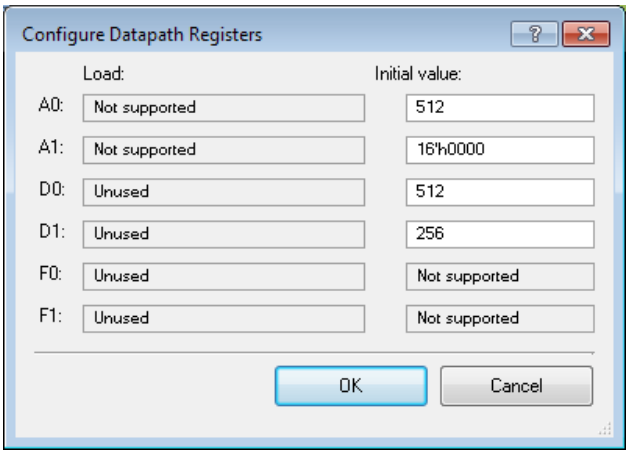
1. Go to the UDB Document (.cyudb).
2. In the **Datapath properties** panel, set **Width (bits)** to 16 as shown in Figure 47.

Figure 47. 16-Bit Configuration



3. Set the period (D0) and the initial period (A0) to 512, and the compare (D1) to 256, as Figure 48 shows.

Figure 48. 16-Bit Initial Values

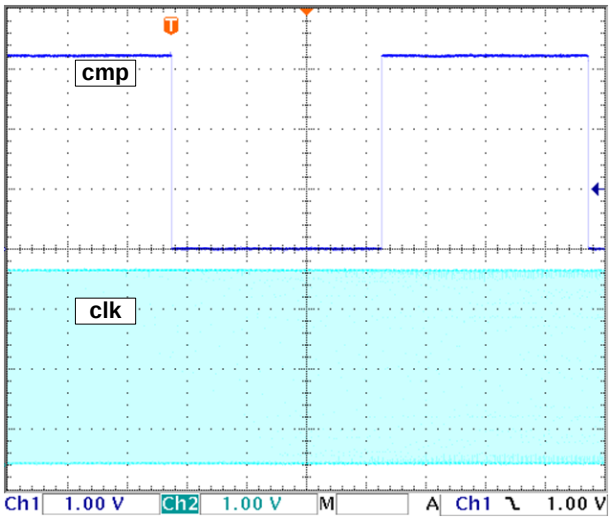


4. Select File > Save All in the project window.
5. Go back to the top design schematic (.cysch), and change the clock from 10 kHz to 1 MHz (this is easier to view on an oscilloscope).
6. Select File > Save All.

7. build the project, 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 49 shows. You can set them to any 16-bit value.

Figure 49. Simple 16-Bit PWM Output



You can use chaining to make functions up to 32 bits wide. Apply the principles described in this example to larger functions.

6.7 16-Bit Component Header Files in PSoC 3

As noted, writing to 16-bit registers is different from writing to 8-bit registers. You can write directly 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, endian differences is a concern.

The PSoC 3's 8051 endian-ness is different from the peripheral registers. To make writing to registers simple, Cypress provides these macros: CY_SET_REG16, CY_SET_REG24, and 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.

Therefore, you must know the address of the datapath registers. This is automatically done for you in the auto-generated header file (_defs.h) with the defines that end in _PTR.

To update a value, use CY_SET_REG16 and the pointer from the header file. Again, look at the attached example project.

6.7.1 16-Bit Parameters

In the previous example we set the symbol parameters to the type of *uint8* you can also add parameters for the 16bit PWM. You just change the type to *uint16.*

7 Project #2 – Up / Down Counter

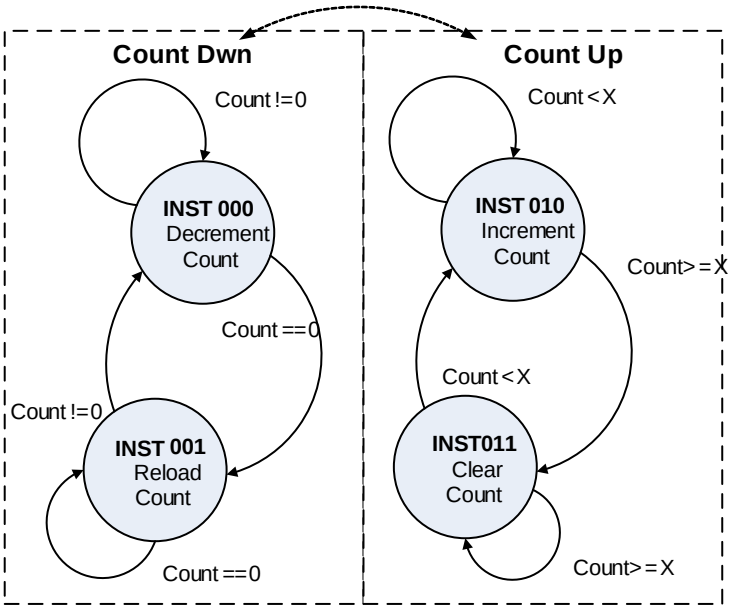
The purpose of this example is to introduce you to the steps to add advanced features to a Datapath Component. The same basic PWM concept is updated to add the ability to count up or down. The direction is based on a parameter that you can set during run time.

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

7.1 Additional Details

The simple down-counting PWM used two states. To implement an up/down counter, four states are needed, as Figure 50 shows.

Figure 50. Up/Down Counter State Diagram



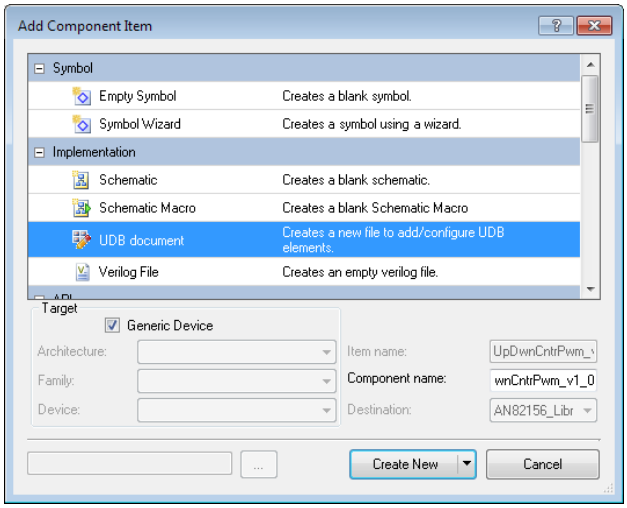
The datapath decrements or increments A0 depending on the parameter that you set. You can also set the period and compare.

7.2 Example Project Steps

To avoid confusion, create a new Component, instead of modifying the one from the previous example project. 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, called "UpDwnCntrPwm", to the workspace.
2. On the Components tab, right-click Project ‘UpDwnCntrPwm’ and select Add Component Item.
3. Select a UDB document and change the Component name: to UpDwnCntrPwm_v1_0. Click Create New.

Figure 51. Creating New Component



4. Drag a Datapath element onto the design canvas.
5. Go to the Datapath properties and change the Name to ‘UpDwn’.

   Configure the datapath to implement the functionality shown in Figure 50. The first two instructions are the same as in the simple 8-bit PWM, with the addition of two more instructions. The other two instructions implement the up-count feature, as Table 3 shows.

Table 3. Up/Down Counter Instruction Table

INSTR_ADDR	Function	Register Write	Comment
000	ALU = A0 - 1	A0 = ALUout	Decrement Count
001	No-op	A0 = D0	Reload Count
010	ALU = A0 + 1	A0 = ALUout	Increment Count
011	ALU = A0 ^ A0	A0 = ALUout	Clear Count

The XOR configuration is used to clear the count (A0) register. After the count (A0) register counts up to the period value, it XORs itself, an act that clears the count back to zero.

Each instruction must be configured as shown in Table 3. Figure 52 shows how each instruction looks in the Datapath element.

Figure 52. UpDwnCntr Instructions

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

6. In the Datapath properties panel, verify that Config A under the Configurable comparator inputs is set to A0 compare to D1.

7. Verify that in each instruction under Compare options is set to ConfigA: A0 compares to D1.
8. Configure a Zero_Detect and Compare output like you did in the first example, with the addition of a third output (Period). The third output goes HIGH when the counter reaches the value in the period register (D0). This indicates when the up counter has reached the period value and must be reset; see Figure 53.

Figure 53. UpDwnCntr Outputs

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

9. Drag a Control Register (CR) onto your schematic, as Figure 54 shows.

Figure 54. Control Register

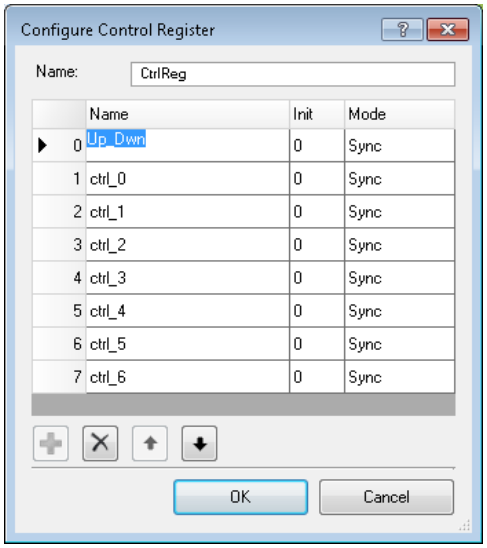
CtrlReg_1

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

10. Double-click the Control Register. Set the Name of the 0th bit to Up_Down. Rename the control register to CtrlReg; see Figure 55.

A control register is written by the CPU. Thus, the CPU controls the counting direction by writing to this control register bit. A value of 1 indicates up counting while a 0 indicates down counting.

Figure 55. Control Register Configuration



In the first example we had two datapath instructions; it was easy to select the instruction executed out of the CFGRAM by using the Zero_Detect signal. Now there are four instructions: two instructions for counting down, two for counting up.

We can still use Zero_Detect for the down count. For up count, use the Period signal. We also have a signal which determines if we are counting up or down (Up_Dwn). We have four instructions, this indicates that we need two address bits. However, we have three signals: Up_Dwn, Period, and Zero_Dectect, as Table 4 shows. Thus, logic must be used to reduce these three signals into two address lines.

Table 4. UpDwnPWM Instruction Decode

Up_Dwn	Period	Zero_Detect	INSTR_ADDR	Function
Down	N/A	0	000	Decrement
Down	N/A	1	001	Reload
Up	0	N/A	010	Increment
Up	1	N/A	011	Clear

Looking at Table 4, we see that one address bit is always connected to the Up_Dwn signal. The other address bit is multiplexed between Period and Zero_Detect. If the Up_Dwn signal is set to Down, then use the Zero_Detect value for the address bit, if Up_Dwn is set to Up, use the Period value for the address.

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

INSTR_ADDR[1] = Up_Dwn

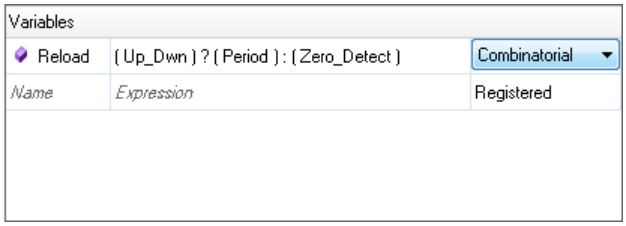
The UDB Editor allows us to input standard Verilog syntax into various fields in the Editor. It also allows us to create variables that can be controlled by logic. For this example, create a new variable that determines if we use Period or Zero_Detect for INSTR_ADDR[0].

11. In the **Properties** window under **Variables** add a new variable named *Reload* and set the expression to:

    (Up_Dwn) ? (Period) : (Zero_Detect)

12. Make sure that it is set to **Combinatorial** see Figure 56.

Figure 56. Variable Definition



The expression Reload = (Up_Dwn) ? (Period) : (Zero_Detect) is a ternary operator. It is a more compact way of writing a simple if-else statement.

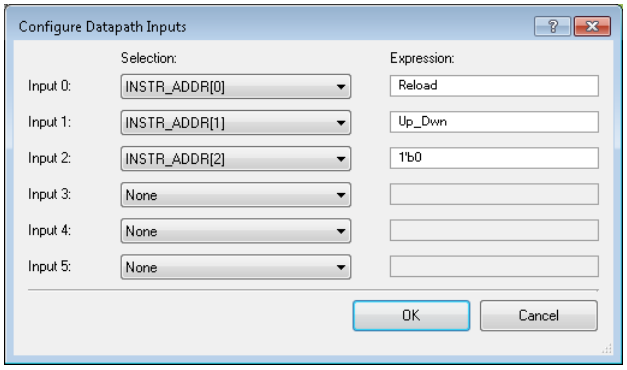
The form is as follows:

A = B ? C: D. If B is true (high) then A = C, if B is false (low) then A = D.

In this example if Up_Dwn is high Reload = Period. If Up_Dwn is low Reload = Zero_Detect.

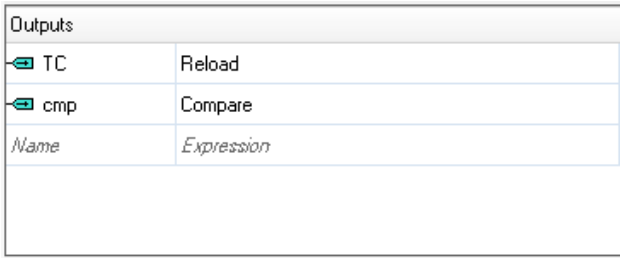
13. Go to the Datapath Inputs, and configure INSTR_ADDR[0] to Reload, and INSTR_ADDR[1] to Up_Dwn, as shown in Figure 57.

Figure 57. Instruction Addressing



14. Next, configure the PWM outputs. As before, we have a terminal count (TC) and a compare (cmp) output. The cmp output comes from the Compare signal. The TC signal comes from the new Reload variable.
15. Configure the outputs as Figure 58 shows.

Figure 58. Outputs

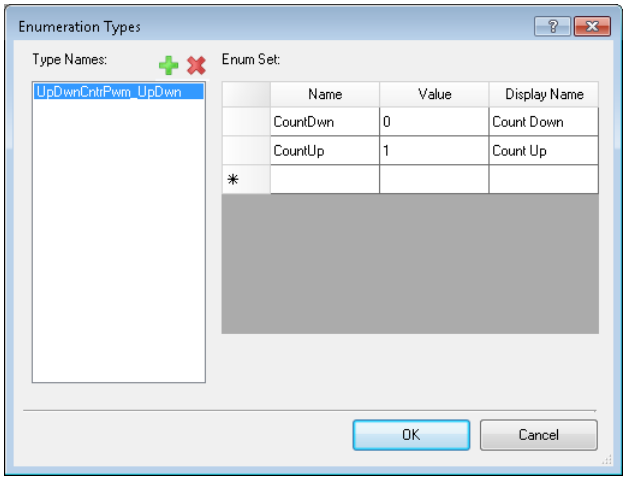


16. Right-click in an empty space in the .cyudb file and select Generate Symbol.
17. Right-click on the symbol schematic page (.cysym) and add symbol properties:
    - Doc.APIPrefix = *UpDwnCntrPwm*
    - Doc.CatalogPlacement = *AN82156/Digital/UpDwnCntrPwm*
    - Doc.DefaultInstanceName = *UpDwnCntrPwm*

Add some parameters that users can modify – counter period, compare value, and count mode (up or down). For the count mode setting, define a new type of parameter and set of values.

18. Right-click the Symbol Editor page and open the Symbol Parameters dialog.
19. Click Types to open the window to create the new parameter type.
20. Click the green '+' button to add a new type. Name it UpDwnCntrPwm_UpDwn.
21. Enter values into the Enum Set fields to define 'CountDwn' and 'CountUp' definitions; see Figure 59.

Figure 59. Creating New Component Parameter Types



22. Click **OK** to return to the Symbol Parameters dialog.

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

23. Enter three new parameters for the Component, just as you did in the previous examples; see Table 5.

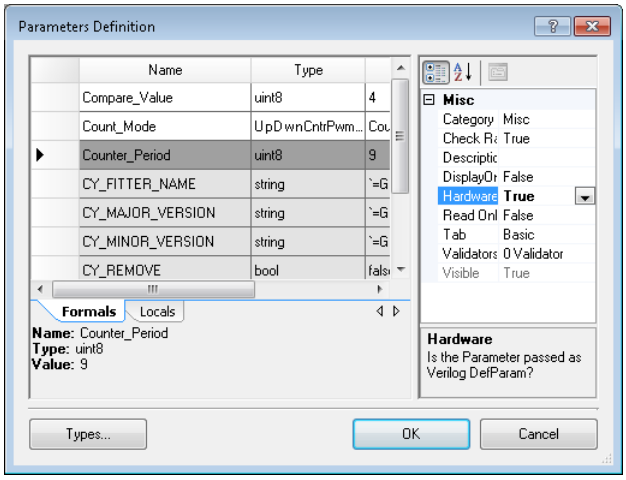
Table 5. Up/Down Counter Parameters

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.

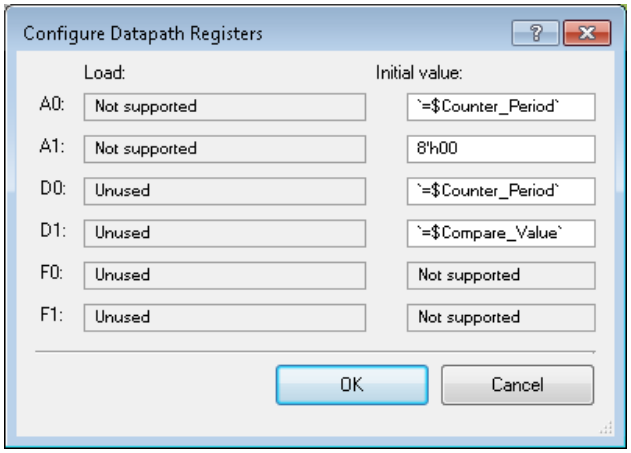
24. Set the **Hardware** flag to 'True' for all three new parameters, as Figure 60 shows.

Figure 60. Adding New Component Parameters



25. Click OK and save the changes to the Component.
26. Go back to the .cyudb file and modify the initial values of the register as you did in the first example; see Figure 61.

Figure 61. Initial Value With Parameters



At this time we cannot set the initial value of the control register in hardware. So, we set it in firmware.

27. Create a header file. On the Components tab, right-click UpDwnCntrPwm_v1_0 and select Add Component Item. Choose API Header File and name it UpDwnCntrPwm.h. Click Create New.
28. In the header file add the following line of code above /* [] END OF FILE */:

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

The define UP_DOWN is now linked to the parameter Count_Mode set in the Component customizer.

Set the direction of the counter in the main.c file using the included control register. Note that all standard control register API are available for use.

29. In the main.c file, add the following line of code:

```
UpDwnCntrPwm_1_CtrlReg_Write(UP_DOWN);
```

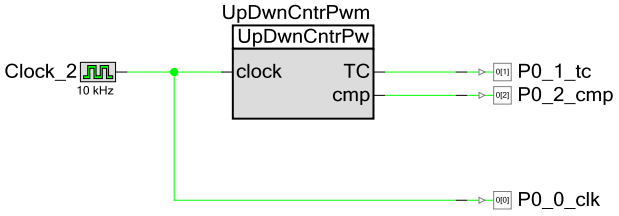
If you named your Component something other than UpDwnCntrPwm_1 then you must replace that with the name of your Component. If you named your control register anything other than CtrlReg then replace that name with the name of your control register. By placing a control register in your .cyudb file, you now have access to the standard control register API, as step 28 shows.

Your Component is ready to use. Add all the same Components as in the first example project.

30. Drag an UpDwnCntrPwm Component to the project schematic.
31. Connect a Clock Component to the 'clock' terminal of the Component and set it to 10 kHz.
32. Connect a Digital Output Pin Component to the Component's terminals – P0_0_clk, P0_1_tc, and P0_2_cmp – as Figure 62 shows.

    Note: For PSoC 6 MCU and PSoC 4, this process is different; see Step 23 in Section 6.2.

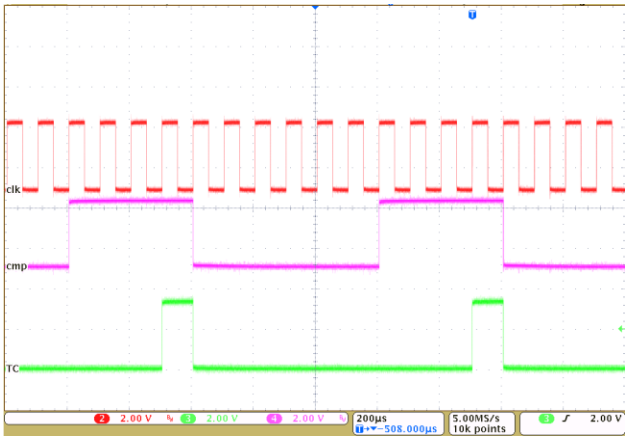
Figure 62. Up/Down Counter PWM Project Schematic



33. Select File > Save All.
34. Build the project, and program the PSoC.

Set the compare value parameter to 4, the period to 9 and the Count_Mode to Count_down. Observe the clk, cmp, and TC waveforms on a scope. You can see that it is counting down because after the TC goes HIGH, the cmp line goes LOW as the count is reloaded as Figure 63 shows.

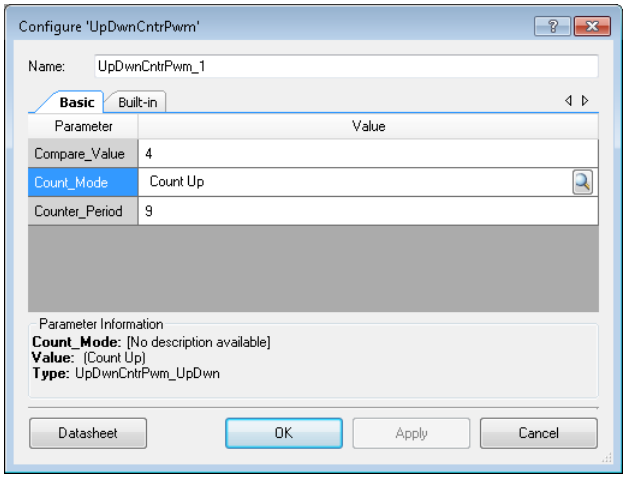
Figure 63. Down-Counting PWM Waveforms



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.

35. Go back to the project schematic and double-click the UpDwnCntrPwm Component to open the properties dialog.
36. Change Count_Mode to 'Count Up', as Figure 64 shows.

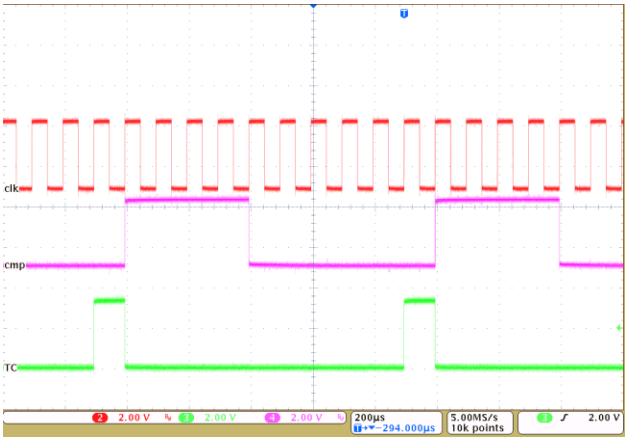
Figure 64. Setting the PWM to Count Up



37. Click OK to apply the changes.
38. Select File > Save All, build the project, and program the PSoC.

You can observe that this is an up counter, because the cmp value is HIGH after a TC as the counter is reloaded with zero at that point, as Figure 65 shows.

Figure 65. Up-Counting PWM Waveforms



Notice that the output was HIGH after the TC 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.

8 Project #3 – Simple UART

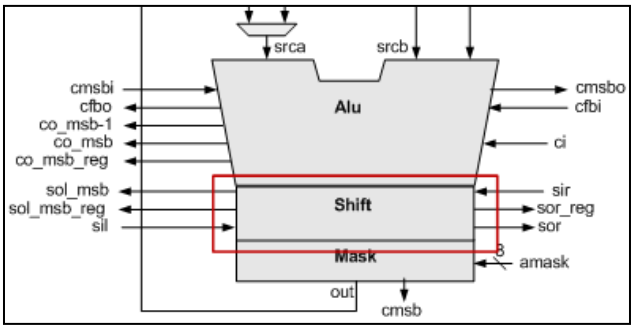
This example project demonstrates a simple TX UART created with a single datapath. We do not walk you through each step of creating the Component. Instead, you can review the Component and UDB Editor document found in the associated example project. Find the Component, called "Simple_Tx" in the *Simple_Tx*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

8.1 TX UART Component Details

This Component uses datapath operations of shifting and loading a value into A0 from F0.

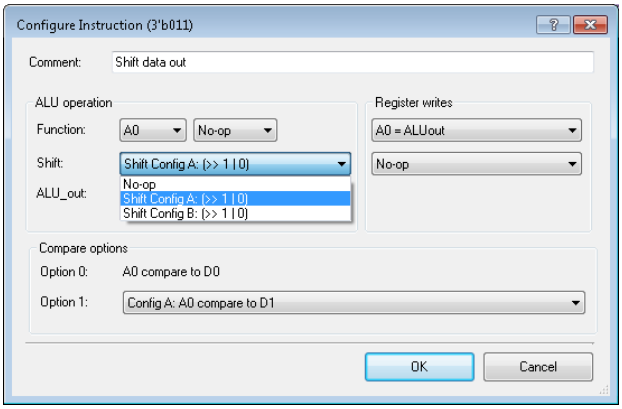
There is a shifter at the output of the ALU, as Figure 66 shows.

Figure 66. Shifter Block



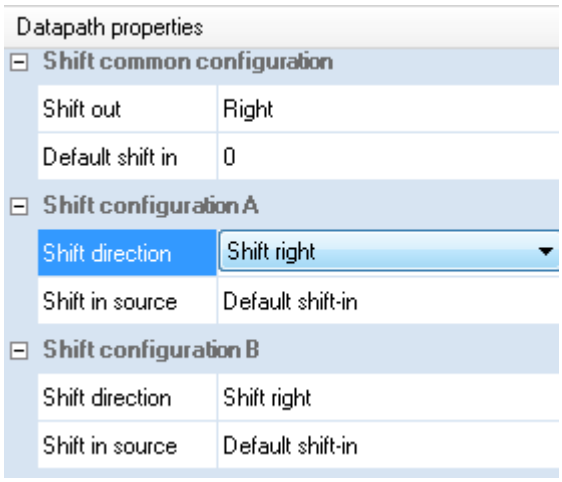
This shifter can shift bits either left or right. Each instruction of the datapath can independently set the operation of the Shifter. This option is set in the **Configure Instruction** dialog of each instruction, as Figure 67 shows.

Figure 67. Shift Operation Settings



To set the configuration of Config A and Config B, go to the **Datapath properties** and configure **Shift configuration A** and **Shift configuration B**; see Figure 68.

Figure 68. Datapath Shift Configuration



The shift in source can either be the Shift In (SI) signal from the DSI, or it can be the default shift in value (0,1).

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. You must configure this mux under the **Shift common configuration** *Shift Out* in Figure 68.

In this example, you create a state machine that controls the instruction of the datapath. This state machine, which is implemented in the UDB PLDs, determines which part of the UART transmission occurs next: IDLE, Start, Data, or Stop.

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

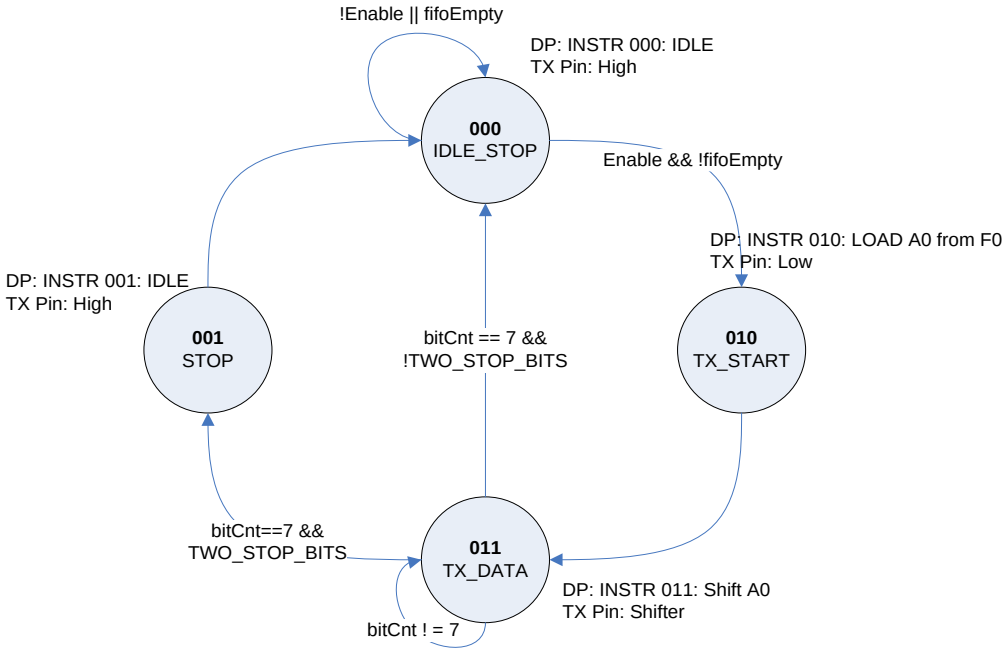
Table 6. Example 3 Datapath Instructions

State	INSTR_ADDR	Function	SHIFT	Register Writes
IDLE_STOP	000	No-op	None	None
STOP	001	No-op	None	None
TX_START	010	No-op	None	A0 = F0
TX_DATA	011	No-op	SR	A0 = ALU

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

Most complex datapath Components require a state machine to sequence the datapath operations. Figure 69 shows how the simple TX state machine works.

Figure 69. TX UART State Machine Flowchart

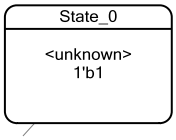


First, the state machine waits for new data to be written to the FIFO by monitoring the fifoEmpty status bit. Once there is data in the FIFO, the state machine moves to the Start state and sends a START bit by setting the TX line LOW. During this state, the datapath loads the value in FIFO (F0) into A0.

In the next state, the datapath shifts out the data in A0 to the TX pin LSB first (right shift). Next, the state machine sends out one or two STOP bit.

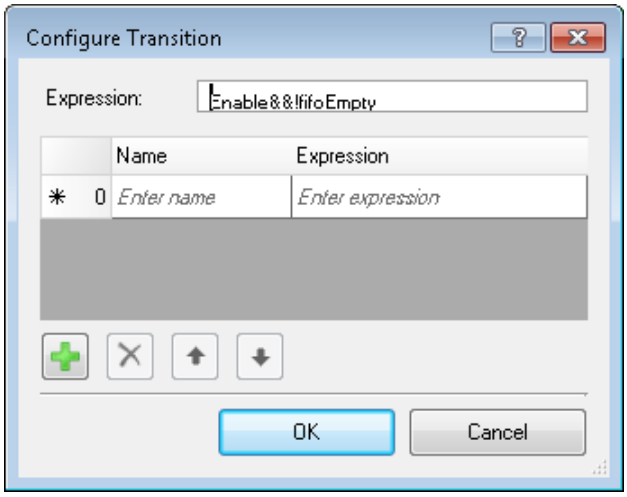
With the UDB Editor you can duplicate the state machine shown in Figure 69. To do this, drag and drop state machine elements onto your design canvas; see Figure 70.

Figure 70. State Machine Element



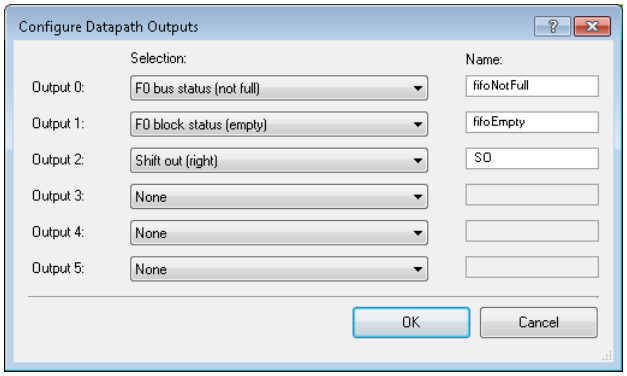
Drag four of these onto your design canvas and connect them as shown in Figure 69. Each time you draw a wire to connect the states a dialog appears which allows you to write an expression that determines when the transition occurs; see Figure 71.

Figure 71. State Transitions Expression



The transition shown in Figure 71 occurs when the Enable Signal is HIGH, and the fifoEmpty signal is not HIGH. This is the transition that controls when the state machine moves from the IDLE state to the Start state. For the transition to occur, the Enable signal that comes from a control register must be HIGH and the datapath FIFO must not be empty. The fifoEmpty signal comes from one of the datapath outputs; see Figure 72. This signal is HIGH when there is no data in the FIFO, and LOW when there is data in the FIFO.

Figure 72. Datapath Outputs



Within each state, you can add variables and perform logic and arithmetic functions on these variables. For example, the simple UART shifts out eight bits of data. This means that the datapath must remain in the Data (shift) state for eight clock cycles.

Table 6 shows that the shifting occurs when INSTR_ADDR is 011b. Figure 73 shows the configuration for state three (011b). Note that under **Variable assignment,** shown in Figure 73, the variable bitCnt has the **Expression** bitCnt +1. This means each time this state is executed, bitCnt = bitCnt+1.

The states in the state machine run at the positive edge of the input clock, the same clock that the datapath runs on.

Figure 73. State Configuration

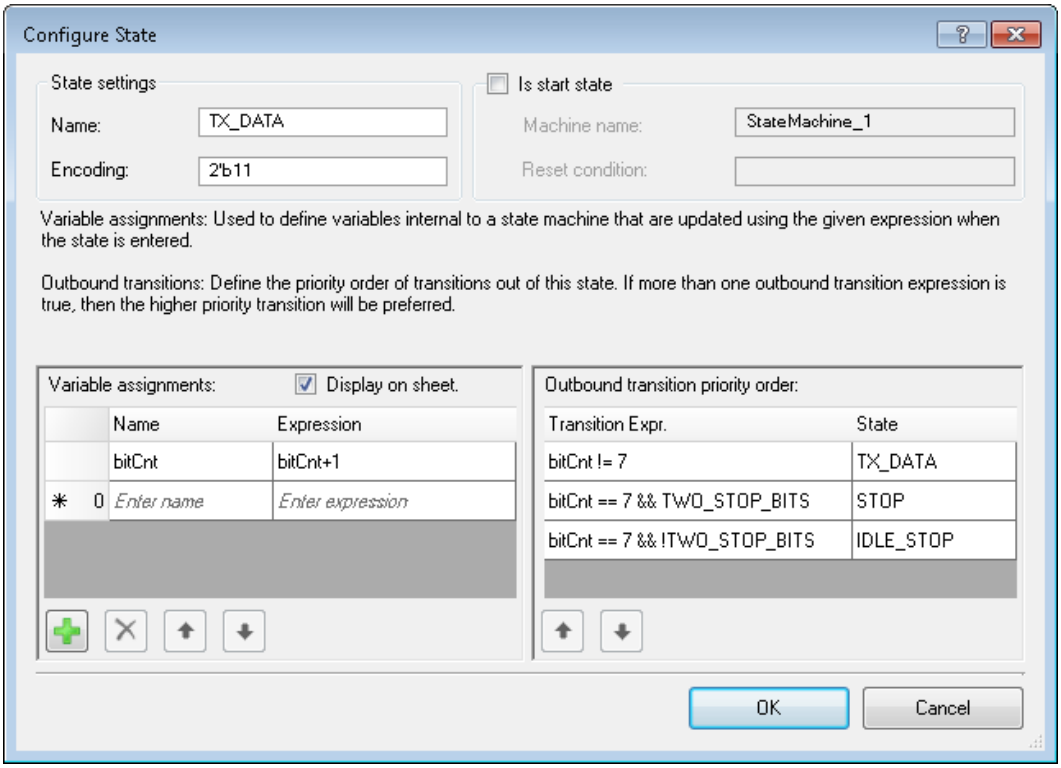
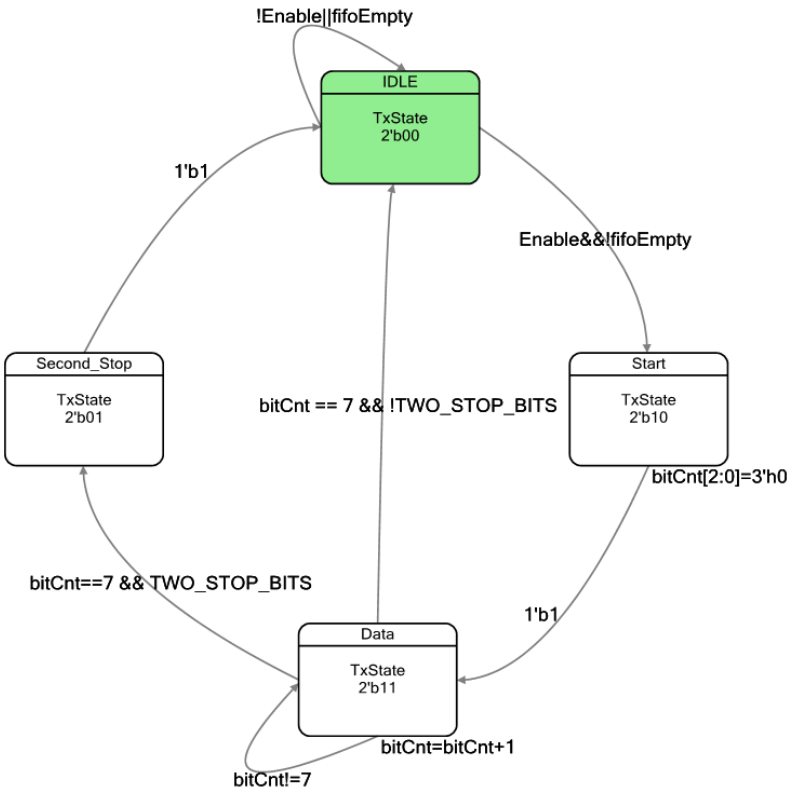


Figure 73 shows the **Outbound transition priority order** dialog, which is used to configure the state transitions. Notice that the state machine stays in the Data state as long as bitCnt does not equal 7. Previously, we said we must shift out 8 bits, so why we are we staying in this state for 7 cycles? We are in fact staying in this state for 8 cycles. The transitions are evaluated before bitCnt is increased. When this expression is evaluated for the first time, bitCnt is zero.

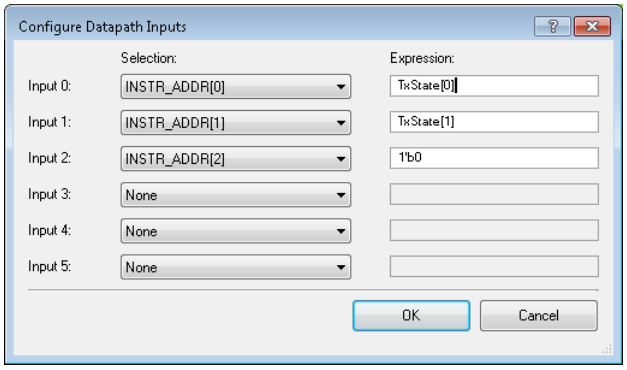
The same steps must be taken with each state and each state transition to match Figure 69. Figure 74 below shows Figure 69 duplicated with the UDB Editor.

Figure 74. UDB Editor State Machine Diagram



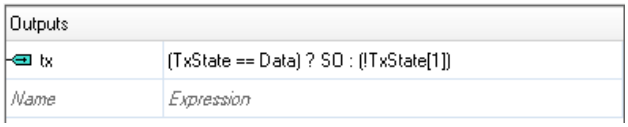
Next, control the Datapath Instructions with the state machine. As you can see in Figure 74 the state machine is named *TXState*. Because there are four states in this machine the UDB Editor creates a 2-bit signal wire for TxState. This 2-bit signal can be routed to the INSTR_ADDR signals of the datapath; see Figure 75.

Figure 75. Datapath Inputs Controlled by State Machine



Next, you may be wondering how to control the TX output line. As stated, when in the Data state, data is shifted out. The Start state sets the TX line LOW, and the stop and IDLE state set it HIGH. This is done through the Output Expression shown in Figure 76.

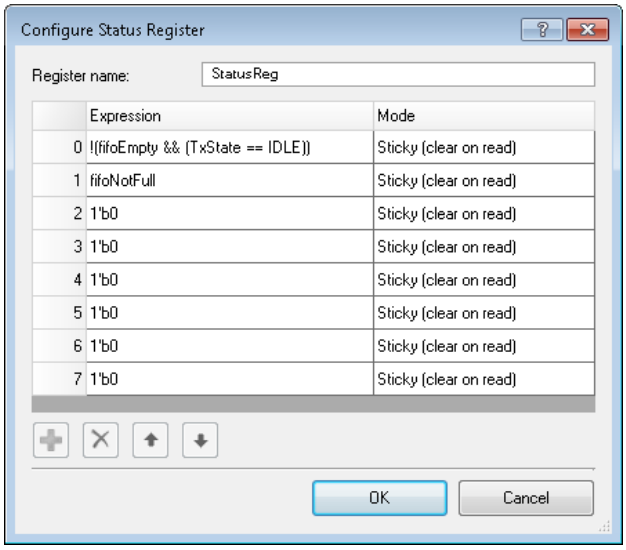
Figure 76. Tx Output Expression



If it is not in the data state it drives the inverse of the msb of 'TxState' on the TX line. For Start, the msb is 1, so it outputs a '0'. For Stop and Idle, the msb is 0, so it outputs a '1'. In the Data state it outputs the SO (Shift Out) signal.

The only other new element to this design is the addition of a status register. The purpose of a status register is to report the status of the state machine and the datapath to the CPU.

Figure 77. Status Register Configuration



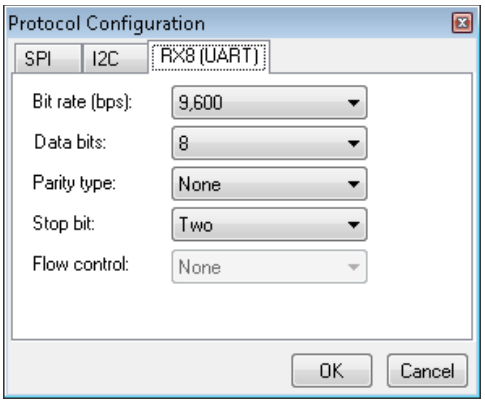
The first bit of this status register tells the CPU when the UART is busy. This status can be used to wait until the UART is done sending the data. The second bit reports if the FIFO is not full. This signal can be used by the CPU to load the FIFO with the data until the FIFO is full.

A header file has been added to this Component. The header file defines some constants for enabling the Component; setting it in one stop bit mode or two stop bits mode, and status register defines. The thing to pay attention to in the header file is that the defines must match the bit positions chosen in the status register and control register.

The main code for this project enables the Component with two stop bits, and then continuously transmits the values 0 through 10. It does this at 9600 baud. Configure your receiver for 9600 baud and two stop bits.

Installed with PSoC Creator is a program called the Bridge Control Panel (BCP). You can use the BCP to receive RX characters. In the BCP, connect to the COM port that you have connected the TX output to. In **Tools > Protocol Configuration,** configure the RX8 (UART) as Figure 78 shows.

Figure 78. BCP RX8 Configuration



In the editor of BCP add the following text: rx8 x x x x x x x x x x. This reads 11 bytes from the selected COM port. You can then hit the Repeat button to continuously receive the data.

9 Porting from UDB Editor to Datapath Configuration Tool

There may come a point where you are not able to implement the functionality you want with the UDB Editor. At that point your only option is moving to a Verilog file and the Datapath Configuration Tool. This step is fairly simple. Follow the instructions in Appendix A for creating a Verilog file and then copy and paste the Verilog code generated by the UDB Editor to your new Verilog file. Now, you can modify the Verilog file with the Datapath Configuration Tool to your heart's content. For examples of creating Components and using the Datapath Configuration Tool, see Appendix A.

10 Summary

UDB datapaths increase flexibility when creating Components in PSoC programmable logic. Understanding and effectively using UDB datapaths allow you to extend the capabilities of PSoC 3, PSoC 4, and PSoC 5LP beyond what traditional microprocessors can offer.

The example projects described in this application note are just the starting point for you to create your own customized solutions. To learn more about adding features and complexity to your Components, read the PSoC Architecture TRMs and the Component Author Guide.

11 Related Resources

11.1 Application Notes

- AN54181 – Getting Started with PSoC 3
- AN79953 – Getting Started with PSoC 4
- AN77759 – Getting Started with PSoC 5
- AN221774 – Getting Started with PSoC 6 MCUs
- AN81623 – PSoC 3 and PSoC 5 Digital Design Best Practices
- AN82250 – PSoC 3, PSoC 4 and PSoC 5 Implementing Programmable Logic Designs

11.2 KB Articles

- KBA86838 – Datapath Configuration Tool Cheat Sheet
- KBA86336 – Just Enough Verilog for PSoC
- KBA86338 – Creating a Verilog-based Component
- KBA81772 – Adding Component Primitives / Verilog Components to a Project
- Basics of Verilog and Datapath Configuration Tool for Component Creation

11.3 TRMs

- PSoC 3 Architecture TRM
- PSoC 4 Architecture TRM
- PSoC 5LP Architecture TRM
- PSoC 6 MCU Architecture TRM

11.4 Videos

The following videos introduce the PSoC Creator and Verilog Component creation process:

11.4.1 Basics

- Creating a New Component Symbol
- Creating a Verilog Implementation

11.4.2 Component Creation

- PSoC Creator 113: PLD Based Verilog Components
- 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
- PSoC Creator Tutorial: Using the UDB Editor – Part 1
- PSoC Creator Tutorial: Using the UDB Editor – Part 2
- PSoC Creator Tutorial: Using the UDB Editor – Part 3
- PSoC Creator Tutorial: Using the UDB Editor – Part 4
- PSoC Creator Tutorial: Using the UDB Editor – Part 5
- PSoC Creator Tutorial: Using the UDB Editor – Part 6

12 About the Authors

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.

Name: Greg Reynolds

Background: Greg Reynolds was with Cypress in several roles for more than a decade.

A Appendix A – Examples with Datapath Configuration Tool

This appendix shows you how to create the same proejcts that were created in the main body of the application note, except this time the datapath configuration tool is used. Also a parallel in and parallel out example is added. Here is a list of the projects created in this appendix:

- 8-bit down counter
- 8-bit PWM
- 16-bit PWM
- 8-bit up/down counting PWM
- TX-only simple UART
- Parallel input and parallel output

You can use the example projects on any PSoC 3, PSoC 4, or PSoC 5LP device. Completed example projects are available on this application note's landing page on Cypress' website.

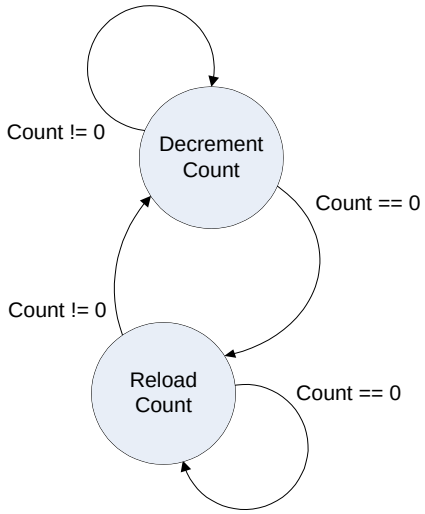
A.1 Project #1 – 8-Bit Down Counter

The purpose of this project is to introduce you to the steps you must take to create a simple datapath-based Component. This is demonstrated by creating a simple 8-bit down counter.

A.1.1 8-Bit Counter Component Details

A simple down-counter can be represented by a state machine with two states, as Figure 6 shows.

Figure 79. 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 datapath instructions to implement this counter. The first instruction decrements the count value. The second reloads the count with the initial value.

The two datapath registers used in this example are:

- A0 – holds the count value and is decremented by the ALU.
- D0 – holds the value that is reloaded into A0 when the count reaches zero.

In order for this counter to work, you need a method to decide which instruction the datapath is executing: loading or decrementing. Figure 6 shows that transitions are controlled by the value of the count, that is, whether the count is zero or not.

The datapath has a zero detector block (ZDET) that monitors the value of the data in A0 and A1. The block has two outputs, z0 (A0 == 0) and z1 (A1 == 0), which indicate the condition of A0 and A1, respectively. Each output is HIGH when the value is zero and LOW when the value is non-zero.

In this example, the z0 output is used to control which instruction is executed.

Remember the datapath can have 8 uinque instructiosns. The instruction used by the datapath is choosen by a 3 bit address line. As stated before of this counter only two instructiosn are needed, so you can tie bit 0 to z0 and hold the other bits LOW. Table 2 is the transition table.

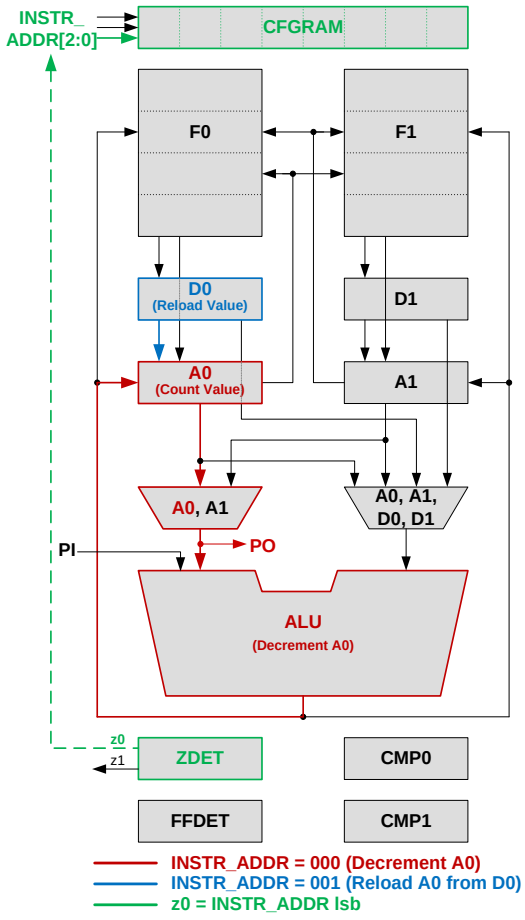
Table 7. Datapath Instructions

CFGRAM Instruction	Instruction Address Bits (INSTR_ADDR)			Operation
	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 in A0 reaches zero, z0 becomes '1'. This causes the datapath instruction to be "Reload Count". When the count is reloaded from D0 into A0, z0 becomes '0' and the instruction becomes "Decrement Count".

To visualize how this works, look at a highlighted datapath block diagram shown in Figure 7.

Figure 80. Simple Counter Block Diagram With Highlights



A.1.2 8-Bit Counter Component Creation Steps

1. Launch PSoC Creator and create a new Design Project targeted at the device you are using. Select an Empty Schematic to start from, set the workspace name and project name to AN82156_Appendix.

Components are stored in a PSoC Creator library project. Choose unique names for your libraries and Components so that they are not confused with the standard Cypress Component libraries.

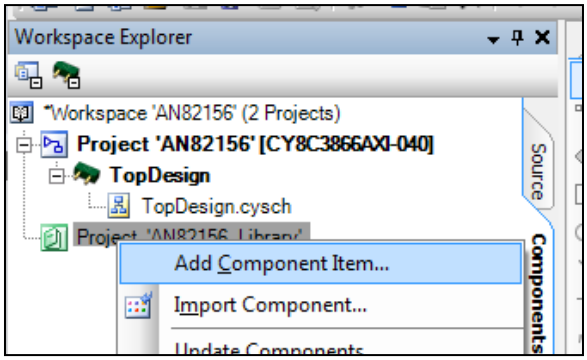
2. Right-click **Workspace 'AN82156_Appendix'** in the **Source** tab of the Workspace Explorer, and select **Add** > **New Project…** from the drop-down menu that appears.
3. Select the **Library project**, Name it "*AN82156_Appendix_Lib*", and leave the location as its default value. This creates the library in the same location as the *AN82156_Appendix* workspace.

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

Next, you need to add a Component to the new library:

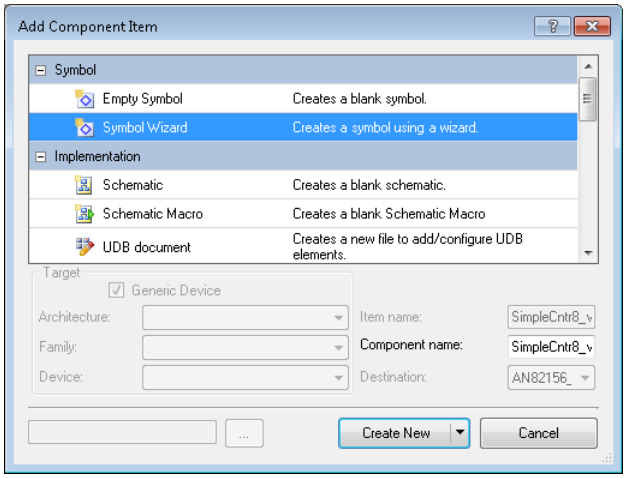
4. Switch to the **Components** tab of the Workspace Explorer and right-click **Project 'AN82156_Appendix_Lib'**. Select **Add Component Item…** from the drop-down menu, as Figure 81 shows.

Figure 81. Add Component Item



5. Select the **Symbol Wizard** Component template and name the Component *"SimpleCntr8_v1_0"*, as Figure 82 shows.

Figure 82. Use Symbol Wizard



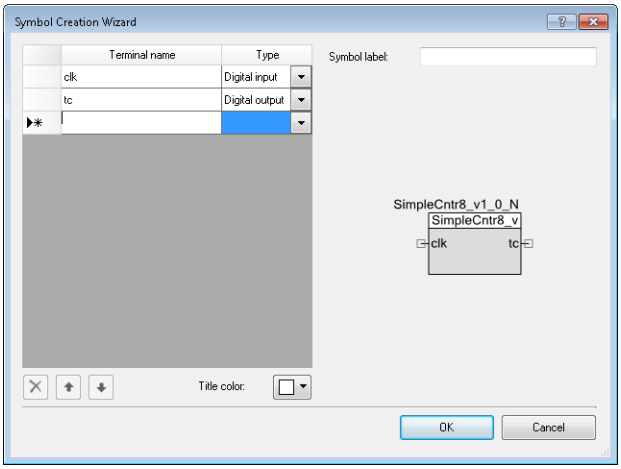
It is a good practice to include a version number in the Component name. Append to the Component name the tag "_vX_Y", where 'X' is the major version and 'Y' is the minor version. PSoC Creator has versioning capabilities that can help you track and use multiple versions of your Components.

6. Click the **Create New** button to launch the Component symbol wizard.

   The wizard asks you to define the inputs and outputs, and it uses this information to create a Component symbol.

7. Add two terminals in the **Terminal Name** field – a "*clk*" input and a "*tc*" output, as Figure 83 shows.

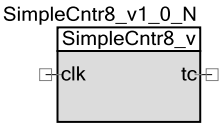
Figure 83. Add Inputs and Outputs



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

8. Click **OK** to generate the symbol in the symbol editor page, as Figure 84 shows.

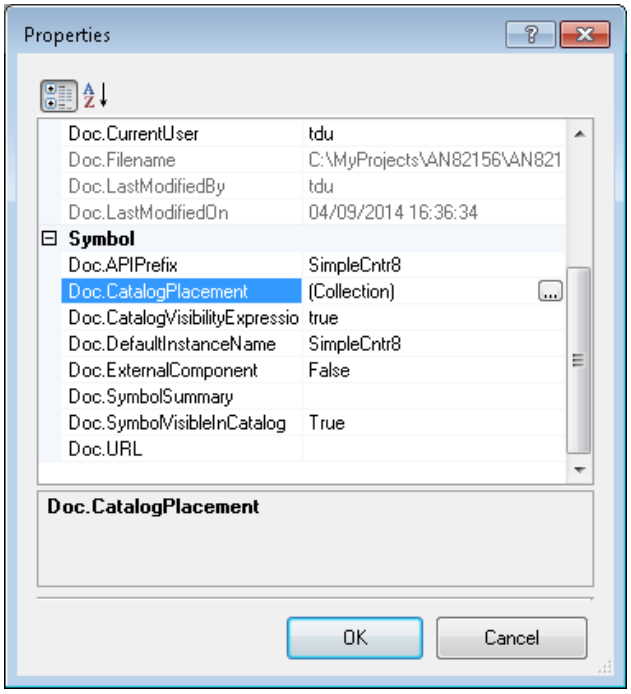
Figure 84. Generated Component Symbol



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

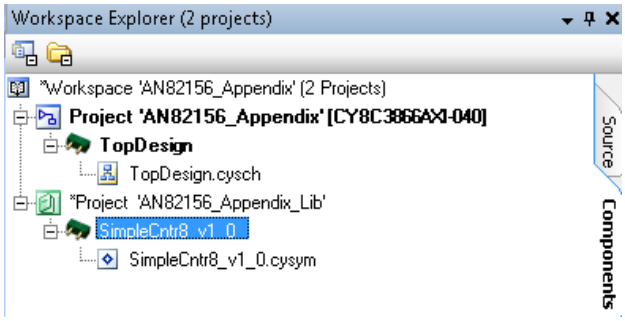
9. Right-click an empty space in the symbol editor – not the symbol itself – and select Properties from the drop-down menu.
10. Enter values in the Symbol section of the property fields, as Figure 85 shows:
    - **Doc.APIPrefix** = *SimpleCntr8*.
      This value is prefixed to any API file names generated for the Component. You do not generate any API for this example, but enter a value here whenever you create a Component.
    - **Doc.CatalogPlacement = AN82156_Appendix/Digital/Cntr8.**
      Click the '**…**' to open the Catalog Placement dialog to enter this value. PSoC Creator uses this value to define the hierarchy of the Component catalog. The first term is the tab under which the Component appears in the catalog. Each subsequent '/' represents a sublevel. The hierarchy must contain at least one sublevel. The value shown here indicates that the Component is visible as 'Cntr8' in the 'Digital' sublevel of the AN82156_Appendix tab.
    - **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 85. Add Symbol Properties



11. Perform a Save All (Ctrl+Shift+S) to ensure that all changes have been applied to the project. The new symbol shows up under **AN82156_Appendix_Lib** in the **Components** tab of the Workspace Explorer, as Figure 86 shows.

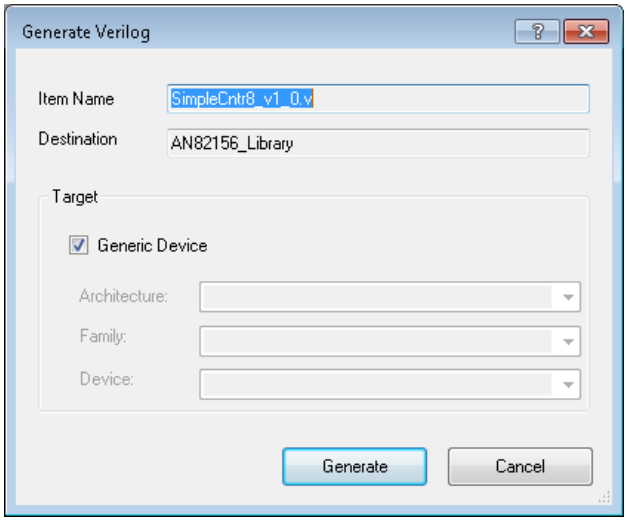
Figure 86. New Component in the Library



Next, you need to link the schematic symbol to a datapath implementation.

12. Right-click an empty space in the Symbol Editor page and select **Generate Verilog** from the drop-down menu.
13. Leave all settings in the Generate Verilog dialog box at the default values and click **Generate**; see Figure 87.

Figure 87. Generate Verilog Dialog

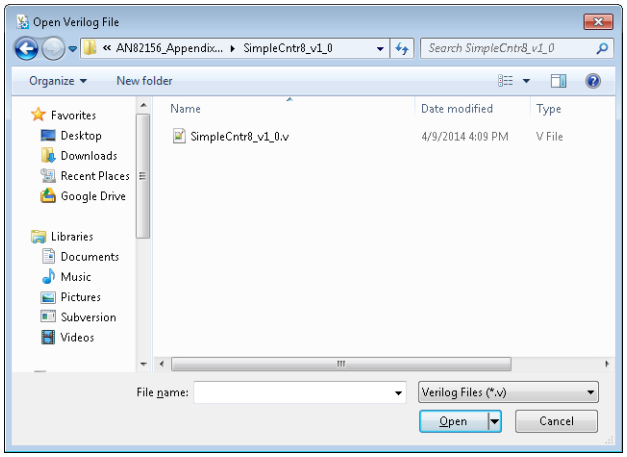


14. Perform a Save All to ensure that all changes have been applied to the project. A new Verilog file is generated and added to the Component.

    The Verilog file is where the datapath implementation goes, as well as all the Verilog code needed to control the datapath. Appendix C has an example of a new Component Verilog file. To add and edit datapath instances, use the *Datapath Configuration Tool*.

15. Launch the Datapath Configuration Tool.
16. You can do this by navigating to **Tools** > **Datapath Config Tool**…. You can also launch the Datapath Configuration Tool from the Start menu (**Start** > **All** Programs > **Cypress** > **PSoC Creator 3.x** > **Component Development Kit** > **Datapath Configuration Tool**).
17. In the Datapath Configuration Tool, select **File** > **Open** and browse to the location of the *SimpleCntr8_v1_0.v* file generated by PSoC Creator. If you followed the steps in this example, the file is located in the *AN82156_Appendix_Lib* project folder inside the *AN82156_Appendix* workspace folder, as Figure 88 shows.

Figure 88. Example Component Verilog File

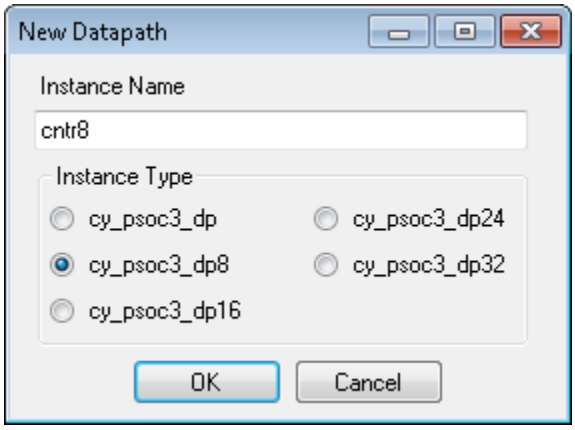


18. Select the file and click **Open** to load the Verilog file into the datapath tool.

    The first step is to create a new datapath configuration.

19. Select **Edit** > **New Datapath** from the menu to open the New Datapath dialog window.
20. Enter an **Instance Name** (use "*cntr8*") and select the **cy_psoc3_dp8** Instance Type, as Figure 89 shows. Click **OK**.

Figure 89. New 8-Bit Datapath Instance



This is an 8-bit counter. You do not need to define the datapath to be anything larger than eight bits (cy_psoc3_dp8). The other example projects demonstrate Components wider than eight bits.

Note cy_psoc3 is a generic name. These datapath instances will work on any device.

21. Click **File** > **Save** to save the configuration to the Verilog file.

    The Datapath Configuration Tool instantiates a datapath construct in the Verilog file the first time you save a configuration, as shown in Appendix C. You can modify the configuration to implement the counter functions.

22. Select values in the Reg0 and Reg1 fields to configure the datapath, as Figure 94 shows. The other fields can remain at their default settings.

23. Reg0 is the same as Configuration 0 or Instruction 0, Reg1 is the same as Configuration 1 or Instruction 1.

Figure 90. Configuration for the SimpleCntr8 Component

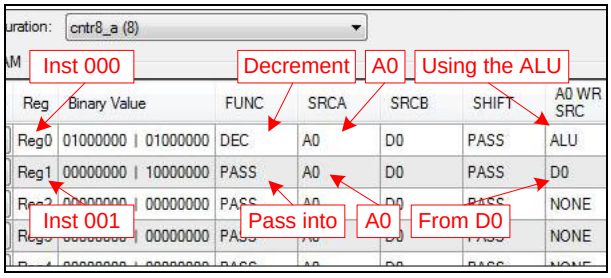


Table 8. Example 1 Datapath Configuration

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

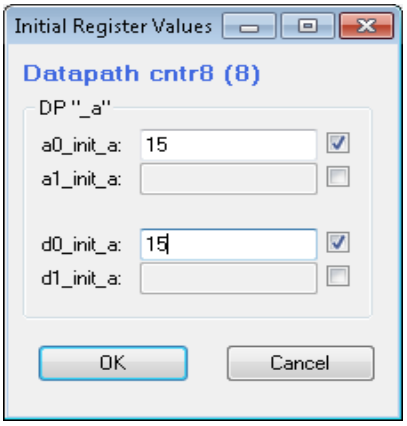
24. The Reg0 and Reg1 rows represent the first two of the eight datapath configurations stored in the CFGRAM. You configured the first one to decrement the counter and the second one to reload the counter.

    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 the *main.c* file 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.

25. Select **View** > **Initial Register Values** to open the register value dialog.

26. Check the boxes for **a0_init_a** and **d0_init_a**. Set the values to *15* to define a starting and reload count, as Figure 91 shows.

Figure 91. Initial Register Values for the SimpleCntr8



27. Click **OK** when done.

    In this case, the D0 and A0 values are identical so that you reload the same count value into A0 that you started with. You can choose different values if you want the first period to be different from the rest. In this case, the period is 16 because the count sequence goes from 15 to 0.

28. 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 PSoC Creator.

    The Datapath Configuration Tool made changes to the Verilog file when you saved the configuration. PSoC Creator may ask to reload the Verilog file when you switch back.

    At this point, you need to make changes to the Verilog file to link the schematic symbol to the datapath logic. The section of code discussed here starts at line 77 in the Verilog file.

29. Change `.clk(1'b0)` to `.clk(clk)`. This links the datapath clock to the symbol's 'clk' terminal. Use this terminal to set the speed at which the datapath operates.
30. 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.
31. Change the `.cs_addr(3'b0)` text to `.cs_addr({2'b00,tc})`. This sets the two most significant bits of the CFGRAM address to '0' and sets bit 0 to the value of `tc`.

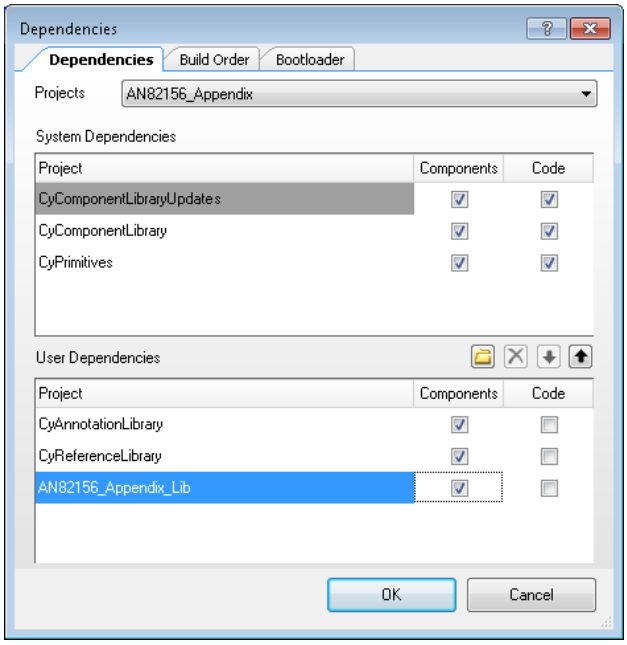
Remember that `tc` always reflect the value of `z0`. This means that `z0` determines the configuration; see Table 7.

32. Save all changes to the Verilog file. Appendix C shows how the finished Verilog code looks if everything was entered properly.

Now that the Component is ready for use, set *AN82156_Library* as a project dependency before you can see the Component in the project's Component Catalog.

33. On the **Source** tab, right-click **Project 'AN82156_Appendix'** and select **Dependencies… .** Add the *AN82156_Appendix_Lib* as a user dependency of the AN82156 project –check the Components box, as Figure 92 shows. To do this click on the folder icon and navigate to the .cylib file for the library you created.

Figure 92. Adding MyLibrary as a Project Dependency



The *AN82156_Appendix_Lib* Components are now visible in the Component Catalog under the *AN82156_Appendix* tab, as Figure 93 shows.

Figure 93. 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.

34. Place the Cntr8 Component in the project schematic.
35. 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 on a scope.
36. Connect a Digital Output Pin Component 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 values.

    Note: For PSoC 4, you cannot directly route a clock out a pin. To route the clock to the pin follow these instructions:

    a. Place a Digital Output pin Component on your schematic.
    a. Select the **Clocking** Tab in the pins customizer.
    b. Set the **Out Clock:** to *External.*
    c. Go back to the **Pins** tab, and go to the **Output** subtab.
    d. Under **Output Mode:** select *Clock.*
    e. Click OK.
    f. Connect the clock signal to the out_clk terminal on the pins Component.

    Note: For PSoC 6 MCU, you cannot directly route a clock out to a pin. To route the clock to the pin, follow these instructions:

    a. Place a Digital Output Pin Component on your schematic.

b. Place a TFF Component on your schematic.

c. Connect the output of the clock Component to the clk terminal of the TFF Component.

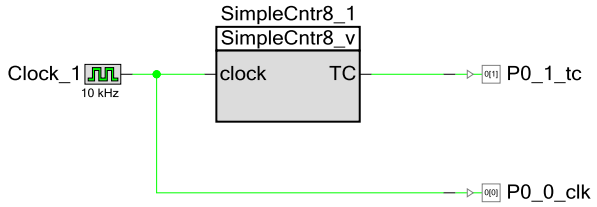
d. Connect a Logic High ‘1” Component to the t input of the TFF Component.

e. Connect the q output of the TFF Component to the Digital Output Pin placed in Step a.

Note: The clock frequency seen on the outside of the device will be half the actual value used by the UDB Component.

37. Connect a Digital Output Pin Component to the 'tc' terminal. Name it P0_1_tc and leave all other settings at their default values. This pin is set HIGH when the count is zero. Figure 94 shows the completed project schematic.

Figure 94. Simple Counter Project Schematic



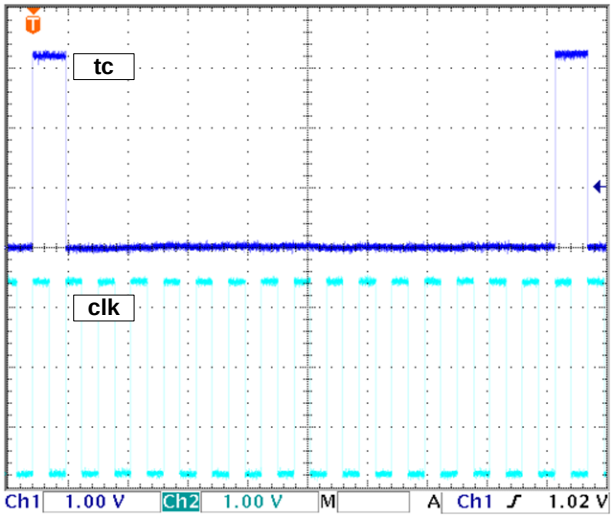
38. Assign the pins to P0[0] and P0[1], according to their names, in the *Pins* tab of the cydwr.

You are ready to build the project and program the PSoC. You can observe the clock and the terminal count on pins P0[0] and P0[1].

39. Save the project, build it, and program the PSoC.

If you connect a scope to the output pins, you can observe the 'clock' and 'tc' outputs, as Figure 95 shows.

Figure 95. Simple Counter Outputs



You loaded A0 with a starting value of 15, so you can see that the period is 16 (Counts down from 15 to 0) 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 configuration transitions back to begin decrementing A0 again.

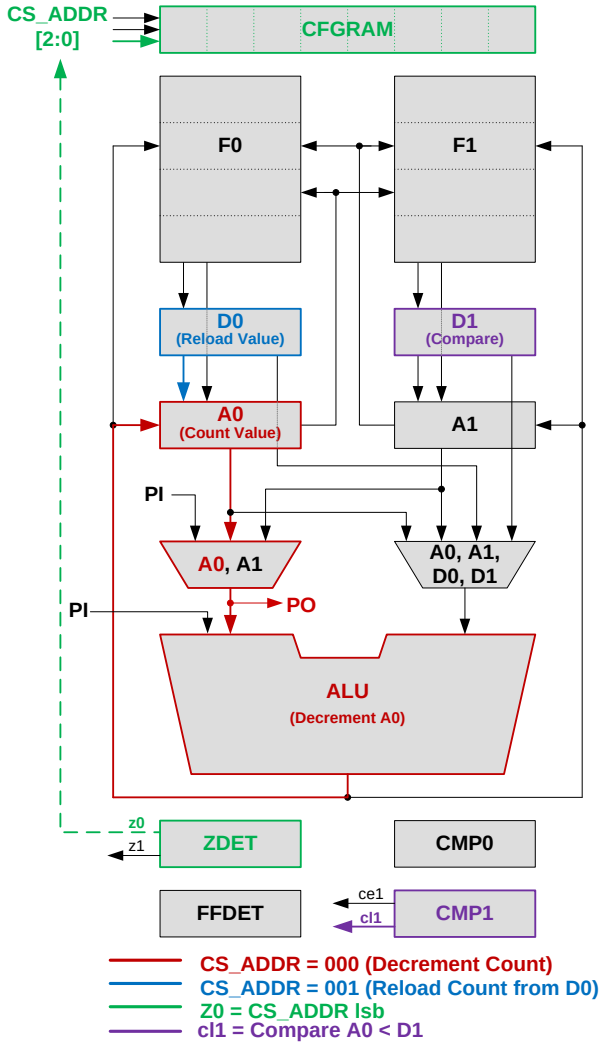
You have just designed your first datapath-based Component, and you didn't write any firmware to do it!

A.1.3 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. 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 96 shows:

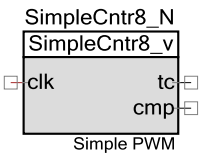
Figure 96. Simple PWM Block Diagram With Highlights



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 SimpleCntr8_v1_0.
2. Press 'o' to add a digital output terminal. Name it “cmp” and place it as Figure 97 shows.

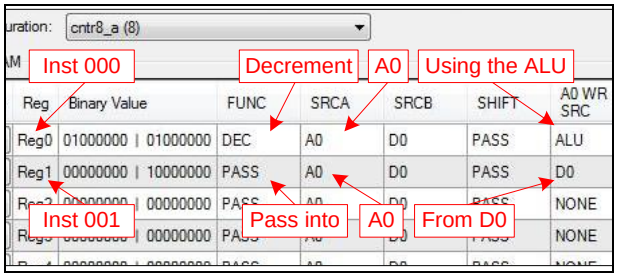
Figure 97. Component Symbol With 'cmp' Terminal



3. Do a Save All to capture the changes.
4. 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 first configuration that is present.

   We said earlier that a PWM is just a counter with a compare. You still need only two configurations, 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 98 shows.

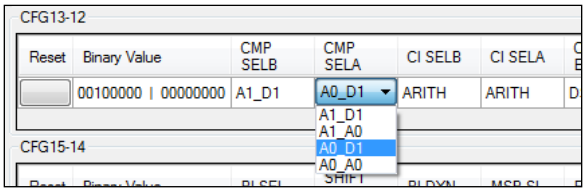
Figure 98. CFGRAM Settings for PWM



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

5. Set **CMP SELA** to "*A0_D1*" to configure the compare function to use A0 and D1 for the compare, as Figure 99 shows.

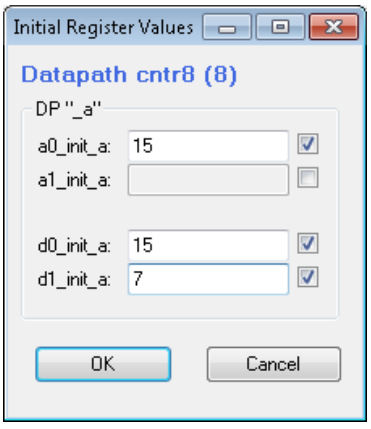
Figure 99. Compare Block 1 Mux Settings



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

6. Select **View** > **Initial Register Values** from the menu and open the dialog window.
7. Check the box for **d1_init_a**.
8. Leave the 'a0' and 'd0' values at *15*. This sets the period to 16 clock cycles.
9. Set the 'd1' value to *7*. This sets the compare value to 7, as Figure 100 shows.

Figure 100. Initial PWM Register Values



10. Click **OK** to apply the values to the registers.

11. Save the changes in the Verilog file.

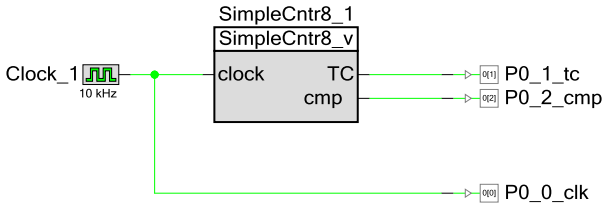
    The compare block outputs HIGH whenever A0 is less than D1. Link the symbol to the datapath hardware using the Verilog file, just as you did with the counter.

12. Close the Datapath Configuration Tool and open the Verilog file in PSoC Creator.
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 AN82156_Appendix 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 101 shows.

Figure 101. Simple PWM Project Schematic

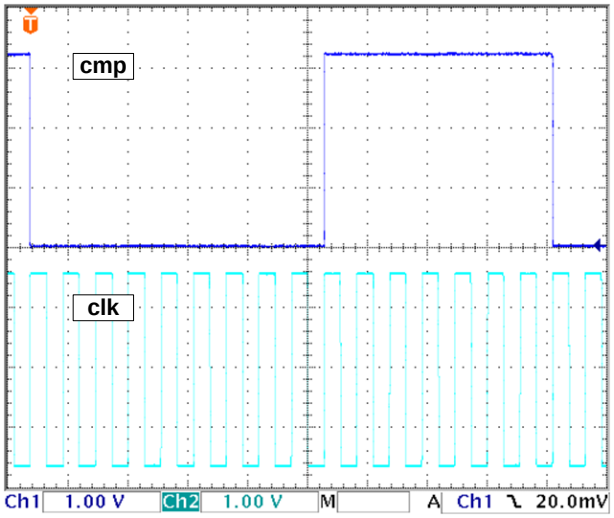


You are ready to build the project and program the PSoC. The clock and the terminal count can still be observed on pins P0[0] and P0[1]. The PWM output can be observed on P0[2].

11. Save the project, build it, and program the PSoC.

If you connect a scope to the output pins, you can observe the 'clock', 'tc', and 'cmp' outputs. Figure 102 shows the 'clk' and 'cmp' signals.

Figure 102. Simple PWM Outputs



You loaded A0 and D0 with a starting value of 15, 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.

A.1.4 Adding Parameters

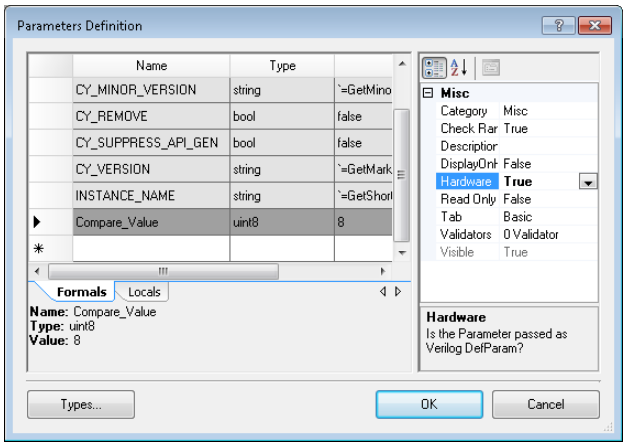
It is inconvenient to modify Verilog code whenever you need to make a change to one of the Component's parameters. And, what if you needed two PWMs with different periods and compare values? You can add user-configurable parameters to your Component, similar to the way most Cypress Components work.

1. Open the Component's Symbol Editor page (.cysym) and right-click 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-hand side of the window, as Figure 103 shows.

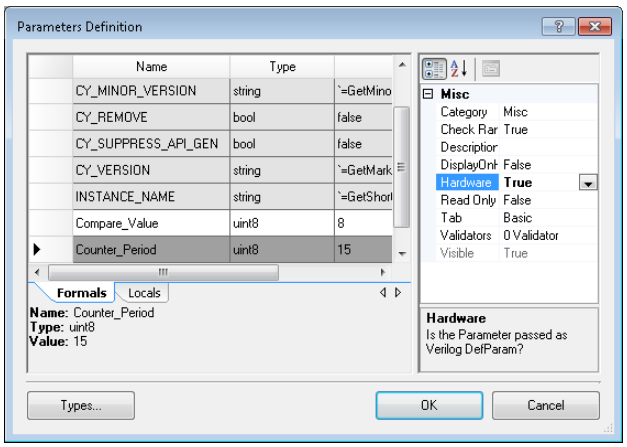
   This exposes the parameter to the Verilog so that the UDB hardware can use it.

Figure 103. 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 104 shows.

Figure 104. Adding Another New Component Parameter



7. Click **OK** and do a Save All to apply the changes to the Component.

   You need to link the new Component symbol parameters to the Verilog logic.
8. Open the Component's Verilog file and locate the text, on or near line 25, 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 29 that holds the initial register values:

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

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. Later examples show you how to do this using Datapath Configuration Tool.

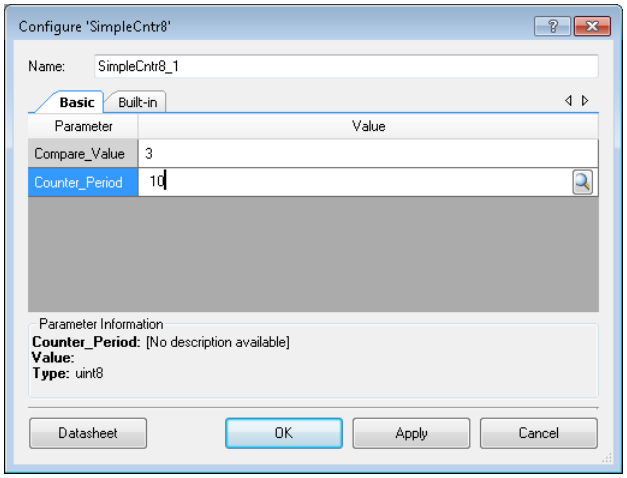
You can set these at compile time without modifying the Component.

12. Do a Save All, build the project, and program the PSoC.

    Previously, you set the default compare value parameter to 7. Now, you can change the parameters from within the project schematic.

13. Go back to the project schematic and double-click the SimpleCntr8_1 Component to open the properties dialog.
14. Change the compare value to '3' and the counter period to '10', as Figure 105 shows.

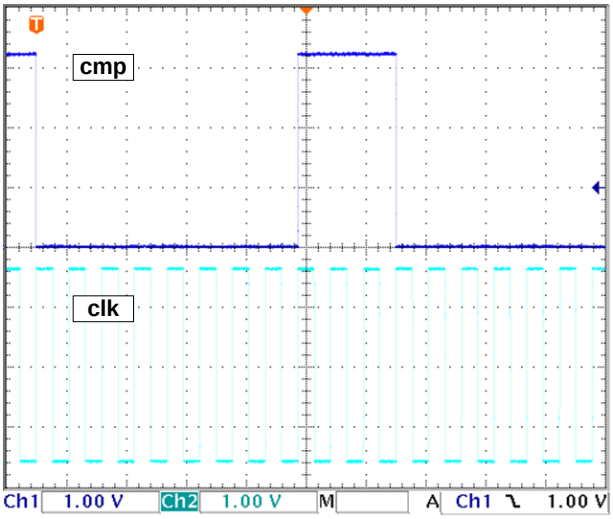
Figure 105. Selecting Component Parameters



15. Click **OK** to apply the changes.
16. Do a Save All, build the project, and program the PSoC.

As you can see in Figure 106, the period and compare output have changed.

Figure 106. PWM Output With New Parameters



You set the period to 11 cycles (the period is 10+1 because the counter goes from 10 to 0 before it reloads), and the compare to 3. The result is 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 a uint8. You can even place multiple instances of the Component in your project and set them to different values.

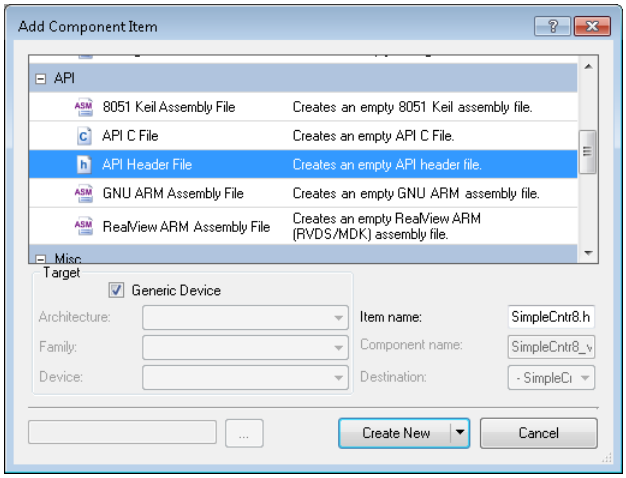
For more information on adding Component parameters, including how to set limits on what value the user can enter, see the *Component Author Guide*.

A.1.5 Adding Header Files

In addition to being able to change the behavior of the PWM at design time, you can change the PWM during run time by modifying the PWM registers via C code. For example, you used register D0 to hold the period value and register D1 to hold the compare value. To make these registers easy to access, create a header file that defines the registers used.

1. In the **Components** tab, right-click **SimpleCntr8_v1_0**, click **Add Component Item**.
2. In the Add Component Item window, navigate down to the **API** section and click **API Header File**.
3. Change the **Item name** to *SimpleCntr8.h,* as Figure 107 shows.

Figure 107. Add Header File



4. Click **Create New**. This adds a header file to your Component.

   It is time to add definitions for the compare value (D1) and the period value (D0).

5. Add the following definitions above `//[]END OF FILE` in the header file:

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

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

   These two definitions allow you to directly write to the D0 and D1 registers in firmware. The *cyfitter.h* file contains a set of definitions for the registers in the Components that are used in the project. If you have done anything different from the steps described in this application note, then you may need to locate the register definitions in the *cyfitter.h* file and use them instead of what we show here.

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

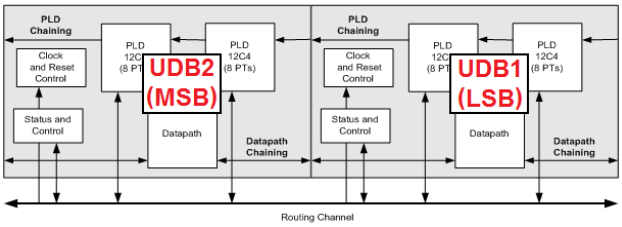
```
SimpleCntr8_1_Period_Reg = 0x08;

SimpleCntr8_1_Compare_Reg = 0x02;
```

   This would update the period to 0x08 and the compare value to 0x02. For more information on this and how to use these defines, refer to the *Component Author Guide*. Note that 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.

A.2 Project #2 – 16-Bit PWM

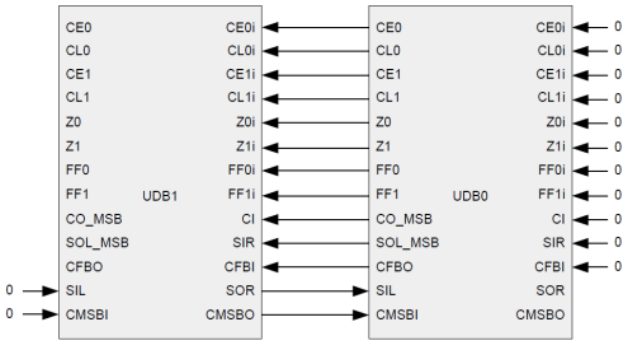
This example project introduces you to the concept of datapath chaining. The datapaths have dedicated signals that are tied to neighboring datapaths. These signals allow you to create functions that are up to 32 bits wide. In this example, you create a PWM that is similar to the first example project, but it is 16 bits wide, as Figure 108 shows.

Figure 108. A 16-Bit Function With Chained UDBs



The ALU in each datapath is designed to chain carries, shifted data, and conditional signals to its nearest neighbor, as Figure 109 shows. All conditional and capture signals chain in the direction of the least significant byte to the most significant. Shift-left also chains from the least to the most significant. Shift-right chains from the most significant to the least significant.

Figure 109. Datapath Chaining Flow



This example assumes that you are familiar with the concepts introduced in the previous example. A completed 16-bit PWM project is included with this application note for your reference.

A.2.1 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.

A.2.2 16-Bit PWM Component Creation Steps

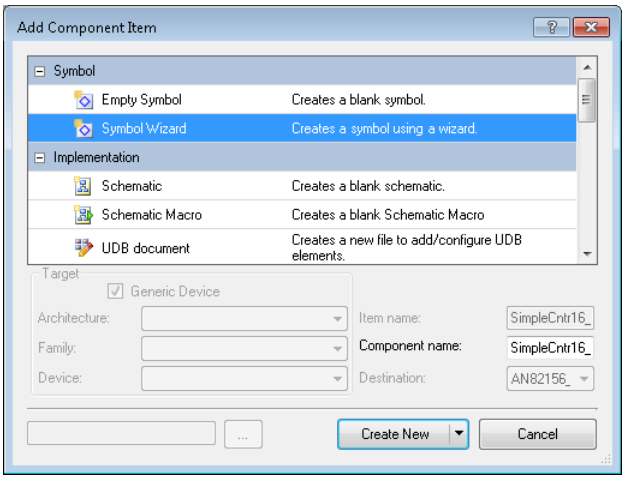
To avoid confusion, create a new Component, instead of modifying the one from the first example project. The basic Component creation steps are the same.

1. Launch PSoC Creator and open the "**AN82156_Appendix**" 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.

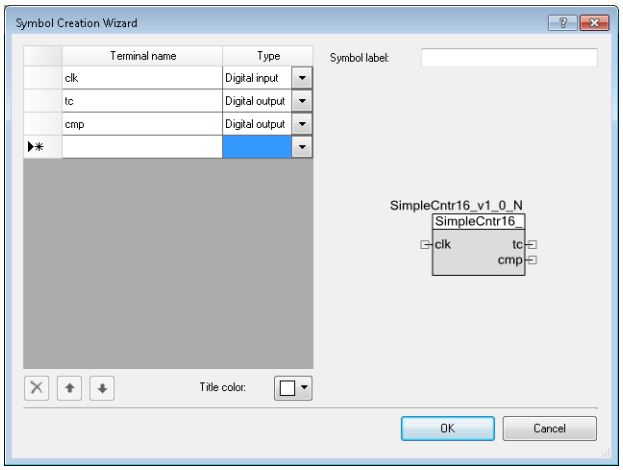
2. Add a new Component item to ***AN82156_Appendix_Lib*** using the Symbol Wizard, as Figure 110 shows. This example uses "*SimpleCntr16_v1_0*" as the Component name.

Figure 110. New Symbol Creation



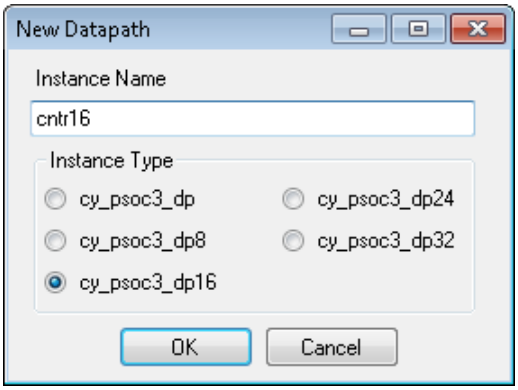
3. Add a *clk* input and *tc* and *cmp* outputs, just as in the 8-bit PWM example, as Figure 111 shows.

Figure 111. Adding Terminals to the PWM



4. Right-click the symbol schematic page and add symbol properties:
   - Doc.APIPrefix = *SimpleCntr16*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Cntr16*
   - Doc.DefaultInstanceName = *SimpleCntr16*
5. Generate the Verilog file for the new Component symbol – leave everything at default values.
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 uses "cntr16" as the configuration name. Select "cy_psoc3_dp16", as Figure 112 shows.

Figure 112. Creating a New PWM Datapath



9. Click OK to create the datapath configuration.

   Notice that there are *cntr16_a(16)* and *cntr16_b(16)* configurations. Two separate datapaths configurations are added to the Verilog file. The 'a' configuration is for the LSB; 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 113 and Figure 114 show.

Figure 113. Configuration for the SimpleCntr16 Component

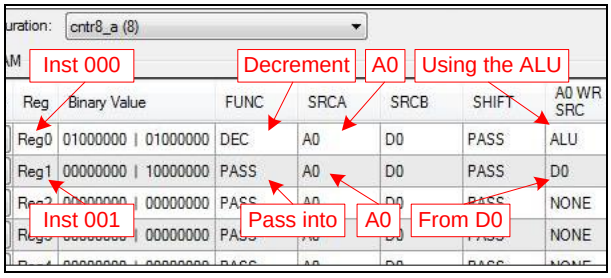
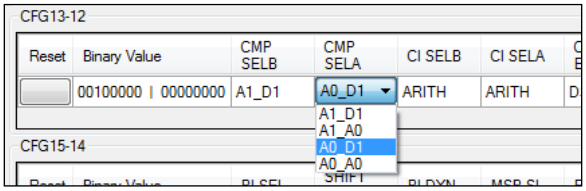


Table 9. Example Two Datapath Configuration

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

Figure 114. Compare Block 1 Mux Settings



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

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

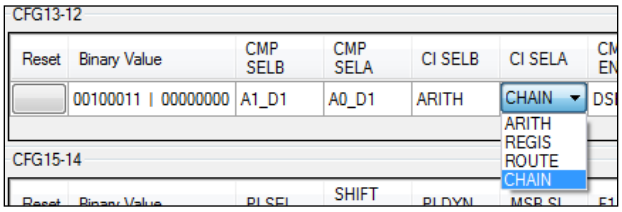
    You need to make the same changes to the "_b" configuration because it is the upper eight bits of our PWM. The Datapath Configuration Tool can copy settings from one configuration to another.

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.

Configure the chaining; only the '_b' configuration uses these settings.

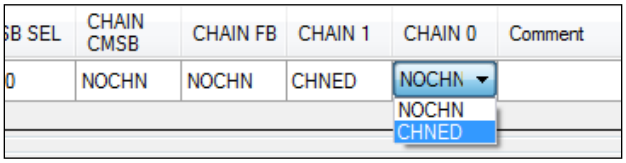
15. In the 'cntr16_b' configuration, set CI_SELA to "CHAIN", as Figure 115 shows. This configures the carry-in signal to come from the previous datapath.

Figure 115. Configuring Carry-In Chaining



16. Set CHAIN1 and CHAIN0 to "CHNED", as Figure 116 shows. These configure the compare conditions (ce0, ce1, cl0, cl1, z0, z1, ff0, ff1) to be chained.

Figure 116. Configuring Datapath Chaining

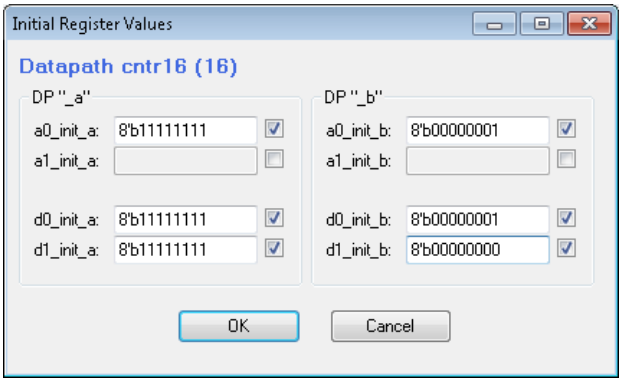


17. Do a Save to apply the changes to the Verilog file.

    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 117 shows. This example shows the value in binary to illustrate that each register is still eight bits wide.

Figure 117. 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 255.

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 configurations in the Verilog code. These two datapaths are chained together to form the 16-bit PWM. Next, you need to add some additional signals for use in the Verilog logic.
22. Add the following code after `'#start body'` on or near line 25:

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

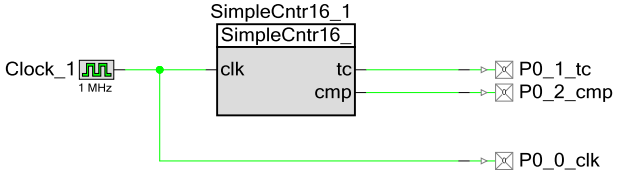
    The hardware links are similar to those of the 8-bit PWM. The difference is that the outputs are two bits wide, instead of one. Because you chained *Chain0 and Chain1* in the Datapath Configuration Tool, the upper bit of the outputs contains the final result. For example, the upper bit of the z0 output 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 generates a warning.
23. Set `.clk()` to `.clk(clk)`.
24. Set `.z0()` to `.z0({tc, tc_lsb})`.
25. Set `.cl1()` 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 ready for use in your project.
28. Set the *AN82156_Appendix_Lib* as a dependency for the *16bitCounter* project.

    The new Component is present in the Component Catalog. It is ready for use in your project.
29. Drag a Cntr16 Component onto the schematic.
30. Connect a clock Component to the 'clk' terminal and set it to 1 MHz.
31. Connect Digital Output Pin Components to the 'clk', 'tc', and 'cmp' terminals, as Figure 118 shows.

    Note: For PSoC 4 and PSoC 6 MCU, this process is different; see Step 36 in Section A.1.2.

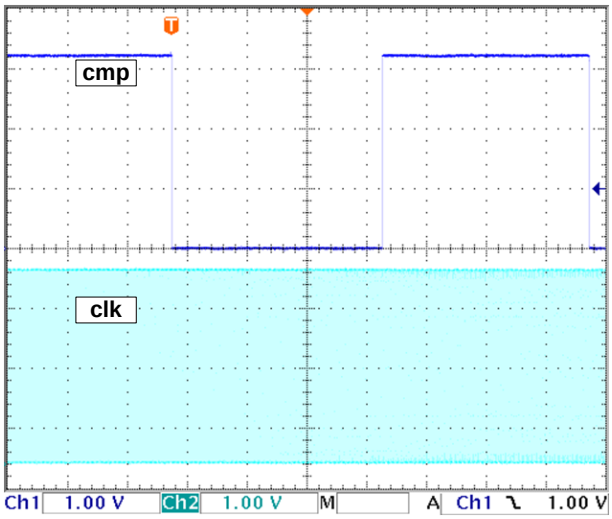
Figure 118. Simple 16-Bit PWM Project Schematic



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, build the project, 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 119 shows. You can set them to any 16-bit value.

Figure 119. Simple 16-Bit PWM Output



You can use chaining to make functions up to 32 bits wide. Apply the principles described in this example to larger functions.

A.2.3 16-Bit Component Header Files in PSoC 3

As noted, writing to 16-bit registers is different from writing to 8-bit registers. You can write directly 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.

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 the same as before. The only difference is that you use `((reg8 *)` instead of `(*(reg8 *)` in the define. This gives us a pointer to the datapath register. It is a good practice to add _PTR to the end of the define name so you can differentiate.

When you want to update a value, use CY_SET_REG16 and the pointer you defined in your header file. Again, look at the attached example project.

A.3 Project #3 – Up/Down Counter

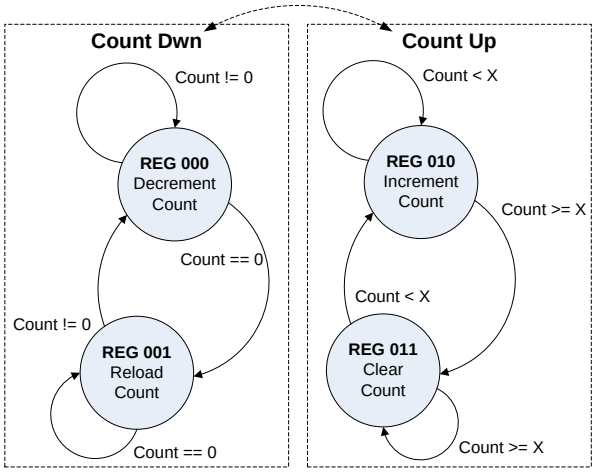
This example introduces you to the steps to add advanced features to a datapath Component. The same basic PWM concept is updated to add the ability to count up or down. The direction is based on a parameter that the user can set at design time or during run time.

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

A.3.1 Additional Details

The simple down-counting PWM used two states. To implement an up/down counter, four states are needed, as Figure 120 shows.

Figure 120. Up/Down Counter State Diagram



The datapath decrements or increments A0 depending on the parameter you set.

A.3.2 Example Project Steps

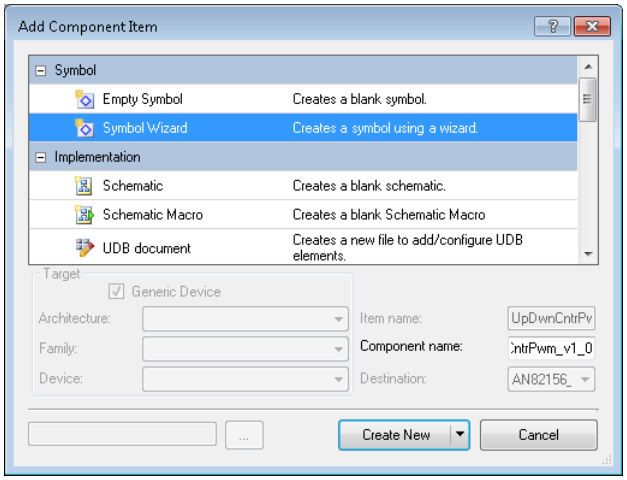
To avoid confusion, create a new Component, instead of modifying the ones from the previous example projects. The basic Component creation steps are the same.

1. Launch PSoC Creator and open the "**AN82156_Appendix**" workspace that you used for the simple 8-bit example. Add a new project, called "*UpDwnCntrPwm*", to the workspace.

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

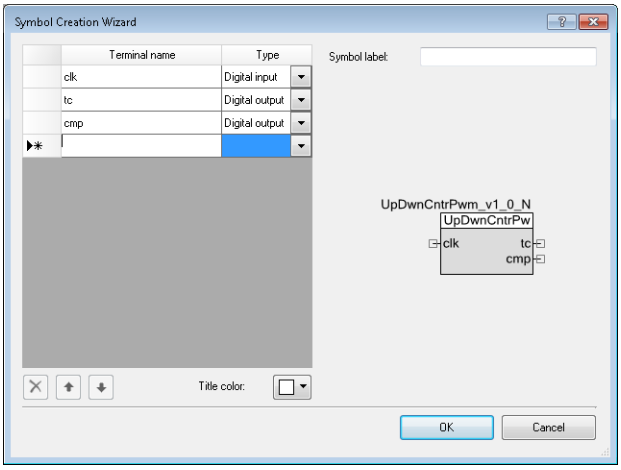
2. Add a new Component item to ***AN82156_Appendix_Lib*** using the Symbol Wizard, as Figure 121 shows. This example uses "UpDwnCntrPwm_*v1_0*" as the Component name.

   Figure 121. 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 122 shows.

Figure 122. Adding Terminals for the Up/Down Counter

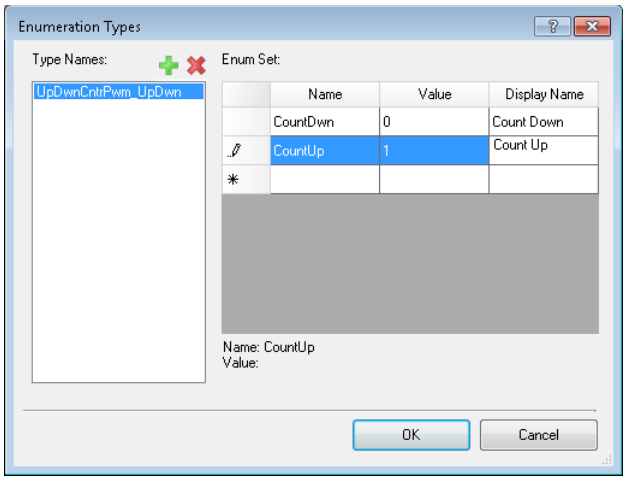


4. Right-click the symbol schematic page and add symbol properties:
   - Doc.APIPrefix = *UpDwnCntrPwm*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/UpDwnCntrPwm*
   - Doc.DefaultInstanceName = *UpDwnCntrPwm*

   You need to add some parameters that the user can modify – counter period, compare value, and count mode (up or down).

   For the count mode setting, define a new type of parameter.
5. Right-click the Symbol Editor page and open the Symbol Parameters dialog.
6. Click the 'Types' button to open the window to create the new parameter type.
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 123 shows.

Figure 123. Creating New Component Parameter Types



9. Click **OK** to return to the Symbol Parameters dialog.

   You can assign parameters to this new type and set 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; see Table 10 and Figure 124:

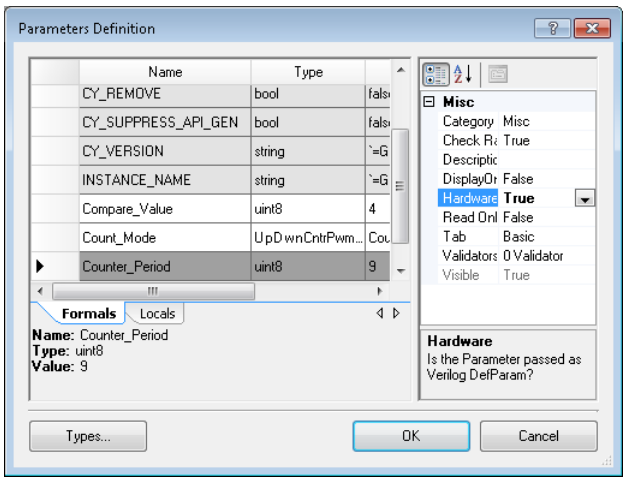
Table 10. Up/Down Counter Parameters

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.

11. Set the **Hardware** flag to 'True' for all three new parameters, as Figure 124 shows.

Figure 124. Adding New Component Parameters



12. Click **OK** and save the changes to the Component.
13. Right-click an empty spot in the Symbol Editor and generate the Verilog file for the new Component symbol – leave all settings at the default values.
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 generated.

    The first two configurations are the same as in the simple 8-bit PWM, but you also need two additional configurations for the up counting.

16. Create a new datapath configuration, called "UpDwn". Use the 'cy_psoc3_dp8' type.
17. Configure the CFGRAM section to match Table 11.

Table 11. Example Three Datapath Configuration

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

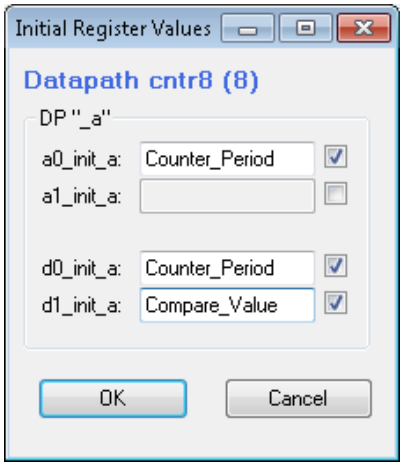
18. The XOR configuration is used to clear the count register. After the count register counts up to the period value, it XORs itself, an act that clears the count back to zero.
19. Set the CMP SELA field to 'A0_D1' to configure D1 as the compare value, just as in the simple 8-bit PWM.

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

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

20. Open the Initial Register Values window and add parameter names into A0, D0, and D1, as Figure 125 shows.

Figure 125. Using Parameters for Initial Register Values



21. Click **OK** and save changes to apply them to the Verilog file.
22. Save the changes, and close the Datapath Configuration Tool.

Next, you need to add some Verilog code to the Component to implement these features.

23. Open the *UpDwnCntrPwm_v1_0.v* file and locate the text that says:

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

24. Replace that text with:

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



// Up/down control
assign upDwn=control[CNTL_CNT_UP_DWN];
// Logic for the 'reload' signal
assign reload = ( upDwn ) ? ( up_reload ) : ( zero_detect );
//assign termnical count output
assign tc = reload;
    // Control register instance. This text is
```

```
// found in the Component Author Guide.
cy_psoc3_control #(.cy_init_value (Count_Mode), .cy_force_order(`TRUE))
pwmCntlReg(
/* output [07:00] */ .control(control)
);
```

25. 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 `.ce0 ()` to `.ce0(up_reload)`.
    - Change `.z0()` to `.zo(tc)`.
    - Change `.cl1()` to `.cl1(cmp)`.

    Now that you have made these changes, let us 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 control registers and the different options, consult the *Component Author Guide*.

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

    Next, notice that the .cs_addr has two bits of control instead of one. In the previous examples, the datapath had only two states, so one bit of control was sufficient. Since there are four states, you need two bits of control.

    In addition, notice that instead of using only the 'tc' signal to control the datapath configuration, 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 ce0 comparison. Therefore, when the counter is configured as an Up counter, it continues to count up as long as the value in A0 is not equal to the value stored in D0; when the counter is equal to D0, it triggers the reload of the register.

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

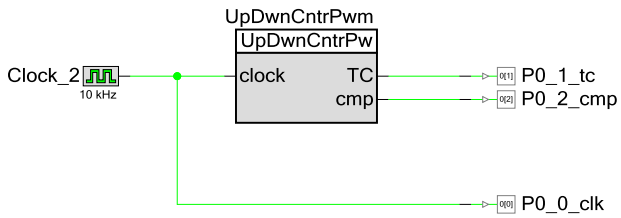
26. Save all changes to the Verilog file.

    After you make these changes to the Verilog file, the Component is ready to be used in your project. Add all the same Components as in the first example project.

27. Add AN82156_Appendix_Lib as a dependency of the UpDwnCntrPwm project.
28. Drag an UpDwnCntrPwm Component to the project schematic.
29. Connect a Clock Component to the 'clk' terminal of the Component and set it to 10 kHz.
34. Connect Digital Output Pin Components to the Component's terminals – P0_0_clk, P0_1_tc, and P0_2_cmp – as Figure 126 shows.

    Note: For PSoC 4 and PSoC 6 MCU, this process is different; see Step 36 in Section A.1.2.

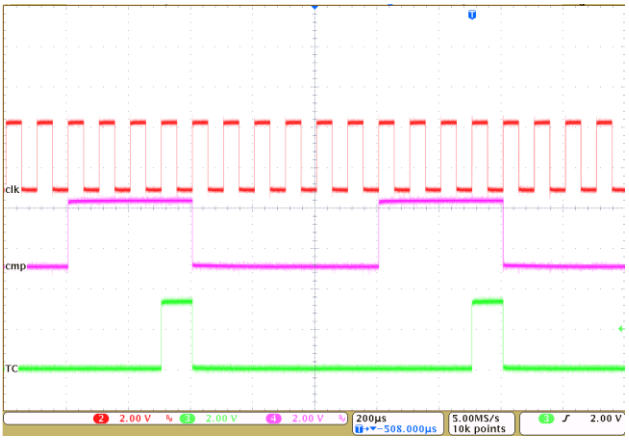
Figure 126. Up/Down Counter PWM Project Schematic



30. Do a Save All, build the project, 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.

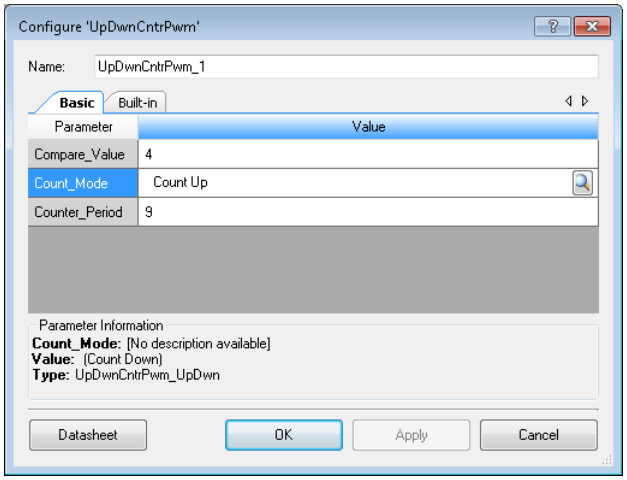
Figure 127. Down-Counting PWM Waveforms



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.

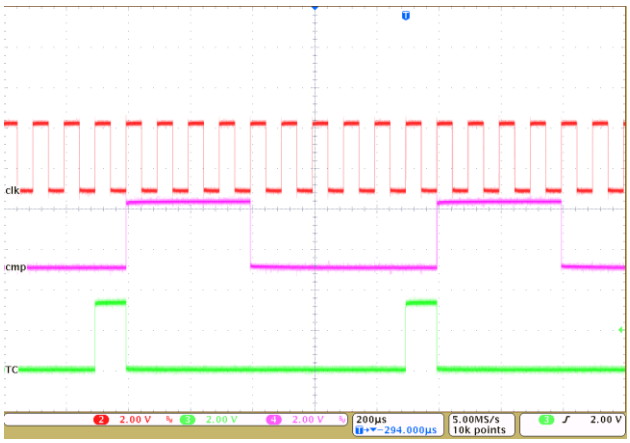
31. Go back to the project schematic and double-click the UpDwnCntrPwm Component to open the properties window.
32. Change the Count_Mode to 'Count Up', as Figure 128 shows.

Figure 128. Setting the PWM to Count Up



33. Click OK to apply the changes.
34. Do a Save All, build the project, and program the PSoC.
35. You can observe that this is an up counter, because the cmp value is HIGH after a TC as the counter is reloaded with zero at that point, as Figure 129 shows.

Figure 129. Up-Counting PWM Waveforms



36. Notice that the output was HIGH after the TC 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.

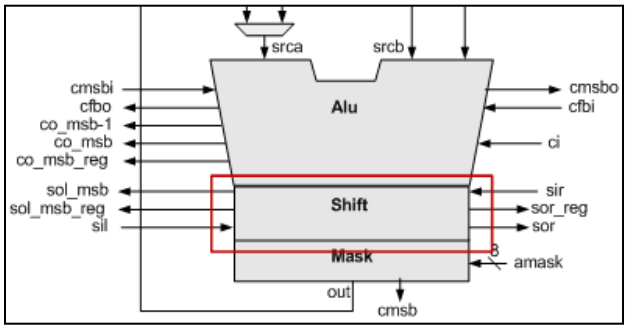
A.4 Project #4 – Simple UART

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 ***AN82156_Appendix_Lib*** project of the completed examples. An example of how this Component is included in the same workspace, in the project "***SimpleTx***".

TX UART Component Details

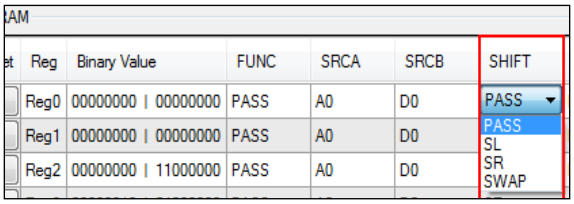
The datapath usage in this Component is simple. The only datapath operations used are shifting and loading a value into A0 from F0. There is a shifter at the output of the ALU, as Figure 130 shows.

Figure 130. Shifter Block



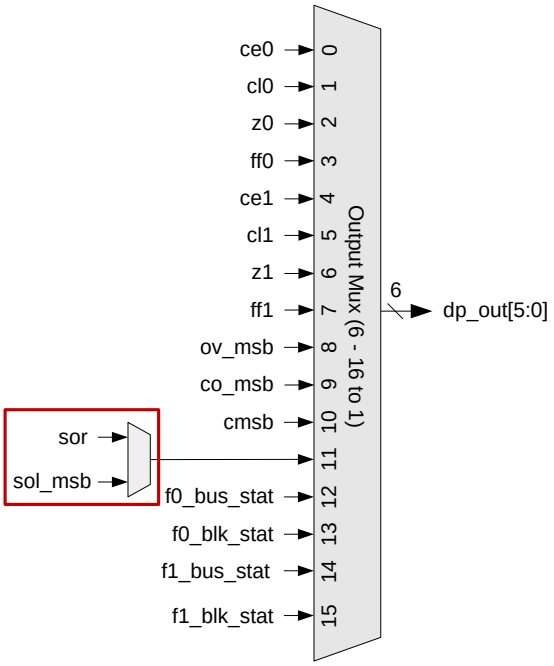
This shifter can shift bits either left or right, and it has the ability to swap nibbles. Each configuration of the datapath can independently set the operation of the Shifter. This option is set in the dynamic configuration area of the Datapath Configuration Tool, as Figure 131 shows.

Figure 131. Shift Operation Settings



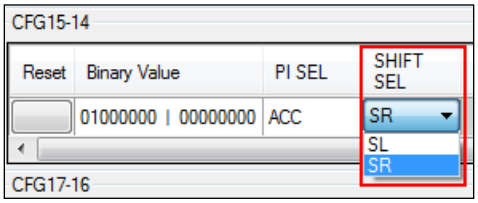
There is only one Shift Out (SO) output on the datapath output mux. This output is shared between the Shift Out Right and the Shift Out Left, as Figure 132 shows.

Figure 132. Datapath Output Mux Diagram



You must configure this mux properly. This is done in the Static Configuration area under CFG15-14 *SHIFT SEL*, as Figure 133 shows.

Figure 133. SHIFT SEL Configuration



In this example, you create a Verilog state machine that controls the configuration 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, notice a line of code that says:

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

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

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

The datapath configuration/instruction is changed by the Verilog state machine implemented with a 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
localparam STATE_IDLE  = 2'b00;
// Stop bit 2
localparam STATE_STOP  = 2'b01;
// Start bit
localparam STATE_START = 2'b10;
// Data
localparam STATE_DATA  = 2'b11;
```

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

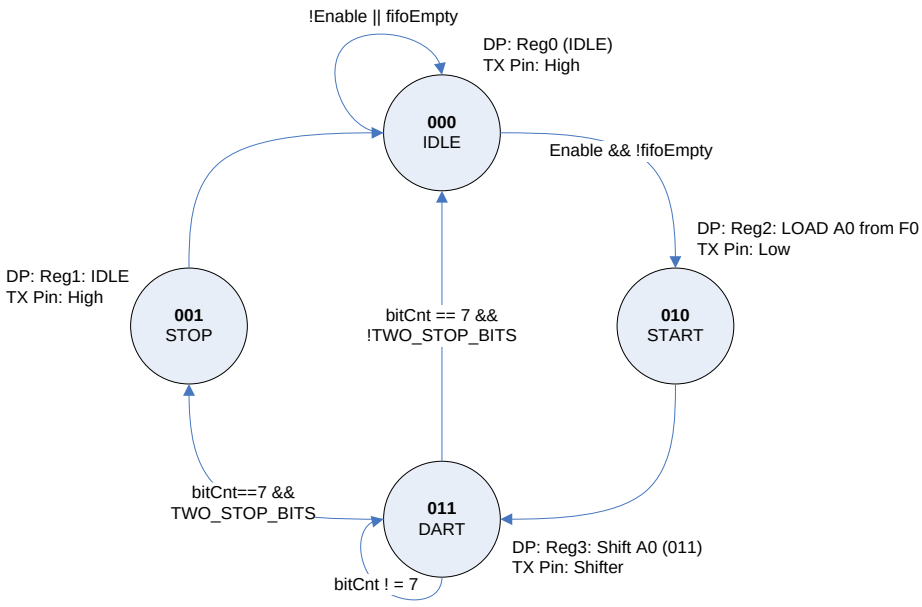
Table 12. Simple Tx Datapath Configuration

REG	FUNC	SRCA	SHIFT	A0 WR SRC
000	Pass	A0	None	None
001	Pass	A0	None	None
010	Pass	A0	None	F0
011	Pass	A0	SR	ALU

As the code moves through the Verilog state machine, it changes the datapath configuration. This is a common use case. Most complex Datapath Components need a Verilog state machine to sequence the datapath configurations.

Figure 134 shows how the simple TX Verilog state machine works.

Figure 134. 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.

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

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.

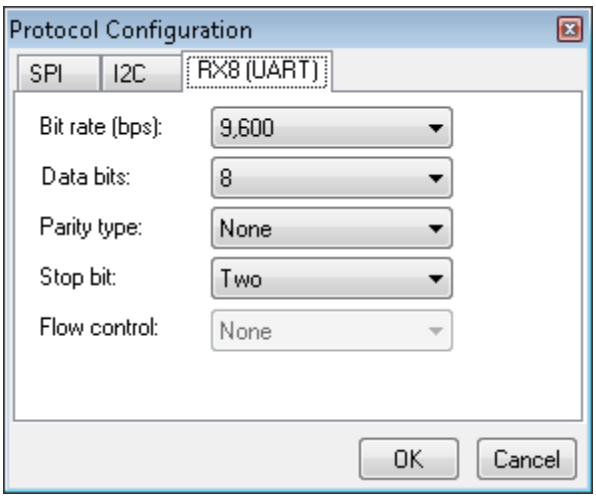
```
// This next statement determines the tx output.  if in data state, output the shift
register output "srOut", else output a low during the start state, and a high during
stop state.
     assign tx = (state[1:0] == STATE_DATA ) ? srOut : ( !state[1] );
```

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

The main code for this project enables the Component with two stop bits, and then continuously transmits the values 0 through 10 at 9600 baud. Configure your receiver for 9600 baud and two stop bits.

Installed with PSoC Creator is a program called the Bridge Control Panel (BCP). You can use the BCP to receive RX characters. In BCP connect to the COM port that you have connected the TX output too. In **Tools > Protocol Configuration** configure the RX8 (UART) as Figure 135 shows.

Figure 135. BCP RX8 Configuration

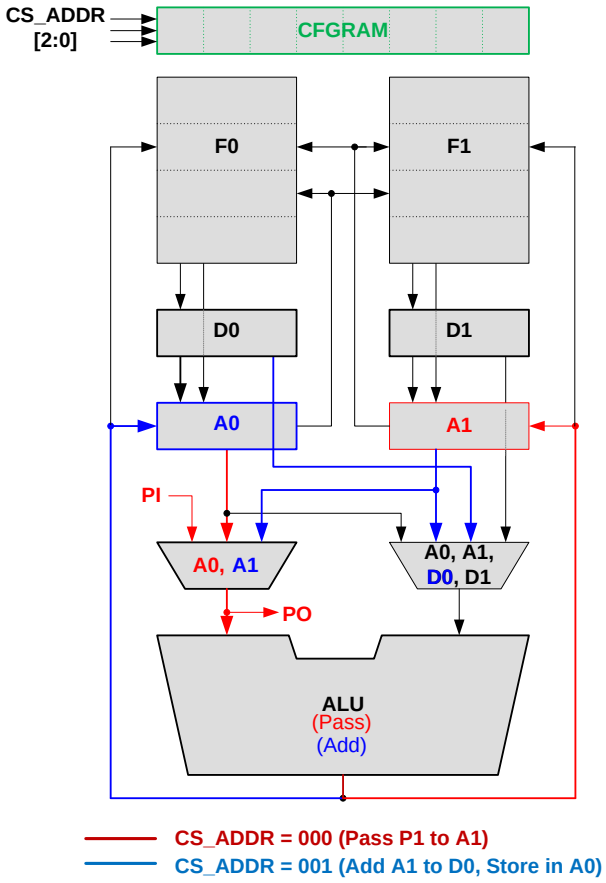


In the editor of the BCP, add the following text: rx8 x x x x x x x x x x. This reads 11 bytes from the selected COM port. You can then hit **Repeat** to continuously receive data.

A.5 Project #5 – Parallel In and Parallel Out

This example project demonstrates how to use the parallel input and parallel output of the datapath. An 8-bit parallel adder is used to demonstrate these features. The adder reads in an 8 bit parallel value from the parallel input (PI) and stores that value in A1. Next, it takes the value stored in A1 and adds it to a fixed value stored in D0. The value is then output on the parallel output (PO).

Figure 136. Parallel Adder Implementation



This example assumes that you are familiar with the concepts introduced in the previous example projects. A completed PI_PO_Example project is included with this application note.

A.5.1 Example Project Steps

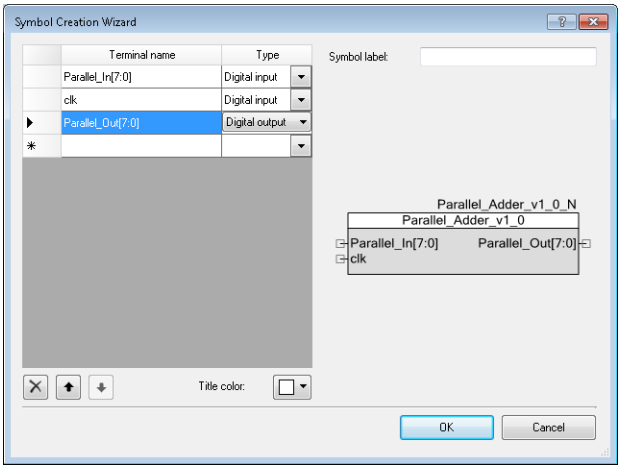
To avoid confusion, create a new Component, instead of modifying the ones from the previous example projects. The basic Component creation steps are the same.

1. Launch PSoC Creator and open the "**AN82156_Appendix**" workspace that you used for the simple 8-bit example. Add a new project, called "*PI_PO_Example*", 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.

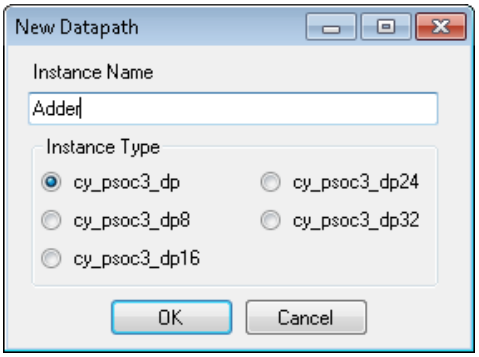
2. Add a new Component item to AN82156_Appendix_Lib using the Symbol Wizard. This example uses "Parallel_Adder_v1_0" as the Component name.
3. Add a clk input and a Parallel_In[7:0] input (this defines an 8 bit input). Also define a Parallel_Out[7:0] output, as Figure 137 shows.

Figure 137. Adding Terminals for the Parallel Adder



4. Right-click the symbol schematic page and add symbol properties:
   - Doc.APIPrefix = *Parallel_Adder*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Parallel_Adder*
   - Doc.DefaultInstanceName = *Parallel_Adder*
5. Add a new symbol parameter called Add_Value, and set the type to uint8 and set Hardware to True.
6. Right-click an empty spot in the Symbol Editor and generate the Verilog file for the new Component symbol – leave all settings at the default values.
7. Do a Save All to apply the changes.
8. Launch Datapath Configuration Tool and open the Parallel_Adder_v1_0.v file that you just generated.
9. Create a new datapath configuration, called "adder". Use the 'cy_psoc3_dp' type; see Figure 138.

Figure 138. Datapath Selection



10. This selection gives you access to the PI and PO signals of the datapath. The other four Instance Types do not allow access to the PI and PO.
11. The cy_psoc3_dp is only an 8-bit instance. If you want more than eight bits you need to place more than one of these instances, and then manually chain them in Verilog. Appendix D shows an example of them chained together to form a 24-bit datapath.

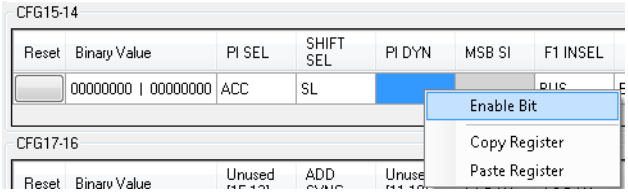
12. Configure the CFGRAM section to match Table 13.

Table 13. Example Five Datapath Configuration

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

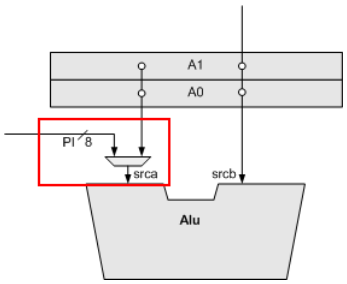
13. The first configuration (REG 000) takes the value on PI and stores it in A1. The second configuration (REG 001) take the value stored in A1 and adds it with D0 and stores that value in A0.
14. Next, configure the PI controls.
15. In the CFG15-14 fields, right-click the field below PI DYN and select Enable Bit, as Figure 139 shows.

Figure 139. Enabling Dynamic PI Control



16. Set PI DYN to EN.
17. This allows for the selection of the SRCA input to be dynamically selectable between the PI, and A0 or A1; see Figure 140.

Figure 140. scra Mux



18. The selection of this mux is controlled in the CFGRAM, under the bit CFB EN. When this is set to DSBLthe srca input comes from either A0 or A1. When it is set to ENBL the srca input comes from the PI.
19. If you look back at Table 13, you will see that for the first configuration, CFB EN is set to ENBL. This means that srca comes from the PI. Thus, for the first configuration the datapath is passing the value from PI into A1.
20. For the second configuration, CFB EN is set to DSBL. This means that the scra input is controlled by the CFGRAM, and in this case, is set to A1.
21. Set the initial value of D0 to Add_Value.
22. Save the changes you have made, and close Datapath Configuration Tool.

    Next, you need to add some Verilog code to the Component to implement these features.
23. Open the *Parallel_Adder_v1_0.v* file and locate the text that says:

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

24. Replace that text with:

```
/* Register to hold state of statemachine*/
reg state;
wire[7:0] po;
```

```
/*State loads the value from the PI into A1, latches value in A0 out to PO*/
localparam STATE_LOAD  = 2'b00;

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

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

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

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

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

25. Make the following links in the datapath logic:
    - Change `.clk()` to `.clk(clk)`.
    - Change `.cs_addr(3'b0)` to `.cs_addr({2'b0, state})`.
    - Change `.pi()` to `.pi(Parallel_In)`.
    - Change `.p0()` to `. p0(po)`

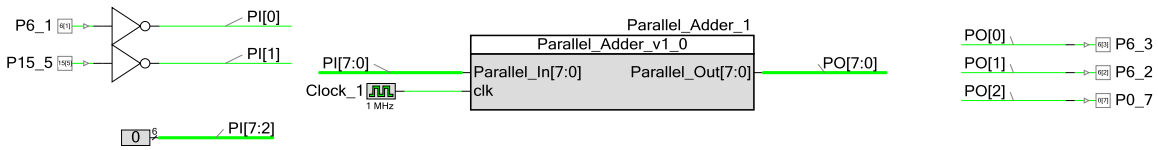
    We've just created a two-state state machine that toggles back and forth between the two datapath configurations. The first one takes the value on PI and stores it in A1. The second one takes the value stored in A1 and adds it to D0 and stores that value in A0.

    PO is always connected to the selection of either A1 or A0. Since our result is only valid when A0 is selected for srca, we must register PO at the appropriate time. This is the need for the line:

    ```
    Parallel_Out <= po;
    ```

    This takes the value directly out of the PO and stores it in the signal Parallel_Out (the component output). In the next state, scra is selected as A1 so the PO is not valid. Thus, the reason it is registered in the load state.
26. Once you have made all the changes, do a Save All.
27. Add AN82156_Appendix_Lib as a dependency of the PI_PO_Example project,
28. Drag a Parallel_Adder onto your schematic.
29. Configure the schematic to look like Figure 141.

Figure 141. Parallel Adder Schematic



P6_1 and P15_5 are Digital Input Pin Components, configured as **Resistive Pull Up**.

P6_3, P6_2, and P0_7 are Digital Output Pin Components.

This schematic is specifically designed for the CY8CKIT-030 and CY8CKIT-050. If you are using a different hardware platform, you will need to change the pinout as:

Table 14. Pin Mapping for PSoC 4 kits

Kit	Reccomended Input Pins	Reccomended Output Pins
CY8CKIT-042	P0[7], P1[0]	P0[5], P0[6], P0[0]
CY8CKIT-043	P0[7], P1[0]	P1[1], P1[2], P1[3]
CY8CKIT-044	P0[7], P1[0]	P1[1], P1[2], P1[3]
CY8CKIT-049-42xx	P0[7], P1[0]	P0[5], P0[6], P0[0]

To name a wire (see Figure 142):

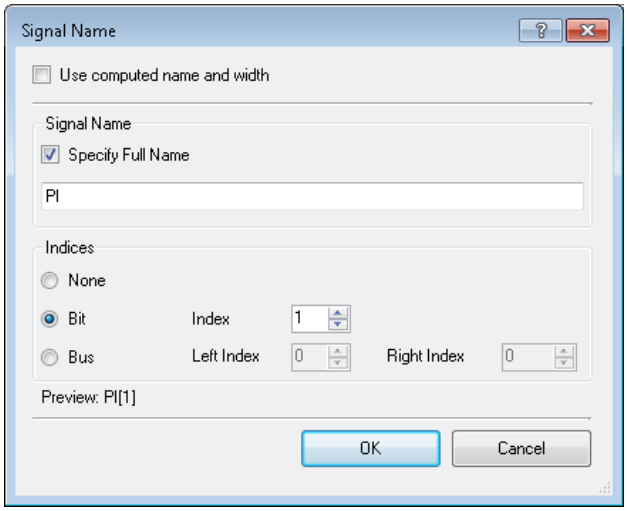
a. Double-click the Wire.

b. Uncheck the Use Computed name and width checkbox.

c. Check the **Specify Full Name** checkbox.

d. Type in the name of the net (wire) and select if it has any **Indices**.

Figure 142. Wire Naming Dialog



30. Set Add_Value to 2 inside the component customizer.
31. Do a Save All, build, and program the project.
32. On the CY8CKIT-050 or CY8CKIT-030 place a wire between P0_7 and LED2.

    The buttons and LED show how the simple adder works. LED 4 represents Bit 0 of the output, LED 3 represents Bit 1, and LED 2 represents Bit 2.

    SW2 represents Bit 0 of the input, and SW3 represents Bit 1 of the input.

    If you are using one of the kits listed in Table 14, you should connect an external button to P1[0], and connect external LEDs to the output pins listed in the table.

    With no switches pressed, LED connected to the output Bit 1 should be illuminated. This is because D0 holds a value of 2. If you press the button connected to input Bit 0, then the LEDs connected to output Bit 0 and output Bit 1 should be illuminated indicating a value of three.

    This is a very simple example demonstrating how to use the PI and PO of the datapath. Now you know how to use these signals to create more complex designs using PI and PO.

B Appendix B – Datapath Configuration Tool Cheat Sheet

This section shows the relationship between the Datapath Configuration Tool and the underlying datapath hardware.

The GUI can be divided into two general sections – the Dynamic Configuration and Static Configuration sections, as Figure 143 shows.

- Dynamic Configuration – Allows you to set up the datapath to behave differently across states
- Static Configuration – Stays the same across states

Figure 143. Datapath Configuration Tool Interface Sections

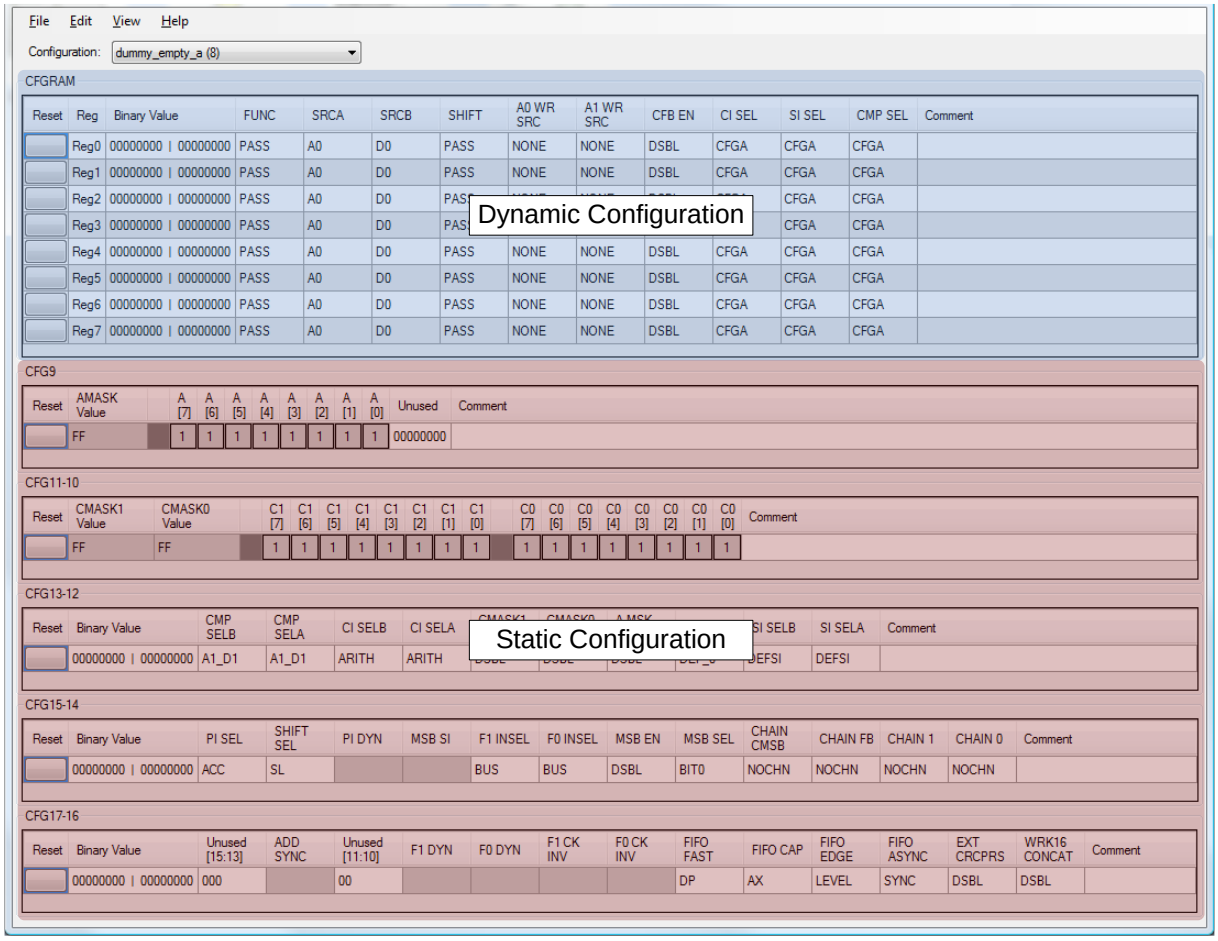
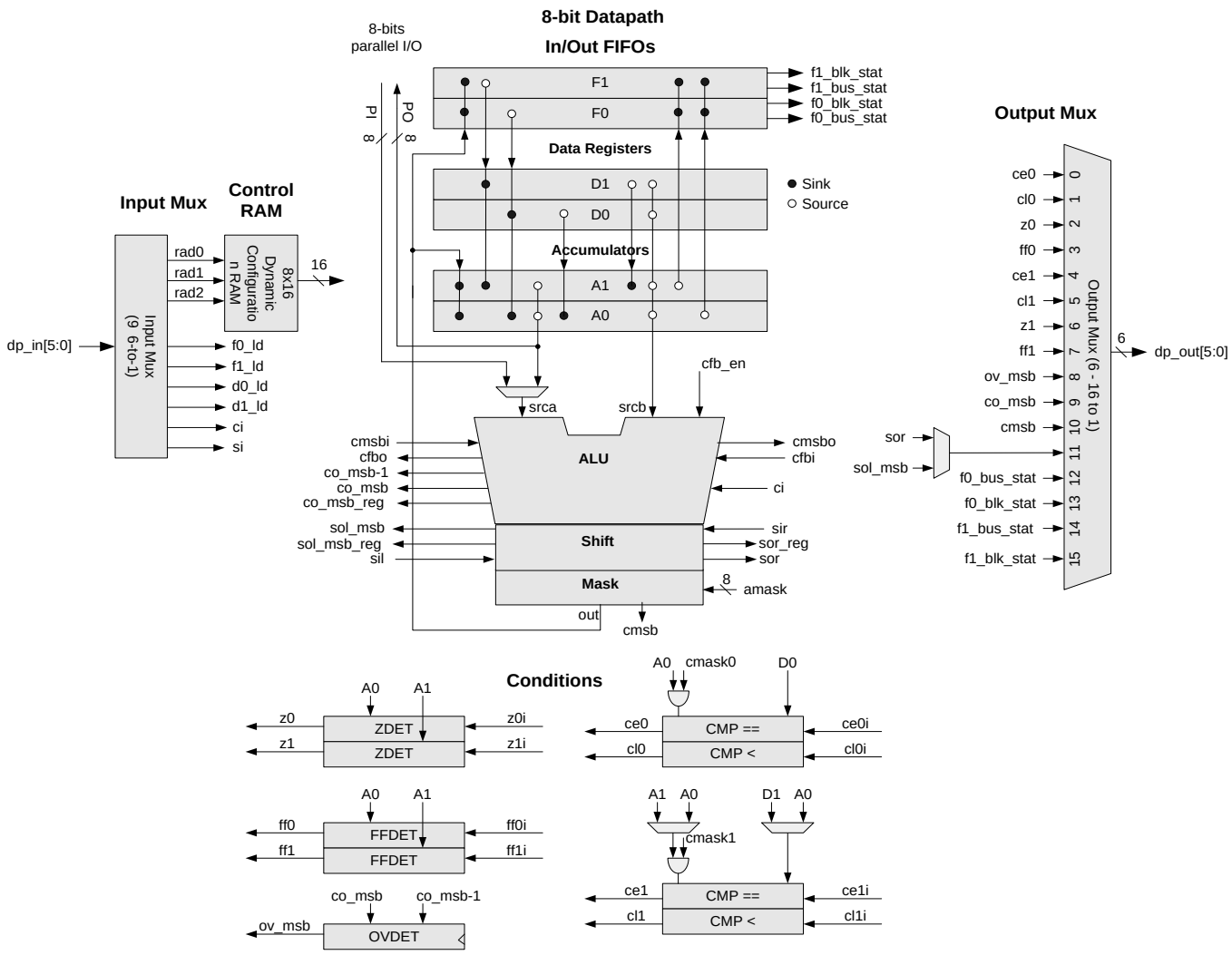


Figure 144. Datapath Block Diagram



B.1 Dynamic Configuration RAM (CFGRAM) Section

The Dynamic Configuration section is a representation of the Dynamic Configuration RAM. It configures the behavior of the datapath for the eight configurations/instructions. The following tables explain the function of each of the fields in the GUI.

Table 15. 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

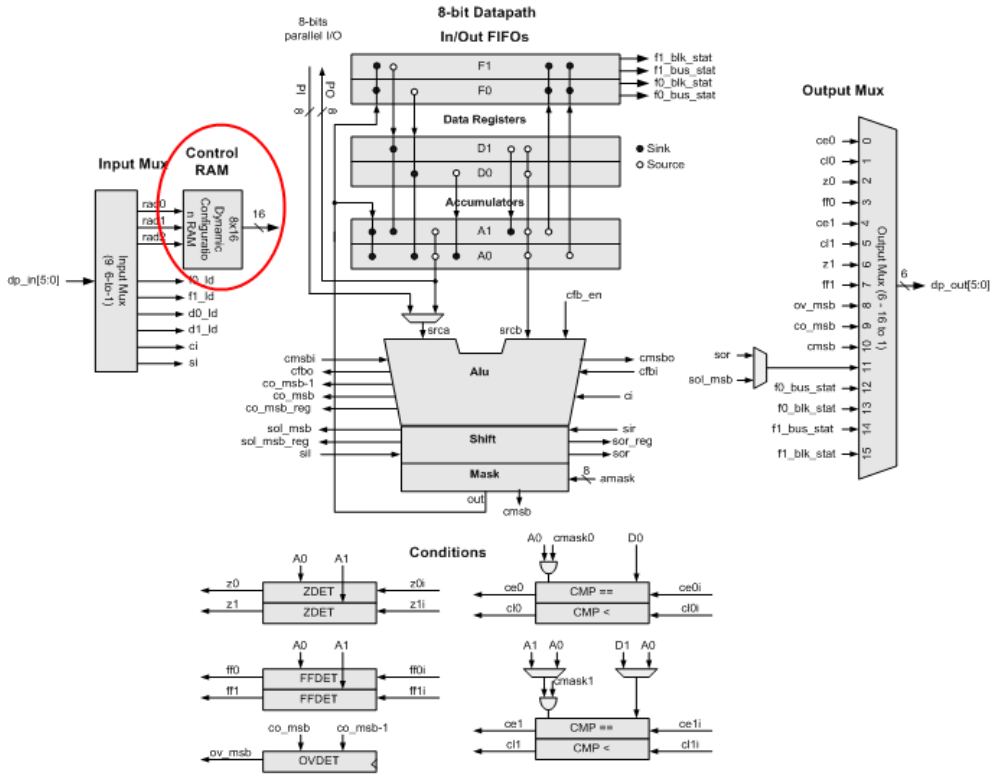


Table 16. Dynamic Configuration Section Column Descriptions

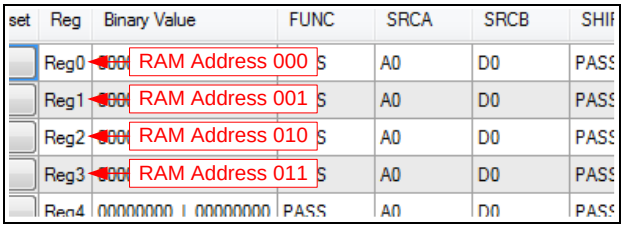
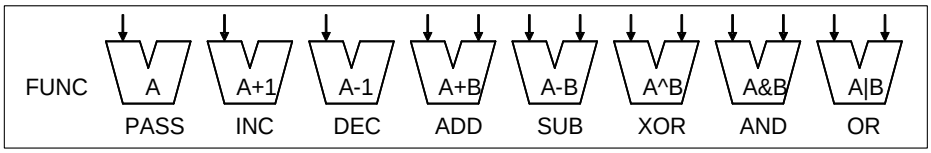
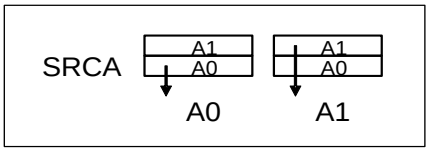
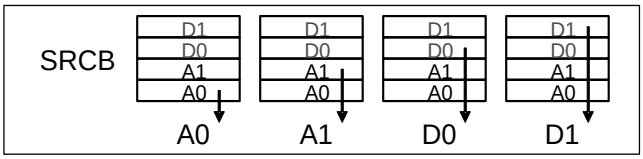
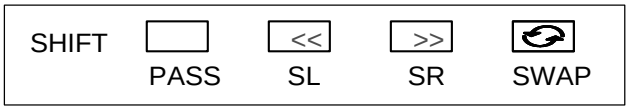
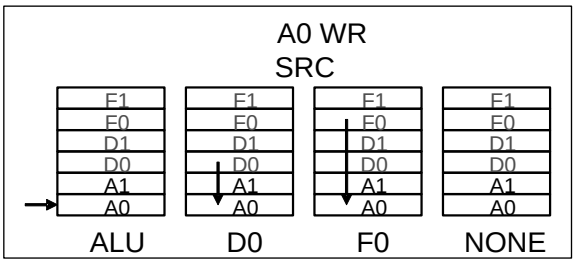
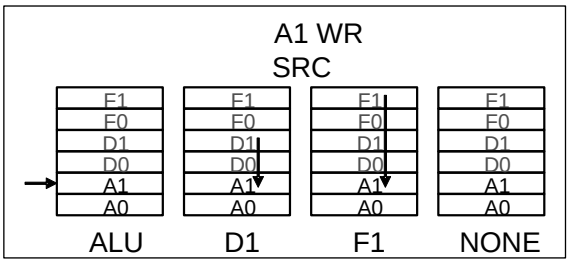
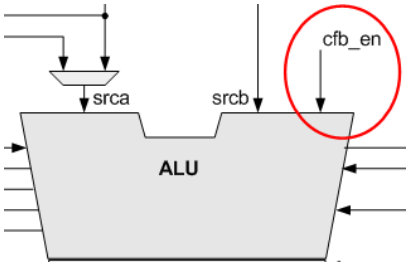
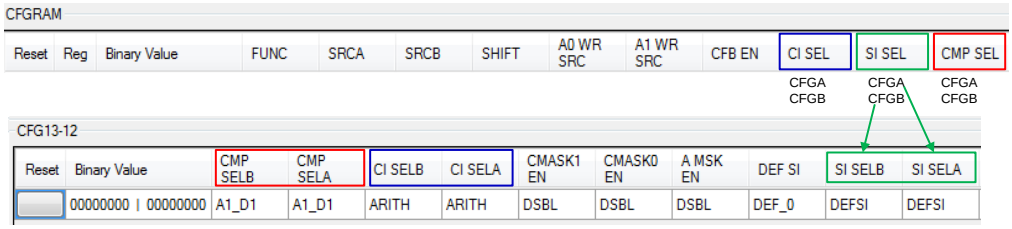
Dynamic Configuration Section	
Reg This column shows the 'configuration/instruction' to which the row corresponds. The value of the three CFGRAM address signals determines which configuration is selected, so ensure that the correct row is selected for each configuration.	
FUNC This column determines which of the eight ALU functions will be performed in that configuration.	
SRCA This column determines the source for the ALU's 'srca' input. srca can also come from PI – see Table 20 (CFG 15-14).	
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 2. Dynamic Configuration Section Column Descriptions (contd.)

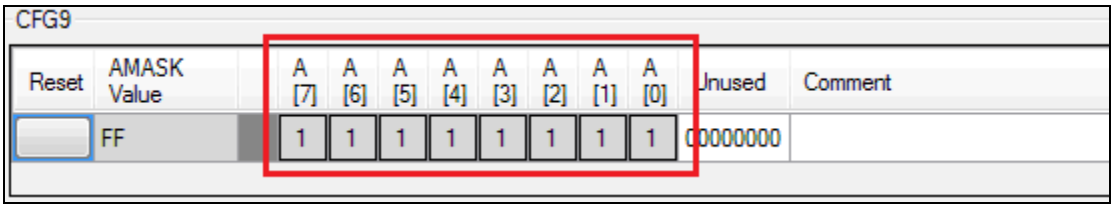
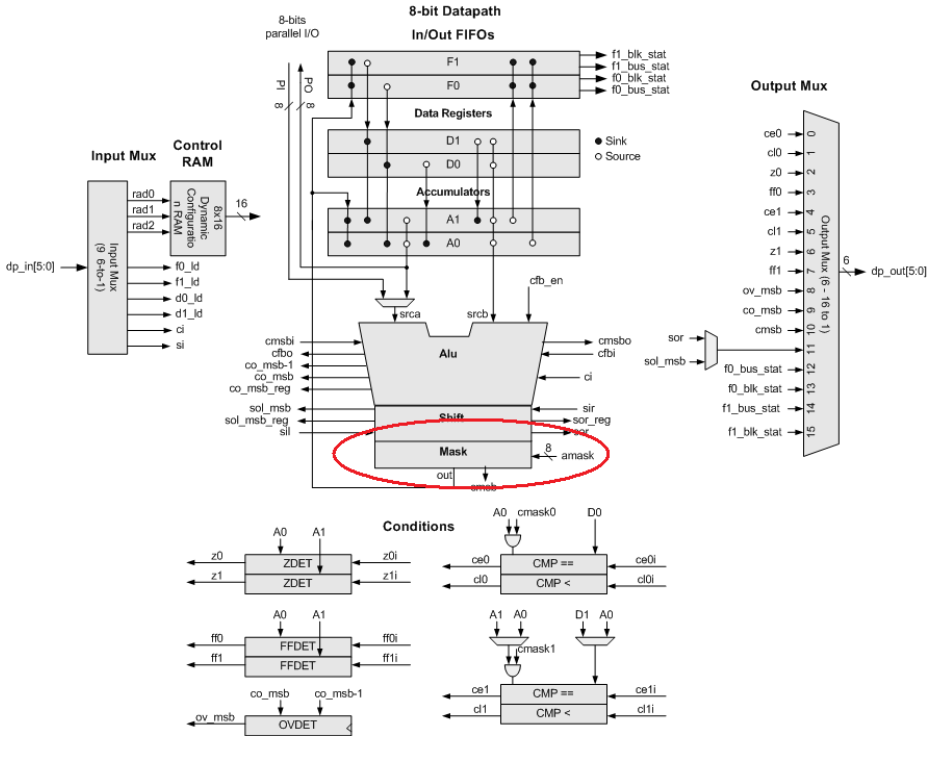
Dynamic Configuration Section	
A1 WR SRC This column determines the contents of the A1 register *after* the ALU operation is complete.	
CFB EN CRC config enable – see PI DYN in CFG15 and CFG14 Registers.	
CI SEL, SI SEL, CMP SEL Carry In Select, Shift In Select, and Compare Select. Each of these has two configuration options: A and B. You can choose the one you want for each configuration. See Table 19 (CFG13-12) for more information.	

B.2 Static Configuration Section

The Static Configuration section represents the datapath registers CFG9 to CFG17, as Table 17, Table 18, Table 19, Table 20, and Table 21 show. They control static functions, including shift direction, masking, FIFO configuration, and chaining.

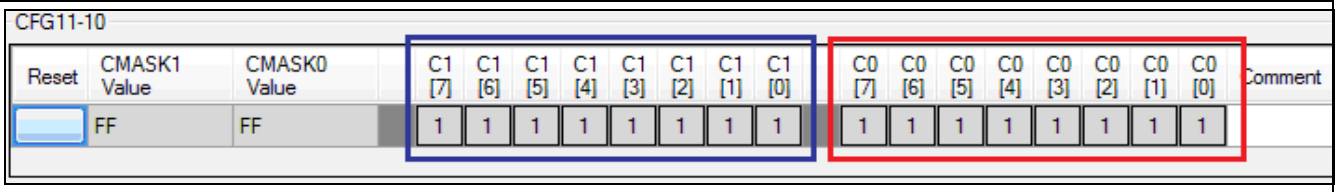
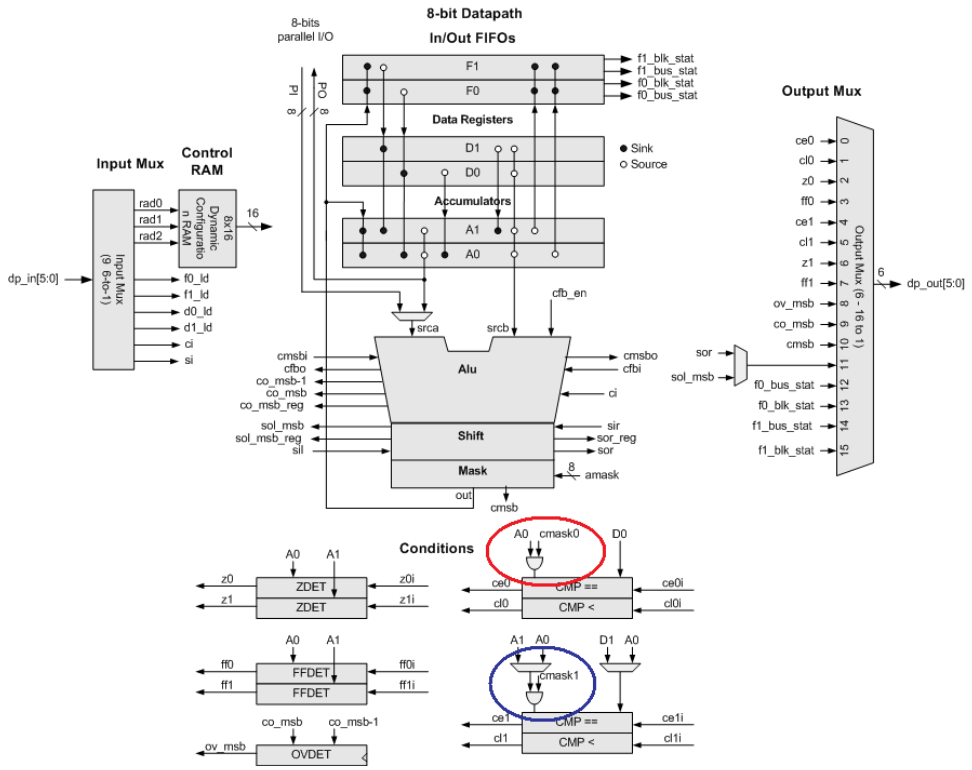
B.2.1 CFG9 Register

Table 17. CFG9 Register Definition

AMASK Value
Datapath Configuration Tool

■ AMASK Value – This field contains the 8-bit mask value that is applied to the output of the Datapath ALU block. The output of the shift register is ANDed with the contents of this register. This feature is off by default. To enable it the AMASK EN bit CFG12 must be set.
Datapath Block Diagram


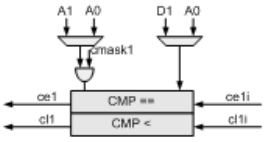
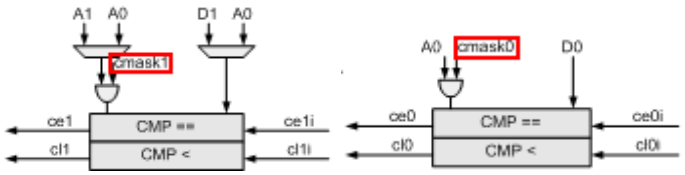
B.2.2 CFG11 and CFG10 Registers

Table 18. CFG11 and CFG10 Register Definitions

CMASK0 and CMASK1 Values
Datapath Configuration Tool

■ 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. To enable them, the CMASK0 EN and CMASK1 EN bits must be set in CFG12 (Table 19).
Datapath Block Diagram


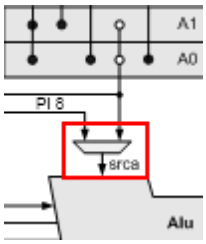
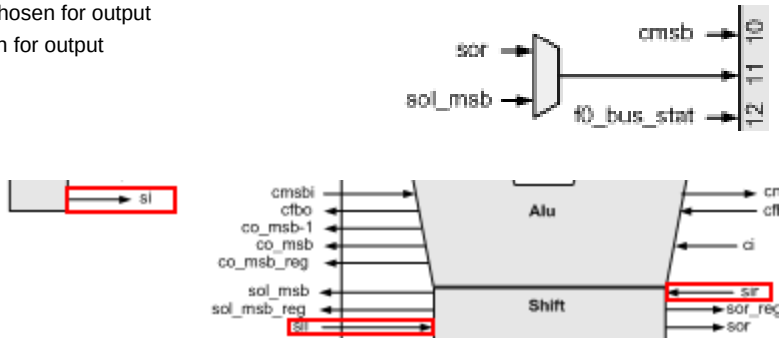
B.2.3 CFG13 and CFG12 Registers

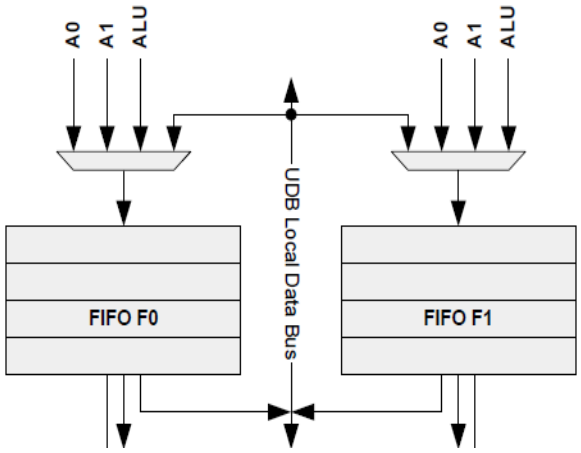
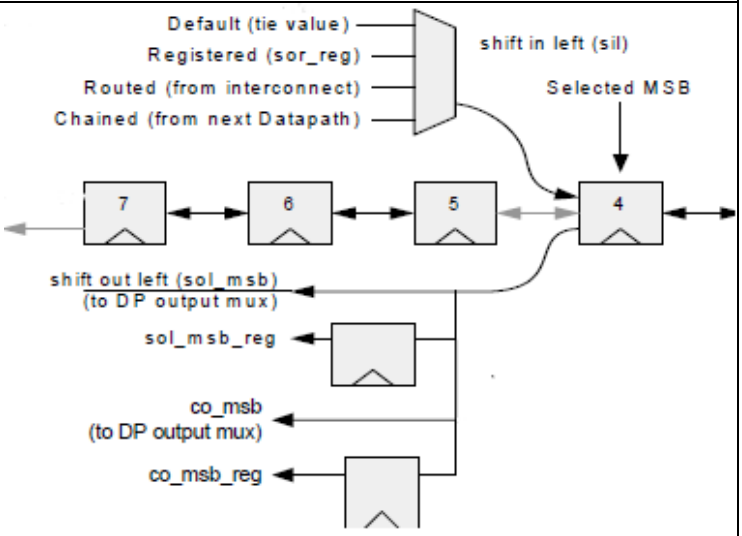
Table 19. CFG13 and CFG12 Register Definitions

CFG13-12 Configuration Selections	
Datapath Configuration Tool	
CFG13-12 Reset | Binary Value | CMP SELB | CMP SELA | CI SELB | CI SELA | CMASK1 EN | CMASK0 EN | A MSK EN | DEF SI | SI SELB | SI SELA 00000000 | 00000000 | A1_D1 | A1_D1 | ARITH | ARITH | DSBL | DSBL | DSBL | DEF_0 | DEFSI | DEFSI	
CFG13-12 Details	
CMP SELB & CMP SEL A Configures the Comparison B and Comparison A Options for compare block 1. The CMP SEL field of the RAM configuration determines if the A or B option is in effect for a given cycle. **Note:** Compare block 0 can compare only D0 and a masked value from A0.	**A1_D1**: A1 < D1, A1 == D1 **A1_A0**: A1 < A0, A1 == A0 **A0_D1**: A0 < D1, A0 == D1 **A0_A0**: A0 < A0 , A0 == A0 
CI SELB & CI SEL A Selects the source of the Carry In.	**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.
CMASK0 EN, CMASK1 EN, AMASK EN Enables the Masks on the Compare 1 and Compare 0 Blocks (shown right) and the Mask block at the output of the ALU (not shown). See Table 17 and Table 18.	
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.	**DEF_0:** Zero **DEF_1:** One
SI SELB & SI SEL A Selects the source of the shift in for the A and B shift in configuration.	**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.

B.2.4 CFG15 and CFG14 Registers

Table 20. CFG15 and CFG14 Register Definitions

<table>
<tr><th colspan="2">CFG15-14 Configuration Selections</th></tr>
<tr><td colspan="2">Datapath Configuration Tool
CFG15-14
<table>
<tr><th>Reset</th><th>Binary Value</th><th>PI SEL</th><th>SHIFT SEL</th><th>PI DYN</th><th>MSB SI</th><th>F1 INSEL</th><th>F0 INSEL</th><th>MSB EN</th><th>MSB SEL</th><th>CHAIN CMSB</th><th>CHAIN FB</th><th>CHAIN 1</th><th>CHAIN 0</th></tr>
<tr><td></td><td>00000000 | 00000000</td><td>ACC</td><td>SL</td><td></td><td></td><td>BUS</td><td>BUS</td><td>DSBL</td><td>BIT0</td><td>NOCHN</td><td>NOCHN</td><td>NOCHN</td><td>NOCHN</td></tr>
</table></td></tr>
<tr><th colspan="2">CFG15-14 Details</th></tr>
<tr><td>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.</td><td>ACC: Source A is sourced by the selection in the dynamic configuration area.
PIN: Source A is sourced by the parallel input to the datapath.
 
</td></tr>
<tr><td>Shift SEL
Controls the selection of the shift out signal at the datapath output mux.</td><td>SL: sol_msb chosen for output
SR: sor chosen for output
 
</td></tr>
<tr><td>PI DYN
Enables dynamic control of parallel in (PI) mux selection to the ALU ASRC input.</td><td>DS: PI mux is controlled statically using the PI SEL bit in this register.
EN: The PI mux is controlled dynamically (assuming PI SEL is ‘0’), using the CFB_EN bit in the dynamic RAM. When this bit is set and CFB_EN is a ‘0’, the ALU ASRC input is A0 or A1, when CFB_EN is a ‘1’, the ALU ASRC input is PI routing.</td></tr>
<tr><td>MSB SI
Supports arithmetic Shift Right operation.</td><td>REG: Default shift in selection is defined by the DEF SI value (CFG12, Table 19).
MSB: Overrides the default shift in value (defined as ‘00’ selection in SELA[1:0]/SELB[1:0]) with the currently defined MSB (MSB EN and MSB SEL fields in CFG14).</td></tr>
</table>

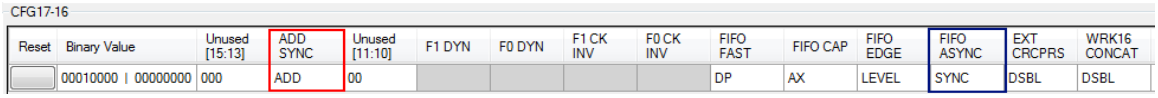
CFG15-14 Details (contd.)	
F1 INSEL & F0 INSEL Defines the input source of FIFO 1 and FIFO 0.	**Bus:** FIFO input is the CPU bus, FIFO output is A0 or D0 Registers **A0:** FIFO input is A0. FIFO Output is CPU BUS. **A1:** FIFO input is A1. FIFO Output is CPU BUS. **ALU:** FIFO input is the ALU. FIFO Output is CPU BUS. 
MSB EN and MSB SEL You can adjust the MSbit of the ALU output. 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 CMSB Enables chaining from the next datapath in the chain to this block for the CRC MSB.	The CRC MSB signal flow is from MS block to LS block. Set this bit when this datapath does not contain the most significant bit of a CRC computation. **NOCHN:** CRC MSB is not chained. **CHNED:** Chain the CRC MSB from the next datapath block in the chain.
Chain FB Enables chaining from the previous datapath in the chain to this block for the CRC feedback.	The CRC FB signal flow is from LS block to MS block. Set this bit when this datapath does not contain the least significant bit of a CRC computation. **NOCHN:** CRC feedback is not chained. **CHNED:** CRC feedback is chained from the previous datapath block in the chain.
Chain 1 & Chain 0 Defines whether the outputs of CL0, CL1, CE0, CE1, Z0, Z1, FF0, and FF1 are chained.	When set to chained (CHNED), the conditions from the previous datapath are chained to this datapath. Chain 0 affects CL0, CE0, Z0, and FF0 conditions. Chain 1 affects CL1, CE1, Z1, and FF1 conditions.

B.2.5 CFG17 and CFG16 Registers

Table 21. CFG17 and CFG16 Register Definitions

FIFO Configurations

Datapath Configuration Tool

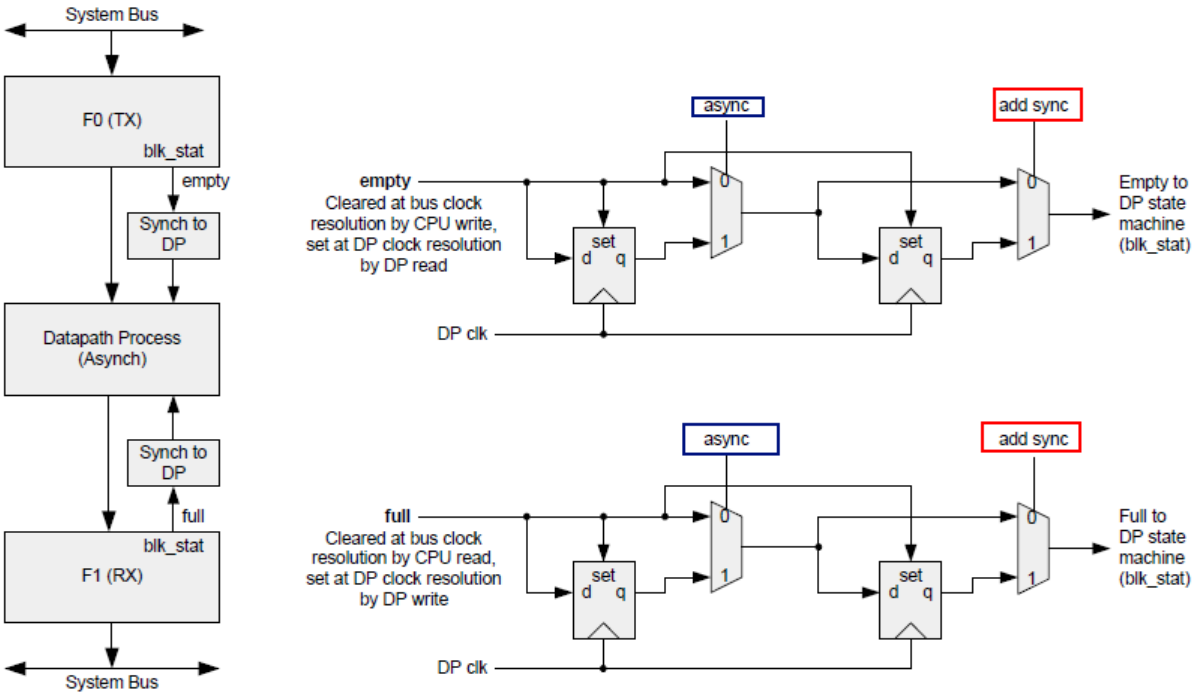


- ADD SYNC – Determines whether an additional sync flip-flop is added to the FIFO block status. This controls the cycle timing between bus reads/writes at bus clock resolution, and the assertion of the new status on datapath output routing. There is only one configuration bit that controls this for both FIFOs. See the FIFO Configurations section on the next page.
  - **NONE**: Does not add a flip-flop to the output of the FIFO block status.
  - **ADD**: Adds a flip-flop to the output of the FIFO block status.
- F1 DYN – Controls whether the FIFO1 direction is static or dynamic. In static mode, the F1_SEL[1:0] bits control the FIFO read and write access. When this bit is set for dynamic mode there are two configurations: internal access, where the FIFO can be read and written to by the Datapath, and external access, where the FIFO can be read and written to by the system bus. In this mode, the F1_SEL[1:0] bits control the FIFO write source in internal access mode.
  - **OFF**: Static Mode. FIFO direction is static and controlled by F1_SEL[1:0].
  - **ON**: Dynamic Mode. FIFO direction is dynamic and controlled between internal and external access by toggling the DP routed signal d1_load.
- F0 DYN – Read description for F1 DYN
- F0 CLK INV and F1 CLK INV – Determine whether the FIFO clock is inverted relative to the datapath clock.
  - **NEG**: FIFO clock is the same polarity as the DP clock
  - **POS**: FIFO clock is inverted with respect to the DP clock
- FIFO FAST – Determines whether the FIFOs are clocked using the datapath clock or the PSoC Bus Clock. In fast mode, the FIFO is clocked by the bus clock, which reduces capture latency.
  - The use of this mode results in slightly higher power consumption because the master and quadrant gating of bus clock must be enabled. This bit controls the mode for both FIFOs in the UDB, but it only applies to FIFOs that are configured in output mode.
  - **DP:** Datapath Clock
  - **BUS:** Bus Clock
- FIFO CAP – Enables FIFO capture mode. If enabled, a read of A0 or A1 will write into F0 or F1, respectively.
  - **AX**: A read of A0 or A1 returns the value in the register directly.
  - **FX**: A read of A0 (or A1) triggers a capture into F0 (or F1).
- FIFO EDGE – 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.
  - **LEVEL**: A FIFO write (output mode) is level sensitive.
  - **EDGE**: A FIFO write (output mode) is edge sensitive.
- FIFO ASYNC – Determines if a flip-flop is needed on the output of the FIFO block status signals. See the FIFO Configurations section on the next page.
  - **SYNC**: Does not add a flip-flop at the output of the FIFO block status.
  - **ASYNC**: Adds a flip-flop to the output of the FIFO block status. Setting ADD SYNC to *NONE* and FIFO ASYNC to *ASYNC* or ADD SYNC to *ADD* and FIFO ASYNC to *SYNC* is acceptable only if the datapath clock has been synchronized to MASTER_CLK.
- EXT CRCPRS – Overrides the internal configuration for CRC/PRS calculation and allows external routing of CRC/PRS signals. When this bit is set, access is given to the raw block inputs for the CRC operation, including the shift in data and the feedback data, and calculations for these signals must be done externally. (Typically in the PLD).
  - **DSBL**: Internal CRC/PRS routing
  - **ENBL**: External CRC/PRS routing.
- WRK16 CONCAT – Controls the working register access mode in the 16-bit access space. By default, when this bit is a '0' the access occurs the same register across a pair of UDBs in chaining order. When this bit is set to '1', a 16-bit read or write accesses concatenated registers within a single UDB. The combinations are {A1,A0}, {D1,D0}, F1,F0}, {CTL,STAT},{MSK, ACTL}, {8'b0,MC}.

FIFO Configurations

- **DSBL**: 16-bit Default Access Mode. A 16-bit access reads/writes a given register in two consecutive UDBs in chain/address order.
- **ENBL**: 16-bit Concat Access Mode. A 16-bit access reads/writes concatenated registers in a single UDB.

FIFO Output Synchronization Diagram

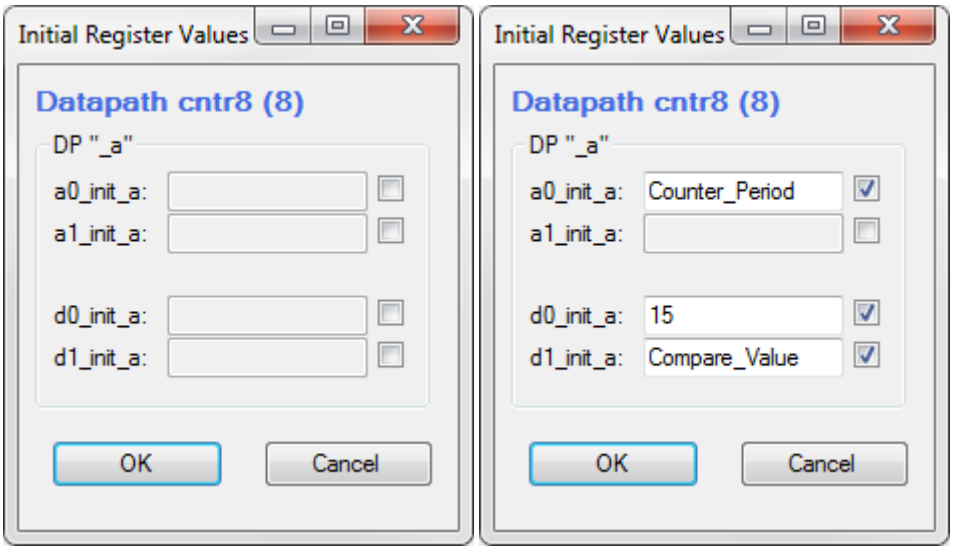


ASYNC	ADD SYNC	Operation	Usage Model
0	0	Synchronous to Bus clock	CPU read/write status changes occur at bus clock resolution. Can be used for minimum latency if the bus clock timing can be met.
0	1	Re-sampled from Bus Clock to DP Clock	This should be the default synchronous operating mode. When the CPU read/write status changes are synchronously resampled with the currently selected DP clock. Gives a full cycle of DP clock setup time to the UDB logic.
1	0	(redundant)	
1	1	Double Synced from Bus Clock to DP Clock	When a free-running asynchronous DP clock is in use, this setting can be used to double sync the CPU read and write actions to the DP clock.

B.3 Setting Initial Register Values

To set the initial values of A0, A1, D0, D1 in the DCT, go to **View** > **Initial Register Values**. For example, if the datapath name is Cntr8, the window would look like Figure 145 (left).

Figure 145. Initial Register Values



You can set the initial values by clicking the checkboxes and entering either a number or a valid parameter name from the destination Verilog file (Figure 145 right).

B.4 Datapath Chaining

Dedicated datapath chaining signals allow efficient implementation of single-cycle 16-, 24-, and 32-bit bit functions without the use of channel routing resources.

As shown in Figure 146, all generated conditional and capture signals chain in the direction of the least significant to the most significant blocks. Shift-left also chains from the least to the most significant. Shift-right chains from the most significant to the least significant. The CRC/PRS chaining signals of CFBO (feedback) chain the least to the most, but the CMSBO (MSB output) chains from the most to the least significant.

Figure 146. Datapath Chaining Signal Flow

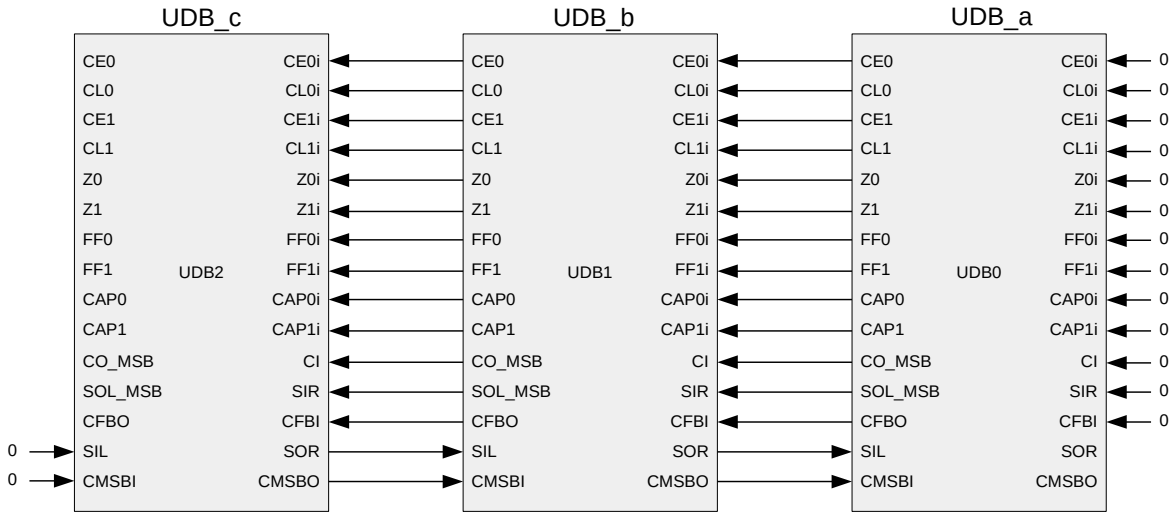
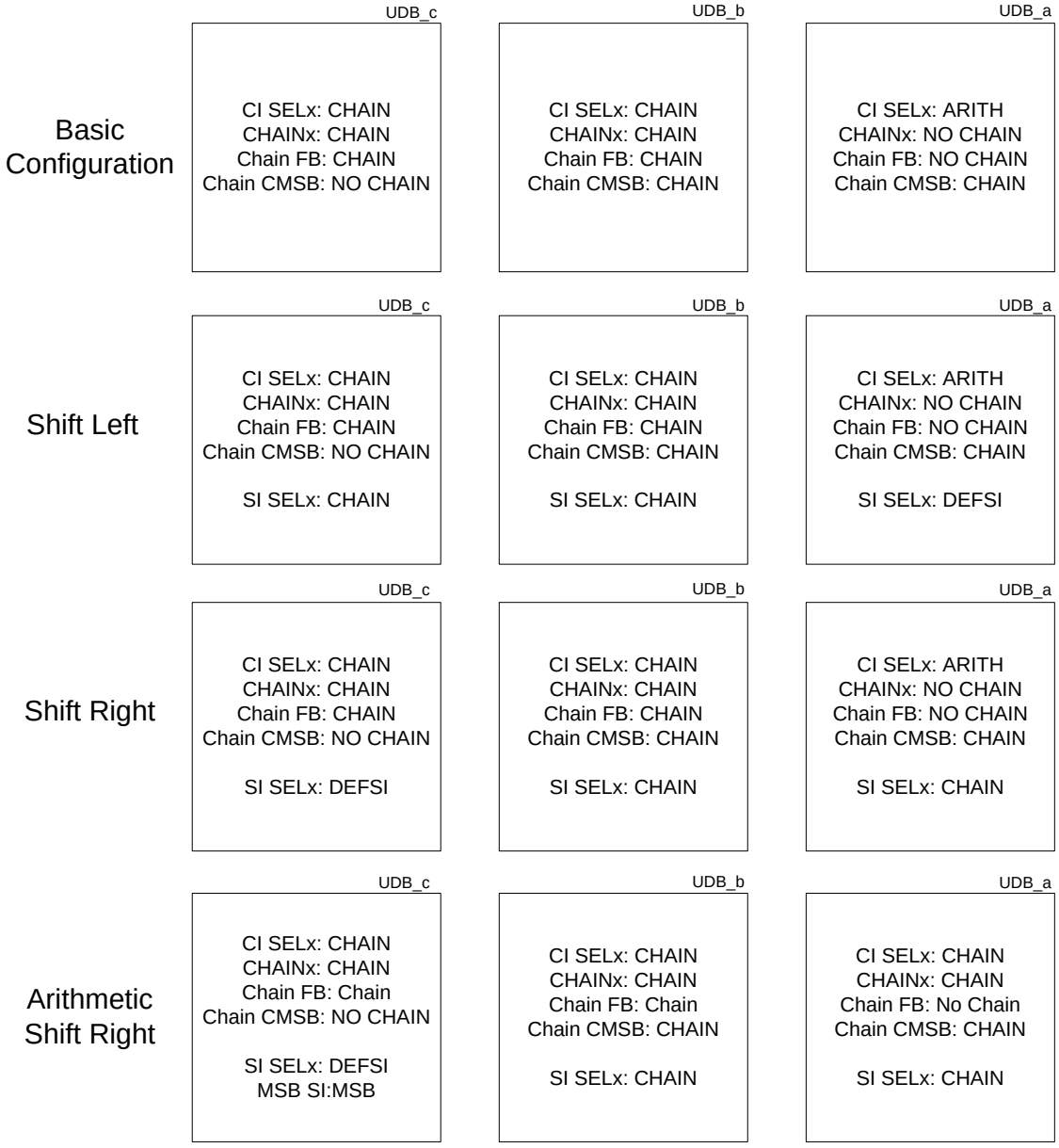


Figure 147 shows the settings required for chaining datapaths for various cases. UDB_a is the least significant block, while UDB_c is the most significant block. Figure 147 describes a 3-UDB (up to 24-bit) function; a 16-bit or 32-bit function can be created by removing or duplicating the middle datapath configuration. The figure shows configuration for Chain FB and Chain CMSB even though they may not be used.

Keeping Figure 146 in mind, when chaining together datapaths, a majority of designs (for example, simple adding or subtracting) would use the 'Basic Configuration' row in Figure 147; that is, chain all signals from LSB (UDB_a) to MSB (UDB_c) except for Chain CMSB. If you perform any shift operations, based on the direction of shift, you need to change the shift-chaining configuration – shown in the Shift-Left, Shift-Right, and Arithmetic Shift-Right rows in Figure 147.

Figure 147. DCT Configuration for Chaining



B.5 Firmware-Control of Datapath Registers

The datapath registers can be accessed in firmware by using the macros CY_SET_REG8 (addr, value) and CY_GET_REG8 (addr), or the corresponding 16-, 24-, or 32-bit versions of these functions as the case may be. The address of the registers can be found in the *cyfitter.h* file (generated after a successful build). For example, if the 8-bit datapath instance named “cntr8” is instantiated in a Component named “SimpleCntr8_1”, the *cyfitter.h* file contains a block of code which lists the addresses of all the datapath registers; see Figure 148.

Figure 148. Addresses of Datapath Registers

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

B.6 Miscellaneous

For more information about the Datapath Configuration Tool, see Appendix B of the Component Author Guide, available in the DCT under **Help** > **Documentation**, or in PSoC Creator under **Help** > **Documentation** > **Component Author Guide**.

C Appendix C – Force Datapath Placement

If you desire to place your datapath component in a specific UDB there is a way to do so. In the .cydwr file under the directives tab you can add directives, one of those directives is ForceComponentUDB. If you search for directives in the PSoC Creator Help Topics file you will find some directions on how to use them. Below is an example:

\ `$INSTANCE_NAME`:DATAPATH0 ForceComponentUDB U(0,0)

You need to replace `$INSTANCE_NAME` with the name of your component, and replace DATAPATH0 with the name of the datapath inside of your component.

The U(0,0) stands for UDB(row, column) in the PSoC Creator Help Topics file if you search for PSoC UDBs in PSoC UDBs in PSoC Creator topic you will see a mapping of the UDBs. Refer to the Row and Column number. The lines connecting the UDBs show how they can be chained.

Also note if you have more than one UDB in a chain you can only force the placement of the MSB Datapath, if you try and force the placement of the other Datapaths Creator will throw an error.

D Appendix D – Auto-Generated Verilog Code

This appendix gives 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.

D.1 New Verilog File Generated by PSoC Creator

When a Verilog file is first generated by PSoC Creator, it looks 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.

D.2 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 22 explains the hardware links found in the datapath configuration.

Table 22. 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
.ce1(),	Compare block 1 "equals" output
.cl1(),	Compare block 1 "less than" output
.z1(),	Zero detect block 1 output

Verilog Code	Definition / Mapping
.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 1 bus status flags*
.f1_blk_stat()	FIFO 1 block status flags*

*For More information on these status outputs, refer to 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.

D.3 Verilog File with SimpleCntr8 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), /* tie clk signal to datapath clock */
        /*  input   [02:00]         */  .cs_addr({2'b0,tc}), /* tc determines address lsb */
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0_load(1'b0),
        /*  input                   */  .f1_load(1'b0),
        /*  input                   */  .d0_load(1'b0),
        /*  input                   */  .d1_load(1'b0),
        /*  output                  */  .ce0(),
        /*  output                  */  .cl0(),
        /*  output                  */  .z0(tc), /* tc signal comes from z0 state */
        /*  output                  */  .ff0(),
        /*  output                  */  .ce1(),
        /*  output                  */  .cl1(),
        /*  output                  */  .z1(),
        /*  output                  */  .ff1(),
        /*  output                  */  .ov_msb(),
        /*  output                  */  .co_msb(),
        /*  output                  */  .cmsb(),
        /*  output                  */  .so(),
        /*  output                  */  .f0_bus_stat(),
```

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

The two configurations 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.

E Appendix E – Example 24-Bit Datapath with PI and PO

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

// Datapath 0
cy_psoc3_dp #(.cy_dpconfig(
{
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
        8'hFF, 8'h00,      /*SC_REG4     Comment: */
        8'hFF, 8'hFF,      /*SC_REG5     Comment: */
      `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
      `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
      `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
      `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
      `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
      1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
      `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
      `SC_FB_NOCHN, `SC_CMP1_NOCHN,
      `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
       10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
      `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
      `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
})) Datapath0(
      /* input */ .clk(),              // Clock
      /* input [02:00] */ .cs_addr(),  // Control Store RAM address
      /* input */ .route_si(1'b0),     // Shift in from routing
      /* input */ .route_ci(1'b0),     // Carry in from routing
      /* input */ .f0_load(1'b0),      // Load FIFO 0
      /* input */ .f1_load(1'b0),      // Load FIFO 1
```

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

// Datapath 1
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
```

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

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

// Datapath 2
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
          8'hFF, 8'h00,      /*SC_REG4     Comment: */
          8'hFF, 8'hFF,      /*SC_REG5     Comment: */
        `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
        `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
        `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
        `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
        `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
       1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
```

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

Document History

Document Title: AN82156 – 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 45 to match related documents. 8. Updated Figure 83, Figure 84, Figure 27, Figure 28, Figure 101, Figure 111, Figure 118, Figure 122, and Figure 62 to correct the default Component color. 9. Minor formatting corrections to accommodate other changes.
*B	3833873	GIR	12/6/2012	1. Updated for PSoC 5LP. 2. Changes throughout the document to align with related documents and incorporate review feedback.
*C	3961344	ANTO	04/10/2013	Update for PSoC 4 and PSoC Creator 2.2 SP1. Qualified references to DMA (only in PSoC 3 and PSoC 5LP). Added new way to access Datapath Config Tool from within PSoC Creator. Updated author information. Added link to Datapath Configuration Tool Cheat Sheet.
*D	4147004	RLIU	10/04/2013	Added link to AN79953, Getting Started with PSoC 4. Updated trademark information at the end of the document.
*E	4384670	TDU	05/23/2014	Update AN for UDB Editor Moved Previous Examples to Appendix A Added PI and PO example Moved in full content from “Cheat Sheet”
*F	4771838	NIDH	06/15/2015	Updated associated project Pin mapping table added for the PI PO project Updated template
*G	4929407	TDU	09/29/2015	Fixed a few minor errors Clarified some wording Added section on forcing placement.
*H	5702158	BENV	04/18/2017	Updated logo and copyright
*I	6385318	TDU	11/15/2018	Adding PSoC 6 MCU

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

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Code Examples | Projects | Videos | Blogs | Training| Components

Technical Support

cypress.com/support

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