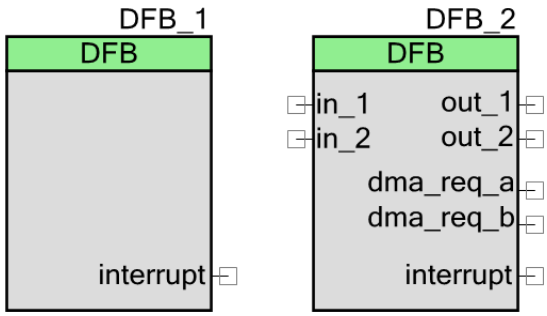
Digital Filter Block (DFB) Assembler

1.40

Features



- Provides an editor to enter the assembler instructions to configure the DFB block
- Provides an assembler that converts the assembly instructions to instruction words
- Supports simulation of the assembly instructions
- Supports a code optimization option that provides a mechanism to incorporate up to 128 very large instruction words inside the DFB Code RAM
- Provides hardware signals such as DMA requests, DSI inputs and outputs, and interrupt lines
- Supports semaphores to interact with the system software and the option to tie the semaphores to hardware signals

General Description

The digital filter block (DFB) in PSoC 3 and PSoC 5LP is a 24-bit fixed point, programmable limited scope DSP engine that can be used as a mini DSP processor in your application. The DFB component allows you to directly configure the DFB using its assembly instructions. The component assembles the instructions entered in the code editor and generates the corresponding hex code words that are then loaded into the DFB. It also includes a simulator, which can aid in simulating and debugging the assembly instructions.

When to Use a Digital Filter Block (DFB) Assembler

The DFB consists of a programmable 24*24 multiplier/accumulator (MAC), an arithmetic logic unit (ALU), shifter, and various program and data memory to store instructions and data. The DFB runs on the bus clock and can interface with both CPU and DMA. It can be used to offload the CPU and can speed up arithmetic calculations that involve intensive multiply accumulate operations. Typical operations you can use the DFB component to implement include: vector operations, matrix operations, filtering operations, and signal processing.

See the Functional Description for details about the DFB.

Input/Output Connections

This section describes the input and output connections of the DFB component. An asterisk (*) in the list of I/Os means that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

in_1 – Input*

Input terminal - Used to allow some DFB control and visibility to other resources on the chip, particularly the UDBs. This input displays when the **Input 1** option is selected in the **Configure** dialog.

in_2 – Input*

Input terminal - Used to allow some DFB control and visibility to other resources on the chip, particularly the UDBs. This input displays when the **Input 2** option is selected in the **Configure** dialog.

out_1– Output*

Output terminal - Allows the DFB signals to control other on-chip resources, particularly the UDBs. This output displays when the **Output 1** option is selected in the **Configure** dialog.

out_2– Output*

Output terminal - Allows the DFB signals to control other on-chip resources, particularly the UDBs. This output displays when the **Output 2** option is selected in the **Configure** dialog.

dma_req_a – Output*

DMA request output signal - Associated with either data being ready in the holding register A or semaphore bits being set. This is particularly useful to trigger DMA channels. The output displays when the **DMA Request A Source** option is selected in the **Configure** dialog.

If the DMA request signal is configured to be associated with the output holding register A, it generates a level-sensed signal to the DMA that is cleared when the register is read.

If the DMA request signal is configured to be generated by a semaphore, it creates a single-cycle high pulse.

dma_req_b – Output*

DMA request output signal - Associated with either data being ready in the holding register B or semaphore bits being set. This is particularly useful to trigger DMA channels. The output displays when the **DMA Request B Source** option is selected in the **Configure** dialog.

If the DMA request signal is configured to be associated with the output holding register B, it generates a level-sensed signal to the DMA that is cleared when the register is read.

If the DMA request signal is configured to be generated by a semaphore, it creates a single-cycle high pulse.

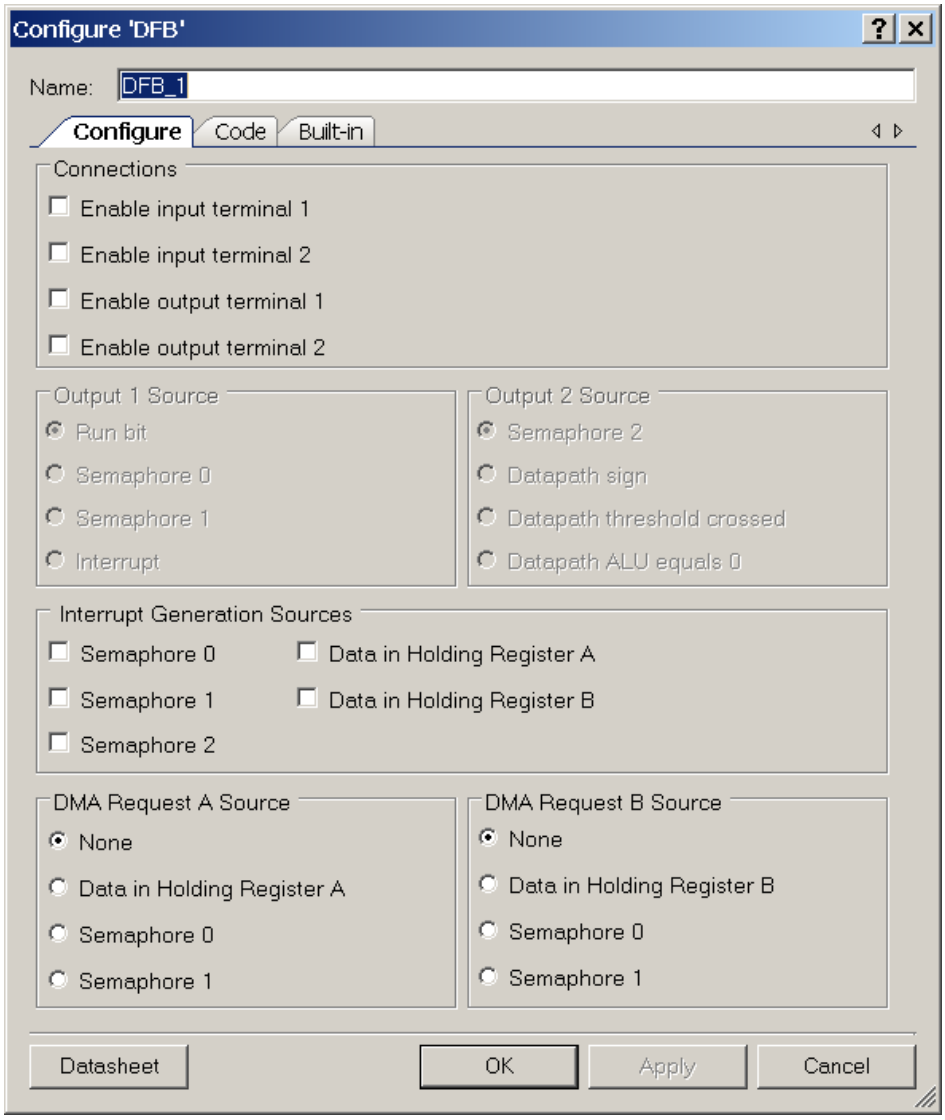
interrupt – Output

Output of system interrupts. It can be associated with either of the holding registers and with semaphore bits.

Component Parameters

Drag a DFB onto your design and double-click it to open the **Configure** dialog. The **Configure** dialog contains several tabs with different parameters to set up the DFB component.

Configure Tab



Connections

- **Enable input terminal 1**
- **Enable input terminal 2**
- **Enable output terminal 1**
- **Enable output terminal 2**

Output 1 Source

Determines the internal signal that is mapped to the output global signal 1.

- **Run bit** – This is the same bit as the RUN bit in the DFB_CR register.
- **Semaphore 0**
- **Semaphore 1**
- **Interrupt** – This is the same signal as the primary DFB Interrupt output signal.

Output 2 Source

Determines the internal signal that is mapped to the output global signal 2.

- **Semaphore 2**
- **Datapath sign** – This signal asserts any time the output of the ALU in the datapath unit is negative. It remains high for each cycle this condition is true.
- **Datapath threshold crossed** – This signal asserts any time the threshold of 0 is crossed in the ALU during the execution of any one of the following instructions: tdeca, tsuba, tsubb, taddabsa, or taddabsb. It remains high for each cycle that this condition remains true.
- **Datapath ALU equals 0** – This signal asserts high when the output of the ALU in the Datapath unit equals 0 during the execution of any one of the following ALU commands: tdeca, tsuba, tsubb, taddabsa, or taddabsb. It remains high for each cycle that this condition remains true.

Interrupt Generation Sources

Configures the events on which the interrupt will be generated:

- **Semaphore 0**
- **Semaphore 1**

- **Semaphore 2**
- **Data in Holding Register A**
- **Data in Holding Register B**

DMA Request Mode

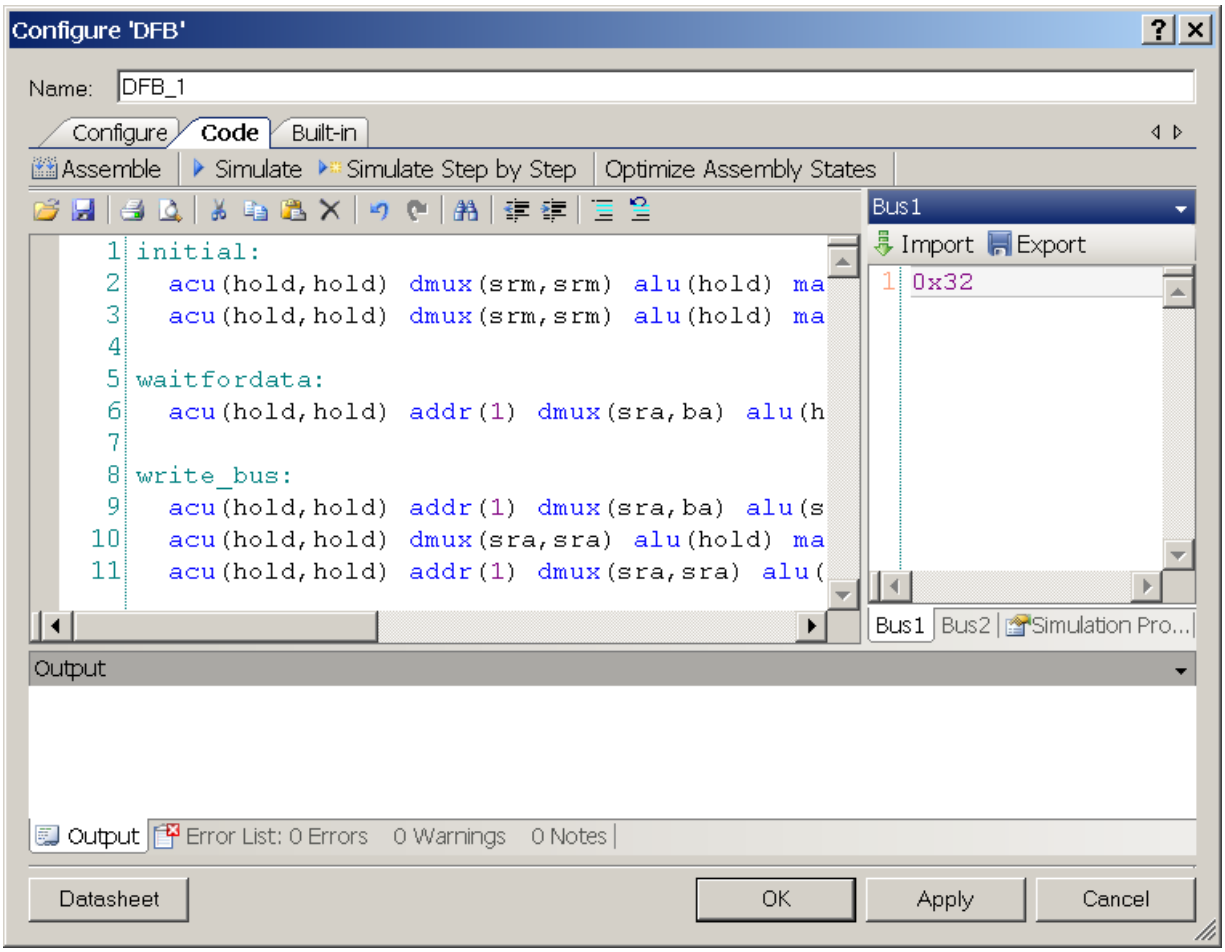
Configures DMA Request sources - If **DMA Request** is not equal to **None**, a DMA request output terminal is created.

- **DMA Request A Source:**
  - None
  - Data in Holding Register A
  - Semaphore 0
  - Semaphore 1
- **DMA Request B Source:**
  - None
  - Data in Holding Register B
  - Semaphore 0
  - Semaphore 1

These are controlled using CSR configuration. When a DMA semaphore is programmed as a DMA_REQ, the HW converts any write of a ‘1’ to that semaphore to a single-cycle strobe. This clears the semaphore after one cycle. So, if a semaphore will be used as a source for a DMA request signal, it is cleared automatically.

Code Tab

The **Code** tab provides an editor to enter the assembler instructions, an assembly mechanism to verify and assemble code, and a simulator to simulate instructions.



Assemble

Assembles the entered DFB assembler instructions. Status and error messages are displayed in the **Output** window of the customizer. Keyboard shortcut – [**F6**]

Simulate Continuously

The simulator runs until Bus Input Data (Bus1 Data and Bus2 Data) is exhausted. Bus1 data and Bus2 data are the input data for the simulation that corresponds to the data that will be streamed to the DFB staging registers. Simulated contents of DFB sub-blocks after each instruction execution are displayed in the **Output** window. Keyboard shortcut – [**F5**]

After simulation starts, use [**Shift**] [**F5**] to stop it.

Simulate Step by Step

The simulator runs step by step per instruction line until the Bus Input Data is exhausted. At each step, the lines of code being executed are highlighted in the code editor. Simulation properties are displayed in the **Simulation Properties** panel. Keyboard shortcut – [**F7**]

After simulation starts, use [**F8**] to execute the next simulation step.

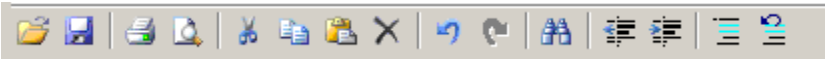
Optimize Assembly States

Enables the compactor feature. The compactor removes the 64-word program size barrier, allowing you to incorporate up to 128 instruction words inside the DFB Code RAM. If the **Optimize Assembly States** option is selected, the code compacting feature runs after successful code assembly.

Code Editor

Displays DFB assembler code with highlighting. It highlights comments, labels, commands, and values with different colors.

Figure 1. Text Editor Toolbar



Text Editor

Open file: Opens a text file with DFB assembler code. Keyboard shortcut – [**Ctrl**] [**O**]

Save file: Saves a text file with DFB Assembler code. Keyboard shortcut – [**Ctrl**] [**S**]

Other text editor features: Includes cut, copy, paste, undo, redo, print, print preview, find/replace text, comment, uncomment, and other functionality.

Output panel

Displays log information for the assembly, compactor, and simulator processes. Lines with error text are highlighted red and lines that indicate successful operations are highlighted green. If you double-click on a line with an error text, the customizer automatically activates the **Code Editor** and selects the line where the error is located.

Use the keyboard shortcut [**Alt**] [**C**] to clear the log.

Simulator Output

The output of the simulator contains information that can be used to debug the design. The data inputs coming into the DFB are simulated using the Bus1 (Stage register A) and Bus2 (Stage register B) windows located at the top right of the **Code** tab. Semaphores, global inputs, and global outputs can be simulated by modifying the contents in the **Simulation Properties** window. Note that the API calls made in the firmware do not affect the simulation properties.

Therefore, these behaviors need to be replicated in the simulator. These include populating the Data RAMs, writing to the Staging registers, reading from the Holding registers, and setting/unsetting the semaphores and global inputs.

Signal Name	Type	Description
Cycle	uint	This is the program counter that counts the number of clock cycles.
RamA	uint	The current RAM location in Control Store RAM A, also referred to as CStoreA.
RamB	uint	The current RAM location in Control Store RAM B, also referred to as CStoreB.
Ram sel	string	The Control Store RAM (either A or B) currently being executed.
CFSM state	uint	Current state of the CFSM. This value is not the same as the RAM values of the CFSM content. The states are ordered as they appear in the assembler code. Therefore the first state to be called in the code is designated CFSM=1, and the second state to be called is designated CFSM=2 etc.
Aaddr next	uint	The current address of Data RAM A
Baddr next	uint	The current address of Data RAM B
A2Mux	Hex	The output from Mux2 (A), which corresponds to either the output of Mux1 (A) or the current content in the Data Ram A.
B2Mux	Hex	The output from Mux2 (B), which corresponds to either the output of Mux1 (B) or the current content in the Data Ram B.
MacOut	Hex	The output of the Multiply and Accumulate Unit (MAC).
AluOut	Hex	The output of the Arithmetic Logic Unit (ALU).
ShiftOut	Hex	The output of the shifter located at the output of the ALU.

Error List

Displays a list of errors, warnings, and notes. If you double-click on an error, the customizer automatically activates the **Code Editor** and selects the line where the error is located.

Bus1

Provides data for STAGEA input for the simulator. Enter it as a 24-bit value in hex, decimal, or binary format. For example, 99 is decimal, 0x63 is hex, and 0b1100011 is binary.

Bus2

Provides data for STAGEB input for the simulator. Enter it as a 24-bit value in hex, decimal, or binary format.

Bus data import

Imports data to the Bus1/Bus2 text fields. Supports .txt and .data (an old data format that was used in the C-based implementation of the simulator) file formats.

Bus data export

Exports data from the Bus1/Bus2 text fields. Supports .txt and .data file formats.

Simulator Properties

Provides functionality to change the input values and semaphores and view the internal simulator values between simulations.

- **GlobalInput1**: Read/Write field
- **GlobalInput2**: Read/Write field
- **Semaphore0**: Read/Write field
- **Semaphore1**: Read/Write field
- **Semaphore2**: Read/Write field
- **Cycle**: Displays current cycle number
- **RamA Index**: Read-only field that represents the current index in RAM A.
- **RamB Index**: Read-only field that represents the current index in RAM B.
- **Ram Selected**: Read-only field that shows which ram (A or B) is currently executed

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections discuss each function in more detail.

By default, PSoC Creator assigns the instance name “DFB_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “DFB.”

Function	Description
DFB_Start()	Initializes and enables the DFB component using the DFB_Init() and DFB_Enable() functions.

Function	Description
DFB_Stop()	Turns off the run bit. If DMA control is used to feed the channels, allows arguments to turn off one of the TD channels.
DFB_Pause()	Pauses DFB and enables writing to the DFB RAM.
DFB_Resume()	Disables writing to the DFB RAM, clears any pending interrupts, disconnects the DFB RAM from the data bus, and runs the DFB.
DFB_SetCoherency()	Sets the coherency key to low/mid/high byte based on the coherencyKey parameter that is passed to the DFB.
DFB_SetDalign()	Allows 9- to 16-bit input and output samples to travel as 16-bit values on the AHB bus.
DFB_LoadDataRAMA()	Loads data to RAMA DFB memory.
DFB_LoadDataRAMB()	Loads data to RAMB DFB memory.
DFB_LoadInputValue()	Loads the input value into the selected channel.
DFB_GetOutputValue()	Gets the value from one of the DFB output holding registers.
DFB_SetInterruptMode()	Assigns the events that will trigger a DFB interrupt.
DFB_GetInterruptSource()	Looks at the DFB_SR register to see which interrupt sources have been triggered.
DFB_ClearInterrupt()	Clears the interrupt request.
DFB_SetDMAMode()	Assigns the events that will trigger a DMA request for the DFB.
DFB_SetSemaphores()	Sets semaphores specified with a 1.
DFB_ClearSemaphores()	Clears semaphores specified with a 1.
DFB_GetSemaphores()	Checks the current status of the DFB semaphores and returns that value.
DFB_SetOutput1Source()	Chooses which internal signals will be mapped to output 1.
DFB_SetOutput2Source()	Chooses which internal signals will be mapped to output 2.
DFB_Sleep()	Prepares the DFB component to go to sleep.
DFB_Wakeup()	Prepares DFB Component to wake up.
DFB_Init()	Initializes or restores the default DFB configuration provided with the customizer.
DFB_Enable()	Enables the DFB hardware block. Sets the DFB run bit. Powers on the DFB block.
DFB_SaveConfig(void)	Saves the user configuration of the DFB nonretention registers. This routine is called by DFB_Sleep() to save the component configuration before entering sleep.
DFB_RestoreConfig()	Restores the user configuration of the DFB nonretention registers. This routine is called by DFB_Wakeup() to restore the component configuration when exiting sleep.

Global Variables

Variable	Description
DFB_initVar	Indicates whether the DFB has been initialized. The variable is initialized to 0 and set to 1 the first time DFB_Start() is called. This allows the component to restart without reinitialization after the first call to the DFB_Start() routine. If reinitialization of the component is required, then the DFB_Init() function can be called before the DFB_Start() or DFB_Enable() function.

void DFB_Start(void)

Description: This function initializes and enables the DFB component using the DFB_Init() and DFB_Enable() functions.

Parameters: None

Return Value: None

Side Effects: None

void DFB_Stop(void)

Description: This function turns off the run bit. If DMA control is used to feed the channels, DFB_Stop() allows arguments to turn off one of the TD channels.

Parameters: None

Return Value: None

Side Effects: Disables power to the DFB core.

void DFB_Pause(void)

Description: This function pauses the DFB and enables writing to the DFB RAM.

- Turns off the run bit
- Connects the DFB RAM to the data bus,
- Clears the DFB run bit and passes the control of all DFB RAMs onto the bus

Parameters: None

Return Value: None

Side Effects: None

void DFB_Resume(void)

Description: This function disables writing to the DFB RAM, clears any pending interrupts, disconnects the DFB RAM from the data bus, and runs the DFB. It passes the control of all DFB RAM to the DFB and then sets the run bit.

Parameters: None

Return Value: None

Side Effects: None

void DFB_SetCoherency(uint8 coherencyKeyByte)

Description: This function sets the coherency key to low, med, or high byte based on the coherencyKeyByte parameter that is passed to the DFB. Note that the function directly writes to the DFB Coherency register. Therefore the coherency for all the registers must be specified when passing the coherencyKeyByte parameter.

DFB_SetCoherency() allows you to select which of the three bytes of each of STAGEA, STAGEB, HOLDA, and HOLDB will be used as the key coherency byte. Coherency refers to the HW added to this block to protect against block malfunctions. This is needed in cases where register fields are wider than the bus access, which leaves intervals when fields are partially written or read (incoherent). The key coherency byte is the way the SW tells the HW which byte of the field will be written or read last when you want to update the field. When the key byte is written or read, the field is flagged coherent. If any other byte is written or read, the field is flagged incoherent.

Parameters: uint8 coherencyKeyByte: Specifies bits in the DFB Coherency register.

Value	Description
DFB_STGA_KEY_LOW	The key coherency byte of the Staging A register is a low byte.
DFB_STGA_KEY_MID	The key coherency byte of the Staging A register is a med byte.
DFB_STGA_KEY_HIGH	The key coherency byte of the Staging A register is a high byte.
DFB_STGB_KEY_LOW	The key coherency byte of the Staging B register is a low byte.
DFB_STGB_KEY_MID	The key coherency byte of the Staging B register is a med byte.
DFB_STGB_KEY_HIGH	The key coherency byte of the Staging B register is a high byte.
DFB_HOLDA_KEY_LOW	The key coherency byte of the Holding A register is a low byte.
DFB_HOLDA_KEY_MID	The key coherency byte of the Holding A register is a med byte.
DFB_HOLDA_KEY_HIGH	The key coherency byte of the Holding A register is a high byte.
DFB_HOLDB_KEY_LOW	The key coherency byte of the Holding B register is a low byte.
DFB_HOLDB_KEY_MID	The key coherency byte of the Holding B register is a med byte.
DFB_HOLDB_KEY_HIGH	The key coherency byte of the Holding B register is a high byte.

Return Value: None

Side Effects: Coherency affects data loading using the DFB_LoadInputValue() function and data retrieval using the DFB_GetOutputValue() function.

Note Default key byte configuration for Staging A and B, and Holding A and B registers is high byte. The coherency for all registers must be specified using an OR operation and passed to coherencyKeyByte. Failure to do so and passing coherency for select registers may result in unintended behavior.

void DFB_SetDalign(uint8 dalignKeyByte)

Description: This feature allows 9- to 16-bit input and output samples to travel as 16-bit values on the AHB bus. These bits, when set high, cause an 8-bit shift in the data to all access of the corresponding staging and holding registers. Note that this function directly writes to the DFB Data Alignment register. Therefore the alignment for all registers must be specified when passing the dalignKeyByte parameter.

Since the DFB datapath is MSB aligned, it is convenient for the system SW to align values on bits 23:8 of the Staging and Holding register to bits 15:0 of the bus. This is because a transfer from a PHUB spoke that is 16 or even 8 bits wide goes to the DFB spoke which is 32 bits wide. The Dalign allows the DFB to justify data so that transfers to and from these different size spokes can happen more efficiently.

Parameters: uint8 dalignKeyByte: Specifies bits in the DFB Data Alignment register.

Value	Description
DFB_STGA_DALIGN_LOW	Writes normally
DFB_STGA_DALIGN_HIGH	Writes shifted left by eight bits
DFB_STGB_DALIGN_LOW	Writes normally
DFB_STGB_DALIGN_HIGH	Writes shifted left by eight bits
DFB_HOLDA_DALIGN_LOW	Reads normally
DFB_HOLDA_DALIGN_HIGH	Writes shifted left by eight bits
DFB_HOLDB_DALIGN_LOW	Reads normally
DFB_HOLDB_DALIGN_HIGH	Writes shifted left by eight bits

Return Value: None

Side Effects: None

Note: The coherency for all registers must be specified using an OR operation and passed to dalignKeyByte. Failure to do so may result in unintended behavior.

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

Description: This function loads data to the DFB RAM A memory.

Parameters: uint32 * ptr: Pointer on data source for load
uint32 * addr: Start address for loading the data in DFB RAM A.
uint8 size: Number of data words to load.

Return Value:

Value	Description
DFB_SUCCESS	Loading data is successful.
DFB_NAME`_ADDRESS_OUT_OF_RANGE	Error code: indicates that the address is out of range.
DFB_DATA_OUT_OF_RANGE	Error code: indicates a data overflow error.

Side Effects: This function does not stop the DFB if it is already started. The recommended method is to call DFB_Init(), DFB_LoadDataRAMA(), then DFB_Enable().

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

Description: This function loads data to DFB RAM B memory.

Parameters: uint32 * ptr: Pointer on data source for load
uint32 * addr: Start address for loading the data in DFB RAM B
uint8 size: Number of data words to load

Return Value:

Value	Description
DFB_SUCCESS	Loading data is successful
DFB_NAME`_ADDRESS_OUT_OF_RANGE	Error code: indicates that the address is out of range
DFB_DATA_OUT_OF_RANGE	Error code: indicates a data overflow error.

Side Effects: This function does not stop the DFB if it is already started. The recommended method is to call DFB_Init(), DFB_LoadDataRAMB(), then DFB_Enable().

void DFB_LoadInputValue(uint8 channel, uint32 sample)

Description: This function loads the input value into the selected channel.

Parameters: channel: Use either DFB_CHANNEL_A (1) or DFB_CHANNEL_B (0) as arguments to the function.

sample: 24-bit, right-justified input sample

Return Value: None

Side Effects: None

Note: The write order is important. When the high byte is loaded, the DFB sets the input ready bit. Pay attention to byte order if coherency or data alignment is changed.

int32 DFB_GetOutputValue(uint8 channel)

Description: This function gets the value from one of the DFB Output Holding registers.

Parameters: channel: Use either DFB_CHANNEL_A (1) or DFB_CHANNEL_B (0) as arguments to the function.

Return Value: The current output value in the chosen channel's holding register. This is a 24-bit number packed into the least-significant three bytes of the output word, or 0xFF000000 for invalid channel numbers

Side Effects: None

Note: Because of the architecture of the DFB, any value read from the holding A or B registers will be MSB aligned unless shifted otherwise by the datapath shifter. Pay attention to byte order if coherency or data alignment is changed.

void DFB_SetInterruptMode(uint8 events)

Description: This function assigns the events that trigger a DFB interrupt.

Parameters: events: Bits[0:5] of events represent the events that trigger DFB interrupts.

Value	Description
DFB_HOLDA	Interrupt is generated each time new valid data is written into output holding register A.
DFB_HOLDB	Interrupt is generated each time new valid data is written into output holding register B.
DFB_SEMA0	Interrupt is generated each time a '1' is written into semaphore register bit 0.
DFB_SEMA1	Interrupt is generated each time a '1' is written into semaphore register bit 1.
DFB_SEMA2	Interrupt is generated each time a '1' is written into semaphore register bit 2.

Return Value: None

Side Effects: None

Note: Do not configure semaphore 0 and semaphore 1 for both a DMA request and for an interrupt event. This is because, after one clock cycle, the system automatically clears any semaphore that triggered a DMA request.

uint8 DFB_GetInterruptSource(void)

Description: This function looks at the DFB_SR register to see which interrupt sources have been triggered.

Parameters: None

Return Value: uint8 value in which bits[0:5] of represent the events that triggered the DFB interrupt

Value	Description
DFB_HOLDA	Holding register A is a source of the current interrupt.
DFB_HOLDB	Holding register B is a source of the current interrupt.
DFB_SEMA0	Semaphore register bit 0 is a source of the current interrupt.
DFB_SEMA1	Semaphore register bit 1 is a source of the current interrupt.
DFB_SEMA2	Semaphore register bit 2 is a source of the current interrupt.

Side Effects: None

void DFB_ClearInterrupt(uint8 interruptMask)

Description: This function clears the interrupt request.

Parameters: interruptMask: Mask of interrupts to clear

Value	Description
DFB_HOLDA	Clear interrupt from holding register A. (Reading holding register A also clears this bit.)
DFB_HOLDB	Clear interrupt from holding register B. (Reading holding register B also clears this bit.)
DFB_SEMA0	Clear interrupt from semaphore register bit 0.
DFB_SEMA1	Clear interrupt from semaphore register bit 1.
DFB_SEMA2	Clear interrupt from semaphore register bit 2.

Return Value: None

Side Effects: Clearing semaphore interrupts also clears semaphore bits.

void DFB_SetDMAMode(uint8 events)

Description: This function assigns the events that trigger a DMA request for the DFB. Two different DMA requests can be triggered.

Parameters: events: A set of four bits that configure what event, if any, triggers a DMA request for the DFB.

Value	Description
DFB_DMAREQ1_DISABLED	No request is generated
DFB_DMAREQ1_HOLDA	Output value ready in the holding register on channel A
DFB_DMAREQ1_SEM0	Semaphore 0
DFB_DMAREQ1_SEM1	Semaphore 1
DFB_DMAREQ2_DISABLED	No request is generated
DFB_DMAREQ2_HOLDB	Output value ready in the holding register on channel B
DFB_DMAREQ2_SEM0	Semaphore 0
DFB_DMAREQ2_SEM1	Semaphore 1

Return Value: None

Side Effects: None

Note: Do not configure semaphore 0 and semaphore 1 as both a DMA request and an interrupt event. This is because, after one clock cycle, the system automatically clears any semaphore that triggered a DMA request.

void DFB_SetSemaphores(uint8 mask)

Description: This function sets semaphores specified with a 1.

Parameters: mask: Mask specifying the bits to set

Value	Description
DFB_SEMAPHORE0	Semaphore 0
DFB_SEMAPHORE1	Semaphore 1
DFB_SEMAPHORE2	Semaphore 2

Return Value: None

Side Effects: None

void DFB_ClearSemaphores(uint8 mask)

Description: This function clears semaphores specified with a 1.

Parameters: mask: Mask specifying the bits to clear.

Value	Description
DFB_SEMAPHORE0	Semaphore 0
DFB_SEMAPHORE1	Semaphore 1
DFB_SEMAPHORE2	Semaphore 2

Return Value: None

Side Effects: None

uint8 DFB_GetSemaphores(void)

Description: This function checks the current status of the DFB semaphores and returns that value.

Parameters: None

Return Value: uint8 value between 0 and 7 where bit 0 represents semaphore 0, and so on.

Value	Description
DFB_SEMAPHORE0	Semaphore 0
DFB_SEMAPHORE1	Semaphore 1
DFB_SEMAPHORE2	Semaphore 2

Side Effects: None

void DFB_SetOutput1Source(uint8 source)

Description: This function allows you to choose which internal signals are mapped to output 1.

Parameters: source: Internal signal that is mapped to output global signal 1.

Signal	Description
DFB_RUN	DFB run bit. This is the same bit as the run bit in the DFB_CR register.
DFB_SEM0	Semaphore Bit 0.
DFB_SEM1	Semaphore Bit 1.
DFB_DFB_INTR	DFB Interrupt. This is the same signal as the primary DFB interrupt output signal.

Return Value: None

Side Effects: None

void DFB_SetOutput2Source(uint8 source)

Description: This function allows you to choose which internal signals are mapped to output 2.

Parameters: source: Internal signal that is mapped to output global signal 2.

Signal	Description
DFB_SEM2	Semaphore bit 2.
DFB_DPSIGN	Datapath sign. This signal asserts any time the output of the ALU in the datapath unit is negative. It remains high for each cycle this condition is true.
DFB_DPTHRASH	Datapath threshold crossed. This signal asserts any time the threshold of 0 is crossed in the ALU and one of the following instructions is executing: tdeca, tsuba, tsubb, taddabsa, or taddabsb. It remains high for each cycle this condition is true.
DFB_DPEQ	Datapath ALU = 0. This signal asserts high when the output of the ALU in the datapath unit equals 0 and one of the following ALU commands is executing: tdeca, tsuba, tsubb, taddabsa, or taddabsb. It remains high for each cycle this condition is true.

Return Value: None

Side Effects: None

void DFB_Sleep(void)

Description: This is the preferred routine to prepare the component for sleep. The DFB_Sleep() routine saves the current component state. Then it calls the DFB_Stop() function and calls DFB_SaveConfig() to save the hardware configuration.

Call the DFB_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. See the PSoC Creator *System Reference Guide* for more information about power management functions.

Parameters: None

Return Value: None

Side Effects: None

void DFB_Wakeup(void)

Description: This is the preferred routine to restore the component to the state when DFB_Sleep() was called. The DFB_Wakeup() function calls the DFB_RestoreConfig() function to restore the configuration. If the component was enabled before the DFB_Sleep() function was called, the DFB_Wakeup() function will also re-enable the component.

Parameters: None

Return Value: None

Side Effects: Calling the DFB_Wakeup() function without first calling the DFB_Sleep() or DFB_SaveConfig() function may produce unexpected behavior.

void DFB_Init(void)

Description: This function initializes or restores the default DFB component configuration provided with the customizer:

- Powers on the DFB (PM_ACT_CFG) and the RAM (DFB_RAM_EN)
- Moves CSA/CSB/FSM/DataA/DataB/Address calculation unit (ACU) data to the DFB RAM using an 8051/ARM core
- Changes RAM DIR to DFB
- Sets the interrupt mode
- Sets the DMA mode
- Sets the DSI outputs
- Clears all semaphore bits and pending interrupts

Parameters: None

Return Value: None

Side Effects: All registers will be reset to their initial values. This reinitializes the component. This function turns off the run bit and enables power to the DFB block.

void DFB_Enable(void)

Description: This function enables the DFB hardware block, sets the DFB run bit, and powers on the DFB block.

Parameters: None

Return Value: None

Side Effects: None

void DFB_SaveConfig(void)

Description: This function saves the component configuration and nonretention registers. It also saves the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the DFB_Sleep() function.

Parameters: None

Return Value: None

Side Effects: None

void DFB_RestoreConfig(void)

Description: This function restores the component configuration and nonretention registers. It also restores the component parameter values to what they were before calling the DFB_Sleep() function.

Parameters: None

Return Value: None

Side Effects: Calling this function without first calling the DFB_Sleep() or DFB_SaveConfig() function can produce unexpected behavior.

Defines

ClearInterruptSource(event) – Macro for clearing interrupts

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The DFB component does not have any specific deviations.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Functional Description

The Digital Filter Block is a 24-bit fixed-point, programmable limited-scope DSP with a 24*24 Multiply and Accumulate Unit (MAC), a multifunction Arithmetic Logic Unit (ALU), and data routing, shifting, holding, and rounding functions.

Other important features of the DFB are:

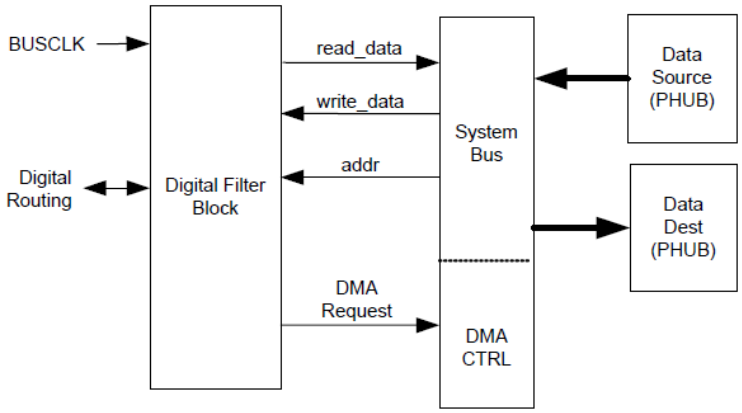
- Two 24-bit-wide streaming data channels
- Two sets of control store RAMs, each of which can store up to 64 control words
- Two sets of data RAMs, each of which can store up to 128 24-bit-wide words
- Address calculation units (ACU) to calculate the data RAM address and two ACU RAMs, each of which can store up to 16 absolute data RAM addresses
- Two sets of 32*32 Finite State Machine RAM to store the control flow (branching) information
- One interrupt and two DMA request channels
- Three semaphore bits to interact with system software
- Data alignment and coherency protection support options for input and output registers

The DFB supports up to two streaming data channels, where programming instructions, historic data and filter coefficients, and results are stored locally with new periodic data samples received from the AHB interface. In addition, the system SW can ‘load sample or coefficient data in or out of the DFB data RAMs, reprogram for different filter operation in ‘block mode’, or both. This allows for multichannel processing or deeper filters than are supported in local memory. The

block provides SW-configurable interrupt and two-DMA-channel support. There are three semaphore bits for SW to interact with the DSP assembly program.

The DFB has two 24-bit input staging registers and two 24-bit output holding registers. These registers can be accessed by both the DFB and the AHB bus (CPU/DMA). Input data is generally streamed into staging registers using CPU or DMA and output is streamed out through DFB holding registers. The two sets of input /output registers make it well suited for stereo data processing applications (two channels in parallel). These inout registers support 32-bit, 16-bit, and 8-bit accesses and have coherency protection HW allowing them to be written or read in less than 32-bit accesses.

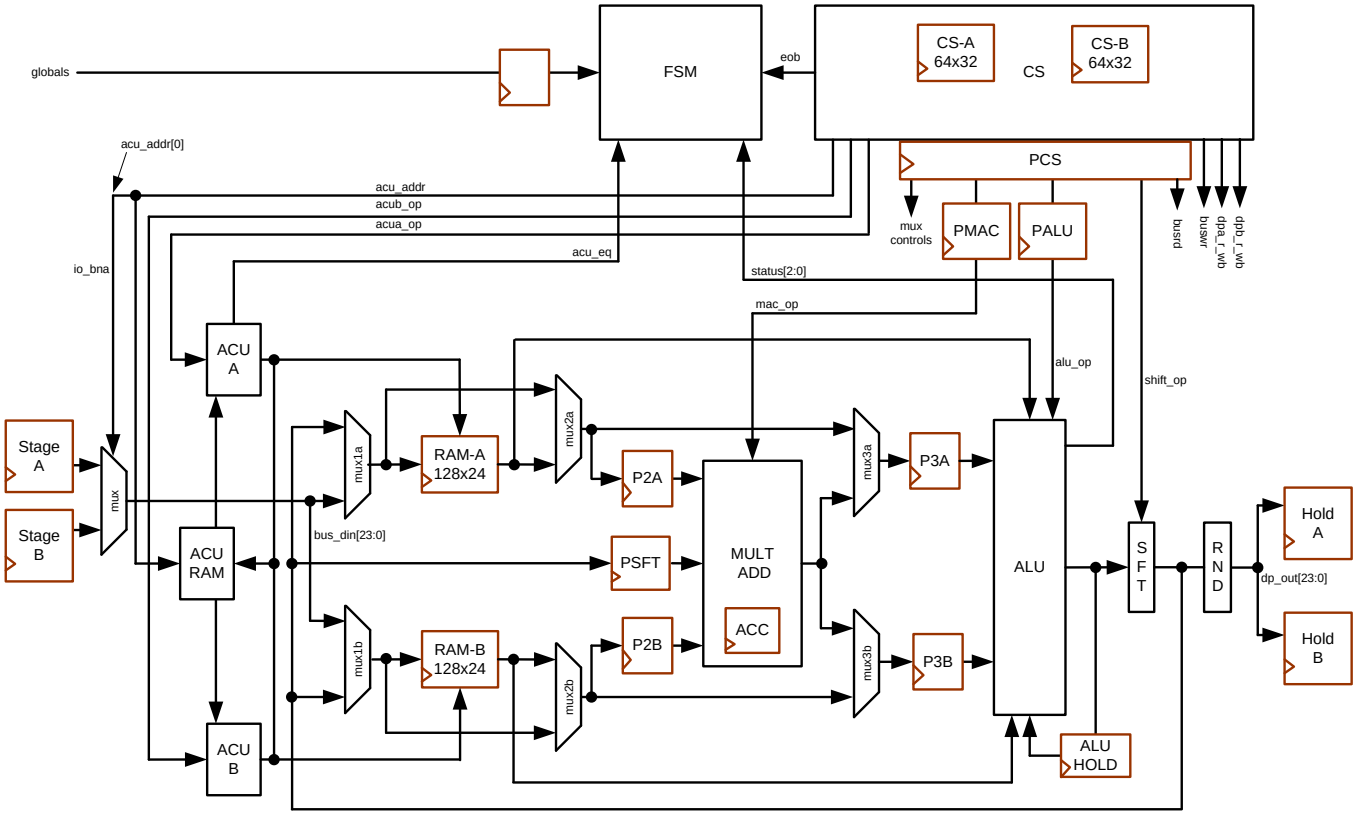
Figure 2. DFB Application Diagram



The typical use model is for data to be supplied to the DFB over the system bus from another on-chip system data source such as an ADC. The data typically passes through main memory or is directly transferred from another chip resource through DMA.

The DFB architecture supports parallel operation of the ACUs, ALU, MAC, and shifter. The operations allowed for each are encoded in bit fields. The basic assembly instruction set is given in terms of these functions. The instruction pipelining follows Figure 3 for the DFB processor. The diagram shows the locations of the pipeline registers so you can determine the instruction pipeline latency. While the ACU/DPRAM, MAC, and ALU/shifter all can operate in parallel, the instruction delay from one block to the next is one cycle of overhead. Suppose, for example, that you wanted to specify a new ACU address, multiply and accumulate based on that address, then see if the output exceeds a threshold. You would need to schedule the ACU address in the first cycle, the MAC in the second cycle, and the threshold in the third cycle after that. If the threshold flag from the data path is to be used in the controller for a branch, that branch cannot happen until the fourth cycle in the sequence. This is not usually an issue, as algorithms can be scheduled to avoid four-cycle delays in branches. Typically, an algorithm will see a one-cycle delay between the last command and the branch, because the steps leading up to the branch can be combined with previous statements in the algorithmic flow. The ACUs are positioned so there is no cycle delay between detection and branching.

Figure 3. Data Flow/Pipelining Diagram



Any of the semaphore bits can optionally be programmed to be associated with the system interrupt signal or either of the DMA_REQ outputs, leaving the DFB, either of the output DSI signals (Out_1, Out_2).

Data in DFB and particularly in DATA RAM A/B memories are represented in two's complement format. The DFB operates on 24-bit signed arithmetic values. Valid values are from 0 to 16,777,215. The DFB component is oriented on filtering algorithms and has a range from –1 to 1. Value 1 (0.9999999) equals 0x7FFFFF (8388607), 0 equals 0x000000 (0), –1 equals 0x800000 (8388608), –0.0000001 equals 0xFFFFFF (16,777,215), 0.0000001 equals 0x000001 (1). The 24th bit is the sign.

DFB Compactor

Using the optimizer feature (Optimize Assembly States tab), all 128 memory entries are available for program store in the DFB. The DFB supports code flow execution alternating from one 64-entry code store to the other, creating zero overhead looping and branches. When the code stores are not identical, all 128 memory entries are available for programming.

The compactor divides the program into states and puts the program states in one of the two control stores. The compactor also generates jump addresses between control stores. In a normal situation, jumps between program states go from one control store to another. A program

cannot require a jump within the same control store. For example, you have a routine called FILTER, and it is partitioned to be in RAM A. You also have two routines, R1 and R2, both of which jump to FILTER. If R1 and R2 are both in RAM B, there is no problem, but if either is in RAM A, then the program cannot jump. This situation causes an error at assembly. To resolve this issue, the output panel provides Compactor information such as: the content of RAM A, RAM B and a description of control finite state machine (CFSM) content. The content of RAM A and RAM B contains information about program states. The CFSM content description contains information about jumps between program states.

A code profiler was added to allow you to see where the lines of code were placed in the control store (cstore). Because the input assembly language is cycle- and line-oriented, there is a one-to-one correspondence between a line in the assembly language and an entry in the cstore. This allows you to gather meaningful information about how to optimize your code.

When a DFB program has an issue with assembly state optimization, the following error appears after assembling:

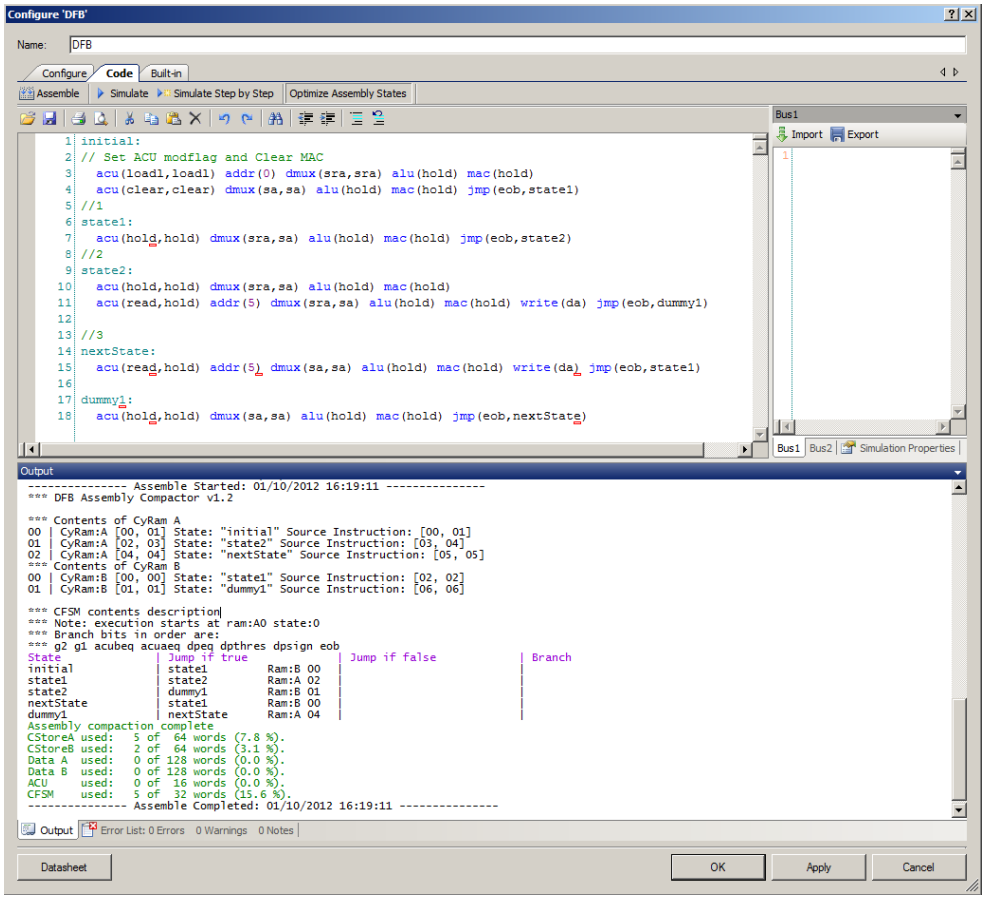
“ERROR: Unable to map to split RAMs. Found N states that can't be mapped. Please analyze results and verify code can be mapped.”

When the state has indexed as -01, the file shows that the code is not in the correct RAM. You can deduce the problem by looking at the RAM and state information.

Using the log file, you can construct the block diagram of the flow and identify the block allocation. To resolve your issue, you can place additional dummy states into your program. In the case of the problem mentioned earlier, it will cost a single dummy state and a single instruction. For example, if you have a problem with a jump to the nextState program state, you can jump to dummy state dummy1. From this state you can jump to nextState. See the following single instruction.

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

The following screenshot shows an example of this process.



Resources

The DFB component uses the dedicated DFB hardware block in the silicon.

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Default	1618 + size of DFB program*	9	1232 + size of DFB program[1]	9

DFB Data Memory (RAM)					
DATA A	DATA B	CS A	CS B	FSM	ACU
128x24	128x24	64x32	64x32	64x32[2]	16x14

The various DFB RAMS can be accessed by the DFB or the system (CPU/DMA) AHB bus, but not by both at once. In cases where bulk data must be moved into the DFB RAMs, pass the control of the DFB RAM to the system AHB bus (CPU/DMA), load the new data to the DFB RAMs, and pass control back to the DFB. The DFB_RAM_DIR register controls whether the DFB RAMs can be accessed by the DFB or the system bus (CPU/DMA).

RAM Name	Size	Functions
DATA A	128x24	Sample/Coeff Store
DATA B	128x24	Sample/Coeff Store
CS A	64x32	Control Store
CS B	64x32	Control Store
FSM	64x32	Finite state machine RAM
ACU	16x14	Address Store

1. The size of the DFB program is restricted by the DFB data memory size shown in the following table
2. For current DFB implementation, only half (32x32) of the FSM memory is available.

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ TA ≤ 85 °C and TJ ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	DFB operating current	64-tap FIR at F_{DFB}				
		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) (only for PSoC 5LP)	–	26.1	42.5	mA

AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
F_{DFB}	DFB operating frequency	PSoC 3	DC	–	67.01	MHz
		PSoC 5LP	DC	–	80.01	MHz

DFB Assembler

Directive Descriptions

AREA

The “area” directive takes an argument specifying the accessibility of a RAM segment. The selected RAM may or may not be enclosed in parenthesis:

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

There are only three RAMs, data_a, data_b, and acu, that you would logically need to access and modify using the dw (define word) command. Control and CFSM RAM are accessible, but it does not make sense to manually code them when the assembler is designed to enter those opcodes for you.

Comments

Comments for the assembler are the same as C style line comments, with the stipulation that they occupy the entire line.

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

ORG

The “org” directive sets the current location counter (CLC) for the current RAM. Initially, each RAM’s CLC is set to zero. The value of a location must be an integer and must be constrained to the set of numbers that represent a valid memory location. Both of the following are acceptable formats for this instruction.

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

dw

Define Word. After the instruction, the argument is the value that is place in the memory of the current section and CLC for that section. The CLC then increments. The customizer generates and displays an error message when the program tries to write a value that exceeds the maximum value of the RAM. Hexadecimal arguments beginning with the prefix “0x” are acceptable inputs for the “acu” area. This greatly clarifies the separation between values for side A and side B in the ACU RAM. Integer values representing $1\text{-}2^{-23}$ to –1 in 24-bit, 2’s complement format (0 to 16,777,215) are acceptable inputs for data areas. In an ACU RAM, two 7-bit sides allow input data using 4-digit hexadecimal inputs. This makes it clear what value is placed in each side of the 14-bit wide RAM, as the first two digits are entered into side A and the second two into side B. The valid value range for each side is from 0x00 to 0x7F.

Example:

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

Labels

Labels are user-defined tags to refer to blocks of cstore code. Blocks of code are sections beginning with a valid label and ending with a jump instruction, which together become a state in the CFSM. With the exception of the keywords listed as instructions in this document, a label can be defined by a set of characters that start with a letter and are followed by a colon. Do not put declarations on the same line as a label. For DFB hardware commands to be executed they must be located inside of a label/jump block; the two of these together create a state. Two labels with no jump-terminated block separating them are not permitted.

An example declaration is given.

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

VLIW Commands

Each line of commands defines opcodes in a 32-bit very long instruction word (VLIW). The most inclusive form of command for a single instruction word is shown below in the required order. Required commands in each line are emphasized using bold text. The following section outlines each individual instruction. An instruction of the proper instruction sets should be entered in between the parentheses. ACU and DMUX instructions both require two instructions separated by a comma, the first for datapath side A, the second for side B.

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

Each instruction contributes a short opcode to the VLIW, resulting in a 32-bit instruction control word for each line.

ACU

The address calculation unit (ACU) outputs the data RAM addresses required for the next instruction cycle. A single ACU is basically a counter with four registers with16 RAM locations. The ACU (including the ACU RAM) is initialized to their default values whenever a hard reset event occurs or when the RUN bit is ‘0’.

- **reg** – reg stores the current value that the ACU is operating on and outputs it on every cycle unless a command specifies otherwise. Default = 0.
- **freg** – freg can be loaded with the value that the data RAMs increment or decrement, when using the addf and subf commands. For example: load three into freg and you can increment through the data RAMs by three using ACU’s ‘addf’ instruction. Default = 2.
- **mreg** – mreg stores the maximum value before wraparound to the lreg value when modulus arithmetic is enabled. Default = 127.
- **lreg** – lreg stores the minimum value before wraparound to the mreg value when modulus arithmetic is enabled. Default = 0.

Modulus arithmetic prevents the ACU from incrementing past the value of mreg and from decrementing below the value of lreg. The ACU will produce unexpected (although deterministic) results when modulus arithmetic is enabled and the current address is located outside of the lreg to mreg range. Good DFB programming practice requires that you ensure, using the “read” command or through careful inspection, that the value the ACU points to begins in a valid location.

A 16-row-deep RAM accompanies the ACU to store values needed for storing absolute addresses of data RAM sections that are required during program execution. It also stores other values that the ACU might need access to, such as values for freg. During run time, the preferred way to place data into the ACU RAM is through system software intervention.

Use of the ACU command is defined as follows:

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

The two different instructions, instruction_A and instruction_B, represent members of the ACU instruction set that control the addresses for two data RAMs independently. The following table shows a comprehensive list of the ACU instruction set.

Instruction	Description
hold	Puts the registered output address value on the output unchanged.
incr	Increases the registered output address value (reg) by one and puts it on the output.
decr	Decreases the registered output address value (reg) by one and puts it on the output.
read	Gets the specified byte from the ACU RAM, loads it to the output address register, and puts the value on the output. (See the addr command.)
write	Puts the registered output address value into the specified ACU RAM row. (See the addr command.)
loadf	Loads freg with the value from the specified ACU RAM. The ACU output value remains the same as the previous cycle. (See the addr command for information about ACU RAM addressing.)
loadl	Loads lreg with the value from the specified ACU RAM. The ACU output value remains the same as the previous cycle. (See the addr command for information about ACU RAM addressing.)
loadm	Loads mreg with value of the specified ACU RAM. The ACU output value remains the same as the previous cycle. (See the addr command for information about ACU RAM addressing.)
writel	Puts the value of lreg on the output and then writes the output to the specified ACU RAM location. (See the addr command.)
setmod	Enables modulus arithmetic in the ACU. Modulus arithmetic is on by default.
unsetmod	Disables modulus arithmetic in the ACU. Modulus arithmetic is on by default.
clear	Sets the registered output value (reg) to zero.
addf	Increases the registered output address value (reg) by the value in register freg. When modulus arithmetic is disabled and the ACU output is outside the region defined between lreg and mreg, do not use this command.
subf	Decreases the registered output address value (reg) by the value in register freg. When modulus arithmetic is disabled and the ACU output is outside the region defined between lreg and mreg, do not use this command.
writem	Puts the value of mreg on the output and writes the output to the specified ACU RAM location. (See the addr command.)
writef	Puts the value of freg on the output and writes the output to the specified ACU RAM location. (See the addr command.)

addr

The addr command takes a value between zero and fifteen as an argument. It is used in several different ways. Be careful to write programs in such a way that commands within the same

instruction do not require more than one access to the addr value. Multiple access of addr does not generate an error, but a warning, if you do not redefine the addr value.

The addr command can act in five different ways. You cannot choose more than one at a time.

- To access a location in ACU RAM. You can specify a single ACU RAM location to be accessed by both the ACUs. (Both sides must access the same row of ACU RAM. Side A cannot read from row 1 while B reads from row 14.)
- To specify which input and output channels to write to or read from. Channel 1 is selected for odd addr values (when addr's LSB is 1) and Channel 2 is selected for even addr values. See the dmux section for more information about input channels and input staging registers, and the
- write section for information on output staging registers.
- To provide a value to write semaphores and enable and disable semaphores as jump conditions. The addr value is automatically set by the 3-bit fields specified in the semaphore commands. (See the
- ALU section.)
- Enabling and disabling the saturation and rounding flags and clearing the saturation detection flag hides the explicit definition of the addr value from you and defines it automatically. (See the
- ALU section.)
- Enabling and disabling the global interrupts as jump conditions defines the addr value using the 2-bit field in the englobal command. (See the
- ALU section.). If the addr command is called as addr(1), both the side A and side B ACU will access ACU RAM row 1.

If the command causes a bus read, bus data is read from staging register 1. If the command causes a bus write, data is written to output staging register 1.

This is an example of the problem with having multiple accesses to the value of the addr command. When the addr value is defined once, all access to it must use the same value, or the customizer will return an error. The explicit definition of addr prevents the use of the semaphore, global jump condition enables, and saturation and rounding register commands.

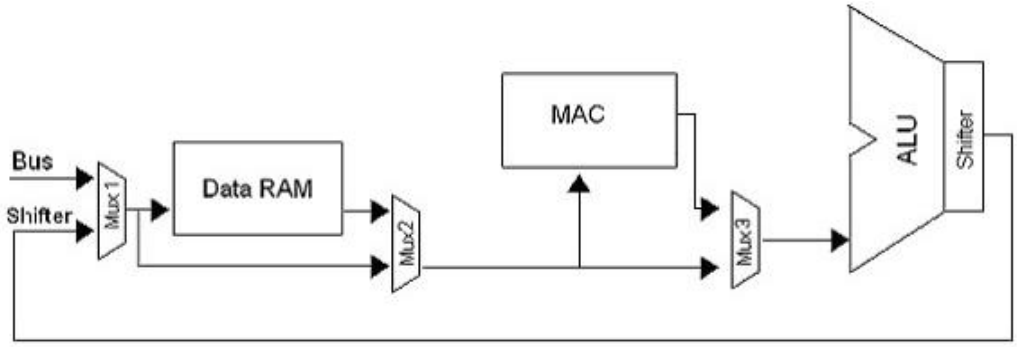
dmux

Six bits in the control store RAM (cstore) output word control datapath muxing and specify the route data takes through the datapath. The MAC, ALU, and output shifter have three levels of muxing for both side A and side B. Similar to the ACU, dmux takes an argument for side A and side B.

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

When mux1 (see Figure 4) is set to allow access to the bus, it consumes whatever value is waiting in the input register. Unless the system is completely deterministic, code should wait for the controller to signal the availability of a value using a wait state that loops until the input channel ready jump condition is valid.

Figure 4. Datapath



The following table shows a comprehensive list of the dmux instruction set.

Instruction	Description
ba	Bus to ALU. mux1 passes the bus data to the data RAM. mux2 circumvents the data RAM and passes mux1 data directly to the MAC and mux3. mux3 circumvents the MAC and passes mux2 directly to the ALU input. The bus data comes from one of the two staging registers based on an addr declaration of either 0 or 1.
sa	Shifter to ALU. mux1 passes the shifter output data to the data RAM. mux2 circumvents the data RAM and passes mux1 data directly to the MAC and mux3. mux3 circumvents the MAC and passes mux2 directly to the ALU input.
bra	Bus to RAM, RAM to ALU. mux1 passes the bus data to the data RAM. mux2 passes data from the data RAM to the MAC and mux3. mux3 circumvents the MAC and passes mux2 directly to the ALU input. T The bus data comes from one of the two staging registers based on an addr declaration of either 0 or 1.
sra	Shifter to RAM, RAM to ALU. mux1 passes the shifter output data to the data RAM. mux2 passes data from the data RAM to the MAC and mux3. mux3 circumvents the MAC and passes mux2 directly to the ALU input.
bm	Bus to MAC, MAC to ALU. mux1 passes the bus data to the data RAM. mux2 circumvents the data RAM and passes mux1 data directly to the MAC and mux3. mux3 passes the MAC output to the ALU input. The bus data comes from one of the two staging registers based on an addr declaration of either 0 or 1.
sm	Shifter to MAC, MAC to ALU. mux1 passes the shifter output data to the data RAM. mux2 circumvents the data RAM and passes mux1 data directly to the MAC and mux3. mux3 passes the MAC output to the ALU input.

Instruction	Description
brm	Bus to RAM, to MAC. mux1 passes the bus data to the data RAM. mux2 passes the data from the data RAM to the MAC and mux3. mux3 passes the MAC output to the ALU input. The bus data comes from one of the two staging registers. Register choice is based on an addr declaration of either 0 or 1.
srm	Shifter to RAM to MAC. mux1 passes the shifter output data to the data RAM. mux2 passes the data from the data RAM to the MAC and mux3. mux3 passes the MAC output to the ALU input.

ALU

The ALU provides data control on the output end of the datapath. In addition to generic functions such as add and subtract, the ALU can set flags signaling that specific conditions for jumps between states have been met. The alu command includes five special instructions that require a bit field of three bits as input data. The ALU's output feeds directly into the shift register.

There are two format options for an "alu" command:

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

The following table shows a comprehensive list of the alu instruction set.

Instruction	Description
set0	Sets the ALU output to zero.
set1	Sets the ALU output to an integer value of one. That is, the LSB is one, all others are zero.
seta	Passes input A to the output.
setb	Passes input B to the output.
nega	Negates A and passes it to the output.
negb	Negates B and passes it to the output.
passrama	Passes RAM A's current location value to the output.
passramb	Passes RAM B's current location value to the output.
add	Evaluates and places 'A + B' on the ALU output.
tdeca	Evaluates and places 'A – 1' on the ALU output. A value of zero sets threshold detection. The design includes this command as a way to wait a set amount of time while a value counts down to zero. It is intended for use in a low-power wait mode.
suba	Evaluates and places 'B – A' on the ALU output.
subb	Evaluates and places 'A – B' on the ALU output.
absa	Evaluates and places '|A|' on the ALU output.
absb	Evaluates and places '|B|' on the ALU output.
addabsa	Evaluates and places '|A| + B' on the ALU output.

Instruction	Description
addabsb	Evaluates and places 'A + |B|' on the ALU output.
hold	Maintains the ALU output value from the previous cycle.

englobals	Enables or disables the two global interrupts and the saturation detection flag as jump conditions for state changes. This is based on the 3-bit field following the command and a separating comma. The global interrupts bit [1:0] are inputs to the DFB. The saturation detection flag (enabled by Saturation detection enable - bit [2]) is set when a wraparound would otherwise happen if the saturation logic holds the value at the maximum or minimum datapath value.
	This command sets the value for addr automatically, based on the entry to the bit field. englobals shares an ALU opcode with ensatrnd. The behavior of these commands is determined in hardware by the value of the addr opcode. The customizer generates errors if the program tries to define the addr command with different values.
	`alu(englobals, [Saturation detection enable, Global interrupt 2 enable, Global interrupt 1 enable] )`
	
	For example, the following would set global interrupt 2 and saturation to be disabled as a jump condition and global interrupt 1 to be enabled.
	`alu(englobals, 001)`

ensatrnd	Enables and disables saturation and rounding in the datapath by writing to the saturation and rounding registers. The value written to the saturation and rounding registers is taken from bit [1:0] of the 3-bit field following both the command and a separating comma. The bit [2] of the 3-bit field strobes the saturation detection flag and clears it. This command uses and sets the value for addr automatically. Additionally this command shares an opcode with englobals; the behaviors of these two commands are determined in hardware by the value of the addr opcode. The customizer generates errors if the program tries to define the addr command with different values.
	`alu(ensatrnd, [Clear saturation detection flag, Saturation detection enable, Rounding enable] )`
	
	For example, the following ALU command turns on rounding, turns off saturation and clears the saturation detection flag if it is set.
	`alu(ensatrnd, 001)`

ensem	Enables specified semaphores as jump conditions based on the 3-bit field following the command and a separating comma. This command uses and sets the value for addr automatically, based on the 3-bit field. Therefore, the customizer generates errors if the program tries to define the addr command with different values using this instruction. This instruction has a two cycle delay before the change becomes a valid jump condition. While using the semaphore as a jump condition, the condition "sem" can be used as a reminder to the programmer that a semaphore condition has been set.
	`alu(ensem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	
	If the program no longer wants the semaphore as a jump condition, it must clear the enable flags with a call of the command with every bit in the field set to zero.
	`alu(ensem, 000)`

Instruction	Description

setsem	Sets the semaphores masked with ones in the 3-bit field to 1. Do not use setsem in the first instruction after reset or the semaphore will be repeatedly set. This command uses and sets the value for addr automatically. The customizer generates errors if the program tries to define the addr command with different values.
	`alu(setsem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	
	The following example sets semaphore 2 to true (1) and leaves semaphores 1 and 0 as they are.
	`alu(setsem, 100)`

clearsem	Clears the semaphores masked with ones in the 3-bit field. (that is, sets them to zero). This command uses and sets the value for addr automatically. The customizer generates errors if the program tries to define the addr command with different values.
	`alu(clearsem, [Semaphore 2, Semaphore 1, Semaphore 0] )`
	
	The following sets the value of semaphore 2 to false (0) and leaves semaphores 1 and 0 as they are.
	`alu(clearsem, 100)`

tsuba	Evaluates and places ‘B – A’ on the output. Sets threshold detection.
tsubb	Evaluates and places ‘A – B’ on the output. Sets threshold detection.
taddabsa	Evaluates and places ‘|A| + B’ on the output. Sets threshold detection.
taddabsb	Evaluates and places ‘A + |B|’ on the output. Sets threshold detection.
sqlcmp	Loads a value from mux3 of side A into the compare register to be used as a cutoff in squelch functions.
sqlcnt	Loads the lower 16 bits of mux3 on side A into the 16-bit count register. This value is decremented every time a squelch command is called, if the current value at the output does not meet the threshold set by the squelch compare register. It is reset to its original value every time the threshold is met.
sqa	Takes the value on mux3 of side A and compares it to the value in the squelch compare register. If the current value is greater than the value of the compare register, the current value is passed to the output and the squelch count register is reset to its original value. If the current value is less than the compare register value, the command checks the squelch counter. If it is not at zero, the value is decremented and the current value on mux3 of side A is passed to the output. If the value of the count register is zero, zero is passed to the output.
sqb	Takes the value on mux3 of side B and compares it to the value in the squelch compare register. If the current value is greater than the value of the compare register, the current value is passed to the output and the squelch count register is reset to its original value. If the current value is less than the compare register value, the command checks the squelch counter. If it is not at zero, the value is decremented and the current value on mux3 of side B is passed to the output. If the value of the count register is zero, zero is passed to the output.

MAC

Multiply and Accumulate unit. Contains the hardware to multiply two fixed-point numbers and then add them to a previous value. '(A × B) + C'

There are four members in the MAC instruction set and they operate as follows:

```
mac(instruction)
```

The following table shows a comprehensive list of the MAC instruction set.

Instruction	Description
loadalu	Adds the previous ALU output (from the shifter) to the product and starts a new accumulation.
clra	Clears the accumulator and stores the current product.
hold	Holds the value in the accumulator from the previous cycle. No multiply.
macc	Multiply and Accumulate. Multiplies the values on mux2 of side A and side B. Adds the product to the current value of the accumulator.

shift

The shift command allows you to scale the ALU output. A valid shift command takes two arguments, direction and magnitude, and evaluates them to produce the correct opcode. Valid directions instructions are 'right', 'left', 'r', and 'l'. A shift to the right allows magnitudes of 1, 2, 3, 4, and 8, while a shift to the left allows only the values 1 and 2. The output of the ALU is passed through the shifter and back out to the start of the datapath regardless of whether a shift occurs.

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

The following table shows a comprehensive list of the shift instruction set.

Instruction	Description
right, r	Equivalent instructions specifying the direction to shift.
left, l	Equivalent instructions specifying the direction to shift.

write

A valid write command has zero to three arguments. A value is written for each argument. You can choose to write the value on mux1 of either side to the data RAM of the same side, or to write the current shifter output value to a staging register on the output. Check for the various pipeline delays in the DFB datapath before using the write instruction.

There are system level implications if the first instruction following a reset contains a bus write command. The assembler prevents writes in the first instruction to prevent unintended problems.

Each of the following is a valid write command:

- write(da, db, bus)
- write(da, db)
- write(db)

The following table shows a comprehensive list of the write instruction set.

Instruction	Description
da	Writes the mux1A value to the specified data_a Ram location after the ACU instruction for the line, No delay in write execution.
db	Writes the mux1B value to the specified data_b Ram location after the ACU instruction for the line, No delay in write execution.
abus	Write shifter output to the bus' holding register A. Holding register A is selected based on an addr definition of 1 (see the addr section).
bbus	Write shifter output to the bus' holding register B. Holding register B is selected based on an addr definition of 0 (see the addr section).
bus	Write shifter output to the bus' holding register. There are two output holding registers available. The register is selected based on an addr definition of either 1 or 0 (see the addr section). This instruction has been left for compatibility; do not use the bus instruction in new projects. Using the instruction generates a warning: "Potential addr() conflict attempting write(bus). Avoid this warning by using channel-specific bus write commands."

Jump Instructions

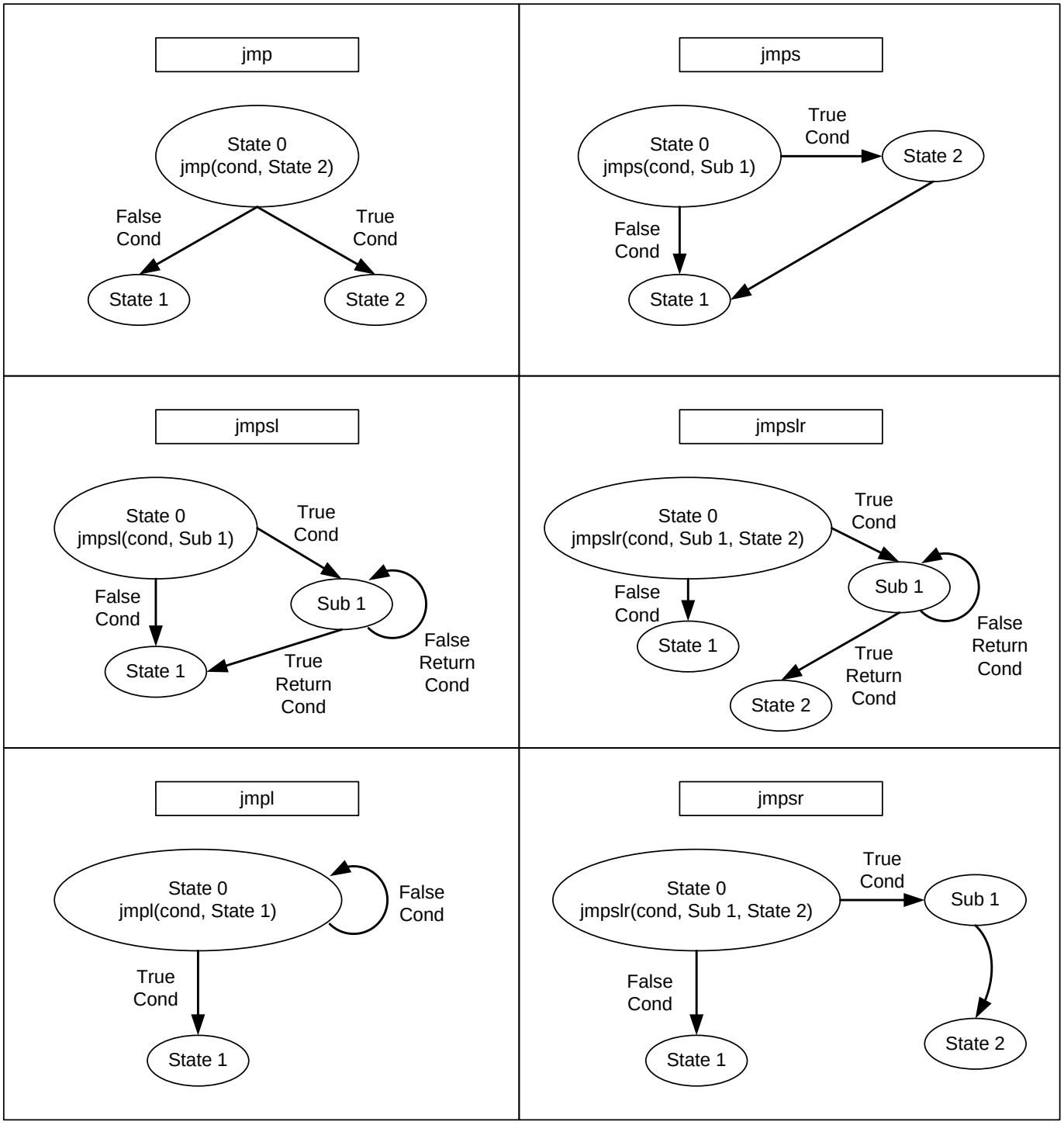
A jump instruction allows the code to change its location to a different subroutine. Jump instructions have a general form of:

```
JumpType(conditions, Target Routine)
```

where "conditions" is an ordered list of any of the enables available to restrict or allow jumps. The literals associated with these flags are described immediately following the description of the jump options. Be careful with jump instruction placement because some conditions require two cycles instead of one before the controller can use them as a valid jump condition. Additionally, a jump instruction cannot be used in the first instruction after a reset because the first state must be two instructions deep to set up pipelining.

Figure 5 contains a diagram of jump instructions.

Figure 5. Jump Instruction Diagrams



The following table shows a comprehensive list of the jump instruction set.

Instruction	Description
jmp	Jump" is similar to a standard "goto" instruction. If conditions are true, the code jumps to the target routine. If not, it falls through to next state numerically. When not in a loop, the fjlim (false jump limit) value is set to the max cstore location. jmp(eob, sign,…, Target State)
jmpl	"Jump Loop" sets the current block of code to be a loop. It is a two-way branch with the target routine as one branch when conditions are met, and the start of the current code block as the other branch when conditions are not met. When in a loop the following occurs: ▪ Bit 23 of the CFSM is set high. ▪ False jump address (FJADDR) is defined as the start of the current code block, which is the control store address of the label. (The label is not stored in the control store; the label references the first command of the block of code.) ▪ False jump limit (FJLIM) is set to the current CLC location, which is the end of the code block ▪ (Same for all jumps) The target routine's label is used to provide the jump address (JADDR) and the CFSM RAM location of the next state on true conditions. Commands execute until the eob is detected. The program evaluates the condition and if it is false, set the program counter to FJADDR, restarting the block. If the condition is true, the program counter is set to JADDR and the state is updated to NextStateOnTrue. The format is identical to the jmp command. jmpl(conditions, Target State)
jmpsl	"Jump to Subroutine Loop" allows a jump to a subroutine block of code that is designated to be a loop. The effect jmpsl has on the current block of code is identical to the jmp command. If conditions are met, the code jumps to the specified subroutine, otherwise it falls through to the next state in code execution. However, jmpsl affects the subroutine's state. Each time a subroutine is referenced, a copy of that subroutine is created as a state for the CFSM and properties for the new state are set. In the case of jmpsl, the created state is designated to be a loop and to have a return state of the next state in the code space. (The current state is the state that ends with the jmpsl command.) For more information about states and subroutines, see the jmpret entry in this table. jmpsl(conditions, Target Subroutine)
jmpslr	"Jump to Subroutine Loop with Return state" is identical to jmpsl in every way except that the return state is specified instead of defaulting to the state following the current state. jmpslr(conditions,…, Target Subroutine, Return State)
jmps and jmpsr	These two are clones of jmpsl and jmpslr except that the created subroutine state is not a loop. Because of this, only the eob condition is specified in the jmpret statement.

Instruction	Description
jmpret	Subroutines are accessed only through one of the jump-to-subroutine commands. Subroutines differ from standard states because that their properties are determined by the state that calls them, instead of the jump condition at the end of the state. If a state calls a subroutine with the jump command “jmpslr(eob, sub1, anotherState), it defines the subroutine to be a loop and have a next state of “anotherState.” Subroutines are terminated with a jump instruction of the type jmpret (jump return), which provides the subroutine with its own set of jump conditions for loop termination (if the subroutine is called as a loop). Subroutines cannot call other subroutines because the jmpret command does not provide the necessary exit information to the sub-subroutine state. The jmpret command is used as a jmp command, but with no target state specified. jmpret(condition,condition,..)

Jump Conditions

Conditions enable or prevent state changes in code. The following conditions are, in essence, enable flags for hardware. When a condition is listed, it requires the signal complementing the enable to be true, in order for the jump to proceed.

Note Datapath conditions have a two-cycle delay. That is, they must be true two cycles before the jump to recognize the condition as true.

The following table shows a comprehensive list of jump conditions.

Instruction	Description
eob	End Of Block. A condition for a jump, which is always met because a jump instruction signifies the end of the block. eob only needs to be specified when an unconditional jump occurs. This is because of a software restriction, not a hardware restriction.
dpsign	A jump based on the MSB of the ALU output. Asserted when the ALU output is negative. Datapath conditions require a two-cycle delay to meet a jump condition.
dpthresh	Datapath Threshold. Asserted when the ALU detects a sign change. The ALU asserts dpthresh only when the program uses ALU threshold detection operands (tsuba, tsubb, taddabsa, …). Datapath conditions require a two-cycle delay to meet a jump condition.
dpeq	Datapath Equity. Asserted when the ALU hardware detects an output value of zero. The ALU asserts dpeq only when the program uses ALU threshold detection operands (tsuba, tsubb, taddabsa, …). Datapath conditions require a two-cycle delay to meet a jump condition.
acuaeq	ACU A Equals. Asserted when ACU A detects a wraparound condition. This can be either zero or the maximum data RAM location if modulus arithmetic is disabled or the max and min modulo counter limit when it is enabled. Datapath conditions require a one-cycle delay to meet a jump condition.
acubeq	ACU B Equals. Asserted when ACU B detects a wraparound condition. This can be either zero or the modulo counter limit. Datapath conditions require a one-cycle delay to meet a jump condition.
in1	Channel 1 Input Register Value Ready signal. When it is asserted, a new input cycle is available for consumption. It remains asserted until cleared by a bus read. Datapath conditions require a one-cycle delay to meet a jump condition.

Instruction	Description
in2	Channel 2 Input Register Value Ready signal. When it is asserted, a new input cycle is available for consumption. It remains asserted until cleared by a bus read. Datapath conditions require a one-cycle delay to meet a jump condition.
sem	The sem condition has no effect on the opcodes. It improves code clarity and reminds the programmer that a semaphore is currently a jump condition. (See the ALU section for details.)
globals	The globals condition has no effect on the opcodes. It improves code clarity reminds the programmer that a global input is currently a jump condition. (See the ALU section for details.)
sat	The “sat” condition has no effect on the opcodes. It improves code clarity and reminds the programmer that a saturation event is currently required to enable a jump. (See the ALU section for details.)

Commands

DMUX Commands

Code	Name	Function Mux 1	Function Mux 2	Function Mux 3
0	ba	Bus register	Bus register	Bus register
1	sa	Previous shifter output	Previous shifter output	Previous shifter output
2	bra	Bus register	Current RAM value	Current RAM value
3	sra	Previous shifter output	Current RAM value	Current RAM value
4	bm	Bus register	Bus register	MAC accumulator
5	sm	Previous shifter output	Previous shifter output	MAC accumulator
6	brm	Bus register	Current RAM value	MAC accumulator
7	srm	Previous shifter output	Current RAM value	MAC accumulator

MAC Commands

Code	Name	Function
0	loadalu	Adds the ALU value to the product and starts a new accumulation.
1	clra	Clears the accumulator. Loads it with the current product.
2	hold	Holds the accumulator, no multiply (no power in mult).
3	macc	Standard operation – multiply and accumulate with the previous values

ACU Commands

Code	Name	Function
0	hold	Puts reg on the output.
1	incr	Puts reg + 1 on the output, writes to reg.
2	decr	Puts reg – 1 on the output, writes to reg.
3	read	Loads reg from ACU RAM, puts the value on the output.
4	write	Puts reg into the specified ACU RAM row.
5	loadf	Loads freg from ACU RAM, puts reg on the output.
6	loadl	Loads lreg from ACU RAM, puts reg on the output.
7	loadm	Loads mreg from ACU RAM, puts reg on the output
8	writel	Puts lreg on the output, writes to ACU RAM.
9	setmod	Sets arithmetic to modulo mreg.
10	unsetmod	Sets arithmetic to wraparound.
11	clear	Sets reg to 0, put 0 on the output.
12	addf	Adds reg to freg, puts the result on the output, stores it in reg.
13	subf	Subtracts freg from reg, puts the result on the output, stores it in reg.
14	writem	Puts mreg on the output, writes to the ACU RAM.
15	writef	Puts freg on the output, writes to the ACU RAM.

ALU Commands

Code	Name	Function
0	set0	Sets the ALU output to 0.
1	set1	Sets the ALU output to 1.
2	seta	Passes A to the ALU output.
3	setb	Passes B to the ALU output.
4	nega	Sets the ALU output to –A.
5	negb	Sets the ALU output to –B.
6	passrama	Passes the RAM A output directly to the ALU output.
7	passramb	Passes the RAM B output directly to the ALU output.
8	add	Adds A and B and puts the result on the ALU output.
9	tdeca	Puts A – 1 on the ALU output, sets threshold detection.

Code	Name	Function
10	suba	Puts B – A on the ALU output.
11	subb	Puts A – B on the ALU output.
12	absa	Puts |A| on the ALU output.
13	absb	Puts |B| on the ALU output.
14	addabsa	Puts |A| + B on the ALU output.
15	addabsb	Puts A + |B| on the ALU output.
16	hold	Holds the ALU output from the previous cycle.
17	englobals	Enables global and saturation jump conditions using a three-bit field to specify which events are active jump conditions.
17	ensatrnd	Writes to the saturation and rounding enable registers using a three-bit field to enable and disable them.
18	ensem	Enables semaphores as jump conditions using a three-bit field to specify which are active.
19	setsem	Sets the semaphores high using the three-bit mask.
20	clearsem	Sets the semaphores low using mask, addr[2:0].
21	tsuba	Puts B – A on the ALU output, sets threshold detection.
22	tsubb	Puts A – B on the ALU output, sets threshold detection.
23	taddabsa	Put |A| + B on the ALU output, set threshold detection
24	taddabsb	Puts A + |B| on the ALU output, sets threshold detection
25	sqlcmp	Loads the squelch comparison register with a value from side A, passes side B.
26	sqlcnt	Loads the squelch count register with a value from side A, passes side B.
27	sqa	Squelch side A. If the value is above the threshold, passes it. If the value is below the threshold and the squelch count register is zero, passes zero.
28	sqb	Squelch side B. If the value is above the threshold, pass it. If the value is below the threshold and the squelch count register is zero, passes zero.
29-31	undefined	Undefined opcodes

Component Errata

This section lists known problems with the component.

Cypress ID	Component Version	Problem	Workaround
209657	All	When performing CPU writes to the Staging registers of the DFB with code that is highly optimized by the compiler, the writes may occur too quickly for the DFB hardware to process correctly. The problem may occur if you are using the DFB_LoadInputValue() API or when you directly write to the three Staging registers of a channel in succession. This problem occurs only for CPU writes to the Staging registers. The problem does not occur for DMA transfers to the Staging registers.	If you use DFB_LoadInputValue() API, then it is recommended to add the keyword “volatile” to the “uint32 value” variable in the function. If you are making direct register writes then you may add a finite delay in between writes to the low, medium and high Staging registers. For example, for GCC, add asm(“nop”) between each register write.

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.40.b	The Configure dialog was updated to fix a scrolling problem in the Code tab.	The scrolling region was very large and caused the Bus and Output panels to be located outside of the viewable area. The DFB component was patched to address this problem. There is no impact on current designs.
1.40.a	Added an Errata section	CPU writes to the Staging registers of the DFB hardware require an extra clock cycle delay for each register write. This requirement may be ignored with high compiler optimization.
1.40	Updated SaveConfig() and RestoreConfig() APIs	Momentary glitch observed on the output for PSoC 5LP upon wakeup.
	Added default values for ACU RAM	Clarified information
1.30	Expanded descriptions of DFB_SetCoherency() and DFB_SetDalign() APIs to show that they are direct register writes.	Vague description of the API usage.
	Modified bit order information and usage for ALU instructions - englobals, ensatrnd, setsem, clearsem	Incorrect bit order for the instructions and inference was required from vague examples.
	Moved Simulator Output section from Functional Description to Code Tab	Fluidity of information needed.

Version	Description of Changes	Reason for Changes / Impact
1.20	Added Simulator Output information to the datasheet	
	Changes to meet MISRA compliance	MISRA compliant with some global deviations
	Updated DFB LoadDataRAMx() API source	To improve the speed of data transfer
1.10	Updated DC and AC Electrical Characteristics section.	
	Added PSoC5LP device support.	

© Cypress Semiconductor Corporation, 2015-2016. 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.