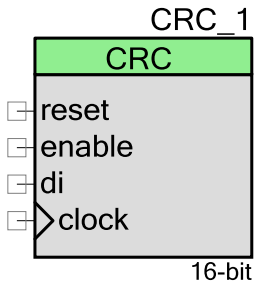
循环冗余校验 (Cyclic Redundancy Check, CRC)

2.10

特性



- 1 到 64 位
- 时分复用模式
- 需要时钟和数据以进行串行比特流输入
- 串行数据输入，并行结果
- 标准 [CRC-1（奇偶校验位）、CRC-4 (ITU-T G.704)、CRC-5-USB 等] 或自定义多项式
- 标准或自定义种子值
- 启用输入提供与其他组件的同步操作

概述

循环冗余校验 (CRC) 组件的默认用途是根据任意长度的串行比特流计算 CRC。在数据时钟的上升沿上对输入数据进行采样。在启动前，CRC 值复位为 0，或可用初始值作为种子值。完成比特流时，可读取计算出的 CRC 值。

何时使用 CRC

可使用默认 CRC 组件作为校验和，以在传输或存储过程中检测数据变化。CRC 很受欢迎，因为它们在二进制硬件中很容易实现，便于以数学方式进行分析，并特别适用于检测由传输通道中的噪声导致的常见错误。

输入/输出连接

本节介绍 CRC 的各种输入和输出连接。I/O 列表中的星号 (*) 表示，在 I/O 说明中列出的情况下，该 I/O 可能不可见。

时钟 — 输入

CRC 需要提供用于计算 CRC 的串行比特流的数据输入。同时需要数据时钟输入，以正确对串行数据输入进行采样。在数据时钟的上升沿上对输入数据进行采样。

复位 — 输入

复位输入定义信号，以对 CRC 进行异步复位。

使能 — 输入

CRC 组件启动后，在使能输入保持高电平状态的情况下一直运行。此输入提供与其他组件的同步操作。

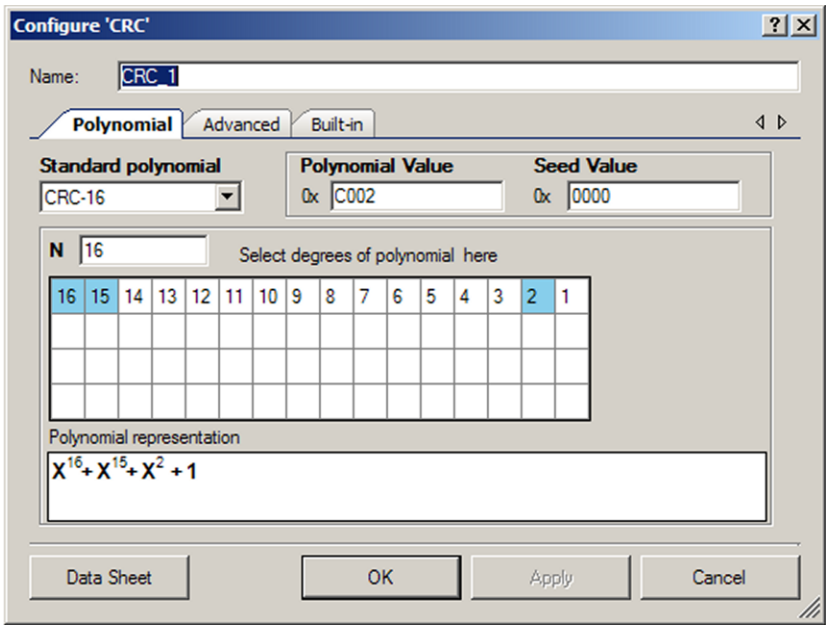
di — 输入

提供用于计算 CRC 的串行比特流的数据输入。

元件参数

将一个 CRC 组件拖放到您的设计上，并双击以打开 **Configure**（配置）对话框。该对话框有若干选项卡，可引导您完成 CRC 组件的设置过程。

“Polynomial”（多项式）选项卡



Standard Polynomial（标准多项式）

此参数允许您选择 **Standard polynomial**（标准多项式）组合框中的任意标准 CRC 多项式，或生成自定义多项式。有关各个标准多项式的更多信息在工具提示中给出。默认值为 **CRC-16**。

多项式名称	多项式	使用说明
自定义	用户定义的	一般
CRC-1	x + 1	奇偶校验
CRC-4-ITU	$x^4 + x + 1$	ITU G.704
CRC-5-ITU	$x^5 + x^4 + x^2 + 1$	ITU G.704
CRC-5-USB	$x^5 + x^2 + 1$	USB
CRC-6-ITU	$x^6 + x + 1$	ITU G.704
CRC-7	$x^7 + x^3 + 1$	电信系统，MMC
CRC-8-ATM	$x^8 + x^2 + x + 1$	ATM HEC
CRC-8-CCITT	$x^8 + x^7 + x^3 + x^2 + 1$	1- 单线总线

多项式名称	多项式	使用说明
CRC-8-Maxim	$x^8 + x^5 + x^4 + 1$	1- 单线总线
CRC-8	$x^8 + x^7 + x^6 + x^4 + x^2 + 1$	一般
CRC-8-SAE	$x^8 + x^4 + x^3 + x^2 + 1$	SAE J1850
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x + 1$	一般
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$	电信系统
CRC-15-CAN	$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$	CAN
CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$	XMODEM,X.25、V.41、Bluetooth、PPP、IrDA、CRC-CCITT
CRC-16	$x^{16} + x^{15} + x^2 + 1$	USB
CRC-24-Radix64	$x^{24} + x^{23} + x^{18} + x^{17} + x^{14} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^4 + x^3 + x + 1$	一般
CRC-32-IEEE802.3	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	以太网，MPEG2
CRC-32C	$x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1$	一般
CRC-32K	$x^{32} + x^{30} + x^{29} + x^{28} + x^{26} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x^2 + x + 1$	一般
CRC-64-ISO	$x^{64} + x^4 + x^3 + x + 1$	ISO 3309
CRC-64-ECMA	$x^{64} + x^{62} + x^{57} + x^{55} + x^{54} + x^{53} + x^{52} + x^{47} + x^{46} + x^{45} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{33} + x^{32} + x^{31} + x^{29} + x^{27} + x^{24} + x^{23} + x^{22} + x^{21} + x^{19} + x^{17} + x^{13} + x^{12} + x^{10} + x^9 + x^7 + x^4 + x + 1$	ECMA-182

Polynomial Value（多项式值）

此参数使用十六进制格式。当选择了标准多项式中的其中一个时，将自动计算此参数。也可手动输入此参数（请参见 Custom Polynomials（自定义多项式））。

Seed Value（种子值）

此参数使用十六进制格式。最大可能值为 $2^N — 1$。

N

此参数定义多项式的次数。这些值可能为 1 - 64 位。带数字的表格指示包括的多项式次数。带选定数字的单元格显示为蓝色，其他为白色。活动单元格数等于 N。数字反向排列。可单击单元格以选择或取消选择数字。

Polynomial representation（多项式表示）

此参数以数学符号显示结果多项式。

Custom Polynomials（自定义多项式）

可通过三种不同的方法输入自定义多项式：

对标准多项式进行少量更改

- 选择标准多项式中的一个。
- 通过单击相应的单元格在表中选择必要的次数；**Standard polynomial**（标准多项式）中的文本更改为 **Custom**（自定义）。
- 根据显示的多项式自动重新计算多项式值。

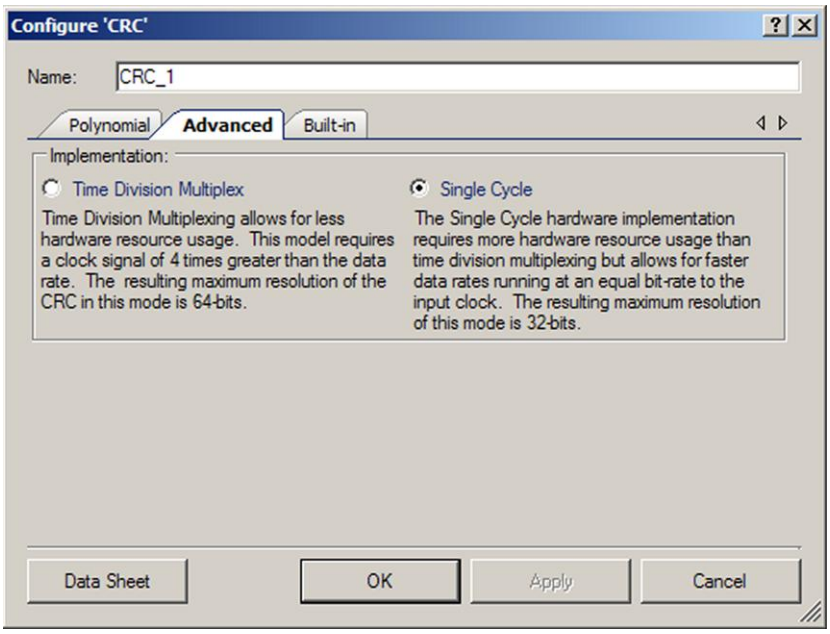
使用多项式次数

- 在 **N** 文本框中输入自定义多项式；**Standard polynomial**（标准多项式）中的文本更改为 **Custom**（自定义）。
- 通过单击相应的单元格在表中选择必要的次数。
- 在 **Polynomial representation**（多项式表示）中检查多项式视图。
- 根据显示的多项式自动重新计算多项式值。

使用十六进制格式

- 在 **Polynomial Value**（多项式值）文本框中输入十六进制格式的多项式值。
- 按 **[Enter]** 或切换到另一个控件；**Standard polynomial**（标准多项式）中的文本更改为 **Custom**（自定义）。
- 基于输入的多项式值重新计算 N 值和多项式次数。

“高级”选项卡



实现

此参数定义 CRC 组件的实现：**Time Division Multiplex**（时分复用）或 **Single Cycle**（单周期）。默认值为 **Single Cycle**（单周期）。

本地参数（供 API 使用）

以下参数用于 API，不在 GUI 中使用：

- **PolyValueLower (uint32) —** 包含十六进制格式的多项式值的下半部分。默认值为 0xB8h (LFSR= [8,6,5,4])，因为默认分辨率为 8。
- **PolyValueUpper (uint32) —** 包含十六进制格式的多项式值的上半部分。默认值为 0x00h，因为默认分辨率为 8。
- **SeedValueLower (uint32) —** 包含十六进制格式的种子值的下半部分。默认值为 0xFFh，因为默认分辨率为 8。
- **SeedValueUpper (uint32) —** 包含十六进制格式的种子值的上半部分。默认值为 0，因为默认分辨率为 8。

时钟选择

此组件中没有内部时钟。您必须附加时钟源。

注意：如果针对 **Implementation**（实现）参数选择了 **Time Division Multiplex**（时分复用），当分辨率大于 8 时，要生成正确的 CRC 序列，时钟信号必须比数据速率大 4 倍。

放置

CRC 放置于整个 UDB 阵列中，并且所有放置信息通过 *cyfitter.h* 文件提供给 API。

资源

单周期实现

资源	资源类型			API Memory（API 存储器）（字节）		Pins（引脚）（每个外部 I/O）
	Data Path 单元	PLD	Control/Count7 单元	Flash（闪存）	RAM	
1 到 8 位分辨率	1	1	1	166	2	4
9 到 16 位分辨率	2	1	1	210	2	4
17 到 24 位分辨率	3	1	1	287	2	4
25 到 32 位分辨率	4	1	1	288	2	4

时分复用实现

资源	资源类型			API Memory（API 存储器）（字节）		Pins（引脚）（每个外部 I/O）
	Data Path 单元	PLD	Control/Count 7 单元	Flash（闪存）	RAM	
9 到 16 位分辨率	1	3	1	242	2	4
17 到 24 位分辨率	2	3	1	538	2	4
25 到 32 位分辨率	2	3	1	615	2	4
33 到 40 位分辨率	3	3	1	763	2	4
41 到 48 位分辨率	3	3	1	894	2	4
49 到 56 位分辨率	4	3	1	999	2	4

57 到 64 位分辨率	4	3	1	1101	2	4

应用程序编程接口

应用程序编程接口 (API) 子程序允许您使用软件配置组件。下表列出了每个函数的接口，并进行了说明。以下各节将更详细地介绍每个函数。

默认情况下，PSoC Creator 将实例名称“CRC_1”分配给指定设计中组件的第一个实例。您可以将其重命名为遵循标识符语法规则的任何唯一值。实例名称会成为每个全局函数名称、变量和常量符号的前缀。出于可读性考虑，下表中使用的实例名称为“CRC”。

函数	说明
CRC_Start()	用初始值初始化种子和多项式寄存器。在输入时钟的上升沿上开始计算 CRC。
CRC_Stop()	停止 CRC 计算。
CRC_Wakeup()	恢复 CRC 配置，并在输入时钟的上升沿上启动 CRC 计算。
CRC_Sleep()	停止 CRC 计算，并保存 CRC 配置。
CRC_Init()	用初始值初始化种子和多项式寄存器。
CRC_Enable()	在输入时钟的上升沿上启动 CRC 计算。
CRC_SaveConfig()	保存种子和多项式寄存器。
CRC_RestoreConfig()	恢复种子和多项式寄存器。
CRC_WriteSeed()	写入种子值。
CRC_WriteSeedUpper()	写入种子值的上半部分。仅针对 33 到 64 位 CRC 生成。
CRC_WriteSeedLower()	写入种子值的下半部分。仅针对 33 到 64 位 CRC 生成。
CRC_ReadCRC()	读取 CRC 值。
CRC_ReadCRCUpper()	读取 CRC 值的上半部分。仅针对 33 到 64 位 CRC 生成。
CRC_ReadCRCLower()	读取 CRC 值的下半部分。仅针对 33 到 64 位 CRC 生成。
CRC_WritePolynomial()	写入 CRC 多项式值。
CRC_WritePolynomialUpper()	写入 CRC 多项式值的上半部分。仅针对 33 到 64 位 CRC 生成。
CRC_WritePolynomialLower()	写入 CRC 多项式值的下半部分。仅针对 33 到 64 位 CRC 生成。
CRC_ReadPolynomial()	读取 CRC 多项式值。
CRC_ReadPolynomialUpper()	读取 CRC 多项式值的上半部分。仅针对 33 到 64 位 CRC 生成。
CRC_ReadPolynomialLower()	读取 CRC 多项式值的下半部分。仅针对 33 到 64 位 CRC 生成。

全局变量

变量	说明
CRC_initVar	指示是否已初始化 CRC。变量将初始化为 0，并在第一次调用 CRC_Start() 时设置为 1。这样，第一次调用 ACRC_Start() 子程序后，组件不用重新初始化即可重启。 如需重新初始化组件，可在 CRC_Start() 或 CRC_Enable() 函数前调用 CRC_Init() 函数。

void CRC_Start(void)

说明： 用初始值初始化种子和多项式寄存器。在输入时钟的上升沿上开始计算 CRC。

参数： None（无）

Return Value（返回值）： None（无）

Side Effects（副作用）： None（无）

void CRC_Stop(void)

说明： 停止 CRC 计算。

参数： None（无）

Return Value（返回值）： None（无）

Side Effects（副作用）： None（无）

void CRC_Sleep(void)

说明： 停止 CRC 计算，并保存 CRC 配置。

参数： None（无）

Return Value（返回值）： None（无）

Side Effects（副作用）： None（无）

void CRC_Wakeup(void)

说明：	恢复 CRC 配置，并在输入时钟的上升沿上启动 CRC 计算。
参数：	None（无）
Return Value（返回值）：	None（无）
Side Effects（副作用）：	None（无）

void CRC_Init(void)

说明：	用初始值初始化种子和多项式寄存器。
参数：	None（无）
Return Value（返回值）：	None（无）
Side Effects（副作用）：	None（无）

void CRC_Enable(void)

说明：	在输入时钟的上升沿上启动 CRC 计算。
参数：	None（无）
Return Value（返回值）：	None（无）
Side Effects（副作用）：	None（无）

void CRC_SaveConfig(void)

说明：	保存初始种子和多项式寄存器。
参数：	None（无）
Return Value（返回值）：	None（无）
Side Effects（副作用）：	None（无）

void CRC_RestoreConfig(void)

说明： 恢复初始种子和多项式寄存器。

参数： None（无）

Return Value（返回值）： None（无）

Side Effects（副作用）： None（无）

void CRC_WriteSeed(uint8/16/32 seed)

说明： 写入种子值。

参数： uint8/16/32 seed：种子值

Return Value（返回值）： None（无）

Side Effects（副作用）：

根据掩码 = $2^{分辨率} — 1$ 剪切种子值。

例如，如果 CRC 分辨率为 14 位，掩码值为：
掩码 = $2^{14} — 1$ = 0x3FFFu。

种子值 = 0xFFFFu 被剪切：种子和掩码 = 0xFFFFu 和 0x3FFFu = 0x3FFFu。

void CRC_WriteSeedUpper(uint32 seed)

说明： 写入种子值的上半部分。仅针对 33 到 64 位 CRC 生成。

参数： uint32 seed：种子值的上半部分

Return Value（返回值）： None（无）

Side Effects（副作用）：

根据掩码 = $2^{分辨率 — 32} — 1$ 剪切种子值的上半部分。

例如，如果 CRC 分辨率为 35 位，掩码值为：

$2^{(35 — 32)} — 1 = 2^3 — 1$ = 0x0000 0007u。

种子值的上半部分 = 0x0000 00FFu 被剪切。

种子和掩码的上半部分 = 0x0000 00FFu 和 0x0000 0007u = 0x0000 0007u。

void CRC_WriteSeedLower(uint32 seed)

说明：	写入种子值的下半部分。仅针对 33 到 64 位 CRC 生成。
参数：	uint32 seed：种子值的下半部分
Return Value（返回值）：	None（无）
Side Effects（副作用）：	None（无）

uint8/16/32 CRC_ReadCRC(void)

说明：	读取 CRC 值。
参数：	None（无）
Return Value（返回值）：	uint8/16/32：返回 CRC 值
Side Effects（副作用）：	None（无）

uint32 CRC_ReadCRCUpper(void)

说明：	读取 CRC 值的上半部分。仅针对 33 到 64 位 CRC 生成。
参数：	None（无）
Return Value（返回值）：	uint32：返回 CRC 值的上半部分
Side Effects（副作用）：	None（无）

uint32 CRC_ReadCRCLower(void)

说明：	读取 CRC 值的下半部分。仅针对 33 到 64 位 CRC 生成。
参数：	None（无）
Return Value（返回值）：	uint32：返回 CRC 值的下半部分
Side Effects（副作用）：	None（无）

void CRC_WritePolynomial(uint8/16/32 polynomial)

说明： 写入 CRC 多项式值。

参数： uint8/16/32 polynomial：CRC 多项式

Return Value（返回值）： None（无）

Side Effects（副作用）： 根据掩码 = $2^{分辨率} — 1$ 剪切多项式值。例如，如果 CRC 分辨率为 14 位，掩码值为：掩码 = $2^{14} — 1 = 0x3FFFu$。

多项式值 = 0xFFFFu 被剪切：

多项式和掩码 = 0xFFFFu 和 0x3FFFu = 0x3FFFu。

void CRC_WritePolynomialUpper(uint32 polynomial)

说明： 写入 CRC 多项式值的上半部分。仅针对 33 到 64 位 CRC 生成。

参数： uint32 polynomial：CRC 多项式值的上半部分

Return Value（返回值）： None（无）

Side Effects（副作用）： 根据掩码 = $2^{(分辨率 — 32)} — 1$ 剪切多项式值的上半部分。例如，如果 CRC 分辨率为 35 位，掩码值为：

$2^{(35 — 32)} — 1 = 2^3 — 1 = 0x0000\ 0007u$。

多项式值的上半部分 = 0x0000 00FFu 被剪切：

多项式和掩码的上半部分 = 0x0000 00FFu 和 0x0000 0007u = 0x0000 0007u。

void CRC_WritePolynomialLower(uint32 polynomial)

说明： 写入 CRC 多项式值的下半部分。仅针对 33 到 64 位 CRC 生成。

参数： uint32 polynomial：CRC 多项式值的下半部分

Return Value（返回值）： None（无）

Side Effects（副作用）： None（无）

uint8/16/32 CRC_ReadPolynomial(void)

说明：	读取 CRC 多项式值。
参数：	None（无）
Return Value（返回值）：	uint8/16/32：返回 CRC 多项式值
Side Effects（副作用）：	None（无）

uint32 CRC_ReadPolynomialUpper(void)

说明：	读取 CRC 多项式值的上半部分。仅针对 33 到 64 位 CRC 生成。
参数：	None（无）
Return Value（返回值）：	uint32：返回 CRC 多项式值的上半部分
Side Effects（副作用）：	None（无）

uint32 CRC_ReadPolynomialLower(void)

说明：	读取 CRC 多项式值的下半部分。仅针对 33 到 64 位 CRC 生成。
参数：	None（无）
Return Value（返回值）：	uint32：返回 CRC 多项式值的下半部分。
Side Effects（副作用）：	None（无）

固件源代码示例

PSoC Creator 在“查找示例项目”对话框中提供了很多包括原理图和代码示例的示例项目。要获取组件特定的示例，请打开组件目录中的对话框或原理图中的组件实例。要获取通用的示例，请打开 Start Page（开始页）或 **File**（文件）菜单中的对话框。根据需要，使用对话框中的 **Filter Options**（筛选选项）可缩小可选项目的列表。

有关更多信息，请参见 PSoC Creator 帮助中的“Find Example Project（查找示例项目）”主题。

功能描述

将 CRC 作为线性反馈移位寄存器 (LFSR) 来实现。移位寄存器计算 LFSR 函数，多项式寄存器保留定义 LFSR 多项式的多项式，种子寄存器启用启动数据的初始化。

启动组件之前，必须初始化种子和多项式寄存器。

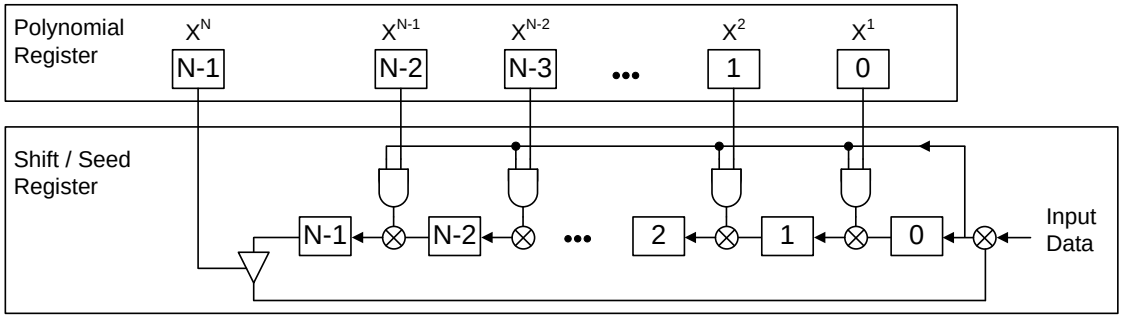
N 位 LFSR 结果的计算由带 N + 1 个乘积项的多项式指定，最后一个乘积项为 X^0 乘积项，其中 $X^0 = 1$。例如，广泛使用的 CRC-CCITT 16 位多项式为 $X^{16} + X^{12} + X^5 + 1$。CRC 算法假设存在 X^0 乘积项，这样，对于 N 位结果，可用 N 位而非 (N + 1) 位规范来表达多项式。

要指定多项式规范，写入对应完全多项式的 (N + 1) 位二进制数，各个乘积项带 1。CRC-CCITT 多项式为 10001000000100001b。然后，放弃最右边的位（X^0 乘积项）以获取 CRC 多项式值。要实现 CRC-CCITT 示例，加载值为 8810h 的多项式寄存器。

输入时钟的上升沿将输入数据流的每一位（先 MSB）移过移位寄存器，计算指定 CRC 算法。需要八个时钟以计算输入数据的各个字节的 CRC。

注意，初始种子值丢失。这无关紧要，因为种子值仅用于针对各个数据集初始化移位寄存器一次。

框图和配置



时序图

图 1. 时分复用实现模式

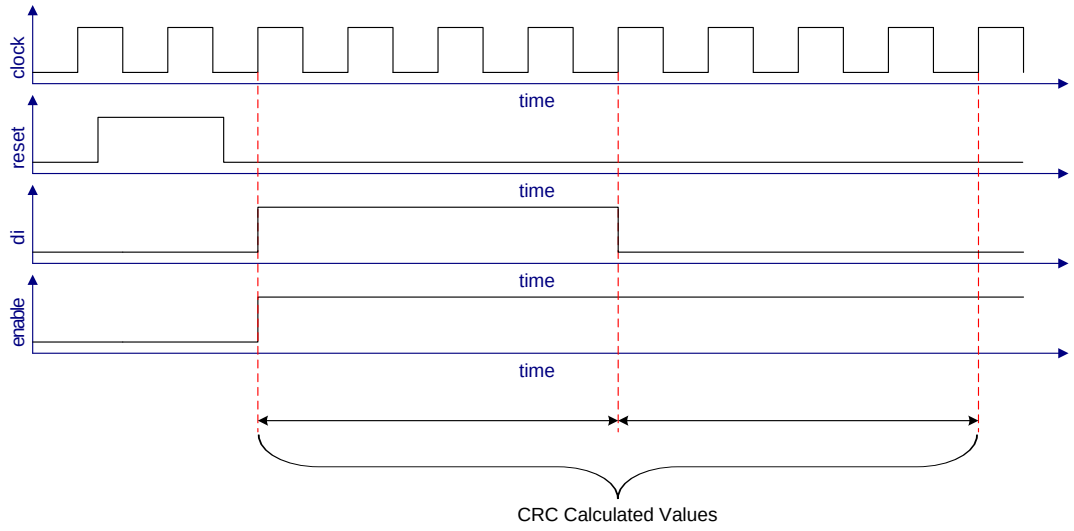
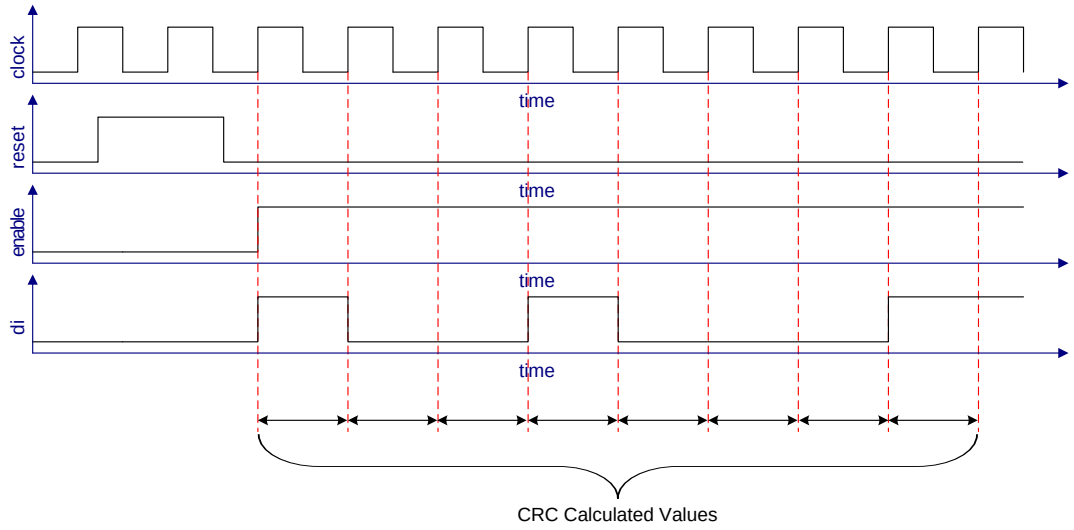


图 2. 单周期实现模式



直流和交流电气特性

下面的值表示了预计性能，它们基于初始特性数据。

时序特性“额定路由的最大值”

参数	说明	配置[1]	最小值	典型值	最大值	单位
f_{CLOCK}	组件时钟频率 [2]	配置 1			45	MHz
		配置 2			30	MHz
		配置 3			41	MHz
		配置 4			24	MHz
		配置 5			35	MHz
		配置 6			21	MHz
t_{CLOCKH}	输入时钟高电平时间 [3]	不可用		0.5		$1/f_{CLOCK}$
t_{CLOCKL}	输入时钟低电平时间 [3]	不可用		0.5		$1/f_{CLOCK}$
输入						
t_{PD_ps}	输入路径延迟，要同步的引脚 [4]	1			STA [5]	ns

[1] 配置：

配置 1：
分辨率：8 位
实现：单周期

配置 2：
分辨率：16 位
实现：单周期

配置 3：
分辨率：16 位
实现：时分复用

配置 4：
分辨率：32 位
实现：单周期

配置 5：
分辨率：32 位
实现：时分复用

配置 6：
分辨率：64 位
实现：时分复用

[2] 如果选择了“时分复用实现”，则组件时钟频率必须比数据速率大 4 倍。

[3] $t_{CY_clock} = 1/f_{CLOCK}$。这是一个时钟周期的循环时间。

[4] 可以在后面所述的“静态时序结果”中找到 t_{PD_ps}。此处列出的数字是基于许多输入的 STA 分析的额定值。

[5] t_{PD_ps} 和 $_{PD_si}$ 是路由路径延迟。由于路由是动态的，这些值可以更改，且将直接影响最大组件时钟和同步时钟频率。静态时序分析结果中能够找到这些值。

[6] 配置 2 中的 t_{PD_ps} 是为器件的每个引脚定义的固定值。此处列出的数字是器件上可用的所有引脚的额定值

参数	说明	配置[1]	最小值	典型值	最大值	单位
t_{PD_ps}	输入路径延迟，要同步的引脚 [6]	2			8.5	ns
t_{PD_si}	输入路径延迟的同步输出（路由）	1,2,3,4			STA [5]	ns
t_{I_clk}	clockX 与时钟的对齐	1,2,3,4	0		1	t_{CY_clock}
t_{PD_IE}	组件时钟的输入路径延迟（边沿敏感输入）	1,2	t_{PD_ps} + t_{SYNC} + t_{PD_si}		t_{PD_ps} + t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{PD_IE}	组件时钟的输入路径延迟（边沿敏感输入）	3,4	t_{sync} + t_{PD_si}		t_{sync} + t_{PD_si} + t_{I_clk}	ns
t_{IH}	输入高电平时间	1,2,3,4	t_{CY_clock} [7]			ns
t_{IL}	输入低电平时间	1,2,3,4	t_{CY_clock} [7]			ns

[7] $t_{CY_clock} = 4 \times [1/f_{CLOCK}]$（如果选择了“时分复用实现”）。

时序特性“所有路由的最大值”

参数	说明	配置 [8]	最小值	典型值	最大值 [9]	单位
f_{CLOCK}	组件时钟频率 [10]	配置 1			23	MHz
		配置 2			15	MHz
		配置 3			21	MHz
		配置 4			12	MHz
		配置 5			18	MHz
		配置 6			11	MHz
T_{CLOCKH}	输入时钟高电平时间 [11]	不可用		0.5		$1/f_{CLOCK}$
T_{CLOCKL}	输入时钟低电平时间 [11]	不可用		0.5		$1/f_{CLOCK}$
输入						
t_{PD_ps}	输入路径延迟，要同步的引脚 [12]	1			STA [13]	ns

[8]配置：

配置 1：
分辨率：8 位
实现：单周期

配置 2：
分辨率：16 位
实现：单周期

配置 3：
分辨率：16 位
实现：时分复用

配置 4：
分辨率：32 位
实现：单周期

配置 5：
分辨率：32 位
实现：时分复用

配置 6：
分辨率：64 位
实现：时分复用

[9] “所有路由”时序号的最大值是通过将“额定路由”时序号减去因子 2 而计算得出的。如果您的组件实例的运行速度没有超过这些速度，则应无须担心此组件会遇到时序问题。

[10] 如果选择了“时分复用实现”，则组件时钟频率必须比数据速率大 4 倍。

[11] $t_{CY_clock} = 1/f_{CLOCK}$。这是一个时钟周期的循环时间。

[12] 可以在后面所述的“静态时序结果”中找到 t_{PD_ps}。此处列出的数字是基于许多输入的 STA 分析的额定值。

[13] t_{PD_ps} 和 $_{PD_si}$ 是路由路径延迟。由于路由是动态的，这些值可以更改，且将直接影响最大组件时钟和同步时钟频率。静态时序分析结果中能够找到这些值。

参数	说明	配置 [8]	最小值	典型值	最大值 [9]	单位
t_{PD_ps}	输入路径延迟，要同步的引脚 [14]	2			8.5	ns
t_{PD_si}	输入路径延迟的同步输出（路由）	1,2,3,4			STA [5]	ns
t_{I_clk}	clockX 与时钟的对齐	1,2,3,4	0		1	t_{CY_clock}
t_{PD_IE}	组件时钟的输入路径延迟（边沿敏感输入）	1,2	t_{PD_ps} + t_{SYNC} + t_{PD_si}		t_{PD_ps} + t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{PD_IE}	组件时钟的输入路径延迟（边沿敏感输入）	3,4	t_{SYNC} + t_{PD_si}		t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{IH}	输入高电平时间	1,2,3,4	t_{CY_clock} [15]			ns
t_{IL}	输入低电平时间	1,2,3,4	t_{CY_clock} [15]			ns

如何将 STA 结果用于特性数据

额定路由最大值是通过使用静态时序分析 (STA) 进行多次测试而收集的。您可以用下列方法，使用 STA 结果计算设计的最大值：

f_{CLOCK} 最大组件时钟频率显示在命名外部时钟的时钟汇总中的时序结果中。下图演示了 *_timing.html* 中的时钟限制示例：

-Clock Summary

Clock	Actual Freq	Max Freq	Violation
BUS_CLK	24.000 MHz	118.683 MHz	
clock	24.000 MHz	56.967 MHz	

输入路径延迟和脉冲宽度

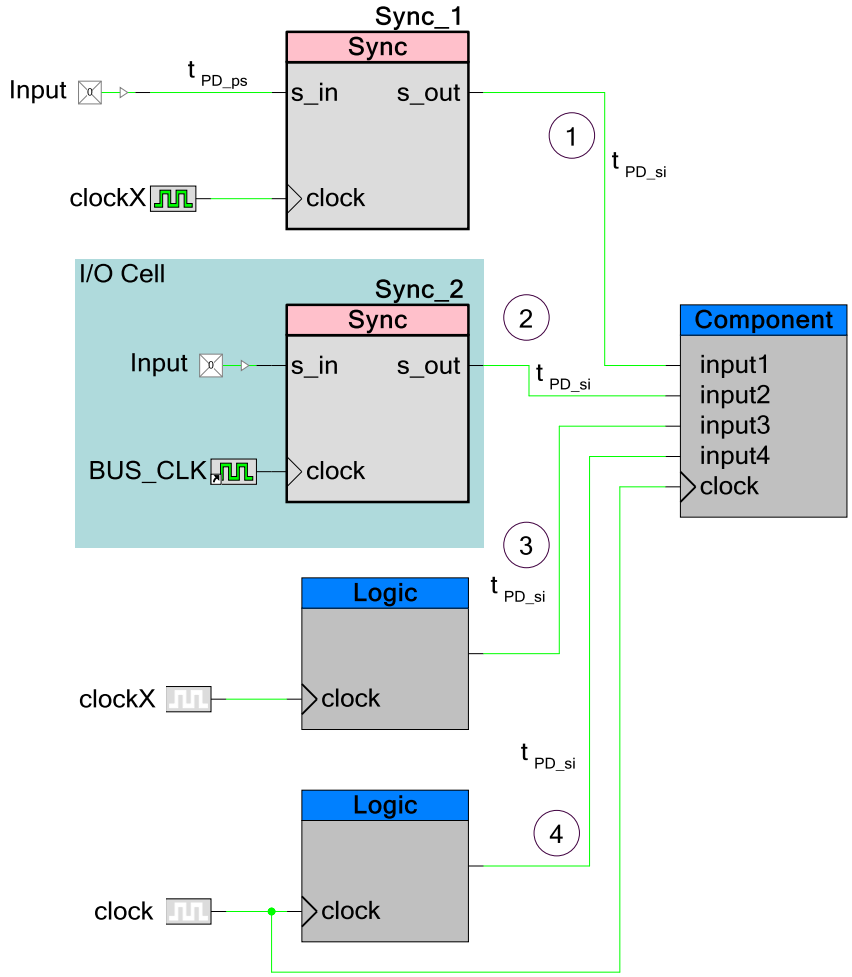
当表现输入功能的特征时，所有输入（无论您如何配置它们）看上去都类似于四种可能配置之一，如图 3 所示。

必须同步所有输入。同步机制取决于组件输入源。为了完全解析您的系统如何工作，您必须了解已为每个输入设置哪个输入配置以及系统的时钟配置。本节介绍如何使用静态时序分析 (STA) 结果确定系统的特性。

[14] 配置 2 中的 t_{PD_ps} 是为器件的每个引脚定义的固定值。此处列出的数字是器件上可用的所有引脚的额定值

[15] $t_{CY_clock} = 4 \times [1/f_{CLOCK}]$（如果选择了“时分复用实现”）。

图 3. 组件时序规范的输入配置



配置	组件时钟	同步器时钟（频率）	图形
1	master_clock	master_clock	图 8
1	clock	master_clock	图 6
1	clock	clockX = 时钟 [16]	图 4
1	clock	clockX > 时钟	图 5
1	clock	clockX < 时钟	图 7
2	master_clock	master_clock	图 8
2	clock	master_clock	图 6
3	master_clock	master_clock	图 13

[16] 时钟频率相等，但是不保证上升沿的对齐。

配置	组件时钟	同步器时钟（频率）	图形
3	clock	master_clock	图 11
3	clock	clockX = 时钟 [16]	图 9
3	clock	clockX > 时钟	图 10
3	clock	clockX < 时钟	图 12
4	master_clock	master_clock	图 13
4	clock	clock	图 9

1. 输入由器件引脚驱动，并在内部与“同步”组件同步。此组件的时钟采用与组件所使用时钟不同的内部时钟（所有内部时钟派生自 master_clock）。

   当表现按此方法配置的输入的特性时，clockX 可以快于、等于或慢于组件时钟。它还可以等于 master_clock，该时钟生成如图 4、图 5、图 7 和图 8 所示的特性参数。

2. 输入由器件引脚驱动，并使用 master_clock 在引脚同步。

   当表现按此方法配置的输入的特性时，master_clock 快于或等于组件时钟（从未慢于组件时钟）。这会生成如图 5 和图 8 所示的特性参数。

图 4. 输入配置 1 和 2；同步器时钟频率 = 组件时钟频率（不保证时钟和 clockX 的边沿对齐）

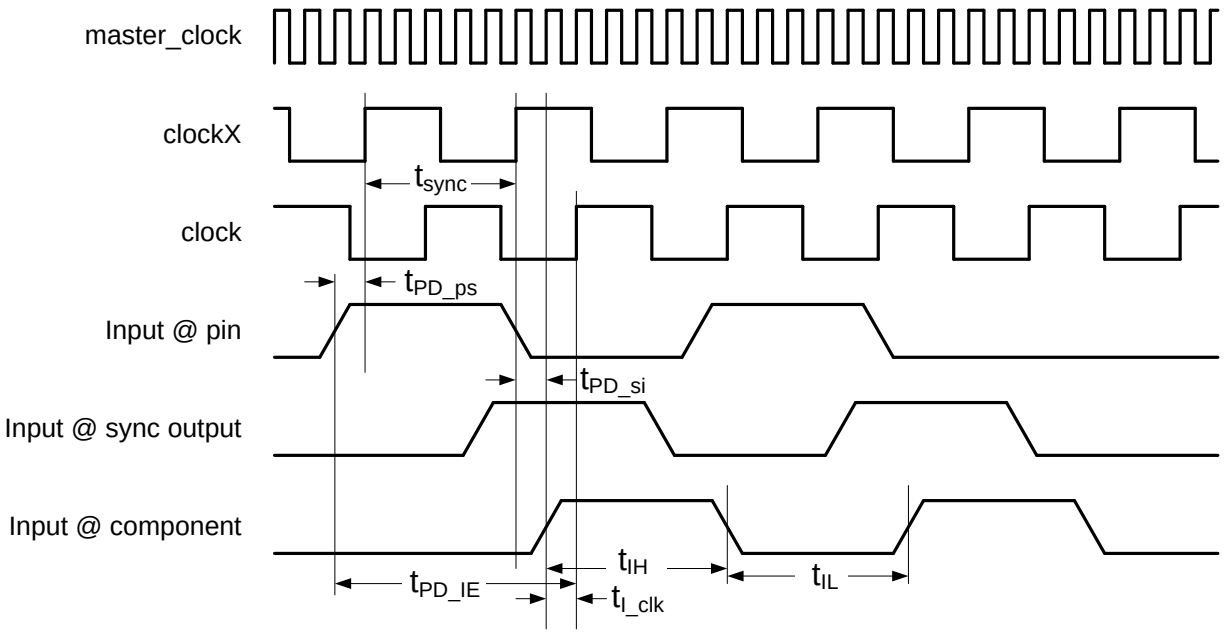


图 5. 输入配置 1 和 2：同步器时钟频率 > 组件时钟频率

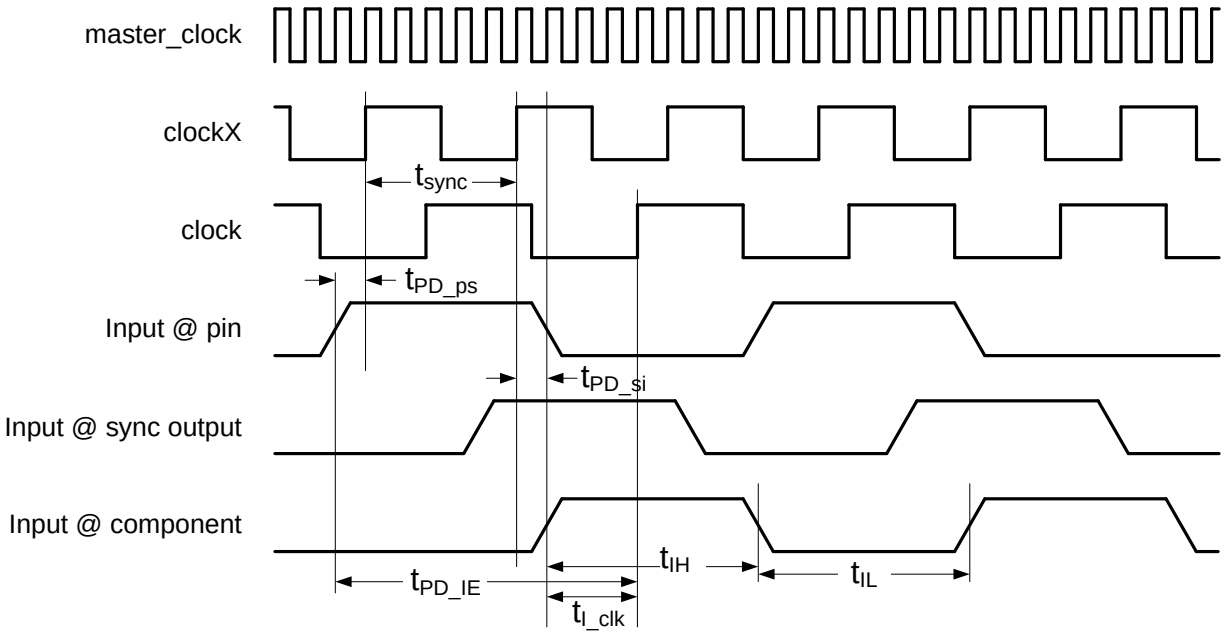


图 6. 输入配置 1 和 2：[同步器时钟频率 = master_clock] > 组件时钟频率

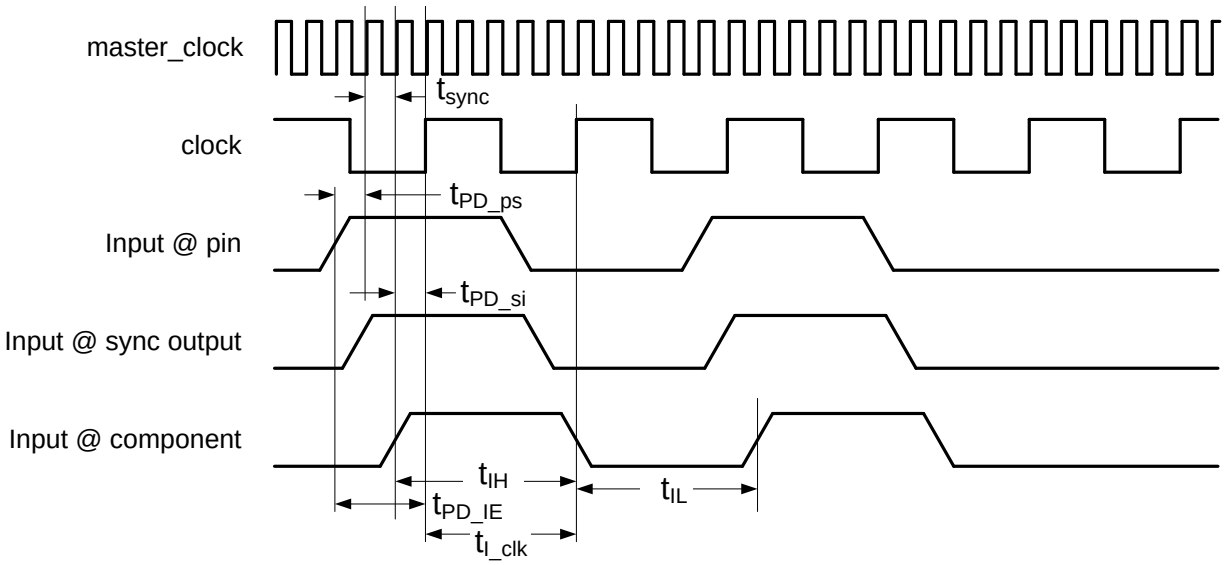


图 7. 输入配置 1：同步器时钟频率 < 组件时钟频率

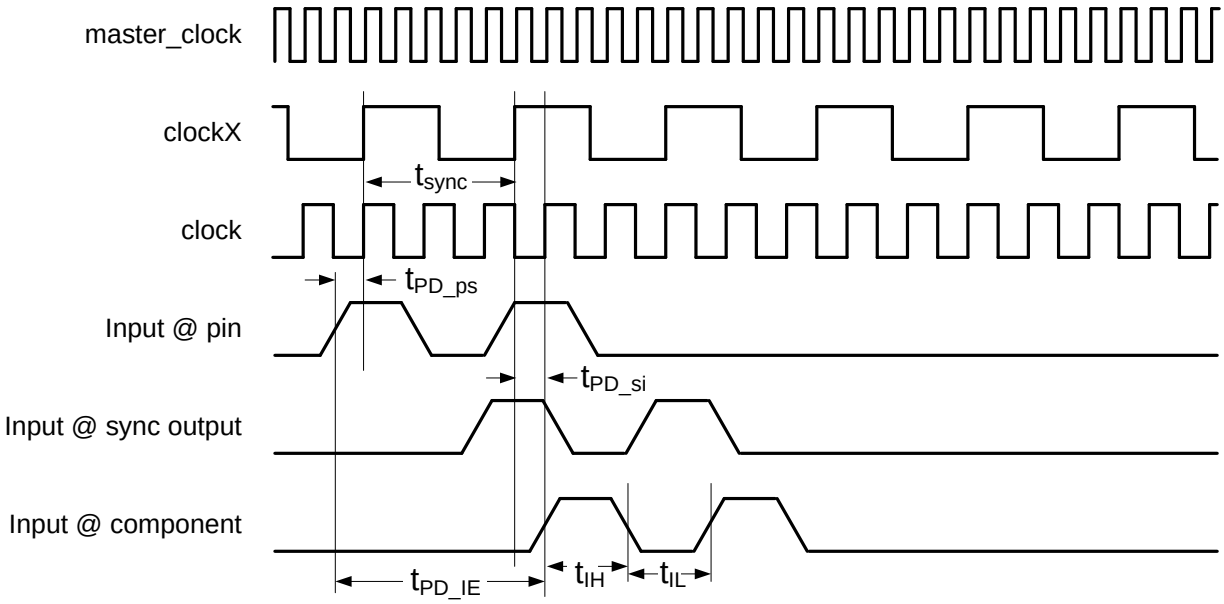
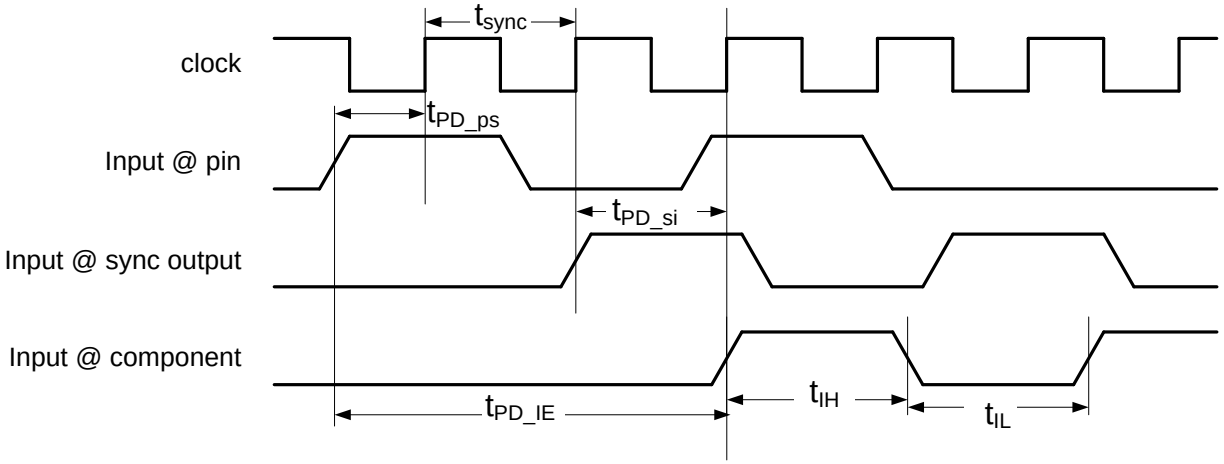


图 8. 输入配置 1 和 2：同步时钟 = 组件时钟 = master_clock



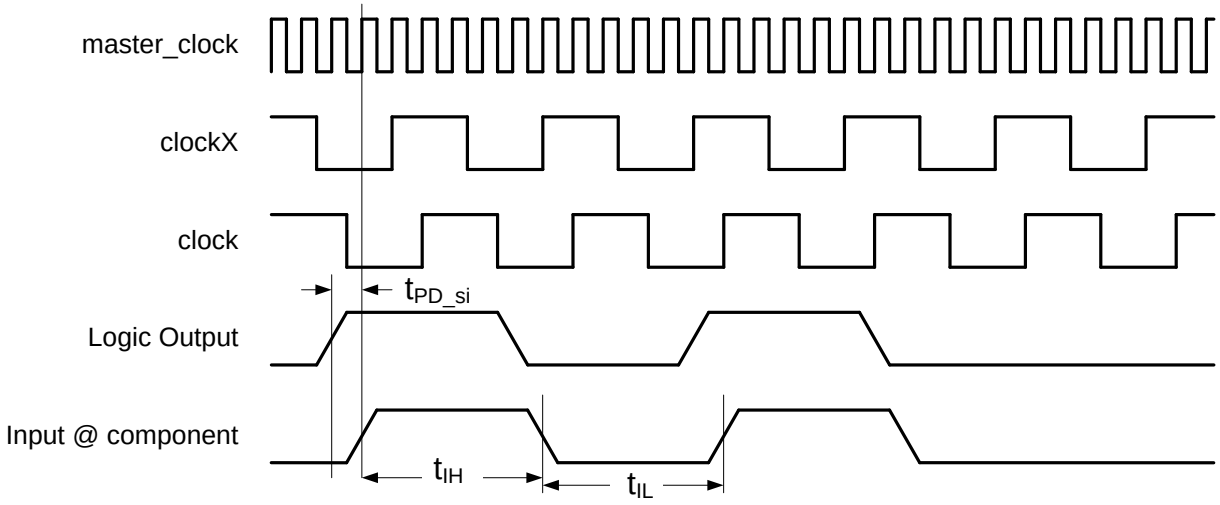
3. 输入由 PSoC 内部逻辑驱动，它基于与组件所使用的时钟不同的时钟同步（所有内部时钟都派生自 master_clock）。

   当表现按此方法配置的输入的特性时，同步器时钟快于、慢于或等于组件时钟，该时钟生成如图 9、图 10 和图 12 所示的特性参数。

4. 输入由 PSoC 内部逻辑驱动，它基于与组件所使用的时钟同步。

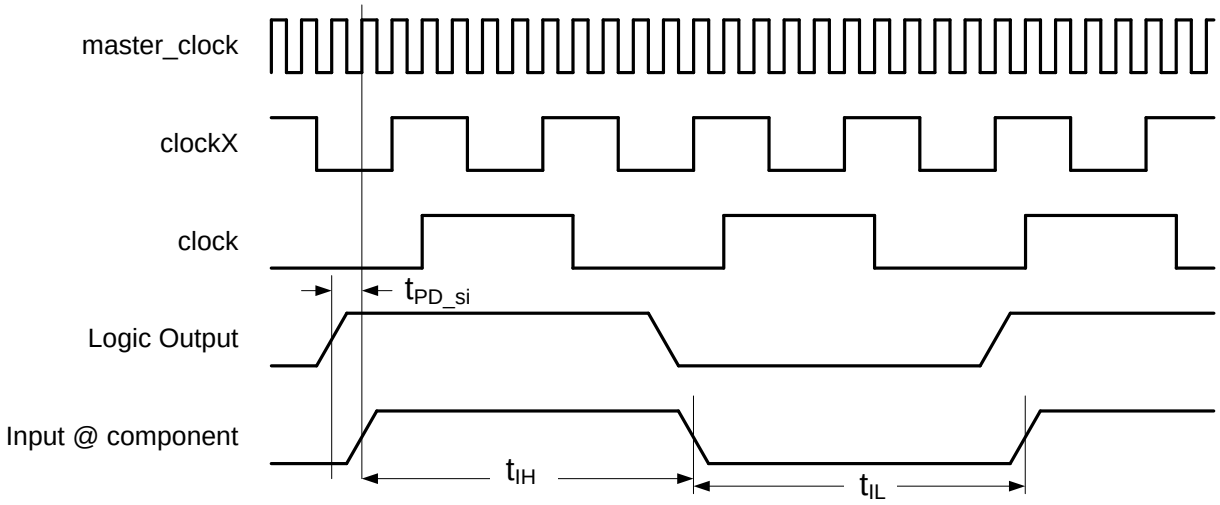
   当表现按此方法配置的输入的特性时，同步器时钟等于组件时钟，该时钟将生成如图 13 所示的特性参数。

图 9. 输入配置 3：同步器时钟频率 = 组件时钟频率（不保证时钟和 clockX 的边沿对齐）



此图表明静态时序分析保持时钟。数字时钟域中的所有时钟与 master_clock 同步。不过，具有相同频率的两个时钟的上升沿很可能不对齐。因此，静态时序分析工具不了解时钟同步到哪个边沿，必须假设最小值为 1 个 master_clock 循环。这意味着 t_{PD_si} 现在对系统的 master_clock 的影响有限。如果此路径延迟太长，则 master_clock 设置时间出现冲突。您必须更改系统的同步时钟，或者以较慢的频率运行 master_clock。

图 10. 输入配置 3：同步器时钟频率 < 组件时钟频率



与图 9 中的方法几乎相同，所有时钟都派生自 master_clock。STA 在此配置中指明了对一个 master_clock 周期的 master_clock 的 t_{PD_si} 限制。如果此路径延迟太长，则 master_clock 设置时间出现冲突。您必须更改系统的同步时钟，或者以较慢的频率运行 master_clock。

图 11. 输入配置 3：同步器时钟频率 = master_clock > 组件时钟频率

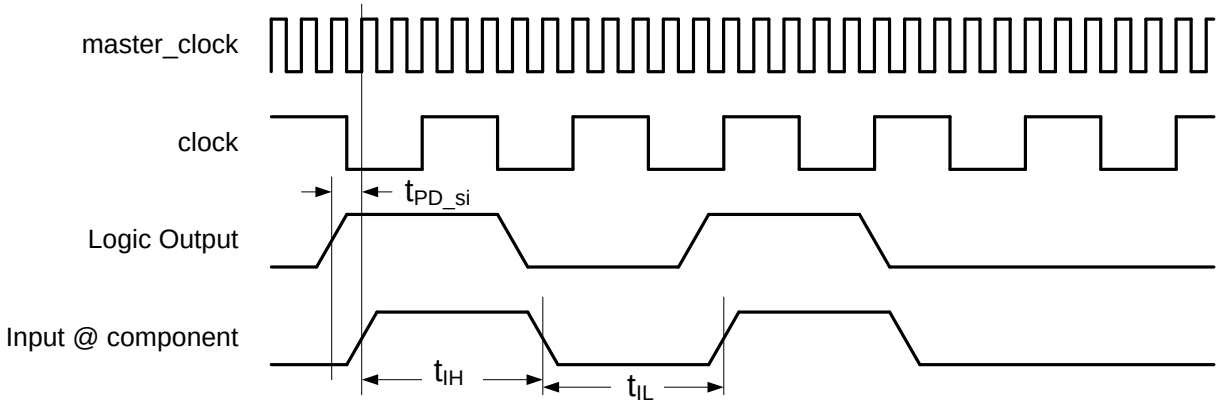
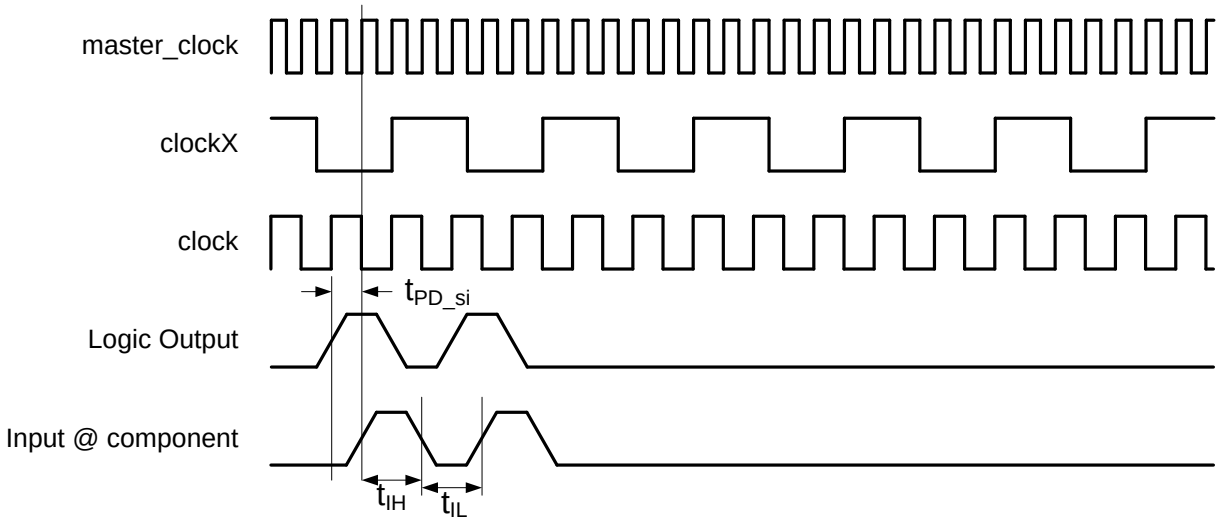
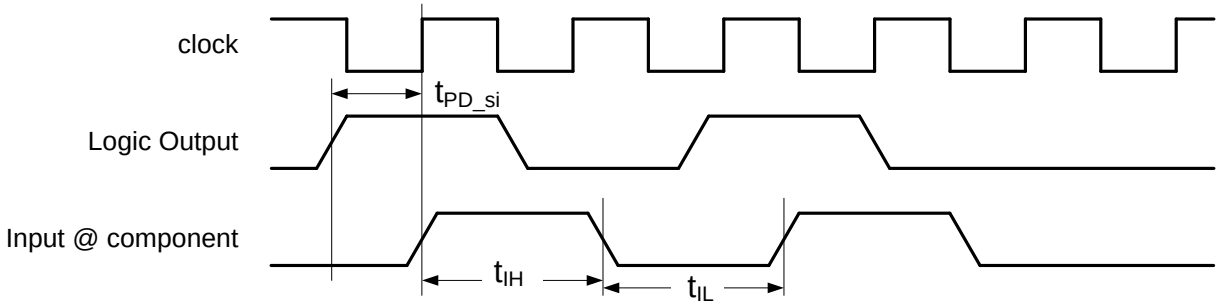


图 12. 输入配置 3：同步器时钟频率 < 组件时钟频率



与图 9 中的方法几乎相同，所有时钟都派生自 master_clock。STA 在此配置中指明了对一个 master_clock 周期的 master_clock 的 t_{PD_si} 限制。如果此路径延迟太长，则 master_clock 设置时间出现冲突。您必须更改系统的同步时钟，或者以较慢的频率运行 master_clock。

图 13. 仅输入配置 4：同步器时钟 = 组件时钟



在本节的所有上述图形中，在了解实现时使用的最关键参数是 f_{CLOCK} 和 t_{PD_IE}。t_{PD_IE} 由 t_{PD_ps} 和 t_{SYNC}（仅针对配置 1 和 2）、t_{PD_si}、和 t_{I_Clk} 定义。最重要的是 t_{PD_si} 定义最大组件时钟频率。t_{I_Clk} 不源自 STA 结果，但是用于表示何时寄存 t_{PD_IE}。这是同步器与组件时钟之间的路由之后余留的余量。

t_{PD_ps} 和 t_{PD_si} 包括在 STA 结果中。

要查找 t_{PD_ps}，请查看 *_timing.html* 文件中定义的输入设置时间。此输入的输出端可以大于 1，因此您需要计算这些路径的最大值。

-Setup times

-Setup times to clock BUS_CLK

Start	Register	Clock	Delay (ns)
input1(0):iocell.pad_in	input1(0):iocell.ind	BUS_CLK	16.500

t_{PD_si} 是在“寄存器至寄存器”时间中定义的。您需要知道使用 *_timing.html* 文件的网络的名称。此路径的输出端可以大于 1，因此您需要计算这些路径的最大值。

-Register-to-register times

-Destination clock clock

Destination clock clock (Actual freq: 24.000 MHz)

+Source clock clock

-Source clock clock_1

Source clock clock_1 (Actual freq: 24.000 MHz)
Affected clock: BUS_CLK (Actual freq: 24.000 MHz)

Start	End	Period (ns)	Max Freq	Frequency	Violation
\Sync_1:genblk1[0]:INST\:synccell.syncq	\PWM_1:PWMUDB:runmode_enable\:macrocell.mc_d	7.843	127.508 MHz	24.000 MHz	

输出路径延迟

当表现输出路径延迟的特性时，必须考虑输出的去向，以了解在 STA 结果中何处可以找到数据。对于此组件，所有输出同步到组件时钟。输出可以是下列两类之一。输出到器件中的另一个组件，或输出到器件外的引脚。在第一种情况下，必须查看为上面“逻辑至输入”说明显示的“寄存器至寄存器”时间（源时钟是组件时钟）。对于第二种情况，可以在 *_timing.html* STA 结果中查看“时钟至输出”时间。

组件更改

本节介绍组件与以前版本相比的主要更改。

版本	更改说明	更改/影响原因
2.10	针对实现参数更改了错误消息及其外观。	
	修正了多项式次数“N”设置，设为 64 位分辨率。	
	修正了多项式值验证。	
2.0.b	对数据表进行了少量编辑和更新	
2.0.a	向数据手册中添加了特性数据	
	对数据表进行了少量编辑和更新	
2.0	添加了对 PSoC 3 ES3 芯片的支持。更改包括： ■ 添加了时分复用实现的 4x 时钟 ■ 1x 时钟上现可用 1 到 32 位的单周期实现。 ■ 4x 时钟上现可用 9 到 64 位的时分复用实现。 ■ 添加了异步输入信号复位。 ■ 添加了同步输入信号使能。 ■ 针对“Implementation”（实现）（Time Division Multiplex（时分复用实现）、Single Cycle（单周期实现））参数将新的“Advanced”（高级）页面添加到“Configure”（配置）对话框。	新增了支持 PSoC 3 ES3 器件的要求，因此创建了新的 2.0 版 CRC 组件。
	添加了 CRC_Sleep()/CRC_Wakeup() 和 CRC_Init()/CRC_Enable() API。	为支持低功耗模式并提供常用接口，以单独控制大多数组件的初始化和启用。
	更新了函数 CRC_WriteSeed() 和 CRC_WriteSeedUpper()。	掩码参数用于剪切种子值，以在写入时定义 CRC 分辨率。
	已将验证器添加到“Resolution”（分辨率）参数。	CRC 分辨率为 1 到 64 位。添加了验证器以限制输入值。

版本	更改说明	更改/影响原因
	将复位 DFF 触发器添加到多项式写入函数：CRC_WritePolynomial()、CRC_WritePolynomialUpper() 和 CRC_WritePolynomialLower()。	计算 CRC 之前，需要将 DFF 触发器设为正确的状态（多项式的最高有效位始终为 1）。要符合此条件，任何对种子或多项式寄存器的写入都将复位 DFF 触发器。
	已更新“Configure”（配置）对话框，以使用以下参数的“Expression View”（表达式视图）：“PolyValueLower”、“PolyValueUpper”、“SeedValueLower”、“SeedValueLower”	“Expression View”（表达式视图）用于直接访问符号参数。此视图允许您用外部参数连接组件参数（如果需要）。
	已更新“Configure”（配置）对话框，以为各种参数添加错误图标。	如果在文本框中输入了错误的值，将显示错误图标以及问题说明的工具提示。这种方法比单独提供错误消息更方便。
1.20	已更改 API 生成方式。在 1.10 版本中是根据定制器的设置来生成 API 的。在 1.20 版本中，像大部分其他组件那样，API 是由 .c 和 .h 文件提供的。	此更改使用户可查看和更改生成的 API 文件，并且在后续构建上不会被覆盖。
	“Seed”（种子）和“Polynomial”（多项式）参数已更改为以十六进制格式显示。	已进行更改，以符合公司标准。

© 赛普拉斯半导体公司，2009-2012。此处所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品的内嵌电路之外，赛普拉斯半导体公司不对任何其他电路的使用承担任何责任。也不根据专利或其他权利以明示或暗示的方式授予任何许可。除非与赛普拉斯签订明确的书面协议，否则赛普拉斯产品不保证能够用于或适用于医疗、生命支持、救生、关键控制或安全应用领域。此外，对于可能发生运转异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

PSoC® 是赛普拉斯半导体公司的注册商标，PSoC® Creator™ 和 Programmable System-on-Chip™ 是赛普拉斯半导体公司的商标。此处引用的所有其他商标或注册商标归其各自所有者所有。

所有源代码（软件和/或固件）均归赛普拉斯半导体公司（赛普拉斯）所有，并受全球专利法规（美国和美国以外的专利法规）、美国版权法以及国际条约规定的保护和约束。赛普拉斯据此向获许可者授予适用于个人的、非独占性、不可转让的许可，用以复制、使用、修改、创建赛普拉斯源代码的派生作品、编译赛普拉斯源代码和派生作品，并且其目的只能是创建自定义软件和/或固件，以支持获许可者仅将其获得的产品依照适用协议规定的方式与赛普拉斯集成电路配合使用。除上述指定的用途之外，未经赛普拉斯的明确书面许可，不得对此类源代码进行任何复制、修改、转换、编译或演示。

免责声明：赛普拉斯不针对此材料提供任何类型的明示或暗示保证，包括（但不仅限于）针对特定用途的适销性和适用性的暗示保证。赛普拉斯保留在不做出通知的情况下对此处所述材料进行更改的权利。赛普拉斯不对此处所述之任何产品或电路的应用或使用承担任何责任。对于可能发生运转异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

产品使用可能受适用的赛普拉斯软件许可协议限制。