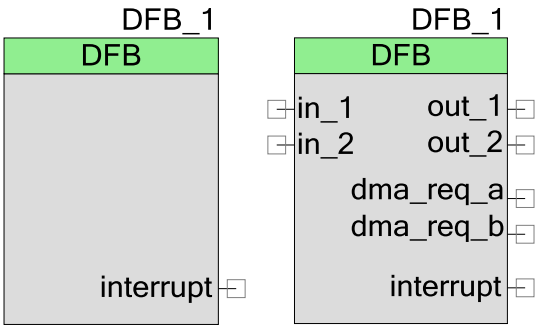
数字滤波器模块（DFB）汇编器

1.30

特性



- 提供了一个编辑器（用于输入汇编指令，用以配置 DFB 模块）以及一个汇编器（用于将汇编指令转换为指令字）。
- 支持汇编指令的仿真。
- 支持代码优化选项，该选项提供了一种机制，允许 DFB 代码 RAM 内可包含高达 128 个非常大的指令字。
- 提供了多种硬件信号，如：DMA 请求、DSI 输入与输出以及中断线。
- 支持与系统软件互交的各信号量以及将信号量捆绑到硬件信号的选项。

概述

可以将PSoC 3和PSoC 5中的数字滤波器模块（DFB）作为DSP小型处理器使用，并且您可以通过使用汇编指令配置DFB。该组件汇编输入到编辑器内的指令，并生成相应的十六进制代码字，然后将它们加载到DFB内。它还包括一个仿真器，从而允许用户仿真与调试各汇编指令。

DFB 具有一个可编程的 24*24 乘法器/累加器（MAC）、一个算术逻辑单元（ALU）、移位器以及各种程序和数据存储器，通过使用上述期间进行存储指令和数据。DFB 使用总线时钟，并且可以与 CPU 和 DMA 相连接。 它可以用于减轻 CPU 的负载，并能够加快与密集的乘积累加操作相关的算术运算的速度。您可使用 DFB 组件来执行的典型操作：向量运算、矩阵运算、滤波操作以及信号处理。

有关 DFB 的详细信息，请参见功能描述一节中的内容。

输入/输出接口

本节介绍 DFB 的输入和输出接口。I/O 列表中的星号（*）表示它是可隐藏的 I/O，其隐藏条件描述在该 I/O 的说明中。

in_1 — 输入*

输入端。通过该输入端，DFB 可以控制并查看芯片上的其他资源，特别是各个 UDB。在 **Configure** 对话框中选择 **Input 1** 选项时，将显示该输入。

in_2 — 输入*

输入端。通过该输入端，DFB 可以控制并查看芯片上的其他资源，特别是各个 UDB。在 **Configure** 对话框中选择 **Input 2** 选项时，将显示该输入。

out_1 — 输出*

输出端。允许 DFB 信号控制芯片上的其他资源，特别是各个 UDB。在 **Configure** 对话框中选择 **Output 1** 选项时，将显示该输出。

out_2 — 输出*

输出端。允许 DFB 信号控制其他的芯片上资源，特别是各个 UDB。在 **Configure** 对话框中选择 **Output 2** 选项时，将显示该输出。

dma_req_a — 输出*

DMA请求输出信号。它可以与保持寄存器或信号量位相连接，并且对触发DMA通道起着极大作用。 在**Configure**对话框中选择**DMA Request A Source**选项时，将显示该输出。

如果配置 DMA 请求信号，使其与输出保持存储器相连接，将生成一个电平感测信号，并将该信号发送到 DMA 通道上。读取此寄存器时，该信号被清除。

如果配置DMA请求信号由一个信号量生成，将创建一个单周期高脉冲。

dma_req_b — 输出*

DMA 请求输出信号。它可以与保持寄存器或信号量位相连接，并且对触发 DMA 通道起着极大作用。 在 **Configure** 对话框中选择 **DMA Request B Source** 选项时，将显示该输出。

如果配置 DMA 请求信号，使其与输出保持存储器相连接，将生成一个电平感测信号，并将该信号发送到 DMA 通道上。读取此寄存器时，该信号被清除。

如果配置 DMA 请求信号由一个信号量生成，将创建一个单周期高脉冲。

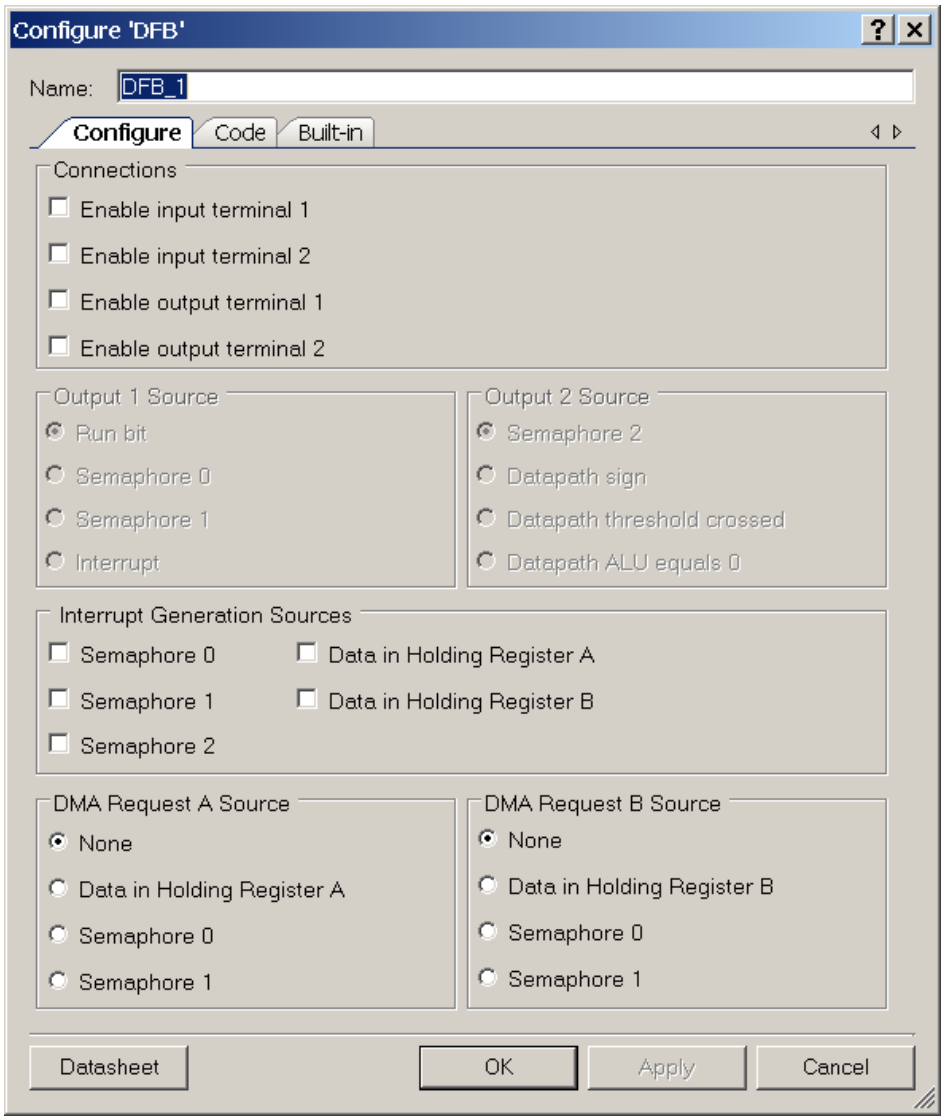
中断 — 输出

系统中断的输出。 它能够与各个保持寄存器以及信号量位相连接。

组件参数

将 DFB 拖入您的设计窗口中，然后双击它，以打开 **Configure** 对话框。**Configure** 对话框中包含多个选项卡和不同的参数，用于设置 DFB 组件。

Configure 选项卡



Connections（各个连接）

- **Enable input terminal 1（使能输入终端 1）**
- **Enable input terminal 2（使能输入终端 2）**
- **Enable output terminal 1（使能输出终端 1）**
- **Enable output terminal 2（使能输出终端 2）**

Output 1 Source（输出 1 之源）

确定被映射到输出全局信号 1 的内部信号。

- **Run bit （运行位）—** 该位与 DFB_CR 寄存器中的 RUN 位相同。
- **Semaphore 0（信号量 0）**
- **Semaphore 1（信号量 1）**
- **Interrupt（中断） —** 该信号与主 DFB 中断输出信号相同。

Output 2 Source（输出 2 之源）

确定被映射到输出全局信号 2 的内部信号。

- **Semaphore 2（信号量 2）**
- **Datapath sign —** 每当数据路径单元中的 ALU 输出为负向时，会激活此信号。在此条件为“True”的周期内，此信号保持高电平状态。
- **Datapath threshold crossed —** 执行下面指令之一：tdeca、tsuba、tsubb、taddabsa 或 taddabsb 时，每当超出 ALU 中的阈值 0 均会激活此信号。在此条件为“True”的周期内，此信号保持高电平。
- **Datapath ALU equals 0 —** 每当数据路径单元中的 ALU 输出等于 0，并且执行下面某条指令：tdeca、tsuba、tsubb、taddabsa 或 taddabsb 时，此信号被置为高电平。在此条件为“True”的周期内，此信号保持高电平。

Interrupt Generation Sources（中断生成之源）

配置会产生中断的事件：

- **Semaphore 0（信号量 0）**
- **Semaphore 1（信号量 1）**

- **Semaphore 2（信号量 2）**
- **Data in Holding Register A（保持寄存器 A 中的数据）**
- **Data in Holding Register B（保持寄存器 B 中的数据）**

DMA 请求模式

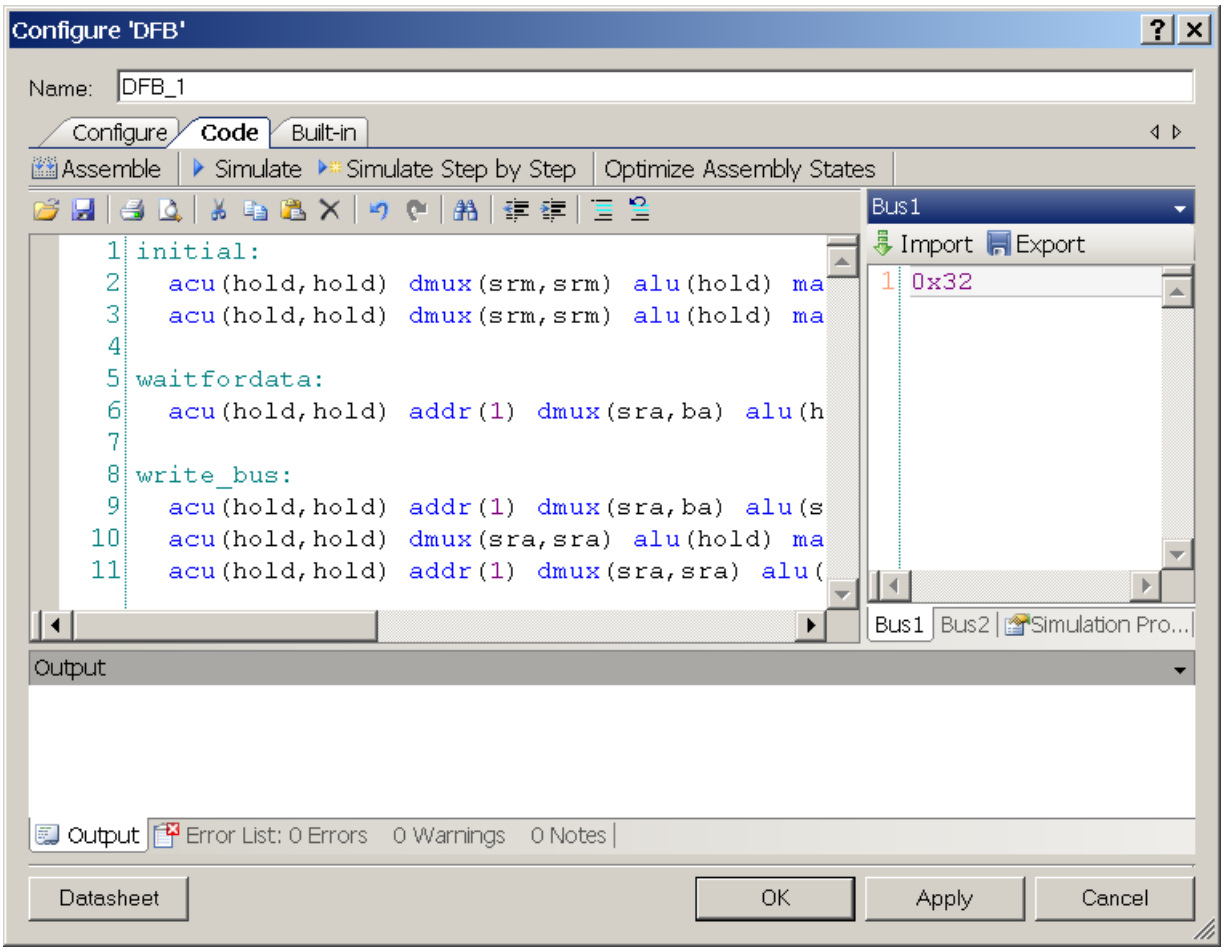
将配置 DMA 请求源。如果 **DMA 请求**不等于 **None**（无）， 将生成一个 DMA 请求输出端。

- **DMA Request A Source（DMA 请求的 A 源）：**
  - **None（不支持）**
  - **Data in Holding Register A（保持寄存器 A 中的数据）**
  - **Semaphore 0（信号量 0）**
  - **Semaphore 1（信号量 1）**
- **DMA Request B Source（DMA 请求的 B 源）：**
  - **None（不支持）**
  - **Data in Holding Register B（保持寄存器 B 中的数据）**
  - **Semaphore 0（信号量 0）**
  - **Semaphore 1（信号量 1）**

通过使用 CSR 的配置可以控制它。当某个 DMA 信号量被编成为 DMA_REC 时，HW 会将向该信号量写入“1”的任何写操作转换为单周期选通。这样，一个周期后会清除信号量。因此，如果将某个信号量作为 DMA 请求信号的源使用，它将自动被清除。

Code 选项卡

Code 选项卡提供一个用于输入汇编指令的编辑器、一个用于验证与汇编代码的汇编机制、以及一个用于仿真各指令的仿真器。



Assemble（汇编）

将汇编所输入的 DFB 汇编指令。状态与错误信息会显示在定制器的 **Output** 窗口中。键盘快捷方式 — **[F6]**

Simulate Continuously（连续仿真）

仿真器会持续运行，直到处理完总线输入数据（即 Bus1 数据和 Bus2 数据）。Bus1 数据和 Bus2 数据均为用于仿真的输入数据，此数据相应于将被移动到各个 DFB 分级寄存器内的数据。执行每个指令后，各个 DFB 子模块的已仿真内容会显示在 **Output** 窗口内。键盘快捷方式 — **[F5]**

开始仿真后，可以使用**[Shift] F5**]停止该操作。

Simulate Step by Step（逐步仿真）

仿真器会逐步运行，直到处理完总线输入数据。在每一步中，正在执行的代码行在代码编辑器内被加亮显示。各个仿真属性则显示在 **Simulation Properties** 面板上。键盘快捷方式 **— [F7]**

开始仿真后，可以使用**[F8]**进行下一步仿真。

Optimize Assembly States（优化汇编状态）

使能压缩功能。压缩器删除了 64 字的程序大小屏障，所以您能够将多达 128 个指令字存储在 DFB 代码 RAM 内。如果选择 **Optimize Assembly States** 选项，成功汇编代码后将运行代码压缩功能。

Code Editor（编码编辑器）

加亮显示 DFB 汇编代码。它使用不同的颜色加亮显示各批注、标签、指令及数值。

图 1. 文本编辑器工具栏



Text Editor（文本编辑器）

Open file：打开某个包含 DFB 汇编代码的文本文件。键盘快捷方式 **— [Ctrl] [O]**

Save file：保存某个包含 DFB 汇编代码的文本文件。键盘快捷方式 **— [Ctrl] [S]**

其他文本编辑功能：包括剪切、复制、粘贴、撤消、重做、打印、打印预览、查找/替换文本、批注、取消批注，以及其他功能。

输出通道

显示用于汇编、压缩以及仿真处理的日志信息。使用红色加亮显示含有错误文本的行；使用绿色加亮显示表示操作成功的行。如果您双击某个包含了错误文本的行，定制器会自动激活 **Code Editor**，并选择错误文本所在的行。

可以使用键盘快捷键**[Alt] [C]**来清除日志信息。

仿真器输出

仿真器的输出包含了可用于调试设计的信息。使用位于 **Code** 选项卡右上角的 Bus1（分级寄存器 A）和 Bus2（分级寄存器 B）窗口对输入 DFB 的数据进行仿真。通过更改 **Simulation Properties** 窗口中的内容，可以仿真信号量、全局输入以及全局输出。请注意，在固件中所调用的 API 不会影响到仿真属性。因此，需在仿真器中重复执行这些操作行为。包括填充数据 RAM、写入分级寄存器 A、读取保持寄存器和设置/取消设置信号量和全局输入。

信号名称	类型	说明
Cycle	uint	这是用于计数时钟周期数的程序计数器。
RamA	uint	当前RAM位于控制存储RAM A中，也称为CStoreA。
RamB	uint	在控制存储RAM B中的当前RAM位置，也简称为CStoreB。
Ram sel	字符串	控制存储RAM（A或B）正在执行。
CFSM state	uint	CFSM的当前状态。此值与CFSM内容的RAM值是不一样的。状态的顺序如在汇编代码中所显示的情况。因此，该代码中的第一个调用状态被指定为CFSM=1，而第二个调用状态为CFSM=2，以此类推。
Aaddr next	uint	数据RAM A的当前地址
Baddr next	uint	数据RAM B的当前地址
A2Mux	Hex	Mux2 （A）的输出，它对应于Mux1 （A）的输出或数据Ram A中的当前内容对应。
B2Mux	Hex	Mux2 （B）的输出，它对应于Mux1 （B）的输出或数据Ram A中的当前内容。
MacOut	Hex	乘法与累加单元（MAC）的输出。
AluOut	Hex	算术逻辑单元（ALU）的输出。
ShiftOut	Hex	移位器（位于ALU的输出）的输出。

Error List（错误列表）

指的是显示错误，警告及注意事项的列表。如果您双击某个错误，定制器会自动激活 **Code Editor**，并选中错误所在的行。

Bus1 （总线 1）

为仿真器提供 STAGEA 输入数据。输入一个十六进制、十进制或二进制格式的 24 位数值。例如：99 是十进制格式、0x63 是十六进制的、0b1100011 是二进制的。

Bus2（总线 2）

为仿真器提供 STAGEB 输入数据。输入一个十六进制、十进制或二进制格式的 24 位数值。

Error List（总线数据输入）

将数据导入 Bus1/Bus2 文本字段内。支持“.txt”和“.data”（一种基于 C 语言的仿真器所使用的旧数据格式）两种文本格式。

Bus data export （总线数据输出）

从 Bus1/Bus2 文本字段中导出数据。支持“.txt”和“.data”的两种文本格式。

Simulator Properties（仿真器属性）

通过所提供的功能，可以更改输入的值和信号量，并查看各个仿真器之间的内部仿真器值。

- **GlobalInput1**：读/写字段
- **GlobalInput2**：读/写字段
- **Semaphore0**：读/写字段
- **Semaphore1**：读/写字段
- **Semaphore2**：读/写字段
- **Cycle**：显示当前的周期编号
- **RamA Index**：只读字段，该字段表示 RAM A 中的当前索引。
- **RamB Index**：只读字段，该字段表示 RAM B 中的当前索引。
- **Ram Selected**：只读字段，该字段显示正在执行的 RAM（A 或 B）

应用编程接口（API）

通过应用编程接口（API）子程序，您可以使用软件对组件进行配置。下表列出并说明了每个函数的接口。后面部分将更详细地介绍每个函数。

默认情况下，PSoC Creator 将实例名称“DFB_1”分配给指定设计中第一个实例组件。您可以将其重新命名为任何一个符合标识符语法规则的值。实例名称会成为所有全局函数名称、变量和符号常量的前缀。为便于阅读，下表使用的实例名称为“DFB”。

函数	说明
DFB_Start()	通过使用DFB_Init()和DFB_Enable()函数可以初始化并使能DFB组件。
DFB_Stop()	关闭运行位。如果DMA控制被用于馈送各个通道，会允许参数关闭某一个TD通道。

函数	说明
DFB_Pause()	暂停DFB，并使能对DFB RAM的写入操作。
DFB_Resume()	禁用对DFB RAM的写操作，清除所有挂起中断，断开DFB RAM与数据总线的连接，并运行DFB。
DFB_SetCoherency()	根据输送到DFB的“coherencyKey”参数将一致性关键字节设置为低/中/高字节。
DFB_SetDalign()	允许9至16位输入输出采样在AHB总线上作为16位值移动。
DFB_LoadDataRAMA()	将数据加载到RAMA DFB存储器内。
DFB_LoadDataRAMB()	将数据加载到RAMB DFB存储器内。
DFB_LoadInputValue()	将输入值加载到所选的通道内。
DFB_GetOutputValue()	从各个DFB输出保持寄存器之一获取值。
DFB_SetInterruptMode()	设置将会触发某个DFB中断的事件。
DFB_GetInterruptSource()	观察DFB_SR寄存器，以确定被触发的中断源。
DFB_ClearInterrupt()	清除中断请求。
DFB_SetDMAMode()	设置将要触发DFB DMA请求的事件。
DFB_SetSemaphores()	设置数值为1的信号量。
DFB_ClearSemaphores()	清除数值为1的信号量。
DFB_GetSemaphores()	检查DFB信号量的当前状态，并返回此值。
DFB_SetOutput1Source()	选择将被映射到输出1的内部信号。
DFB_SetOutput2Source()	选择将被映射到输出2的内部信号。
DFB_Sleep()	为DFB组件进入睡眠状态做好准备。
DFB_Wakeup()	为DFB组件唤醒做好准备。
DFB_Init()	初始化或恢复定制器所提供的DFB默认配置。
DFB_Enable()	使能DFB硬件模块。设置DFB运行位。启动DFB模块。
DFB_SaveConfig(void)	保存DFB非保留寄存器的用户配置。通过DFB_Sleep()调用该例程，以在进入睡眠状态之前保存组件配置。
DFB_RestoreConfig()	恢复DFB非保留寄存器的用户配置。通过DFB_Wakeup()调用该例程，以在退出睡眠状态时恢复组件配置。

全局变量

变量	说明
DFB_initVar	标志DFB是否已初始化。该变量被初始化为0，并在第一次调用DFB_Start()时将该变量设置为1。这样，第一次调用DFB_Start()例程后，组件不用重新初始化即可重启。 如需重新初始化组件，可在DFB_Start()或DFB_Enable()函数前调用DFB_Init()函数。

void DFB_Start(void)

说明： 该函数通过使用DFB_Init()和DFB_Enable()函数来初始化并使能DFB组件。

参数： 无

返回值： 无

其他影响： 无

void DFB_Stop(void)

说明： 该函数会关闭运行位。如果DMA控制被用于馈送各个通道，DFB_Stop()会允许参数关闭某个TD通道。

参数： 无

返回值： 无

其他影响： 禁用DFB核的电源。

void DFB_Pause(void)

说明： 该函数暂停DFB，并使能对DFB RAM的写入操作。

- 关闭运行位
- 将 DFB RAM 连接到数据总线上，
- 清除 DFB 运行位，并将所有 DFB RAM 的控制权移交给总线。

参数： 无

返回值： 无

其他影响： 无

void DFB_Resume(void)

说明： 该函数禁用了对DFB RAM的写操作，清除所有挂起的中断，断开DFB RAM与数据总线的连接，并运行DFB。它将所有DFB RAM的控制权传递给DFB，然后设置运行位。

参数： 无

返回值： 无

其他影响： 无

void DFB_SetCoherency(uint8 coherencyKeyByte)

说明： 该函数根据输送到DFB的“coherencyKeyByte”参数将一致关键字节设置为低、中或高字节。请注意，该函数直接写入DFB连贯寄存器。因此，当传送coherencyKeyByte参数时，需指定所有寄存器的一致性。

DFB_SetCoherency()允许您选择每个STAGEA、STAGEB、HOLDA和HOLDB中的三个字节作为关键一致字节使用。一致是针对添加到此模块的硬件，用于防止模块故障。在寄存器字段大小超过了总线访问尺寸的情况下，当这些字段被部分写入或读取（不一致）时，将造成间隔。这时，需要使用一致性字节。当需要更新字段时，软件通过关键一致字节“告诉”硬件最后写入或读取的字段字节。当写入或读取了关键字节时，字段被标识为“一致”。如果对其他字节进行写入或读取，字段被标识为“不一致”。

参数： uint8 coherencyKeyByte：指定DFB一致寄存器中的位。

数值	说明
DFB_STGA_KEY_LOW	分级A寄存器的关键一致字节是低位字节。
DFB_STGA_KEY_MID	分级A寄存器的关键一致字节是中位字节。
DFB_STGA_KEY_HIGH	分级A寄存器的关键一致字节是高电平字节。
DFB_STGB_KEY_LOW	分级B寄存器的关键一致字节是低位字节。
DFB_STGB_KEY_MID	分级B寄存器的关键一致字节是中位字节。
DFB_STGB_KEY_HIGH	分级B寄存器的关键一致字节是高电平字节。
DFB_HOLDA_KEY_LOW	保持A寄存器的关键一致字节是低位字节。
DFB_HOLDA_KEY_MID	保持A寄存器的关键一致字节是中位字节。
DFB_HOLDA_KEY_HIGH	保持A寄存器的关键一致字节是高电平字节。
DFB_HOLDB_KEY_LOW	保持B寄存器的关键一致字节是低位字节。
DFB_HOLDB_KEY_MID	保持B寄存器的关键一致字节是中位字节。
DFB_HOLDB_KEY_HIGH	保持B寄存器的关键一致字节是高电平字节。

返回值： 无

其他影响： 一致性会影响到使用DFB_LoadInputValue()函数的数据加载，并使用DFB_GetOutputValue()函数的数据检索。

注意 分级A和B寄存器以及保留A和B寄存器的默认关键字节的配置为高电平字节。通过使用OR运算指定所有寄存器的一致性，然后传送coherencyKeyByte参数。如果该操作失败并且为所选寄存器传递一致性，可导致发生意外行为。

void DFB_SetDalign(uint8 dalignKeyByte)

说明： 该功能允许9至16位输入输出采样在AHB总线上作为16位值移动。如果这些位被置为高电平时，每当对相应的分级和保留寄存器进行访问时，将会引起数据中的8位移位。请注意，该函数直接写入DFB数据对齐寄存器。因此，当数据输送到 dalignKeyByte参数时，需指定所有寄存器的对齐。

因为DFB数据路径为MSB对齐，所以系统软件可以便捷的将分级和保留寄存器上的23:8位与总线上的15:0位进行对齐。这是因为从16或者甚至是8位宽的PHUB辐轮到32位宽的DFB辐轮的传输。Dalign位允许DFB对数据进行对齐操作 ，以便能够更加有效的进行输送前往这些不同大小的辐轮。

参数： uint8 dalignKeyByte：指定DFB数据对齐寄存器中的位。

数值	说明
DFB_STGA_DALIGN_LOW	正常写入
DFB_STGA_DALIGN_HIGH	写入时，将执行8位左移。
DFB_STGB_DALIGN_LOW	正常写入
DFB_STGB_DALIGN_HIGH	写入时，将执行8位左移。
DFB_HOLDA_DALIGN_LOW	正常读取
DFB_HOLDA_DALIGN_HIGH	写入时，将执行8位左移。
DFB_HOLDB_DALIGN_LOW	正常读取
DFB_HOLDB_DALIGN_HIGH	写入时，将执行8位左移。

返回值： 无

其他影响： 无

注意： 所有寄存器的一致性必须通过一个OR操作指定，并且传递给dalignKeyByte参数。此操作失败可导致意外行为。

void DFB_LoadDataRAMA(int32 * ptr, uint32 * addr, uint8 size)

说明： 该函数将数据加载到DFB RAM A存储器内。

参数： uint32 * ptr：指向需要加载的数据源的指针

uint32 * addr：用于加载DFB RAM A内的数据的起始地址。

uint8 size：需要加载的数据字的数量

返回值：

值	说明
DFB_SUCCESS	成功加载数据。
DFB_NAME`_ADDRESS_OUT_OF_RANGE	错误代码：表示越出范围的地址。
DFB_DATA_OUT_OF_RANGE	错误代码：表示数据溢出的错误。

其他影响： 如果DFB已被启动，该函数不能将其停止。推荐的方法是先调用DFB_Init(), DFB_LoadDataRAMA()，然后调用DFB_Enable()。

void DFB_LoadDataRAMB(uint32 * ptr, uint32 * addr, uint8 size)

说明： 通过该函数可以将数据加载到DFB RAM B存储器内。

参数： uint32 * ptr：指向需要加载的数据源的指针

uint32 * addr：用于加载DFB RAM B内的数据的起始地址。

uint8 size：需要加载的数据字的数量

返回值：

值	说明
DFB_SUCCESS	成功加载数据
DFB_NAME`_ADDRESS_OUT_OF_RANGE	错误代码：表示越出范围的地址
DFB_DATA_OUT_OF_RANGE	错误代码：表示数据溢出的错误。

其他影响： 如果DFB已被启动，该函数不能将其停止。推荐的方法是先调用DFB_Init()和DFB_LoadDataRAMB()，然后调用DFB_Enable()。

void DFB_LoadInputValue(uint8 channel, uint32 sample)

说明： 通过该函数可以将输入值加载到所选的通道内。

参数： channel：将DFB_CHANNEL_A (1)或DFB_CHANNEL_B (0)作为函数的参数使用。
采样数据：24位，右对齐输入的采样数据

返回值： 无

其他影响： 无

注意： 写入顺序非常重要。当加载高位字节时，DFB将设置输入的就绪位。 如果更改数据的连贯性或对齐方式，则应注意字节的顺序。

int32 DFB_GetOutputValue(uint8 channel)

说明： 该函数从一个DFB输出保持寄存器中取值。

参数： channel：将DFB_CHANNEL_A (1)或DFB_CHANNEL_B (0)作为函数的参数使用。

返回值： 当前输出值位于所选定的保持寄存器内。返回值可以是存储在输出字中三个最低有效字节内的24位。如果是无效通道数字，则该值为0xFF000000。

其他影响： 无

注意： 由于DFB的结构，从保持寄存器A和B读取的值都与MSB对齐，否则数据路径移位器将移位该值。 如果更改数据的连贯性或对齐方式，则应注意字节的顺序。

void DFB_SetInterruptMode(uint8 events)

说明： 该函数对触发DFB中断的事件进行分配。

参数： events：[0:5]位上的事件表示触发DFB中断的事件。

值	说明
DFB_HOLDA	每次向输出保持寄存器A写入一个新的有效数据时都会产生一个中断。
DFB_HOLDB	每次向输出保持寄存器B写入一个新的有效数据时都会产生一个中断。
DFB_SEMA0	每次向信号量寄存器位0写入‘1’时都会产生一个中断。
DFB_SEMA1	每次向信号量寄存器位1写入‘1’时都会产生一个中断。
DFB_SEMA2	每次向信号量寄存器位2写入‘1’时都会产生一个中断。

返回值： 无

其他影响： 无

注意： 不能同时将DMA请求和中断事件配置为信号量0和信号量1。因为在一个时钟周期后，系统将自动删除所有触发DMA请求的信号量。

uint8 DFB_GetInterruptSource(void)

说明： 该函数将在DFB_SR寄存器内查找已被触发的中断源。

参数： 无

返回值： uint8值中的位[0:5]表示触发DFB中断的事件。

值	说明
DFB_HOLDA	保持寄存器A是当前的中断源。
DFB_HOLDB	保持寄存器B是当前的中断源。
DFB_SEMA0	信号量寄存器位0是当前的中断源。
DFB_SEMA1	信号量寄存器位1是当前的中断源。
DFB_SEMA2	信号量寄存器位2是当前的中断源。

其他影响： 无

void DFB_ClearInterrupt(uint8 interruptMask)

说明： 该函数清除中断请求。

参数： interruptMask：需要清除的中断的掩码

值	说明
DFB_HOLDA	清除保持寄存器A的中断（读取该寄存器也会清除该中断）。
DFB_HOLDB	清除保持寄存器B的中断（读取该寄存器也会清除该中断）。
DFB_SEMA0	清除信号量寄存器位0的中断。
DFB_SEMA1	清除信号量寄存器位1的中断。
DFB_SEMA2	清除信号量寄存器位2的中断。

返回值： 无

其他影响： 清除信号量中断，同时清除信号量位。

void DFB_SetDMAMode(uint8 events)

说明： 该函数对触发DMA请求的事件进行分配。可以触发两个不同的DMA请求。

参数： events：四位组配置触发DMA请求用于DFB的事件（若有）。

值	说明
DFB_DMAREQ1_DISABLED	没有生成请求
DFB_DMAREQ1_HOLDA	已经将输出值保存在保持寄存器的通道A上。
DFB_DMAREQ1_SEM0	Semaphore 0（信号量0）
DFB_DMAREQ1_SEM1	Semaphore 1（信号量1）
DFB_DMAREQ2_DISABLED	没有生成请求
DFB_DMAREQ2_HOLDB	已经将输出值保存在保持寄存器的通道B上。
DFB_DMAREQ2_SEM0	Semaphore 0（信号量0）
DFB_DMAREQ2_SEM1	Semaphore 1（信号量1）

返回值： 无

其他影响： 无

注意： 不能同时将信号量0和信号量1配置为DMA请求和中断事件。因为一个时钟周期后，系统将自动删除所有触发DMA请求的信号量。

void DFB_SetSemaphores(uint8 mask)

说明： 该函数设置数值为1的信号量。

参数： mask：指定需要进行设置的各个位的掩码。

值	说明
DFB_SEMAPHORE0	信号量0
DFB_SEMAPHORE1	Semaphore 1（信号量1）
DFB_SEMAPHORE2	Semaphore 2（信号量2）

返回值： 无

其他影响： 无

void DFB_ClearSemaphores(uint8 mask)

说明： 该函数清除数值为1的信号量。

参数： mask：指定需要清除的各个位的掩码。

值	说明
DFB_SEMAPHORE0	信号量0
DFB_SEMAPHORE1	Semaphore 1（信号量1）
DFB_SEMAPHORE2	Semaphore 2（信号量2）

返回值： 无

其他影响： 无

uint8 DFB_GetSemaphores(void)

说明： 该函数检查DFB信号量的当前状态，并返回该值。

参数： 无

返回值： 是位0和位7间的uint8值。其中，位0表示信号量0，等等。

值	说明
DFB_SEMAPHORE0	信号量0
DFB_SEMAPHORE1	Semaphore 1（信号量1）
DFB_SEMAPHORE2	Semaphore 2（信号量2）

其他影响： 无

void DFB_SetOutput1Source(uint8 source)

说明： 通过此函数可以选择映射到输出1的内部信号。

参数： source：映射到输出全局信号1的内部信号。

信号	说明
DFB_RUN	DFB运行位。该位与DFB_CR寄存器中的运行位相同。
DFB_SEM0	信号量位0。
DFB_SEM1	信号量位1。
DFB_DFB_INTR	DFB中断。该信号与主DFB中断输出信号相同。

返回值： 无

其他影响： 无

void DFB_SetOutput2Source(uint8 source)

说明： 通过此函数可以选择映射到输出2的内部信号。

参数： source：映射到输出全局信号2的内部信号。

信号	说明
DFB_SEM2	信号位2。
DFB_DPSIGN	数据路径标志。每当数据路径单元中的ALU输出为负值时，会激活此信号。在此条件为“True”的周期内，此信号保持高电平状态。
DFB_DPTHRASH	超出数据路径的阈值。每当超出ALU中的阈值0并且执行下面某一条指令：tdeca、tsuba、tsubb、taddabsa或taddabsb，则会激活此信号。在此条件为“True”的周期内，此信号保持高电平。
DFB_DPEQ	数据路径ALU = 0。每当数据路径单元中的ALU输出等于0，并且执行下面某一条指令：tdeca、tsuba、tsubb、taddabsa或taddabsb时，此信号被置为高电平。在该条件为“True”的周期内，此信号保持高电平。

返回值： 无

其他影响： 无

void DFB_Sleep(void)

说明： 这是组件准备进入睡眠模式的首选子程序。DFB_Sleep()例程保存当前的组件状态。然后调用DFB_Stop()函数，并调用DFB_SaveConfig()以保存硬件配置。

在调用CyPmSleep()或CyPmHibernate()函数之前，需要调用DFB_Sleep()函数。有关功耗管理函数的详细信息，请参考*《系统参考指南》*。

参数： 无

返回值： 无

其他影响： 无

void DFB_Wakeup(void)

说明： 该函数是将组件恢复到调用DFB_Sleep()时的状态的首选子程序。DFB_Wakeup()函数调用DFB_RestoreConfig()函数以恢复配置。如果组件在调用DFB_Sleep()函数前已经启用，则DFB_Wakeup()函数也将重新启用组件。

参数： 无

返回值： 无

其他影响： 调用DFB_Wakeup()函数前未调用DFB_Sleep()或DFB_SaveConfig()函数，则可能会产生意外行为。

void DFB_Init(void)

说明： 该函数初始化或恢复自定义程序提供的默认DFB组件配置：

- 给 DFB (PM_ACT_CFG)和 RAM (DFB_RAM_EN)供电
- 将 CSA/CSB/FSM/DataA/DataB/地址计算单元（ACU）的数据转移到使用 8051/ARM 内核的 DFB RAM 内
- 将 RAM DIR 改为 DFB
- 设置中断模式
- 设置 DMA 模式
- 设置 DSI 输出
- 清除所有信号量位和挂起中断

参数： 无

返回值： 无

其他影响： 所有寄存器将复位为各自的初始值。这将重新初始化组件。该函数关闭了运行位，并使能对DFB模块供电。

void DFB_Enable(void)

说明： 该函数启用DFB硬件模块、设置DFB运行位，并给DFB模块供电。

参数： 无

返回值： 无

其他影响： 无

void DFB_SaveConfig(void)

说明： 此函数会保存组件配置以及非保留寄存器。它还保存Configure（配置）对话框中定义的或通过相应API修改的当前器件参数值。通过DFB_Sleep()函数调用该函数。

参数： 无

返回值： 无

其他影响： 无

void DFB_RestoreConfig(void)

说明： 此函数会恢复组件配置以及非保留寄存器。它还将组件参数值恢复为在调用DFB_Sleep()函数之前的值。

参数： 无

返回值： 无

其他影响： 调用此函数前未调用DFB_Sleep()或DFB_SaveConfig()函数，则可能会产生意外行为。

定义

ClearInterruptSource(event) — 用于清除中断的宏

MISRA 合规性

本节介绍了 MISRA-C:2004 合规性和本器件的偏差情况。定义了下面两种类型的偏差：

- 项目偏差 — 适用于所有 PSoC Creator 组件的偏差
- 特定偏差 — 仅适用于该组件的偏差

本节提供了有关组件特定偏差的信息。在*系统参考指南*的“MISRA 合规性”章节中介绍了项目偏差以及有关 MISRA 合规性验证环境的信息。

DFB 组件没有任何特定偏差。

固件源代码示例

PSoC Creator 在“Find Example Project”（查找示例项目）对话框中提供了多种包括原理图和代码示例的示例工项目。要查看特定组件示例，请打开“Component Catalog”中的对话框或原理图中的组件实例。要查看通用示例，请打开“Start Page”或“**File**”菜单中的对话框。根据要求，可以通过使用对话框中的“**Filter Options**”选项来限制可选的项目列表。

更多有关信息，请参考《PSoC Creator 助手》部分中主题为“查找示例项目”的内容。

功能描述

数字滤波器模块是一个 24 位定点的范围受限的可编程 DSP。该模块具有一个 24*24 的乘法累加单元（MAC）、一个多功能算术逻辑单元（ALU），并且具有数据路由、移位、保留和取整等功能。

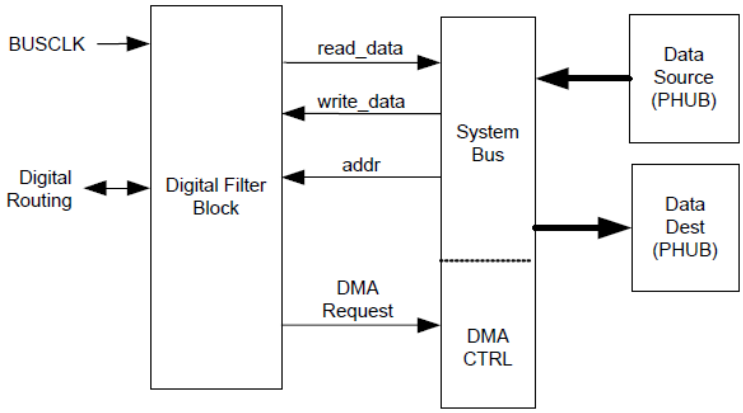
DFB 的其它重要特性如下：

- 两个 24 位宽的数据流通道
- 两套控制存储 RAM，每一套可以存储多达 64 个控制字
- 两套数据 RAM，每一套可以存储多达 128 个 24 位宽的字
- 用于计算数据 RAM 地址的地址计算单元（ACU）和两套 ACU RAM（每一套可以存储多达 16 个绝对数据 RAM 地址）
- 两套 32*32 的有限状态机的 RAM，用于存储控制流（分支）信息
- 一个中断和两个 DMA 请求通道
- 三个与系统软件交互的信号位
- 针对输入和输出寄存器中的数据对齐和一致性保护支持选项

DFB 支持两个数据流通道。其中编程指令、历史数据、滤波器系数以及结果与来自 AHB 接口的新接收定期数据均被本地存储。另外，系统软件可以‘对 DFB 数据 RAM 进行上传或下载样例或系数数据’，或在模块模式下重新编程不同的滤波器操作，或执行上述两个操作。这样，可以处理多个通道，并且与本地存储器相比，可以执行更深的过滤功能。该模块提供了软件可配置的中断，并且支持两个 DMA 通道。软件可通过三个信号量位与 DSP 汇编工具交互。

DFB 具有两个 24 位宽的输入分级寄存器以及两个 24 位宽的输出保留寄存器。可以通过 DFB 或 AHB 总线（CPU/DMA）访问这些寄存器。一般情况下，输入数据通过 CPU 或 DMA 被传输到分级寄存器内，输出则从 DFB 保留寄存器内流出。由于具有两套输入/输出寄存器，所以 DFB 很适合进行立体声数据处理应用（两个通道并行）。这些输入/输出寄存器支持 32 位、16 位和 8 位访问，同时具备一致性保护硬件特性，从而允许对它们进行小于 32 位的访问。

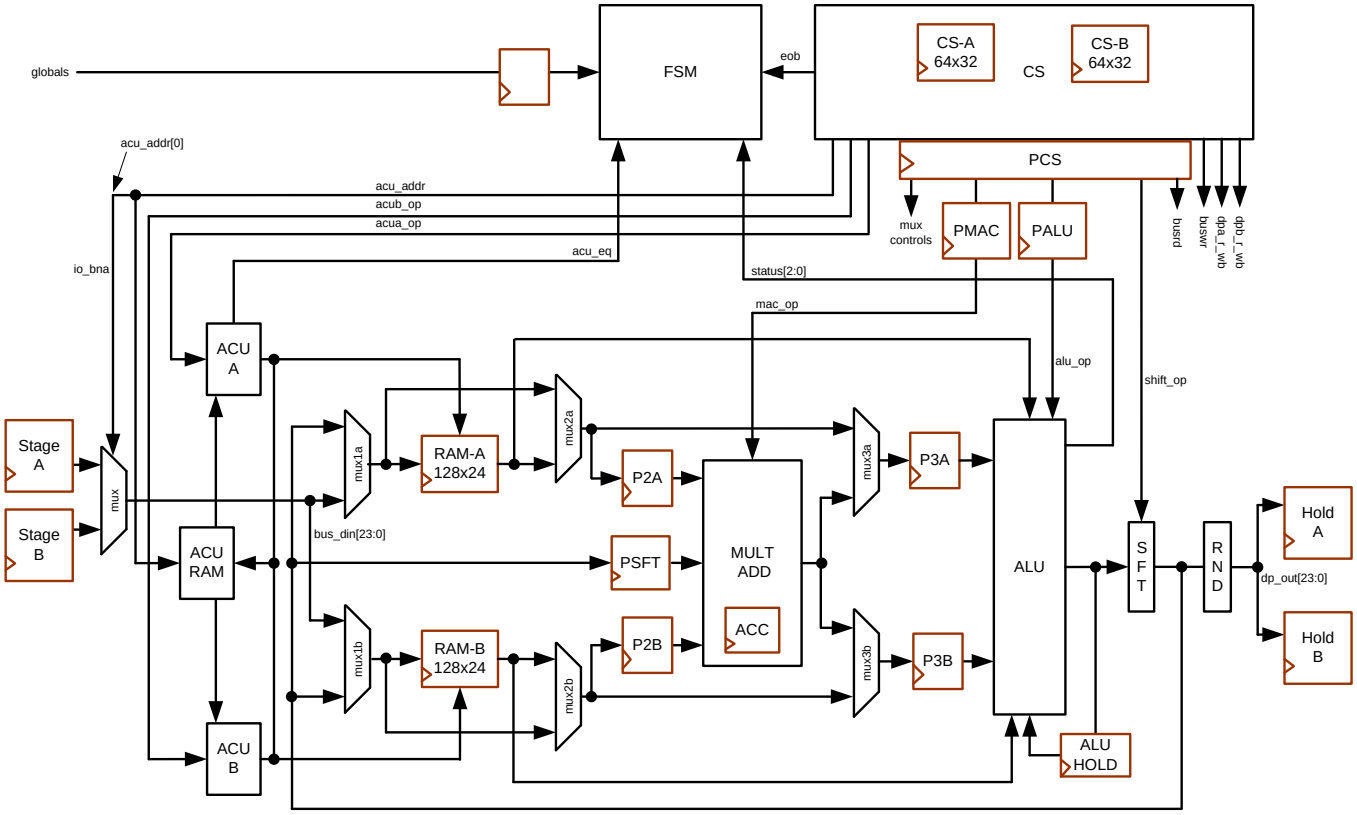
图 2. DFB 应用图



典型的使用模型是通过系统总线将数据从另一个片上系统数据源（例如ADC）提供给DFB。数据通常通过主存储器进行传输，或通过DMA直接从另一个芯片资源进行传输。

DFB架构支持ACU、ALU、MAC和移位器的并行操作。每个模块允许的操作均在位域中进行编码。这些函数中提供了基本的汇编指令集。DFB处理器的指令流水线如图3所示。该框图显示了流水线寄存器的位置，从而可以确定指令流水线的延迟。可以并行执行所有ACU/DPRAM、MAC和ALU/移位器，但将指令从一个模块传到下一个模块时会有一个周期的延时。例如，想要指定新的ACU地址，根据该地址进行乘累加，然后观察它的输出是否超出了阈值。您需要在第一、第二和第三个周期上分别指定ACU地址、MAC和阈值。如果控制器标志了数据路径的阈值用于某个分支，那么直到序列中的第四个周期才会发生分支操作。但其影响不大，因为可以设置算法来避免四周期延迟的分支操作。一般，由于可以将分支前的步骤与先前的语句结合在算法流程中，所以一个算法允许最后指令和分支之间有一个周期的延迟。各ACU被定位，以便在检测和分支操作之间不存在延迟。

图 3.数据流/流水线框图



可以选择编程任何信号量位，以符合系统中断信号、任一个DMA_REQ输出（从DFB输出）或任一个DSI信号（Out_1、Out_2）。

DFB和DATA RAM A/B存储器中的数据格式为二进制补码。DFB使用24位带符号的算术值。它的取值范围是0至16,777,215。DFB组件面向滤波算法，它的范围为–1至1。数值1（0.9999999）表示为0x7FFFFF（8388607），数值0表示为0x000000（0），数值–1表示为0x800000（8388608），–0.0000001表示为0xFFFFFF（16,777,215），0.0000001表示为0x000001（1）。第24位是数据的符号。

DFB 压缩器

通过使用优化功能（Optimize Assembly States选项卡），所有128个存储项可用于程序存储区。DFB允许在执行代码流时进行切换各64存储项的代码存储区，并制定0系统周期的循环和分支。当代码存储不相同时，可以对所有128个存储项进行编程。

压缩器将程序分为多个状态，然后将这些程序状态放置在两个控制存储区中的一个内。压缩器也生成各控制存储区间的跳转地址。一般情况下，进行各程序状态间的跳转时需要从一个存储区跳转到另一个。一个程序不能在同一个控制存储区内跳转。例如，有一个被称为“FILTER”的程序，它被划分为在RAM A中的不同状态。同时，您有两个子程序，即R1和R2，这两个子程序都要跳转到FILTER。如果R1和R2都位于RAM B，便可以正常操作。但是，如果有任意一个位于RAM

A，则程序便不能跳转。该情况会引起汇编错误。为了解决这个问题，输出面板要将下面的信息提供给压缩器：RAM A和RAM B的内容，以及控制有限状态机（CFSM）内容的说明。RAM A和RAM B的内容包含与各程序状态相关的信息。CFSM内容说明包含有关在各程序状态跳转的信息。

通过被添加的代码分析器，您可以查看代码行在控制存储区（cstore）内的位置。由于输入汇编语言是定向周期和行的，所以汇编语言中的行和 cstore 内的存储项之间具有一一对应的关系。这样，您可以收集有关优化代码的有用信息。

某个DFB程序的汇编状态优化出了问题时，汇编后会显示如下的错误警告：

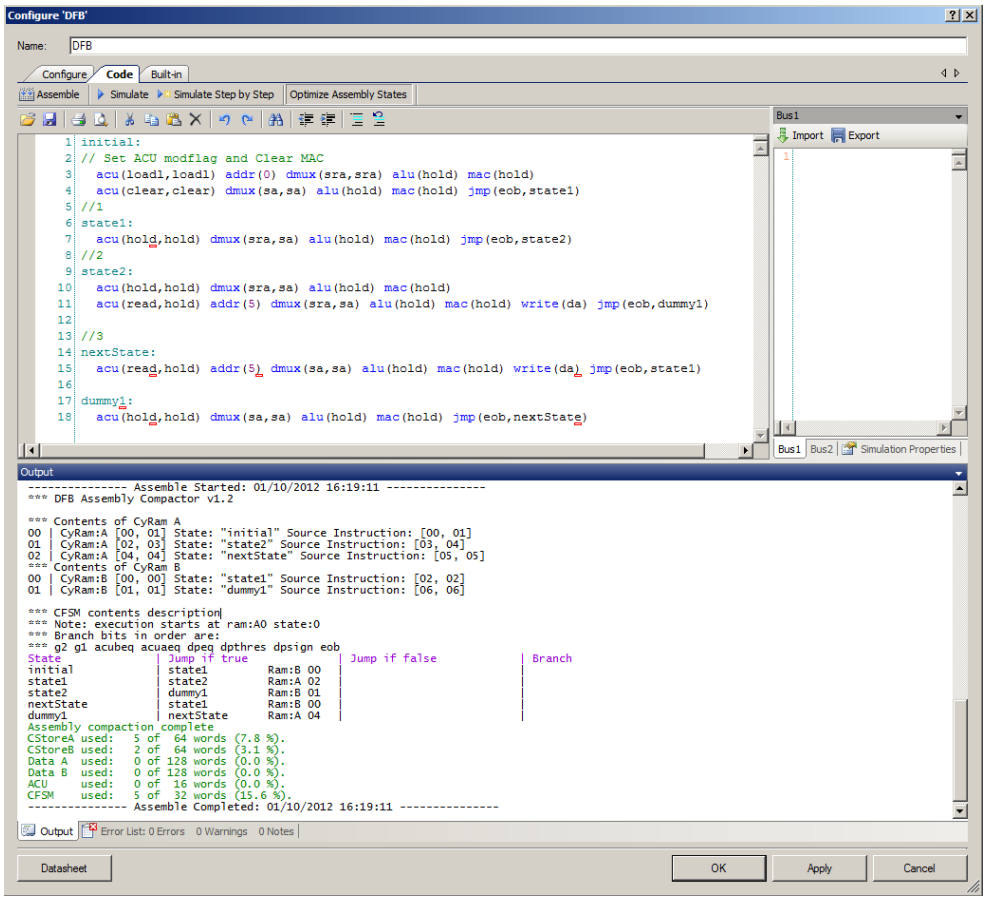
“ERROR: Unable to map to split RAMs. Found N states that can't be mapped. Please analyze results and verify code can be mapped.”

状态的索引为-01 时，该文件指出代码位于错误 RAM 中。您可以查看 RAM 和状态的信息，从而推断所遇到的问题。

通过使用日志文件，可以构建流程的框图，并且确定模块的分配情况。想要解决问题，您可以将额外的虚拟状态添加到您的程序中。在遇到上述问题时，需要单一虚拟状态和单一指令。例如，您的程序不能跳到 nextState 状态时，您可以跳转到虚拟状态 dummy1 中。从这个状态的位置，您可以跳转到 nextState。请参考下面的单一指令。

```
dummy1:
acu(hold,hold) dmux(sa,sa) alu(hold) mac(hold) jmp(eob, nextState)
```

下面的屏幕截图显示的是该过程的实例。



资源

DFB组件使用硅片中的专用DFB硬件模块。

API 存储器使用情况

根据编译器、器件、所使用的 API 数量以及组件的配置不同，组件对存储资源的占用也不一样。下表提供了在某种器件配置中所有 API 占用存储器的大小。

数据是在将编译器设置为 Release 模式并将优化等级设置为 Size 的情况下测得的。对于特定的设计，分析完编译器生成的映射文件后可以确定组件占用存储器的大小。

配置	PSoC 3（Keil_PK51）		PSoC 5LP（GCC）	
	闪存字节	SRAM字节	闪存字节	SRAM字节
默认值	1618 + DFB程序的大小*	9	1232 + DFB程序的大小[1]	9

DFB数据存储器(RAM)					
DATA A	DATA B	CS A	CS B	FSM	ACU
128x24	128x24	64x32	64x32	64x32[2]	16x14

可以通过DFB或系统（CPU/DMA）AHB总线访问不同的DFB RAMS，但是不能同时执行两个方式。如果需要将大量数据转到各个DFB RAM内，那么先将DFB RAM的控制权转给系统AHB总线（CPU/DMA），将新数据加载到DFB RAM内，然后再将控制权转给DFB。DFB_RAM_DIR寄存器控制着通过DFB还是通过系统总线（CPU/DMA）来访问DFB RAM。

RAM的名称	大小	功能
DATA A	128x24	样例/系数存储
DATA B	128x24	样例/系数存储
CS A	64x32	控制存储
CS B	64x32	控制存储
FSM	64x32	有限状态机的 RAM
ACU	16x14	地址存储

1. DFB 程序的大小受 DFB 数据存储器大小的限制，如下表所示。
2. 对于当前的 DFB 执行操作，只能使用 FSM 存储器的一半（32x32）。

直流和交流电的电气特性

除非另有说明，否则这些规范的适用条件是：-40 °C ≤ TA ≤ 85 °C 且 TJ ≤ 100 °C。除非另有说明，否则这些规范的适用范围为 1.71 V 到 5.5 V。

直流规范

参数	说明	条件	最小值	典型值	最大值	单位
	DFB工作电流	F_{DFB}时为64抽头FIR				
		100 kHz（1.3 ksps）	–	0.03	0.05	mA
		500 kHz （6.7 ksps）	–	0.16	0.27	mA
		1 MHz（13.4 ksps）	–	0.33	0.53	mA
		10 MHz（134 ksps）	–	3.3	5.3	mA
		48 MHz（644 ksps）	–	15.7	25.5	mA
		67 MHz（900 ksps）	–	21.8	35.6	mA
		80 MHz （1.07 Msps）（仅用于PSoC 5LP）	–	26.1	42.5	mA

交流电规范

参数	说明	条件	最小值	典型值	最大值	单位
F_{DFB}	DFB工作频率	PSoC 3	DC	–	67.01	MHz
		PSoC 5LP	DC	–	80.01	MHz

DFB 汇编器

指令说明

AREA

“area”指令的参数用于指定某一段 RAM 的可访问性。已选定的 RAM 可以包括括号，也可以不包括括号：

```
area(RAM_Name) or area RAM_Name
```

需要通过使用 dw（定义字）指令对 data_a、data_b 和 acu 三个 RAM 进行访问和修改。可以对控制和 CFSM RAM 进行访问。但汇编器能够输入操作码时，对控制和 CFSM RAM 进行手动编码则不会产生任何作用。

注释

汇编器的注释格式与 C 语言行注释的格式相同（占用整个行）。

```
// Designates the line to be a comment line. Everything is
// ignored by the assembler
```

ORG

使用“org”指令可以为当前 RAM 设置当前地址计数器（CLC）。最初，每个 RAM 上的 CLC 被设置为 0。每个地址的值必须是一个整数，并且是代表有效内存地址的一组数字。下面两项是该指令可用的格式。

```
org(location) or org location
```

dw

表示定义字。指令后面的参数表示放置在当前节的存储器和 CLC 内的值。然后 CLC 递增。当程序写入的值超过 RAM 的最大值时，自定义程序将生成并显示一个错误信息。“acu”区允许输入前缀为“0x”的十六进制参数。这样可以明确区分 ACU RAM 中的 A 侧和 B 侧的值。数据区允许使用二进制补码格式输入从 $1\text{-}2^{-23}$ 到–1 的 24 位整数（0 到 16、777、215）。在 ACU RAM 中，两个 7 位侧允许输入 4 位十六进制格式的数据。这样可以清楚 14 位宽的 RAM 两侧的数值，即前两位数位于 A 侧，后两位数位于 B 侧。每侧的有效值范围是 0x00 到 0x7F。

示例：

```
dw 0x123F      // (Decimal 18 in ACU RAM side A, 63 in side B)
```

标签

指引用 cstore 代码模块的用户定义标签。代码模块包括多个节，它们开始于有效标签，并以一个跳转指令结束。其中，标签和跳转指令耦合在一起构成 CFSM 中的某个状态。除了本文档所列出的关键字（作为指令使用）外，标签还可以通过一组字符定义，这个字符组以一个字母开始，最后一位是一个冒号。请勿将声明和标签放在同一行上。执行 DFB 硬件指令时，必须将声明放在标签/跳转模块内；这两个标签/跳转模块耦合在一起构成一个状态。两个标签之间必须使用跳转终止模块进行区分。

下面是一个声明示例：

```
// MyLabel defines a new state for the cfsm. The location of
// the state’s start in Cstore is attached
MyLabel:
```

VLIW 指令

每个指令行将各操作码定义为一个 32 位的超长指令字（VLIW）。下面显示的是单一指令字中最通用的指令形式，必须遵循该顺序：每行中必需的指令使用粗体字显示。下面概述每个单一指令。应该在括号间输入合适指令集中的某一条指令。ACU 和 DMUX 指令都需要两个指令（这两个指令由逗号分离），第一个指令使用于数据路径 A 侧，第二个指令使用于 B 侧。

```
acu(,) addr() dmux(,) alu() mac() shift() write() jump()
```

每个指令为 VLIW 提供了一个短操作码，它们一起构成每行的 32 位指令控制字。

ACU

地址计算单元（ACU）输出数据 RAM 地址，使用于下一条指令周期。单个 ACU 基本上是一个具有四个寄存器的计数器，这些寄存器的默认值为 0。

- **reg —** reg 保存 ACU 正在运行的当前值，并在每个周期内输出该值，除非某一指令指定使用另一个数值。
- **freg** — 使用 addf 和 subf 指令时，freg 可以加载数据 RAM 递增或递减的值。例如：通过使用 ACU 的 addf 指令将 2 加载到 freg 后，数据 RAM 递增 2。
- **mreg —** 使能模算术时，mreg 保存包裹 lreg 值前的最大值。
- **mreg** — 使能模算术时，lreg 保存包裹 mreg 值前的最小值。

模算术防止 ACU 进行递增时的值超过 mreg 的值，并防止其递减时的值小于 lreg 的值。使能模算术，另外在当前地址处于 lreg 到 mreg 范围外时，ACU 将产生意外结果（然而已确定）。良好的 DFB 编程实践需要您通过使用“读”指令或仔细检查保证 ACU 指向的值是以一个有效的位置开始。

16 行深的 RAM 伴随着 ACU，这样是为了保存执行程序时所需要的各个数值，这些值用于存储数据 RAM 节的绝对地址。另外，它还保存着 ACU 将要访问的值，如 freg 的值。在运行过程中，将数据保存在 ACU RAM 中的首选方法是使用系统软件进行干预。

ACU 指令的使用定义如下：

```
acu(instruction_A, instruction_B)
```

两个不同的指令 instruction_A 和 instruction_B 是 ACU 指令集的成员，它们单独控制着两个数据 RAM 的地址。下表显示的是 ACU 指令集的完整列表。

指令	说明
hold	将输出上的寄存输出地址值设置为不变。
incr	将寄存输出地址值（reg）递增1，并把它置于输出端。
decr	将寄存输出地址值（reg）递减1，并把它置于输出端。
read	读取ACU RAM所指定的字节，并将该值加载到输出地址寄存器内，然后把它置于输出端。（请参考addr指令部分的内容。）
write	将寄存输出地址值置于已指定的ACU RAM行。（请参考addr指令部分的内容。）
loadf	将指定ACU RAM中的值加载到freg内。ACU输出值与前周期的值相同。（有关ACU RAM寻址的内容，请参考addr指令部分的内容。）
loadl	将指定ACU RAM中的值加载到lreg内。ACU输出值与前周期的值相同。（有关ACU RAM寻址的内容，请参考addr指令部分中的内容。）
loadm	将指定ACU RAM中的值加载到mreg内。ACU输出值与前周期的值相同。（有关ACU RAM寻址的内容，请参考addr指令部分的内容。）
writel	将lreg值置于输出端，然后把输出值写入到指定的ACU RAM地址内。（请参考addr指令部分的内容。）
setmod	使能ACU中的模算术。默认情况下，模算术被使能。
unsetmod	未使能ACU中的模算术。默认情况下，模算术被使能。
clear	将寄存输出值（reg）设置为0。
addf	使用寄存器freg中的值递增寄存输出地址值（reg）。未使能模算术，并且ACU输出处于lreg和mreg的定义范围外时，请勿使用该指令。
subf	使用寄存器freg中的值递减寄存输出地址值（reg）。未使能模算术，并且ACU输出处于lreg和mreg的定义范围外时，请勿使用该指令。
writem	将mreg值置于输出上，然后将输出写到指定的ACU RAM地址上。（请参考addr指令部分的内容。）
writef	将freg值置于输出上，然后将输出写入到指定的ACU RAM地址上。（请参考addr指令部分的内容。）

addr

addr 指令参数的取值范围为 0 到 15。可以通过各种方法使用该指令。编写程序时需要注意，相同指令中的各条指令不应多次访问 addr 值。如果未重新定义 addr 值，对该 addr 进行多次访问将不会生成错误信息，但会生成警告信息。

可以通过下列五种不同的方式使用 addr 指令。您不能同时选用多个方式。

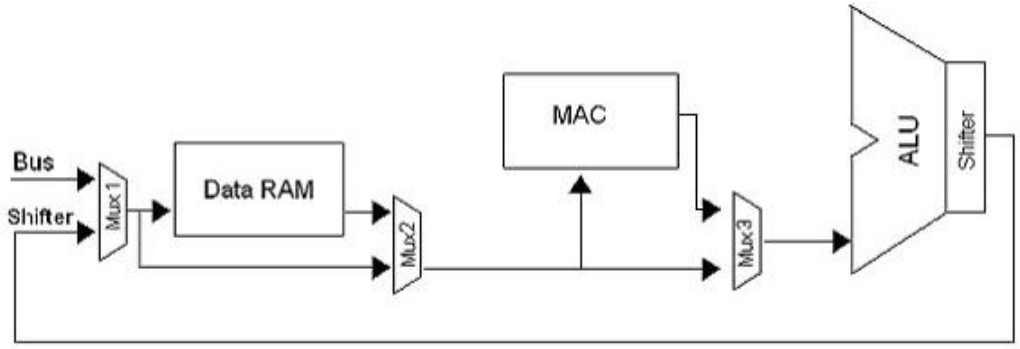
- 访问 ACU RAM 中的某一地址。您可指定单个 ACU RAM 地址，该地址由两个 ACU 访问。（两侧必须访问 ACU RAM 的相同行。A 侧不能从行 1 进行读取，B 侧从行 14 进行读取。）
- 指定将要进行读或写操作的输入和输出通道。选择通道 1 作为奇数的 addr 值（如果 addr 的最低有效位为 1），选择通道 2 作为偶数的 addr 值。有关输入通道和输入分级寄存器的信息，请参考 dmux 部分的内容，有关输出分级寄存器的信息，请参考写部分的内容。
- 提供一个数值，作为写入信号量，并使能或禁用作为跳转条件的信号量。可通过信号量指令中指定的 3 位字段自动设置该 addr 值。（请参考 ALU 中的内容。）
- 使能和禁用饱和及舍入标志以及清除饱和检测标志，将隐藏 addr 值的明确定义并自动定义它。（请参考 ALU 中的内容。）
- 使能和禁用作为跳转条件的全局中断将定义 addr 值（该值为 englobal 指令中的 2 位字段）。（请参考 ALU 中的内容。）。如果将 addr 指令作为 addr(1)调用，A 侧和 B 侧都将访问 ACU RAM 的行 1。

  如果该指令执行总线读取操作，则将从分级寄存器 1 读取总线数据；如果该指令执行总线写入操作，则数据将被写入到输出分级寄存器 1 内。

  该示例说明了对 addr 指令值进行多重访问时所造成的问题。一旦定义了 addr 值，对它进行的所有访问都必须使用相同的值，否则自定义程序将返回一个错误信息。明确定义 addr 时，可以防止使用信号量（全局跳转条件的使能事项），并防止饱和及舍入寄存器的指令。

dmux

图 4. 数据路径



控制存储 RAM （cstore）中的六位输出字控制数据路径复用，并指定通过数据路径的数据。MAC、ALU 和输出移位器为 A 侧和 B 侧提供三个复用级。与 ACU 相同，dmux 也有一个使用于 A 侧和 B 侧的参数。

```
dmux(instruction_A, instruction_B)
```

当 mux1（参考图 4）允许对总线进行访问时，它将使用在输入寄存器中等待的任何值。如果还未完全确定该系统，代码将等待控制器通知处于等待状态的值可用。等待状态将一直循环，直到输入通道的就绪跳转条件有效为止。

下表显示的是 dmux 指令集的完整列表。

指令	说明
ba	总线到ALU。mux1将总线数据传递到数据RAM内。mux2绕过数据RAM并将mux1数据直接传递到MAC和mux3内。mux3绕过MAC并将mux2直接传递到ALU输入。根据addr的声明为0或1，总线数据是两个分级寄存器中的一个。
sa	从移位器到ALU。mux1将移位器输出数据传递到数据RAM内。mux2绕过数据RAM并将mux1数据直接传递到MAC和mux3内。mux3绕过MAC并将mux2直接传递到ALU输入。
bra	从总线到RAM，从RAM到ALU。 mux1将总线数据传递到数据RAM内。 mux2将数据RAM中的数据传递到MAC和mux3。 mux3绕过MAC将mux2直接传递到ALU输入。根据addr的声明为0或1，总线数据是两个分级寄存器中的一个。
sra	从移位器到RAM，从RAM到ALU。mux1将移位器输出数据传递到数据RAM内。mux2将数据RAM中的数据传递到MAC和mux3内。mux3绕过MAC并将mux2直接传递到ALU输入。
bm	从总线到MAC，从MAC到ALU。mux1将总线数据传递到数据RAM内。mux2绕过数据RAM并将mux1数据直接传递到MAC和mux3内。mux3将MAC输出传递到ALU输入。根据addr的声明为0或1，总线数据是两个分级寄存器中的一个。
sm	从移位器到MAC，从MAC到ALU。mux1将移位器输出数据传递到数据RAM内。mux2绕过数据RAM并将mux1数据直接传递到MAC和mux3内。mux3将MAC输出传递到ALU输入。
brm	从总线到RAM和MAC。mux1将总线数据传递到数据RAM内。mux2将数据RAM中的数据传递到MAC和mux3内。mux3将MAC输出传递到ALU输入。总线数据来自于两个分级寄存器中的一个。寄存器的选择取决于addr的声明（0或1）。
srm	从移位器到RAM和MAC。mux1将移位输出数据传递到数据RAM内。mux2将数据RAM中的数据传递到MAC和mux3内。mux3将MAC输出传递到ALU输入。

ALU

ALU 在数据路径输出端上提供了数据控制。除了通用功能（如加法和减法）以外，ALU 可以设置标志以通知满足某些特定的各状态间跳转条件。alu 指令包括五个特殊指令，这些指令需要有一个长度为 3 位字段的输入数据。ALU 输出被直接传递到移位寄存器内。

“alu”指令有两种格式选项：

```
alu(instruction) or alu(special_instruction, 3-bit_field)
```

下表显示的是 alu 指令集的完整列表。

指令	说明
set0	将ALU输出设置为0。
set1	将ALU输出设置为1的整数值。这意味着，最低有效位为1，其他所有位为0。
seta	将输入A传递到输出端。
setb	将输入B传递到输出端。
nega	对A进行取反并将A传递到输出端。
negb	对B进行取反并将B传递到输出端。
passrama	将RAM A当前地址的值传递到输出端。
passramb	将RAM B当前地址的值传递到输出端。
add	计算‘A + B’并将结果放置在ALU输出端。
tdeca	计算‘A – 1’并将结果放置在ALU输出端。如果该值为0，将设置阈值检测。通过该指令，在某一值递减计数到0时，可以在一定的时间内保持等待状态。该指令适用于低功耗下的等待模式。
suba	计算‘B – A’并将结果放置在ALU输出端。
subb	计算‘A – B’并将结果放置在ALU输出端。
absa	计算‘|A|’并将结果放置在ALU输出端。
absb	计算‘|B|’并将结果放置在ALU输出端。
addabsa	计算‘|A| + B’并将结果放置在ALU输出端。
addabsb	计算‘A + |B|’并将结果放置在ALU输出端。
hold	保持前一个周期中ALU输出值。

指令	说明

englobals	确定是否使能两个全局中断和饱检测标志作为状态修改的跳转条件。该设置取决于指令和分隔逗号后面的3位字段。全局中断位 [1:0]是DFB的输入。当饱和逻辑保持某一值的最大或最小的数据路径值时，环绕式处理没有发生，那么将设置饱和检测标志（由bit [2]饱和检测启用）。
	该指令根据位字段的输入自动设置addr值。“englobals”指令和“ensatrnd”指令共同使用一个ALU操作码。在硬件中，这些指令的性能通过addr操作码的值确定。如果程序尝试使用不同的值定义addr指令，自定义程序将生成错误信息。
	`alu(englobals, [Saturation detection enable, Global interrupt 2 enable, Global interrupt 1 enable] )`
	例如，下面的指令禁用全局中断2及饱和，并使能全局中断1作为跳转条件。
	`alu(englobals, 001)`

ensatrnd	通过写入饱和及舍入寄存器内使能或禁用数据路径中的饱和及舍入。使用指令和分隔逗号后面3位字段中的 [1:0] 位写入饱和及舍入寄存器内。3位字段的[2]位用于探针饱和检测标志并清除它。该指令会自动使用并设置addr值。另外，该指令和englobals指令共同使用一个操作码；在硬件中，这两个指令的性能通过addr操作码的值确定。如果程序尝试使用不同的值定义addr指令，则自定义程序将生成错误信息。
	`alu(ensatrnd, [Clear saturation detection flag, Saturation detection enable, Rounding enable] )`
	例如，下面的ALU指令用来打开舍入、关闭饱和及清除饱和检测标志（若被设置）。
	`alu(ensatrnd, 001)`

ensem	根据指令和分隔逗号后面的3位字段使能所指定的信号量作为跳转条件。该指令根据3位字段的值自动使用并设置addr值。因此，如果该程序通过使用该指令尝试将addr指令定义为不同的值，则自定义程序将生成错误信息。延迟时间段为两个周期，自从该指令执行到这些更改变成有效跳转条件为止。信号量作为跳转条件时，可将条件“sem”作为一个提示使用，以便提醒程序员已经设置了信号量条件。
	`alu(ensem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	如果程序不再需要将信号量作为跳转条件使用，它必须通过调用该指令（此时必须将字段中的每一位都设置为0）来清除使能标志。
	`alu(ensem, 000)`

setsem	将3位字段中被屏蔽的信号量设置为1。复位后，请勿在第一个指令中使用“setsem”，否则将重复设置信号量。该指令会自动使用并设置addr值。如果程序尝试使用不同的值定义addr指令，则自定义程序将生成错误信息。
	`alu(setsem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	下面示例将信号量2设置为“true”（1），并保持信号量1和0的原有设置。
	`alu(setsem, 100)`

指令	说明

clearsem	清除3位字段中被1屏蔽的信号量。（将该信号量设置为0）。该指令会自动使用并设置addr值。如果程序尝试使用不同的值定义addr指令，则自定义程序将生成错误信息。
	`alu(clearsem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	
	下面示例将信号量2设置为“false”（0），并保持信号量1和0的原有设置。
	`alu(clearsem, 100)`

tsuba	计算‘B – A’并将结果放置在输出端上。设置阈值检测。
tsubb	计算‘A – B’并将结果放置在输出端上。设置阈值检测。
taddabsa	计算‘|A| + B’并将结果放置在输出端上。设置阈值检测。
taddabsb	计算‘A + |B|’并将结果放置在输出端上。设置阈值检测。
sqlcmp	将A侧上mux3中的值加载到比较寄存器内，以便作为抑制函数中的截断使用。
sqlcnt	将A侧上mux3中的低16位加载到16位计数寄存器内。如果输出端的当前值未满足抑制比较寄存器设置的阈值，则每次调用抑制指令时，该值都会递减。如果满足该阈值，则该值将复位到它的原始值。
sqa	读取A侧上mux3中的值，并将该值与抑制比较寄存器中的值进行比较。如果当前值大于比较寄存器的值，它将被传递到输出上，并且抑制计数寄存器将复位它的原始值。 如果当前值小于比较寄存器的值，则该指令将检查抑制计数器。如果计数器的值不是0，它将递减，另外A侧上mux3中的当前值将被传递到输出上。如果计数寄存器的值为0，0值将被传递到输出端。
sqb	读取B侧上mux3中的值，并将该值与抑制比较寄存器中的值进行比较。如果当前值大于比较寄存器的值，它将被传递到输出端，而且抑制计数寄存器将复位它的原始值。 如果当前值小于比较寄存器的值，则该指令将检查抑制计数器。如果计数器的值不是0，它将递减。另外B侧上mux3中的当前值将被传递到输出端。如果计数寄存器的值为0，0值将被传递到输出端。

MAC

乘法与累加单元。包含一个硬件，用于对两个定点数字进行乘法计算，然后将该值加上前一值。‘(A × B) + C’

MAC 指令集有四个成员，它们的使用如下所示：

```
mac(instruction)
```

下表显示的是 MAC 指令集的完整列表。

指令	说明
loadalu	将从移位器的前一个ALU输出添加到该乘积，并启动新的累积。
clra	清除累加器，并存储当前的乘积。
hold	保持前个周期内累加器中的值。无乘法。
macc	乘法与累加。将A侧和B侧的mux2上的各个值相乘。然后，将该乘积添加给累加器的当前值。

移位

移位指令允许定标 ALU 输出。有效的移位指令需要两个参数，即方向和数量级，并评估它们，以产生正确的操作码。有效的方向指令包括： “right”（右）、“left”（左）、“r”和“l”。右移位允许 1、2、3、4 和 8 共五个数量级，而左移位则仅允许 1 和 2 两个数量级。ALU 的输出通过移位器传递，并退回到数据路径的开始位置，不管是否发生了移位。

```
shift(direction, magnitude)
```

下表显示的是移位指令集的完整列表。

指令	说明
right、r	相应的指令决定移位的方向。
left、l	相应的指令决定移位的方向。

写

有效的写指令有 0 到 3 个参数。对于每个参数都将写入某一值。您可以将任何一侧的 mux1 上的值写入到相应的数据 RAM 内，也可以将移位器的当前输出值写入到输出端上的分级寄存器内。使用写入指令前，需要检查 DFB 数据路径中的各种流水线的延迟。

如果复位后所使用的第一个指令包含总线写指令，将导致系统级的影响。汇编器不允许在第一个指令中进行写操作，这样是为了防止发生意外问题。

下面都是有效的写指令。

- write（da、db、bus）

- write（da、db）
- write（db）

下表显示的是写指令集的完整列表。

指令	说明
da	执行该行的ACU指令后，会将mux1A的值写入到所指定的data_a Ram位置，并且在执行写操作时没有延迟。
db	执行该行的ACU指令后，会将mux1B的值写入到所指定的data_b Ram位置，并且在执行写操作时没有延迟。
abus	将移位器的输出值写入到总线的保持寄存器A内。addr被定义为1时，将选择保持寄存器A（请参考addr节中的内容）。
bbus	将移位器的输出值写入到总线的保持寄存器B内。addr被定义为0时，将选择保持寄存器B（请参考addr节中的内容）。
总线	将移位器的输出值写入到总线保持寄存器内。有两个可用的输出保持寄存器。根据addr定义为1或0，选择寄存器（请参考addr节中的内容）。该指令可用于兼容性；在新项目中，请勿使用总线指令。使用该指令会产生警告：“Potential addr() conflict attempting write(bus). Avoid this warning by using channel-specific bus write commands.”

跳转指令

跳转指令允许代码将它的位置转到不同的子程序。跳转指令的通用形式为：

JumpType(conditions, Target Routine)其中：“条件”是指任何的使可限制或允许跳转的有序列表。跳转选项的说明后面紧接着的是与这些标志有关的文字描述。请注意跳转指令的位置，因为某些情况下需要两个周期而不是一个后，控制器才可以将它们作为有效的跳转。此外，复位后不能在第一个指令中使用跳转指令，因为第一个状态的长度必须为两个指令，以便设置流水线。

跳转指令图部分中包含一个跳转条件图。

下表显示的是跳转指令集的完整列表。

指令	说明
jmp	“Jump”（跳转）类似于一个标准的“goto”指令。如果条件为真，则代码将跳转到目标程序。否则，它会下降到下一个数字状态。如果不在一个循环中，该fjlim（假跳限制）值被设置为最大cstore位置。 jmp（eob，符号，……，目标状态）

指令	说明
jmpl	“Jump Loop”（跳转循环）将当前的代码模块设置为一个循环。这个循环是双向分支，其中，一个分支是目标程序（如果满足条件）；另一个分支是当前代码模块的起始（如果不满足条件）。 处于一个循环时，会发生下面的事项： ▪ 将 CFSM 的位 23 设置为高电平。 ▪ 将假跳地址（FJADDR）定义为当前代码块的开始，它是标签的控制存储地址。（标签不存储在控制库存；标签引用代码块的第一个指令。） ▪ 将假跳限制（FJLIM）设定为当前的 CLC 位置，它是代码块的结尾。 ▪ （所有跳转类型相同）条件为真实时，通过使用目标程序的标签可以提供下一个状态的跳转地址（JADDR）以及 CFSM RAM 位置。 执行指令，直到检测到eob为止。程序将评估条件。如果条件为假，会将程序计数器设置为FJADDR，并重新启动模块。如果条件为真实，则程序计数器被设为JADDR，并且状态被更新为“NextStateOnTrue”。该指令的格式与jmp指令的格式相同。 jmpl（条件，目标状态）
jmpsl	“Jump to Subroutine Loop”（跳转到子程序循环）允许跳转到一个循环的子程序代码块。jmpsl在当前的代码块上的影响与jmp指令相同。如果满足了各条件，代码将跳转到所指定的子程序。否则，代码执行时会降低到下一个状态。然而，jumsl会影响到子程序的状态。每当引用子程序时，会创建子程序的副本作为CFSM的状态，并且设置新状态的属性。对于jmpsl指令，所创建的状态被指定为一个循环，并具有代码空间中下一个状态的返回状态。（当前状态是以jmpsl指令结束的状态。）欲了解更多有关状态及子程序，请参考该表中的jmpret项。 jmpsl（条件，目标子程序）
jmpslr	“跳转到子程序环路并返回状态”，在各方面与jmpsl几乎相同，只是会返回所指定的状态，而不是默认返回当前状态的下一个状态。 jmpslr（条件，…，目标子程序，返回状态）
jmps and jmpsr	这两个指令是jmpsl和jmpslr的复制， 不同的是所创建的子程序的状态不是一个循环。因此，在jmpret语句中仅指定了eob条件。
jmpret	仅通过使用各条“跳转到子程序”指令中的某一条才能访问子程序。子程序不同于标准状态，因为其属性通过调用它们的状态（而不是状态末尾的跳转条件）决定。如果某个状态通过跳转指令“jmpslr(eob, sub1, anotherState)”来调用子程序，则它会将子程序定义为一个循环，并且下一个状态即为“anotherState”。 通过某个jmpret类型（返回跳转）的跳转指令，可以终止各个子程序。该类型的指令为子程序提供了循环终止的跳转条件集（如果子程序被调用为一个循环）。子程序不能调用其他子程序，因为jmpret指令不向子程序提供必要的退出信息。 将jmpret指令作为一个jmp指令使用，但没有指定目标状态。 jmpret（条件，条件，…）

跳转条件

表示使能或禁止修改状态代码的条件。下面各条件其实是硬件的使能标志。列出某个条件时，使能的信号补偿必须是真实的，以便进行跳转。

注意：数据路径条件需要两个周期的延迟。换句话说，条件必须在两个周期内为真后，跳转指令才会识别此条件为真。

下表显示的是跳转条件的完整列表。

指令	说明
eob	模块的结束。这是跳转的条件。因为跳转指令表示模块已结束，该条件始终被满足。只在发生某个无条件跳转时，才需要指定eob。这是软件的限制，而不是硬件的限制。
dpsign	基于ALU输出的MSB的一个跳转。ALU输出为负值时，将确认该跳转。数据路径需要两个周期的延迟才会满足跳转条件。
dpthresh	数据路径阈值。ALU检测到某个符号发生改变时，将确认该阈值。仅在程序使用ALU阈值检测操作数（tsuba、tsubb、taddabsa、……）时，ALU才会激活dpthresh。数据路径需要两个周期的延迟才会满足跳转条件。
dpeq	数据路径的权益。当ALU硬件检测到一个输出值0时，将激活它。仅在程序通过使用ALU阈值检测操作数（tsuba、tsubb、taddabsa、……）时，ALU才会激活dpeq。数据路径需要两个周期的延迟才会满足跳转条件。
acuaeq	ACU A的平衡。当ACU A检测到某个包裹条件时，将激活它。如果禁用了模算术，ACU A的平衡可以是0或数据RAM的最大位置。当使能模算术时，则可以是模数计数器的上限值或下限值。数据路径需要一个周期的延迟才会满足跳转条件。
acubeq	ACU B的平衡。当ACU B检测到某一个包裹条件时，将激活它。它可以是0值或模数计数器的限制。数据路径需要一个周期的延迟才会满足跳转条件。
in1	通道1输入寄存器值就绪信号。激活该信号时，可以使用一个新的输入周期。该信号仍被激活，直到由一个总线读取操作清除为止。数据路径需要一个周期的延迟才会满足跳转条件。
in2	通道2输入寄存器值就绪信号。激活该信号时，可以使用一个新的输入周期。该信号仍被激活，直到由一个总线读取操作清除为止。数据路径需要一个周期的延迟才会满足跳转条件。
sem	sem条件不会影响到操作码。它提高了代码的清晰度，并提醒编程器：当前的跳转条件是信号量。（更多详细信息，请参见ALU部分的内容。）
globals	globals条件不会影响到操作码。它提高了代码的清晰度，并提醒编程器：当前的跳转条件是全局输入。（更多详细信息，请参见ALU章节中的内容）
sat	“sat”条件不会影响到操作码。它提高了代码的清晰度，并提醒程序员：当前需要饱和事件来使能跳转操作。（请参考ALU中的内容，了解详细的信息。）

指令

DMUX 指令

代码	名称	函数复用1	函数Mux 2	函数Mux 3
0	ba	总线寄存器	总线寄存器	总线寄存器
1	sa	前移位器的输出	前移位器的输出	前移位器的输出
2	bra	总线寄存器	当前RAM值	当前RAM值
3	sra	前移位器的输出	当前RAM值	当前RAM值
4	bm	总线寄存器	总线寄存器	MAC累加器
5	sm	前移位器的输出	前移位器的输出	MAC累加器
6	brm	总线寄存器	当前RAM值	MAC累加器
7	srm	前移位器的输出	当前RAM数值	MAC累加器

MAC 指令

代码	名称	函数
0	loadalu	将ALU的值加上该乘积，并启动新的累积计算。
1	clra	清除累加器。使用当前的乘积加载它。
2	hold	保持累加器，无乘法（即无乘幂）计算。
3	macc	标准操作 — 与前值进行乘法及累加计算

ACU 指令

代码	名称	函数
0	hold	将reg放在输出端上。
1	incr	将‘reg + 1’放在输出端上，并对reg进行写操作。
2	decr	将‘reg – 1’放在输出端上，并对reg进行写操作。
3	read	从ACU RAM加载reg，并将此值放在输出端上。
4	write	将reg放在特定的ACU RAM列。
5	loadf	从ACU RAM加载freg，并将reg放在输出端上。
6	loadl	从ACU RAM加载lreg，并将reg放在输出端上。
7	loadm	从ACU RAM加载mreg，并将reg放在输出端上

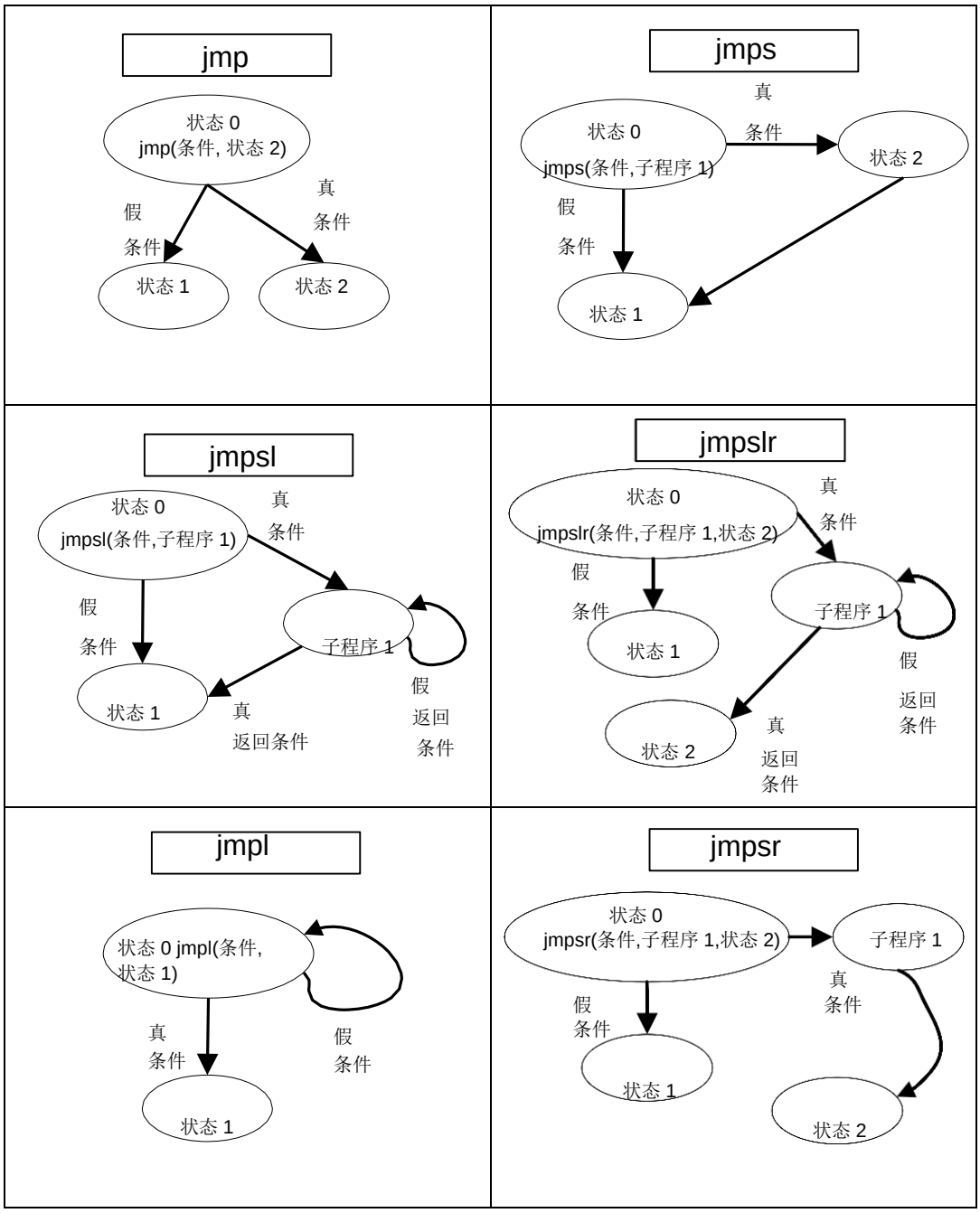
代码	名称	函数
8	writel	将lreg放在输出端上，并对ACURAM进行写操作。
9	setmod	将算术设置为模数mreg。
10	unsetmod	将算术设置为包裹。
11	clear	将reg设置为0，并将0放在输出端。
12	addf	将reg和freg相加，然后将结果放在输出端，并存储在reg寄存器中。
13	subf	将reg减去freg，然后将结果放在输出端，并存储在reg寄存器中。
14	writem	将mreg放在输出端，并对ACU RAM进行写操作。
15	writef	将freg放在输出端，并对ACU RAM进行写操作。

ALU 指令

代码	名称	函数
0	set0	将ALU输出设置为0。
1	set1	将ALU输出设置为1。
2	seta	将A传递到ALU输出端。
3	setb	将B传递到ALU输出端。
4	nega	将ALU输出设置为‘–A’。
5	negb	将ALU输出设置为‘–B’。
6	passrama	直接将RAM A输出传递到ALU输出端。
7	passramb	直接将RAM B输出传递到ALU输出端。
8	add	将A加上B，然后将结果放在ALU输出端。
9	tdeca	将‘A – 1’放在ALU输出端，并设置阈值检测。
10	suba	将‘B – A’放在ALU输出端。
11	subb	将‘A – B’放在ALU输出端。
12	absa	将‘|A|’放在ALU输出端。
13	absb	将‘|B|’放在ALU输出端。
14	addabsa	将‘|A|+ B’放在ALU输出端。
15	addabsb	将‘A + |B|’放在ALU输出端。
16	hold	保持前一个周期的ALU输出。

代码	名称	函数
17	englobals	通过使用三位字段来使能全局及饱和跳转条件，以确定作为有效跳转条件的事件。
17	ensatrnd	使用三位字段写入到各个饱和及舍入使能寄存器内，这样可以使能或禁用它们。
18	ensem	通过使用一个三位字段使能信号量，使其作为跳转条件，以便指定有效的信号量。
19	setsem	使用三位屏蔽将信号量设为高电平。
20	clearsem	使用屏蔽（addr[2:0]）将信号量设为低电平。
21	tsuba	将‘B – A’放在ALU输出端，并设置阈值检测。
22	tsubb	将‘A – B’放在ALU输出端，并设置阈值检测。
23	taddabsa	将‘|A| + B’放在ALU输出端，并设置阈值检测
24	taddabsb	将‘A + |B|’放在ALU输出端，并设置阈值检测。
25	sqlcmp	将A侧的值加载到抑制比较寄存器内，并且传递B侧的值。
26	sqlcnt	将A侧的值加载到抑制计数寄存器内，并且传递B侧的值。
27	sqa	抑制A侧。如果该值超过阈值，便可以传递它。如果该值低于阈值，并且抑制计数寄存器为零，则传递0值。
28	sqb	抑制B侧。如果该值超过阈值，便可以传递它。如果该值低于阈值并且抑制计数寄存器为零，则传递0值。
29-31	undefined	未定义的操作码

跳转指令图


jmp
状态 0
jmp(条件, 状态 2)
假
条件
真
条件
状态 1
状态 2
jmps
真
状态 0
jmps(条件,子程序 1)
条件
状态 2
假
条件
状态 1
jmpsl
状态 0
jmpsl(条件,子程序 1)
真
条件
假
条件
子程序 1
状态 1
真
返回条件
假
返回
条件
jmpslr
状态 0
jmpslr(条件,子程序 1,状态 2)
真
条件
假
条件
子程序 1
状态 1
假
返回
条件
状态 2
真
返回
条件
jmpl
状态 0 jmpl(条件,
状态 1)
真
条件
假
条件
状态 1
jmpsr
状态 0
jmpsr(条件,子程序 1,状态 2)
子程序 1
真
条件
假
条件
状态 1
状态 2

组件更改

本节列出了该组件各版本中的主要更改内容。

版本	更新内容	更改原因/影响
1.30.a	对数据手册的少量更新。	替代质量不好的数据流/流水线框图。
1.30	对DFB_SetCoherency()和DFB_SetDalign() API加以说明，以表示这些函数被直接写入寄存器。	API使用的模糊说明。
	更改了位序信息和下面各ALU指令的使用：englobals、ensatrnd、setsem、clearsem	在模糊例子中，指令的位顺序不正确，并且需要推断。
	将功能说明中的仿真器输出移转到Code选项卡内	所需的信息流动性
1.20	在数据手册中，添加了仿真器输出信息	
	进行了更改，以使其符合MISRA合规性	MISRA合规性具有某些全局偏差
	更新了DFB LoadDataRAMx() API源	提高数据传输的速度
1.10	更新了直流和交流电气特性章节。	
	添加了PSoC5LP器件支持。	

©赛普拉斯半导体公司，2014 - 2015。此处，所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品内嵌的电路外，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，否则赛普拉斯不保证产品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，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®和 CapSense®是注册商标；SmartSense™、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。除上述指定的用途外，未经赛普拉斯明确的书面许可，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。对于合理预计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于赛普拉斯软件许可协议的限制。