Contents
1 Introduction...........................................................1
2 PSoC UDBs..........................................................2
  2.1 Architecture of PLDs in PSoC UDB...........................2
3 PSoC Creator........................................................4
4 Example Project .....................................................6
  4.1 Create Verilog Component: Counter4Bit...............8
  4.2 Create Verilog Component: SeqDetector............15
5 Datapath vs. PLD-based Designs..............................19
6 Summary............................................................19
6.1 Additional Information.........................................19
7 Related Resources................................................20
   A Appendix A: PSoC PLD Resource Comparison with Competitive CPLDs.................................21
   B Appendix B: Macrocell Configuration Diagrams........22
   C Appendix C: Sequence Detector Verilog Code..........23
   D Appendix D: Post-Build Design Considerations........26
   Worldwide Sales and Design Support.........................28

1 Introduction

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

Note: This application note does not apply to devices that do not have UDBs, consult the device datasheet to determine if your device has 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.

This is an advanced application note – it assumes familiarity with PSoC and PSoC Creator. If you are new to PSoC, see introductions in:

- AN54181 Getting Started with PSoC 3
- AN79953 Getting Started with PSoC 4
- AN77759 Getting Started with PSoC 5LP
- AN221774 Getting Started with PSoC 6 MCUs
If you are new to PSoC Creator, see:

- PSoC Creator home page

Note:
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.

2 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.

![UDB Block Diagram](image)

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.

2.1 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 eight PTs and four macrocells, as shown in Figure 2. 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

X = (A & B) | (~C & D)
Y = (A & B) | (C & D)

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 the Macrocell section of the UDB chapter in the Technical Reference Manual for the device you are using.
Figure 4. PSoC PLD Macrocell Architecture

XOR Feedback (XORFB)
00: D FF
01: Arithmetic (Carry)
10: T FF on high
11: T FF on low

Set Select (SSEL)
0: Set not used
1: Set from input

XORFB[1:0]  SSEL

Constant (CONST)
0: D FF true in
1: D FF inverted in

COEN
Carry Out Enable (COEN)
0: Carry Out disabled
1: Carry Out enabled

RSEL
Reset Select (RSEL)
0: Set not used
1: Set from input

BYP
Output Bypass (BYP)
0: Registered
1: Combinational

clk
To macrocell
read-only register

cpt0
cpt1

reset
selin
(set from input)

pld_en

out

From OR gate

selout
(to next MC)

3 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.
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 5. Verilog-based Component

```verilog
module SRFlopFlip (output reg q, input clk, input R, input S);
always @ (posedge clk)
begin
    case {R,S}
        2'b00: // preserve state
            q <= q;
        end
        2'b01: //Reset
            q <= 1'b0;
        end
        2'b10: //Set
            q <= 1'b1;
        end
        2'b11: //illegal state
            q <= q;
        end
        default: // never get here
            q <= q;
    endcase
endmodule
```

Figure 6. Digital Components in PSoC Creator
4 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 5LP on the CY8CKIT-001 PSoC development kit (DVK) 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

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. 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 (see 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; otherwise, it 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 (see Figure 10).
- The 'restart' and 'detect' signals control the output mux to drive four LEDs based on the state diagram in Figure 8.

Figure 8. State Diagram for Example Project
4.1 Create Verilog Component: Counter4Bit

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

4.1.1 4-bit Counter Component Creation Steps

You can 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.
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 Create Project dialog, select Library Project under Design project, and click Next. In the next window, select the CPU cores this library needs to be built for. Lastly, leave the location as default and change the Project name: to “MyLibrary.

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 (see Figure 11).
   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 11. Creating a Custom Component

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 shown in Figure 12.

Figure 12. Symbol Creation Wizard for Count4Bit

Click OK to generate the symbol in the symbol schematic, as shown in Figure 13.
You can resize the component, and modify the appearance of the component, as shown in Figure 14.  

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 15.  

9. Enter Community/Digital/Logic/Counter 4-bit in the **Catalog Placement** dialog, as shown in Figure 16.  
   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’.

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** (see Figure 17).
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 (see Figure 10).

Figure 17. Symbol Parameters for Count4Bit

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

Set a validator for the parameter by clicking 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 18. Click **OK** to make the changes.

Figure 18. Adding Validators for Count4Bit

12. In the **Parameter Definition** dialog box, set the **Hardware** field to **True**, as shown in Figure 19. This is necessary to pass the parameter to the Verilog file.
The next step is to link the schematic symbol to a Verilog file. PSoC Creator generates a Verilog shell based on the component symbol.

13. To do this, right-click on empty space in the symbol schematic and then click **Generate Verilog**. Retain the default settings in the **Generate Verilog** dialog box and click **Generate**, as shown in **Figure 20**.

**Figure 19. Passing Parameter to Hardware**

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

**Figure 20. Generate the Verilog File for the Symbol**

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**.

### 4.1.2 Verilog Design: 4-bit Counter

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

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

```verilog
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’.

```verilog
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.

```verilog
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.

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

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

```verilog
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

```verilog
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
        end

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

endmodule
```

To use this component in the ‘MyComponents’ project, you need to set ‘MyLibrary’ as a dependency. To do this, right-click MyComponents in the Source tab, and select Dependencies. Ensure that the checkbox for ‘MyLibrary’ under User Dependencies is checked, as shown in Figure 21.
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.

### 4.2 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.

#### 4.2.1 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 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 22.
5. Click **OK** to generate the symbol in the symbol schematic, as shown in Figure 23.

![Figure 23. Sequence Detector Initial Symbol](image)

You can resize the component as shown in Figure 24.

![Figure 24. Final Symbol for Sequence Detector](image)

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. 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 25.
8. Enter a description for this parameter by clicking 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 26. Click OK to make the changes.

10. When you return to the Parameter Definition dialog box, set the Hardware field to True.

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

   The next step is to link the schematic symbol to a Verilog file.

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.

### 4.2.2 Verilog Design: SeqDetector

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

```verilog
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 block 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, follow the steps described in the counter section.
5 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource Consumption in PLDs Only</th>
<th>Resource Consumption in Datapaths Only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PLDs</td>
<td>% Used</td>
</tr>
<tr>
<td>ADD8</td>
<td>5</td>
<td>10.4%</td>
</tr>
<tr>
<td>SUB8</td>
<td>5</td>
<td>10.4%</td>
</tr>
<tr>
<td>CMP8</td>
<td>3</td>
<td>6.3%</td>
</tr>
<tr>
<td>SHIFT8</td>
<td>3</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

*Note* The percentages are calculated from a device with 24 UDBs.

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.

6 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, read AN82156.

6.1 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.
7 Related Resources

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
AN221774 Getting Started with PSoC 6 MCUs

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.

Name: Antonio Rohit De Lima Fernandes
Title: Applications Engineer
Background: B.E in EE, BITS, Pilani, Rajasthan, India.
A  Appendix A: PSoC PLD Resource Comparison with Competitive CPLDs

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

Table 1. PSoC PLD Macrocell Comparison with Competitive PLDs

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

* Altera MAX-II is not traditional product term architecture
Appendix B: Macrocell Configuration Diagrams

Figure 27 and Figure 28 show the data flow through the macrocell for D flip-flop (D-FF) and T flip-flop (T-FF) functionality, respectively.

Figure 27. Macrocell with D-FF Function Enabled

Figure 28. Macrocell with T-FF Function Enabled
Appendix C: Sequence Detector Verilog Code

```verilog
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 */
      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 START state, 0 otherwise*/
if (state_next == START)
  begin
    restart <= 1'b1;
  end
else
  begin
    restart <= 1'b0;
  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 */
              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
        end
      endcase
    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 beginning 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

endmodule
Appendix D: Post-Build Design Considerations

D.1 Project Report File

Access the project build report (*.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 29.

![Figure 29. PSoC Creator Project Build Report File](image)

- Synthesis Results - Lists the errors and warnings generated at each phase of synthesis: Verilog compilation, parsing, high level synthesis, optimization, and so on. This section contains the details of logic which is optimized away by the synthesizer. This section is useful for debugging and troubleshooting.

- Digital Placement - PLD Packing Summary. Figure 30 shows an example of PLD utilization.

![Figure 30. Example PLD Packing Summary Report](image)

- Digital Placement: PLD Packing Summary: PLD Statistics - Figure 31 shows an example of PLD PTs and macrocells utilization in terms of average per logic array block (LAB).

![Figure 31. Example PLD Usage Report](image)

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

D.2 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: AN82250 - PSoC Creator - Implementing Programmable Logic Designs with Verilog
Document Number: 001-82250

<table>
<thead>
<tr>
<th>Revision</th>
<th>ECN</th>
<th>Orig. of Change</th>
<th>Submission Date</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>3758092</td>
<td>VJYA</td>
<td>09/27/2012</td>
<td>New Application Note</td>
</tr>
</tbody>
</table>
| 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 |
| E        | 4514150 | TDU             | 09/25/2014      | Updated the associated project to PSoC Creator 3.0 SP1  
|          |         |                 |                 | Updated Figure 7  
| F        | 4541956 | RLOS            | 11/06/2014      | Updated the Software Version on page 1 to PSoC Creator 3.0 SP1 or later.  
|          |         |                 |                 | Updated the associated project files. |
| G        | 4848733 | TDU             | 10/06/2015      | Updated to new template.  
|          |         |                 |                 | Added a note in Step 1 and Step 4 of 4-bit Counter Component Creation Steps.  
|          |         |                 |                 | Added a note in Datapath vs. PLD-based Designs. |
| H        | 5068178 | RNJT            | 12/30/2015      | Updated the associated project to PSoC Creator 3.3 SP1 |
| I        | 5701881 | BENV            | 04/19/2017      | Updated logo and copyright |
| J        | 6385371 | TDU             | 12/05/2018      | Adding PSoC 6 MCU, Updating Images for Creator 4.2 |
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

<table>
<thead>
<tr>
<th>Category</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm® Cortex® Microcontrollers</td>
<td>cypress.com/arm</td>
</tr>
<tr>
<td>Automotive</td>
<td>cypress.com/automotive</td>
</tr>
<tr>
<td>Clocks &amp; Buffers</td>
<td>cypress.com/clocks</td>
</tr>
<tr>
<td>Interface</td>
<td>cypress.com/interface</td>
</tr>
<tr>
<td>Internet of Things</td>
<td>cypress.com/iot</td>
</tr>
<tr>
<td>Memory</td>
<td>cypress.com/memory</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>cypress.com/mcu</td>
</tr>
<tr>
<td>PSoC</td>
<td>cypress.com/psoc</td>
</tr>
<tr>
<td>Power Management ICs</td>
<td>cypress.com/pmic</td>
</tr>
<tr>
<td>Touch Sensing</td>
<td>cypress.com/touch</td>
</tr>
<tr>
<td>USB Controllers</td>
<td>cypress.com/usb</td>
</tr>
<tr>
<td>Wireless Connectivity</td>
<td>cypress.com/wireless</td>
</tr>
</tbody>
</table>

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

© Cypress Semiconductor Corporation, 2012-2018. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) (1) under its copyright rights in the Software for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited. TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.