PSoC® 3, PSoC 4, and PSoC 5LP Implementing Programmable Logic Designs with Verilog

Authors: Vijay Kumar Marrivagu and Antonio Rohit De Lima Fernandes
Associated Project: Yes
Associated Part Family: CY8C3xxx, CY8C5xxx, CY8C42xx
Software Version: PSoC Creator™ 2.2 SP1 or later
Related Application Notes: AN81623, AN82156
If you have a question, or need help with this application note, contact the authors at vjya@cypress.com or anto@cypress.com

AN82250 describes how to implement programmable digital logic designs in the PLD portion of PSoC® 3, PSoC 4 and PSoC 5LP. It introduces the PSoC Universal Digital Blocks (UDBs) and their Programmable Logic Device (PLD) sub-blocks. An example project illustrates how you can use the PLDs in a design by creating Verilog-based components in PSoC Creator™.

Contents

Introduction

PSoC® 3, PSoC 4 and PSoC 5LP (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.

Note This application note does not apply to PSoC 41xx parts which do not contain UDBs.

This application note introduces the PLDs in the PSoC Universal Digital Block (UDB), and then teaches how to use them by creating PSoC Creator components. It is an effective first step in porting complex programmable logic device (CPLD) functionality to PSoC. After reading this application note, you should be familiar with PSoC PLDs, and be able to create your own custom Verilog-based components using PSoC Creator™.

To take full advantage of PSoC's digital features, the next step is to read AN82156 – PSoC 3, PSoC 4 and PSoC 5LP Designing PSoC Creator Components with UDB Datapaths.

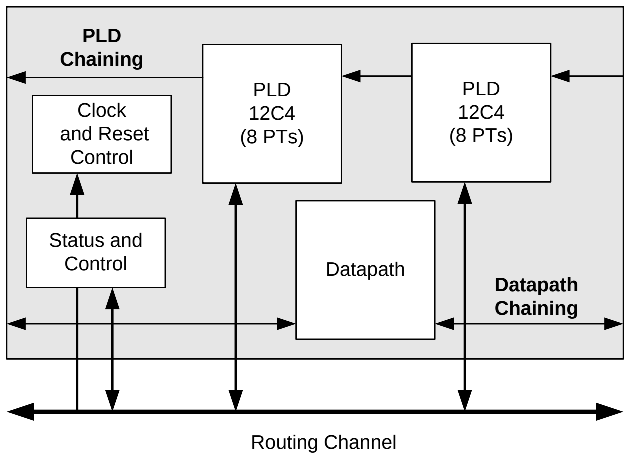
Note This is an advanced application note – it assumes familiarity with PSoC Creator. If you are new to PSoC, see AN54181 – Getting Started with PSoC 3, AN79953 – Getting Started with PSoC 4, and AN77759 – Getting Started with PSoC 5LP. If you are new to PSoC Creator, see the PSoC Creator home page.

This application note also assumes a basic understanding of digital design and Verilog. If you are new to these concepts, see AN81623 – PSoC 3 and PSoC 5LP Digital Design Best Practices and KBA86336 – Just Enough Verilog for PSoC. The References section lists related PSoC digital design resources.

PSoC UDBs

PSoC implements programmable logic through an array of small, fast, low-power digital blocks called Universal Digital Blocks (UDBs). PSoC devices have as many as 24 UDBs. As shown in Figure 1, a UDB consists of two small programmable logic devices (PLDs), a datapath module, and status and control logic.

Figure 1. UDB Block Diagram



Programmable logic, as the name implies, is a family of devices that contain arrays of logic elements: AND, OR, INVERT, and FLIP-FLOP. In general, a PLD is a circuit that can be configured to perform a specific logic function.

PSoC PLDs can be used to form registered or combinatorial sum of products logic, lookup tables, multiplexers, state machines, and as control for datapath operations. For more information on UDB datapaths, see AN82156.

Architecture of PLDs in PSoC UDB

PSoC PLDs, like most standard PLDs, consist of an AND array followed by an OR array, both of which are programmable. This is commonly referred to as a sum of products architecture.

There are 12 inputs which feed across eight product terms (PTs) in the AND array. In each PT, either the true (T) or complement (C) of the input can be selected. The outputs of the PTs are inputs into the OR array. The outputs of the OR gates are fed to macrocells (MC). Macrocells are flip-flops with additional combinatorial logic.

There are two PLDs in each UDB; each with 8 PTs and 4 macrocells as Figure 2 shows. PSoC has as many as 48 PLDs and thus 192 macrocells and 384 PTs. Each PLD is independent and can be connected through carry chains or to the digital system interconnect (DSI).

Appendix A compares PSoC PLD resources with similar-sized competitive PLDs.

Figure 2. PSoC PLD Structure

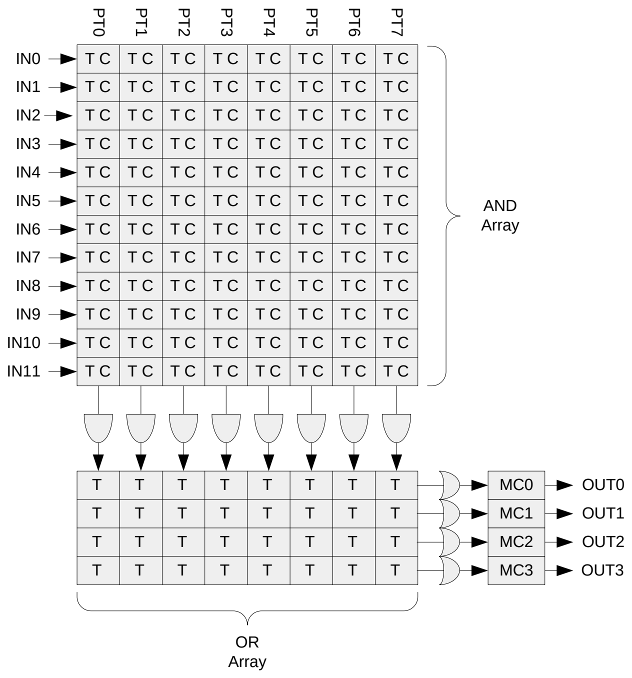
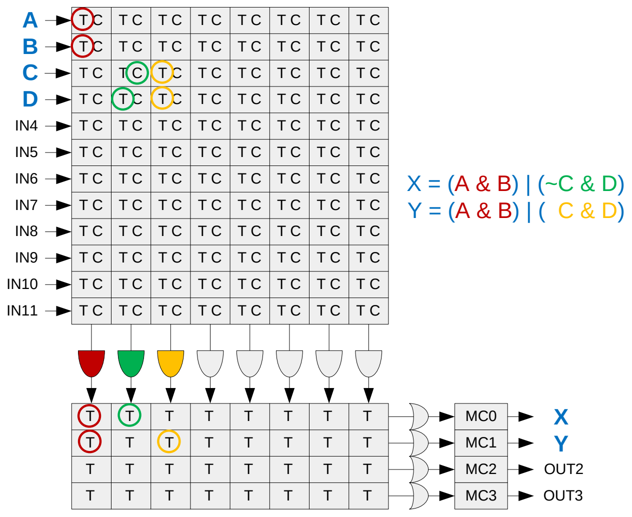


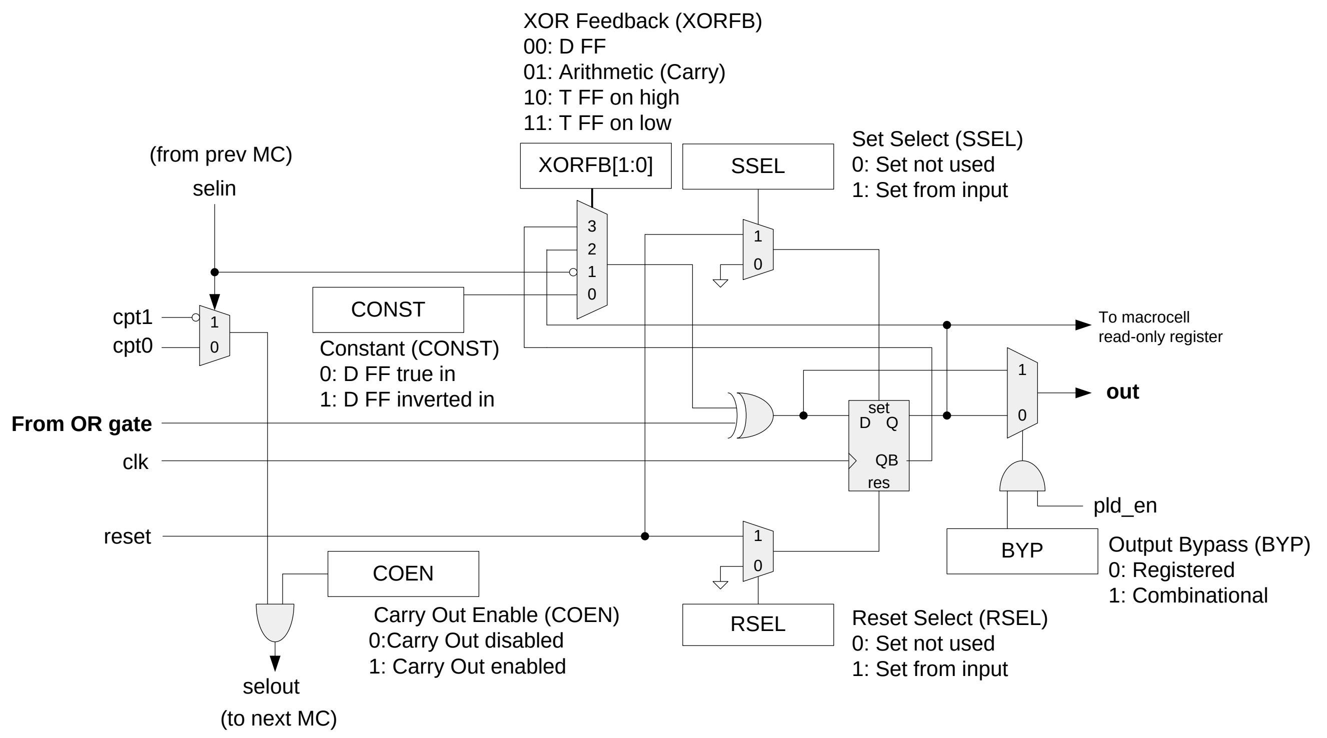
Figure 3 shows an example of logic equations mapped to a PSoC PLD.

Figure 3. Logic Equations Mapped to the PLD



The macrocell architecture is shown in Figure 4. The macrocell output can be registered or combinatorial. Appendix B explains the data flow through a macrocell using two examples. For more information, see section 23.3.1.1 of the PSoC 3 Technical Reference Manual.

Figure 4. PSoC PLD Macrocell Architecture



PSoC Creator™

PSoC Creator provides a schematic-based environment for hardware development. It enables you to implement logic functions and state machines in the UDB PLDs via two broad methods:

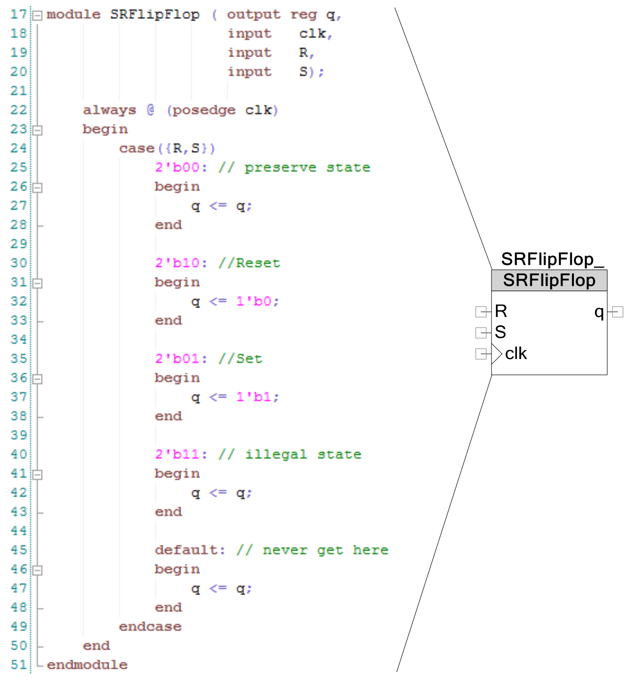
1. Verilog: PSoC Creator supports Verilog, which is a hardware description language (HDL). Using Verilog, you can implement digital functions which then map to the PSoC UDBs. This process uses the Warp™ synthesis tool which is a Verilog compiler included with PSoC Creator.

   In this application note you will learn how to create Verilog-based components (see Figure 5).

   To learn about Verilog, see KBA86336 – Just Enough Verilog for PSoC.

Note For information about Warp, see the Warp Verilog Reference Guide in PSoC Creator, under **Help > Documentation**.

Figure 5. Verilog-based Component

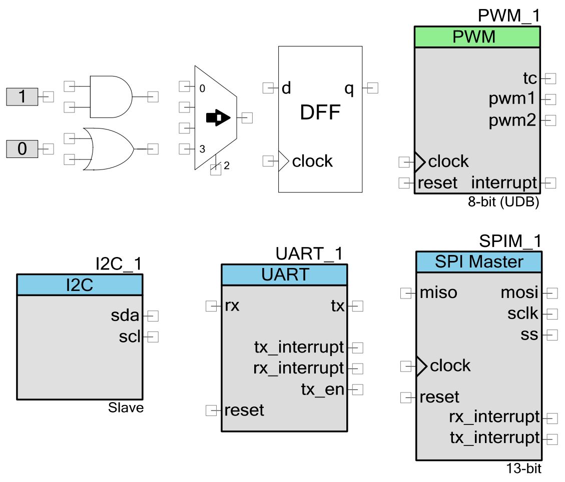


```
module SRFlipFlop ( output reg q,
                    input   clk,
                    input   R,
                    input   S);

    always @ (posedge clk)
    begin
        case({R,S})
            2'b00: // preserve state
            begin
                q <= q;
            end

            2'b10: //Reset
            begin
                q <= 1'b0;
            end

            2'b01: //Set
            begin
                q <= 1'b1;
            end

            2'b11: // illegal state
            begin
                q <= q;
            end

            default: // never get here
            begin
                q <= q;
            end
        endcase
    end
endmodule
```

2. Schematic: This process involves wiring individual gates (AND, OR, XOR, NOT), DFFs and other digital logic blocks to perform required functions. PSoC Creator offers gate symbols for all logic operations, as well as multiplexers, lookup tables (LUTs), and other simple PLD-based functions.

   PSoC Creator also provides a library of pre-built and tested standard peripheral components. These components are mapped onto the UDB array which includes both PLDs and datapaths. Some of these components are shown in Figure 6. Using these components is the quickest and simplest way to use the PLD capabilities of the PSoC without using Verilog.

Figure 6. Digital Components in PSoC Creator



Example Project

One of the best ways to learn about PSoC is to use it. This example project teaches the steps to create simple PLD-based Verilog components.

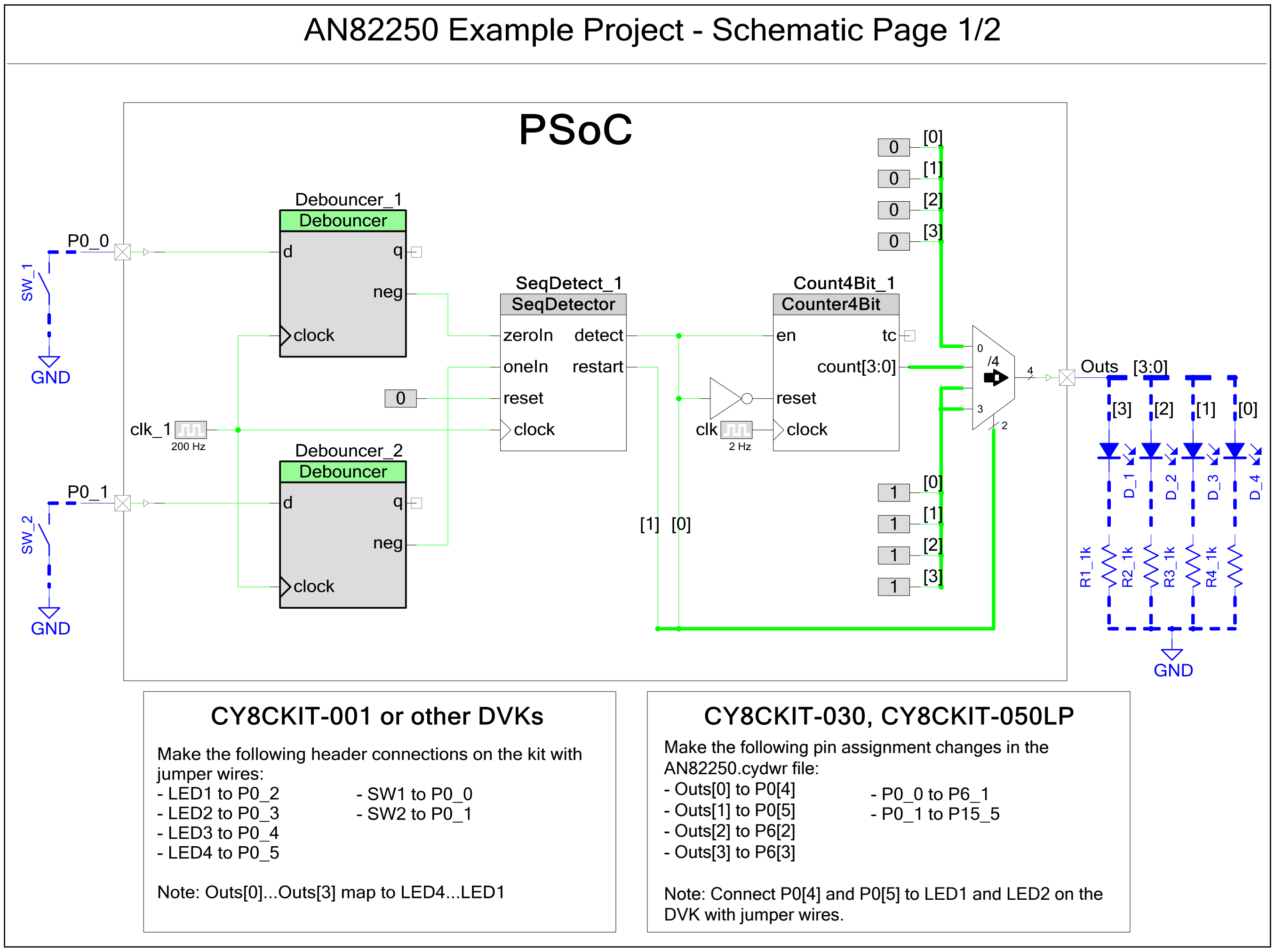
To begin, download the AN82250.zip file from the application note landing page. To view the project, unzip the folder and then open the AN82250.cywrk file in PSoC Creator. The project is designed to work with PSoC 3 on the PSoC development kit (DVK) CY8CKIT-001 by following the instructions on the schematic. With minor modifications, it can be run on other development platforms. Build and program this project onto the PSoC DVK.

This example project implements a 5-bit sequence detector completely in hardware – no firmware is required. For details of the schematic, see Figure 7. An important feature of this project is that, with the exception of the clocks and pins, all the components shown on the schematic are implemented in the UDB PLDs.

The project takes a binary pattern as its input. Two push-button switches on the PSoC DVK generate the pattern. A button press on switch ‘SW_1’ is interpreted as logic 0 and a press on switch ‘SW_2’ is interpreted as logic 1. Four outputs drive LEDs on the DVK to indicate the detector status.

On resetting the PSoC, the LEDs glow to indicate that the PSoC is ready for an input (that is, a switch press). The PSoC then follows the state diagram shown in Figure 8. If you enter an incomplete, but correct sequence, the LEDs turn off indicating that you have entered a partial sequence. A wrong switch press causes the four LEDs to turn on. If you enter the complete 5-bit sequence correctly, the LEDs begin counting in a binary fashion.

Figure 7. TopDesign Schematic for the Example Project



CY8CKIT-001 or other DVKs

Make the following header connections on the kit with jumper wires:

- LED1 to P0_2
- LED2 to P0_3
- LED3 to P0_4
- LED4 to P0_5
- SW1 to P0_0
- SW2 to P0_1

Note: Outs[0]...Outs[3] map to LED4...LED1

CY8CKIT-030, CY8CKIT-050LP

Make the following pin assignment changes in the AN82250.cydwr file:

- Outs[0] to P0[4]
- Outs[1] to P0[5]
- Outs[2] to P6[2]
- Outs[3] to P6[3]
- P0_0 to P6_1
- P0_1 to P15_5

Note: Connect P0[4] and P0[5] to LED1 and LED2 on the DVK with jumper wires.

The signal flow from input to output is as follows:

■ The two push-button switch inputs are debounced and edge-detected using the debouncer component. Please update PSoC Creator to Component Pack 4 or later to access this component.

■ The input pins are configured as resistive pull-up. The push-button inputs thus transition from high to low on a switch press event. Hence, the debouncer's 'negative edge detect' output is used to indicate a valid switch press.

■ These signals then go to the SeqDetector component which is configured to detect a sequence of 10110. This pattern can be changed by entering a value between 0 (00000) and 31 (11111) in the component customizer (Figure 9).

■ If the entire sequence is entered correctly, the 'detect' output is asserted. Even if a single wrong entry is made, the restart output is asserted.

■ The 4-bit counter begins to count when the 'detect' signal is asserted, or else is held in reset. The counter's period can be adjusted by entering a desired 4-bit period value between 1 and 15 in the component customizer (Figure 12).

■ The 'restart' and 'detect' signals control the output mux to drive 4 LEDs based on the state diagram in Figure 8.

Figure 8. State Diagram for Example Project

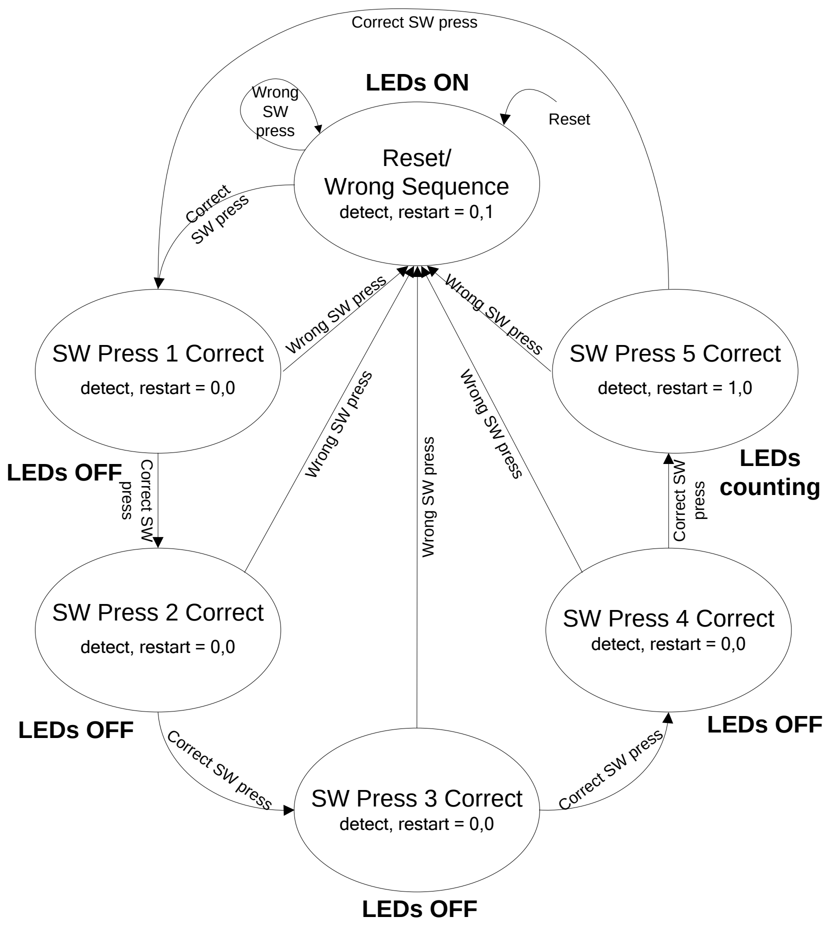
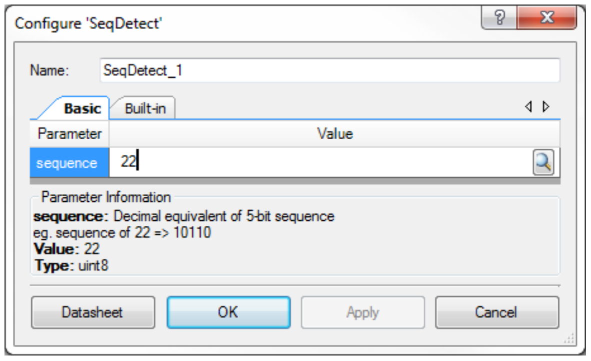


Figure 9. SeqDetector Component Customizer

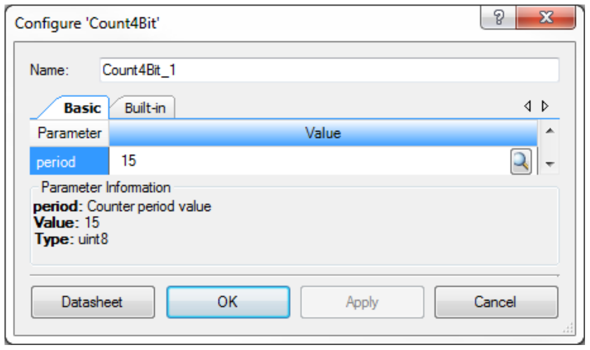


Name: SeqDetect_1

Parameter	Value
sequence	22

Parameter Information
sequence: Decimal equivalent of 5-bit sequence
eg. sequence of 22 => 10110
Value: 22
Type: uint8

Figure 10. Counter Component Customizer



Name: Count4Bit_1

Parameter	Value
period	15

Parameter Information
period: Counter period value
Value: 15
Type: uint8

Note To view the Verilog files for the 4-bit counter and sequence detector components navigate to the **Components** tab of the **Workspace Explorer**.

The key to using the PSoC PLDs effectively is to create Verilog-based components in PSoC Creator. KBA86338 – Creating a Verilog-based Component summarizes the Verilog-based Component creation process. You can become familiar with this process, using the SeqDetector and Counter4Bit components as examples.

Create Verilog Component: Counter4Bit

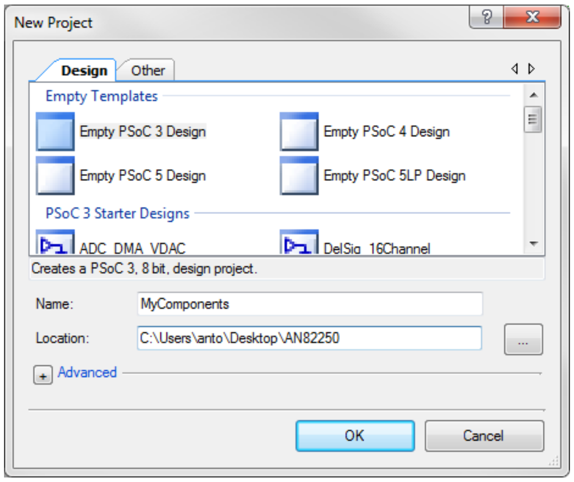
One of the simplest custom Verilog-based components is a 4-bit up-counter with synchronous reset and enable.

4-bit Counter Component Creation Steps

You could use an existing project and add a new component to it, but for this example, use an empty project as a starting point.

1. Launch **PSoC Creator** and start a new project. For this example, 'MyComponents' is used as the project name, as Figure 11 shows.

   Figure 11. New Project Dialog Box



Creates a PSoC 3, 8 bit, design project.

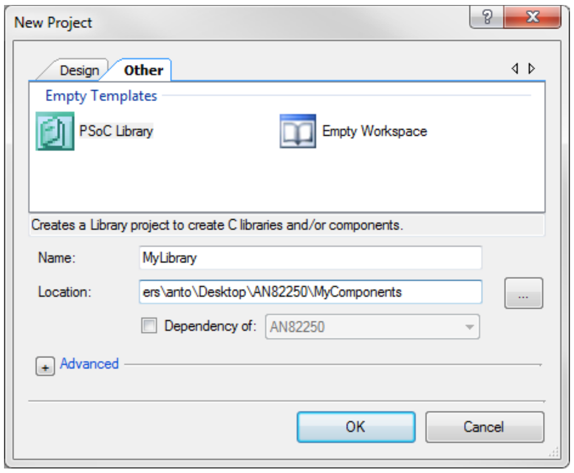
Name: MyComponents

Location: C:\Users\anto\Desktop\AN82250

2. In the **Source** tab of the **Workspace Explorer,** right-click on the **MyComponents** workspace and then click **Add** > **New Project**.
3. To set this new project to be a component library, in the **New Project** dialog box, click on the **Other** tab and select **PSoC Library** (Figure 12). Name it 'MyLibrary' for this example, and leave the location at its default value.

   It is best to create custom components in separate library projects. This simplifies component management and reuse.

Figure 12. Adding a Library Project



Creates a Library project to create C libraries and/or components.

Name: MyLibrary

Location: ers\anto\Desktop\AN82250\MyComponents

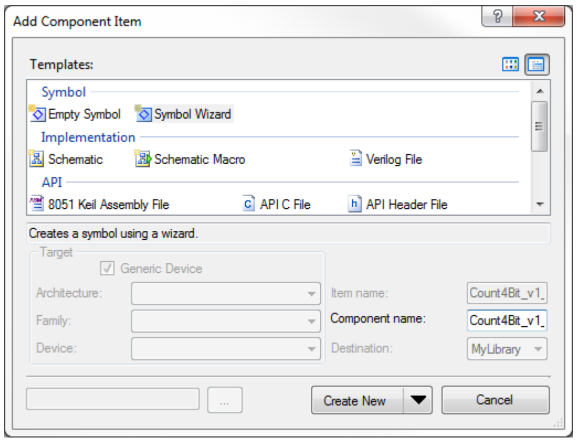
Dependency of: AN82250

Now add a new component to the library just created:

4. On the **Components** tab, right-click on the 'MyLibrary' project and then click **Add Component Item** from the context menu (Figure 13).

   It is 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 and helps track multiple versions of your components.
5. Select the **Symbol Wizard** component template and name the component 'Count4Bit_v1_00'.

   Figure 13. Creating a Custom Component



Creates a symbol using a wizard.

Item name: Count4Bit_v1_

Component name: Count4Bit_v1_

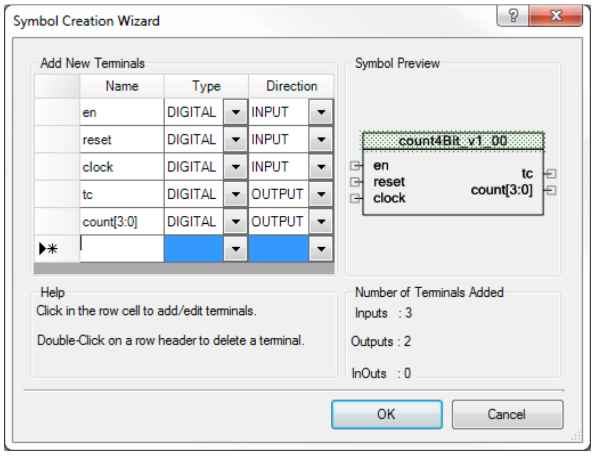
Destination: MyLibrary

   You can start from an empty symbol, but this example uses the wizard to save time. For more information, see the Component Author Guide under **Help > Documentation**.
6. To launch the component symbol wizard, click the **Create New** button.

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

7. Define three input terminals and two output terminals for the schematic symbol as Figure 14 shows.

Figure 14. Symbol Creation Wizard for Count4Bit



Symbol Creation Wizard

Add New Terminals

Name	Type	Direction
en	DIGITAL	INPUT
reset	DIGITAL	INPUT
clock	DIGITAL	INPUT
tc	DIGITAL	OUTPUT
count[3:0]	DIGITAL	OUTPUT

Symbol Preview

Help

Click in the row cell to add/edit terminals.

Double-Click on a row header to delete a terminal.

Number of Terminals Added

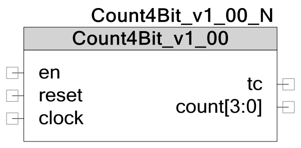
Inputs : 3

Outputs : 2

InOuts : 0

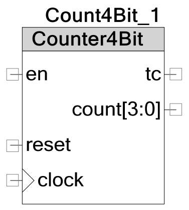
Click **OK** to generate the symbol in the symbol schematic, as Figure 15 shows.

Figure 15. Initial Symbol for 4-bit Counter



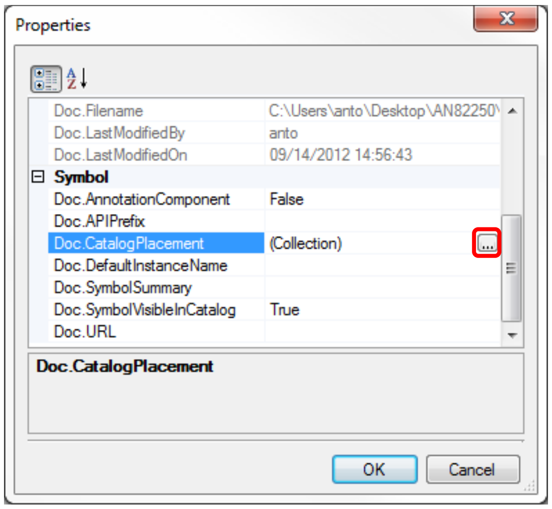
You can resize the component, and modify the appearance of the component as shown in Figure 16.

Figure 16. 4-Bit Counter Final Symbol



8. Right-click on an empty space in the symbol schematic, and then click **Properties**. In the **Symbol** section of the property fields, click on the ellipsis (…) on **Doc**.**CatalogPlacement**, as shown in Figure 17.

Figure 17. Symbol Properties Dialog Box



Properties

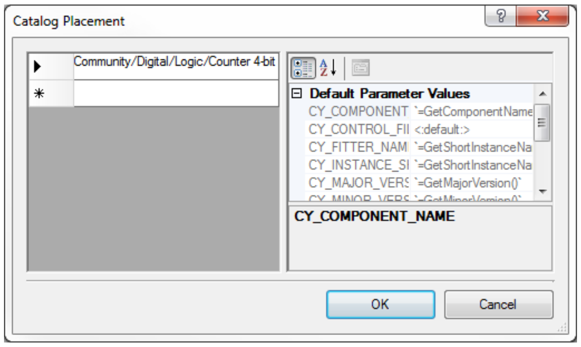
Doc.Filename	C:\Users\anto\Desktop\AN82250\
Doc.LastModifiedBy	anto
Doc.LastModifiedOn	09/14/2012 14:56:43
Symbol	
Doc.AnnotationComponent	False
Doc.APIPrefix	
Doc.CatalogPlacement	(Collection)
Doc.DefaultInstanceName	
Doc.SymbolSummary	
Doc.SymbolVisibleInCatalog	True
Doc.URL	

Doc.CatalogPlacement

9. Enter Community/Digital/Logic/Counter 4-bit in the **Catalog Placement** dialog, as shown in Figure 18.

   This places the counter in the **Community** tab of the **Component Catalog** window, under the ‘Logic’ sub-folder of the ‘Digital’ folder, with the catalog name of ‘Counter 4-bit’.

Figure 18. Setting Catalog Placement



Catalog Placement

Community/Digital/Logic/Counter 4-bit

Default Parameter Values

CY_COMPONENT	`=GetComponentName
CY_CONTROL_FII	<:default:>
CY_FITTER_NAM	`=GetShortInstanceNa
CY_INSTANCE_SI	`=GetShortInstanceNa
CY_MAJOR_VERS	`=GetMajorVersion()`

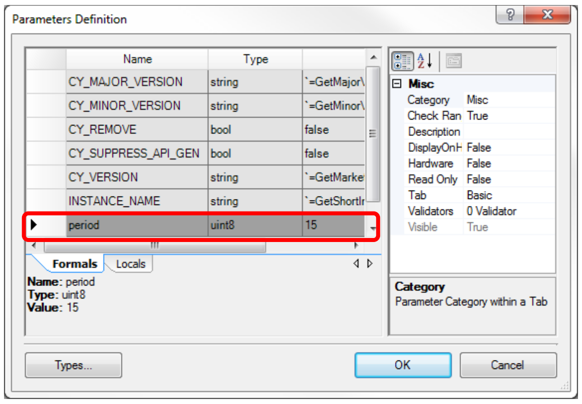
CY_COMPONENT_NAME

In order to have a configurable period value for the counter, you must add a component parameter.

10. Right-click on an empty space in the symbol schematic and then click ***Symbol Parameters*** (Figure 19).

    Specify the *name*, *type,* and default *value* of the parameter as period, uint8, and 15, respectively. This parameter allows a user to specify a period value for the counter in its customizer (Figure 10).

Figure 19. Symbol Parameters for Count4Bit



Parameters Definition

	Name	Type	
	CY_MAJOR_VERSION	string	`=GetMajor\
	CY_MINOR_VERSION	string	`=GetMinor\
	CY_REMOVE	bool	false
	CY_SUPPRESS_API_GEN	bool	false
	CY_VERSION	string	`=GetMarke
	INSTANCE_NAME	string	`=GetShortIr
▶	period	uint8	15

Formals | Locals

Name: period
Type: uint8
Value: 15

Misc	
Category	Misc
Check Ran	True
Description	
DisplayOnH	False
Hardware	False
Read Only	False
Tab	Basic
Validators	0 Validator
Visible	True

Category
Parameter Category within a Tab

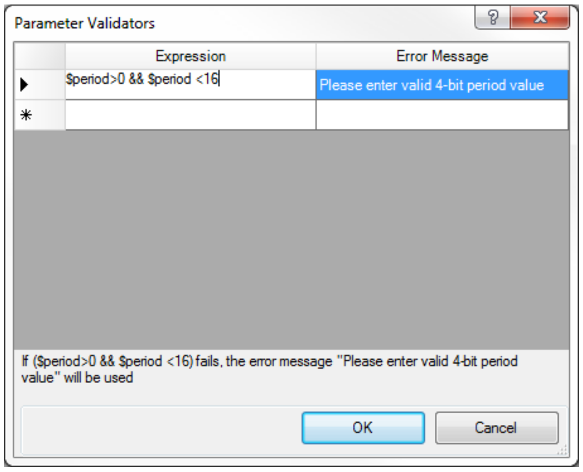
Types... OK Cancel

11. Enter a description for this parameter by clicking on the **Description** field in the **Misc** section of the **Parameter Definition** dialog box.

    Set a validator for the parameter by clicking on the **Validators** field. A validator checks whether a parameter is within an acceptable input range.

    Set a validator to ensure that the period value is between 1 and 15, as shown in Figure 20. Click **OK** to make the changes.

Figure 20. Adding Validators for Count4Bit



Parameter Validators

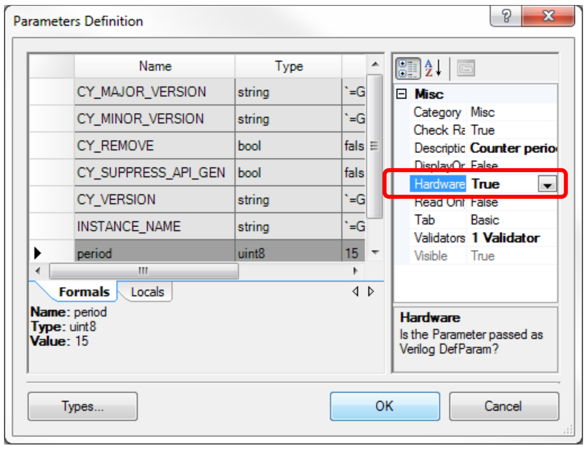
	Expression	Error Message
▶	$period>0 && $period <16	Please enter valid 4-bit period value
✳		

If ($period>0 && $period <16) fails, the error message "Please enter valid 4-bit period value" will be used

OK Cancel

12. In the **Parameter Definition** dialog box, set the **Hardware** field to **True**, as Figure 21. This is necessary to pass the parameter to the Verilog file.

Figure 21. Passing Parameter to Hardware



Parameters Definition

	Name	Type	
	CY_MAJOR_VERSION	string	`=G
	CY_MINOR_VERSION	string	`=G
	CY_REMOVE	bool	fals
	CY_SUPPRESS_API_GEN	bool	fals
	CY_VERSION	string	`=G
	INSTANCE_NAME	string	`=G
▶	period	uint8	15

Formals | Locals

Name: period
Type: uint8
Value: 15

Misc	
Category	Misc
Check Ra	True
Descriptic	**Counter perio**
DisplayOr	False
Hardware	**True**
Read Onl	False
Tab	Basic
Validators	**1 Validator**
Visible	True

Hardware
Is the Parameter passed as Verilog DefParam?

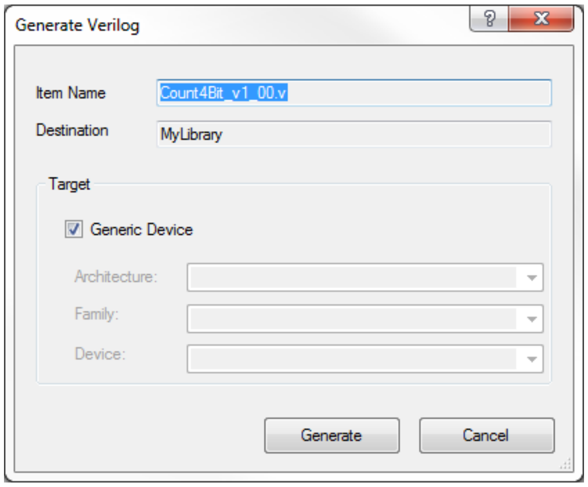
Types... OK Cancel

13. The next step is to link the schematic symbol to a Verilog file. PSoC Creator generates a Verilog shell based on the component symbol.

    To do this, right-click on an empty space in the symbol schematic and then click **Generate Verilog**.

    Leave all settings in the **Generate Verilog** dialog box at the defaults and click **Generate**, as shown in Figure 22.

Figure 22. Generate the Verilog File for the Symbol



Generate Verilog

Item Name: Count4Bit_v1_00.v
Destination: MyLibrary

Target
☑ Generic Device
Architecture:
Family:
Device:

Generate Cancel

The **Target** values can be used to limit the configuration to a specific device, but for this example use the default setting.

A Verilog file for the symbol just created appears.

Note There are three `#start header` - `#end` pairs in the Verilog file. When editing the file, put all your code within these sections. Changes made to the Verilog file outside these three sections will be lost if you regenerate the Verilog file.

You are now ready to describe the counter in Verilog. For reference, the complete code is shown in Code 1.

Verilog Design: 4-bit Counter

First, make the outputs registered. Modify the input/output list of the Count4Bit_v1_00 module to:

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

Note If you regenerate the Verilog file, you must make these changes again. Also, these definitions cannot be made anywhere else in the file.

Then, since this is a synchronous design, add an *always* block (with clock edge) between the `#start body` and `#end` comments in the Verilog file:

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

Note To reduce the likelihood of timing and synchronization failures, it is preferable to use posedge clocking in PSoC designs.

The counter has a synchronous reset which when asserted clears both ‘tc’ and ‘count’.

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

Note Asynchronous reset/preset signals are supported as well. Refer to the *Warp Verilog Reference Guide* section 3.3.2 for information on asynchronous flip-flop synthesis.

The `en` input signal is a hardware enable. If this input is low the outputs are still active but the component does not change states.

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

When `count` reaches the period value, the terminal count output `tc` should be at logic 1 as long as `count` is equal to `period`.

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

Else, the 4-bit counter must count from 0 to `period`, and increment the `count` output every positive clock edge.

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

After you finish making changes to the Verilog file, save it. You have now completed the Verilog description for a 4-bit up-counter. The completed code is shown in Code 1.

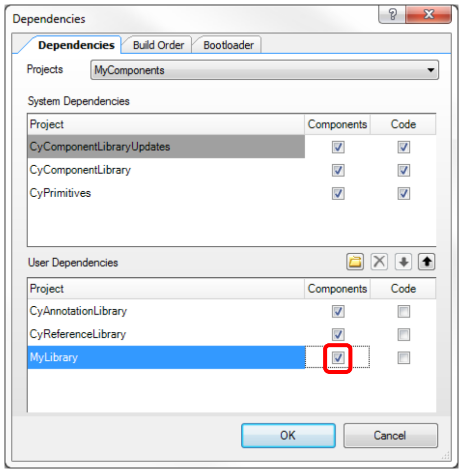
Code 1. Complete 4-bit Counter Verilog Design

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

  parameter period = 0;

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

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

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

In order to use this component in the ‘MyComponents’ project, you need to set ‘MyLibrary’ as a dependency.

To do this, right-click on 'MyComponents' in the **Source** tab, and select **Dependencies**. Ensure that the checkbox for 'MyLibrary' under **User Dependencies** is checked, as Figure 23 shows.

Figure 23. Adding a Component Library Dependency


Dependencies

Dependencies | Build Order | Bootloader

Projects: MyComponents

System Dependencies

Project	Components	Code
CyComponentLibraryUpdates	✓	✓
CyComponentLibrary	✓	✓
CyPrimitives	✓	✓

User Dependencies

Project	Components	Code
CyAnnotationLibrary	✓	☐
CyReferenceLibrary	✓	☐
MyLibrary	✓	☐

To use it in a design, go to the TopDesign.cysch file of the MyComponents project and navigate to the **Component Catalog**. The 4-bit counter is located in the **Community** tab. Drag and drop it onto the schematic, and make the required connections.

Note For more information on how to create and use library projects, see PSoC Creator help articles 'Library Component Project' and 'Basic Hierarchical Design Tutorial'.

Continue by creating the sequence detector component in Verilog.

Create Verilog Component: SeqDetector

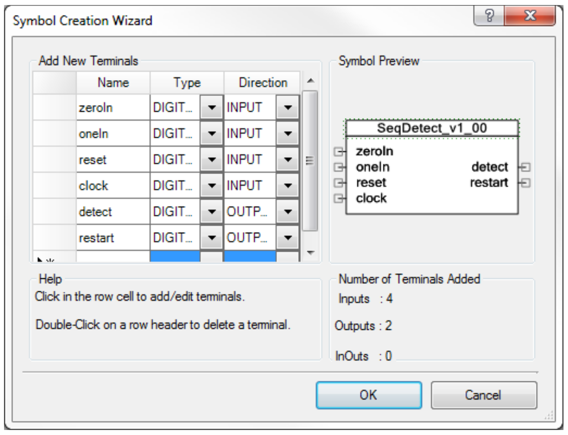
The SeqDetector component is the heart of this example project. It is a configurable 5-bit binary sequence detector implemented in the PSoC PLDs.

SeqDetector Component Creation Steps

The steps to create the SeqDetector are similar to those for the counter.

1. Select the **Components** tab of the **Workplace Explorer**. Right-click on the **MyLibrary** project and then click **Add Component Item**.
2. Select the **Symbol Wizard** component template and name the component **SeqDetect_v1_00**.
3. Click the **Create New** button to launch the component symbol wizard.
4. Define four input terminals and two output terminals for the schematic symbol as shown in Figure 24 .

Figure 24. Symbol Creation Wizard for SeqDetector


Symbol Creation Wizard

Add New Terminals

Name	Type	Direction
zeroIn	DIGIT...	INPUT
oneIn	DIGIT...	INPUT
reset	DIGIT...	INPUT
clock	DIGIT...	INPUT
detect	DIGIT...	OUTP...
restart	DIGIT...	OUTP...

Symbol Preview

Help

Click in the row cell to add/edit terminals.

Double-Click on a row header to delete a terminal.

Number of Terminals Added

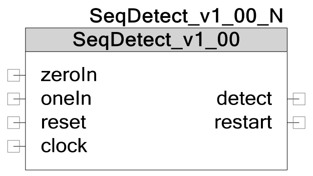
Inputs : 4

Outputs : 2

InOuts : 0

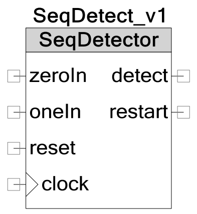
5. Click **OK** to generate the symbol in the symbol schematic, as shown in Figure 25.

Figure 25. Sequence Detector Initial Symbol



You can resize the component to look like Figure 26:

Figure 26. Final Symbol for Sequence Detector



6. Right-click on an empty space in the symbol schematic and then click **Properties**.

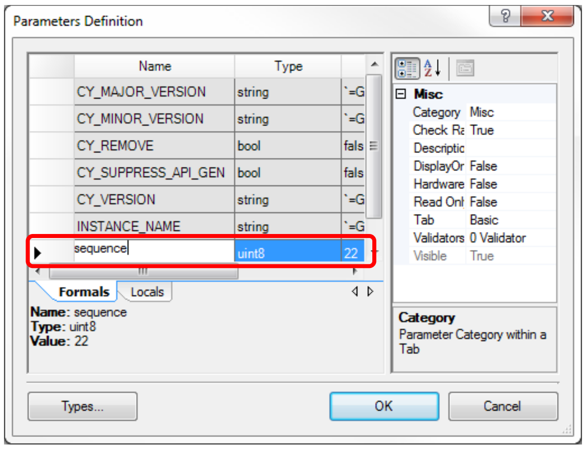
   In the **Symbol** section of the property fields, click on the ellipsis (…) on **Catalog Placement**. Enter **Community/Digital/Logic/Sequence Detector 5-bit** as the **CatalogPlacement**.

   This places the SeqDetector on the **Community** tab of Component Catalog, under the 'Logic' sub-folder of the 'Digital' folder, with the catalog name 'Sequence Detector 5-bit'.
7. In order to have a configurable sequence value for the SeqDetector, you must add a component parameter.

   Right-click on an empty space in the symbol schematic and then click **Symbol Parameters**.

   Specify the *name*, *type,* and default *value* of the parameter as sequence, uint8, and 22, respectively, as shown in Figure 27.

Figure 27. Defining a Parameter for SeqDetect



Parameters Definition

	Name	Type	
	CY_MAJOR_VERSION	string	`=G
	CY_MINOR_VERSION	string	`=G
	CY_REMOVE	bool	fals
	CY_SUPPRESS_API_GEN	bool	fals
	CY_VERSION	string	`=G
	INSTANCE_NAME	string	`=G
▶	sequence	uint8	22

Formals | Locals

Name: sequence
Type: uint8
Value: 22

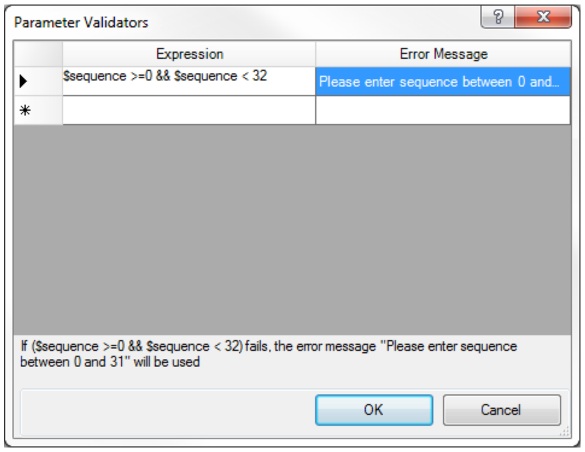
Misc

Category	Misc
Check Ra	True
Descriptio	
DisplayOr	False
Hardware	False
Read Onl	False
Tab	Basic
Validators	0 Validator
Visible	True

Category
Parameter Category within a Tab

8. Enter a description for this parameter by clicking on the **Description** field in the **Misc** section of the **Parameter Definition** dialog box.
9. Set a validator for the sequence by clicking on the **Validators** field. Set a validator to ensure that the sequence value is between 0 and 31, as shown in Figure 28. Click **OK** to make the changes.

   Figure 28. Validator for SeqDetect



Parameter Validators

	Expression	Error Message
▶	$sequence >=0 && $sequence < 32	Please enter sequence between 0 and...
*		

If ($sequence >=0 && $sequence < 32) fails, the error message "Please enter sequence between 0 and 31" will be used

10. Once back to the **Parameter Definition** dialog box, set the **Hardware** field to **True**.
11. The next step is to link the schematic symbol to a Verilog file.

    To do this, right-click on an empty space in the symbol schematic and then click **Generate Verilog**.
12. Leave all settings in the **Generate Verilog** dialog box at the defaults and click **Generate**.

The complete Verilog code for the sequence detector module is included in Appendix C. The next section explains major parts of the code.

Verilog Design: SeqDetector

As with the counter, the first step is to register the outputs in the terminal list of the SeqDetect module:

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

The backbone of the sequence detector is a state machine with a total of six states (Figure 8). Create local parameters for each of the states by adding the following code just after the `#start body` comment:

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

Notice that the state definition constants are declared using the `localparam` keyword. This prevents them from conflicting with constants with the same names in other modules.

Declare the state variables as 'register' type and the pattern variable as 'wire' type.

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

The sequence detector module has two `always` blocks – a sequential block, and a combinatorial block.

The sequential `always` block models positive edge triggered flip-flops.

The combinatorial `always` block has a sensitivity list defined as:

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

Note When writing Verilog for PSoC Creator, the `always` statement must have sensitivity list.

The combinatorial `always` decodes the inputs and assigns the next state based on the inputs and current state. If either a one or zero input is detected, the input is checked, or else the current state is maintained.

For example, the start state looks like:

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

Note `pattern[4]` holds the first correct value of the sequence.

The other states are similar – comparing the component inputs to `pattern[3]`, … , `pattern[0]` in order to decode the next state.

After you finish making changes to the Verilog file, save it. The sequence detector design is now complete. To use it in a design, please follow the steps described in the counter section.

Datapath vs. PLD-based Designs

Communication, timing, and control applications have different requirements in terms of the logic structures underpinning the functions.

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

- PLDs (Random Logic): Control functions, CPLD-integration, glue logic.
- Datapaths (Structured Logic): Communications, timing, calculations.

For example, consider the following 8-bit arithmetic and logic operations implemented in PLDs versus the datapaths.

Function	**Resource consumption in PLDs only**		**Resource consumption in datapaths only**	
	PLDs	**% Used**	**Datapath**	**% Used**
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%

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

Summary

This application note introduced UDB PLDs and explained the design process for Verilog-based component creation in PSoC Creator. After reading this application note, you should understand the PLD architecture, and be able to create your own custom Verilog-based components.

PSoC UDBs provide a flexible and efficient architecture for your digital designs. You can port a wide range of simple to moderately complex logic designs to PSoC PLDs. Designs with high complexity are best implemented by using a combination of both PLDs and datapaths. For more information on the UDB datapaths, please read AN82156.

Additional Information

Appendix A contains a comparison of PSoC PLD and competitive CPLDs with respect to resource count

Appendix B explains the data flow through a macrocell using two examples.

Appendix C contains the complete Verilog code for the sequence detector module.

Appendix D briefly discusses the project report file and static timing analysis.

References

Application notes

AN82156 - PSoC 3, PSoC 4 and PSoC 5LP Designing PSoC Creator Components with UDB Datapaths

AN81623 – PSoC 3 and PSoC 5LP Digital Design Best Practices

AN62510 – Implementing State Machines with PSoC 3 and PSoC 5LP

AN61290 – PSoC 3 and PSoC 5LP Hardware Design Considerations

AN72382 – Using PSoC 3 and PSoC 5LP GPIO Pins

AN60580 – SIO Tips and Tricks in PSoC 3 and PSoC 5LP

AN54181 – Getting started with PSoC 3

AN79953 – Getting Started with PSoC 4

AN77759 – Getting started with PSoC 5LP

KB Articles

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

Videos

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

Basics

Creating a New Project

Using the Start Page

Component Creation

PSoC Creator 113: PLD Based Verilog Components

Creating a New Component Symbol

Creating a Verilog Implementation

Creating a Schematic Implementation

About the Authors

Name:	Vijay Kumar Marrivagu
Title:	Systems Engineer Principal
Background:	Several years of experience in digital design and validation.
Contact:	vjya@cypress.com

Name:	Antonio Rohit De Lima Fernandes
Title:	Applications Engineer
Background:	B.E in EE, BITS, Pilani, Rajasthan, India.
Contact:	anto@cypress.com

Appendix A: PSoC PLD Resource Comparison with Competitive CPLDs

Table 1 compares PSoC PLD resources to similar-sized CPLDs. Keep in mind that this table does not take the programmable logic in the UDB Datapath into consideration. When using the PSoC PLDs and Datapaths both, PSoC is competitive with much larger CPLDs.

Table 1. PSoC PLD Macrocell Comparison with Competitive PLDs

Device	Macrocells (MCs)	Blocks	MC per Block	Inputs per Block	Product Terms (PTs)	PTerms per Block
Cypress PSoC						
Superset PSoC 3, PSoC 5LP	192	48	4	12	384	8
CY8C42	32	8	4	12	64	8
Altera MAX-II						
EPM240	128 to 240*	24	10	36	*	*
Lattice ispMACH						
4032ZE	32	2	16	36	160	80
4064ZE	64	4	16	36	320	80
40128ZE	128	8	16	36	640	80
Xilinx Coolrunner-II						
XC2C32A	32	2	16	56	112	56
XC2C64A	64	4	16	56	224	56
XC2C128	128	8	16	56	448	56

** Altera MAX-II is not a traditional Product Term architecture*

Appendix B: Macrocell Configuration Diagrams

Figure 29 and Figure 30 show the data flow through the macrocell for D flip flop (D-FF) and T flip flop (T-FF) functionality respectively.

Figure 29. Macrocell with D-FF Function Enabled

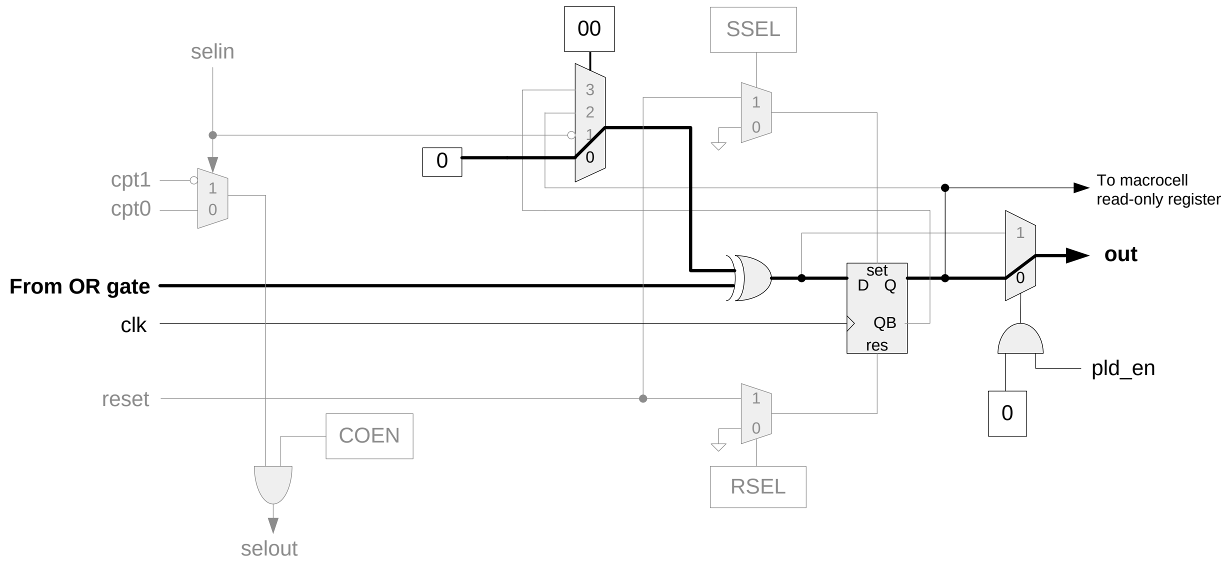
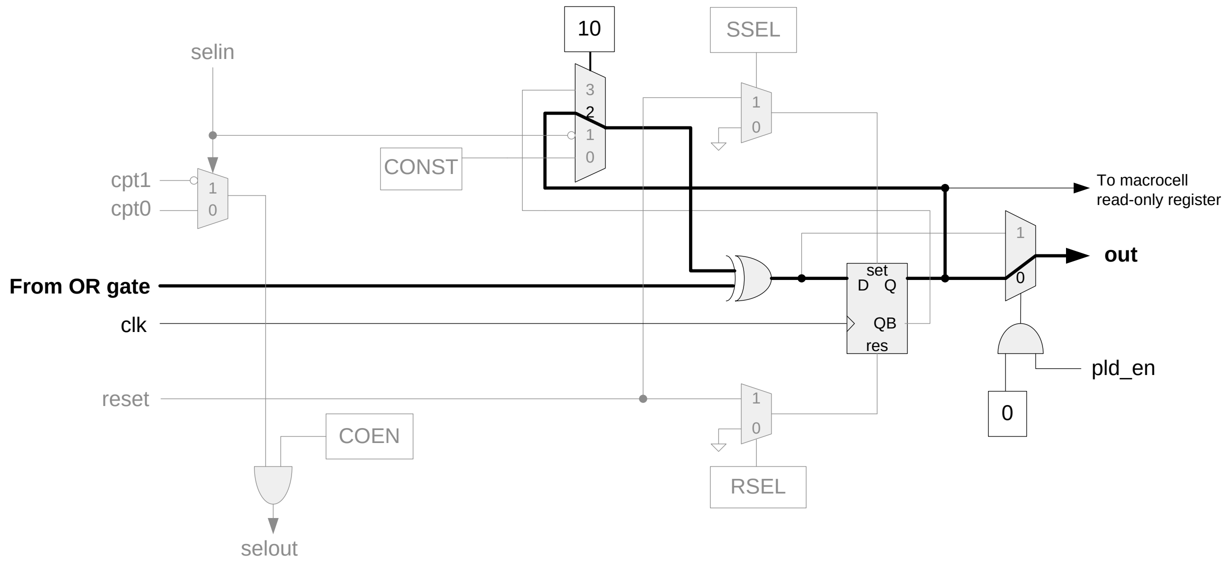


Figure 30. Macrocell with T-FF Function Enabled



Appendix C: Sequence Detector Verilog Code

```
module SeqDetect_v1_20 (
       output reg detect,
       output reg restart,
       input   clock,
       input   oneIn,
       input   reset,
       input   zeroIn
);
       /* Note that the value assigned to the parameter in this line
        * has no effect. The actual parameter value is taken from
        * the component customizer.
        */
       parameter sequence = 0;

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

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

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

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

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

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

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

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

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

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

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

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

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

                    default:       /* we should never get here - reset the component */
                    begin
                          state_next <= START;
                    end
              endcase
        end
        else   /* if neither 1 or 0 have been entered, stay in same state */
        begin
              state_next <= state_curr;
        end
  end

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

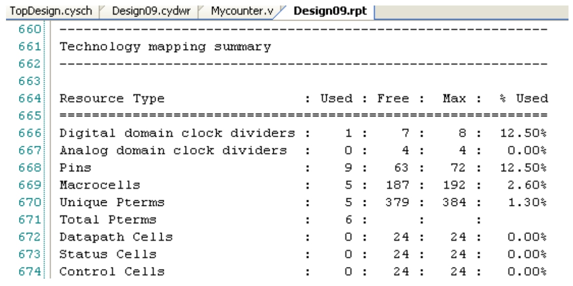
Appendix D: Post-Build Design Considerations

Project Report File

Access the project build report (<project_name>.rpt) from the **Results** tab of the **Workspace Explorer** window. It is created after a successful build. Following are the major sections in the report file.

- Technology mapping summary section – the utilization of macrocells, pterms, datapaths, pins, clock dividers, and so on is shown in Figure 31.

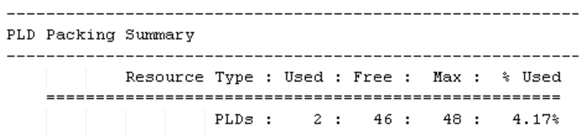
Figure 31. PSoC Creator Project Build Report File



```
------------------------------------------------------------
Technology mapping summary
------------------------------------------------------------

Resource Type                 : Used : Free :  Max :  % Used
============================================================
Digital domain clock dividers :    1 :    7 :    8 :  12.50%
Analog domain clock dividers  :    0 :    4 :    4 :   0.00%
Pins                          :    9 :   63 :   72 :  12.50%
Macrocells                    :    5 :  187 :  192 :   2.60%
Unique Pterms                 :    5 :  379 :  384 :   1.30%
Total Pterms                  :    6 :      :      :
Datapath Cells                :    0 :   24 :   24 :   0.00%
Status Cells                  :    0 :   24 :   24 :   0.00%
Control Cells                 :    0 :   24 :   24 :   0.00%
```

- Synthesis Results - Lists the errors and warnings generated at each phase of synthesis: Verilog compilation, parsing, high level synthesis, optimization etc. This section contains the details of logic which is optimized away by the synthesizer. This section can be useful for debugging and troubleshooting.
- Digital Placement :: PLD Packing Summary. Figure 32 shows an example of PLD utilization.

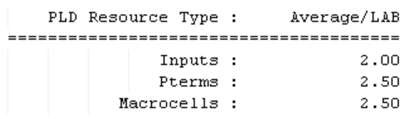
Figure 32. Example PLD Packing Summary Report



```
------------------------------------------------------------
PLD Packing Summary
------------------------------------------------------------
            Resource Type : Used : Free :  Max :  % Used
    ====================================================
                     PLDs :    2 :   46 :   48 :   4.17%
```

- Digital Placement :: PLD Packing Summary :: PLD Statistics - Figure 33 shows an example of PLD PTs and macrocells utilization in terms of average per logic array block (LAB).

Figure 33. Example PLD Usage Report



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

- Final Placement summary – Gives component details. This section of the report shows UDB utilization, occupancy, statistics, and placement (coordinate) details.

Static Timing Analysis

An important part of debugging digital designs is static timing analysis (STA). STA evaluates a digital design and calculates delays between signal outputs and inputs. From those delays it computes the maximum allowable frequency of each clock used in the design.

PSoC Creator automatically creates a static timing analysis report when you build a project. The report shows the critical paths in the design that limit the frequency of each clock. If the calculated maximum frequency is less than the required clock frequency, then a warning displays indicating that a timing violation exists in the design.

For more information on avoiding timing violations and handling PSoC Creator STA warnings, see AN81623 – PSoC 3 and PSoC 5LP Digital Design Best Practices.

Document History

Document Title: PSoC® 3, PSoC 4, and PSoC 5LP Implementing Programmable Logic Designs with Verilog

Document Number: 001-82250

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3758092	VJYA	09/27/2012	New Application Note
*A	3774553	VJYA/ANTO	10/11/2012	Changed/Edited Fig1, Fig3, Fig4, Fig9 Added Verilog code for components in the Appendix Minor edits throughout the document Updated component versions to 1.10 Added Verilog Synthesis sub-section to the HDL Coding Guidelines section of the appendix
*B	3811941	VJYA/ANTO	11/14/2012	Changed Title Modified abstract Deleted appendices C-F, section on build settings Added appendix A in this document. Moved section on project report file and STA to appendix D. Optimized code for sequence detector Enhanced figures 2,3,4, 14, 15, 16, 25, 26; added figures 5,6, 23 Updated for PSoC 5LP Minor modifications throughout the document
*C	3841114	VJYA/ANTO	12/13/2012	Added Appendix A – comparison of PSoC PLD resources with similar-size CPLDs Moved Appendix containing counter Verilog code to main body of text (Code 1)
*D	3943324	ANTO	03/25/2013	Updated for PSoC 4 and PSoC Creator 2.2 SP1 Added references to KB articles esp. Just Enough Verilog for PSoC. Minor changes throughout the document

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5LP | PSoC 4

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

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