PSoC® Creator™

Universal Digital Block (UDB) Editor Guide

Document Number 001-94131, Rev. **, 9/24/2014

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone): 408.943.2600
http://www.cypress.com

Copyrights

Copyright © 2014 Cypress Semiconductor Corporation. All rights reserved.

"Programmable System-on-Chip," PSoC, PSoC Designer, and PSoC Express are trademarks of Cypress Semiconductor Corporation (Cypress), along with Cypress® and Cypress Semiconductor™. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

The information in this document is subject to change without notice and should not be construed as a commitment by Cypress. While reasonable precautions have been taken, Cypress assumes no responsibility for any errors that may appear in this document. No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Cypress. Made in the U.S.A.

Disclaimer

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Flash Code Protection

Cypress products meet the specifications contained in their particular Cypress PSoC Data Sheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods, unknown to Cypress, that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.

Contents

Introduction **4**
What is a UDB? 4
Conventions 6
Acronyms and Abbreviations 6
References 6
Revision History 6

UDB Editor Overview **7**
Opening the UDB Editor 8
UDB Editor Elements 9
Datapath 10
Control Register 18
Status Register 19
Status Interrupt Register 20
Count7 Counter 21
State Machine 22
UDB Editor APIs 25

Example UDB Editor Design **27**
Step 1: Create a Custom Component 28
Step 2: Define the Component Inputs and Outputs 29
Step 3: Create a State Machine to Control the Datapath Inputs 30
Step 4: Configure the Datapath 34
Step 5: Create the Component Symbol 41
Step 6: Build the Component APIs 42
Step 7: Use the New Component 44
Additional projects 46

Appendix A: Datapath Operation **47**
Datapath Instructions 47
Datapath Registers 48
Datapath Inputs/Outputs 49
FIFO Modes 49

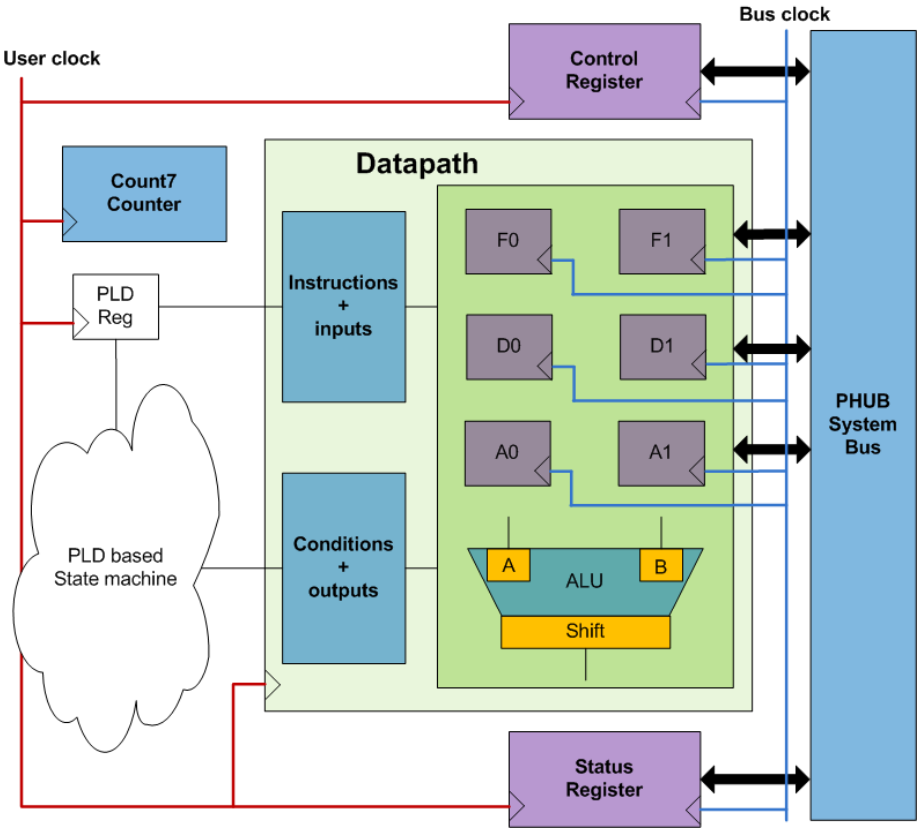
Appendix B: UDB Editor Syntax **51**

Introduction

This document provides a guide to learn about and use the PSoC Creator Universal Digital Block (UDB) Editor. It provides basic information about UDBs, an overview and description of the UDB Editor, and an example UDB Editor design. The appendices provide more detailed information about UDB elements and expressions.

What is a UDB?

A UDB is a flexible, programmable digital block inside a PSoC device that is designed to realize synchronous State Machines. The following figure shows the main blocks in a UDB from a high level. For detailed information about the UDB architecture of a specific device, refer to that device *Technical Reference Manual (TRM)*.



A UDB is capable of an array of functions including:

- Cascading multiple UDBs to make a wider than 8-bit function
- Communication components (like SPI, I^2C, and UART)
- Flexible logic machines (like PWMs)

Each 8-bit wide UDB has five elements:

- A Datapath element is an 8-bit wide processor that can be used to perform simple arithmetic and bitwise operations on data words. It can be chained to form 16-, 24-, and 32-bit wide processors. It can have up to 8 user-defined instructions that are often driven using the programmable logic device (PLD) based State Machine. Datapaths form the core of many UDB designs and should be used in preference over PLD designs when 8-bit words or larger are used. A DP element has:
  - A programmable 8-bit wide Arithmetic Logic Unit (ALU)
  - Two 8-bit accumulator registers (A0, A1)
  - Two 8-bit data registers (D0, D1)
  - Two 8-bit wide, 4 deep FIFOs (F0, F1)
  - A shifting function
  - A masking function

  See Datapath Registers for more details about registers.
- Two 12C4 PLDs (used to create State Machines). These are most often used to create logic to control the other structured resources available in a UDB. PLD-based designs are composed of both combinational logic and sequential logic. The sequential logic is driven with the user clock. Although PLDs are the most flexible element in a UDB, it is also relatively limited resource. Therefore it is recommended to use the other structured blocks as much as possible when implementing large designs.
- An 8-bit wide Control Register (accessible from the processor and the PSoC hardware). A Control Register is used in a UDB to control the digital hardware with the CPU. Using a Control Register, it is possible for the CPU to directly send logic values to the UDB hardware. The reading and writing of the Control Register from the CPU is performed at the bus clock, unless it is in sync or pulse mode with the user defined clock. In this case, it will run at the clock rate of the UDB Editor component (specified by the "clock" input of the component).
- An 8-bit wide Status Register (accessible from the processor and the PSoC hardware). A Status Register is used to read logic values from the UDB and CPLD logic into the CPU. The rate at which the Status Register reads the digital logic is controlled by the bus clock. Status registers also have the ability to mask 7 bits in the status register for generating an interrupt. This is accomplished by using one pin for the interrupt output and the other 7 bits as the maskable triggers for the interrupt.
- A 7-bit down counter (Count7) that can be used instead of implementing a counter in a State Machine/Verilog or a Datapath. This uses the same resource as a Control Register. The terminal count of the counter can then be used throughout your design.

These elements can be used to form many types of logic and can be chained together to form large designs. A design may communicate with the CPU, with other hardware blocks in a PSoC device, or both. This flexibility allows the UDB to form logic that links other hardware in your design, or can be a stand-alone block that performs a new function.

UDBs can be divided into uncommitted logic and structured logic. The structured logic includes the Datapath, Control Register, Status Register (Status Interrupt Register if enabled), and Count7 counter. These can be controlled by the CPU through the Control Register and Status Register. Alternatively, the uncommitted PLD logic can be used to design State Machines that can generate control signals for these blocks.

UDBs are driven with a user clock and the bus clock. The bus clock synchronizes reads from and writes to the registers in the Datapath and the Control Register and Status Register. These data words travel through the Peripheral Hub (PHUB) system bus. The user clock drives the blocks in the UDB. Signals in a UDB can be routed to form a hardware output in a component or can be used to drive the inputs of the structured blocks in a UDB.

Conventions

The following table lists the conventions used throughout this guide:

Convention	Usage
`Courier New`	Displays file locations and source code: `C:\ …cd\icc\`, user entered text
Italics	Displays file names and reference documentation: *sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > New Project	Represents menu paths: **File > New Project > Clone**
Bold	Displays commands, menu paths and selections, and icon names in procedures: Click the **Debugger** icon, and then click **Next**.

Acronyms and Abbreviations

This guide contains the following acronyms and abbreviations:

- ALU – Arithmetic Logic Unit
- API – Application Programming Interface
- FIFO – First In, First Out
- HDL – Hardware Description Language
- PHUB – Peripheral Hub
- TRM – Technical Reference Manual
- UDB – Universal Digital Block

References

This guide is one of a set of documents pertaining to UDBs. Refer to the following other documents as needed:

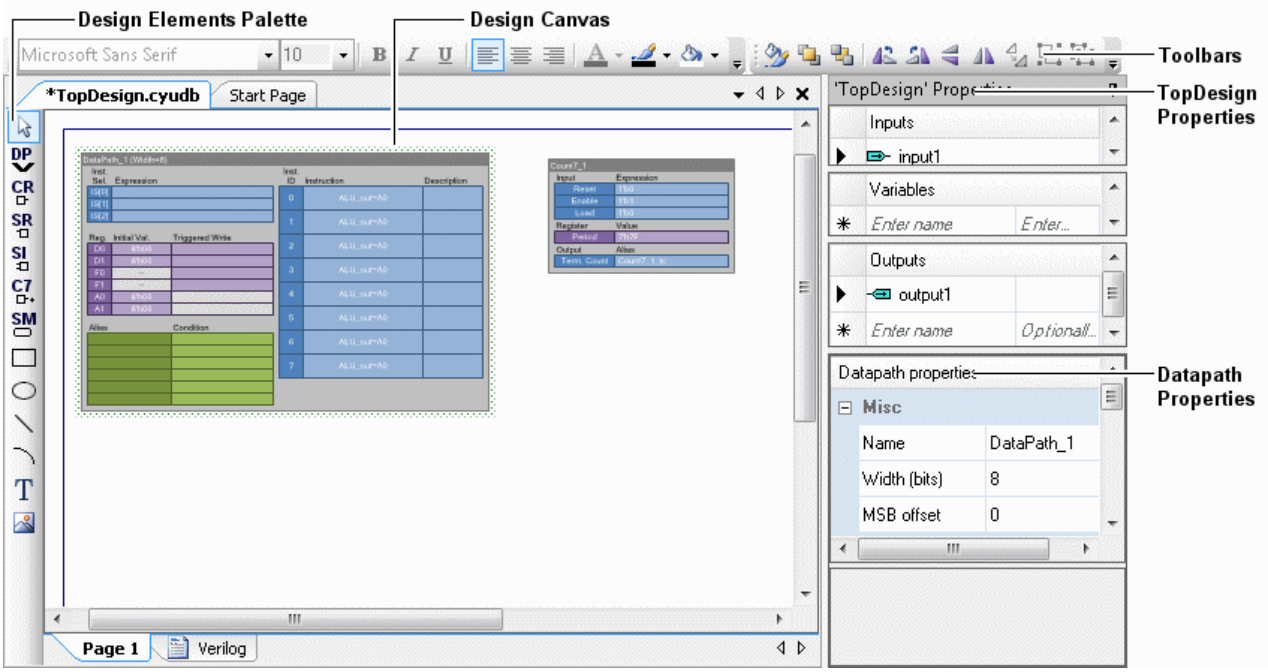
- PSoC Creator Help
- PSoC Creator Component Datasheets
- PSoC Creator Component Author Guide
- PSoC Technical Reference Manual (TRM)
- Application Note AN82156: Designing PSoC Creator Components with UDB Datapaths

Revision History

Document Title: PSoC® Creator™ UDB Editor Guide **Document Number: 001-94131**		
Revision	**Date**	**Description of Change**
**	9/24/14	New document.

UDB Editor Overview

The UDB Editor allows you to create UDB-based designs with very little knowledge of digital logic or Verilog code. Using this graphical tool, you drag, drop, and then configure your hardware without having to write Verilog code. The UDB Editor takes care of many internal configuration details simply by specifying the parameters of the UDB blocks on the design canvas. The tool translates your design to Verilog in real time, allowing you to see how the UDB blocks translate to Verilog hardware description language (HDL).



Note The UDB Editor allows you to construct UDB-based designs without the need of writing Verilog or using the more advanced Datapath Configuration Tool. However, using this tool 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, and this may be limiting for complex designs. For more information about using Verilog and the Datapath Configuration Tool, refer to the *PSoC Creator Component Author Guide*, as well as *Application Note AN82156: Designing PSoC Creator Components with UDB Datapaths*.

The main areas of the UDB Editor include:

- **Design Elements Palette –** The design elements palette is a menu used to choose the UDB elements to include in your design. See UDB Editor Elements.
- **Design Canvas Pages –** Once you open a UDB Editor document, you will see an editable page like a schematic page. This is your design canvas, used to place and configure your UDB elements. Additional UDB Editor pages may be added by right-clicking on the **Page 1** tab. These pages then are translated to Verilog as a single unit. **Verilog –** Next to the **Page** tab, you can find the **Verilog** tab. This is a read-only view of the translated HDL for your design. It is dynamically updated so whenever a change is made in

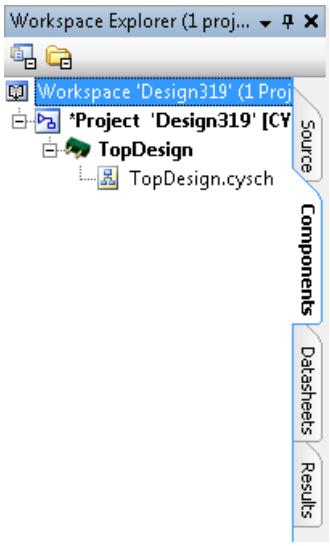
your design, it will also update the code. This code is not editable, and you cannot delete it; any desired changes must be made in the design canvas. You may also copy and paste this code to a Verilog file if you wish to edit your design using Verilog. Refer to Application Note AN82156: Designing PSoC Creator Components with UDB Datapaths for more details.

- **Design Properties –** Located to the right of the design canvas, the design properties allow you to configure the inputs, outputs, and variables used in your design. The inputs and outputs will then form your symbol terminals when your design is complete. The design properties window is also used to set the global Datapath configuration when a Datapath is selected in your design.

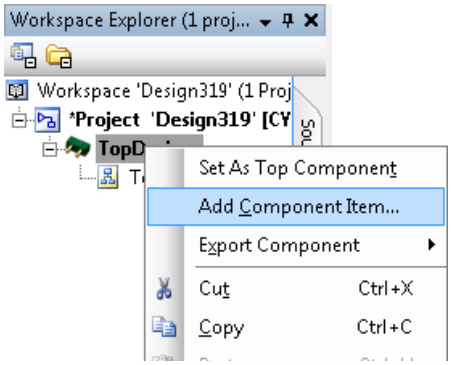
Opening the UDB Editor

The UDB Editor is used to create a custom UDB-based component, which you then use in a design. So, to open and use the UDB Editor, you have to create a component. For more detailed information about creating components, refer to the *PSoC Creator Component Author Guide*.

1. Create a new project or open an existing project.
2. In the Workspace Explorer, click the **Components** tab.



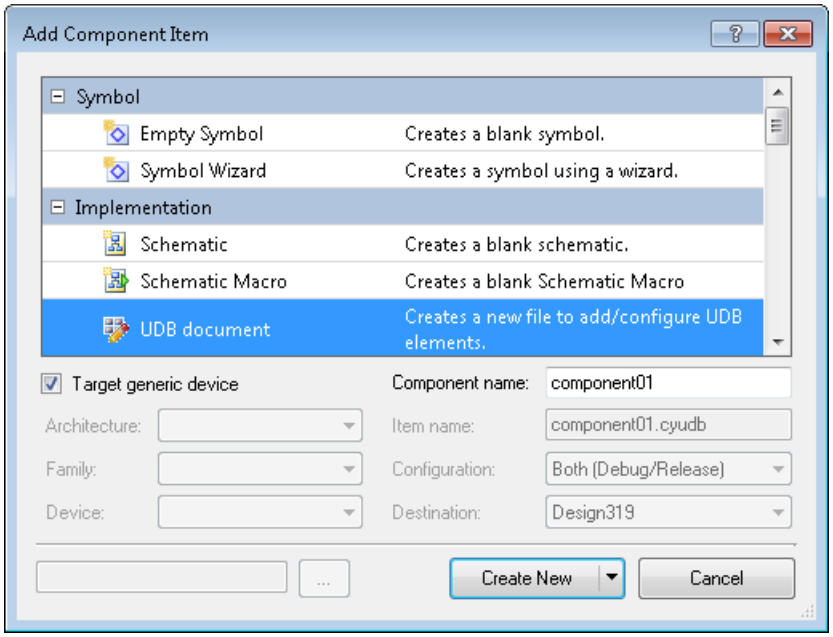
3. Right-click on the project or component, and select **Add Component Item…**



Note If you add a component item to a Project, you will create a new component; if you add the item to an existing component, the component item will inherit properties from that component.

4. On the Add Component Item dialog, select the **UDB document** template under the **Implementation**

category.



Note For a new component, enter a **Component name**. If desired, you can also specify target options by unselecting **Target generic device** and choosing a specific **Architecture**, **Family**, and/or **Device**.

5. Click **Create New** and PSoC Creator opens the UDB Editor.

UDB Editor Elements

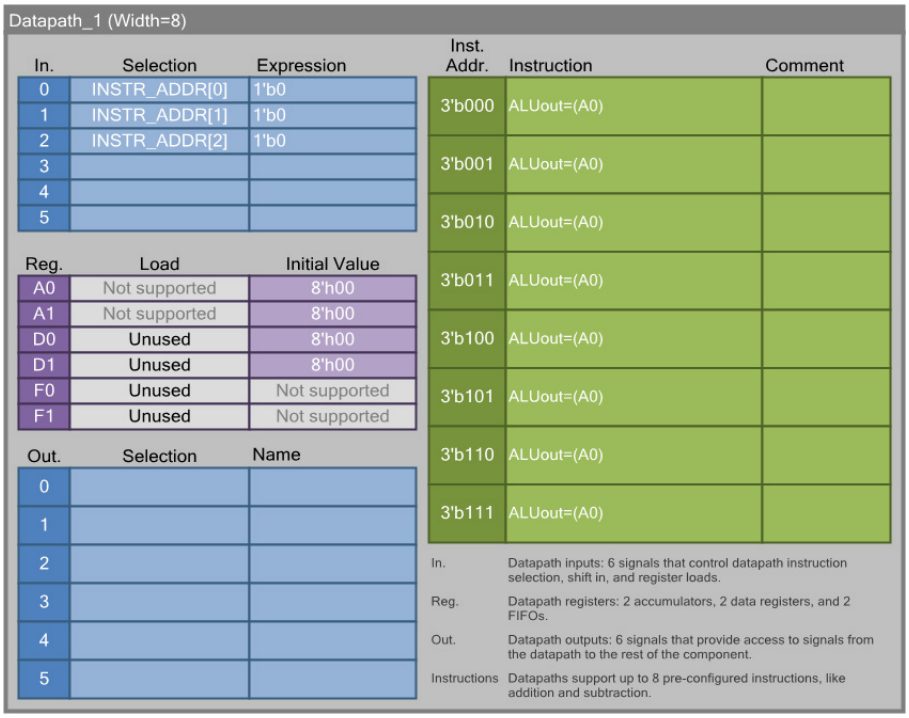
UDB Editor elements are the graphical versions of UDB elements. These are available to drag and drop from the Design Elements Palette. They include:

- Datapath (DP)
- Control Register (CR)
- Status Register (SR)
- Status Interrupt Register (SI)
- Count7 counter (C7)
- State Machine (SM)

When placed on the design canvas, an element becomes an instance in your design. Each of these elements/instances is described in the following sections.

Datapath

The following shows an instance of a Datapath element in the design canvas of the UDB Editor. It contains six inputs and six outputs shown in blue, six registers shown in purple, and eight instructions shown in green.



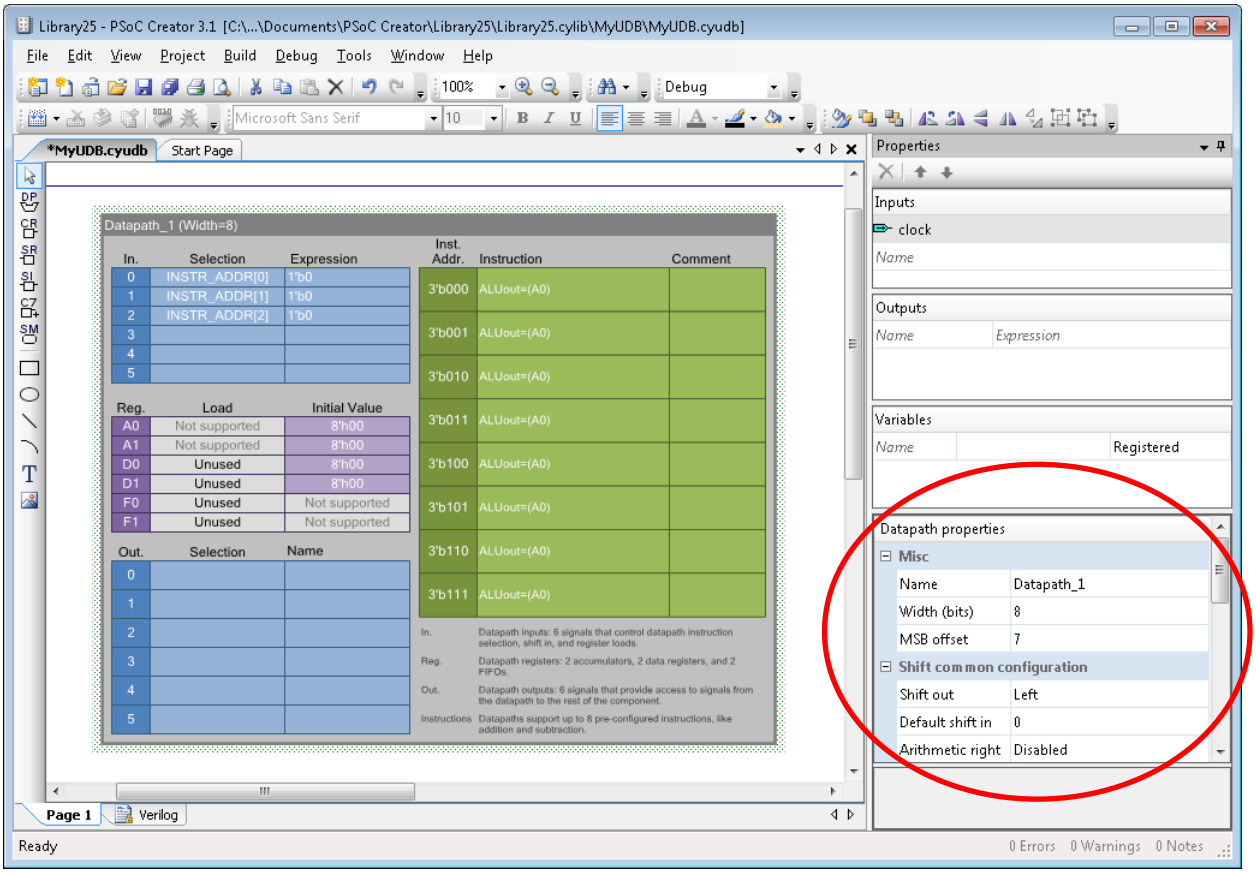
Each UDB block contains an 8-bit Datapath; therefore, chaining these or using multiple Datapaths consumes multiple UDB blocks. Designing with the UDB Editor allows the chaining of these Datapaths to be done automatically, so you do not need to do anything special beyond selecting the datawidth.

When using the UDB Editor, the input/output direction of the FIFO is automatically detected based on the Datapath input configuration. If one of the inputs is set as a load trigger for a FIFO, then that FIFO will be configured as output. Otherwise, it is set to input.

Of the six available input bits, up to three bits can be used to control the Datapath instructions for that clock cycle. If not all three instruction bits are driven, then those instruction bits that are not used will be driven with 1'b0. Note that in this case, not all of the eight available instructions will be usable. For more information on the available instructions, see Datapath Instructions.

Datapath Properties

Clicking on the Datapath will also show the Datapath properties in the UDB Editor window. This allows the global Datapath settings to be specified.



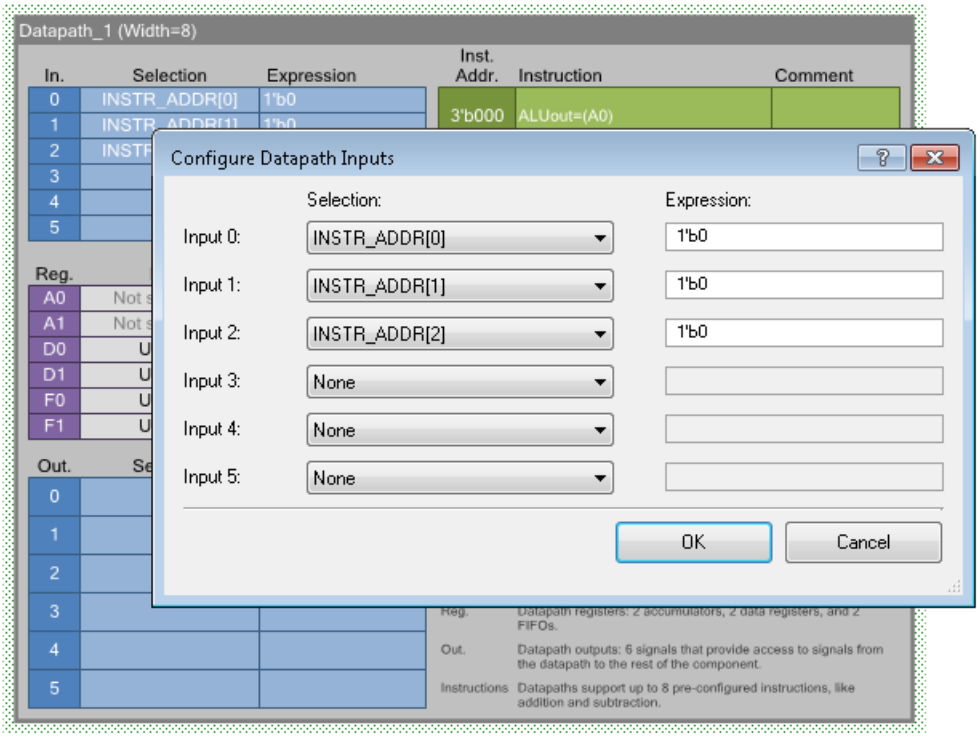
Datapath properties globally affect the selected Datapath instance. These include the bit width size of the Datapath, its shift configurations, compare operation configurations, mask definitions and FIFO modes. Once these are set, all the specific properties of the Datapath follow these global settings. The Datapath properties appear in the Properties panel. The configuration properties available include:

Category	Property	Description
Misc	Name	Instance name of the Datapath.
	Width (bits)	The bit width of the Datapath (8/16/24/32).
	MSB offset	Selects the most significant bit in the Datapath. Used in functions that utilize words that are not multiples of 8. This only impacts Carry Out and Shift Out.
Shift Common Configuration	Shift out	Selects whether shift out left or shift out right is routed to the dedicated shift out output (not one of the 6 configurable outputs).
	Default shift in	Determines the value shifted in when Default is chosen as the Shift source in the Shift Configuration A/B.
	Arithmetic right	Enables arithmetic right shifting when a right shift is selected / used (the value shifted into the MSb is maintained. That is, 1000000 shifted to the right by 1 bit would become 1100000, and 0111111 would become 00111111).
Shift	Shift direction	Chooses the direction in which the shift will occur.

Category	Property	Description
Configuration A/B	Shift in source	Chooses whether to use the dedicated route Shift-in or Default Shift-in expression.
Configurable comparator inputs	Config A	Selects the type of comparison to be made if Compare Config A is selected in an instruction. This comparison is made by comparator 1 at each instruction cycle. Compare operations are A == B and A < B.
	Config B	Selects the type of comparison to be made if Compare Config B is selected in an instruction. This comparison is made by comparator 1 at each instruction cycle. Compare operations are A == B and A < B.
Masks	amask	Mask value that is applied to the output of the ALU. The mask is used only if amask check box is checked. When enabled, the output of the ALU is **always** ANDed with the mask value.
	cmask0	Mask value that is applied to the A (first) input of the comparator. The mask is ANDed with the register then compared. The mask is always applied when checked.
	cmask1	Mask value that is applied to the A (first) input of the comparator, the mask is ANDed with the register then compared. The mask is always applied when checked.
FIFOs	FIFO sync mode	Determines how the FIFO block status signal is synchronized to the datapath clock.
	Capture mode	Specifies whether a read from the accumulator registers is direct or if it also triggers a capture into the FIFOs.
	Edge mode	Specifies whether a FIFO write trigger is level sensitive or rising edge sensitive.
	Fast mode	Specifies the clocking source for the FIFO capture. Can be captured using the Datapath clock or can use the bus clock for a faster performance.

Datapath Inputs

Double-clicking on any of the input fields in the Datapath will open the Configure Datapath Inputs dialog for specific configuration of each input.



Datapath inputs can be used to: control the execution of the instructions, control the loading of the FIFOs/Data/Accumulator registers, or route in a value for the shift in from the DSI. The inputs are also used to define the serial input into the shifter and for controlling the state of the instructions in the Datapath. The following are the available inputs from the UDB Editor.

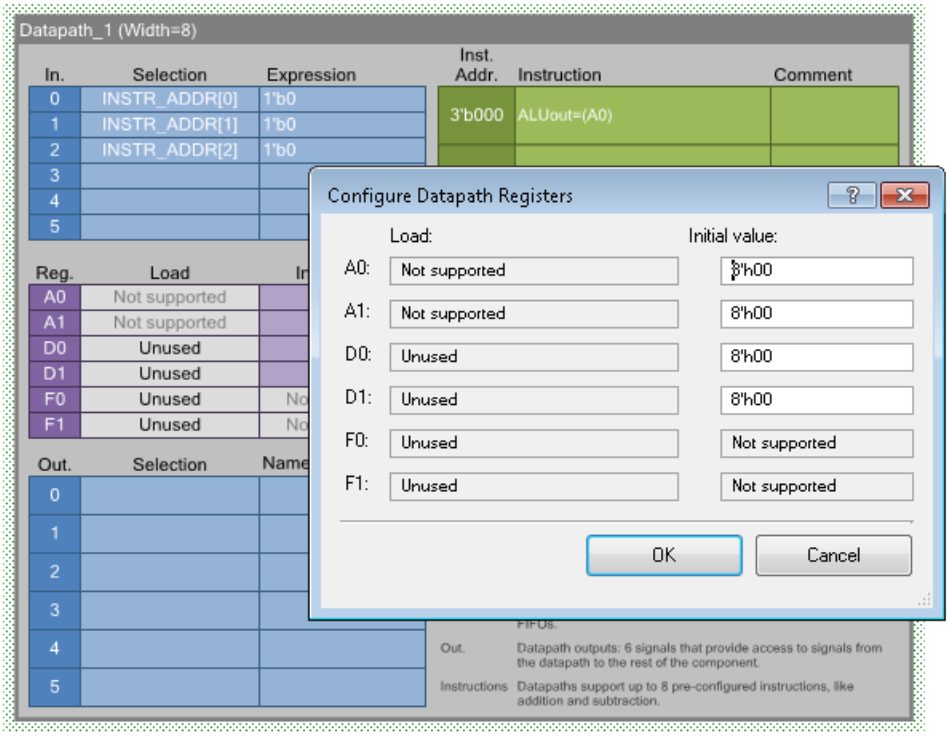
Selection	Name
INSTR_ADDR[0]	LSB of the instruction select bits
INSTR_ADDR[1]	Middle bit of the instruction select bits
INSTR_ADDR[2]	MSB of the instruction select bits
Load D0 with F0	Loads the first element of F0 in D0 at the rising edge of the clock
Load D1 with F1	Loads the first element of F1 in D1 at the rising edge of the clock
Load F0 with A0	Loads the content of A0 to F0 at the rising edge of the clock
Load F0 with A1	Loads the content of A1 to F1 at the rising edge of the clock
Load F0 with ALUout	Loads the content of ALU output to F0 at the rising edge of the clock
Load F1 with A0	Loads the content of A0 to F0 at the rising edge of the clock
Load F1 with A1	Loads the content of A1 to F1 at the rising edge of the clock
Load F1 with ALUout	Loads the content of ALU output to F1 at the rising edge of the clock
Shift In	The expression to use when 'Routed Shift-in' is selected as the shift source for Shift Configuration A/B.

Of the six available inputs, up to three inputs can be used to define which Datapath instruction will be executed. The instruction input is not synchronized to the Datapath clock, it is combinatorial and the Datapath will immediately execute whatever instruction is selected on the Datapath instruction line. These are shown by default in the inputs section: INSTR_ADDR[2:0]. The expressions for these inputs default to 1'b0 (logic low). For each instruction bit that is used, generally the expression for that bit will correspond to a signal being driven by a separate state machine placed into the UDB Editor document.

For example, a variable named myInstr[1:0] in a State Machine is being assigned various values to control the instructions in the Datapath. Then INSTR_ADDR[1] and INSTR_ADDR[0] should assign myInstr[1] and myInstr[0] respectively in its expression fields. Similarly if a "Load F0 with A0" signal is used in the Datapath and the signal used to control this is called loadF0, then loadF0 should be placed in the expression field next to the "Load F0 with A0" input. For more information, see State Machine.

Datapath Registers

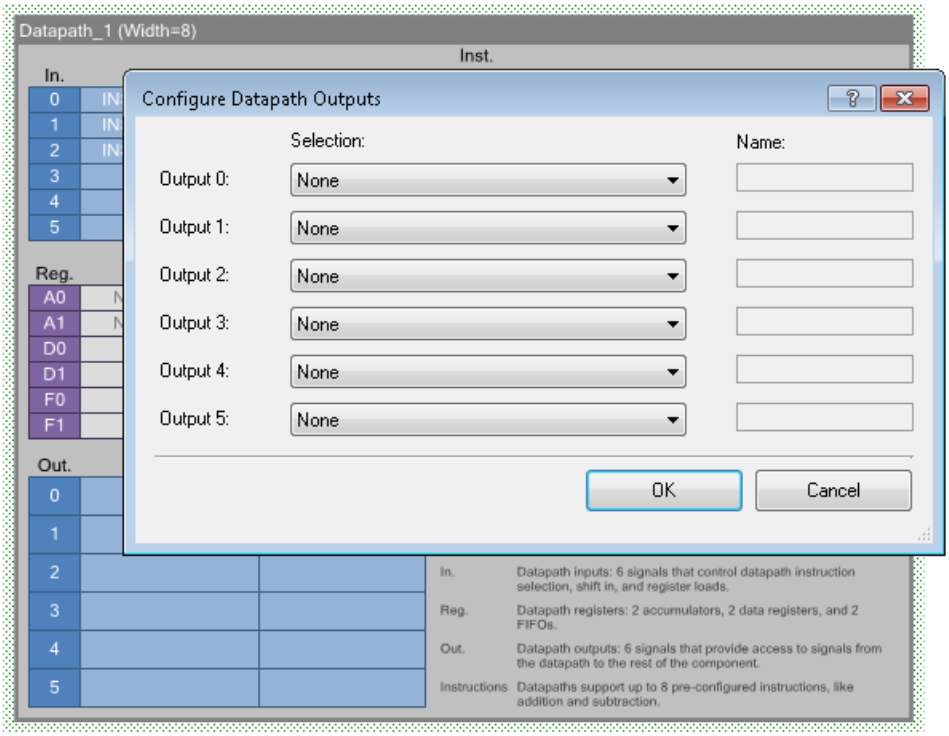
Double-clicking on any of the registers fields will open the Configure Datapath Registers dialog.



This dialog is used to set the initial values of the registers at start-up of the PSoC during boot initialization. If the register does not support an initial value in its current configuration, then it will be grayed out. For reference, the load expression (if any) is shown for each register. This is a read-only display; load expressions must be edited via the Configure Datapath Inputs dialog. For more information, see Datapath Inputs. See also Datapath Registers.

Datapath Outputs

Double-clicking on any of the output fields in the Datapath will open the Configure Datapath Outputs dialog for specific configuration of each output.



Datapath outputs include various comparator status values, FIFO status values, shift output, and various overflows. The following is a list of available outputs from the UDB Editor.

Selection	Name
A0 == D0	Status of A0 equals D0 performed by comparator 0
A0 < D0	Status of A0 less than D0 performed by comparator 0
A0 == 0	A0 equal to 0 comparison.
A0 == 0xFF	A0 equal to 0xFF comparison.
Config A: "equal" Config B: "equal"	Status for equality comparison for comparator 1.
Config A: "less than" Config B: "less than"	Status for less than comparison for comparator 1.
A1 == 0	A1 equal to 0 comparison.
A1 == 0xFF	A1 equal to 0xFF comparison.
Overflow	This is used to monitor whether an overflow has occurred in the most recent ALU operation.
Carry out	Carry out of the ALU arithmetic operation
CRC MSB	CRC feedback out

Selection	Name
Shift out	Shift out bit. This can be either shift out left or shift out right. This is the option selected in the Datapath Properties.
F0 bus status (not full)	FIFO 0 bus status used to flag whether the FIFO 0 is full or not.
F0 block status (not empty)	FIFO 0 block status to flag whether FIFO 0 is empty.
F1 bus status (not full)	FIFO 1 bus status used to flag whether the FIFO 1 is full or not.
F1 block status (not empty)	FIFO 1 block status to flag whether FIFO 1 is empty.

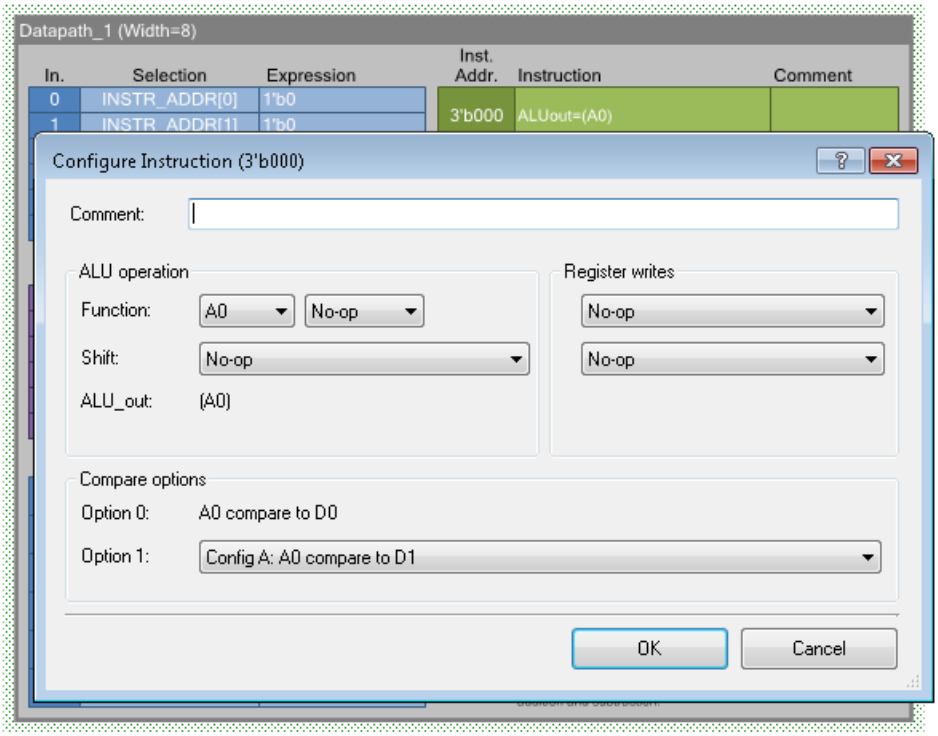
Note For more information about F0 and F1 bus/clock status, refer to Appendix A: Datapath Operation.

Outputs from the Datapath can be used in a State Machine to control transitions (for example, to change the instruction code for the next cycle) or can be directly linked to the output terminal to be used by other hardware blocks (for example DMA or an interrupt). These signals can also be routed internally to other hardware blocks such as another Datapath, a Status Register, or a Count7 counter. For more information on how a State Machine interacts with a Datapath, see Controlling Datapath Instructions.

Note A maximum of six outputs are available per Datapath.

Datapath Instructions

Individual configurations for the eight available Datapath instructions can be set by double-clicking each specific instruction field to open the Configure Instruction dialog.



Each instruction is divided into three parts: ALU operation, Register Writes, and Compare options. For a full list of operations, see Datapath Instructions.

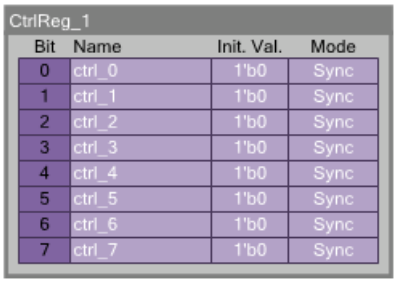
- The ALU operation determines what arithmetic or Boolean operation is performed for that instruction cycle.

- The ALU accepts data from two register sources and performs a function on them. The first input (A) is limited to either A0 or A1 whereas the second input (B) can also accept D0, D1 and can internally provide a 1 as an input. The result is then passed to ALUout.
- ALUout can be shifted by 1 bit before being placed at the output. Specify the shift operation if the result should be shifted by a single bit. The shift properties are controlled by the global shift configuration in the Datapath properties window.
- The expression for ALUout will be shown once you've completed the function and shift definitions. Use this to check whether the ALU operation is correct.

- Register Writes are used to load A0 and A1 with values for the next Datapath clock cycle. These can be used as a feedback from ALUout or to accept new data from the data register or the FIFO.
  - A0 can either be left as it is, or can be over written by D0 or F0. It can also be over written with ALUout to form a feedback.
  - Similarly, A1 can either be left as it is, or can be over written by D1 or F1. It can also be over written with ALUout to form a feedback.
  - If ALUout is not assigned to any registers after an instruction cycle, the result will be overwritten by the result of the next instruction. Therefore it is advised that ALUout be written to either of the accumulators or loaded to a FIFO through the "load FIFO" signals in the Datapath inputs section.
- Compare Options are used to set the comparisons being made using comparator 0 and comparator1. Comparator 0 is always set to compare A0 with D0, and this is shown in option 0. Comparator 1 on the other hand is chosen by specifying whether to use Config A or Config B specified in the Configurable comparator inputs properties located in the Datapath properties window. The status of these comparisons can then be determined by monitoring the comparator outputs in the Datapath output.

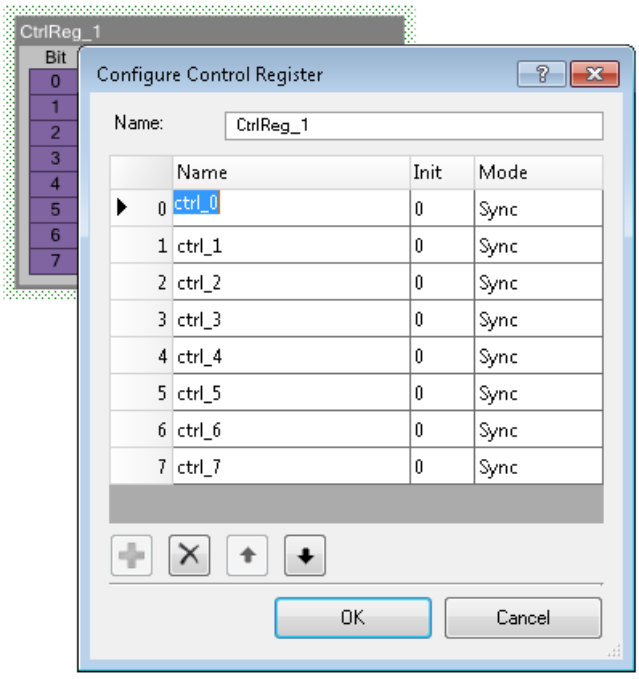
Control Register

The following shows an instance of a Control Register element in the design canvas of the UDB Editor.



Control Registers are used by the CPU to send commands to the digital logic. Each Control Register has eight available bits that can be used throughout the design to control the various aspects of the component operation.

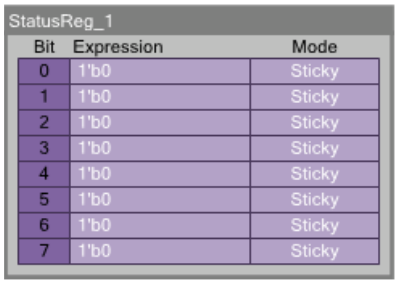
Double-click on the instance to open the Configure Control Register dialog.



A Control Register has a name for each of the bits. These are the signal names that can be used throughout the design. It also has initial values that can be set to either 1'b1 or 1'b0. These values are set during device start-up and will be lost if the device goes to sleep / hibernate. The mode of each of the bits in the Control Register can be set to Direct, Sync or Pulse. The clock used for the Sync and Pulse modes is the component clock. Refer to the Control Register component datasheet (from the separate Control Register component in the PSoC Creator Component Catalog) and the *TRM* for more information.

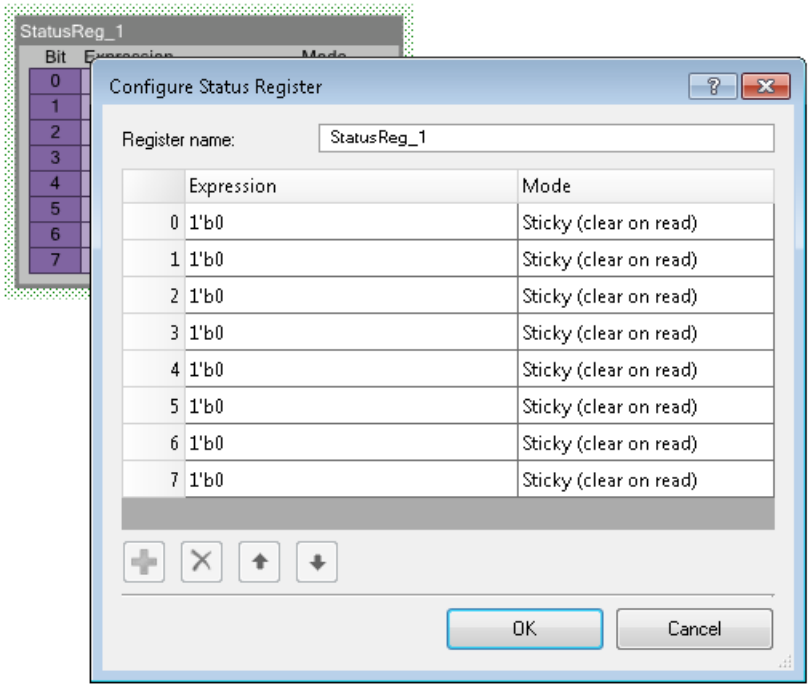
Status Register

The following shows an instance of a Status Register element in the design canvas of the UDB Editor.



Status Registers are used by the CPU to read hardware signals. Eight bits are available in a single Status Register and allows signals from the component to be seen by the CPU.

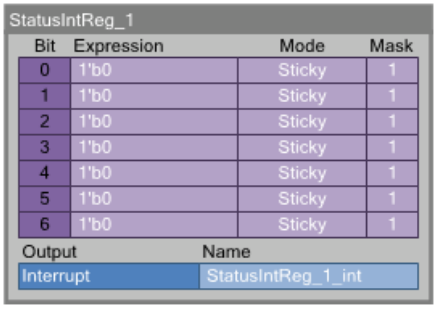
Double-click on the instance to open the Configure Status Register dialog.



The expression field of the Status Register bits allows several signals to go through PLD logic to form one signal before being placed in the register. This allows for some usage of PLDs without the use of State Machines. For example, two signals from the Datapath output called "lessComp0" and "equalComp0" may be combined together at the status expression field as "lessComp0 | equalComp0". Note that the expressions follow the Verilog syntax. The mode of the status bits can be set to either Transparent or Sticky. Status registers in the UDB Editor are always clocked at bus clock. Refer to the Status Register component datasheet (from the separate Status Register component in the PSoC Creator Component Catalog) and the *TRM* for more information.

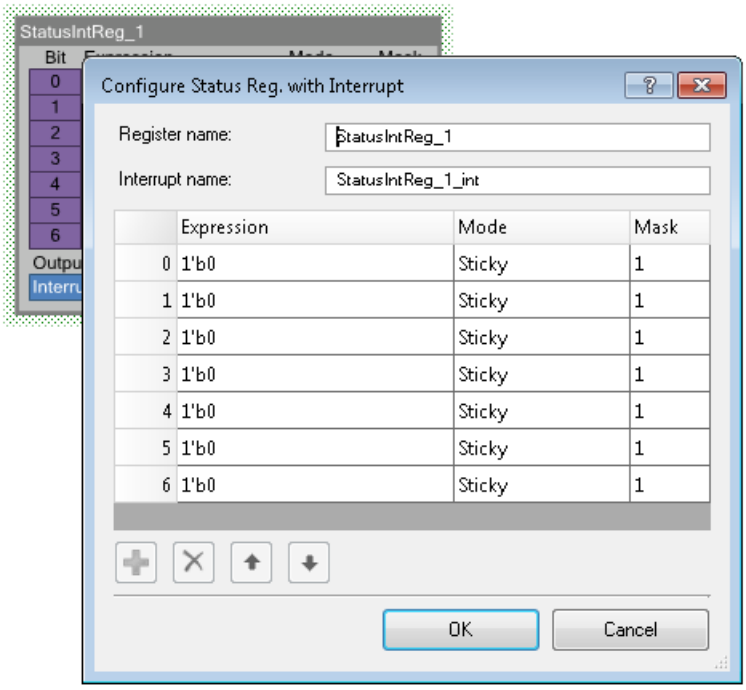
Status Interrupt Register

The following shows an instance of a Status Interrupt Register element in the design canvas of the UDB Editor. This element consumes the same resources as a Status Register.



Status Interrupt Registers are used to generate a maskable interrupt from the status bits. Seven bits are used as the inputs and one bit is used as the interrupt output.

Double-click on the instance to open the Configure Status Reg. with Interrupt dialog.

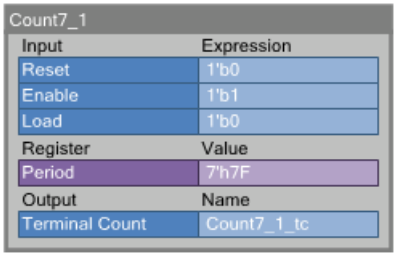


Like the Status Register, each bit has an expression field that can be used to form logic for that bit. The mode of the bits can be set to either Transparent or Sticky. The mask field determines whether that bit should be masked to generate the interrupt.

Note Only one interrupt per instance can be used when using the Status Interrupt Register. Refer to the Status Register component datasheet and the *TRM* for more information.

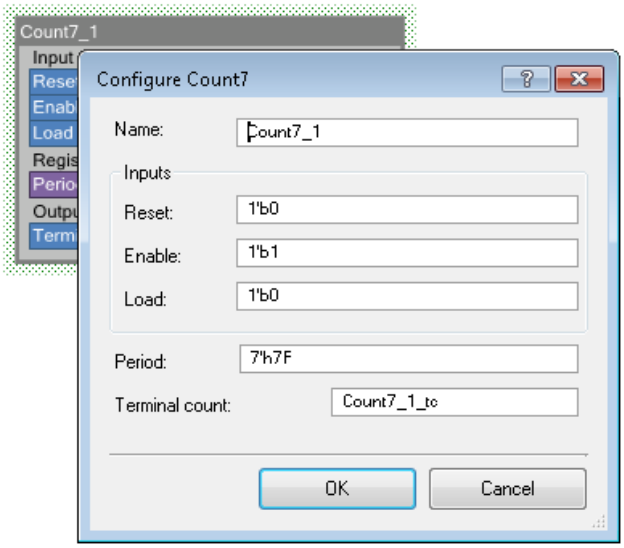
Count7 Counter

The following shows an instance of a Count7 counter element in the design canvas of the UDB Editor. This element consumes the same resources as a Control Register.



The Count7 counter is a 7-bit down counter that should be used when a counter of up to seven bits is needed. It can be more efficient to use PLD logic for counters of less than 3 bits. This gives savings compared to PLDs or Datapath-based counter designs.

Double-click on the instance to open the Configure Count7 dialog.

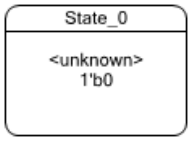


The Count7 counter has three inputs: Reset, Enable, and Load. These are used to reset the counter, enable the counter, and to load the counter with the period value during counter operation. The period value is set to 7'h7F by default. The counter has one output called terminal count, and this is driven high when the counter reaches 0. The counter is driven by the component clock. Refer to the Counter component datasheet and the *TRM* for more information.

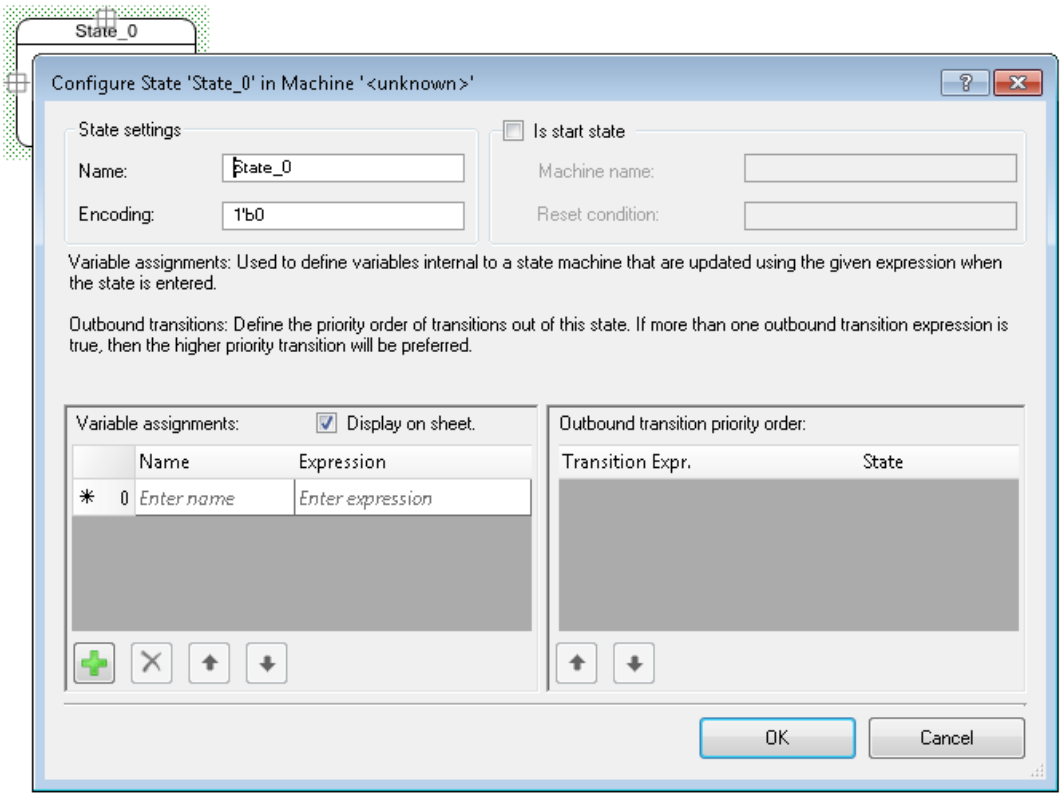
State Machine

A State Machine contains one or more states that implement control logic using PLDs. A State Machine is used to send control signals to the UDB elements in your design and to keep track of the operations happening in your hardware. Once a condition is achieved, the State Machine will transition to another state.

The following shows an instance of a State Machine element in the design canvas of the UDB Editor.



Double-click on the instance to open the Configure State dialog.



This dialog contains four sections: **State settings**, **Is start state** settings, **Variable assignments**, and **Outbound transition priority order**. Depending on the state operation and type, it may not be necessary to define all sections.

- State settings specify the **Name** of the state and its corresponding **Encoding** state value. Each state within the current State Machine must have unique Name and Encoding values. These should correspond to a specific operation performed by the target block(s). Therefore the generated control signals in a state should be unique.
- A state can be a starting state or some other state in the State Machine. Every State Machine requires a starting state, and there can only be one starting state with a State Machine.
  - Select the **Is start state** box to turn a State Machine instance into a start state. This changes the instance color from white to green and sets the encoding to zero (this is the reset state). The State Machine will be reset to this state when returning from sleep / hibernate.

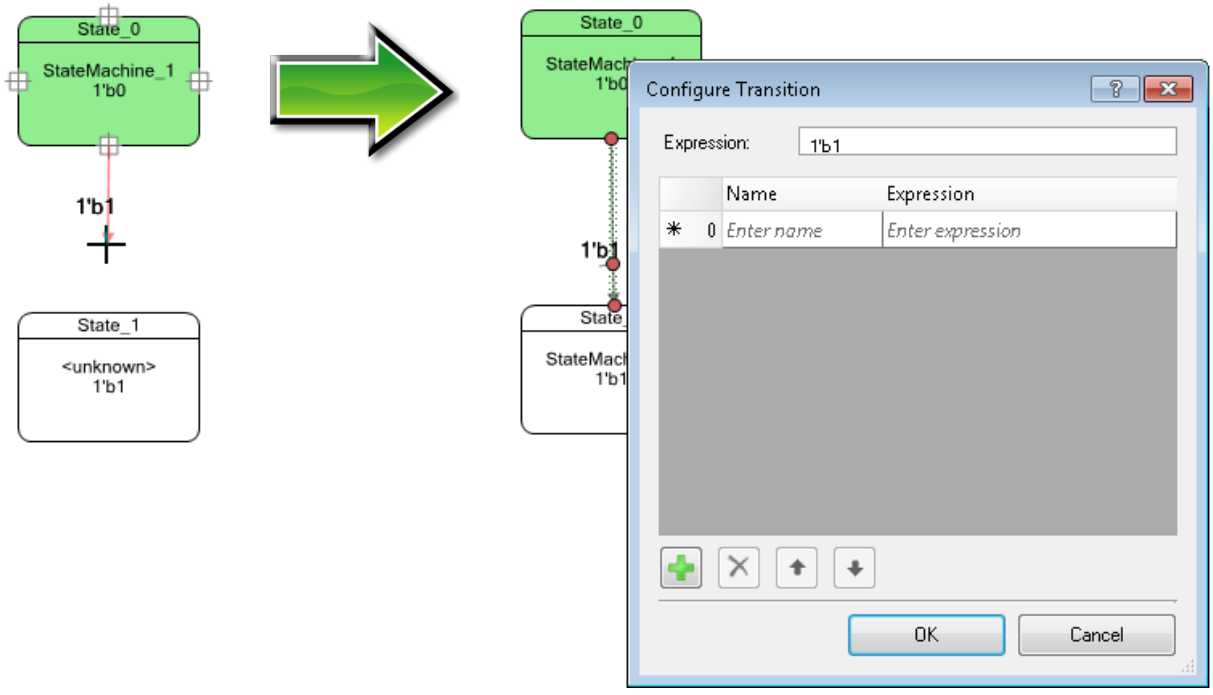
- Define a unique **Machine name** and its **Reset condition**. In most cases the reset would be triggered using a "not enable" or a separate "reset" input signal. The **Reset condition** allows all states in a State Machine to transition back to the start state upon reaching a reset condition.

- Variable assignments are used to define variables that are internal to the State Machine. They can then be used to control other elements, such as a Datapath or a Count7 counter in the UDB design.
  - A variable can be read from anywhere in the UDB Editor design, but it may only be *written* from within one State Machine.
  - Design-wide variables and outputs may not be *written* from within a State Machine, but they may derive their value from a State Machine variable.
- Outbound transition expressions can be used to set the priority of transitions in cases where more than one condition is true. This field will be populated as transitions are added to the state (see the State Machine Transitions section).
  - Moving a transition expression higher in the order gives it higher priority over those below.
  - However, it is generally a good practice to make transition expressions mutually exclusive.

State Machine Transitions

To add a transition from one State Machine instance to another:

1. Configure the first instance as a start state.
2. Hover the mouse on the edge of the start State Machine instance to show the anchor points.
3. Then, click the mouse and drag from one state to the edge of another.
4. Release the mouse button and the Configure Transition dialog opens.

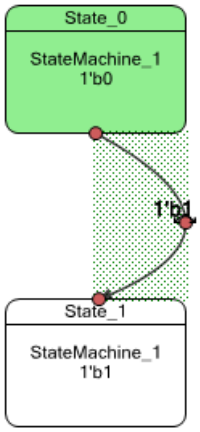


In the dialog, use the **Expression** field to set the transition expression. Whenever this expression is true, the

transition will occur in the State Machine. If more than one transition condition is true, then the higher priority transition will be followed. The Configure Transition dialog also permits assignments to be specified. These are similar to Variable assignments specified in the Configure State dialog, but they only take place when the specific transition is followed, rather than when the state is entered (via any transition).

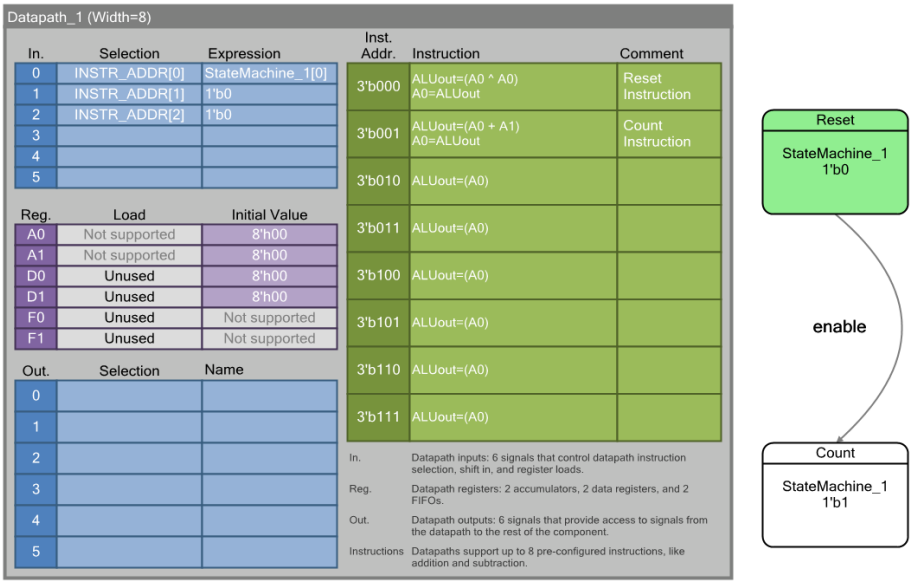
Note Linking a new state to an existing State Machine will cause propagation of the existing State Machine name. The encoding on the other hand must be unique and does not propagate.

To adjust the shape of the transition arc, click and drag the anchor points on the transition to the desired shape.



Controlling Datapath Instructions

State Machines are often used to control the INSTR_ADDR bits for the Datapath instructions. These are also referred to as the "dynamic configuration" bits in the *TRM*. They are accessed by mapping signals to the INSTR_ADDR bits in the inputs section of the Datapath. The following shows how a State Machine controls a Datapath.



Note D0, D1, F0, and F1 registers are not used in this design. Also, the LSB of the three available INSTR_ADDR

bits are assigned a constant value of '0'. The unassigned bits are then tied to 1'b0.

Two states exist in the State Machine. A start state named Reset, and a count state called Count. Reset has a unique encoding of 1'b0 and Count has a unique encoding of 1'b1. The Reset state selects the datapath instruction that clears the A0 register by XORing the A0 register with itself and writing the result back into the A0 register. When enable is set high, the State Machine will transition to Count, and the Datapath will start counting by adding the value stored in A1 to A0. If enable is low, then the State Machine will transition back to the Reset state, hence setting the Datapath to Reset.

The Datapath is configured to be an up counter that counts in multiples of whatever value is stored in the A1 register. A0 is used as the accumulator register used to hold the accumulated result. When the "enable" signal is low, the accumulator is emptied. When the "enable" signal is driven high, the counter starts counting. This is achieved using two Datapath instructions.

- Instruction 0: The first instruction is used as a reset. A0 XOR A0 is performed in the ALU to produce 0 at ALUout. This result is then stored in A0, effectively emptying the A0 register. Instruction 0 is chosen by assigning the INSTR_ADDR bits to 3'b000.
- Instruction 1: This instruction takes the value of A0 and A1 and adds them. The result is passed to ALUout, and this is fed into A0. Instruction 1 is chosen by assigning the INSTR_ADDR bits to 3'b001.

In order to assign the instructions to the Datapath, the State Machine must assign values to the INSTR_ADDR bits. The State Machine name itself is a signal that can be used as the control signal. Since in this example we only have two states with encoding 1'b0 and 1'b1, we assign the lsb (StateMachine_1[0]) to INSTR_ADDR[0]. The other INSTR_ADDR bits can be left as is since they are not used.

- When the "enable" signal is low, the State Machine remains in Reset and assigns StateMachine_1[0] = 1'b0. This places the Datapath in Instruction 0.
- When "enable" is driven high, the State Machine transitions to Count and assigns StateMachine_1[0] = 1'b1. The Datapath can now execute Instruction 1.
- If at some point "enable" is driven low, then the State Machine transitions back to Reset, and assigns StateMachine_1[0] = 1'b0. This transition is automatically triggered using the Reset Condition (in this case, "!enable"). The Datapath then executes Instruction 0.

UDB Editor APIs

When compiled, a component design using the UDB Editor will generate a header file named *$`INSTANCE_NAME`_defs.h*, where $`INSTANCE_NAME` is the instance name of your component. This file contains defines for Datapath register accesses and macros for performing Datapath FIFO configuration tasks.

Definitions for the Datapath registers can be used in software to read and write to these registers. Each definition will be named using the following syntax, where the first is the pointer to the register, and the second is the register:

```
<COMPONENT INSTANCE NAME>_<DATAPATH INSTANCE NAME>_<REGISTER>_PTR
<COMPONENT INSTANCE NAME>_<DATAPATH INSTANCE NAME>_<REGISTER>_REG
```

The pointer definitions can be used in the CY_GET_REGn and CY_SET_REGn macros (defined in *cytypes.h*), where n is the width of the Datapath with which they are associated. These macros allow data to be written to and retrieved from these registers.

Macros for Datapath FIFO configuration tasks include: clearing the Datapath FIFOs, setting the FIFO level mode, setting the FIFO to single-buffer mode, and returning the FIFO to normal mode. These macros are defined in the *<project>_defs.h* header file and noted below for the individual functions.

- **Clear DP FIFOs –** These macros empty the specified FIFOs in the design using the CPU. The FIFO returns to normal mode after this operation:

  ```
  <COMPONENT_INSTANCE>_<DATAPATH_INSTANCE>_<FIFO>_CLEAR
  ```

- **FIFO level mode –** These control the level at which the 4-byte FIFO asserts the bus status. Two modes can be set: NORMAL and MID. For more information on the FIFO levels, see FIFO Modes.

  ```
  <COMPONENT_INSTANCE>_<DATAPATH_INSTANCE>_<FIFO>_SET_LEVEL_NORMAL
  <COMPONENT_INSTANCE>_<DATAPATH_INSTANCE>_<FIFO>_SET_LEVEL_MID
  ```

- **FIFO single buffer mode –** These macros set the specified FIFO to single buffer mode. Single buffer mode allows the FIFO to act as a 1 word deep buffer instead of a 4-word deep FIFO. See FIFO Modes for more details.

  ```
  <COMPONENT_INSTANCE>_<DATAPATH_INSTANCE>_<FIFO>_SINGLE_BUFFER_SET
  ```

- **Return to normal mode –** These macros place the FIFOs to the normal 4-word deep FIFO configuration.

  ```
  <COMPONENT_INSTANCE>_<DATAPATH_INSTANCE>_<FIFO>_SINGLE_BUFFER_UNSET
  ```

The macros access an auxiliary control configuration register to set these configurations. An auxiliary control configuration register is a standard configuration register for the datapaths and FIFOs, not to be confused with the Control Register. Since multiple bits in an auxiliary control configuration register can be for different components, an interrupt safe implementation should be done using Critical Region APIs to avoid corruption. The following is an example for clearing FIFO 0 of a component instance named "MyComponent" with a Datapath instance named "MyDatapath".

```
uint8 interruptState;
/* Enter critical section */
interruptState = CyEnterCriticalSection();
/* Clears FIFO 0 */
MyComponent_MyDatapath_F0_CLEAR
/* Exit critical section */
CyExitCriticalSection(interruptState);
```

Example UDB Editor Design

This section provides instructions to create an example design using the UDB Editor. It does not describe the inner workings of the sub-blocks in the UDB Editor in detail. Nor does it describe how the sub-blocks fit together to form a functional design. For those details, refer to the UDB Editor Elements section.

Also, this example covers a portion of the process to create a component. For complete details about creating a component, refer to the PSoC Creator Component Author Guide. Refer also to the PSoC Creator Help as needed.

For this example, you will create a component that will:

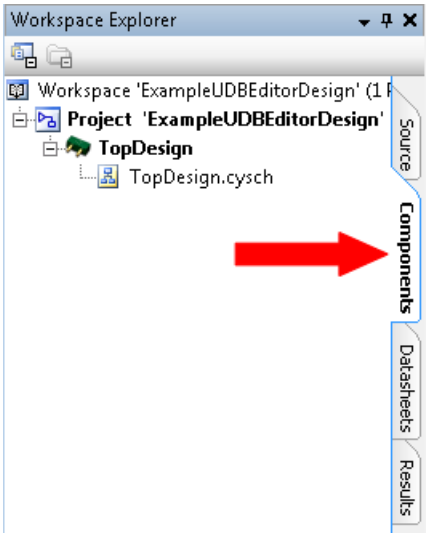
- Take in an 8-bit number (from the processor).
- Wait for a start signal from the hardware.
- Shift the input twice (aka, multiple by 4).
- Add 127.
- Write it back into a register (that the processor can read).
- Send an end signal to the hardware.

The basic design flow of the UDB Editor is:

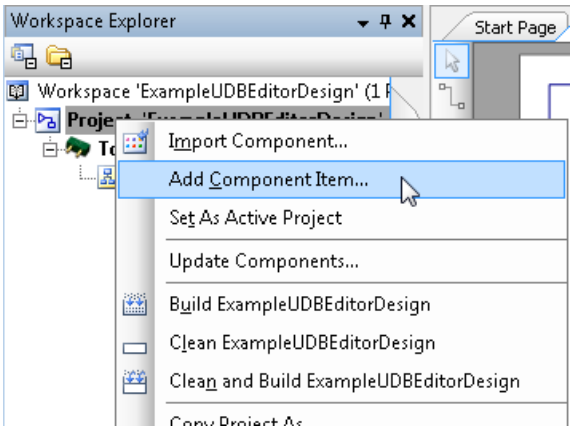
1. Create a custom component.
2. Define the component inputs and outputs.
3. Create a State Machine to control the Datapath inputs.
4. Configure the Datapath instructions (which will be triggered by the PLD).
5. Create the component symbol.
6. Build the component APIs.
7. Use the new component.

Step 1: Create a Custom Component

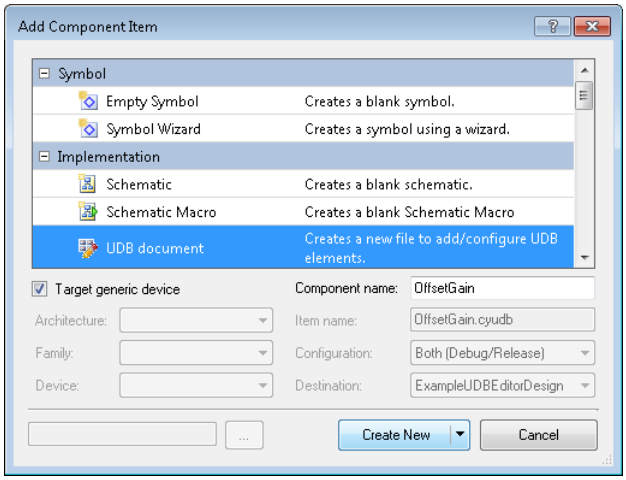
1. After opening or creating a project, click on the **Components** tab in the Workspace Explorer.



2. Right-click on a project, and select **Add Component Item…**



3. On the dialog, select the "UDB document template", type the **Component name** "OffsetGain" and click **Create New**.

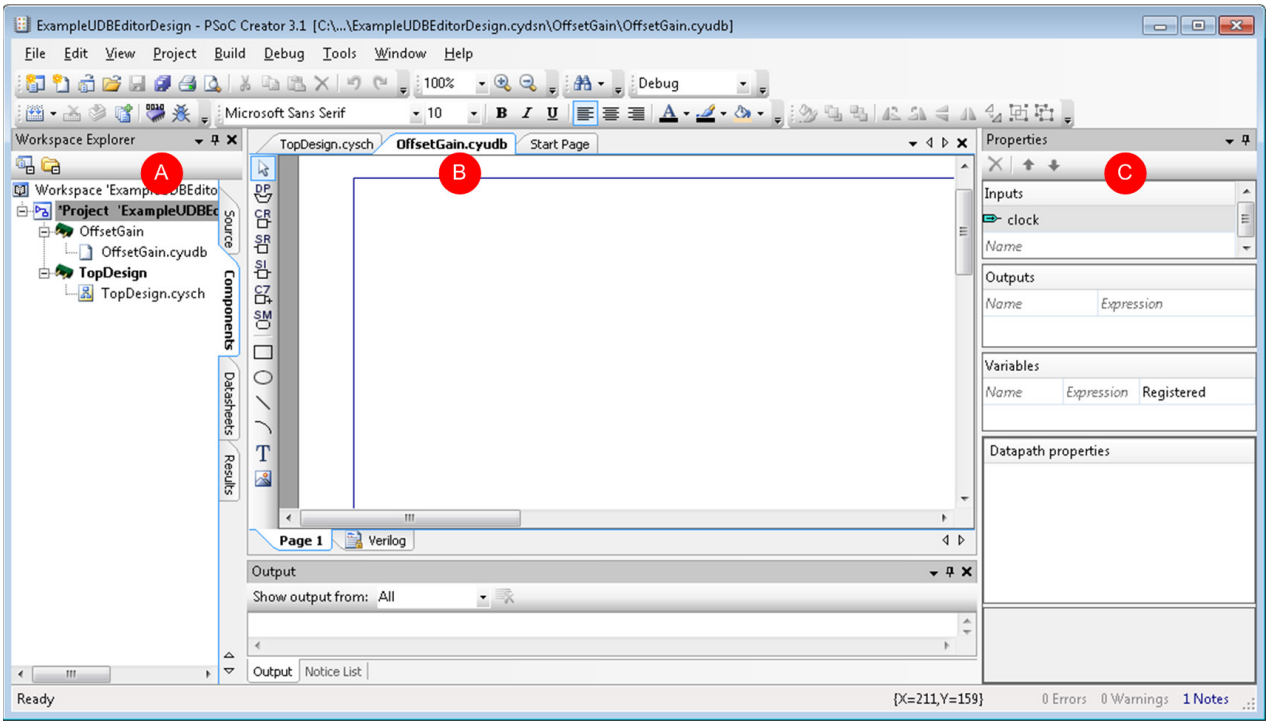


The UDB Editor opens in PSoC Creator, showing:

A. A new component called "OffsetGain" in the Components tab of the Workspace Explorer.

B. A new, empty, UDB Editor design canvas named *OffsetGain.cyudb*.

C. Under Properties, only a "Clock" input with no Outputs, no Variables, and no Datapath properties.



Note A clock terminal will always be present for a design implemented with a UDB Editor. Also, the names of all controls, inputs, outputs, variables, states, Datapaths, Control Registers, Status Registers, and Count7 counters must be unique across a UDB Editor document.

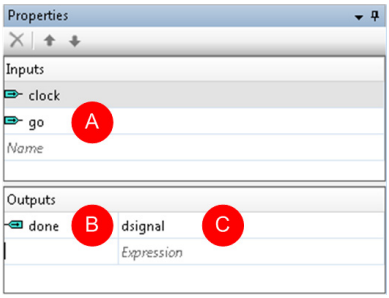
Step 2: Define the Component Inputs and Outputs

In the Properties window:

A. Add the "go" input signal which will create an input terminal.

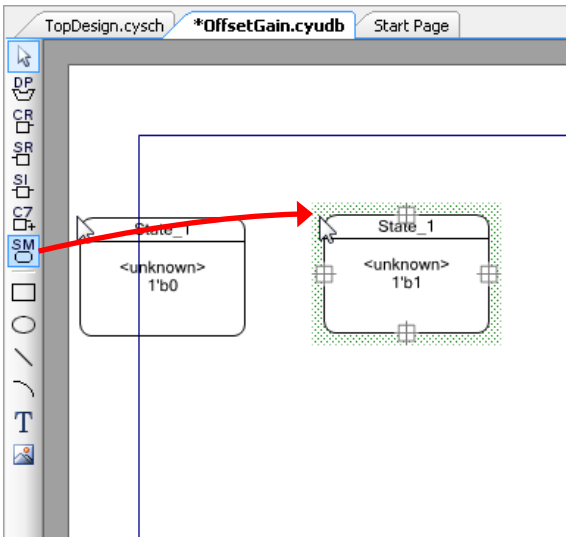
B. Add the "done" output signal which will create an output terminal.

C. Assign the "done" output signal to a Verilog variable called "dsignal." This will be setup in the PLD in a future step.



Step 3: Create a State Machine to Control the Datapath Inputs

1. To begin creating a State Machine, drag a state (SM) instance from the Design Elements Palette onto your design canvas.

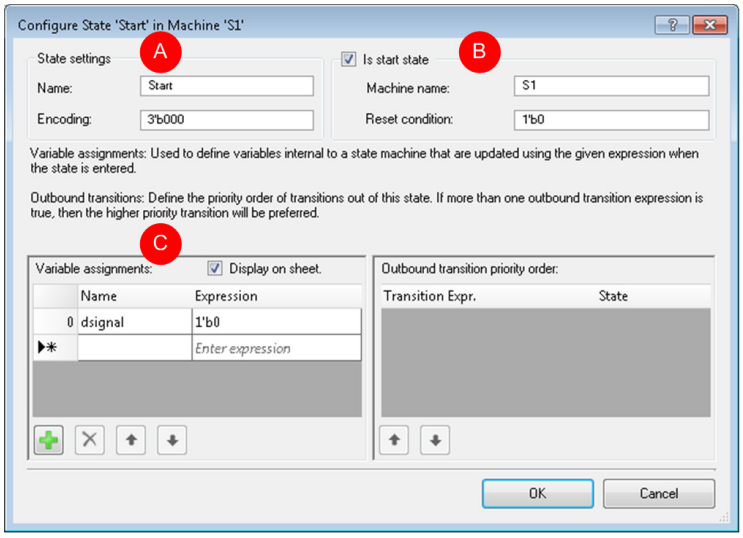


2. Double-click on the on the state instance to open the Configure State dialog. Configure this instance as a start state:

   A. Under **State settings**, enter the **Name** as "Start" and the **Encoding** value for the state as "3'b000."

   B. Select the **Is start state** check box, and then enter the **Machine name** as "S1" and the **Reset condition** as "1'b0."

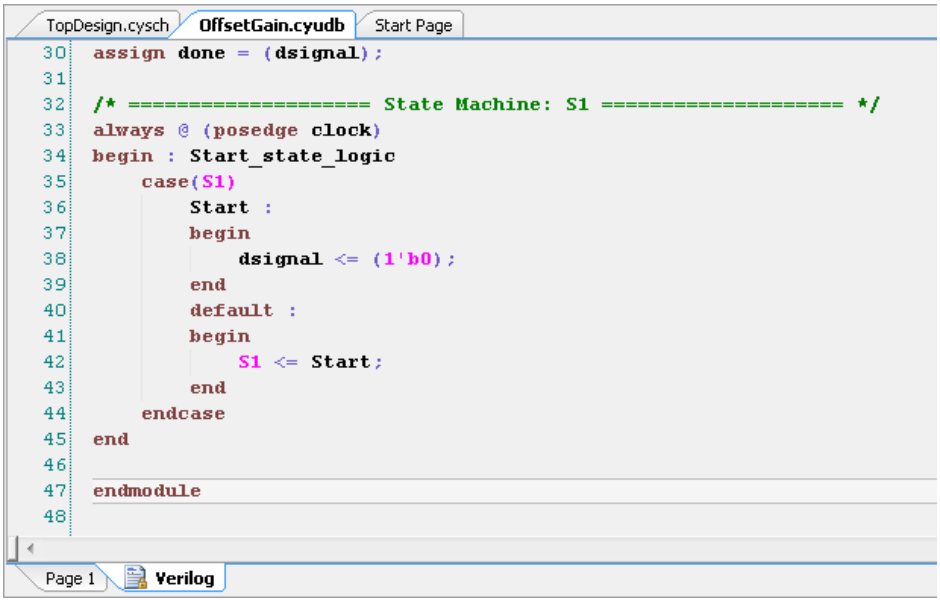
   C. Under Variable assignments, type a variable named "dsignal" and set its value to '1'b0."



Note You are allowed to use other legal Verilog combinational logic statements, such as A&B (if A and B were variables in your design).

When you see "Expression" in the UDB Editor, you are able to type a legal Verilog expression, which will then be embedded into your PLD State Machine.

You can see the automatically generated Verilog (which is used by PSoC Creator to create the component) by clicking on the **Verilog** tab at the bottom of the *OffsetGain.cyudb* schematic.

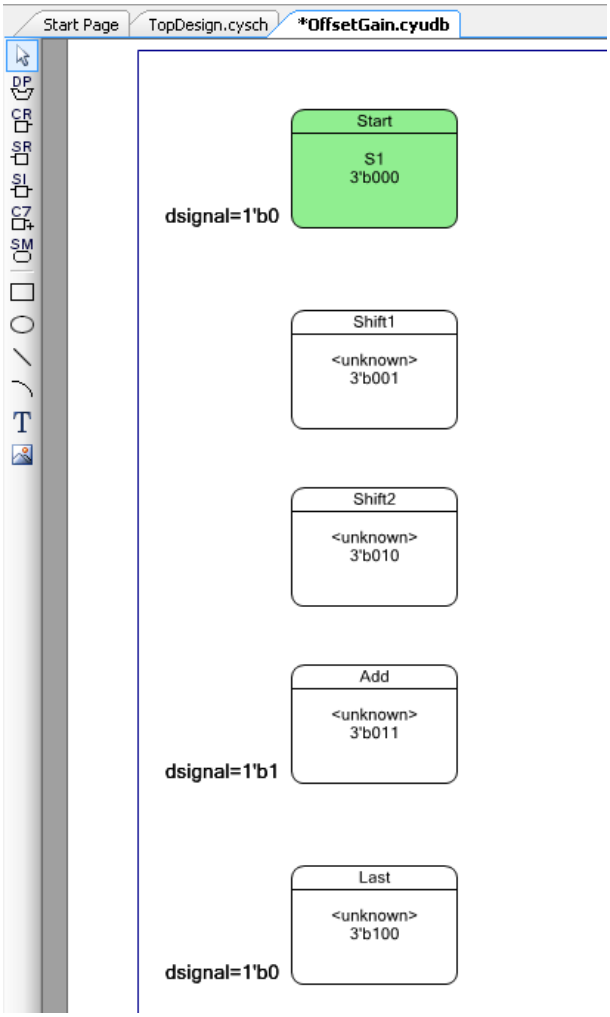


This Verilog code is "read only" so you can only modify it by updating the UDB Editor design. However, you can copy the Verilog code into your own manually-built custom component. That process is beyond the scope of this document. Refer to Application Note AN82156 for information about building a Verilog component.

3. Add more state instances to the design canvas. Configure the additional states as follows:

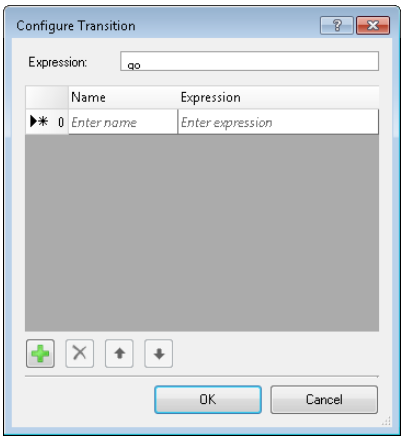
Name	Encoding	Variable Assignment	Comment
Start	3'b000		Starts the State Machine.
Shift1	3'b001		Shift the input once.
Shift2	3'b010		Shift the input again.
Add	3'b011	dsignal = 1'b1	Set the "dsignal" to 1'b1 which will make the "done" output 1 at the beginning of the next clock signal.
Last	3'b100	dsignal = 1'b0	Set the "dsignal" to 1'b0 which will make the "done" output 0 at the beginning of the next clock signal.

At this point, your State Machine should look similar to this:



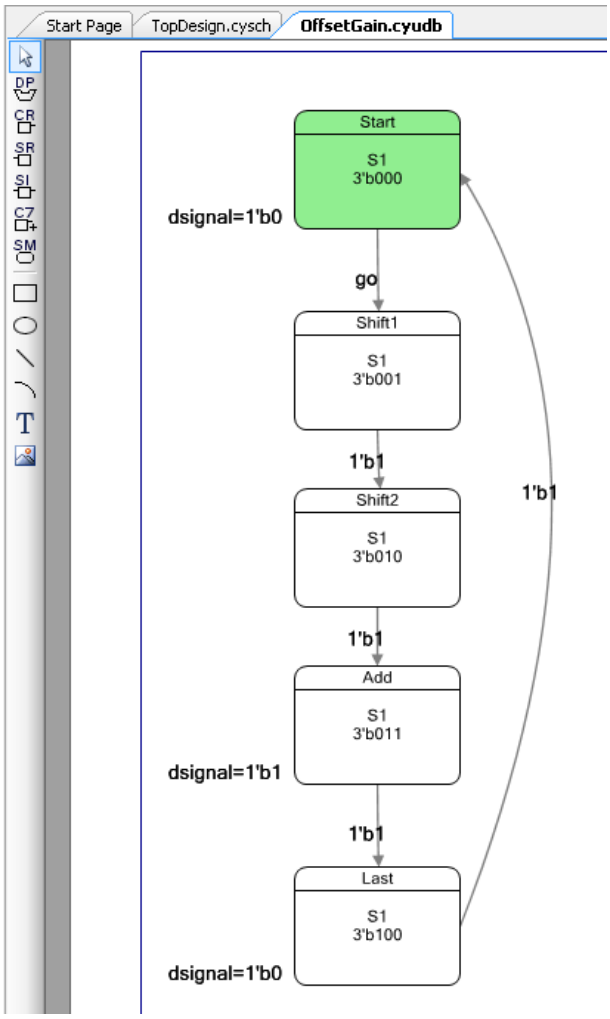
Note The "dsignal= … " labels were move to make room for the state transition arcs.

4. Create a transition from the "Start" state to the "Shift1" state. See State Machine Transitions. In this case you want to move from "start" to "Shift1" when the signal "go" becomes true. On the Configure Transition dialog, type "go" in **Expression** and click **OK**.



5. Add transitions to the rest of the states. For all of them, leave **Expression** as "1'b1." These states will always transition because 1'b1 is always true.

   Now your State Machine should look similar to this:



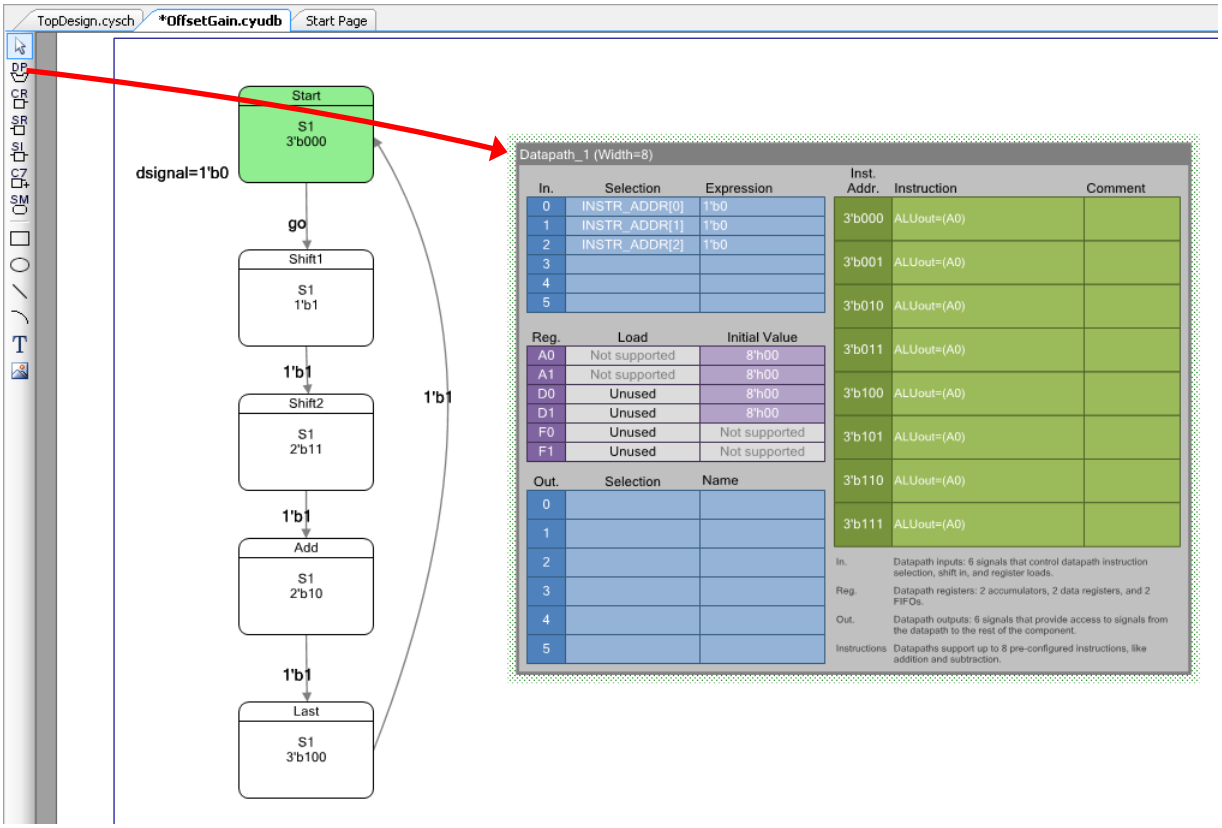
This State Machine will yield the following cycle-by-cycle results.

Clock Cycle	go	done	State	S1[0..2]
1	0	x	Start	3'b000
2	0	0	Start	3'b000
3	1	0	Start	3'b000
4	0	0	Shift1	3'b001
5	0	0	Shift2	3'b010
6	0	0	Add	3'b011
7	0	1	Last	3'b100
8	0	0	Start	3'b000

Step 4: Configure the Datapath

This step adds the Datapath element to the design. The Datapath will perform the "shift" and "add" instructions. For more information, see Datapath.

1. Drag a Datapath (DP) instance from the Design Elements Palette onto your design canvas. By default, the instance will be named "**Datapath_1**". This name is used later in the API generation.

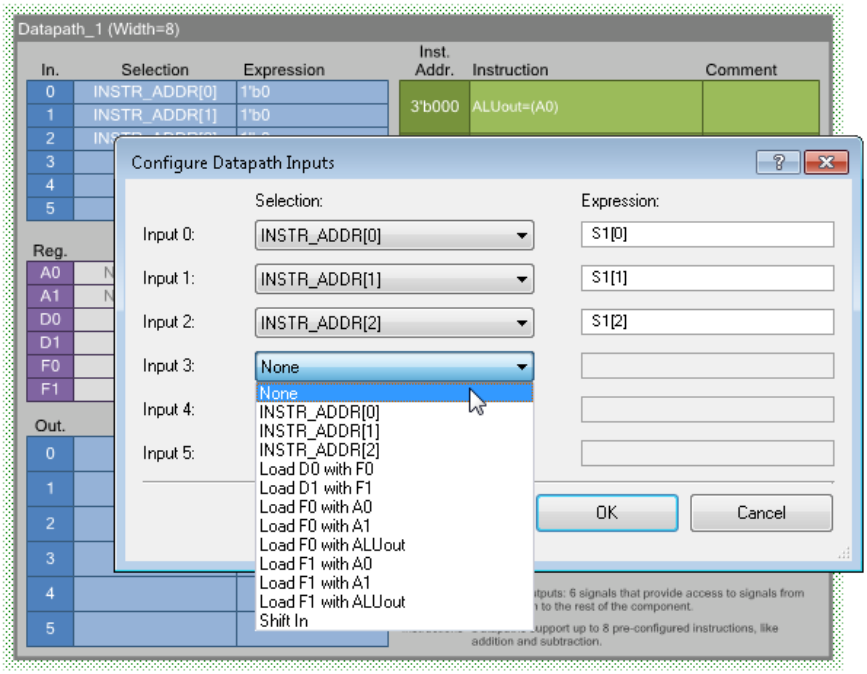


2. Double-click on the Inputs section (left top) in the Datapath instance to open the Configure Datapath Inputs dialog. Configure the inputs to connect to the State Machine.

   There are 6 inputs to the Datapath that select the function of the block. In general, three of the inputs are used to select which ALU operation occurs. In this case, assign the "INSTR_ADDR[x]" (which is the ALU command selection) to S1[x] (which is the S1 State Machine that is defined in the PLD).

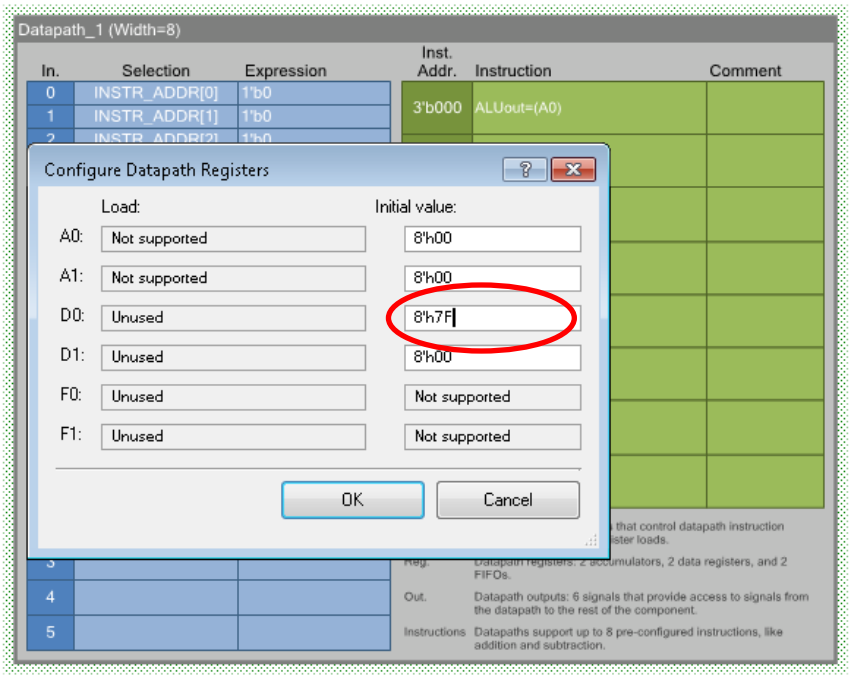
   You can write many different Verilog expressions in the **Expression** field, including the constants (1'b0, 1'b1), combinational logic expressions (A&B), or input signals from the component ("go").

   You can have the Datapath do other selectable operations including loading the D0/D1 data registers, FIFOs, etc. In this example, the FIFO will be loaded with the output of the ALU on every clock cycle. Set **Input 3 - 5** to "None" for this example.



   Click **OK** to close the dialog.

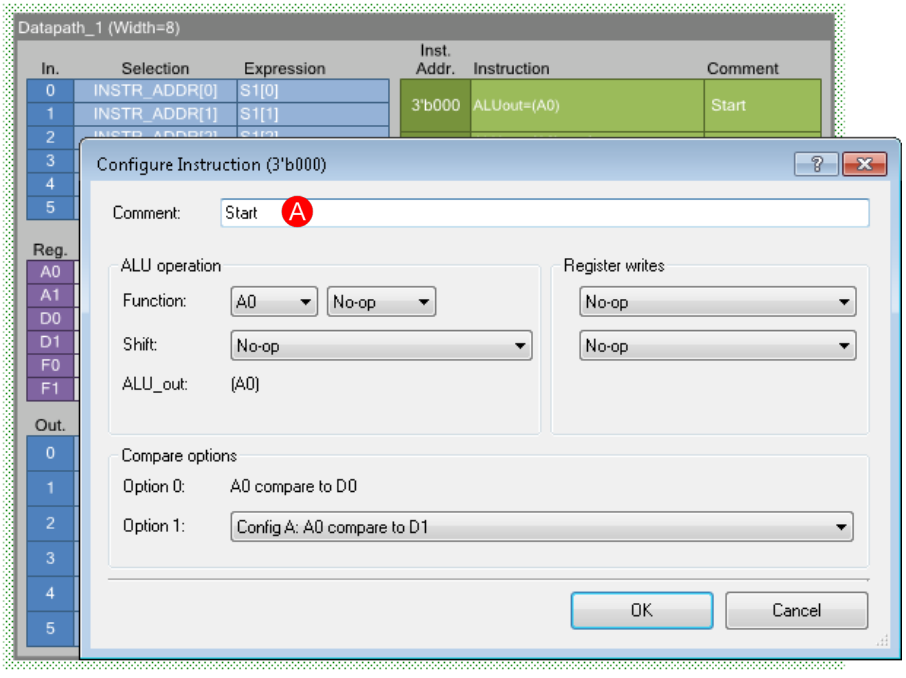
3. Double-click on the Registers section (left middle) in the Datapath instance to open the Configure Datapath Registers dialog. Configure the D0 register to default to 8'h7F for the **Initial value**.



Click **OK** to close the dialog.

4. On the right side of the Datapath instance, in the Instructions section (green), double-click on the 3'b000 row (1st row) to open the Configure Instruction dialog.

   A. Type "Start" in the **Comment** field.



Click **OK** to close the dialog.

5. Double-click on the 3'b001 row (2nd row) to open the Configure Instruction dialog.

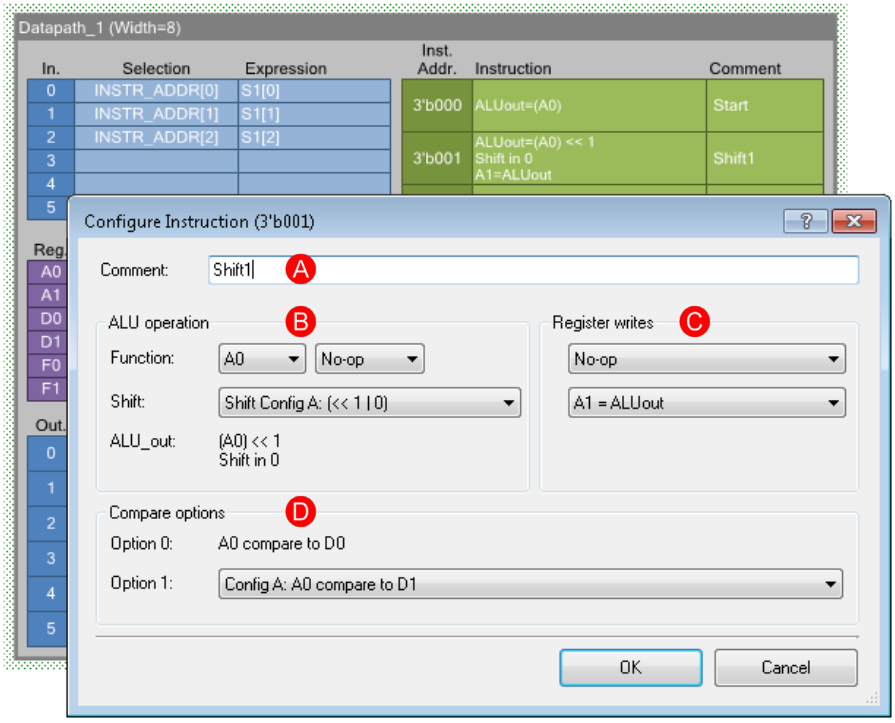
   Configure the ALU to perform the "Shift" operation for the Shift1 state as follows:

   A. Type "Shift1" in the **Comment** field.

   B. Under **ALU operation**, leave the **Function** as "A0" and "No-op". For **Shift**, select "Shift Config A: (<<1|0)," which will shift the output of the ALU one bit to the left (and insert a 0 as the LSB).

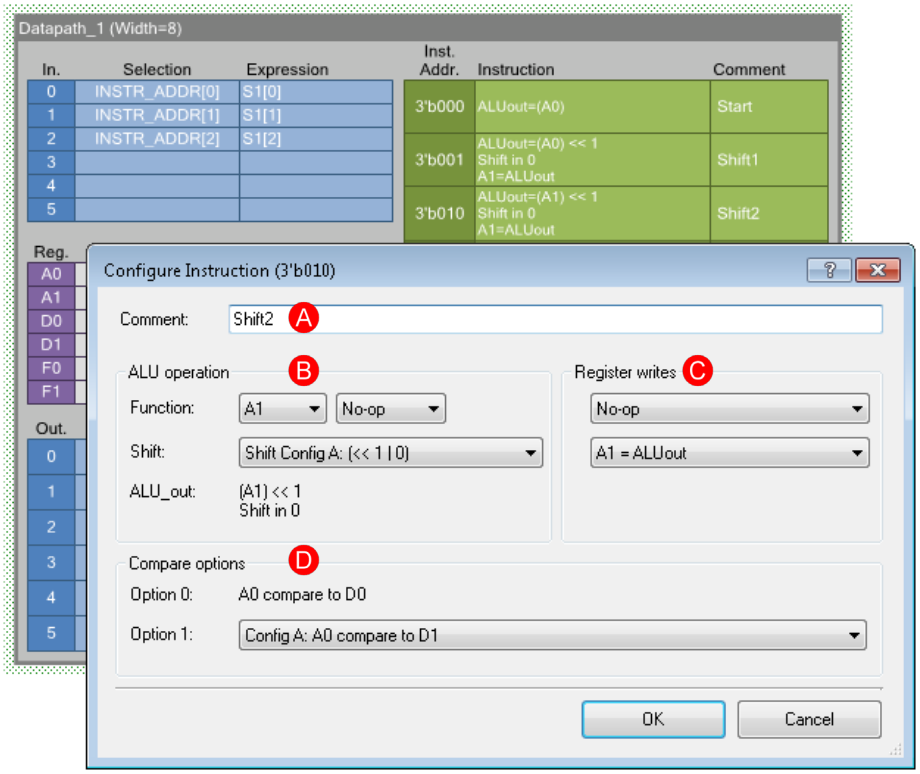
   C. Under **Register writes**, leave the top selection as "No-op" and select "A1 = ALUout" for the second selection.

   D. Under **Compare options**, leave **Option 1** as "Config A: A0 compare to D1."

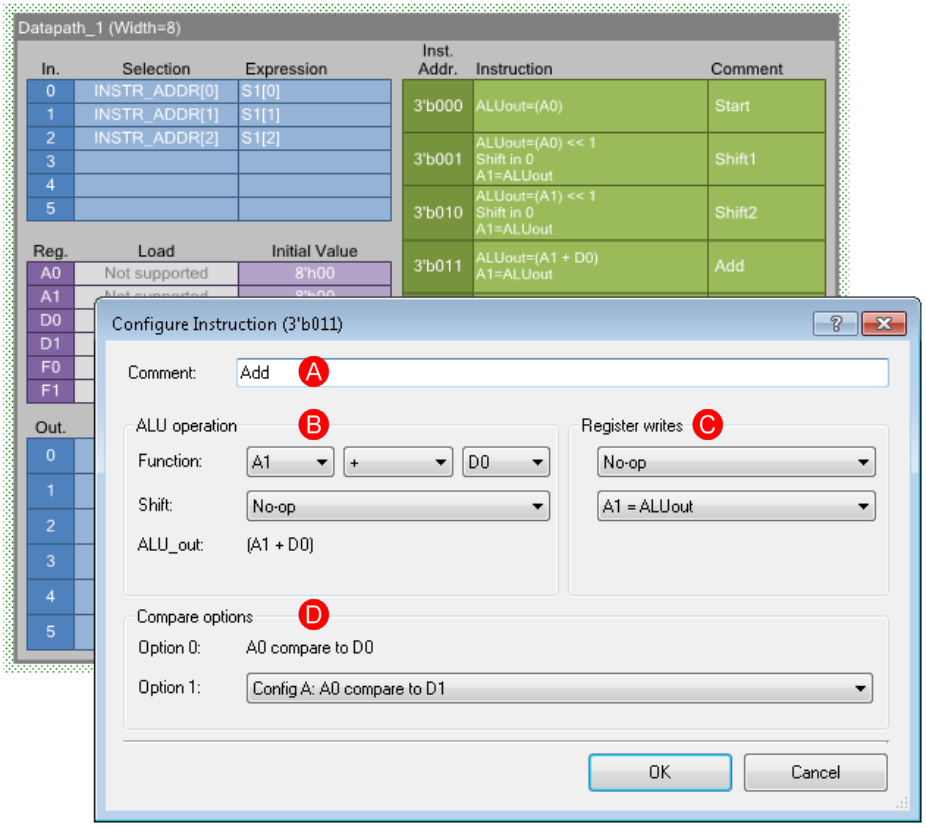


   Click **OK** to close the dialog.

6. Double-click on the 3'b010 row (3rd row) to open the Configure Instruction dialog.
   A. Type "Shift2" in the **Comment** field.
   B. Under **ALU operation**, select the **Function** "A1" and "No-op". For **Shift**, select "Shift Config A: (<<1|0)," which will shift the output of the ALU one bit to the left (and insert a 0 as the LSB).
   C. Under **Register writes**, leave the top selection as "No-op" and select "A1 = ALUout" for the second selection.
   D. Under **Compare options**, leave **Option 1** as "Config A: A0 compare to D1."



7. Double-click on the 3'b010 row (3rd row) to open the Configure Instruction dialog.
   A. Type "Add" in the **Comment** field.
   B. Under **ALU operation**, select the **Function** "A1," "+," and "D0". For **Shift**, leave as "No-op."
   C. Under **Register writes**, leave the top selection as "No-op" and select "A1 = ALUout" for the second selection.
   D. Under **Compare options**, leave **Option 1** as "Config A: A0 compare to D1."

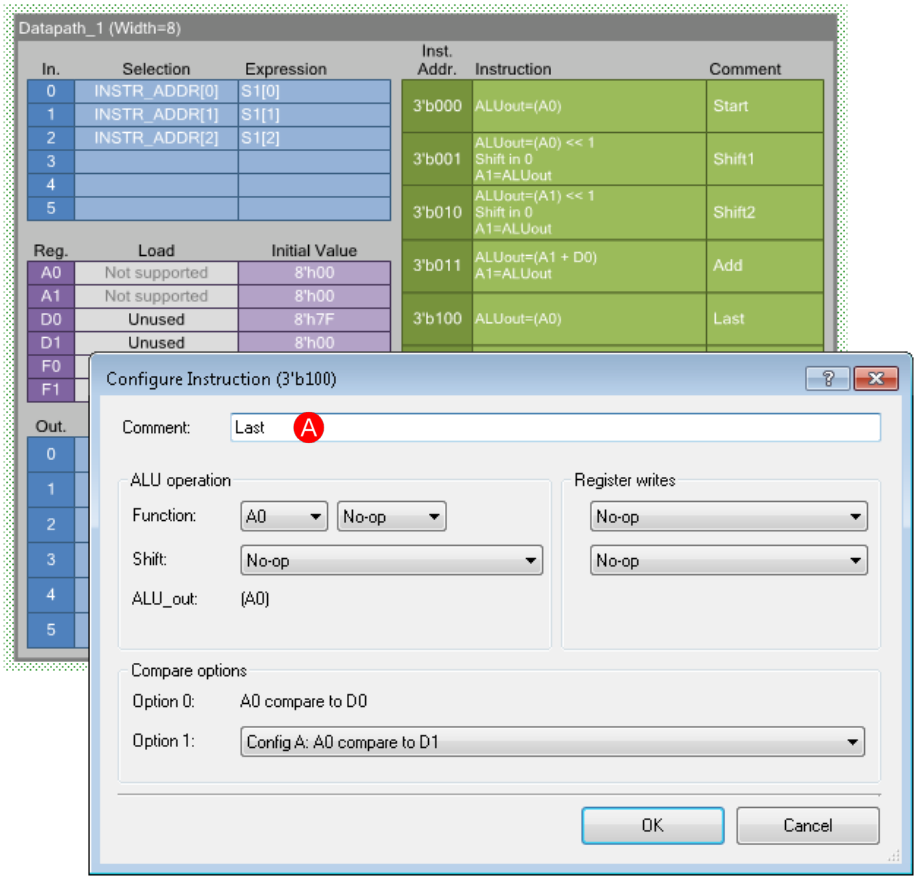


Click **OK** to close the dialog.

8. Double-click on the 3'b100 row (5th row) to open the Configure Instruction dialog.

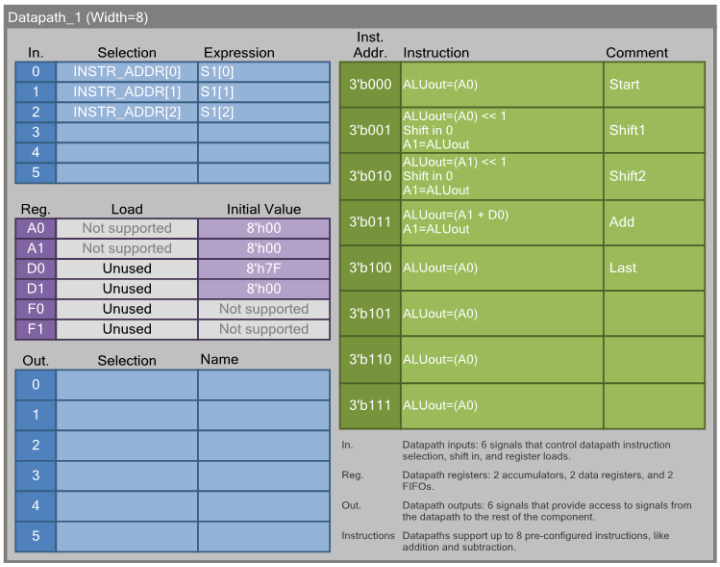
   Configure the ALU to perform the "Shift" operation for the Shift1 state as follows:

   A. Type "Last" in the **Comment** field.



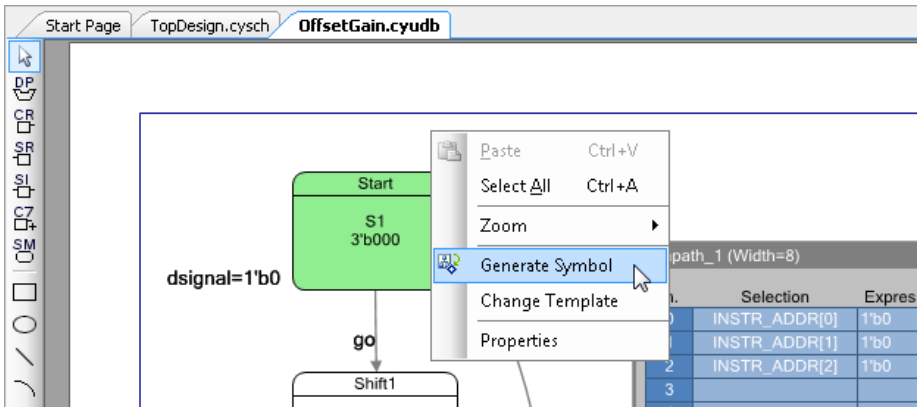
   Click **OK** to close the dialog.

   Your Datapath instance should now look like this:

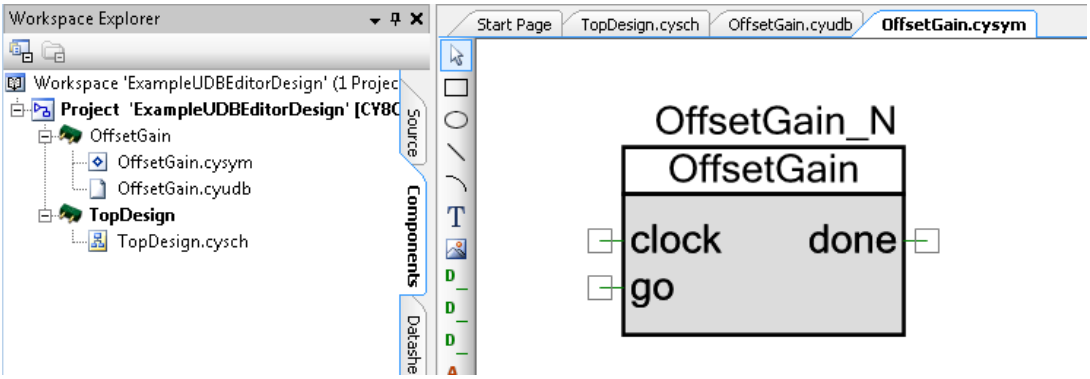


Step 5: Create the Component Symbol

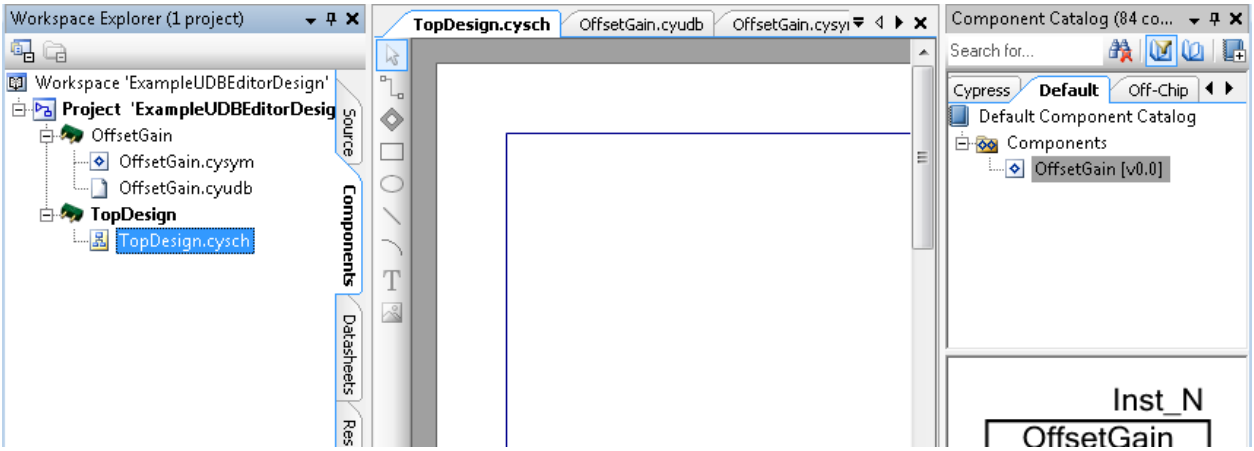
Right-click in an empty area of the design canvas, and select **Generate Symbol**.



This action adds a new symbol file (.cysym) to the component (shown in the Workspace Explorer) and opens it in the Symbol Editor.



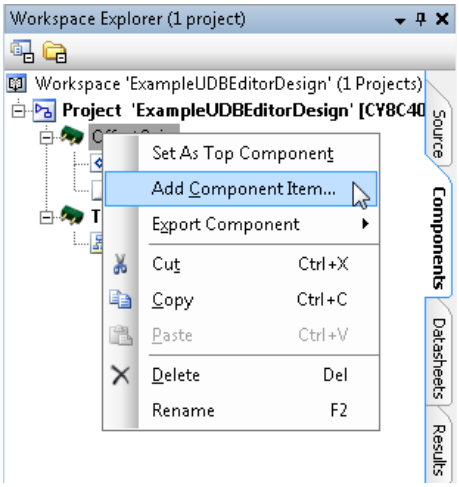
If you double-click the *TopDesign.cysch* file in the Workspace Explorer to open the Schematic Editor, you will see a **Default** tab in the Component Catalog that contains the new component.



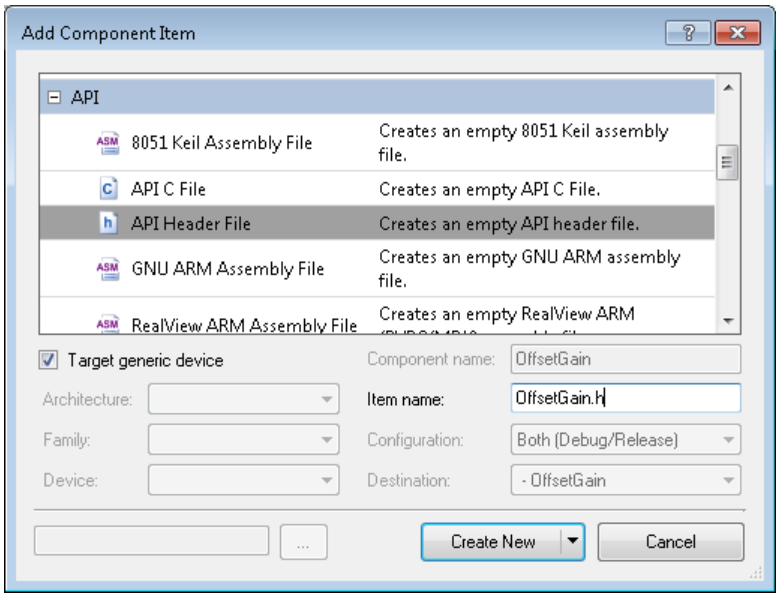
Step 6: Build the Component APIs

For this step, add API header and c files to provide user functions for the component.

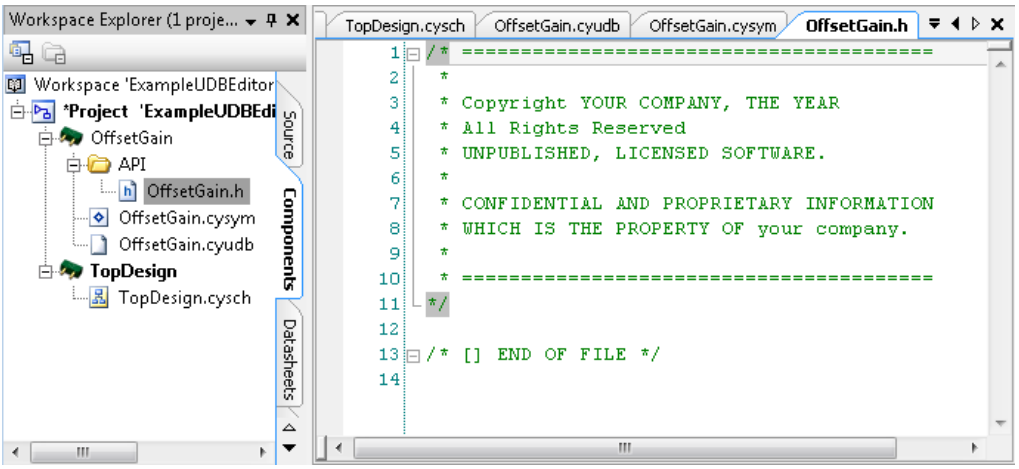
1. In the Workspace Explorer, under the **Components** tab, right-click on the component and **Add Component Item**.



2. On the dialog, scroll down to the API section and select the "API Header File" template. For **Item name**, type "OffsetGain.h."



Click **Create New**. This action adds a new header file to the component (shown in the Workspace Explorer) and opens it in the Code Editor.



3. Edit the header file as follows to create "SetInput," "GetInput," and "GetOutput" functions:

```
#if !defined (`$INSTANCE_NAME`_OFFSETGAIN_H)
    #define `$INSTANCE_NAME`_OFFSETGAIN_H

#include "cytypes.h"

void `$INSTANCE_NAME`_SetInput(uint8);
uint8 `$INSTANCE_NAME`_GetInput();
uint8 `$INSTANCE_NAME`_GetOutput(void);

#endif
```

4. Now add an API C file to the component with the name "*OffsetGain.c*." Edit the C file as follows:

```
#include "cytypes.h"
#include "`$INSTANCE_NAME`_OffsetGain.h"
#include "`$INSTANCE_NAME`_defs.h"

void `$INSTANCE_NAME`_SetInput(uint8 val)
{
    /* Datapath_1 is the instance name from the UDB Editor */
    CY_SET_REG8(`$INSTANCE_NAME`_Datapath_1_A0_PTR,val);
}

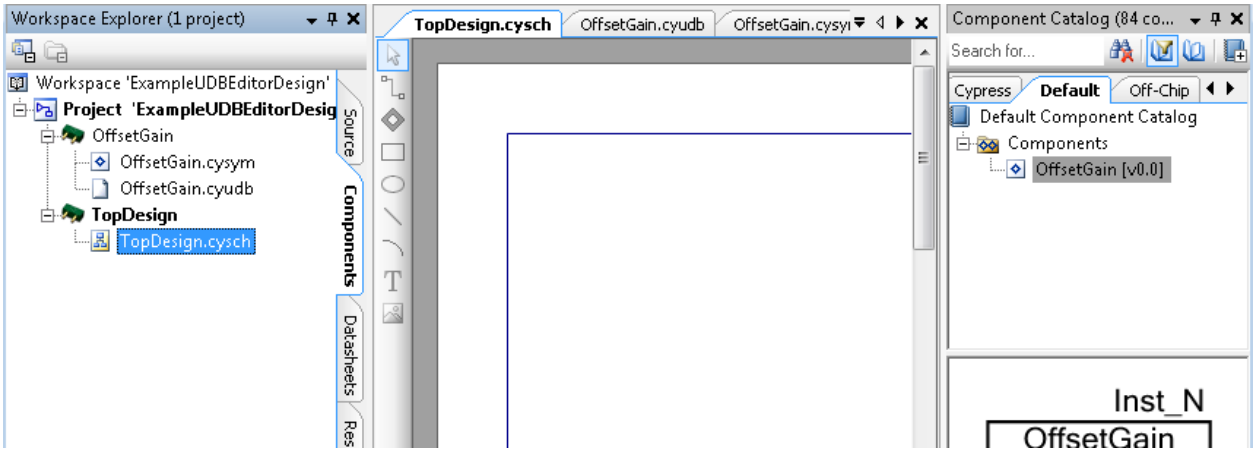
uint8 `$INSTANCE_NAME`_GetInput()
{
    return CY_GET_REG8(`$INSTANCE_NAME`_Datapath_1_A0_PTR);
}

uint8 `$INSTANCE_NAME`_GetOutput(void)
{
    return CY_GET_REG8(`$INSTANCE_NAME`_Datapath_1_A1_PTR);
}
```

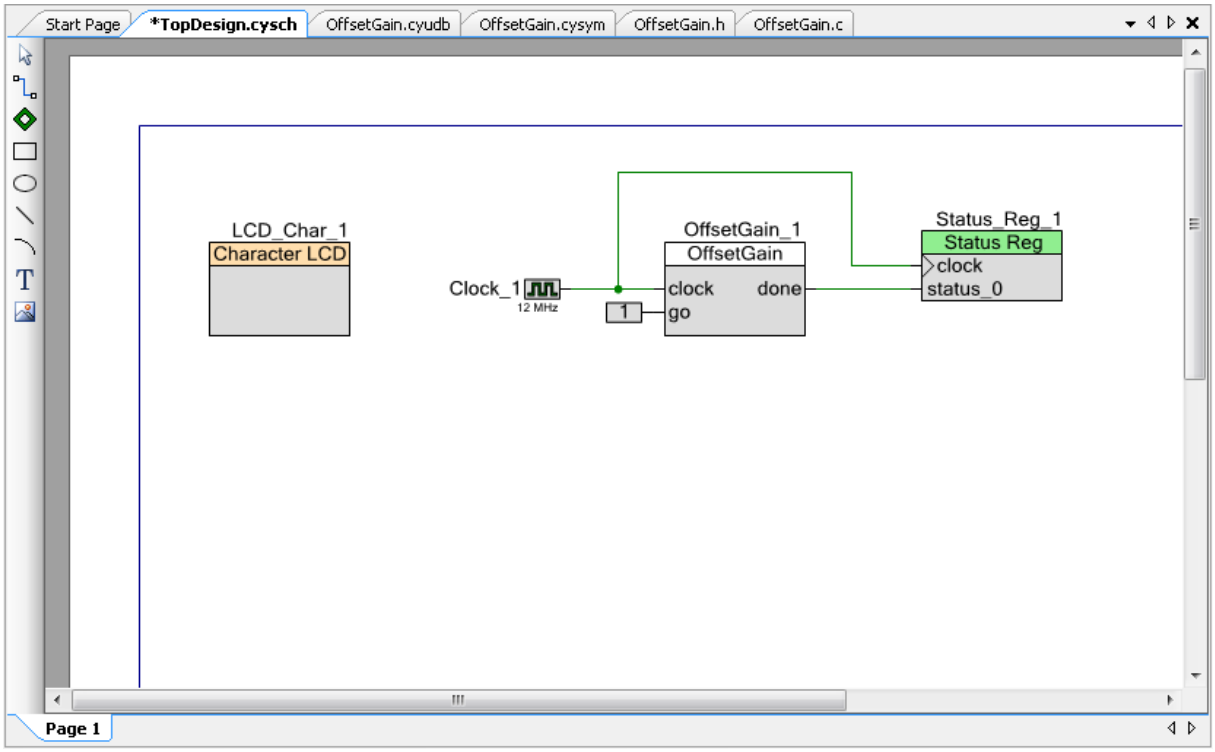
Step 7: Use the New Component

Now that the component is complete, we'll use it in a design.

1. Double-click the *TopDesign.cysch* file in the Workspace Explorer to open the Schematic Editor; you will see a **Default** tab in the Component Catalog that contains the new component.

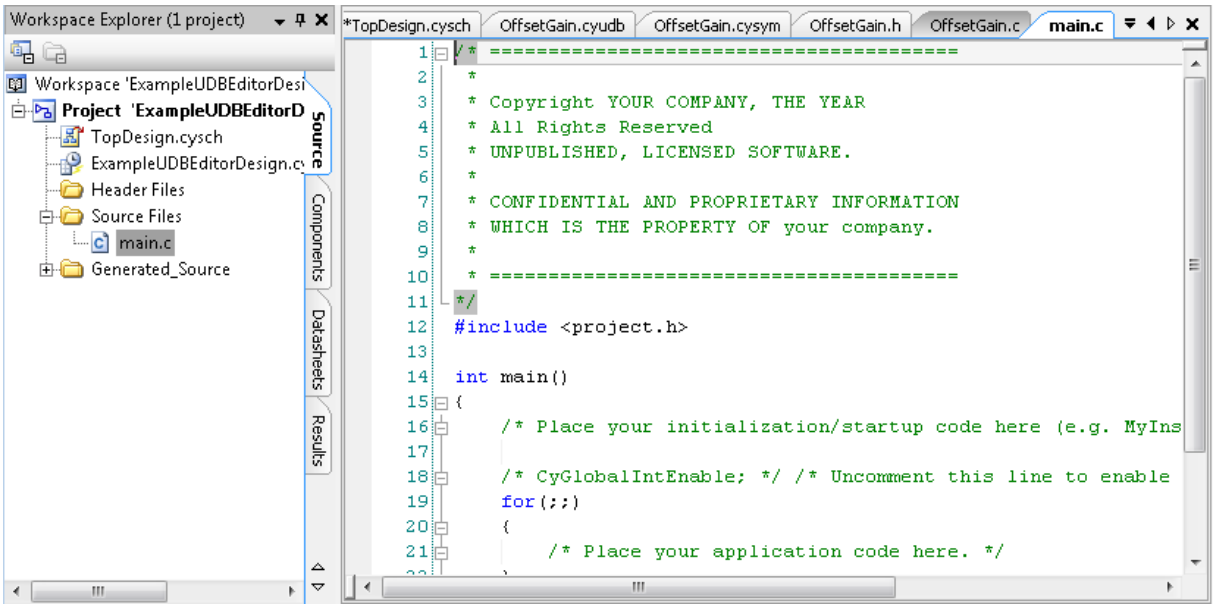


2. Drag the OffsetGain component onto the *TopDesign.cysch* canvas. Connect a Clock component to the "clock" terminal, a Logic High component to the "go" terminal, and a status register with a sticky bit to the "done" terminal. Also, drag a Character LCD to the canvas. When complete, your design should look similar to this:



In this design, we will write a value into the component, and then read it out and display it on the LCD. The UDB component will require an input clock. The OffsetGain component will require the "go" signal, in this case being set to logic 1, that will run and then re-run the component. The status register will check the status of the "done" signal.

3. In the Workspace Explorer, click the **Source** tab, and then double-click the *main.c* file to open it in the Code Editor.



Edit the *main.c* file as follows. Make sure that names used for the various functions match the instance names of the components on the canvas.

Use the **Generate Application** command on the toolbar to generate the source code so that the PSoC Creator Code Editor will help you with the Code Editor auto-complete feature.

```
#include <project.h>

int main()
{
    LCD_Char_1_Start();

    OffsetGain_1_SetInput(10);

    while(Status_Reg_1_Read() == 0); // wait for UDB component to be done

    LCD_Char_1_Position(0,0);
    LCD_Char_1_PrintString("I:");
    LCD_Char_1_PrintInt8(OffsetGain_1_GetOutput());
    LCD_Char_1_PrintString(" O:");
    while(1);
}
```

4. Connect the appropriate kit for your device and click **Program**. When complete, you should see "I:0A O:A7" displayed on the LCD.

Additional projects

If you successfully built and programmed this project, try a few other options to expand the design.

- Put the State Machine states to pins.
- Use the FIFOs.
- Make the "gain" of the block be programmable.
- Implement a PWM.

Appendix A: Datapath Operation

A UDB-based PSoC device Datapath is essentially a very small 8-bit wide processor with 8 states defined in a "dynamic configuration." Consecutive Datapaths can be tied together to operate on wider data widths using one of the following pre-defined modules. The following description provides a high-level description of the Datapath and how it is used. For full details on the Datapath, refer to the *TRM*.

Datapath Instructions

The Datapath is broken into the following sections:

- ALU – An ALU is capable of the following operations on 8-bit data. When multiple Datapaths are tied together to form 16, 24, and 32 bits, then the operations act on the full datawidth.
  - Pass-Through
  - Increment (INC) (add one)
  - Decrement (DEC) (subtract one)
  - Add (ADD)
  - Subtract (SUB)
  - XOR
  - AND
  - OR
- Shift – The output of the ALU is passed to the shift operator, which is capable of the following operations. When multiple Datapaths are tied together to form 16, 24, and 32 bits, then the operations act on the full datawidth.
  - Pass-Through (no-op)
  - Shift Left
  - Shift Right
- Mask – The output of the shift operator is passed to a mask operator, which is capable of masking off any of the 8 bits of the Datapath. The mask is ANDed with the output of the shifter and cannot be disabled on an instruction-by-instruction basis.
- Registers – The Datapath has the following registers available to the hardware and to the CPU with various configuration options defined in the static configuration registers.
  - Two Accumulator Registers: A0 and A1
  - Two Data Registers: D0 and D1
  - Two 4-byte deep FIFOs: F0 and F1 capable of multiple modes of operation
- Comparison Operators

- Zero Detection: Z0 and Z1 which compare A0 and A1 to zero respectively and output the binary true/false to the interconnect logic for use by the hardware as necessary.
- FF Detection: FF0 and FF1 which check whether A0 or A1, respectively, contain all 1's and output the binary true/false to the interconnect logic for use by the hardware as necessary.
- Compare 0:

  Compare equal (ce0) – Compare (A0 & Cmask0) is equal to D0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask0 is configurable in the static configuration.)

  Compare Less Than (cl0) – Compare (A0 & Cmask0) is less than D0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask0 is configurable in the static configuration.)
- Compare 1:

  Compare equal (ce1) – Compare ((A0 or A1) & Cmask1) is equal to (D1 or A0) and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask1 is configurable in the static configuration)

  Compare Less Than (cl1) – Compare (A0 & Cmask0) is less than D0 and output the binary true/false to the interconnect logic for use by the hardware as necessary. (Cmask1 is configurable in the static configuration)
- Overflow Detection: Indicates the msb has overflowed by driving ov_msb output as a binary true/false to the interconnect logic for use by the hardware as necessary.

The Datapath allows for many different configurations that are common in almost every component that will be designed. Many functions within a Datapath that can be implemented with Verilog fit into the PLDs. However, the PLDs will be used up very quickly, whereas the Datapath is a fixed block. There will always be a trade-off between the number of Datapaths and PLDs available. It is up to the designer to decide which of these is a more precious resource. Note that some functions, such as FIFOs, cannot be implemented in the PLDs.

Datapath Registers

Each Datapath contains 6 registers - A0, A1, D0, D1, F0 and F1. These serve different functions with certain restrictions, and can be used in a variety of ways to form your design.

- Accumulator registers A0 and A1 are often used like RAM to hold temporary values entering and coming out of the ALU. These are the most versatile registers and are also the most accessed. The values in the A0/A1 registers are not retained in sleep / hibernate.
- Data registers D0 and D1 are not as versatile as the accumulator registers. They can be written by the Datapath only from the FIFO and by consuming an extra d0_load or d1_load input signal. For this reason it is often used like ROM in the design. The values in the D0/D1 registers are retained in sleep / hibernate
- 4-word deep FIFOs F0 and F1 are often used as the input and output buffers for the Datapath. These cannot directly source the ALU and the value in the FIFO must be loaded in to the accumulator register before it can be used by the ALU. See FIFO Modes for more details on the FIFO configurations. The data in the FIFOs is not retained in sleep / hibernate.

Datapath Inputs/Outputs

A Datapath, regardless of data width can contain up to 6 input bits. Of the 6 input bits, up to 3 bits can be used to control the Datapath instructions for that clock cycle. Therefore 8 unique instructions per Datapath can be used in the design. Each of these instructions can perform multiple operations in the same clock cycle, which can further optimize performance.

Similarly, a Datapath can contain up to 6 outputs regardless of the data width. These outputs are used to send status signals from the Datapath to either a Status Register or to other blocks in the design such as the State Machine or the Count7 counter. These status signals are generated from comparisons and internal logic in the Datapath and do not include data bits directly from the registers or the ALU. It is possible however to access this information by using the shifter to serially shift out the bits in the ALU.

FIFO Modes

The 4-word deep FIFOs in Datapaths can be configured to several modes by modifying an auxiliary control configuration register. This register is used by the CPU/DMA to dynamically control the interrupt, counter, and FIFO operations. Refer to the *TRM* for more information about the auxiliary control configuration register.

FIFOs are set to either single buffer or normal mode.

- **Single buffer mode –** This mode configures the FIFO to be a 1-word deep buffer instead of the normal 4-word deep FIFO. Any value written to the FIFO immediately overwrites its content. This mode should be used if only a 1-register FIFO with its corresponding FIFO bus and block status signals are needed.
- **Normal mode –** Normal mode is the standard 4-word deep FIFO that can be used to fill up to four data words.

The data transfers to and from the FIFO are often controlled using the FIFO bus and block status signals. These are FIFO 0/1 block status (f0_block_stat, f1_block_stat), and FIFO 0/1 bus status (f0_bus_stat, f1_bus_stat) signals. The behaviors of these are dependent on the input/output mode and auxiliary control configuration register settings. The following table shows the possible configurations. FIFO is an output if it has a load signal (load with ALUOut/A0/A1) in the inputs. It is an input if is assigned to A0/A1 in the "register writes" section of any Datapath instruction.

Note This is for illustrative purposes only. For more detail descriptions on the FIFO configuration, refer to the *TRM*.

Direction	Level Mode	Signal	Status	Description
Input	N/A	Block status	Empty	Asserted when there are no bytes left in the FIFO.
	NORMAL	Bus status	Not full	Asserted when there is room for at least 1 word in the FIFO.
	MID	Bus status	At least half empty	Asserted when there is room for at least 2 words in the FIFO.
Output	N/A	Block status	Full	Asserted when the FIFO is full.
	NORMAL	Bus status	Not empty	Asserted when there is at least 1 word available to be read from the FIFO.
	MID	Bus status	At least half full	Asserted when there are at least 2 words available to be read from the FIFO.

Level mode can be configured by setting the FIFO level mode of an auxiliary control configuration register to either NORMAL or MID.

Note Level mode changes the meaning of the "bus status" signal.

- NORMAL FIFO level - A NORMAL FIFO level allows the bus status signal to assert whenever there is at least 1 word that is ready to be read or written (depending on the FIFO direction).
- MID FIFO level - A MID FIFO level allows the bus status signal to assert whenever there are at least 2 words that are ready to be read or written (depending on the FIFO direction).

Appendix B: UDB Editor Syntax

The UDB Editor uses Verilog expressions for describing the signal operations. Some of these are similar to C expressions. The following is a list of supported operators. These can be used in the expression fields of the UDB elements. The logic will be implemented using PLDs.

Category	Expression	Description	Example
Arithmetic Operators	*	Multiplication	A * B
	+	Addition	A + B
	-	Subtraction	A – B
	/	Division	A / B
	%	modulus	A % B
Shift Operators	<<	Shift left	A << 1
	>>	Shift right	A >> 1
Relational Operators	<	Less than	A < B
	>	Greater than	A > B
	<=	Less than or equal to	A <= B
	>=	Greater than or equal to	A >= B
Equality Operators	==	Equal to	A == B
	!=	Not equal to	A != B
Bit-wise Operators	~	NOT (bit-wise negation)	~A
	&	AND (bit-wise AND operation)	A & B
	|	OR (bit-wise OR operation)	A | B
	^	XOR (bit-wise XOR operation)	A ^ B
	^~	XNOR (bit-wise XNOR operation)	A ^~ B
	~^	XNOR (bit-wise XNOR operation)	A ~^ B
Reduction Operators	&	AND (reduction unary AND)	& A
	~&	NAND (reduction unary NAND)	~& A

Category	Expression	Description	Example
	|	OR (reduction unary OR)	| A
	~|	NOR (reduction unary NOR)	~| A
	^	XOR (reduction unary XOR)	^ A
	^~	XNOR (reduction unary XNOR)	^~ A
	~^	XNOR (reduction unary XNOR)	~^ A
Logical Operators	!	NOT (logical negation)	! A
	&&	AND (logical AND)	A && B
	||	OR (logical OR)	A || B
Conditional Operators	?:	Similar to ternary operator in C.	(A) ? 1'b1 : 1'b0
Concatenation	{ }	Concatenate bits	{ A, B }