Radar Hardware Accelerator 2.0 User Guide – Part 1

Version 0.97: 1 September 2020

Note: This is a preliminary draft version of the HWA2.0 user guide. This document is subject to changes.

Radar Hardware Accelerator 2.0 - Part 1

The *Radar Hardware Accelerator 2.0 User's Guide* (in two parts) describes the Radar Hardware Accelerator architecture, features, and operation of various blocks and their register descriptions. The purpose is to enable the user to understand the capabilities offered by the Radar Hardware Accelerator and to program it appropriately to achieve the desired functionality.

This user's guide is divided into two parts. The first part (this document) provides an overview of the overall architecture and features available in the Radar Hardware Accelerator. The main features, such as, windowing, FFT, and log-magnitude are covered in this part.

The second part of the user's guide covers additional features like CFAR and other advanced usage possibilities. The second part of the user's guide is optional and can be skipped if the user is interested only in the FFT computation capability.

This user's guide is organized as follows: Section 1 covers the introduction and high-level architecture. Section 2, Section 3, and Section 4 describe the state machine, trigger mechanisms, input/output formatting, and general framework for using the accelerator. Section 5 describes the primary computational unit features, namely, windowing, FFT, and log-magnitude.

Contents

Figures

Tables

1 Radar Hardware Accelerator – Overview

This section provides an overview of the Radar Hardware Accelerator 2.0. The section covers the key features of the accelerator and overall architecture.

1.1 Introduction

The Radar Hardware Accelerator 2.0 is a hardware IP that enables off-loading the burden of certain frequently used computations in FMCW radar signal processing from the main processor. It is well known that FMCW radar signal processing involves the use of FFT and log-magnitude computations to obtain a radar image across the range, velocity, and angle dimensions. Some of the frequently used functions in FMCW radar signal processing can be done within the Radar Hardware Accelerator, while still retaining the flexibility of implementing other proprietary algorithms in the main processor.

1.2 Key Features

The main features of the Radar Hardware Accelerator 2.0 are as follows.

- Fast FFT computation, with programmable FFT sizes (2, 4, 8…, 2048-pt, and 3, 6, 12, …, 1536-pt) complex FFT
- Internal FFT bit width of 24 bits (for each I and Q) for good SQNR performance, with fully programmable butterfly scaling at every stage for user flexibility
- Built-in capabilities for pre-FFT processing – specifically DC estimation and removal, interference localization and mitigation, channel equalization, channel combination, zero insertion, and programmable windowing
- Magnitude (absolute value) and log-magnitude computation capability
- Flexible data flow and data sample arrangement to support efficient multidimensional FFT operations and transpose accesses as required
- Chaining, looping and context switching mechanisms to sequence a set of accelerator operations one-after-another with minimal intervention from the main processor
- CFAR-CA and CFAR-OS detector support (linear and logarithmic), Local Maxima engine
- Statistics including 2D maxima, Histogram and CDF
- Radar data compression / decompression capability
- Miscellaneous other capabilities of the accelerator:
  - Stitching two or four 2K-point FFTs to get the equivalent of 4096-point or 8192-point FFT for industrial level sensing applications where large FFT sizes are required
  - Slow DFT mode, with resolution equivalent to 16K size FFT, for FFT peak interpolation purposes (for example, range interpolation)
  - Complex vector multiplication and Dot product capability for vectors up to 1024 in size

This user's guide is divided into two parts. The first part covers the high-level architecture and key features such as windowing, FFT, and log-magnitude. The (optional) second part covers additional features such as CFAR, complex multiplication, advanced statistics, radar data compression engine and so forth.

1.3 High Level Architecture

The Radar Hardware Accelerator module is loosely coupled to the main processor (eg. C6x DSP). The accelerator is connected to a 128-bit bus that is present in the main processor system, as shown in Figure 1.

The Radar Hardware Accelerator module comprises an accelerator engine and eight memories, each of 16KB size, which are used to send input data to and pull output data from the accelerator engine. These memories are referred to as *local memories* of the Radar Accelerator (ACCEL_MEM). For convenience, these local memories are referred to as ACCEL_MEM0, ACCEL_MEM1, … and ACCEL_MEM7.

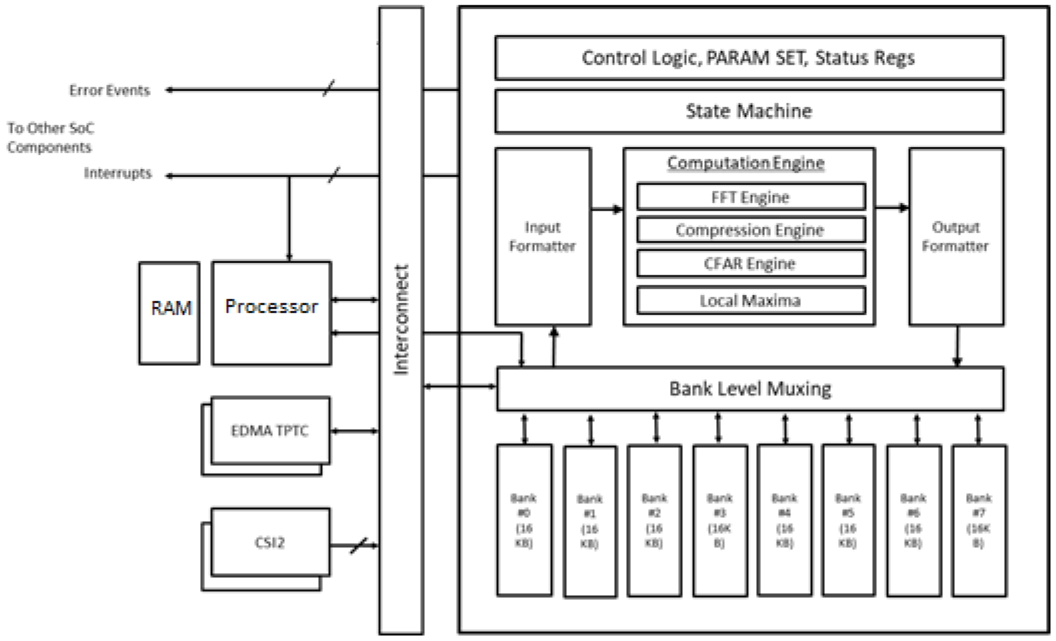


Figure 1: Radar Hardware Accelerator (TPR12 device)

1.3.1 High-Level Data Flow

The typical data flow is that the DMA module is used to bring samples (for example, FFT input samples) into the local memories of the Radar Hardware Accelerator, so that the main accelerator engine can access and process these samples. Once the accelerator processing is done, the DMA module reads the output samples from the local memories of the Radar Hardware Accelerator and stores them back in the Radar data memory for further processing by the main processor.

The purpose behind the eight separate local memories (16KB each) inside the Radar Hardware Accelerator is to enable the *ping-pong* mechanism, for both the input and output, such that the DMA write (and read) operations can happen in parallel to the main computational processing of the accelerator. The presence of multiple memories enables such parallelism. For example, the DMA can be configured to write FFT input samples (ping) into ACCEL_MEM0 and read FFT output samples (ping) from ACCEL_MEM2. At the same time, the accelerator engine can be working on FFT input samples (pong) from ACCEL_MEM1 and writing FFT output samples (pong) into ACCEL_MEM3. However, both the DMA and the accelerator cannot access the same 16KB memory at the same time. This would lead to an error (refer to the MEM_ACCESS_ERR_STATUS register description in Table 3). As will be explained in later sections, the accelerator engine can perform multiple computational steps one after another autonomously. In each step, the input samples are read from one of the eight local memories and the output samples are written into another one of the eight local memories.

The Radar Hardware Accelerator operates on a single clock domain and the operating clock frequency is 400 MHz (in TPR12 device).

The accelerator local memories are 128-bits wide, for example, each of the 16KB banks is implemented as 1024 words of 128 bits each. This allows the DMA to bring data into the accelerator local memories efficiently (up to a maximum throughput of 128 bits per interconnect clock cycle, depending upon the DMA configuration). Two ports for accessing the HWA local memories are available and these map the same 128KB into two different address spaces [REF : TRM]

It is important to note that any of the eight local memories can be the *source* of the input samples to the accelerator engine and any of the eight local memories can be the *destination* for the output samples from the accelerator engine – with the important restriction that the source and destination memories cannot be the same 16KB bank. Note also that the accelerator local memories do not necessarily need to be used in ping-pong mode and can instead be used as larger 32KB input and output memories, if the use case requires. The address space for the 16KB memories is contiguous (including a wrap-around form the end of ACCEL_MEM7 to the start of ACCEL_MEM0). Therefore the source as well as destination memory addresses can span beyond 16KB.

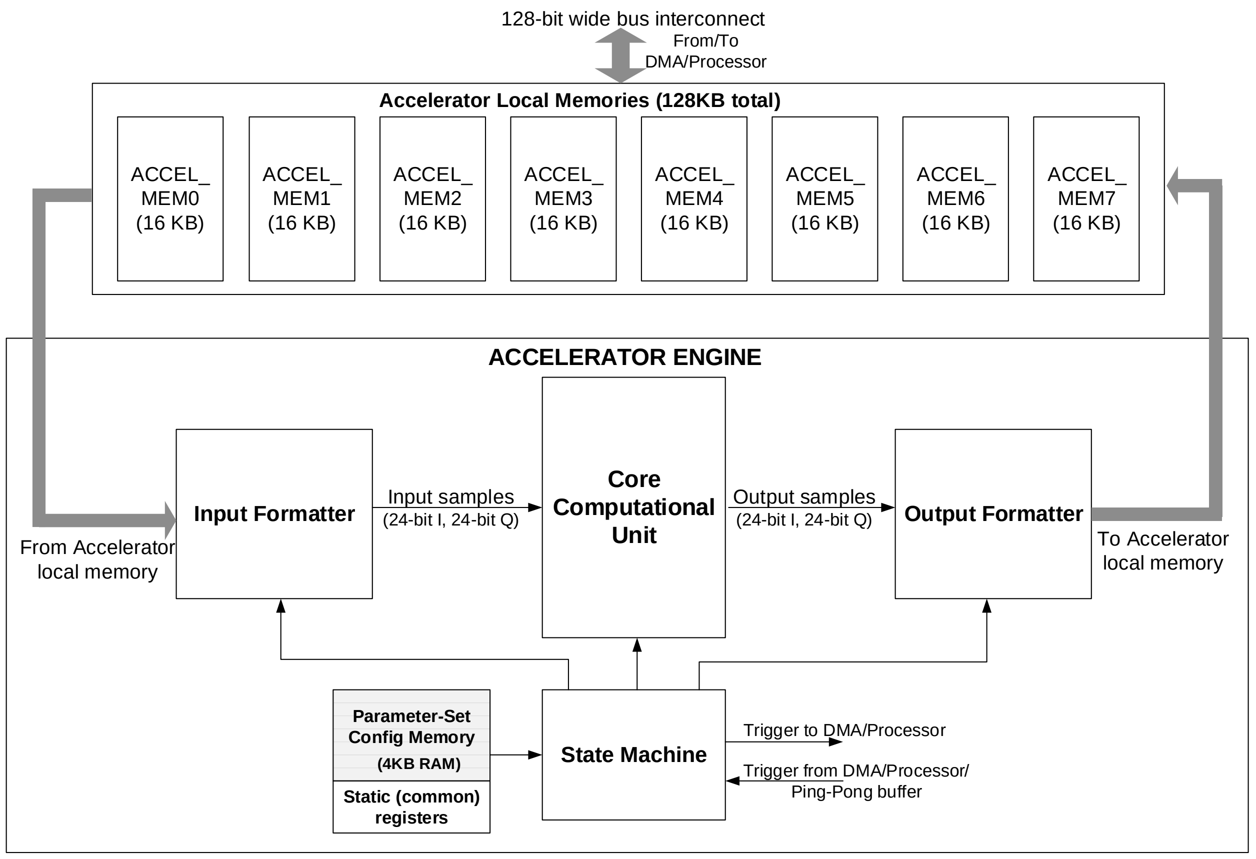
1.3.2 Configuration

The operations of the Radar Hardware Accelerator are configured using registers, which are of two types – *parameter sets* and *common* (common for all parameter sets) registers. The purpose of the parameter sets is to enable a complete sequence of various accelerator operations to be preprogrammed (with appropriate source and destination memory addresses and other configurations specified for each operation in that sequence), such that the accelerator can perform them one after the other, with minimal intervention from the main processor.

The parameter-set register configurations are programmed into a separate 4KB *parameter-set configuration memory*. A state machine built into the accelerator handles the loading of one parameter-set configuration at a time and sequences the preprogrammed operations one after another. This process is further explained in later sections of this user's guide.

1.4 Accelerator Engine Block Diagram

As previously mentioned, the Radar Hardware Accelerator module consists of eight local memories of 16KB each (ACCEL_MEM) and the main accelerator engine. The accelerator engine has the following five components (as shown in Figure 2) – a state machine, input formatter block, output formatter block, core computational unit, and the 4KB parameter-set configuration memory.



128-bit wide bus interconnect
From/To DMA/Processor
Accelerator Local Memories (128KB total)
ACCEL_ MEM0 (16 KB)
ACCEL_ MEM1 (16 KB)
ACCEL_ MEM2 (16 KB)
ACCEL_ MEM3 (16 KB)
ACCEL_ MEM4 (16 KB)
ACCEL_ MEM5 (16 KB)
ACCEL_ MEM6 (16 KB)
ACCEL_ MEM7 (16 KB)
ACCELERATOR ENGINE
Input Formatter
Input samples (24-bit I, 24-bit Q)
Core Computational Unit
Output samples (24-bit I, 24-bit Q)
Output Formatter
From Accelerator local memory
To Accelerator local memory
Parameter-Set Config Memory (4KB RAM)
Static (common) registers
State Machine
Trigger to DMA/Processor
Trigger from DMA/Processor/ Ping-Pong buffer

Figure 2: Accelerator Engine Block Diagram

The purpose of these components is as follows.

- State machine: the state machine is responsible for controlling the overall operation of the accelerator – specifically, the starting, looping, stopping, as well as triggering and handshake mechanisms between the accelerator, DMA, and main processor. The state machine is also closely connected to the parameter-set configuration memory and takes care of sequencing and chaining a sequence of multiple accelerator operations as programmed in the parameter-set configuration memory.
- Input formatter: the input formatter block is responsible for reading the input samples from any one of the local memories and feeding them into the core computational unit. In this process, this block provides flexible ways of accessing the input samples, in terms of 16-bit versus 32-bit aligned input samples, transpose read-out, flexible scaling, and sign extension to generate internal bit-width of 24 bits, and so on. Lastly the input formatter block provides 24-bit complex samples as input to the core computational unit. The local memory (memories) from which the input formatter reads the input samples is called the *source* memory.
- Output formatter: the output formatter block is responsible for writing the output samples from the core computational unit into the local memories. This block also provides flexible ways of formatting the output samples, in terms of 16-bit versus 32-bit aligned output samples, transpose write, flexible scaling from internal bit-width of 24 bits, to 16-bit or 32-bit aligned output samples, sign-extension, and so on. The local memory (memories) to which the output formatter writes the output samples is called the *destination* memory.
- Core computational unit: the core computational unit contains the main computational logic for various operations, such as windowing, FFT, magnitude, log2, and CFAR calculations. The unit

accepts a streaming input from the input formatter block (at the rate of one input sample per clock cycle), performs computations, and produces a streaming output to the output formatter block (typically at the rate of one output sample per clock cycle), with some initial latency depending on the nature of the computations involved.

- Parameter-set configuration memory: this is a 4096-byte RAM that is used to preconfigure the sets of parameters (register settings) for a chained sequence of accelerator operations, which can then be executed by the state machine in a loop. This allows the accelerator to perform a preprogrammed sequence of operations in a loop without frequent intervention from the main processor.

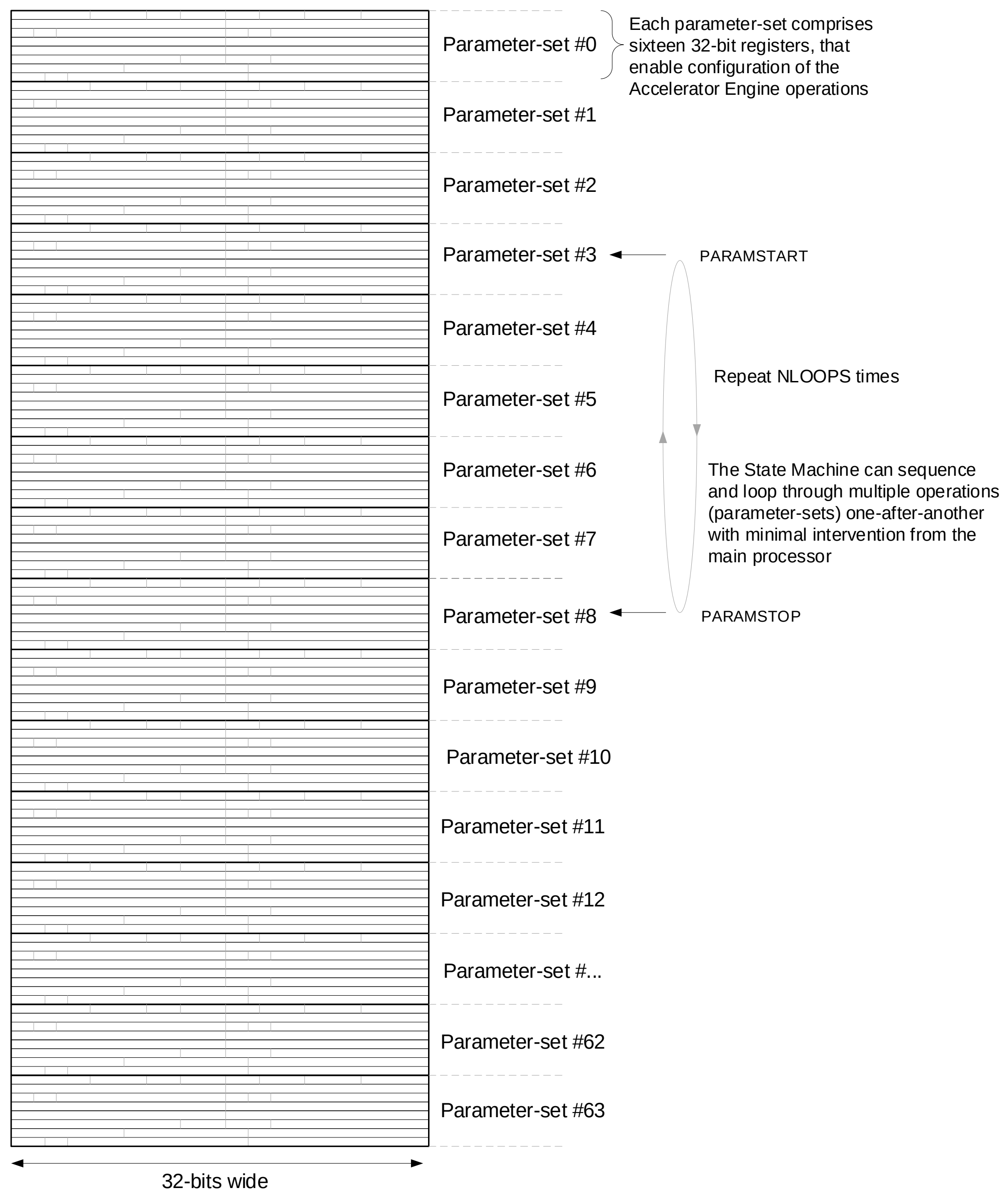


Figure 3: Parameter-Set Configuration Memory (4KB)

The number of parameter sets that can be preconfigured and sequenced (chained) is 64. This means that up to 64 accelerator operations can be chained together and these can then be looped as well, with minimal intervention from the main processor. For example, operations like FFT, log-magnitude, and CFAR detection can be preconfigured in the parameter-set configuration memory and the state machine can be made to sequence them one after another and run them in a loop for specified number of times. There is a provision available to interrupt the main processor and/or trigger a DMA channel at the end of each parameter set if required. This allows various ways by which the accelerator, DMA, and the main processor can work together to establish a data and processing flow. As shown in Figure **3**, each parameter set contains the equivalent of sixteen 32-bit registers, which corresponds to total RAM size of 64 × 16 × 32 bits = 4KB for the parameter-set configuration memory.

The layout of the parameter-set register map is provided in Appendix A. Note that the parameter-set RAM must be programmed using 32-bit word writes only (i.e., byte-writes and half-word writes are not supported). The detailed descriptions of the registers is provided in the various sections, as and when the functionality of each component is presented.

Typically, all necessary parameter sets can be pre-configured before triggering execution by the accelerator. If needed, the parameter sets can be modified on-the-fly but without modifying the parameter set being executed at that time.

1.5 Accelerator Engine Operation

The accelerator engine and the local memories run on a single clock domain. The overall operation of the accelerator can be summarized as follows. The accelerator engine is configured by the main processor through common configuration registers (common for all parameter sets), as well as the parameter-set configuration memory. As explained earlier, the former comprises common register settings for overall control of the accelerator engine, and the latter comprises the 64 parameter-set specific settings which control the functioning of the accelerator for each of its *chained* sequence of operations.

When the accelerator engine is enabled, the state machine kicks off and controls the overall operation of the accelerator, which involves loading the parameter sets one at a time from the parameter-set configuration memory into various internal registers of the accelerator engine and running the accelerator as per the programmed configuration for each parameter set one after another. The entire procedure then repeats in a loop for a programmed number of times (NUMLOOPS described later).

Each parameter set includes various configuration details such as the accelerator mode of operation (FFT, Log2, and so on), the source memory address, number of samples, the destination memory address, input formatting, output formatting, trigger mode for controlling the start of computations to ensure proper handshake with the DMA, and so on.

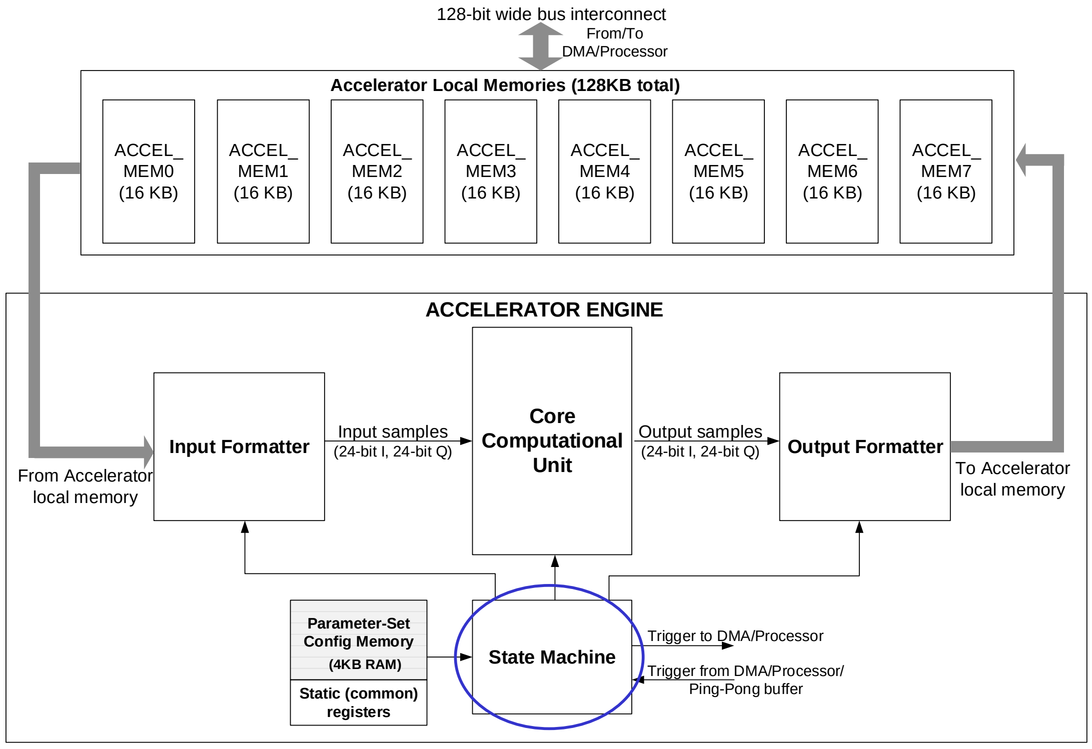
1.5.1 Data Throughput

Once the state machine has loaded the registers corresponding to the current parameter set to be executed, the data flow happens as follows: at each clock cycle, one sample from the source memory is read by the input formatter and fed into the core computational unit with appropriate scaling and formatting as configured. The data interface between the input formatter and the core computational unit is a 24-bit complex bus (24-bit for I and 24-bit for Q) which streams one input sample every clock cycle. The core computational unit processes this streaming sequence of input samples and in general, produces a streaming output also at one sample every clock cycle, after an initial latency period. Thus for most operations (FFT, log-magnitude, CFAR, and so on), in steady state the core computational unit maintains a streaming data rate of one sample per clock cycle. The data interface between the core computational unit and the output formatter is also a 24-bit complex bus (24-bit for I and 24-bit for Q) and the output formatter is responsible for writing into the destination memory, with appropriate scaling and formatting as configured.

The next section provides more details regarding the state machine, including its detailed operation, registers, trigger mechanisms, and so on.

2 Accelerator Engine – State Machine

This section describes the state machine block present in the accelerator engine (see Figure 4). This block, together with the input formatter and output formatter blocks described in the next two sections, provides the overall framework for establishing the data flow and using the accelerator for various computations.



128-bit wide bus interconnect
From/To DMA/Processor
Accelerator Local Memories (128KB total)
ACCEL_ MEM0 (16 KB)
ACCEL_ MEM1 (16 KB)
ACCEL_ MEM2 (16 KB)
ACCEL_ MEM3 (16 KB)
ACCEL_ MEM4 (16 KB)
ACCEL_ MEM5 (16 KB)
ACCEL_ MEM6 (16 KB)
ACCEL_ MEM7 (16 KB)
ACCELERATOR ENGINE
Input Formatter
Input samples (24-bit I, 24-bit Q)
Core Computational Unit
Output samples (24-bit I, 24-bit Q)
Output Formatter
From Accelerator local memory
To Accelerator local memory
Parameter-Set Config Memory (4KB RAM)
Static (common) registers
State Machine
Trigger to DMA/Processor
Trigger from DMA/Processor/ Ping-Pong buffer

Figure 4: State Machine

2.1 State Machine

The state machine controls the overall functioning of the Radar Hardware Accelerator. The state machine controls the enabling and disabling of the accelerator, as well as supports sequencing an entire set of operations (configured using parameter-set configuration memory), and looping through those operations one after another without needing frequent intervention from the main processor.

2.1.1 State Machine – Operation

The state machine block and the entire accelerator remain in reset and disabled state by default. The state machine (and hence the accelerator in general) is enabled by setting the HWA_CLK_EN register bit, followed by writing 111b into the HWA_EN register.

Note that a complete list of registers pertaining to the state machine is provided in Table 1. Some of the registers are common (common for all parameter sets) registers, whereas some other registers are parameter-set registers, which as explained in the previous section means that they can be uniquely programmed for each of the 64 parameter sets. For each register, Table 1 lists whether it is part of the parameter set or not. Table 1 also provides a brief description of each register.

When enabled, the state machine steps through (one after another) the parameter sets programmed in the parameter-set configuration memory and executes the computations as per the configuration of each parameter set. The registers PARAM_START_IDX and PARAM_END_IDX define the starting index and ending index within the 64 parameter sets, so that only those parameter sets between the start and end indices are executed by the accelerator, as shown in Figure **3**. The state machine also loops through these parameter sets for a total of NUMLOOPS times (unless NUMLOOPS is programmed as 0 or 4095, in which case the loop does not run or runs infinite times respectively). As an example, if the state machine needs to be configured to run the first four parameter sets in a loop 64 times, then the registers should be programmed as follows: PARAM_START_IDX = 0, PARAM_END_IDX = 3, and NUMLOOPS = 64.

For each parameter set, there is a TRIGMODE register, which is used to control when the state machine starts executing the computations for that parameter set. This control is useful, for example, to ensure that the input data is ready in the accelerator local memory (source memory) before the computations are started. Specifically, it is possible to trigger the start of computations after completion of a DMA transfer, or, after a CSI2 line is received, and so on. The TRIGMODE register setting thus controls when the accelerator operation is triggered for the current parameter set and there are four trigger mechanisms supported as listed in the next subsection. Once triggered, the state machine loads all the registers from the parameter-set configuration memory for the current parameter set into corresponding internal registers of the accelerator and starts the actual computations for that parameter set. After completion of computations of the current parameter set, it moves to the next parameter set.

After a sequence of operations as programmed in the parameter set(s) for the specified number of loops is complete, the accelerator provides a completion interrupt (DSS_HWA_THREAD1_LOOP_INT) to the processor. The accelerator can be reconfigured as desired. For reconfiguration, the following procedure must be followed. The accelerator must be disabled by writing 000b to the HWA_EN register. Then, a reset must be asserted by writing 111b followed by 000b to the HWA_RESET register. The new configurations can now be written in to the accelerator, and then the accelerator can be enabled again by writing 111b to HWA_EN.

2.1.2 State Machine – Trigger Mechanisms (Incoming)

As mentioned in the previous subsection, for each parameter set, the start of the computations can be triggered based on specific events. Four trigger mechanisms are supported as follows.

Immediate trigger (TRIGMODE = 000b): In this case, the state machine does not wait for any trigger and starts the accelerator computations immediately for the current parameter set. This mode is applicable when chaining (sequencing) a set of operations one after another in the accelerator without any need for control handshake or data exchange outside the accelerator (for example, when chaining FFT and log-magnitude operations) with no need to wait for a trigger in between.

Wait for processor-based software trigger (TRIGMODE = 001b, TRIGMODE = 111b): This is a software-triggered mode that is useful when the main processor(s) must directly control the data flow and start or stop of accelerator computations. In this trigger mode, the state machine waits for a software-based trigger, which involves the main processor(s) setting a separate self-clearing bit in a FW2HWA_TRIGGER_0 or FW2HWA_TRIGGER_1 register (single-bit register). The state machine keeps monitoring that register bit and waits as long as the value is zero. When the value becomes 1 (set), the state machine gets triggered to start the accelerator operations for the current parameter set. FW2HWA_TRIGGER_0 register bit corresponds to TRIGMODE = 001b and FW2HWA_TRG_1 corresponds to TRIGMODE = 111b.

TRIGMODE = 010b : mode is reserved.

Wait for the DMA-based trigger (TRIGMODE = 011b): This trigger mode is useful when a DMA transfer completion must be used to trigger the start of the accelerator computations for the current parameter set. The primary purpose of this trigger mode is as follows; when performing second dimension FFT, the DMA is used to bring the FFT input samples from the Radar data memory to the local memory of the accelerator. Upon completion of each DMA transfer, it is useful to automatically trigger the accelerator to perform the FFT.

To achieve this, the state machine of the accelerator has a 32-bit register called the DMA2HWA_TRIGGER register, where each register bit maps to one of 32 DMA channels that are associated with the accelerator. To use the DMA-based trigger mode, the HWA_TRIGSRC register in the current parameter set must be programmed to the DMA channel whose completion we wish to monitor. The state machine then monitors the corresponding register bit in the DMA2HWA_TRIGGER register, and triggers the execution of the current parameter set only when that register bit gets set. For e.g. if HWA_TRIGSRC is programmed to 5, then the current parameter set will execute only once the register bit #5 gets set in DMA2HWA_TRIGGER.

The user may utilize the EDMA's linking capability to set the appropriate register bit in DMA2HWA_TRIGGER. Linking is a programmable feature of the EDMA, where the completion of a DMA transfer can automatically trigger a second DMA transfer. In the present context, the DMA transfer that moves data to the local memory of the accelerator can be linked to a second DMA whose purpose is to write a one-hot signature into DMA2HWA_TRIGGER to set a specific register bit and trigger the accelerator. Note that there are 32 read-only, one-hot, signature registers (SIG_DMACH1_DONE, SIG_DMACH2_DONE, and more) that are available. These registers are simply read-only registers which contain hard-coded values (each register is a one-hot signature – 0x0001, 0x0002, 0x0004, 0x0008, and so on). For convenience, these hard-coded 32 read-only signatures can be used, so that the second DMA can simply copy from one of these SIG_DMACHx_DONE registers into the DMA2HWA_TRIGGER register to set the appropriate register bit.

Wait for hardware trigger (TRIGMODE = 0b100): These trigger modes are useful when a hardware signal such as CSI interrupt needs to be used to trigger the start of the accelerator computations for the current parameter set. The hardware trigger sources can be either CSI #1 frame start, CSI #1 line-end, or CSI #2 frame start, or CSI #2 line end. The HWA_TRIG_SRC register in the parameter-set decides which of these trigger sources is selected. . The valid range for HWA_TRIG_SRC register is 0 to 19, and these values correspond to different trigger sources – specifically, a value of 0 selects the right most trigger and 19 selects left most trigger in the following trigger sources: RCSS_CSI2A_SOF_INT[0,1] RCSS_CSI2A_EOL_CNTX[0..7]_INT, RCSS_CSI2B_SOF_INT[0,1], RCSS_CSI2B_EOL_CNTX[0..7]_INT. Please refer to the device TRM for more details.

2.1.3 State Machine – Trigger Mechanisms (Outgoing)

After the accelerator computations for the current parameter set are triggered (using one of the four incoming trigger mechanisms mentioned in the previous subsection), it performs the actual computation operations for that parameter set. These computations typically take several tens or hundreds of clock cycles, depending on the nature of the configuration programmed. Once the accelerator completes its computation operations for the current parameter set, the state machine advances to the next parameter set and repeats the same process. But before advancing to the next parameter set, it can interrupt the main processor and/or trigger a DMA channel. This provision is useful if the main processor is required to read or write registers or memory locations at the end of the current parameter set. Also, this provision is useful for triggering a DMA channel, so that the output of the accelerator can be copied out of the accelerator local memories.

There are two trigger mechanisms provided as follows:

- Interrupt(s) to main processor (CPU_INTR1_EN = 1, CPU_INTR2_EN = 1): The accelerator interrupts the main processor(s) at the end of completion of computations for the current parameter set, if the register bit CPU_INTR1_EN or CPU_INTR2_EN is set. Two interrupt signals are available and they are enabled or disabled for each parameter-set by these two register bits. Setting CPU_INTR1_EN in a parameter-set enables DSS_HWA_PARAM_DONE_INTR1 interrupt to be generated at the end of that parameter-set. Setting CPU_INTR2_EN in a parameter-set enables DSS_HWA_PARAM_DONE_INTR2 interrupt to be generated at the end of that parameter-set.
- Trigger to DMA (DMATRIG_DMATRIG_ENEN = 1): The accelerator gives a trigger to a DMA channel at the end of completion of computations for the current parameter set, if the register bit DMATRIG_EN is set. If DMATRIG_EN is set, then the particular DMA channel as specified in a separate HWA2DMA_TRIGDST register (valid values are 0 to 31, for the 32 DMA channels dedicated for the accelerator) is triggered. Thus, it is possible to preconfigure up to 32 DMA channels and trigger the appropriate one at the end of the computations of the current parameter set. The trigger from accelerator to the DMA channels can also be emulated by the processor, by writing to a FW2DMA_TRIGGER register. This can be used by the processor to kick-start a full/repetitive chain of operations, that are then subsequently managed between the DMA and the accelerator without further processor involvement – for example, the processor writes to the FW2DMA_TRIGGER register to trigger a DMA channel for the first time, and this kicks off a series of back-to-back data transfers and accelerator computations, with the DMA and accelerator hand-shaking with each other.

2.1.4 State Machine – Register Descriptions

Table 1 lists all the registers of the state machine block. As explained previously, some of the registers are common (common for all parameter sets) registers, whereas some others are *part of each parameter set*. For each register, this distinction is captured as part of the register description in Table 1.

Table 1 : State Machine Registers

Register	**Width**	**Parameter Set**	**Description**
HWA_EN	3	No	Enable and Disable Control: This register enables or disables the entire Radar Hardware Accelerator. The reason for a 3-bit register (instead of 1-bit) is to avoid an accidental bit-flip (for example, transient error caused by a neutron strike) from unintentionally turning on the accelerator engine. A value of HWA_EN = 111b enables the Radar Hardware Accelerator and any other value of the register keeps the accelerator engine in disabled state.

HWA_CLK_EN	1	No	Clock-gating Control: This register bit controls the enable/disable for the clock of the Radar Accelerator. This register bit can be set to 0 to clock-gate the accelerator when not using the accelerator. Before enabling the accelerator or before configuring the accelerator's registers, this register bit should be set first, so that the clock is available.
HWA_RESET	3	No	Software Reset Control: This register provides software reset control for the Radar Hardware Accelerator. The assertion of these register bits by the main processor will bring the accelerator engine to a known reset state. This is mostly applicable for resetting the accelerator in case of unexpected behavior. Under normal circumstances, it is expected that whenever the accelerator is enabled (from disabled state), it always comes up in a known reset state automatically. The recommended sequence to be followed in case software reset is desired is to write 111b to this register and then a 000b, before the clock is enabled to the accelerator.
NUMLOOPS	12	No	Number of loops: This register controls the number of times the state machine will loop through the parameter sets (from a programmed start index till a programmed end index) and run them. The maximum number of times the loop can be made is run is 4094. A value of 4095 (0xFFF) programmed in this register should be considered as a special case and it should be interpreted as an infinite loop mode, for example, keep looping and never stop the accelerator engine unless reset by the main processor. A value of zero programmed in this register means that the looping mechanism is disabled. In this case, the accelerator engine can still be used under direct control of the main processor (without the state machine looping provision coming into the picture).
PARAM_START_IDX	4	No	Parameter-set Start Index: These registers are used to control the start and stop index of the parameter set through which the state machine loops through. The state machine starts at the parameter set specified by PARAM_START_IDX and loads each parameter set one after another and runs the accelerator as per that configuration. When the state machine reaches the parameter set specified by PARAM_END_IDX, it loops back to the start index as specified by PARAM_START_IDX.
PARAM_END_IDX	4	No	Parameter-set Stop Index: Refer register description for PARAM_START_IDX
HWA_DYN_CLK_EN	1	No	Dynamic Clock-gating Control: Setting this register bit to '1' enables the capability to clock gate the unused computation engines (i.e., from the four computation engines, namely FFT, CFAR, Memory compression, Local Maxima) to save power consumption, based on the specific parameter-set being executed.

TRIGMODE	4	Yes	Trigger mode select: 0000b – Immediate trigger 0001b – Software trigger 0010b – Reserved 0011b – DMA-based trigger 0100b – Hardware based trigger 0101b – Reserved 0110b – Reserved 0111b – M4 Micro-controller based trigger (equivalent to software trigger, differentiating trigger source between external and HWA CPU.
FW2HWA_TRIGGER_0	1	No	Software trigger bit: This register bit is relevant whenever software triggered mode is used (TRIGMODE = 001b). Whenever this software triggered mode is configured for a parameter set, the state machine keeps monitoring this register bit and waits as long as the value is zero. The main processor software can set this register bit, so that the state machine gets triggered and starts the accelerator operations for that parameter set.
FW2HWA_TRIGGER_1	1	No	Software trigger bit: This register bit is similar to FW2HWA_TRIGGER_0, except that this register bit corresponds to TRIGMODE = 111b.
DMA2HWA_TRIGGER	32	No	DMA trigger register: This register is relevant whenever DMA triggered mode is used (for example, TRIGMODE = 011b). Whenever a DMA channel has finished copying input samples into the local memory of the accelerator and wants to trigger the accelerator, the procedure to follow is to use a second linked DMA channel to write a 32-bit one-hot signature into this register to trigger the accelerator. In DMA triggered mode, the state machine keeps monitoring this 32-bit register and waits as long as a specific bit (see DMA2HWA_TRIGSRC) in this register is zero. The second linked DMA channel writes a one-hot signature that sets the specific bit, so that the state machine gets triggered and starts the accelerator operations for that parameter set.
DMA2HWA_TRIGSRC	5	Yes	DMA channel select for DMA completion trigger: This parameter-set register is relevant whenever DMA triggered mode is used (for example, TRIGMODE = 011b). This register selects the bit number in DMA2HWA_TRIGGER for the state machine to monitor to trigger the operation for that parameter set.
CPU_INTR1_EN	1	Yes	Completion interrupt to main processor: This parameter-set register is used to enable/disable interrupt to the main processor upon completion of the accelerator operation for that parameter set. If enabled, the main processor receives an interrupt from the Radar Hardware Accelerator at the end of operations for that parameter set, so that the main processor can take any necessary action. Two interrupts are available, and this register bit enables or disables the first interrupt.

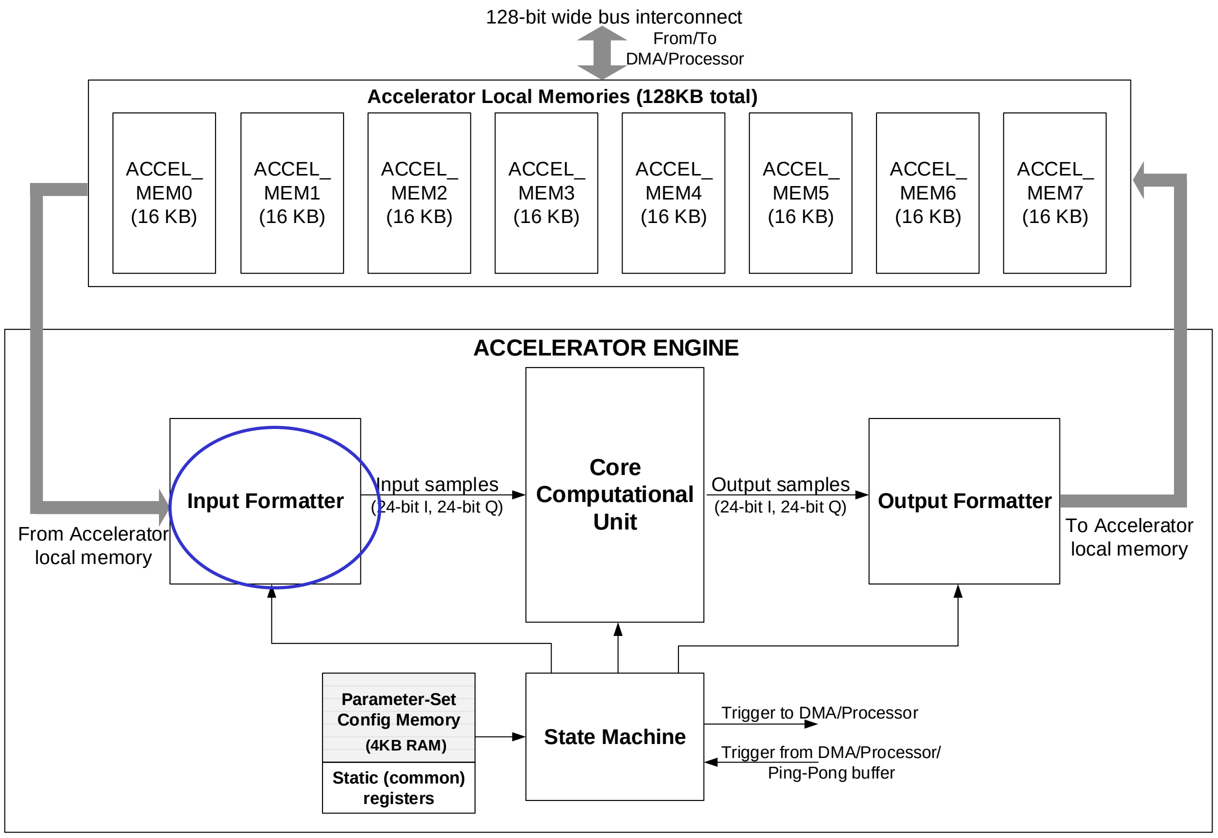
CPU_INTR2_EN	1	Yes	Completion interrupt to main processor: Similar to CPU_INTR1_EN. This register bit enables or disables the second interrupt to the main processor.
PARAM_DONE_SET_STATUS_0 PARAM_DONE_SET_STATUS_1	32	No	Parameter-set done status: These read-only status registers can be used by the main processor to see which parameter sets are complete that led to the interrupt to the main processor. The individual bits in these 32-bit status register indicate which of the 64 parameter sets have completed. These status bits are not automatically cleared, but they can be individually cleared by writing to another set of 32-bit registers (PARAM_DONE_STATUS_CLR)
PARAM_DONE_STATUS_CLR_0 PARAM_DONE_STATUS_CLR_1	32	No	Refer register description for PARAM_DONE_SET_STATUS
DMATRIG_EN	1	Yes	Completion trigger to DMA: This parameter-set register is used to enable DMA channel trigger upon completion of the accelerator operation for that parameter set. This trigger mechanism enables the accelerator to hand-shake with the DMA so that output data samples are copied out of the accelerator local memory. If enabled, the accelerator triggers a specified DMA channel, so that the output samples can be shipped from the local memory to Radar data memory.
HWA2DMA_TRIGDST	5	Yes	DMA channel select for accelerator completion trigger: This parameter-set register is used to select which of the 32 DMA channels allocated to the accelerator should be triggered upon completion of the accelerator operation for that parameter set. This register is to be used in conjunction with DMATRIG_EN.
FW2DMA_TRIGGER	32	No	Trigger from processor to DMA: This register can be used by the processor to trigger a DMA channel for the first time, so that a full sequence of repeated operations between the DMA and the accelerator gets kick-started.
PARAMADDR	6	No	Debug register for current parameter-set index: This read-only status register indicates the index of the current parameter set that is under execution. This is useful for debug, where parameter sets can be executed in single-step manner (one-by-one) using SW trigger mode for each of them. In such a debug, this register indicates which parameter set is currently waiting for the SW trigger.
LOOP_CNT	12	No	Debug register for current loop count: This read-only status register indicates what is the loop count that is presently running. When the state machine is programmed for NUMLOOPS loops, this register shows the current loop count that is running.
TRIGGER_SET_STATUS_0	32	No	Debug register for trigger status: This is a read-only status register, which indicates the trigger status of the accelerator, for example, whether a DMA trigger was ever received (refer TRIGMODE). The 32 bits in this register correspond to the 32 DMA trigger bits (refer DMA2HWA_TRIGGER).
TRIGGER_SET_STATUS_1	32	No	Debug register for trigger status: This is a read-only status register, which indicates the trigger status of the accelerator, for example, whether a specific hardware trigger or software trigger or context switch trigger has even been received. <TO_BE_UPDATED_IN_FUTURE_DOC_VERSION>

TRIGGER_SET_IN_CLR_0	1	No	Clear trigger status read-only register: This register-bit when set clears the trigger status register TRIGGER_SET_STATUS_0 described above.
TRIGGER_SET_IN_CLR_1	1	No	Clear trigger status read-only register: This register-bit when set clears the trigger status register TRIGGER_SET_STATUS_1 described above.
FORCED_CONTEXTSW_EN	**1**	Yes	Force context switch: This register bit is useful in Context Switching. If this bit is set, the state machine switches the context to other thread after the completion of the current parameter-set. Refer the Context Switching section in Part 2 of the user guide for details.
CONTEXTSW_EN	1	Yes	Enable context switch: This register bit is useful in Context Switching. If this bit is set, the state machine is allowed to switch context at the end of execution of this parameter-set. Refer the Context Switching section in Part 2 of the user guide for details
PARAM_START_INDX_ALT	10	No	Refer Context Switching section in Part 2 of user guide.
PARAM_END_IDX_ALT	10	No	Refer Context Switching section in Part 2 of user guide.
ALT_NUMLOOPS	12	No	Refer Context Switching section in Part 2 of user guide.

The next two sections cover the Input Formatter and Output Formatter blocks, including their detailed operation, registers and usage procedure.

3 Accelerator Engine – Input Formatter

This section describes the input formatter block present in the accelerator engine (see Figure 5).



128-bit wide bus interconnect
From/To DMA/Processor
Accelerator Local Memories (128KB total)
ACCEL_ MEM0 (16 KB)
ACCEL_ MEM1 (16 KB)
ACCEL_ MEM2 (16 KB)
ACCEL_ MEM3 (16 KB)
ACCEL_ MEM4 (16 KB)
ACCEL_ MEM5 (16 KB)
ACCEL_ MEM6 (16 KB)
ACCEL_ MEM7 (16 KB)
ACCELERATOR ENGINE
From Accelerator local memory
Input Formatter
Input samples (24-bit I, 24-bit Q)
Core Computational Unit
Output samples (24-bit I, 24-bit Q)
Output Formatter
To Accelerator local memory
Parameter-Set Config Memory (4KB RAM)
Static (common) registers
State Machine
Trigger to DMA/Processor
Trigger from DMA/Processor/ Ping-Pong buffer

Figure 5: Input Formatter

3.1 Input Formatter

The input formatter is used to access, format, and feed the data from the local memories of the accelerator as 24-bit I and 24-bit Q samples into the core computational unit. The input formatter provides various capabilities to access and format the samples from the local memories – especially, various multidimensional access patterns (for example transpose access), 16-bit or 32-bit aligned word access, scaling using bit-shifts to generate 24-bit wide samples from 16-bit or 32-bit words, real versus complex input, sign extension, conjugation, and more.

3.1.1 Input Formatter – Operation

The input formatter block is responsible for reading the input samples from the accelerator local memory and feeding them into the core computational unit (see Figure 5). The data flow from the input formatter, through the core computational unit, to the output formatter is designed to sustain a steady-state throughput of one complex sample per clock cycle. The input formatter thus feeds one sample (24-bit I and 24-bit Q) into the core computational unit every clock cycle.

To make the best use of the capabilities of the core computational unit and to allow meaningful chaining of radar signal processing operations with minimal intervention from the processor, the input formatter supports flexibility in how the input samples are accessed from the memory and how they are formatted and fed into the core computational unit.

The memory from which the input formatter picks up the data is referred to as *source memory*. Note that any of the eight accelerator local memories can be the source memory. However, as will be described in a subsequent section, there is an important restriction which explains that the source memory cannot be the same as the destination memory (which is the memory to which the output formatter writes the output data).

3.1.2 Input Formatter – 2D Indexed Addressing for Source Memory Access

The parameter-set register SRCADDR specifies the start address at which the input samples must be accessed. This register is a byte-address, and a value of 0x00000 corresponds to the first memory location of ACCEL_MEM0 memory. The SRCADDR register maps to the entire 128 KB address space of the eight accelerator local memories (8x16KB). Note that even though SRCADDR register is a 20-bit register, only the 17 LSB bits are used. The 3 MSB bits are reserved for future extension purpose.

The input data can be read from the memory as either 16-bit wide samples or 32-bit wide samples. Also, they can be read as real samples or complex samples. These two aspects are configured using register bits SRC16b32b and SRCREAL. See Table 2 for a description of these and other registers pertaining to the input formatter block. As an example, if SRC16b32b = 0 and SRCREAL = 0, then the input samples are read from the memory as 16-bit complex samples (16-bit I and 16-bit Q), shown in Figure 6.

An important feature of the input formatter block is that it supports flexible access pattern to fetch data from the source memory, which makes it convenient when the data corresponding to multiple RX channels are interleaved or when performing multi-dimensional (FFT) processing. This feature is facilitated through the SRCAINDX, SRCACNT, SRCBINDX, and BCNT registers, which are part of each parameter-set configuration.

The register SRCAINDX specifies how many bytes separate successive samples to be fetched from the source memory and the register SRCACNT specifies how many samples need to be fetched per *iteration*. An *iteration* is typically one computational routine, such as one FFT operation. It is possible to perform multiple iterations back-to-back – for example, four FFT operations corresponding to four RX channels. The register SRCBINDX specifies how many bytes separate the start of input samples for successive iterations and BCNT specifies how many iterations to perform back-to-back. These registers can be better understood using the example given in Figure 6.

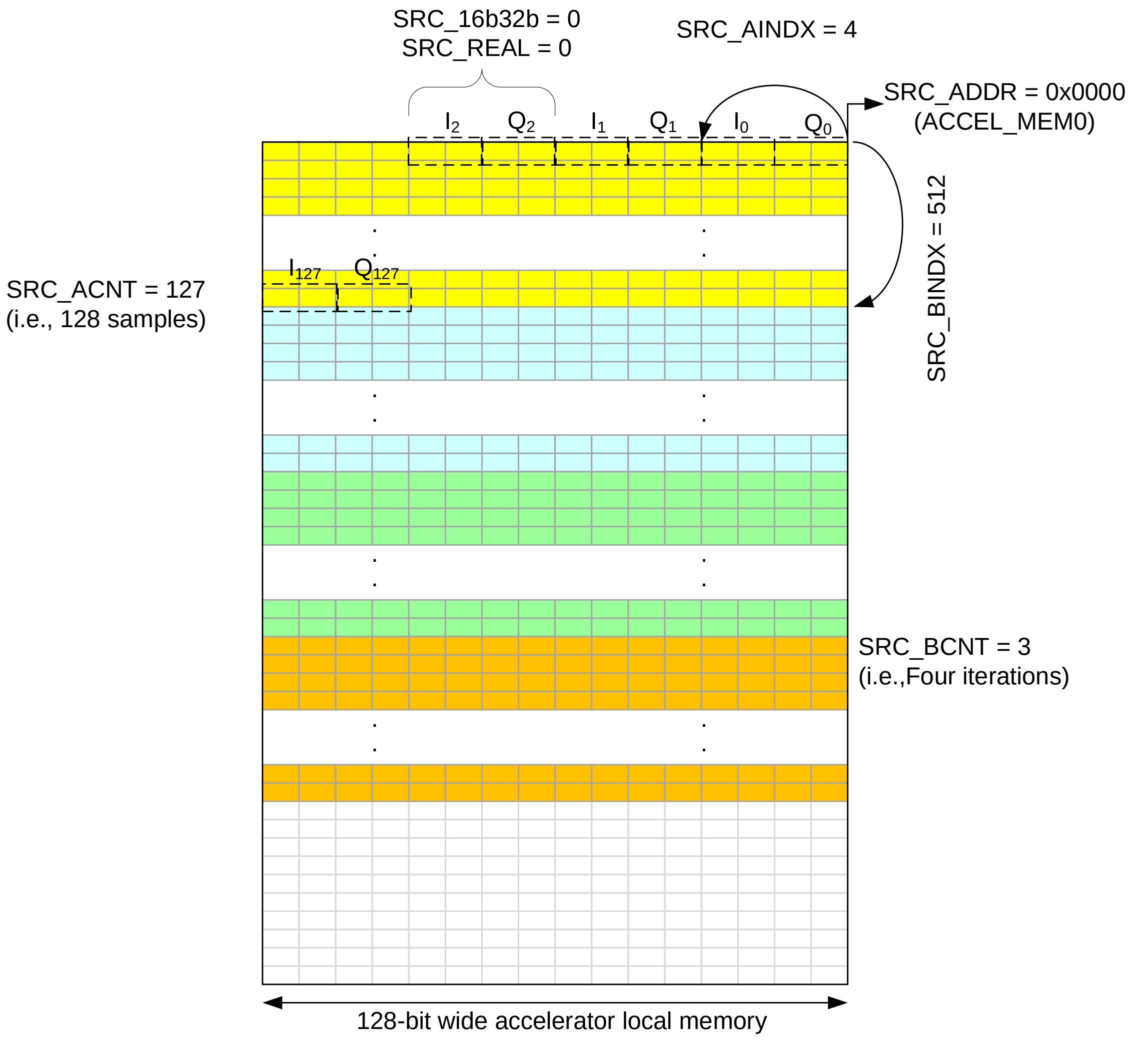


Figure 6: Input Formatter Source Memory Access Pattern (Example)

In the example shown in Figure 6, the input data consists of complex data (16-bit I and 16-bit Q) that is contiguously present in ACCEL_MEM0. The data in memory consists of four sets of 128 samples each (say, corresponding to four RX antennas) and these are shown in four different colors. Because each sample occupies 4 bytes and the samples are contiguously placed in the memory starting at the beginning of ACCEL_MEM0, values of SRCADDR = 0 and SRCAINDX = 4 are used to fetch these samples.

In each clock cycle, the input formatter fetches one complex sample from the memory and feeds it into the core computational unit (with appropriate scaling, as described later). Because there are 128 samples to be fed for the first iteration (computational routine), a value of SRCACNT = 127 is used. For the second iteration, the samples are fetched starting from a memory location that is SRCBINDX (=128 × 4 = 512) bytes away from SRCADDR.

This process repeats for the programmed number of iterations as per the BCNT register. For example, the value of BCNT = 3 used in this example corresponds to four iterations. Note that the registers shown here are part of parameter-set configuration registers and the four iterations described here can be performed using a single parameter set. Thus, A-dimension is used to run through samples of a given vector, and B-dimension is used to repeat (iterate) the same operation for multiple vectors.

In addition to A and B dimensions, the Input Formatter also includes a provision for C-dimension. The C-dimension is only available in a certain restricted mode of operation called the Local Maxima Engine and its usage is explained in the part 2 of the user guide.

An important restriction in programming the registers related to source memory access pattern is that the input formatter can only read data from one memory row (128-bit memory location) in a clock cycle. Therefore, if a sample is placed in memory such that the real-part (I value) is at the end of one memory location and the imaginary part (Q value) is at the beginning of the next memory location, then that would be an invalid configuration (see Figure 7). Further, although the accelerator supports byte-addresses, only even values are allowed for SRCADDR, SRCAINDX, SRCBINDX and SRCCINDX.

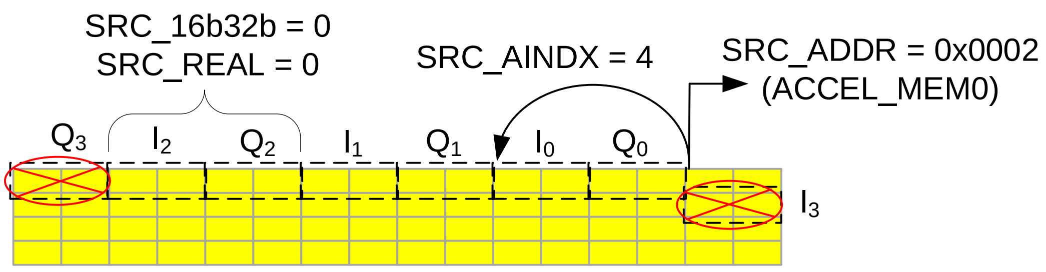


Figure 7: Invalid Configuration Example

3.1.3 Input Formatter – Circular and Shuffled addressing

The input formatter additionally supports circular addressing in each of the A, B and C dimensions. For the A dimension, the register, SRCA_CIRCSHIFT controls the initial offset, while the registers, SRCA_CIRCSHIFTWRAP and SRC_CIRCSHIFTWRAP3X (bit 0) indicate the circular modulus at which the wrap-around happens.

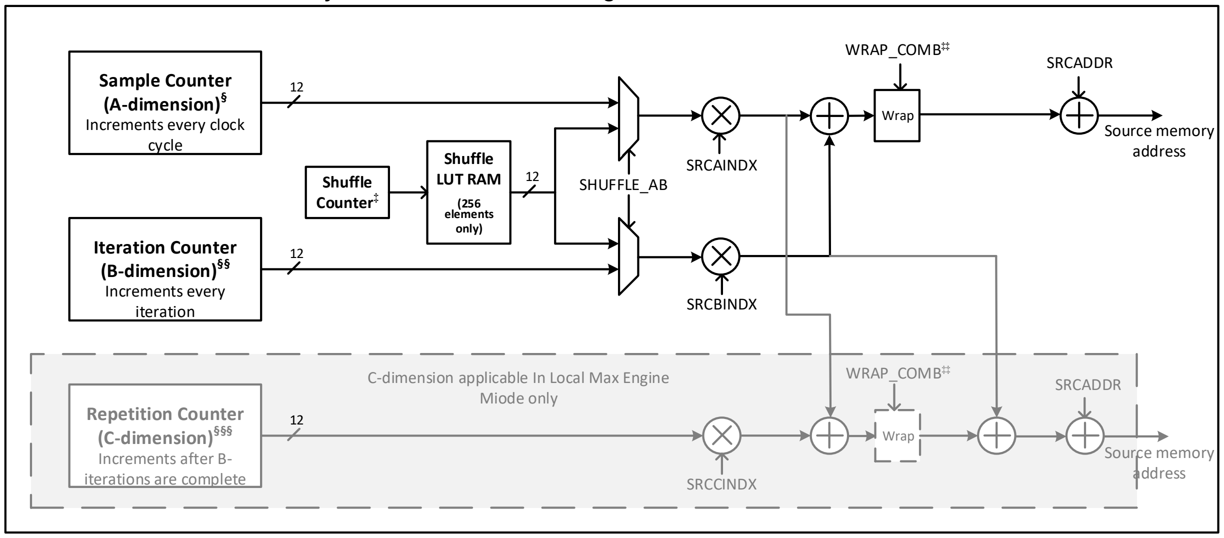
For example, if SRCA_CIRCSHIFT is programmed as 7, then the input formatter skips sample indices 0 to 6 and reads samples from the source memory starting directly from index 7 (i.e., the 8th sample). Then, for wrap-around, if SRCA_CIRCSHIFTWRAP is programmed with a non-zero value and the LSB bit (bit 0) of SRC_CIRCSHIFTWRAP3X is 0, then the sample index wraps around (i.e., resets to 0) at 2^SRCA_CIRCSHIFTWRAP. Continuing the previous example, if SRCA_CIRCSHIFT = 7, SRCA_CIRCSHIFTWRAP = 9, and the LSB bit of SRC_CIRCSHIFTWRAP3X = 0, then the sample indices will be in the following order: 7, 8, 9, 10, 11, …, 510, 511, 0, 1, 2, 3, 4, 5, 6. On the other hand, in the same example, if the LSB bit (bit 0) of SRC_CIRCSHIFTWRAP3X is set equal to 1, then the sample index wraps around at 3*2^SRCA_CIRCSHIFTWRAP, and then the sample indices will be in the following order: 7, 8, 9, 10, 11, …, 1534, 1535, 0, 1, 2, 3, 4, 5, 6. Circular shifting is generally useful in CFAR and Local Maxima engines. Its usage is explained in more detail in the CFAR engine section in the part 2 of the user guide.

Circular shifting is also supported in B-dimension with registers SRCB_CIRCSHIFT, SRC_CIRCSHIFTWRAP and SRC_CIRCSHIFTWRAP3X (bit 1).

In addition to linear and circular addressing, the input formatter allows a shuffled access, in either the A-dimension or the B-dimension (selectable through the register SHUFFLE_AB). When engaged in A-dimension, the A-dimension's sample index is remapped (or shuffled) through a programmable look up table, Shuffle LUT. The Shuffle LUT is a 256 element vector (RAM) storing 12-bit numbers and this LUT can be programmed by the user with the shuffled index pattern that is desired. This feature is useful in rearranging angle dimension input samples for FFT. Note that the Shuffle LUT can hold only 256 elements atmost, and therefore the shuffled access is only supported for a maximum count of 256 (eg. SRCACNT of 255). Since Angle dimension FFT is usually small in size, this limitation is acceptable.

The start index within the Shuffle LUT (RAM) is also configurable in the parameter-set using the 4-bit register SHUFFLE_INDX_START_OFFSET. This 4-bit register value along with 4 LSBs padded as zeros becomes the 8-bit start index for the 256-element look-up table RAM. This feature allows multiple small shuffle patterns to be pre-stored in the Shuffle LUT, so that any of those patterns can be selected by appropriately setting this start index register in the parameter-set. For example, if SHUFFLE_INDX_START_OFFSET = 0010b, then the shuffled indices are picked up starting from index 32 (i.e., 00100000b) of the Shuffle LUT, and these shuffled indices in turn are used to pick up appropriate input samples from the source memory.
While A- and B-dimension have independent circular wrap around capabilities through the registers explained above, some use cases may need a combined (A, B) wrap around capability. A limited wrap around capability in the combined (A, B) dimension is provided through the register WRAP_COMB (see Figure 8). If shuffling is enabled, it is possible that after combining (A, B) dimension, the combined value attempts to access an address outside the valid range. The register WRAP_COMB can be programmed to overcome this, and the input formatter wraps back any access outside the range [0, WRAP_COMB) to fall within this range.

The below figure shows how the sample count value (A-dimension sample index) and the iteration count value (B-dimension count) are used to calculate the address of the input sample to be fetched from the source memory. The exact addressing calculation is indicated in the schematic of the


Sample Counter (A-dimension)§
Increments every clock cycle
12
Shuffle Counter‡
Shuffle LUT RAM (256 elements only)
12
SHUFFLE_AB
Iteration Counter (B-dimension)§§
Increments every iteration
12
SRCAINDX
SRCBINDX
WRAP_COMB‡‡
Wrap
SRCADDR
Source memory address
C-dimension applicable In Local Max Engine Miode only
Repetition Counter (C-dimension)§§§
Increments after B-iterations are complete
12
SRCCINDX
WRAP_COMB‡‡
Wrap
SRCADDR
Source memory address

§ Starts at SRCA_CIRCSHIFT. If SRCA_CIRCSHIFTWRAP is non-zero, then wraps around at $2^{SRCA\_CIRCSHIFTWRAP}$, or at $3*2^{SRCA\_CIRCSHIFTWRAP}$

§§ Starts at SRCB_CIRCSHIFT. If SRCB_CIRCSHIFTWRAP is non-zero, then wraps around at $2^{SRCB\_CIRCSHIFTWRAP}$, or at $3*2^{SRCA\_CIRCSHIFTWRAP}$

§§§ Starts at SRCC_CIRCSHIFT. If SRCC_CIRCSHIFTWRAP is non-zero, then wraps around at $2^{SRCC\_CIRCSHIFTWRAP}$, or at $3*2^{SRCA\_CIRCSHIFTWRAP}$. Only applicable for Local Maxima Engine.

‡ Starts at SHUFFLE_INDX_START_OFFSET * 16. Increments every clock cycle if engaged in A-dim, or every iteration if engaged in B-dim.

‡‡ Wrap around based on WRAP_COMB register applies after combining A and B dimension address offsets. In local maxima engine mode, the WRAP_COMB register applies after combining A and C dimension address offsets.

below figure. The bottom section of the figure showing C-dimension is only applicable in the case of a Local Max Engine mode of operation, which will be introduced later.

Figure 8: Input Formatter's source memory address generation

3.1.4 Input Formatter – Scaling and Formatting

The input formatter allows the input samples read from the source memory to be scaled and formatted before feeding them as 24-bit complex samples into the core computational unit.

Even though the data read from the source memory is initially 16-bits or 32-bits wide (for each I and Q), the samples expected by the core computational unit are 24-bit complex samples (24-bits each for I and Q). There is a SRCSCAL register which provides scaling options using bit-shift to generate 24-bit samples from the original 16- or 32-bit data (see Figure 9).

For the 16-bit case, the 24-bit sample is generated by padding (8-SRCSCAL) zeros at the LSB and SRCSCAL redundant MSBs. For the 32-bit case, the 24-bit sample is generated by dropping SRCSCAL bits at the LSB and clipping (8-SRCSCAL) bits at the MSB. Note that the register bit SRCSIGNED is used to indicate whether the input samples are signed or unsigned. When this register bit is set, the input samples are treated as signed numbers and hence any extra MSB bits are sign- extended and any clipping of MSB bits takes care of signed saturation. In most cases of interest in part one of this user guide (for example, when performing FFT operation), the input samples would be signed and hence SRCSIGNED should be set (i.e., equal to 1).

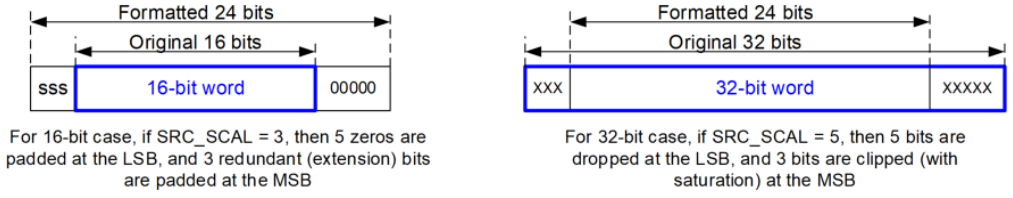

Formatted 24 bits
Original 16 bits
sss
16-bit word
00000
For 16-bit case, if SRC_SCAL = 3, then 5 zeros are padded at the LSB, and 3 redundant (extension) bits are padded at the MSB
Formatted 24 bits
Original 32 bits
xxx
32-bit word
xxxxx
For 32-bit case, if SRC_SCAL = 5, then 5 bits are dropped at the LSB, and 3 bits are clipped (with saturation) at the MSB

Figure 9: Input Formatter Data Scaling

When the input samples are complex (SRCREAL = 0), there is a provision to conjugate the input samples. Setting the register bit SRCCONJ conjugates the input samples before feeding them to the core computational unit. This feature (together with a corresponding DSTCONJ register bit in the output formatter block) enables an IFFT mode from the FFT engine. Note that conjugating the input and output of an FFT block is equivalent to an IFFT function.

There is also provision for swapping I and Q samples read from memory. This can be controlled using SRCIQSWAP register bit. If SRCIQSWAP = 0, then the I sample is located at the LSB bits, and the Q sample is located at the MSB bits.

There are other registers in the input formatter, such as BPM_EN, BPMPATTERNLSB and BPMPATTERNMSB, BPMRATE, and so on, which are beyond the scope of part one of this user's guide and these registers are described in part two. For the immediate purpose of the first part of the user's guide, it is important to note that BPM_EN & unused dimension (C) registers must be kept 0.

3.1.5 Input Formatter – Register Descriptions

Table 2 lists all the registers of the input formatter block.

Table 2: Input Formatter Registers

Register	Width	Parameter Set	Description
SRCADDR	20	Yes	Source start address: This register specifies the starting address of the input samples, for example, it specifies the source memory start address from which input samples have to be fetched by the input formatter. This is a byte-address but only even values are valid. This register covers the entire address space of the eight local memories (8 × 16KB = 128 KB). Only the 17 LSB bits of this register are relevant for TPR12 device, and the 3 MSB bits are reserved for future use. The eight accelerator local memories are contiguous in the memory address space and any of them can act as the source memory (as long as the same memory bank is not configured to be used as destination memory at the same
SRCACNT	12	Yes	Source sample count: This register specifies the number of samples (minus 1) from the source memory to process for every iteration. The sample count is in number of samples, not number of bytes. For example, the sample count can be specified as 255 (SRCACNT = 0x0FF) in a case where a 256-point FFT is required to be performed. Note however that the sample count register does not always match the FFT size. This can happen when zero-padding of input samples is required. For example, a sample count of 192 could be used with an FFT size of 256, in which case, the input formatter will automatically append 64 zeros.
SRCAINDX	20	Yes	Source sample index increment: This register specifies the number of bytes separating successive samples in the source memory. For example, a value of SRCAINDX = 16 means that successive samples are separated by 16 bytes in memory. Only even values are allowed for this register. The maximum value allowed for this register is 65534.
BCNT	12	Yes	Number of iterations: This register specifies the number of times (minus 1) the processing should be repeated. This register can be used to process the four RX chains back-to-back – for example, a value of BCNT = 3 means that the processing (say first dimension FFT processing) is repeated four times. Note the distinction between the NUMLOOPS register of the state machine block and the BCNT register of the input formatter block. The NUMLOOPS register specifies how many times the state machine loops through all the configured parameter sets (with each time possibly awaiting a trigger), whereas the register BCNT specifies how many times the input formatter and the computational processing of the accelerator is iterated back-to-back for the current parameter set (without any intermediate triggers). Non-zero BCNT should be used only with non-zero ACNT.

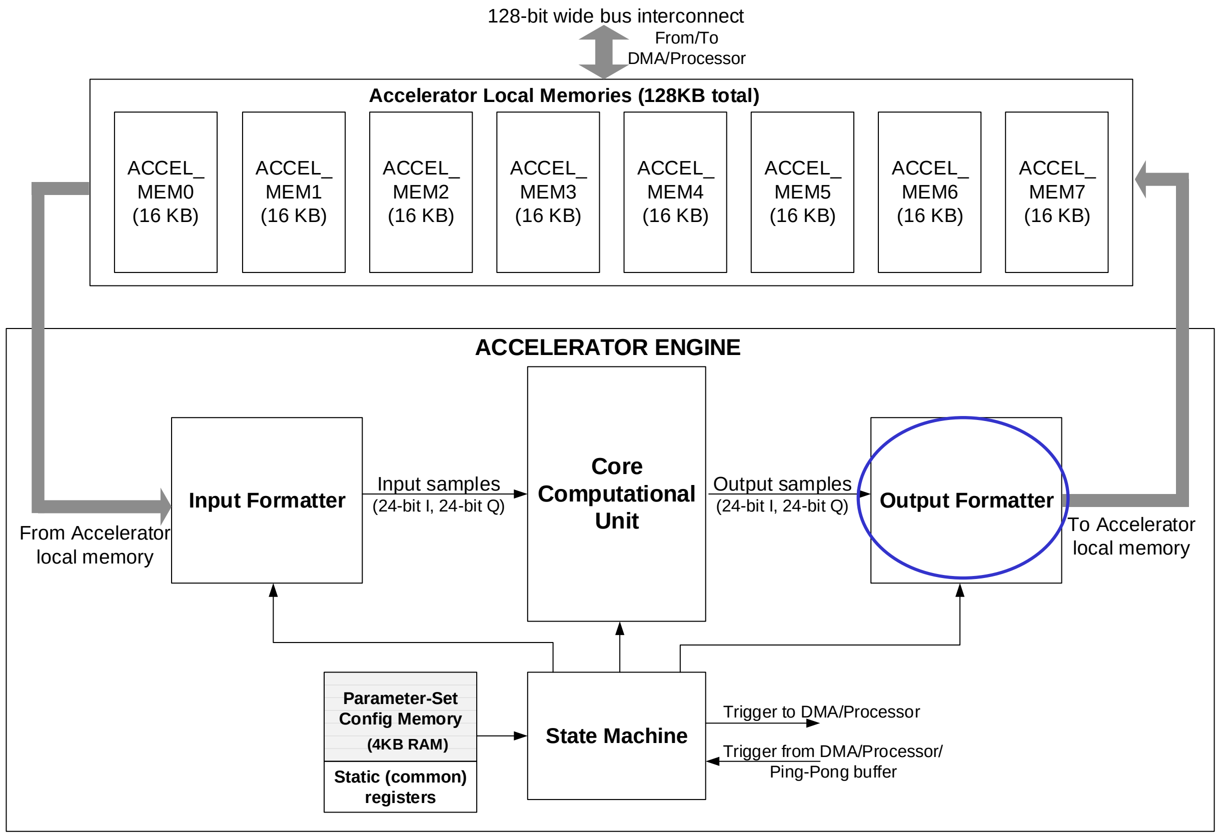
SRCBINDX	20	Yes	Source offset per iteration: This register specifies the number of bytes separating the starting address of input samples for successive iterations. For example, when using four iterations to process the four RX chains, this register can be used to specify the offset in the starting address between the successive RX chains. Note the distinction that SRCAINDX specifies the number of bytes separating successive samples for a particular iteration, whereas SRCBINDX specifies the number of bytes separating the starting address of the first sample for successive iterations. Only even values are allowed for this register. The maximum
CCNT	12	Yes	C-dimension count – i.e., Number of times (minus 1) that the B dimension iterations are performed. This register specifies the C-dimension count. C-dimension is applicable only to Local Maxima Engine. Non-zero CCNT should be used only with non-zero BCNT.
SRCCINDX	20	Yes	Source offset in C-dimension: This register specifies the number of bytes separating the starting address of input samples for successive sets of B-dimension iterations. Only even values are allowed for this register. Refer to the description section for details on the address generation
SRCREAL	1	Yes	Complex or Real Input: This register-bit specifies whether the input samples are real or complex. A value of SRCREAL = 0 implies complex input and a value of SRCREAL = 1 implies real input. When real input is selected, the input formatter block automatically feeds zero for the imaginary part into the core computational unit.
SRCA_CIRCSHIFT	12	Yes	Start index for circular shift in A-dimension. Input Formatter reads samples from the source memory with this start offset to the sample index.
SRCB_CIRCSHIFT	12	Yes	Start index for circular shift in B-dimension (similar to A-dimension circular shift).
SRCA_CIRCSHIFTWRAP	4	Yes	Circular shift wrap-around point for A-dimension: This register, when set to a non-zero value, specifies the wrap-around point for A-dimension sample counter. If SRC_CIRCSHIFTWRAP3X (A-dimension bit) is set to 0, the A-dimension sample index counter wraps around (i.e., resets to 0) when the counter exceeds (2^SRCA_CIRCSHIFTWRAP-1). When that bit is 1, the A-dimension sample index counter wraps around when the counter exceeds (3*2^SRCA_CIRCSHIFTWRAP-1).
SRCB_CIRCSHIFTWRAP	4	Yes	Circular shift wrap-around point for B-dimension: This register, when set to a non-zero value, specifies the wrap-around point for B-dimension iteration counter. Functionality is similar to SRCA_CIRCSHIFTWRAP.

SRC_CIRCSHIFTWRAP3X	2	Yes	3X enable for circular shift wrap-around: This register is used in conjunction with SRCA_CIRCSHIFTWRAP and SRCB_CIRCSHIFTWRAP to specify the wrap-around point when circular shift is used. Bit 0 of this register corresponds to A-dimension, and bit 1 corresponds to B-dimension. Refer description of SRCA_CIRCSHIFTWRAP for details.
SHUFFLE_AB	2	Yes	Shuffled addressing: If this register is set to 0b01, the Shuffle LUT is used for sample index re-ordering in A-dimension. If it is set to 0b10, it is used in B-dimension. If this register is set to 0b00, the Shuffle LUT is bypassed / ignored.
SHUFFLE LUT RAM[256]	12	No	This RAM stores the Shuffle LUT contents
SHUFFLE_INDX_START_OFFSET	4	Yes	Start index for the Shuffle LUT: This register together with 4 zeros padded at the LSB becomes the 8-bit starting index for the 256-element Shuffle LUT (RAM).
WRAP_COMB	20	*Yes*	Combined wrap around for A and B dimension: This is applicable in Shuffled addressing mode. The combined A & B dimension based address offset is wrapped around this number.
SRC16b32b	1	Yes	16-bit or 32-bit input word alignment: This register-bit specifies whether the input samples fetched from source memory are to be read as 16-bits or 32-bits wide. A value of SRC16b32b = 0 implies that the input samples are 16-bits wide each (in case of complex input, real and imaginary parts are each 16 bits wide). A value of SRC16b32b = 1 implies that the input samples are 32-bits wide each.
SRCSIGNED	1	Yes	Input sign-extension mode: This register-bit, when set, specifies that the input samples are signed numbers and hence, sign-extension or signed-saturation at the MSB is required when converting 16- bit or 32-bit input words to the 24-bit wide samples to be fed into the core computational
SRCCONJ	1	Yes	Input conjugation: This register-bit specifies whether the input samples should be conjugated before feeding them into the core computational unit. If SRCCONJ is set, then the input samples are conjugated. Setting this register-bit only makes sense if the samples are complex numbers (for example, SRCREAL = 0). This register, together with its counterpart in the output formatter block, enable an IFFT mode for the FFT engine. Note that conjugating the input and output of an FFT block is equivalent to an IFFT function.

SRCSCAL	8	Yes	Input scaling: This register specifies a programmable scaling using bit-shift, when converting the 16-bit or 32-bit wide input data to 24-bit wide samples before feeding into the core computational unit. See Figure 8 and its description for more details regarding this register.
SRCIQSWAP	1	Yes	Swap the I & Q samples drawn from memory. LSB bits drawn from memory is used as I, and the MSB bits are used a Q of input
IP_FORMATTER_CLIP_STATUS	1	No	Read-only register that indicates clip status for input formatter (during scaling).
CLR_CLIP_MISC	1	No	Below clip status read-registers will be cleared upon writing to this self-clearing register bit: channel_comb_clip_status, dc_acc_clip_status, dc_est_clip_status, intf_stats_mag_accumulator_clip_status, intf_stats_magdiff_accumulator_clip_status, intf_stats_thresh_mag_clip_status, intf_stats_thresh_magdiff_clip_status, twid_incr_delta_frac_clip_status, ip_formatter_clip_status,
BPM_PATTERN_0, BPMPATTERN_1, … BPMPATTERN_7	–	–	Described in part two of this user's guide. For the immediate purposes relevant to part one of this user's guide, all of these registers must be kept as 0.
BPMRATE	–	–	Described in part two of this user's guide.
BPMPHASE	–	–	Described in part two of this user's guide.

4 Accelerator Engine – Output Formatter

This section describes the output formatter block present in the accelerator engine (see Figure 10).



128-bit wide bus interconnect
From/To DMA/Processor
Accelerator Local Memories (128KB total)
ACCEL_ MEM0 (16 KB)
ACCEL_ MEM1 (16 KB)
ACCEL_ MEM2 (16 KB)
ACCEL_ MEM3 (16 KB)
ACCEL_ MEM4 (16 KB)
ACCEL_ MEM5 (16 KB)
ACCEL_ MEM6 (16 KB)
ACCEL_ MEM7 (16 KB)
ACCELERATOR ENGINE
From Accelerator local memory
Input Formatter
Input samples (24-bit I, 24-bit Q)
Core Computational Unit
Output samples (24-bit I, 24-bit Q)
Output Formatter
To Accelerator local memory
Parameter-Set Config Memory (4KB RAM)
Static (common) registers
State Machine
Trigger to DMA/Processor
Trigger from DMA/Processor/ Ping-Pong buffer

Figure 10: Output Formatter

4.1 *Output Formatter*

The output formatter is used to format and write the data coming out of the core computational unit into the accelerator local memory. Similar to the input formatter block discussed in the previous section, the output formatter block also provides various capabilities to format and write the samples written to the local memory – especially, various multidimensional access patterns (for example, transpose writes), 16- bit or 32-bit aligned word writes, scaling using bit-shifts to generate 16-bit or 32-bit words from 24-bit wide samples, real versus complex output write, and more.

4.1.1 Output Formatter – Operation

The output formatter block is responsible for storing the samples coming out of the core computation unit into the accelerator local memory (see Figure 10). As mentioned in the previous section, the data flow from the input formatter, through the core computational unit, to the output formatter, is designed to sustain a steady-state throughput of one complex sample per clock cycle. Thus, typically, the output formatter accepts one sample (24-bit I and 24-bit Q) from the core computational unit every clock cycle and writes it to the accelerator local memory. Just like the input formatter, the output formatter also supports lot of flexibility in how the samples are formatted and written into the memory.

The memory into which the output formatter writes the data is referred to as destination memory. Note that any of the four accelerator local memories can be the destination memory, with the important restriction that the source memory cannot be same as the destination memory. In other words, each of the eight 16KB memory banks can either function as source memory, or as destination memory at any time (for example, in any given parameter set).

4.1.2 Output Formatter – 2-D Indexed Addressing for Destination Memory Access

The parameter-set register DSTADDR specifies the start address at which the output samples must be written into the accelerator local memory. Similar to the SRCADDR register of the input formatter, the DSTADDR register of the output formatter is a byte-address and a value of 0x0 corresponds to the first memory location of ACCEL_MEM0 memory. The DSTADDR register maps to the entire 128KB address space of the eight accelerator local memories (each 16KB). As mentioned in the previous paragraph, in a given parameter set, SRCADDR and DSTADDR cannot be configured such that the input samples being fetched and the output samples being written out are accessing the same memory bank.

Even though the core computational unit produces a 24-bit complex output stream, this output data can be written to the memory as either 16-bit wide samples or 32-bit wide samples. Also, they can be written out as complex samples or real samples (for example, drop imaginary part – applicable when performing log- magnitude computation). These two aspects are configured using register bits DST16b32b and DSTREAL. See Table 3 for a description of these and other registers pertaining to the output formatter block. As an example, if DST16b32b = 0 and DSTREAL = 0, then the output samples are written to the memory as 16- bit complex samples (16-bit I and 16-bit Q), shown in Figure 11.

Similar to the input formatter block, the output formatter block also supports flexible patterns to write multidimensional data to the destination memory and this makes it convenient when the data corresponding to multiple RX channels must be interleaved, or when performing multidimensional (FFT) processing. This feature is facilitated through the DSTAINDX, DSTACNT, DSTBINDX, and BCNT registers, which are part of each parameter-set configuration.

The register DSTAINDX specifies how many bytes separate successive samples to be written to the destination memory and the register DSTACNT specifies how many samples must be written per iteration. Note that DSTACNT can be different from SRCACNT – this is useful when only a subset of the output samples need to be stored in the output memory (for example, if some FFT output bins must be discarded). The register DSTBINDX specifies how many bytes separate the start of output samples for successive iterations and BCNT specifies the number of iterations. The BCNT register is common for input formatter and output formatter. These registers can be better understood using the example given in Figure 11.

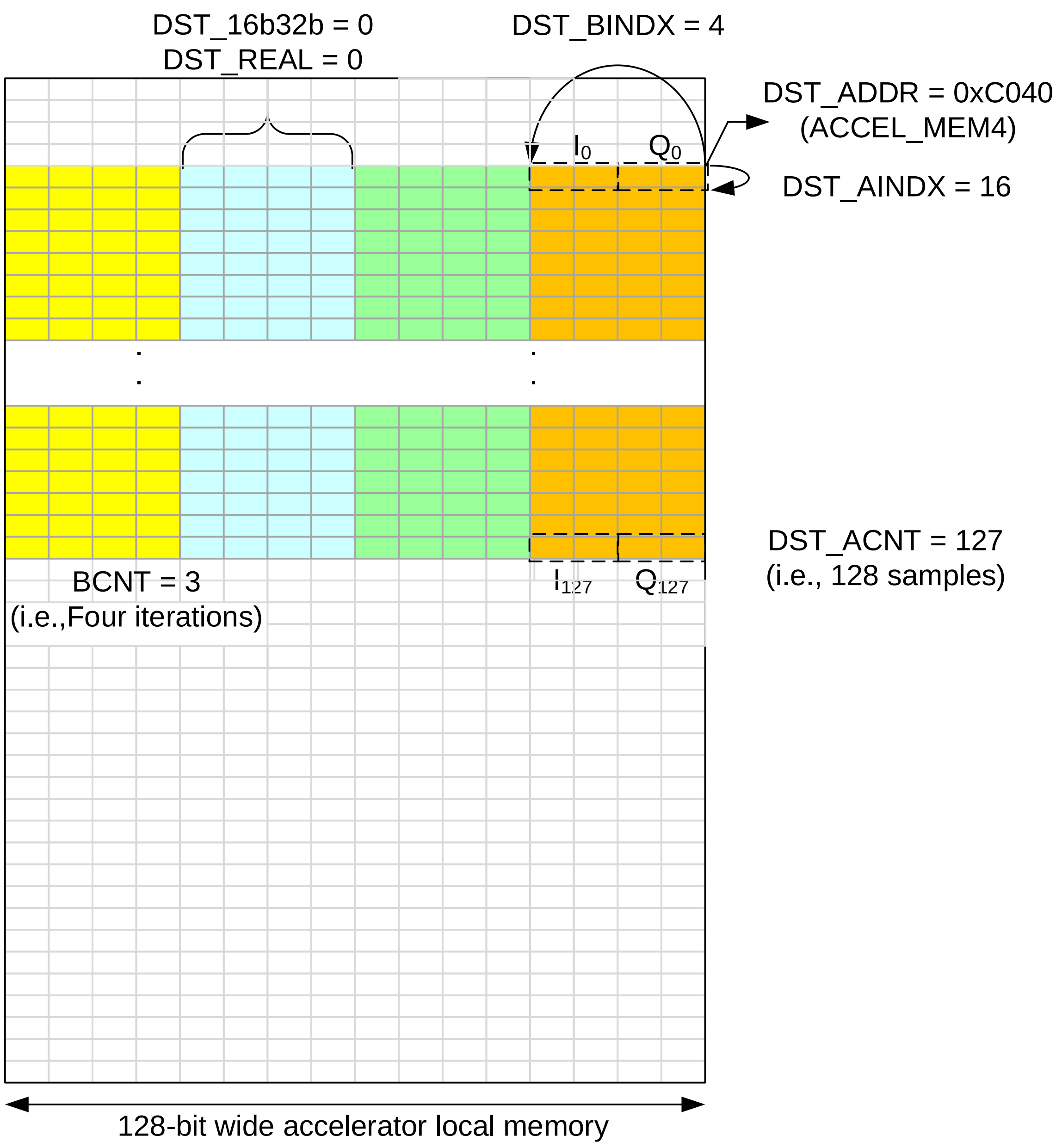


Figure 11: Output Formatter Destination Memory Access Pattern (Example)

In the example shown in Figure 11, the output data consists of complex data (16-bit I and 16-bit Q) that is written to ACCEL_MEM3. The output data consists of four sets of 128 samples each (say, corresponding to FFT output of four RX antennas) and these are shown in four different colors. Each sample occupies 4 bytes and the samples are written to the output memory at a specific start address inside ACCEL_MEM3, as shown in Figure 11. The samples for the four RX antennas are written to the memory in an interleaved manner. Thus, for this example, a value of DSTADDR = 0xC040, DSTAINDX = 16, DSTACNT = 127, and DSTBINDX = 4 are used. The register BCNT (common for input formatter and output formatter) is configured with a value of 3, corresponding to the four iterations required (for the four RX antennas). In steady state, for each clock cycle, the output formatter accepts one complex sample from the core computational unit and writes it into the memory as per the 2-D indexed addressing pattern programmed.

The register DSTACNT, which corresponds to the number of samples written to the destination memory for each iteration does not need to be equal to SRCACNT. This is useful in cases where some of the output samples (for example, some FFT bins at the end) can be dropped and do not need to be written into the destination memory. Another register, DST_SKIP_INIT is also available, which can be used to skip some samples in the beginning as well. The number of samples written to the destination memory for each iteration is equal to (DSTACNT + 1) –DST_SKIP_INIT.

Note that when performing FFT operations, internally the core computational unit sends out FFT output data in bit-reversed addressing order, but this is automatically handled in the output formatter, such that when the FFT output samples are written into the destination memory, they are written out in the correct normal order. Therefore, no special procedure is required on the part of the main processor to read the FFT output samples in the right sequence.

Similar to the input formatter, the output formatter can write data into only one memory row (128-bit memory location) in a clock cycle. Therefore DSTADDR, DSTAINDX, and DSTBINDX should be programmed such that no sample needs to be partially written in one memory row and the next (e.g. if DST16b32b=1). Also, these address parameters should be restricted to even values.

4.1.3 Output Formatter – Scaling and Formatting

The output formatter allows the 24-bit output samples from the core computational unit to be scaled and formatted before writing them to the destination memory as 16-bit or 32-bit words. There is a DSTSCAL register which provides scaling options using bit-shift, to take the 24-bit samples and convert them to 16-bit or 32-bit data.

For the 16-bit case (Figure 12), the 24-bit sample (24-bits for each I and Q) is converted to 16-bit word by dropping DSTSCAL bits at the LSB and by clipping with saturation (8-DSTSCAL) bits at the MSB. For the 32-bit case, the 24-bit sample is padded with DSTSCAL extra bits at the MSB and with (8- DSTSCAL) extra zeros at the LSB. Note that the register bit DSTSIGNED is used to indicate whether the output samples are signed or unsigned. When this register bit is set, the output samples are treated as signed numbers and therefore any extra MSB bits are sign-extended and any clipping of MSB bits handles signed saturation. In most cases of interest in part one of this user's guide (for example, when performing FFT operation), the output samples would be signed and therefore DSTSIGNED should be set (for example, equal to 1). However, if the log-magnitude operation in the core computational unit is enabled, then the output samples are unsigned and therefore DSTSIGNED is cleared (for example, equal to zero).

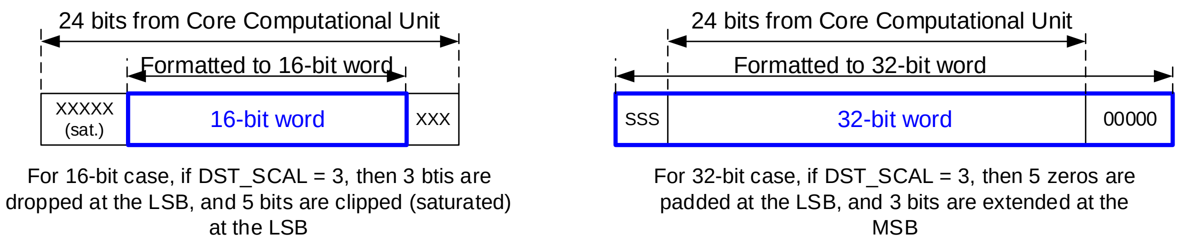

24 bits from Core Computational Unit
Formatted to 16-bit word
XXXXX (sat.)
16-bit word
XXX
For 16-bit case, if DST_SCAL = 3, then 3 btis are dropped at the LSB, and 5 bits are clipped (saturated) at the LSB
24 bits from Core Computational Unit
Formatted to 32-bit word
SSS
32-bit word
00000
For 32-bit case, if DST_SCAL = 3, then 5 zeros are padded at the LSB, and 3 bits are extended at the MSB

Figure 12: Output Formatter Data Scaling

When the output samples are complex (for example, DSTREAL = 0), there is a provision to conjugate the output samples. Setting the register bit DSTCONJ conjugates the output samples before writing them to the destination memory. This feature (together with a corresponding SRCCONJ register bit in the input formatter block) enables an IFFT mode from the FFT engine.

In addition, there is provision for swapping I and Q samples written into the destination memory. This is controlled using DSTIQSWAP register bit.

4.1.4 Output Formatter – Register Descriptions

Table 3 lists all the registers of the output formatter block.

Table 3: Output Formatter Registers

Register	Width	Parameter Set	Description
DSTADDR	20	Yes	Destination start address: This register specifies the starting address of the output samples, for example, it specifies the destination memory start address at which the output samples have to be written by the output formatter. This is a byte-address but only even values are valid. This register covers the entire address space of the eight local memories (8 × 16KB = 128 KB). Only the 17 LSB bits of this register are relevant for TPR12 device, and the 3 MSB bits are reserved for future use. The eight accelerator local memories are contiguous in the memory address space and any of them can act as the destination memory (as long as the same memory bank is not configured to be used as source memory at the same time).
DSTACNT	12	Yes	Destination sample count: This register specifies the number of samples (minus 1) to be written to the destination memory for every iteration. The sample count is in number of samples, not number of bytes. For example, the sample count can be specified as 191 (DSTACNT = 0x0BF) in a case where 192 samples must be written. Note that the DSTACNT register can be different from SRCACNT or even the FFT size. This is useful when only a part of the FFT bins must be written to memory and the remaining (far-end FFT bins) can be discarded. This register description is true when the DST_SKIP_INIT register value is zero (see further for more information related to DST_SKIP_INIT).
DSTAINDX	20	Yes	Destination sample index increment: This register specifies the number of bytes separating successive samples to be written to the destination memory. For example, a value of DSTAINDX = 16 means that successive samples written to the destination memory should be separated by 16 bytes. Only even values are allowed for this register. The maximum value allowed for this register is 65534.

DSTBINDX	20	Yes	Destination offset per iteration: This register specifies the number of bytes separating the starting address of output samples for successive iterations. For example, when using four iterations to process four RX chains, this register can be used to specify the offset in the starting address between the successive RX chains. Note the distinction that DSTAINDX specifies the number of bytes separating successive samples for a particular iteration, whereas SRCBINDX specifies the number of bytes separating the starting address of the first sample for successive iterations. Only even values are allowed for this register. The maximum value allowed for this register is 65534.
DST_SKIP_ INIT	10	Yes	Destination skip sample count: This register specifies how many output samples should be skipped in the beginning, before starting to write to the destination memory. This is useful if only a certain part of the FFT output (skipping the first several bins) need to be stored in memory. The total number of samples written to destination memory is equal to DSTACNT+1-DST_SKIP_INIT.
DSTREAL	1	Yes	Complex or real output: This register-bit specifies whether the output samples are real or complex. A value of DSTREAL = 0 implies complex output and a value of DSTREAL = 1 implies real output. When real output is selected, the output formatter block automatically stores only the real part into the destination memory. This is useful when the core computational unit is configured to output magnitude or log-magnitude values.
DST16b32b	1	Yes	16-bit or 32-bit output word alignment: This register-bit specifies whether the output samples are to be written as 16-bits or 32- bits wide in the destination memory. A value of DST16b32b = 0 implies that the output samples are to be written as 16-bit words (in case of complex output, real and imaginary parts are each 16 bits wide). A value of DST16b32b = 1 implies that the output samples are 32-bits wide each.
DSTSIGNED	1	Yes	Output sign-extension mode: This register-bit, when set, specifies that the output samples are signed numbers and therefore, sign-extension or signed-saturation at the MSB is required when converting the 24-bit wide samples coming from the core computational unit into 16-bit or 32-bit output words to be written to the destination memory.
DSTCONJ	1	Yes	Output conjugation: This register-bit specifies whether the output samples must be conjugated before writing them into the destination memory. If DSTCONJ is set, then the output samples are conjugated. Setting this register-bit only makes sense if the samples are complex numbers (for example, DSTREAL = 0). This register, together with its counterpart in the output formatter block, enables an IFFT mode for the FFT engine.

DSTSCAL	8	Yes	Output scaling: This register specifies a programmable scaling using bit-shift, when converting the 24-bit samples coming from the core computational unit into 16-bit or 32-bit wide words to be written to the destination memory. See Figure 11and its description for more details regarding this register.
DSTIQSWAP	1	Yes	IQ Swapping : Swap the I & Q samples written out to memory.
OP_FORMATTER_CLIP_STATUS	1	No	Read-only register that indicates clip status for output formatter (during scaling).
DMEM0, DMEM1… DMEM7 (MEM_ACCESS_ERR_STATUS)	1	No	Memory access error: This set of 8 1-bit read-only registers indicates if there is a memory access error caused by incorrect configuration or usage of the accelerator, where both the DMA and the accelerator are attempting to access the same 16KB memory at the same time. The composite 8- bit register indicates the error status for the 8 16KB memories (DMEM0 bit corresponds to ACCEL_MEM0).

5 Accelerator Engine – Core Computational Unit

This section describes the core computational unit present in the accelerator engine (see Figure 13).

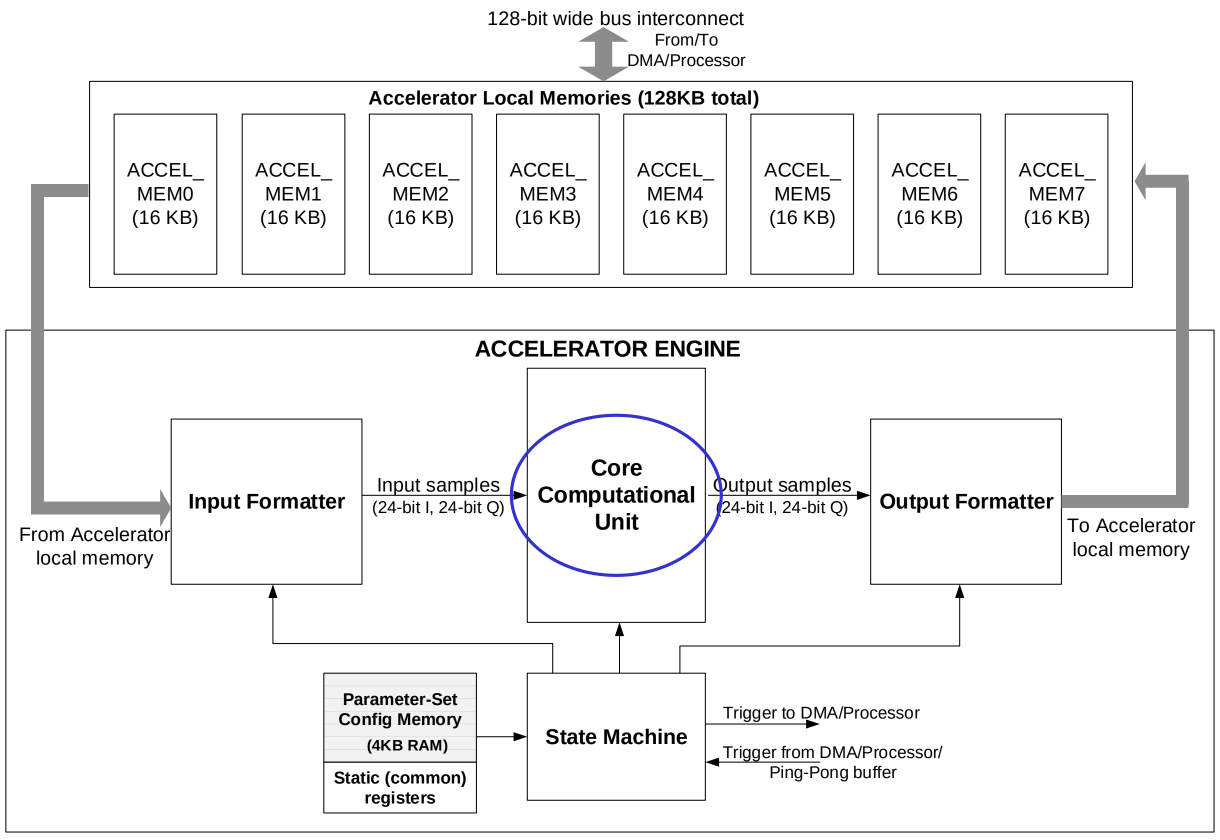

128-bit wide bus interconnect
From/To DMA/Processor
Accelerator Local Memories (128KB total)
ACCEL_ MEM0 (16 KB)
ACCEL_ MEM1 (16 KB)
ACCEL_ MEM2 (16 KB)
ACCEL_ MEM3 (16 KB)
ACCEL_ MEM4 (16 KB)
ACCEL_ MEM5 (16 KB)
ACCEL_ MEM6 (16 KB)
ACCEL_ MEM7 (16 KB)
ACCELERATOR ENGINE
Input Formatter
Input samples (24-bit I, 24-bit Q)
Core Computational Unit
Output samples (24-bit I, 24-bit Q)
Output Formatter
From Accelerator local memory
To Accelerator local memory
Parameter-Set Config Memory (4KB RAM)
Static (common) registers
State Machine
Trigger to DMA/Processor
Trigger from DMA/Processor/ Ping-Pong buffer

Figure 13: Core Computational Unit

5.1 *Core Computational Unit*

The core computational unit performs the mathematical operations required for the key functions, such as FFT, log-magnitude, CFAR detection and so on. The core computational unit accepts a streaming 24-bit complex input (24 bits for each I and Q) from the input formatter block and it outputs a streaming 24-bit complex output (24 bits for each I and Q) to the output formatter block.

Figure 13 shows the block diagram of the core computational unit. The core computational unit has four main computation engines, namely:

- FFT Engine: Performs Pre-processing, Windowing, FFT and Log-magnitude.
- CFAR Engine: Performs CFAR detection using CFAR-CA or CFAR-OS method.
- Local Maxima Engine: Performs threshold and local maxima based peak identification.
- Compression Engine: Used for compression and decompression of radar data.

Only one of these four engines can be operational at any given instant. However, in separate parameter sets, different engines can be configured and used, so that multiple parameter sets executing one after another can accomplish a sequence of computational operations as desired. The register ACCEL_MODE controls which engine gets used in a given parameter set.

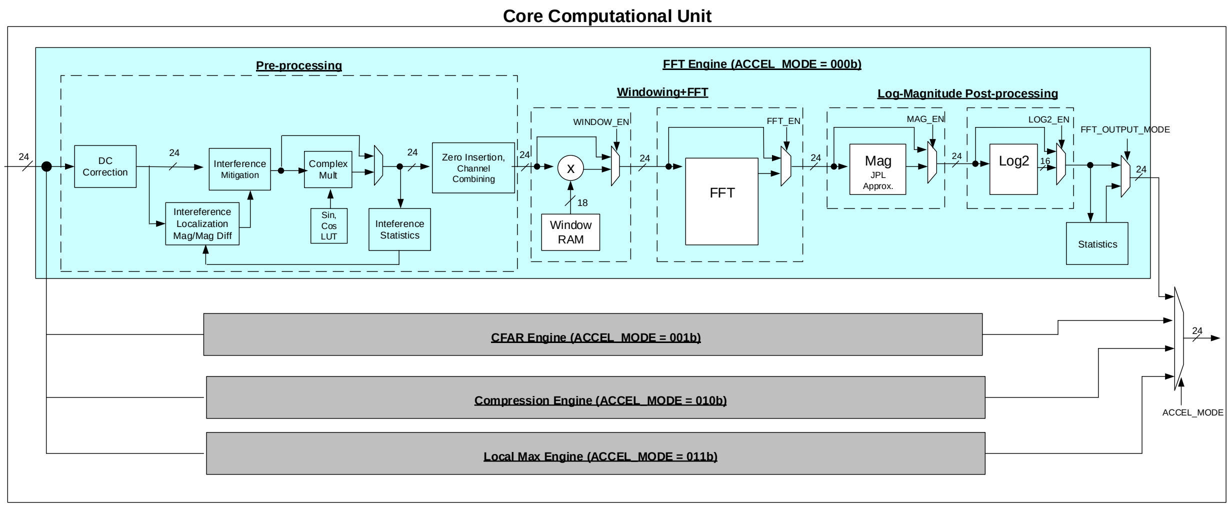


Figure 14: Core Computational Unit Block Diagram

For the purpose of part one of the user's guide, only the FFT Engine path is described. Specifically, the windowing, FFT, and log-magnitude operations are covered in this document. The greyed-out blocks in Figure 14, namely the Pre-processing, Statistics, and CFAR Engine, are covered in part two of the user's guide and can be ignored for the present purpose.

5.1.1 Core Computational Unit – FFT Engine – Operation

The core computational unit operates on the streaming input of samples coming from the input formatter block, and in general outputs a stream of samples (after an initial latency in some cases) to the output formatter block. In general, at steady-state, one input sample is processed and one output sample is produced every clock.

The FFT Engine in the core computational unit has the ability to perform pre-processing, windowing, FFT, and log-magnitude computations. Each of these computational subblocks operate on a streaming input and produce a streaming output at the throughput of one sample per clock. These computational subblocks are stitched together one after the other in a series, as shown in Figure 14. This architecture allows multiple operations to be done in a streaming manner (for example, windowing and FFT can be done together), while at the same time, providing the user flexibility to choose one operation at a time.

The parameter-set registers WINDOW_EN, FFT_EN, ABS_EN, and LOG2_EN control the multiplexers (see Figure 14), which decide what operations are performed on the input samples for that parameter set.

Note that for the purpose of part one of the user's guide, the registers ACCEL_MODE and FFTOUT_MODE must be kept at zero. The purpose of these registers is covered in part two.

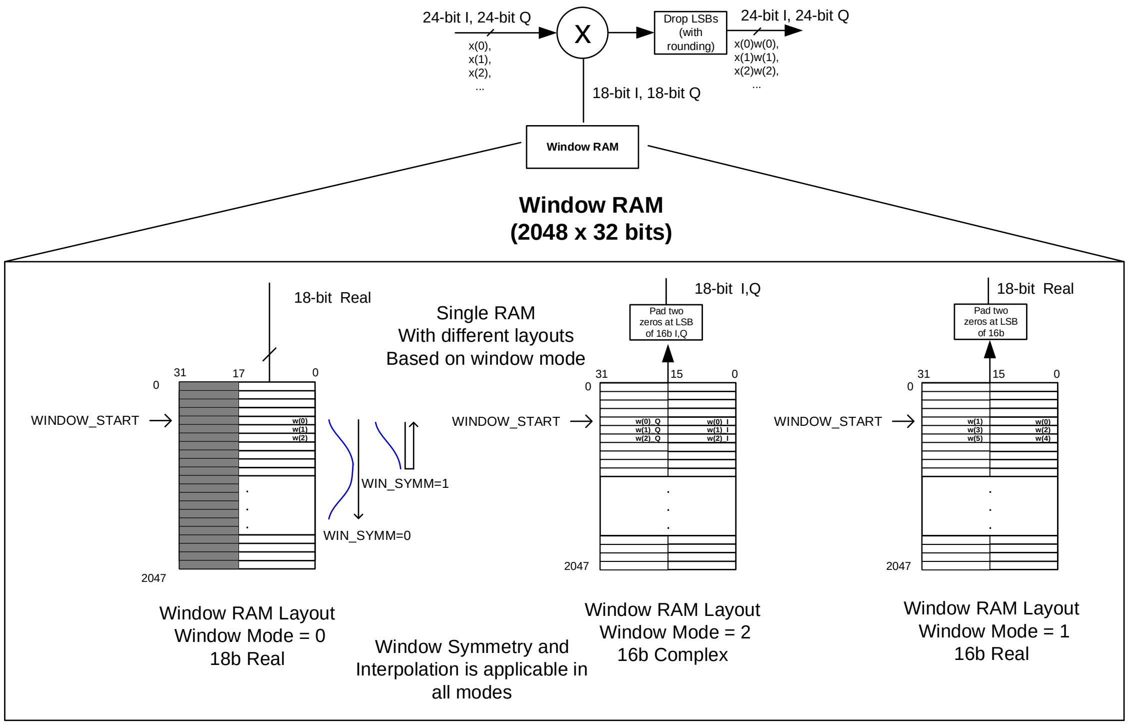
5.1.2 Core Computational Unit – FFT Engine – Windowing

The incoming samples from the input formatter to the core computational unit are passed through the (optional) windowing operation (see Figure 15). Windowing operation is often required prior to performing FFT, to mitigate the sinc roll-off leakage from one strong FFT bin to the adjacent bins.

As the incoming samples from the input formatter stream in, each sample is multiplied by the appropriate window coefficient read from a Window RAM. Because the incoming samples are complex 24-bits wide (24-bits for each I and Q), the windowing operation involves multiplying the 24-bit I and 24-bit Q of the incoming sample with the window coefficient (see Figure 15). The output of this multiplication is rounded back to 24-bit I and 24-bit Q by dropping excess LSBs. Note that windowing can be enabled by setting the register bit WINDOW_EN to 1.

The window RAM can hold up to 2048 32-bit words. The window coefficients can be stored in these words in one of the following three formats:

1.) **18-bit real** coefficients: If WINDOW_MODE = 0b00, the window coefficients are assumed to be 18-bit signed real values. Up to 2048 real coefficients can be stored in the window RAM in this mode (one 32-bit word in the RAM stores one 18-bit coefficient).
2.) **16-bit real** coefficients: If WINDOW_MODE = 0b01, the window coefficients are assumed to be 16-bit signed real values. Up to 4096 real coefficients can be stored in the window RAM in this mode (one 32-bit word in the RAM stores two successive 16-bit coefficients: the 16LSBs store coefficients 0, 2, 4, etc. and 16MSBs store coefficients 1, 3, 5, etc.).
3.) **16-bit complex coefficients**: If WINDOW_MODE = 0b10, the window coefficients are assumed to be 16-bit I and 16-bit Q complex values. Up to 2048 complex coefficients can be stored in the window RAM in this mode (one 32-bit word in the RAM stores the real part of the coefficient in the 16 LSBs of the word and the imaginary part in the 16 MSBs of the word).



24-bit I, 24-bit Q
x(0), x(1), x(2), ...
X
Drop LSBs (with rounding)
24-bit I, 24-bit Q
x(0)w(0), x(1)w(1), x(2)w(2), ...
18-bit I, 18-bit Q
Window RAM
Window RAM (2048 x 32 bits)
18-bit Real
Single RAM With different layouts Based on window mode
18-bit I,Q
Pad two zeros at LSB of 16b I,Q
18-bit Real
Pad two zeros at LSB of 16b
WINDOW_START
31
17
15
0
2047
w(0)
w(1)
w(2)
w(0) Q
w(1) Q
w(2) Q
w(0) I
w(1) I
w(2) I
w(1)
w(3)
w(5)
w(0)
w(2)
w(4)
WIN_SYMM=1
WIN_SYMM=0
Window RAM Layout Window Mode = 0 18b Real
Window Symmetry and Interpolation is applicable in all modes
Window RAM Layout Window Mode = 2 16b Complex
Window RAM Layout Window Mode = 1 16b Real

Figure 15: Window RAM Layout for 18b Real, 16b Complex and 16b Real modes

The start location (32-bit word index) in the window RAM is programmed in a 11-bit register WINDOW_START as part of the parameter set, so that the windowing computation can pick the appropriate window coefficients starting from that index. For each incoming sample, the index keeps incrementing, so that each successive sample is multiplied by the successive window coefficient. At the end of each iteration (for example, when SRCACNT number of samples have been processed), the index resets back to the starting coefficient index programmed for the parameter set, so that the next iteration can be performed. At the end of all the iterations of the current parameter set, the next parameter set can use a different window if desired. For example, when performing second- and third-dimension FFTs one after another (in two parameter sets), the window functions for both these FFTs can be pre-stored in the Window RAM and appropriate start index can be provided for each of the FFT operation dimensions.

If the window function is symmetric, the user may store only one half of the window coefficients in the Window RAM. The register bit WINSYMM, when set, indicates that after SRCACNT / 2 samples (or, if SRCACNT is odd, (SRCACNT + 1) / 2 samples) are processed, the window coefficients read-indexing must be reversed, so that the same set of coefficients used for the first SRCACNT / 2 samples are reused in the reverse order for the next SRCACNT / 2 samples. (See Figure 15). If SRCACNT is odd, then the last window coefficient is read only once, when the direction is reversed. If SRCACNT is even, then the last window coefficients is read twice, when the direction is reversed. In the dynamic window mode, the coefficients are read from the corresponding bank only.

The output of the windowing computation is 24-bit I and 24-bit Q, which is streamed into the FFT subblock.

5.1.3 Core Computational Unit – FFT Engine – FFT

The FFT subblock performs FFT on the incoming 24-bit I and 24-bit Q data stream. The FFT size is programmable, in the range, 2 to 2048. FFTs of length 2^N for N = 0 to 11, and $3x2^N$ for N = 0 to 9 are supported. Advanced features such as direct computation of two-dimensional FFTs, as well as an FFT *stitching* feature that realizes FFTs of larger lengths using a two-step process, are also supported. But their description is deferred to the part two of the user's guide.

The lowest FFT size of 2 is mostly useful as a *complex add-subtract* feature or while using the *FFT* stitching feature. FFT sizes of 4, 8, 16, and 32 can be used for third dimension (angle estimation) FFT.

The FFT operation can be enabled or disabled by using the register bit FFT_EN. When enabled, the FFT subblock computes the FFT of the input data stream and produces a 24-bit I and 24-bit Q output stream. This output stream is initially in bit-reversed order, but the output formatter handles appropriately writing the output to the destination memory in the correct order.

The FFT implementation comprises a series of butterfly stages. Depending on the FFT size needed, an appropriate number of butterfly stages are employed. The FFT size is programmed using the registers, FFTSIZE and FFTSIZE_3X_EN.

For power-of-2 FFT sizes, the register bit FFTSIZE_3X_EN should be kept 0. The FFT size is configured using the register FFTSIZE and the actual FFT size in this case is 2^FFTSIZE. When an FFT size of the form (3 * power-of-2) is needed, then the register bit FFTSIZE_3X_EN should be set to 1. In this case, the actual FFT size is 3 * 2^FFTSIZE.

For example, if FFTSIZE_3X_EN is 0, then FFTSIZE = 5 means 32-point FFT, FFTSIZE = 7 means 128-point FFT, and so on. In this case, the FFT is realized using a series of FFTSIZE number of radix-2 butterfly stages. On the other hand, if FFTSIZE_3X_EN is set to 1, then FFTSIZE = 5 means a 96-point FFT, FFTSIZE = 7 means 384-point FFT. In these cases, an additional radix 3 butterfly stage is engaged before feeding to the original series of radix-2 butterfly stages.

Note that the FFT size must be equal to or larger than SRCACNT, and the input formatter block automatically zero-pads extra samples to account for the difference between FFT size and SRCACNT.

5.1.4 Core Computational Unit – FFT Engine – FFT – Zero Padding

The FFT engine has provision for *zero padding*, which is important when performing FFT of a set of samples whose length doesn't match a supported FFT size. The FFT engine automatically feeds the required number of zeros into the core computational unit, whenever the FFT size (as programmed using the FFTSIZE register, which is described in a later section) does not match the SRCACNT setting.

For example, if the number of input samples read by the input formatter is 56 (for example, SRCACNT = 55) and the FFT size is programmed to be 64 (FFTSIZE = 6, FFTSIZE_3X_EN = 0), then the FFT engine feeds 8 zeros at the end of each iteration, before starting to read the input samples for the next iteration from the source memory. This zero-padding provision enables the core computational unit to perform 64-point FFT with the correct set of zero-padded input samples.

The zero padding is effective only when performing FFT operation in the core computational unit (i.e., when FFT_EN = 1) and not otherwise. Advanced features such as zero insertion (at programmable locations) and channel combining are described in part 2 of this user guide.

5.1.5 Core Computational Unit – FFT Quantization and Speed performance

As is well known, a butterfly stage typically consists of add-subtract and twiddle multiplication operations. At the output of each add-subtract structure, the bit-width would increase by 1 bit (for example, 24-bit input would grow to 25-bit output). To handle this one-bit growth due to add-subtract operation, there is a provision at the output of each butterfly add-subtract stage to scale the result back to 24 bits, by either dividing the output by 2 (round off one LSB) or by saturating one MSB, shown in Figure 16.

The multi-bit register BFLY_SCALING is used to control this divide-by-2 scaling operation at each stage, so that the user has full flexibility to control the signal level through the different butterfly stages. If BFLY_SCALING = 0 for a particular stage, then the 25-bit output is saturated at the MSB to get back to 24 bits. Otherwise, it is convergent-rounded at the LSB to get back to 24 bits. The user can thus control the scaling at each of the butterfly stages. The LSB of this multi-bit register corresponds to the last stage and the MSB of this register corresponds to the first stage. For an FFT size of 64, only the LSB 6 bits are relevant. Similarly, for an FFT size of 3*64, the LSB 6 bits are relevant. Additionally the register, BFLY_SCALING_FFT3X indicates the scaling option for the single radix-3 stage (i.e., it supports removing 0 or 1 or 2 LSBs).

There is a multi-bit read-only register FFTCLIP which indicates whether there was any clipping in any of the butterfly stages. This register is a sticky register that gets set when a clipping event occurs and remains set until it is cleared using the CLR_FFTCLIP register bit. See the register description of FFTCLIP in Table 5.

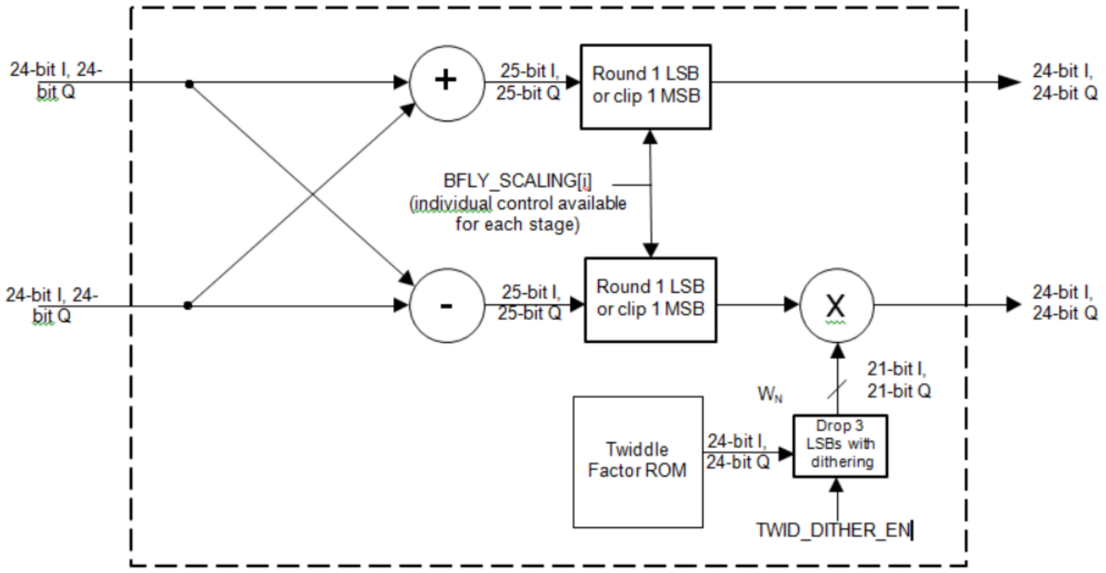


Figure 16: Butterfly Stage Fixed-Point

The twiddle factors are stored as 24-bit I and 24-bit Q coefficients. Prior to twiddle factor multiplication, the coefficients are reduced to 21-bit I and 21-bit Q by dropping three LSBs (with optional dithering). The purpose of dithering is to eliminate any repetitive quantization noise patterns from degrading the SFDR of the FFT. TI recommends that dithering be enabled (DITHERTWIDEN should be set). For dithering, an LFSR is used to generate a random pattern, for which the LFSR seed must be loaded with a non-zero value (see LFSRSEED in the register descriptions).

The SFDR performance of the FFT, with dithering enabled, is better than –140 dBc, as shown in Figure 17.

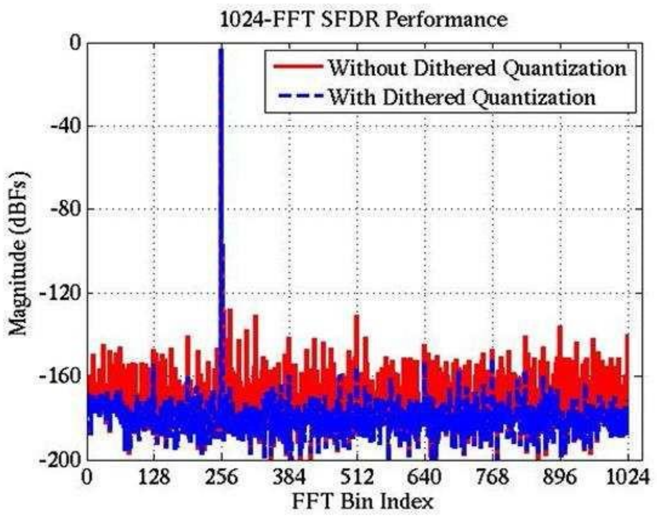


Figure 17: FFT SFDR Performance With and Without Dithering

The architecture of the FFT is such that it can take a streaming input (one sample per clock) and produce a streaming FFT output (one sample per clock), in steady-state. There is an initial latency of approximately *FFT size* number of clocks. This latency only comes into picture once for a given parameter set. Within a parameter set, multiple FFT iterations can be performed back-to-back (for example, for four RX) with no additional latency between iterations.

Because the implementation uses 400 MHz clock in TPR12 device, a 256-point complex FFT for four RX chains would take 256 + 256 × 4 clock cycles to complete, which corresponds to 3.2 µs (plus a few clocks of implementation latencies, which are not accounted here since it is negligible). Table 4 lists the approximate computation time needed for various FFT sizes.

Table 4: FFT Computation Time

Example	FFT Size	Number of Back-to-Back Iterations	Number of Clock Cycles (Initial latency + Computation)	Total Duration
1	256	4	256 + (256 × 4)	3.2 µs
2	128	4	128 + (128 × 4)	1.6 µs
3	8	64	8 + (64 × 8)	1.3 µs

The output of the FFT can be fed to the output formatter or it can be sent to the magnitude/log-magnitude computation subblock.

Note that the FFT is a complex FFT implementation. If the input samples are real-only, then the SRCREAL register bit can be set, such that the imaginary part (Q-part) will be forced to zero by the input formatter block.

5.1.6 Advanced FFT features – FFT Stitching

FFT Engine additionally allows the computation of 4096 & 8192-point FFTs by using FFT stitching. This is done in two-passes. In the first pass, 2 or 4 sets of 2048-point FFT are computed. In the next pass, 2048 sets of 2-point or 4-point FFTs are computed with appropriate twiddle pre-multiplications using the complex multiplier in mode 3 (described in part 2 of user guide) yielding the final 4096 or 8192-point FFT. The relevant register (CMULT_MODE, TWIDINCR, WINDOW_INTERP) settings are indicated in table below A detailed example of FFT stitching along with window interpolation is provided in HWA 2.0 Examples document.

Table 5: FFT Stitching Registers

FFT Size	CMULT_MODE	TWIDINCR	WINDOW_INTERP
4096	3	1	2
8192	3	2	1

5.1.7 Advanced FFT features – 2D FFT

The traditional approach for realizing an M×N point 2-D FFT involves two passes: first pass computing M number of N-point FFTs on the input arrays, and then a second pass computing N number of M-point FFTs on the result of the first pass. HWA2.0 instead supports a direct and faster computation of M×N 2-D FFT, for small M×N sizes (M×N≤2048, N being a power of 2).

The following illustrates the necessary programming for an example with M=16 and N=4. The register FFT_SIZE should be set to $log2(M \times N) = log2(64) = 6$, and the register FFT_SIZE_DIM2 should be set to $log2(N) = log2(4) = 2$. Further the M×N samples of input data should be placed in the memory and fed sequentially to the FFT engine in an interleaved manner as follows. If A is the 2-D 16×4 size input array for the 2-D FFT, then the data placement should ensure that the FFT engine's input order is $A_{0,0}$, $A_{0,1}$, $A_{0,2}$, $A_{0,3}$, $A_{1,0}$, $A_{1,1}$, $A_{1,2}$, $A_{1,3}$, $A_{2,0}$, $A_{2,1}$, …, $A_{15,0}$, $A_{15,1}$, $A_{15,2}$, $A_{15,3}$. The FFT engine computes the 2-D FFT results, and automatically stores them in the destination memory in the right order. The output memory contents would be linear order: $B_{0,0}$, $B_{0,1}$, $B_{0,2}$, $B_{0,3}$, $B_{1,0}$, $B_{1,1}$, $B_{1,2}$, $B_{1,3}$, $B_{2,0}$, $B_{2,1}$, …, $B_{15,0}$, $B_{15,1}$, $B_{15,2}$, $B_{15,3}$, where B represents FFT2D(A).

If zero padding is required in one or both dimensions, it can be achieved through the accelerator's zero insertion feature. It can be done by adding zeros at right locations in the M×N long linear array streaming into the FFT engine (subject to the 256 element limit in zero insertion feature). If 2-D windowing is required before the 2-D FFT, then the 2-D M×N window coefficients can be unrolled into one long linear array and placed in the window RAM. This is illustrated in the below figure for a 32×16 2-D FFT case. If 2-D FFT feature is enabled in any parameter set, it is recommended that the advanced 2D statistics feature be disabled in that parameter set.

This 2-D FFT feature can be used in performing small sized 2-D FFTs, repeatedly over several iterations (e.g. Azimuth × Elevation 2-D FFTs using A dimension, performed for multiple Doppler or Range bins using B dimension). The computation time for B iterations of M×N 2-D FFTs would be (B+1)×(MN) = BMN+MN. In comparison, the traditional two pass approach mentioned earlier, uses (BM+1)×N cycles for first pass (if first passes of all B iterations are performed together) and (BN+1)×M cycles for second pass, in total 2BMN+M+N.

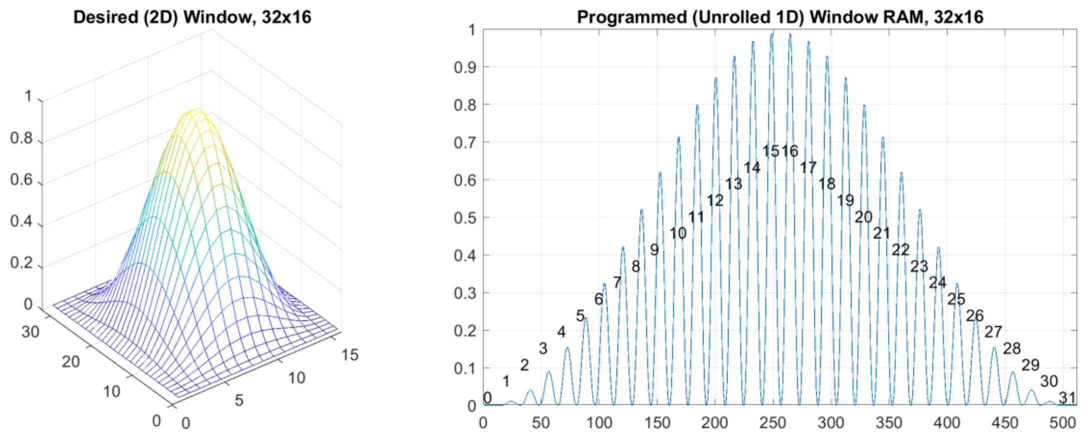


Figure 18: Unrolling 32x16 2D array Window coefficients to a 512 1D vector for placement in window RAM

5.1.8 Core Computational Unit – FFT Engine – Magnitude and Log-Magnitude Post-Processing

The magnitude and log-magnitude post-processing block computes absolute value or log2 of the absolute value of its input. Because this block is connected to the output of the FFT engine, the computation of absolute value (and log2) can be directly performed on the streaming FFT output.

Alternately, the FFT block can be bypassed and only the magnitude and log-magnitude block can be employed.

The processing in this block first involves computation of magnitude (absolute value) of the input samples in the magnitude subblock (using JPL approximation). The result of the magnitude computation is fed into a Log2 computation subblock, which uses a look-up table-based approximation to compute logarithm- base-2 of the magnitude.

As shown in Figure 14, if the register-bit ABS_EN is set, the magnitude computation subblock is enabled. In addition, if the register-bit LOG2_EN is set, then the Log2 computation subblock is also enabled. Note that setting LOG2_EN makes sense only when ABSEN is also set.

The magnitude computation uses JPL (Levitt and Morris) approximation. This approximation for magnitude of a complex number (I + jQ) is defined as follows, let U = max(│I│,│Q│) and V = min(│I│,│Q│).

Then, the magnitude can be approximated as follows in Equation 1.

$$\text{Magnitude} \approx \max(U + V/8,\ 7U/8 + V/2) \quad (1)$$

The magnitude output is 24-bits wide (real number).

Next, the log2 computation of the magnitude value is achieved as follows. Any unsigned input number N can be written as N = $2^k(1 + f)$ and the log2(N) can then be written as follows in Equation 2.

$$\log_2(N) = k + \log_2(1+f) \quad (2)$$

The implementation of log2 computation uses the previous formula, where a look-up table approximation is used to generate the second term, for example, log2(1 + f). The accuracy of the log2 computation is shown in Figure 19. The log2 output is 16-bits wide. The 16-bit logarithm output consists of 5 bits of integer part and 11 bits of fractional part.

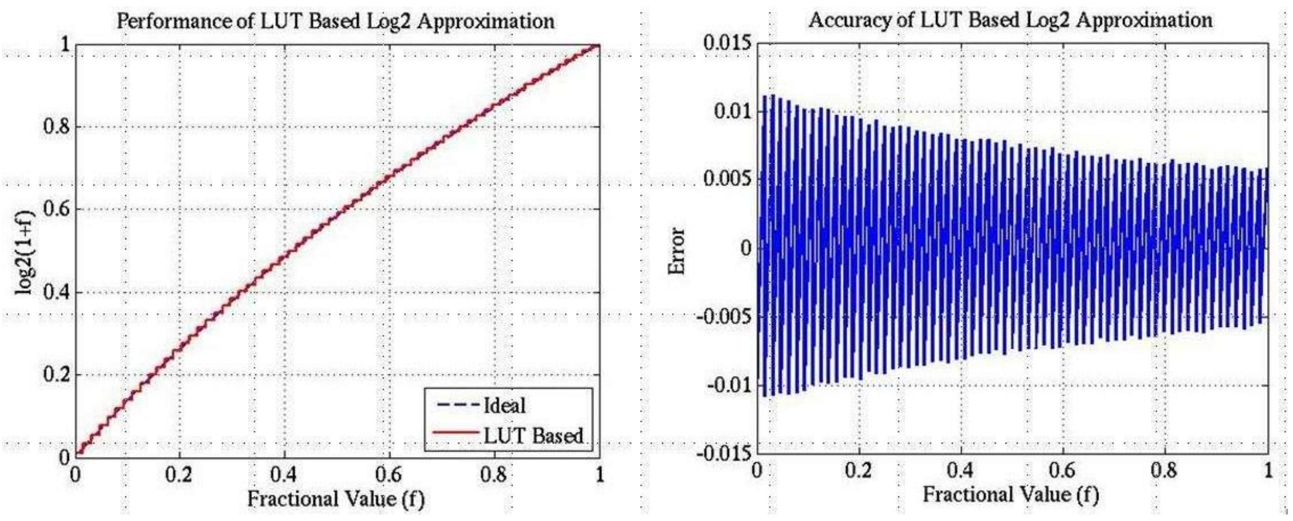


Figure 19: Accuracy of Log2 Computation

Depending on the settings of ABS_EN and LOG2_EN, either the magnitude or the log-magnitude is sent as the final output of the core computational unit. The final output of the core computational unit going to the output formatter is 24-bits I and 24-bits Q. Thus, if either magnitude or log-magnitude is enabled, the Q- values are just made zeros. Similarly, when log2 is enabled, because the output is 16-bits, 8 MSBs are filled as zero.

The output formatter handles writing the samples to the destination memory as per the configured destination memory access pattern described in a previous section

5.1.9 Core Computational Unit – FFT Engine – Register Descriptions

Table 6 lists all the registers of the FFT Engine within the core computational unit.

Table 6: FFT Engine Registers

Register	**Width**	**Parameter Set**	**Description**
WINDOW_EN	1	Yes	Windowing Enable: This register-bit enables or disables the pre-FFT windowing operation. If this register is set to 1, then the windowing is enabled, otherwise, it is disabled. The exact window function (coefficients) to be applied is specified in a dedicated Window RAM, which is 2048 × 32 bits in size.
FFT_EN	1	Yes	FFT Enable: This register-bit is used to enable the FFT computation. If FFT_EN = 1, then the FFT computation is enabled. Otherwise, it is disabled (bypassed).
ABS_EN	1	Yes	Magnitude Enable: This register-bit is used to enable the magnitude calculation. If this register bit is set, then the magnitude calculation is enabled, else it is bypassed. When enabled, the magnitude (absolute value) of the input complex samples are calculated using JPL approximation and the resulting magnitude value is sent on the I-arm of the output. The Q-arm is made zeros.
LOG2_EN	1	Yes	Log2 Enable: This register-bit is used to enable the Log2 computation. If this register bit is set, then the Log2 computation is enabled, else it is bypassed. Note that setting this register bit only makes sense if the inputs to the Log2 computation are unsigned real numbers, such as when the Magnitude Enable bit (ABS_EN) is also set. When enabled, the Log2 of the magnitude of the input samples is calculated and sent out on the I-arm of the output. The Q-arm is made zeros.
WINDOW_START	11	Yes	Windowing coefficients start location in Window RAM: This register specifies the starting location (32-bit word index) of the window coefficients within the Window RAM. The purpose of this register is to allow multiple windows (for example, one window of 512 coefficients and another window of 256 coefficients) to be stored in the Window RAM and one of these windows can be used by programming this start location register appropriately in the current parameter set.

WINSYMM	1	Yes	Window symmetry: This register-bit indicates whether the complete set of window coefficients are stored in the Window RAM or whether one half of the coefficients are stored. If this register bit is set, it means that the window function is symmetric and therefore, only one half of the window function coefficients are stored in the Window RAM. See the description section related to Windowing computation for more details.
WINDOW_MODE	2	Yes	Window Mode: 00b : 18-bit signed real coefficients, 01b: 16-bit signed real coefficients, 10b: 16-bit I, 16-bit Q complex coefficients
FFTSIZE	4	Yes	FFT size: This register indicates the number of FFT radix-2 butterfly stages employed. Refer detailed description section for more details on this register.
FFTSIZE3X_EN	1	Yes	FFT size 3X enable: This register indicates whether to engage the additional FFT radix-3 butterfly stage. Together with FFTSIZE, this register specifies the FFT size. This can be used to realize FFTs of length $3x2^N$ point FFTs..
FFTSIZE_DIM2	4	Yes	2D FFT dimension specification: This register can be used to realize 2 dimensional FFTs. If this register is set to 0 (default), the FFT engine computes the usual one dimensional FFT. Otherwise, it computes a two dimensional FFT of size 2^(FFTSIZE–FFTSIZE_DIM2) x 2^FFTSIZE_DIM2. <TO_BE_UPDATED_IN_A_FUTURE_DOC_VERSION>
BFLY_SCALING	12	Yes	Butterfly scaling for radix-2 stages: This register is used to control the butterfly scaling at each of the radix-2 butterfly stages. If the Nth bit in this register is set to 0, then the 25-bit output of the Nth radix-2 stage from the last is saturated to 24-bit. Otherwise it is scaled down by 2 and rounded to produce a 24-bit output.
BFLY_SCALING_FFT3X	2	Yes	Butterfly scaling for radix-3 stage: This register is applicable only if FFTSIZE3X_EN is set to 1. This register is used to control the butterfly scaling in the 3-point FFT structure that precedes the powers-of-2 FFT structure. If this register is set to 0, then that 26-bit output after radix-3 stage is saturated at the MSB to get back to 24 bits. If it is 2, it is rounded to remove 2 LSBs to get back to 24 bits. A middle option exists by setting this to 1. In this case, the 26 bit temporary output is convergent-rounded to remove 1 LSB and the 25-bit output thus obtained is saturated to 24 bits.
DITHER_TWID_EN	1	No	Twiddle factor dithering enable: This register-bit is used to enable and disable dithering of twiddle factors in the FFT. The twiddle factors are 24-bits wide (24-bits for each I and Q), but they are quantized to 21-bits before twiddle factor multiplication. This quantization is implemented with dithering on the LSB, to avoid periodic quantization pattern affecting SFDR performance of the FFT. It is recommended to keep this register bit set to 1 (dithering enabled), with appropriate LSFR seed loaded.

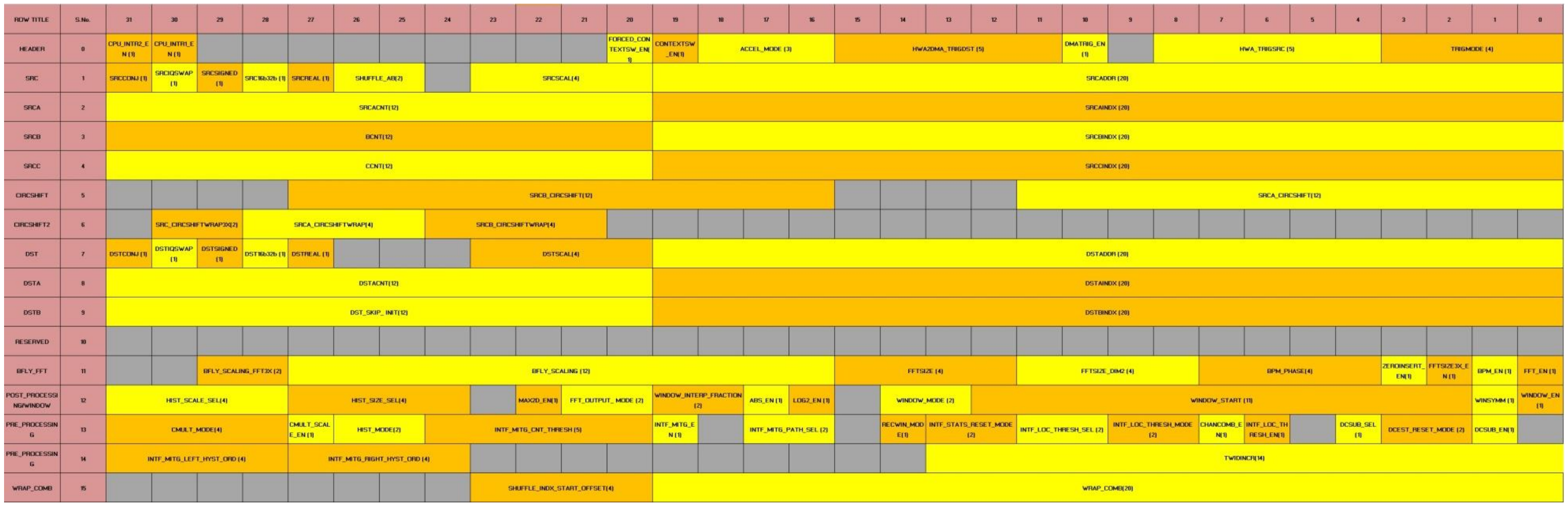
LFSR_SEED	29	No	Seed for LFSR (random pattern): For twiddle factor dithering, there is an LFSR that is used, whose seed value is loaded by writing to this 29-bit LFSRSEED register. The LFSRSEED register should be set to any non-zero value, say 0x1234567. To load the LFSR seed, a pulse signal needs to be provided, by writing a 1 followed by a 0 (i.e., by setting and clearing) the LFSRLOAD
LFSR_LOAD	1	No	For twiddle factor dithering, there is an LFSR that is used, whose seed value is loaded by writing to this 29-bit LFSRSEED register. The LFSRSEED register should be set to any non-zero value, say 0x1234567. To load the LFSR seed, a pulse signal needs to be provided, by writing a 1 followed by a 0 (i.e., by setting and clearing) the LFSRLOAD register-bit.
FFT_CLIP	13	No	FFT Clip Status (read-only): This is a read-only status register, which indicates any saturation/clipping events that have happened in the FFT butterfly stages. Note that each of the individual butterfly stages in the FFT can be programmed to either saturate the MSB or round the LSB. Whenever saturation of MSB is used in any stage, there is a possibility that that stage can saturate or clip samples. In that case, this saturation event is indicated in the corresponding bit in this status register, so that the processor can read it. If multiple FFTs are performed, this status register includes any saturation events happening in any of them. This status register can only be cleared by the R4F, by setting another single-bit register CLR_FFTCLIP, so that the saturation status indication gets cleared back to 0 and any subsequent saturation events can be freshly monitored. The MSB of this register indicates clip status corresponding to the radix 3 butterfly (note: it is the MSB, independent of the number of radix-
CLR_FFTCLIP	1	No	Clear FFT Clip Status register: This register bit, when set, clears the FFTCLIP register.
WINDOW_RAM[2048]	32b each	No	This RAM stores the window co-efficients. Note that there is only one RAM and based on Window_mode, the samples are accordingly chosen as illustrated in Fig. 15

ACCEL_MODE	3	Yes	Select Core Computational Unit Data Path: This register selects the data-path of the accelerator's core computational unit – for example, it selects which of the paths: the FFT engine path, or the CFAR engine path, or the compression/decompression path, or the local maxima engine path, or none is active. Value = 0b000: FFT engine path
			Value = 0b001: CFAR engine path
			Value = 0b010: Compression / decompression engine path
			Value = 0b011: Local Maxima engine path
			Value = 0b111: No Operation.
			The No Operation setting can be used together with an appropriate trigger mode to cause the state machine to wait for an event before moving to the next parameter-set.

WINDOW_INTERP_FRACTION CMULT_MODE TWIDINCR FFT_OUTPUT_MODE FFTSUMDIV MAX<n>_VALUE ISUM<n>, QSUM<n>	–	–	Described in part two of this user's guide. For the immediate purposes relevant to part one of this user's guide, all of these registers should be kept as 0.

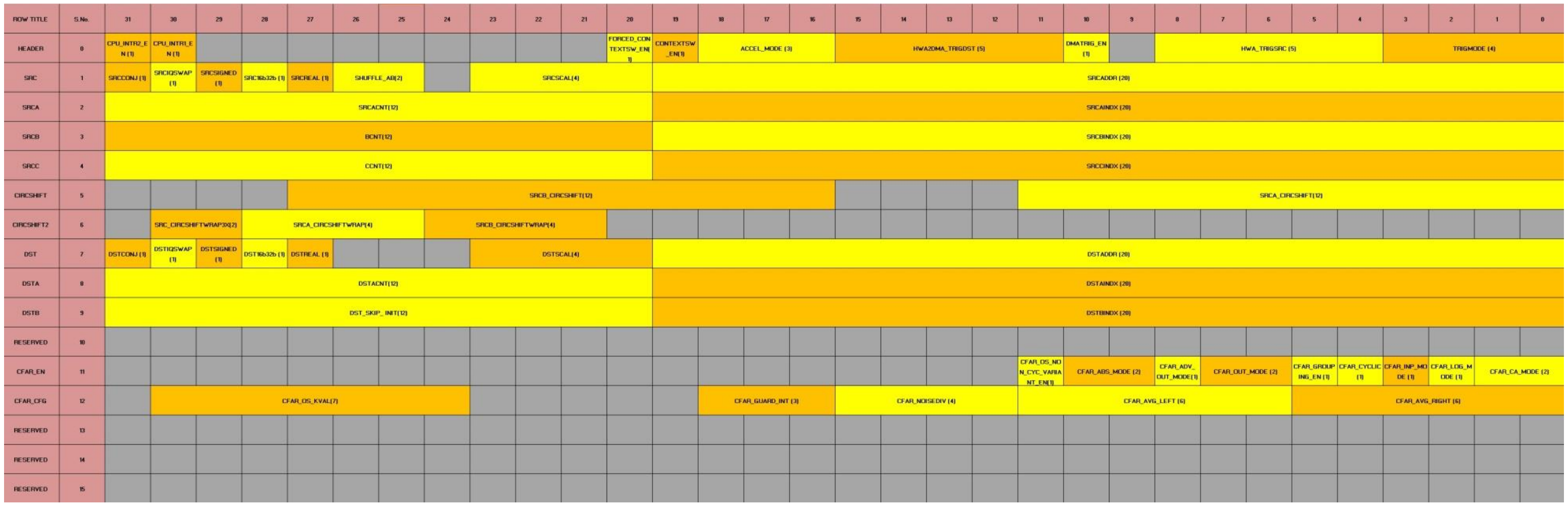
Appendix A

The parameter-set register layout is provided below for all the accelerator modes. Note that the parameter-set RAM must be written using 32-bit word writes only (i.e., byte-writes and half-word writes are NOT supported).



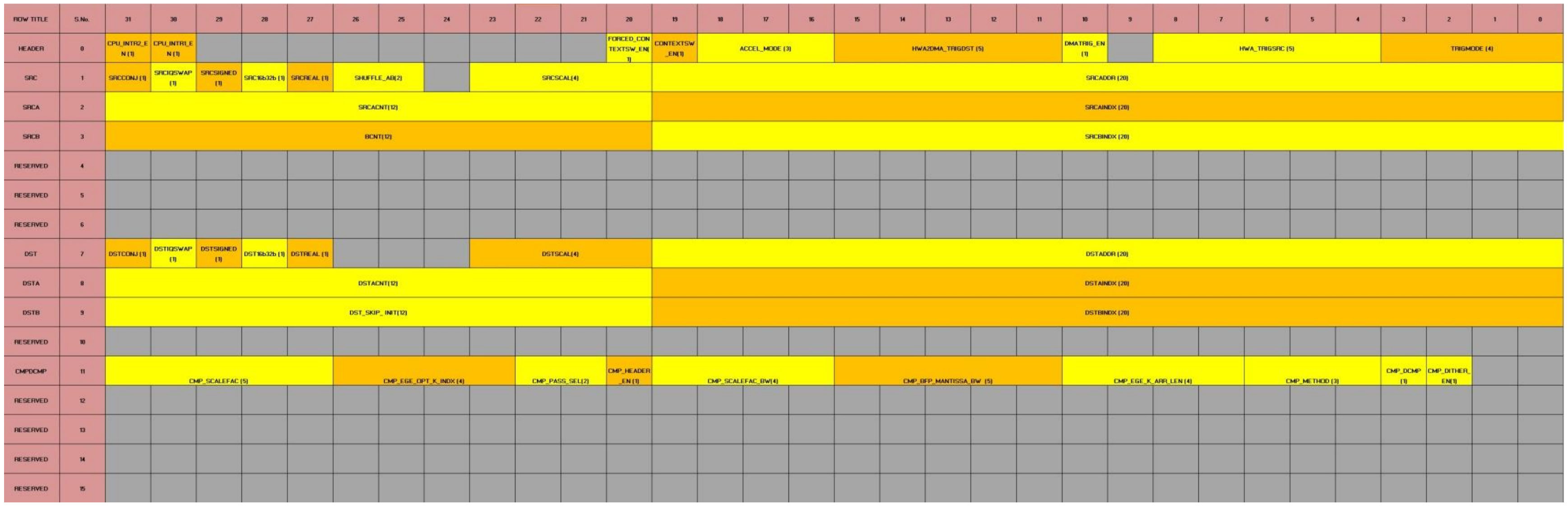
ROW TITLE	S.No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEADER	0	CPU_INTR2_EN (1)	CPU_INTR1_EN (1)										FORCED_CONTEXTSW_EN(1)	CONTEXTSW_EN(1)	ACCEL_MODE (3)			HWA2DMA_TRIGDST (5)					DMATRIG_EN (1)		HWA_TRIGSRC (5)					TRIGMODE (4)			
SRC	1	SRCCONJ (1)	SRCIQSWAP (1)	SRCSIGNED (1)	SRC16b32b (1)	SRCREAL (1)	SHUFFLE_AB(2)			SRCSCAL(4)				SRCADDR (20)																			
SRCA	2	SRCACNT(12)												SRCAINDX (20)																			
SRCB	3	BCNT(12)												SRCBINDX (20)																			
SRCC	4	CCNT(12)												SRCCINDX (20)																			
CIRCSHIFT	5					SRCB_CIRCSHIFT(12)																SRCA_CIRCSHIFT(12)											
CIRCSHIFT2	6		SRC_CIRCSHIFTWRAP3X(2)		SRCA_CIRCSHIFTWRAP(4)				SRCB_CIRCSHIFTWRAP(4)																								
DST	7	DSTCONJ (1)	DSTIQSWAP (1)	DSTSIGNED (1)	DST16b32b (1)	DSTREAL (1)				DSTSCAL(4)				DSTADDR (20)																			
DSTA	8	DSTACNT(12)												DSTAINDX (20)																			
DSTB	9	DST_SKIP_ INIT(12)												DSTBINDX (20)																			
RESERVED	10																																
BFLY_FFT	11			BFLY_SCALING_FFT3X (2)		BFLY_SCALING (12)												FFTSIZE (4)				FFTSIZE_DIM2 (4)				BPM_PHASE(4)				ZEROINSERT_EN(1)	FFTSIZE3X_EN (1)	BPM_EN (1)	FFT_EN (1)
POST_PROCESSING/WINDOW	12	HIST_SCALE_SEL(4)				HIST_SIZE_SEL(4)					MAX2D_EN(1)	FFT_OUTPUT_ MODE (2)		WINDOW_INTERP_FRACTION (2)		ABS_EN (1)	LOG2_EN (1)		WINDOW_MODE (2)		WINDOW_START (11)											WINSYMM (1)	WINDOW_EN (1)
PRE_PROCESSING	13	CMULT_MODE(4)				CMULT_SCALE_EN (1)	HIST_MODE(2)		INTF_MITG_CNT_THRESH (5)					INTF_MITG_EN (1)		INTF_MITG_PATH_SEL (2)			RECWIN_MODE(1)	INTF_STATS_RESET_MODE (2)		INTF_LOC_THRESH_SEL (2)		INTF_LOC_THRESH_MODE (2)		CHANCOMB_EN(1)	INTF_LOC_THRESH_EN(1)		DCSUB_SEL (1)	DCEST_RESET_MODE (2)		DCSUB_EN(1)	
PRE_PROCESSING	14	INTF_MITG_LEFT_HYST_ORD (4)				INTF_MITG_RIGHT_HYST_ORD (4)														TWIDINCR(14)													
WRAP_COMB	15									SHUFFLE_INDX_START_OFFSET(4)				WRAP_COMB(20)																			

Figure 20. FFT path parameter set layout



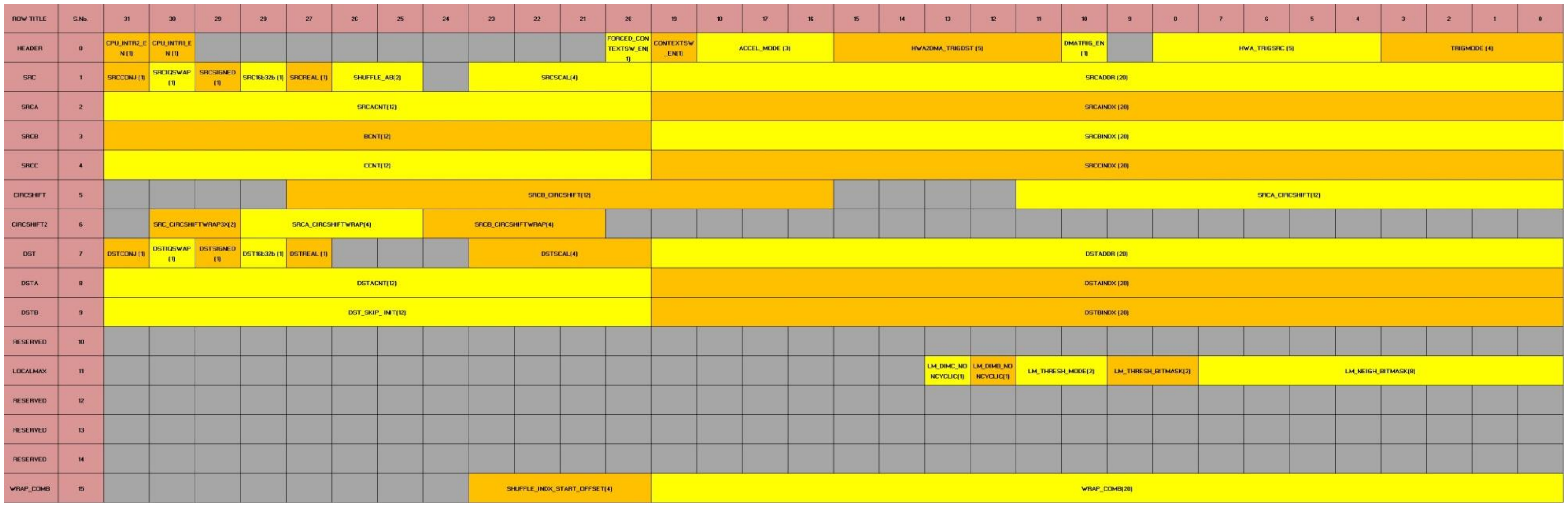
ROW TITLE	S.No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEADER	0	CPU_INTR2_EN (1)	CPU_INTR1_EN (1)										FORCED_CONTEXTSW_EN(1)	CONTEXTSW_EN(1)	ACCEL_MODE (3)			HWA2DMA_TRIGDST (5)					DMATRIG_EN (1)		HWA_TRIGSRC (5)					TRIGMODE (4)			
SRC	1	SRCCONJ (1)	SRCIQSWAP (1)	SRCSIGNED (1)	SRC16b32b (1)	SRCREAL (1)	SHUFFLE_AB(2)			SRCSCAL(4)				SRCADDR (20)																			
SRCA	2	SRCACNT(12)												SRCAINDX (20)																			
SRCB	3	BCNT(12)												SRCBINDX (20)																			
SRCC	4	CCNT(12)												SRCCINDX (20)																			
CIRCSHIFT	5					SRCB_CIRCSHIFT(12)																SRCA_CIRCSHIFT(12)											
CIRCSHIFT2	6		SRC_CIRCSHIFTWRAP3X(2)		SRCA_CIRCSHIFTWRAP(4)				SRCB_CIRCSHIFTWRAP(4)																								
DST	7	DSTCONJ (1)	DSTIQSWAP (1)	DSTSIGNED (1)	DST16b32b (1)	DSTREAL (1)				DSTSCAL(4)				DSTADDR (20)																			
DSTA	8	DSTACNT(12)												DSTAINDX (20)																			
DSTB	9	DST_SKIP_ INIT(12)												DSTBINDX (20)																			
RESERVED	10																																
CFAR_EN	11																					CFAR_OS_NON_CYC_VARIANT_EN(1)	CFAR_ABS_MODE (2)		CFAR_ADV_OUT_MODE(1)	CFAR_OUT_MODE (2)		CFAR_GROUPING_EN (1)	CFAR_CYCLIC (1)	CFAR_INP_MODE (1)	CFAR_LOG_MODE (1)	CFAR_CA_MODE (2)	
CFAR_CFG	12		CFAR_OS_KVAL(7)												CFAR_GUARD_INT (3)			CFAR_NOISEDIV (4)				CFAR_AVG_LEFT (6)						CFAR_AVG_RIGHT (6)					
RESERVED	13																																
RESERVED	14																																
RESERVED	15																																

Figure 21. CFAR path parameter set layout



ROW TITLE	S.No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEADER	0	CPU_INTR2_EN (1)	CPU_INTR1_EN (1)										FORCED_CONTEXTSW_EN(1)	CONTEXTSW_EN(1)	ACCEL_MODE (3)			HWA2DMA_TRIGDST (5)					DMATRIG_EN (1)		HWA_TRIGSRC (5)					TRIGMODE (4)			
SRC	1	SRCCONJ (1)	SRCIQSWAP (1)	SRCSIGNED (1)	SRC16b32b (1)	SRCREAL (1)	SHUFFLE_AB(2)			SRCSCAL(4)				SRCADDR (20)																			
SRCA	2	SRCACNT(12)												SRCAINDX (20)																			
SRCB	3	BCNT(12)												SRCBINDX (20)																			
RESERVED	4																																
RESERVED	5																																
RESERVED	6																																
DST	7	DSTCONJ (1)	DSTIQSWAP (1)	DSTSIGNED (1)	DST16b32b (1)	DSTREAL (1)				DSTSCAL(4)				DSTADDR (20)																			
DSTA	8	DSTACNT(12)												DSTAINDX (20)																			
DSTB	9	DST_SKIP_ INIT(12)												DSTBINDX (20)																			
RESERVED	10																																
CMPDCMP	11	CMP_SCALEFAC (5)					CMP_EGE_OPT_K_INDX (4)				CMP_PASS_SEL(2)		CMP_HEADER_EN (1)	CMP_SCALEFAC_BW(4)				CMP_BFP_MANTISSA_BW (5)					CMP_EGE_K_ARR_LEN (4)				CMP_METHOD (3)			CMP_DCMP (1)	CMP_DITHER_EN(1)		
RESERVED	12																																
RESERVED	13																																
RESERVED	14																																
RESERVED	15																																

Figure 22. Compression decompression path parameter set layout



ROW TITLE	S.No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEADER	0	CPU_INTR2_EN (1)	CPU_INTR1_EN (1)										FORCED_CONTEXTSW_EN(1)	CONTEXTSW_EN(1)	ACCEL_MODE (3)			HWA2DMA_TRIGDST (5)					DMATRIG_EN (1)		HWA_TRIGSRC (5)					TRIGMODE (4)			
SRC	1	SRCCONJ (1)	SRCIQSWAP (1)	SRCSIGNED (1)	SRC16b32b (1)	SRCREAL (1)	SHUFFLE_AB(2)			SRCSCAL(4)				SRCADDR (20)																			
SRCA	2	SRCACNT(12)												SRCAINDX (20)																			
SRCB	3	BCNT(12)												SRCBINDX (20)																			
SRCC	4	CCNT(12)												SRCCINDX (20)																			
CIRCSHIFT	5					SRCB_CIRCSHIFT(12)																SRCA_CIRCSHIFT(12)											
CIRCSHIFT2	6		SRC_CIRCSHIFTWRAP3X(2)		SRCA_CIRCSHIFTWRAP(4)				SRCB_CIRCSHIFTWRAP(4)																								
DST	7	DSTCONJ (1)	DSTIQSWAP (1)	DSTSIGNED (1)	DST16b32b (1)	DSTREAL (1)				DSTSCAL(4)				DSTADDR (20)																			
DSTA	8	DSTACNT(12)												DSTAINDX (20)																			
DSTB	9	DST_SKIP_ INIT(12)												DSTBINDX (20)																			
RESERVED	10																																
LOCALMAX	11																			LM_DIMC_NONCYCLIC(1)	LM_DIMB_NONCYCLIC(1)	LM_THRESH_MODE(2)		LM_THRESH_BITMASK(2)		LM_NEIGH_BITMASK(8)							
RESERVED	12																																
RESERVED	13																																
RESERVED	14																																
WRAP_COMB	15									SHUFFLE_INDX_START_OFFSET(4)				WRAP_COMB(20)																			

Figure 23. **Local maxima path parameter set layout**