AN62510

Implementing State Machines with PSoC® 3, PSoC 4, and PSoC 5LP

Author: Jaya Kathuria and Chris Keeser
Associated Project: Yes
Associated Part Family: PSoC 3, PSoC 4, PSoC 5
Software Version: PSoC Creator™ 4.0 SP1 and higher
Related Application Notes: None

This application note explains the method to implement state machines using the PSoC® 3/PSoC 4/PSoC 5LP family of devices. Mealy and Moore state machine implementations are shown with associated projects.

Contents

Introduction 1
Designing State Machines with PSoC 3/PSoC 4/PSoC 5LP 1
PSoC Creator Look Up Table (LUT) Component – Brief Description 1
Moore State Machine Implementation 2
Implementing Moore State Machine using Single LUT Component 2
Method 1 3
Mapping the State Machine into LUT 3
Testing the Example Project: 8
Comparison of Method 1, Method 2, and Method 3 9
Example 2 – Method 4: Moore State Machine 11
Mealy State Machine Implementation 13
Example 3: Rising Edge Detector – Mealy Implementation 14
Summary 16
Document History 17

Introduction

State machines are commonly used to implement decision making algorithms. State machines are used in applications where distinguishable states exist. A finite state machine (FSM) is based on the idea that a given system has a finite number of states. There are two types of FSMs (Mealy and Moore) that are distinguished by their output generation:

- A Mealy machine has outputs that depend on the state and the input.
- A Moore machine has outputs that depend only on the state.

This application note shows you how to implement both Mealy and Moore state machines using the lookup table (LUT) component in PSoC Creator with the PSoC 3/PSoC 4/PSoC 5LP family of devices. Example projects are included.

Designing State Machines with PSoC 3/PSoC 4/PSoC 5LP

PSoC Creator Look Up Table (LUT) Component – Brief Description

You can use the LUT in applications where a particular input combination generates a specific set of outputs; that is, you can use the LUT to perform any logic function. The LUT can have five inputs and eight outputs.

You can create state machines with a LUT by “registering the outputs” and routing some of the outputs back to the inputs.

Note that, the LUT is implemented using Universal Digital Blocks (UDB). There are PSoC devices which do not have a UDB and will not be able to implement the LUT. Please refer the respective device datasheet for details.

Moore State Machine Implementation

This section guides you through a step-by-step procedure on how to implement Moore state machines using one or more LUT components.

Implementing Moore State Machine using Single LUT Component

In a Moore state machine, the output is the function of the present state. A rising edge detector is shown as an example here. The rising edge detector produces a single-cycle pulse each time its input goes high (a rising edge is detected). Figure 1 shows the implementation of the Rising Edge Detector using a Moore state machine.

Figure 1. Rising Edge Detector – Moore State Machine

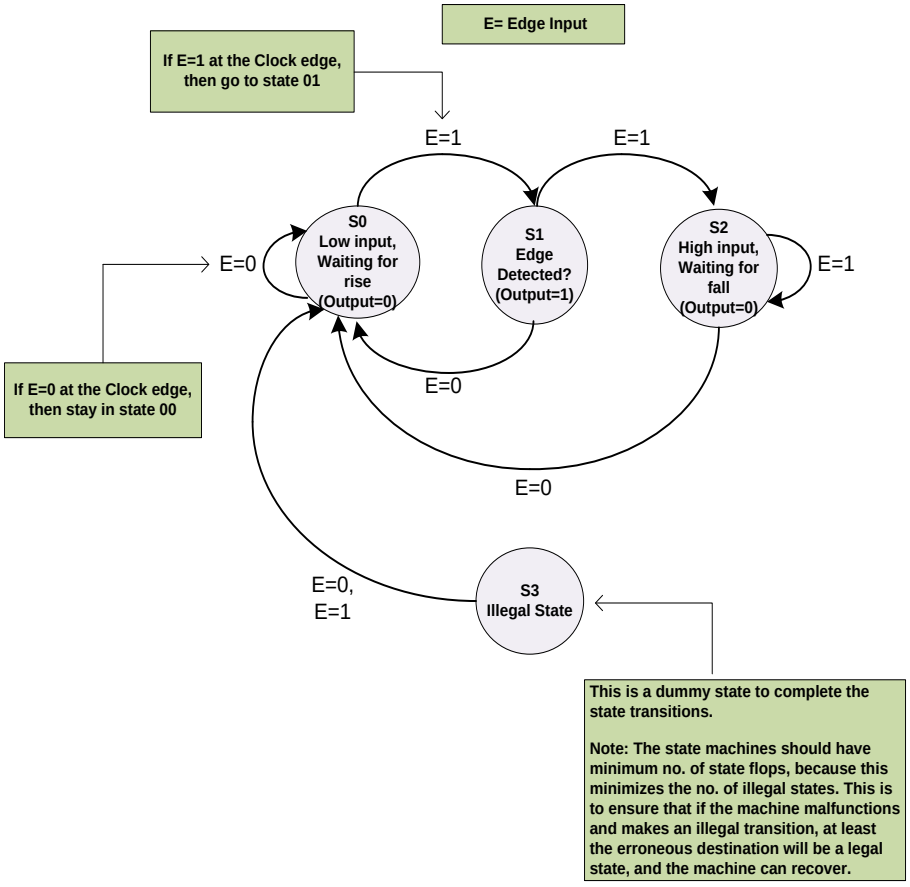


Table 1. State Description of Moore State Machine

State (Binary Value)	Description
S0 (00)	Low input, waiting for the rise
S1 (01)	Is Edge detected?
S2 (10)	High input, waiting for fall
S3 (11)	This is dummy state to complete the state transitions.

Note The state machines should have a minimum number of state flops, because this minimizes the number of illegal states. This ensures that if the machine malfunctions and makes an illegal transition, at least the erroneous destination will be a legal state, and the machine can recover.

Method 1

Mapping the State Machine into LUT

The following step-by-step procedure shows you how to map a Moore state machine into a single LUT component.

1. Generate a table with the state value and all possible combinations of the input.

Table 2. State Value

Present State	Input: E=?
[00]	0
[00]	1
[01]	0
[01]	1
[10]	0
[10]	1
[11]	Don't Care

2. Fill out the appropriate next state and desired output. Also fill the 'Don't Care' values with the output values of the next state condition, regardless of its input value.

Table 3. State Transition Table for Moore State Machine

Present State	Input: Edge (E)=?		Next State	Output: Pulse (P)=?
[00]	0	→	[00]	0
[00]	1	→	[01]	0
[01]	0	→	[00]	1
[01]	1	→	[10]	1
[10]	0	→	[00]	0
[10]	1	→	[10]	0
[11]	0	→	[00]	0
[11]	1	→	[00]	0

3. From Table 3 you can directly fill in the LUT entries if you assign 'Present State' to input [2:1], 'E' to input [0], 'Next State' to output [2:1], and 'P' to output [0]. Table 4 is derived from reformatting Table 3 with the designations.

Table 4. Table 3 with Inputs and Outputs Designations for LUT Component

Input[2:1]	Input[0]		Output[2:1]	Output[0]
[00]	0	→	[00]	0
[00]	1	→	[01]	0
[01]	0	→	[00]	1
[01]	1	→	[10]	1
[10]	0	→	[00]	0
[10]	1	→	[10]	0
[11]	0	→	[00]	0
[11]	1	→	[00]	0

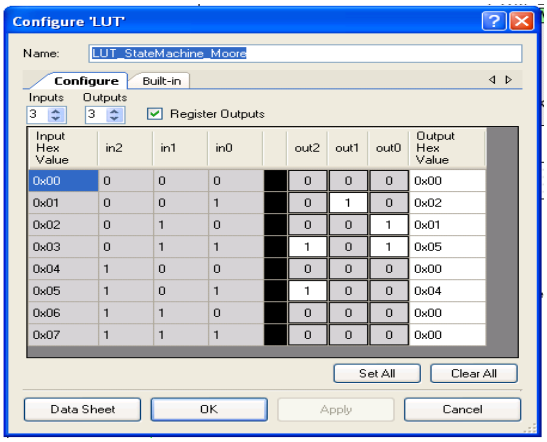
4. Create a final condensed form of the table.

Table 5. Condensed Form of Table 4

LUT Input	LUT Output
[000]	[000]
[001]	[010]
[010]	[001]
[011]	[101]
[100]	[000]
[101]	[100]
[110]	[000]
[111]	[000]

Figure 2 shows the final entries in the LUT component.

Figure 2. LUT Entries



For implementation details of the design, refer to the example project named *EdgeDetector_Moore_SingleLUT_Method1* provided with this application note. Figure 3 shows the top design of the project for PSoC 3 and PSoC 5LP.

Figure 3. PSoC Creator Top Design for PSoC 3 and PSoC 5LP project – Moore implementation using a single LUT (Method 1)

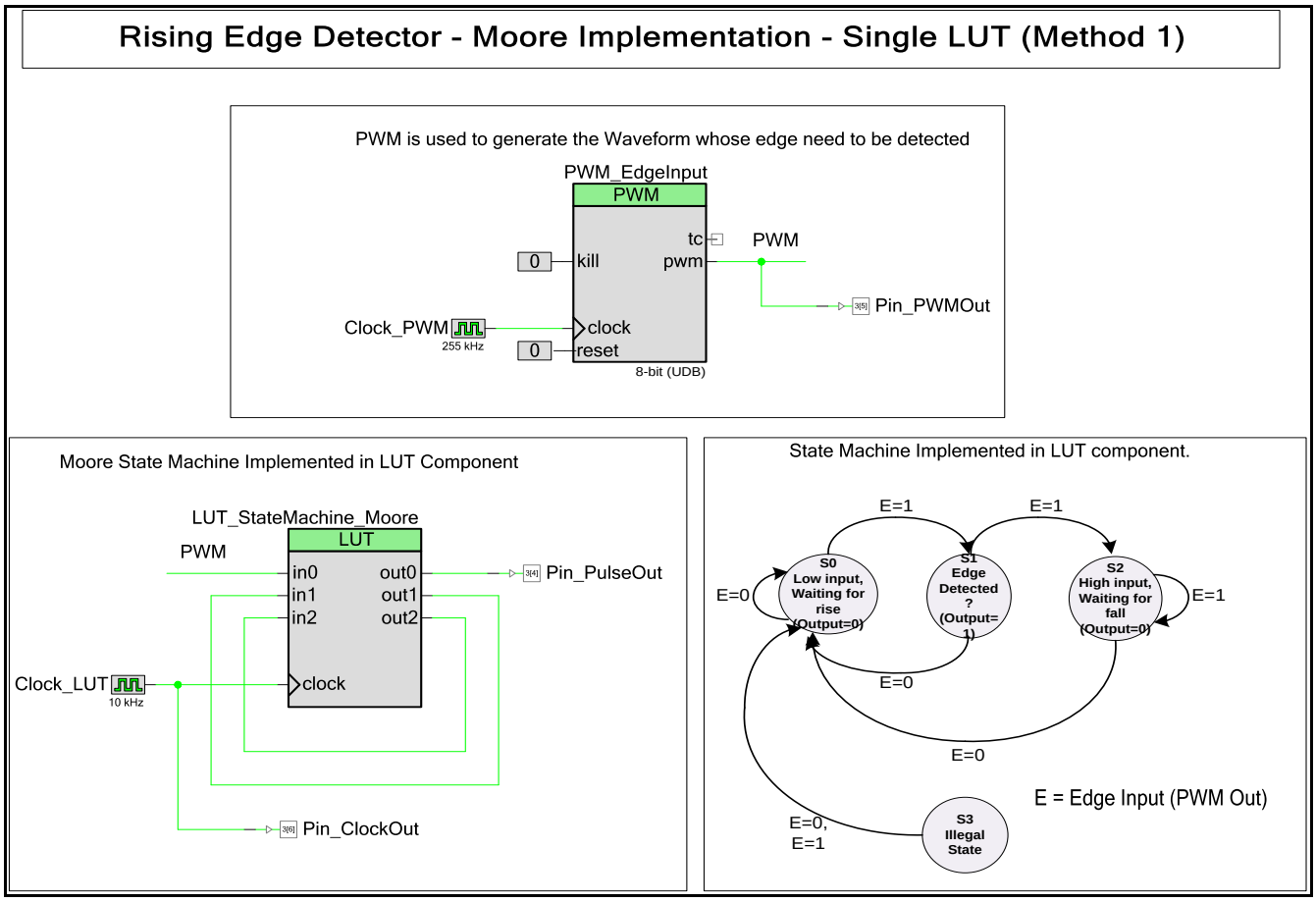
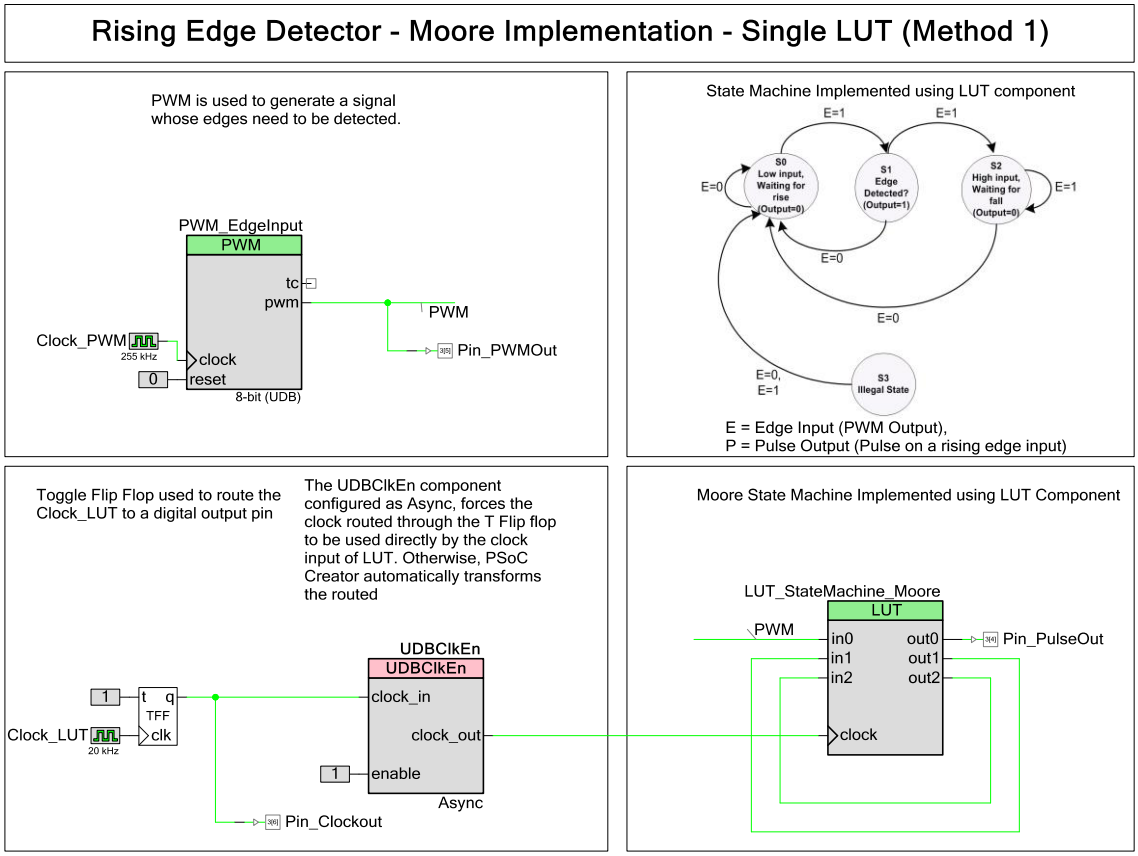


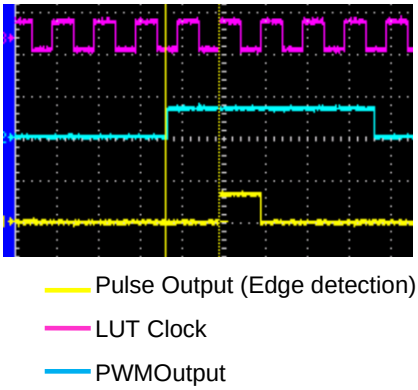
Figure 4 shows the top design of the project for PSoC 4. This differs from the top design of the PSoC 3 or PSoC 5LP project. In PSoC 4, to bring out the Clock_LUT on a digital output pin, it has to be routed through a UDB component (Toggle flip-flop is used for this purpose). The output of the Toggle flip-flop is given as the clock input to the LUT. This becomes a routed clock. By default, PSoC Creator transforms the routed clock circuitry into one which uses the Global Clock. To override this implementation, a UDBClkEn component is used. This component, when configured in Async mode, forces the LUT component to directly use the routed clock. For more information, please refer to the Routed Clock Implementation section of the System Reference Guide (*Help* > *Documentation* > *System Reference* within the PSoC Creator software tool).

Figure 4. PSoC Creator Top Design for PSoC 4 project – Moore implementation using a single LUT (Method 1)



When the Moore state machine is implemented in the LUT component and the registered option is selected, the output passes through a flip-flop. This will result in a delay of one clock cycle to obtain the output. This is shown in Figure 5.

Figure 5: Pulse Output for Edge Detection – Delay of One Clock Cycle

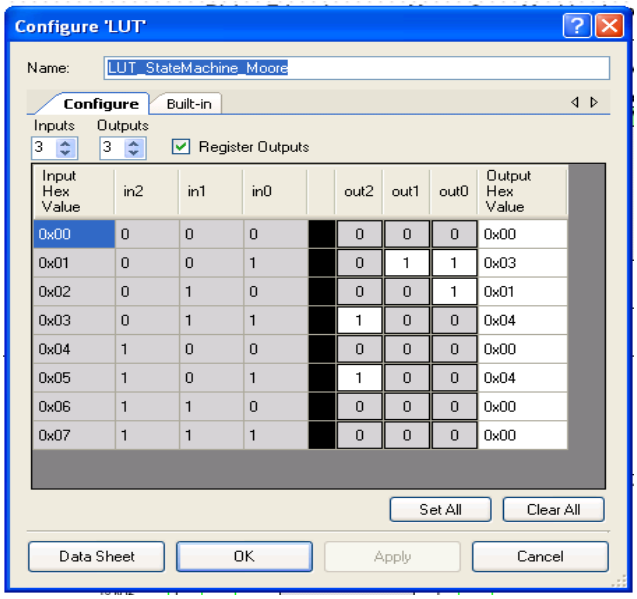


There are several ways for avoiding the one cycle delay encountered in Method 1:

1. The LUT table can be filled out so that the outputs are decoded to be one step ahead of the present state. After they are registered, they are in sync with the present state. This is explained in Example 1 – Method 2.
2. Use two LUTs – one to implement the next state logic and the other to implement the output logic. This is explained in detail in the Example 1 – Method 3.
3. Using one or more Next states as the output for the state machine. This is explained in detail in Example 2 – Method 4.

Example 1 – Method 2: Modifying the LUT entries in Method 1 so that the output is decoded to be one step ahead of the present state. After the output is registered, they are in sync with the present state and therefore, there will not be any delay in the output pulse. Figure 6 shows the modified LUT entries:

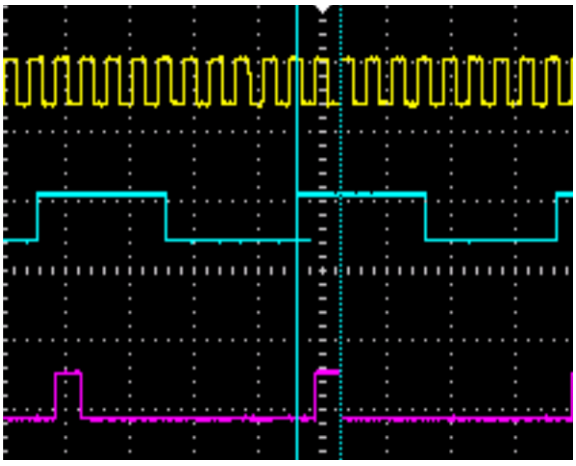
Figure 6. LUT Entries



The outputs are decoded one step ahead of the present state.

Figure 7 shows the pulse output for the rising edge detection, without one cycle delay.

Figure 7. Edge Output without one cycle delay



For implementation details of the design, refer to the example project named *EdgeDetector_Moore_SingleLUT_Method2* provided with this application note.

Example 1 – Method 3: Rising Edge Detector – Moore Implementation Using Two LUT Components

An advantage of using a second LUT for implementing the output logic is that you do not have to share the outputs with the present state register. By using two LUTs, the design can take the advantage of using the maximum number of outputs (8, according to the LUT component in PSoC 3/PSoC 4/PSoC 5LP). The aim of this method is same as Method 1 and Method 2, but here instead of a single LUT component, two LUT components are used.

Table 6. State Transition Table for Moore State Machine

Present State	Input: Edge (E)=?		Next State	Output: Pulse (P)=?
[00]	0	→	[00]	0
[00]	1	→	[01]	0
[01]	0	→	[00]	1
[01]	1	→	[10]	1
[10]	0	→	[00]	0
[10]	1	→	[10]	0
[11]	0	→	[00]	0
[11]	1	→	[00]	0

The first LUT is the state machine which has the next state logic and present state register implemented.

The table is created with Input [2:1] as the present state input, Input [0] as Edge input, and Output [1:0] will be the next state entries as shown in Table 7.

Table 7. First (Next State Logic) LUT Component Entries

Input[2:0] (Present State[2:1]:Edge Input[0])	Output[1:0] (Next State)
[000]	[00]
[001]	[01]
[010]	[00]
[011]	[10]
[100]	[00]
[101]	[10]
[110]	[00]
[111]	[00]

The second LUT is filled with output logic that depends on the present state. Input [1:0] is the Present state and Output [0] is pulse output for the rising edge detection.

Note The second LUT must not be registered because no feedback is required.

Table 8. Second (Output Logic) LUT Component Entries

Input[1:0] (Present State)	Output[0] (Pulse Output)
[00]	[0]
[01]	[1]
[10]	[0]
[11]	[0]

Figure 8 and Figure 9 show the configuration of the LUT component in PSoC Creator:

Figure 8. First (Next State Logic) LUT Entries

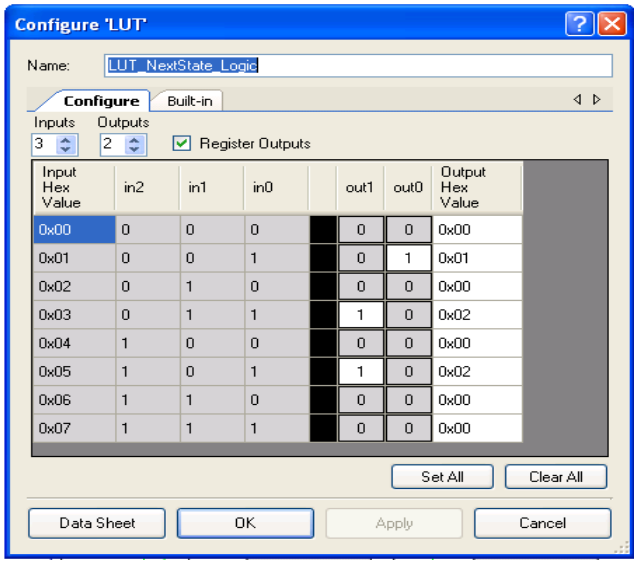
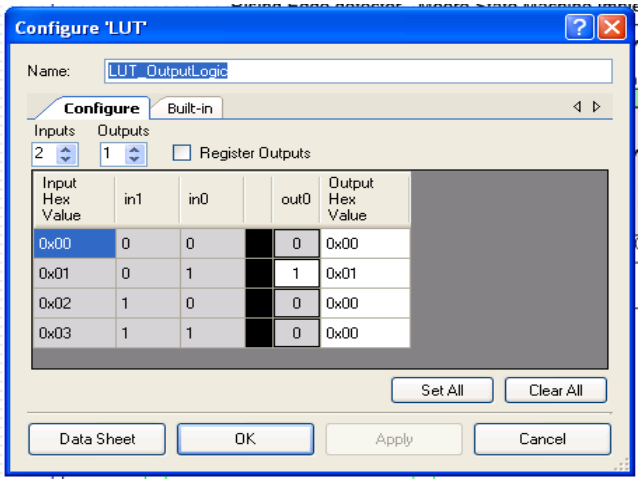


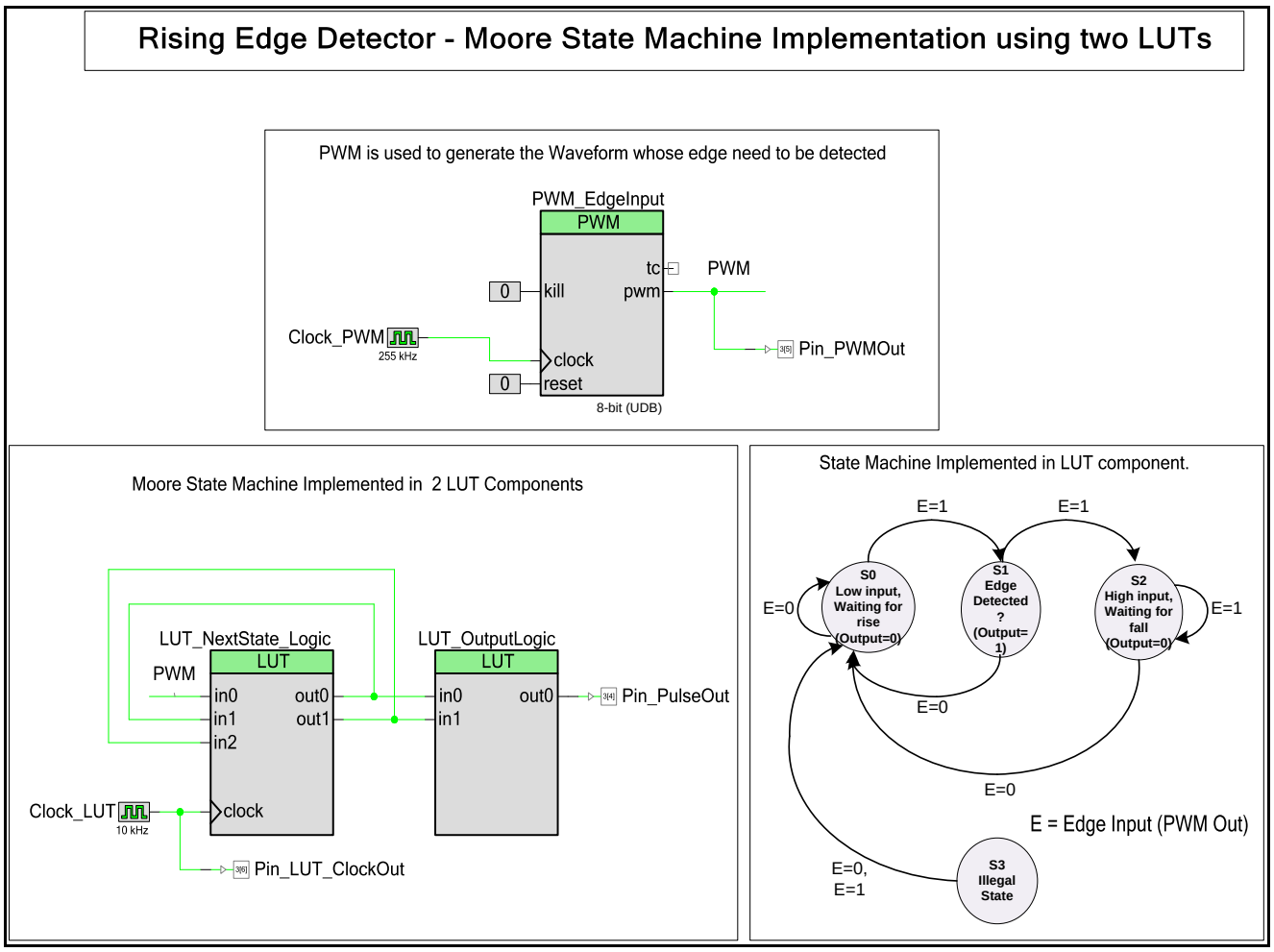
Figure 9. Second (Output) LUT Entries



Note Only the next state logic LUT is registered and the Output logic LUT component is unregistered.

For implementation details of the design, refer to the example project named *EdgeDetector_Moore_2LUT_Method2* provided with this application note. Figure 10 is provided for reference.

Figure 10. PSoC Creator Top Design for PSoC 3 and PSoC 5LP project – Moore Implementation using Two LUTs



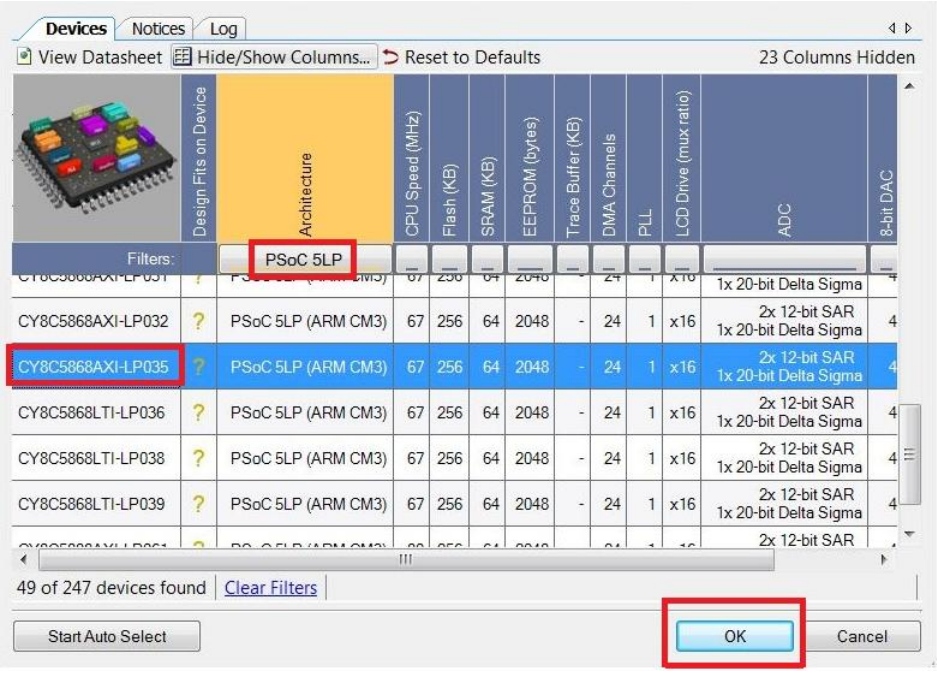
Testing the Example Project:

1. Open the project (*AN62510.zip* is provided with this application note. It contains separate projects for PSoC 4 and PSoC 3/5LP).
2. Build and program the PSoC 3 device (or PSoC 4 device).

Note In the PSoC 3 / 5LP project, the default device selection is PSoC 3 (CY8C3866AXI-040). To use this project with the PSoC 5LP family, follow this step:

Go to **Project → Device Selector →** Select **PSoC 5LP** device (CY8C5868AXI-LP035), build the project again and program the PSoC 5LP device as shown in Figure 11.

Figure 11. PSoC Creator Device Selector



3. Reset the device by pressing the **Reset** button on the DVK (SW4 on CY8CKIT-001 and SW1 on CY8CKIT-030 and CY8CKIT-050).
4. Observe the Pulse output at P3[4], PWM output at P3[5], and the clock for the LUT component at P3[6].

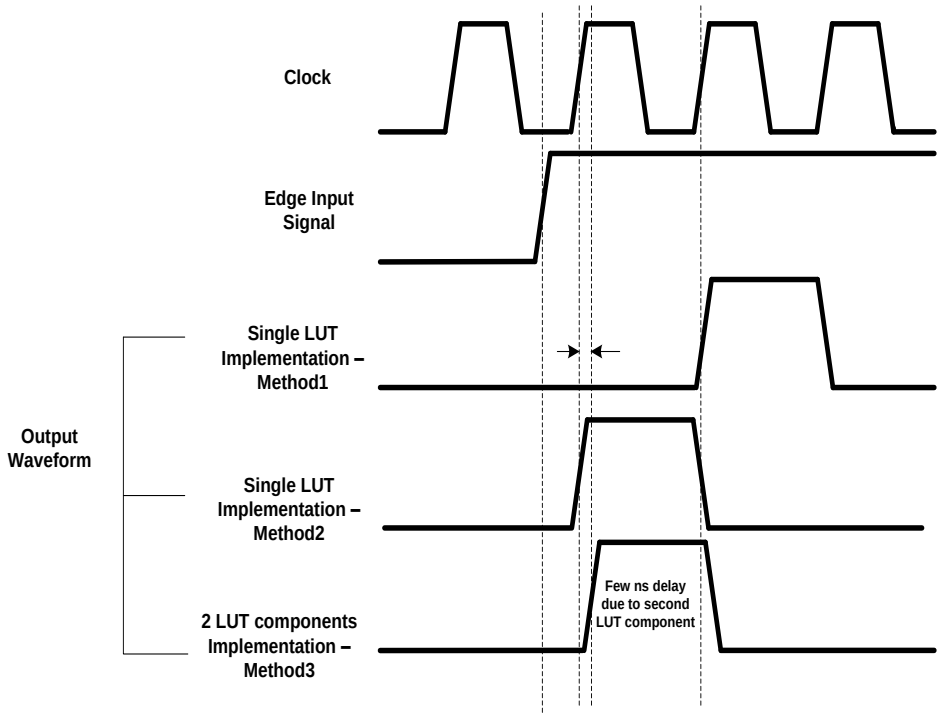
Note PWM is used in the design to generate the waveform whose rising edges need to be detected.

Comparison of Method 1, Method 2, and Method 3

Method 1, Method 2, and Method 3 give an example of implementing the same state machine in three different ways. Figure 12 gives a timing diagram comparing the three methods:

- Method 1: Implementation of the Moore state machine with a single LUT component.
- Method 2: Implementation of Moore state machine with a single LUT component and decoding the output ahead of the present state.
- Method 3: Implementation of the Moore state machine with 2 LUT components.
- Method 2 and Method 3 avoid the unexpected delay of one clock cycle, unlike Method 1.

Figure 12. Timing Diagram for Method 1, Method 2, and Method 3



The output in Method 1 is delayed by one clock because the LUT output is registered which forces the output to pass through a flip-flop. This results in one clock cycle of delay in output. This delay is avoided by using Method 2 where the output is decoded ahead of the present state and still uses the single LUT component to implement the state machine. Method 3 also avoids the delay of one clock cycle in the output, but uses an extra LUT component to produce the output. Hence Method 2 is the best way of implementing Moore state machines.

The following table lists the number of states, inputs, and outputs that is possible with a state machine implemented in single and two LUTS.

Table 9. Comparison of Number of Inputs and Outputs Possible for Single and Two LUT Component Implementations

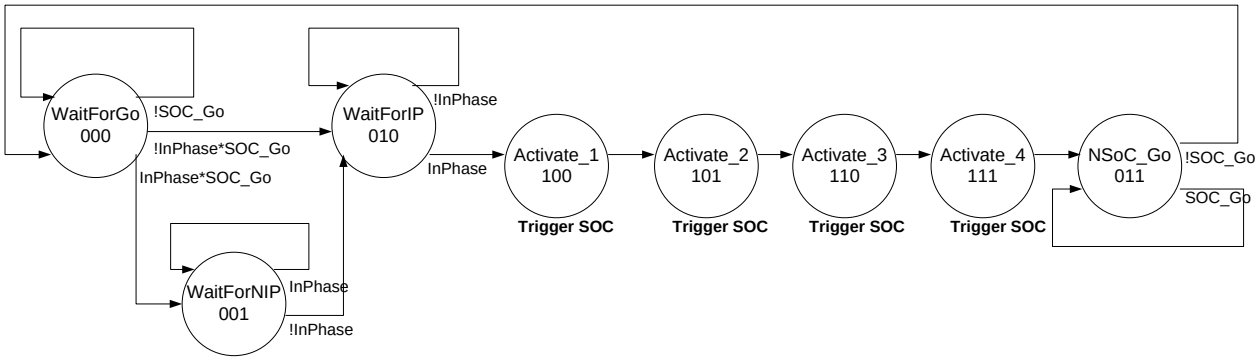
No. of States	No. of inputs possible	No. of outputs possible with single LUT Implementation	No. of outputs possible with 2 LUT Implementation
1	0-4	0-7	0-8
2	0-3	0-6	0-8
3	0-2	0-5	0-8
4	0-1	0-4	0-8
5	0	0-3	0-8

Example 2 explains how you can use one or more Next states as the output for the state machine to avoid one cycle of delay in output for the single LUT implementation for the Moore state machine implementation of Method 1.

.

Example 2 – Method 4: Moore State Machine

Figure 13. State Diagram: Example 2 – Method 4



The system described in the state machine is ‘armed’ by setting the ‘SOC_Go’ bit in a control register. When it is armed, it waits for the rising edge of the ‘InPhase’ clock and then generates a Start of Conversion (SOC) pulse that lasts for four state machine clocks. The system then waits for the ‘SOC_Go’ bit to clear before the system can be armed again. This state machine is useful if you want to synchronize ADC conversions to some reference clock, but do not need to sample on every edge of the clock.

You can simplify the output logic by selecting the right state values. Table 10 goes through each state and lists its possible inputs.

Table 10. State Transition and Output Table – Example Project 5

Current State	Input1: ‘Inphase’ Clock	Input2: ‘SOC_Go’	Next State	Output: Pulse (P)=?	Output Trigger SOC at the Next State
[000]	0	0	→	[000]	0
[000]	0	1	→	[010]	0
[000]	1	0	→	[000]	0
[000]	1	1	→	[001]	0
[001]	0	Don’t Care	→	[010]	0
[001]	1	Don’t Care	→	[001]	0
[010]	0	Don’t Care	→	[010]	0
[010]	1	Don’t Care	→	[100]	1
[011]	Don’t Care	0	→	[000]	0
[011]	Don’t Care	1	→	[011]	0
[100]	Don’t Care	Don’t Care	→	[101]	1
[101]	Don’t Care	Don’t Care	→	[110]	1
[110]	Don’t Care	Don’t Care	→	[111]	1
[111]	Don’t Care	Don’t Care	→	[011]	0

You can expand Table 10 and fill in the ‘don’t cares’ by generating the same value regardless of the value of the ‘don’t care’ input. The output ‘Trigger_SOC’ is only high when the MSB of the next state is high, so you can eliminate an output by just taking the ‘Trigger_SOC’ off of the MSB of the state output.

Table 11. State Transition and Output Table

Current State	Input1: 'Inphase' Clock	Input2: 'SOC_Go'	Next State	Output: Pulse (P)=?	Output Trigger SOC at Next State
[000]	0	0	[000]	[000]	0
[000]	0	1	[010]	[000]	0
[000]	1	0	[000]	[000]	1
[000]	1	1	[001]	[000]	1
[001]	0	0 (Don't Care)	[010]	[001]	0
[001]	0	1 (Don't Care)	[010]	[001]	0
[001]	1	0 (Don't Care)	[001]	[001]	1
[001]	1	1 (Don't Care)	[001]	[001]	1
[010]	0	0 (Don't Care)	[010]	[010]	0
[010]	0	1 (Don't Care)	[010]	[010]	0
[010]	1	0 (Don't Care)	[100]	[010]	1
[010]	1	1 (Don't Care)	[100]	[010]	1
[011]	0 (Don't Care)	0	[000]	[011]	0 (Don't Care)
[011]	0 (Don't Care)	1	[011]	[011]	0 (Don't Care)
[011]	1 (Don't Care)	0	[000]	[011]	1 (Don't Care)
[011]	1 (Don't Care)	1	[011]	[011]	1 (Don't Care)
[100]	0 (Don't Care)	0 (Don't Care)	[101]	[100]	0 (Don't Care)
[100]	0 (Don't Care)	1 (Don't Care)	[101]	[100]	0 (Don't Care)
[100]	1 (Don't Care)	0 (Don't Care)	[101]	[100]	1 (Don't Care)
[100]	1 (Don't Care)	1 (Don't Care)	[101]	[100]	1 (Don't Care)
[101]	0 (Don't Care)	0 (Don't Care)	[110]	[101]	0 (Don't Care)
[101]	0 (Don't Care)	1 (Don't Care)	[110]	[101]	0 (Don't Care)
[101]	1 (Don't Care)	0 (Don't Care)	[110]	[101]	1 (Don't Care)
[101]	1 (Don't Care)	1 (Don't Care)	[110]	[101]	1 (Don't Care)
[110]	0 (Don't Care)	0 (Don't Care)	[111]	[110]	0 (Don't Care)
[110]	0 (Don't Care)	1 (Don't Care)	[111]	[110]	0 (Don't Care)
[110]	1 (Don't Care)	0 (Don't Care)	[111]	[110]	1 (Don't Care)
[110]	1 (Don't Care)	1 (Don't Care)	[111]	[110]	1 (Don't Care)
[111]	0 (Don't Care)	0 (Don't Care)	[011]	[111]	0 (Don't Care)
[111]	0 (Don't Care)	1 (Don't Care)	[011]	[111]	0 (Don't Care)
[111]	1 (Don't Care)	0 (Don't Care)	[011]	[111]	1 (Don't Care)
[111]	1 (Don't Care)	1 (Don't Care)	[011]	[111]	1 (Don't Care)

Now you can construct the LUT input/output relationship. Use Input[4:2] as State[2:0], Input[1] as 'InPhase' and Input[0] as 'SOC_Go'. Output[2:0] is State[2:0] and Output[2] is 'Trigger_SOC'.

Figure 14. LUT Entries

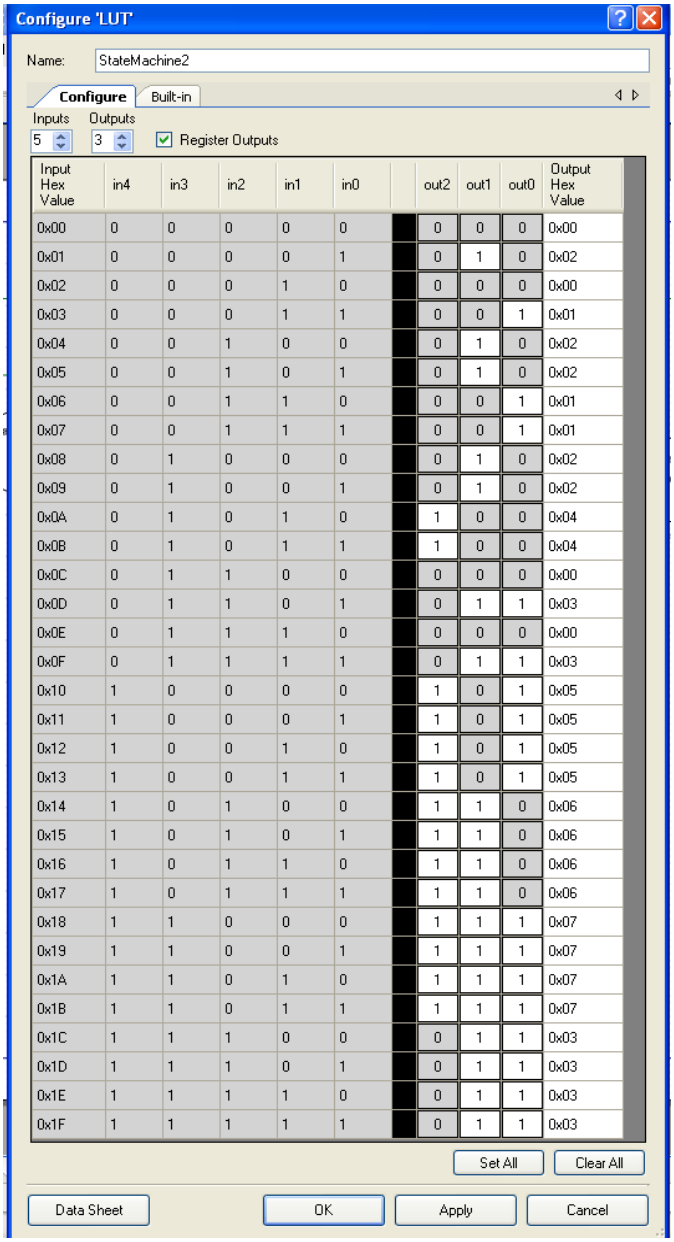
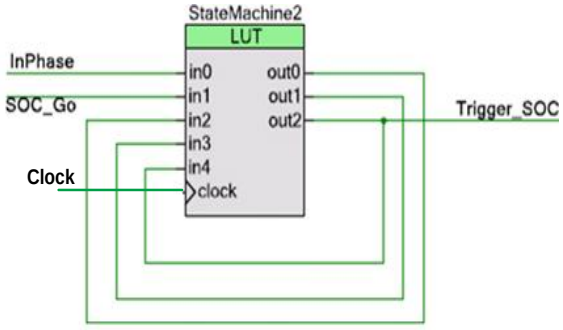


Figure 15. Component Wiring Connection



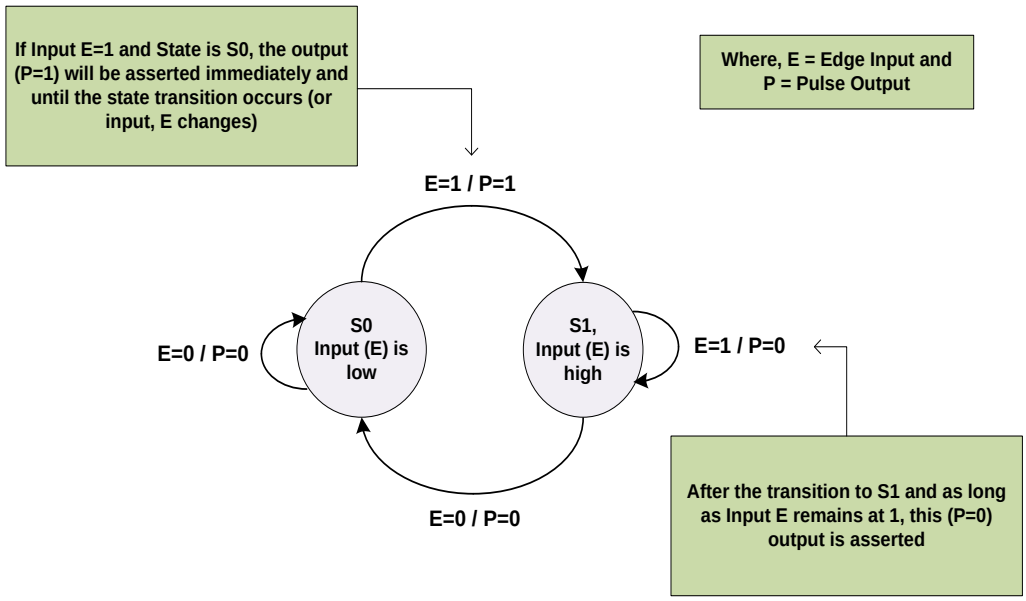
Mealy State Machine Implementation

This section guides you through a step-by-step procedure on how to implement a Mealy state machine using an LUT component.

In the Mealy state machine, the output is a function of the Present state and input. The output depends on the present state and on the input; Mealy FSM needs fewer states than Moore FSM implementation. Figure 16 shows the Mealy implementation of the Rising Edge Detector.

Example 3: Rising Edge Detector – Mealy Implementation

Figure 16. Rising Edge Detector - Mealy State Machine



Note The state S0 is given a binary value of 0 and the state S1 is given a binary value of 1.

Table 12 shows the transition and output table for the Mealy state machine.

Table 12. State Transition and Output Table for Mealy State Machine

Present State	Input (Edge)		Next State	Output Pulse (P)=?
[0]	0	→	[0]	0
[0]	1	→	[1]	1
[1]	0	→	[0]	0
[1]	1	→	[1]	0

The first LUT component is the state machine that has the next state logic and present state register implemented. Table 12 is constructed with Input [1] as the present state input and Input [0] as Edge input and Output [0] as next state entries.

Table 13. First (Next state logic) LUT Component Entries

Input[1:0] (Present State:Input)	Output[0] (Next State)
[00]	[0]
[01]	[1]
[10]	[0]
[11]	[0]

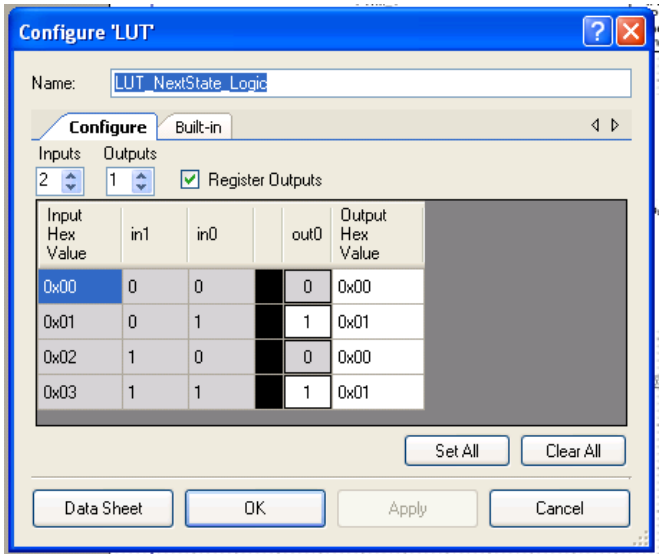
The second LUT component is filled with output logic that depends on the present state. Input [1] is the present state, Input [0] is the Edge input and Output [0] is the pulse output for the rising edge detection. This is shown in Table 14.

Table 14. Second (Output Logic) LUT Component Entries

Input[1:0] (Present State:Input)	Output[0] (Next State)
[00]	[0]
[01]	[1]
[10]	[0]
[11]	[1]

Figure 17 and Figure 18 show the configuration of the LUT components in PSoC Creator.

Figure 17. First LUT Component for Next State Logic.



Note The register output option is selected.

Figure 18. Second LUT Component for Output Logic

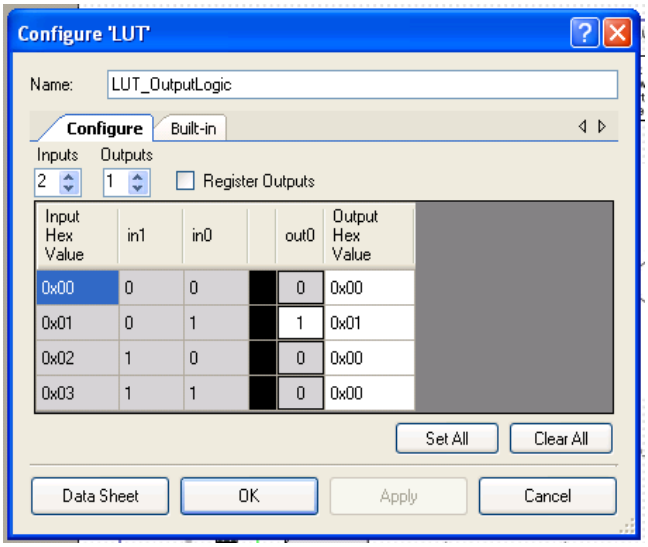
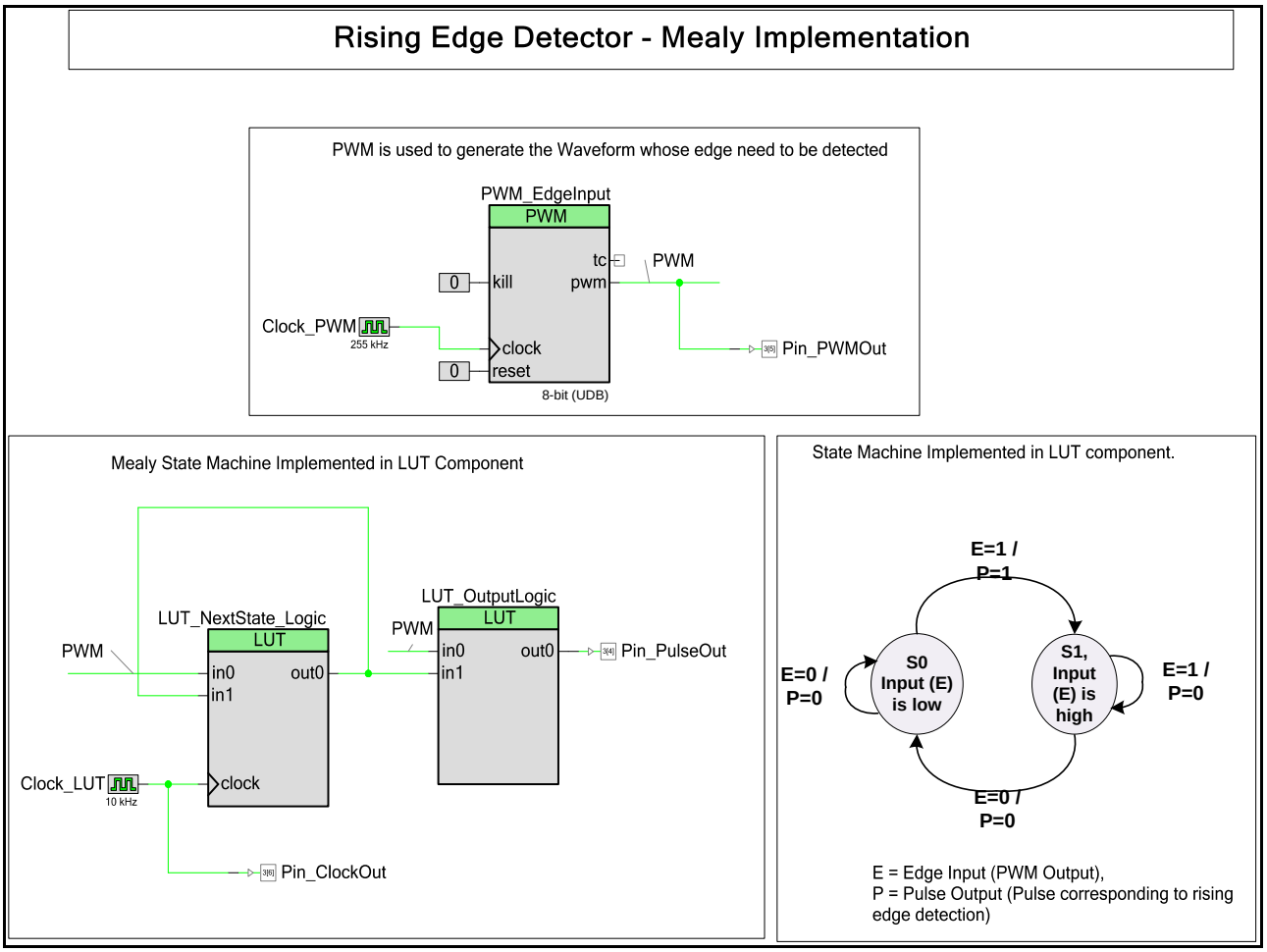


Figure 19 shows the top design of the project for PSoC 3 and PSoC 5LP. See the example project named *EdgeDetector_Mealy_StateMachine* provided with this application note for implementation details of the design.

Figure 19. PSoC Creator Top Design for PSoC 3 and PSoC 5LP project – Mealy Implementation



Summary

This application note documents how to implement the Moore and the Mealy state machines using PSoC 3/PSoC 4/PSoC 5LP LUT components with examples. It also explains the different ways of implementing the Moore state machines using only a single LUT component.

Document History

Document Title: AN62510 - Implementing State Machines with PSoC® 3, PSoC 4, and PSoC 5LP

Document Number: 001-62510

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2965620	XKJ	06/30/2010	New Application Note.
*A	3134424	XKJ	01/11/2011	Updated Software Version in page 1 as “PSoC Creator™”. Updated with FCS.
*B	3452505	DASG	12/01/2011	Updated Software Version in page 1 as “PSoC Creator™ 2.0”. The clock tolerance is changed from 5% to 10% to have the clock accuracy range within the specified tolerance range. Updated to new template.
*C	3809659	PHAL	11/26/2012	Updated Associated Part Family in page 1 as “All PSoC 3 and PSoC 5LP parts”. Updated Software Version in page 1 as “PSoC Creator™ 2.1 SP1 and higher”. Updated for PSoC 5LP.
*D	4035884	PHAL	06/21/2013	Updated Associated Part Family in page 1 as “CY8C3xxx, CY8C42xx, CY8C5xxx”. Updated Software Version in page 1 as “PSoC Creator™ 2.2 SP1 and higher”. Updated the projects and the document for PSoC 4.
*E	5568248	QVS	01/24/2017	Updated Associated Part Family in page 1 as “PSoC 3, PSoC 4, PSoC 5”. Updated Software Version in page 1 as “PSoC Creator™ 4.0 SP1 and higher”. Updated attached associated project to PSoC Creator 4.0. Updated document to reflect the PSoC Creator 4.0 support. Updated Figure 13 (for better clarity). Updated to new template. Completing Sunset Review.
*F	5732907	AESATP12	05/16/2017	Updated logo and copyright.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

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

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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

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

© Cypress Semiconductor Corporation, 2010-2017. 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, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) 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. 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.