Intel® HD Graphics OpenSource PRM

Volume 4 Part 1: Subsystem and Cores - Shared Functions

For the all new 2010 Intel Core Processor Family
Programmer's Reference Manual (PRM)

February 2010

Revision 1.0

IHD-022810-R1V4PT1

Contents

Revision History

Document Number	Revision Number	Description	Revision Date
IHD-022810-R1V4PT1	1.0	First Release.	February 2010

1. Introduction

This Programmer's Reference Manual (PRM) describes the architectural behavior and programming environment of the Sandy Bridge chipset family, Havendale/Auburndale chipset family, Intel® 965 Chipset family and Intel® G35 Express Chipset GMCH graphics devices (see Table 1-1). The GMCH's Graphics Controller (GC) contains an extensive set of registers and instructions for configuration, 2D, 3D, and Video systems. The PRM describes the register, instruction, and memory interfaces and the device behaviors as controlled and observed through those interfaces. The PRM also describes the registers and instructions and provides detailed bit/field descriptions.

The Programmer's Reference Manual is organized into five volumes:

PRM, Volume 1: Graphics Core

Volume 1, Part 1, 2, 3, 4 and 5 covers the overall Graphics Processing Unit (GPU), without much detail on 3D, Media, or the core subsystem. Topics include the command streamer, context switching, and memory access (including tiling). The Memory Data Formats can also be found in this volume.

The volume also contains a chapter on the Graphics Processing Engine (GPE). The GPE is a collective term for 3D, Media, the subsystem, and the parts of the memory interface that are used by these units. Display, blitter and their memory interfaces are *not* included in the GPE.

PRM, Volume 2: 3D/Media

Volume 2, Part 1, 2, 3, and 4 covers the 3D and Media pipelines in detail. This volume is where details for all of the "fixed functions" are covered, including commands processed by the pipelines, fixed-function state structures, and a definition of the inputs (payloads) and outputs of the threads spawned by these units.

This volume also covers the single Media Fixed Function, VLD. It describes how to initiate generic threads using the thread spawner (TS). It is generic threads which will be used for doing the majority of media functions. Programmable kernels will handle the algorithms for media functions such IDCT, Motion Compensation, and even Motion Estimation (used for encoding MPEG streams).

PRM, Volume 3: Display Registers

Volume 3, Part 1, 2, 3, and 4 describes the control registers for the display. The overlay registers and VGA registers are also cover in this volume.

PRM, Volume 4: Subsystem and Cores

Volume 4, Part 1 and 2 describes the GMCH programmable cores, or EUs, and the "shared functions", which are shared by more than one EU and perform functions such as I/O and complex math functions.

The shared functions consist of the sampler: extended math unit, data port (the interface to memory for 3D and media), Unified Return Buffer (URB), and the Message Gateway which is used by EU threads to signal each other. The EUs use messages to send data to and receive data from the subsystem; the messages are described along with the shared functions although the generic message send EU instruction is described with the rest of the instructions in the Instruction Set Architecture (ISA) chapters.

This latter part of this volume describes the GMCH core, or EU, and the associated instructions that are used to program it. The instruction descriptions make up what is referred to as an Instruction Set Architecture, or ISA. The ISA describes all of the instructions that the GMCH core can execute, along with the registers that are used to store local data.

Device Tags and Chipsets

Device "Tags" are used in various parts of this document as aliases for the device names/steppings, as listed in the following table. Note that stepping info is sometimes appended to the device tag, e.g., [**DevBW-C**]. Information without any device tagging is applicable to all devices/steppings.

Table 1-1. Supported Chipsets

Chipset Family Name	Device Name	Device Tag
Intel® Q965 Chipset Intel® Q963 Chipset Intel® G965 Chipset	82Q965 GMCH 82Q963 GMCH 82G965 GMCH	[**DevBW**]
Intel® G35 Chipset	82G35 GMCH	[**DevBW-E**]
Intel® GM965 Chipset Intel® GME965 Chipset	GM965 GMCH GME965 GMCH	[**DevCL**]
Mobile Intel® GME965 Express Chipset Mobile Intel® GM965 Express Chipset Mobile Intel® PM965 Express Chipset Mobile Intel® GL960 Express Chipset		[**DevCL**]
[Cantiga A-step (not productized)]	N/A	[**DevCTG**], **[DevCTG-A]**
[Cantiga B-step/Eaglelake converged core (not productized)]	TBD	**[DevCTG-B],**
[Havendale/Auburndale]	TBD	**[DevILK]**
[Sandy Bridge]	TBD	**[DevSNB]**

NOTES:

1. Unless otherwise specified, the information in this document applies to all of the devices mentioned in Table 1-1. For Information that does not apply to all devices, the Device Tag is used.
2. Throughout the PRM, references to "All" in a project field refters to all devices in Table 1-1.
3. Throughout the PRM, references to [DevBW] apply to both [DevBW] and [DevBW-E]. [DevBW-E] is referenced specifically for information that is [DevBW-E] only.
4. Stepping info is sometimes appended to the device tag (e.g., [DevBW-C]). Information without any device tagging is applicable to all devices/steppings.
5. A shorthand is used to (a) identify all devices/steppings prior to the device/stepping that the item pertains (e.g., "[Pre-DevSNB"], .

1.1 Notations and Conventions

1.1.1 Reserved Bits and Software Compatibility

In many register, instruction and memory layout descriptions, certain bits are marked as "Reserved". When bits are marked as reserved, it is essential for compatibility with future devices that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:
Do not depend on the states of any reserved bits when testing values of registers that contain such bits. Mask out the reserved bits before testing. Do not depend on the states of any reserved bits when storing to instruction or to a register. When loading a register or formatting an instruction, always load the reserved bits with the values indicated in the documentation, if any, or reload them with the values previously read from the register.

1.2 Terminology

Term	Abbr.	Definition
3D Pipeline	--	One of the two pipelines supported in the GPE. The 3D pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See *3D Pipeline*.
Adjacency	--	One can consider a single line object as existing in a strip of connected lines. The neighboring line objects are called "adjacent objects", with the non-shared endpoints called the "adjacent vertices." The same concept can be applied to a single triangle object, considering it as existing in a mesh of connected triangles. Each triangle shares edges with three other adjacent triangles, each defined by an non-shared adjacent vertex. Knowledge of these adjacent objects/vertices is required by some object processing algorithms (e.g., silhouette edge detection). See *3D Pipeline*.
Application IP	AIP	Application Instruction Pointer. This is part of the control registers for exception handling for a thread. Upon an exception, hardware moves the current IP into this register and then jumps to SIP.
Architectural Register File	ARF	A collection of architecturally visible registers for a thread such as address registers, accumulator, flags, notification registers, IP, null, etc. ARF should not be mistaken as just the address registers.
Array of Cores	--	Refers to a group of Gen4 EUs, which are physically organized in two or more rows. The fact that the EUs are arranged in an array is (to a great extent) transparent to CPU software or EU kernels.
Binding Table	--	Memory-resident list of pointers to surface state blocks (also in memory).
Binding Table Pointer	BTP	Pointer to a binding table, specified as an offset from the Surface State Base Address register.
Bypass Mode	--	Mode where a given fixed function unit is disabled and forwards data down the pipeline unchanged. Not supported by all FF units.
Byte	B	A numerical data type of 8 bits, B represents a signed byte integer.

Term	Abbr.	Definition
Child Thread		A branch-node or a leaf-node thread that is created by another thread. It is a kind of thread associated with the media fixed function pipeline. A child thread is originated from a thread (the parent) executing on an EU and forwarded to the Thread Dispatcher by the TS unit. A child thread may or may not have child threads depending on whether it is a branch-node or a leaf-node thread. All pre-allocated resources such as URB and scratch memory for a child thread are managed by its parent thread.
Clip Space	--	A 4-dimensional coordinate system within which a clipping frustum is defined. Object positions are projected from Clip Space to NDC space via “perspecitive divide” by the W coordinate, and then viewport mapped into Screen Space
Clipper	--	3D fixed function unit that removes invisible portions of the drawing sequence by discarding (culling) primitives or by “replacing” primitives with one or more primitives that replicate only the visible portion of the original primitive.
Color Calculator	CC	Part of the Data Port shared function, the color calculator performs fixed-function pixel operations (e.g., blending) prior to writing a result pixel into the render cache.
Command	--	Directive fetched from a ring buffer in memory by the Command Streamer and routed down a pipeline. Should not be confused with instructions which are fetched by the instruction cache subsystem and executed on an EU.
Command Streamer	CS or CSI	Functional unit of the Graphics Processing Engine that fetches commands, parses them and routes them to the appropriate pipeline.
Constant URB Entry	CURBE	A UE that contains “constant” data for use by various stages of the pipeline.
Control Register	CR	The read-write registers are used for thread mode control and exception handling for a thread.
Degenerate Object	--	Object that is invisible due to coincident vertices or because does not intersect any sample points (usually due to being tiny or a very thin sliver).
Destination	--	Describes an output or write operand.
Destination Size		The number of data elements in the destination of a Gen4 SIMD instruction.
Destination Width		The size of each of (possibly) many elements of the destination of a Gen4 SIMD instruction.
Double Quad word (DQword)	DQ	A fundamental data type, DQ represents 16 bytes.
Double word (DWord)	D or DW	A fundamental data type, D or DW represents 4 bytes.
Drawing Rectangle	--	A screen-space rectangle within which 3D primitives are rendered. An objects screen-space positions are relative to the Drawing Rectangle origin. See *Strips and Fans*.
End of Block	EOB	A 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer.
End Of Thread	EOT	a message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set in order to properly terminate.
Exception	--	Type of (normally rare) interruption to EU execution of a thread’s instructions. An exception occurrence causes the EU thread to begin executing the System Routine which is designed to handle exceptions.
Execution Channel	--	

Term	Abbr.	Definition
Execution Size	ExecSize	Execution Size indicates the number of data elements processed by a GEN4 SIMD instruction. It is one of the GEN4 instruction fields and can be changed per instruction.
Execution Unit	EU	Execution Unit. An EU is a multi-threaded processor within the GEN4 multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a GEN4 Core.
Execution Unit Identifier	EUID	The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU a thread is located. A thread can be uniquely identified by the EUID and TID.
Execution Width	ExecWidth	The width of each of several data elements that may be processed by a single Gen4 SIMD instruction.
Extended Math Unit	EM	A Shared Function that performs more complex math operations on behalf of several EUs.
FF Unit	--	A Fixed-Function Unit is the hardware component of a 3D Pipeline Stage. A FF Unit typically has a unique FF ID associated with it.
Fixed Function	FF	Function of the pipeline that is performed by dedicated (vs. programmable) hardware.
Fixed Function ID	FFID	Unique identifier for a fixed function unit.
FLT_MAX	fmax	The magnitude of the maximum representable single precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's.
Gateway	GW	See Message Gateway.
GEN4 Core		Alternative name for an EU in the GEN4 multi-processor system.
General Register File	GRF	Large read/write register file shared by all the EUs for operand sources and destinations. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.
General State Base Address	--	The Graphics Address of a block of memory-resident "state data", which includes state blocks, scratch space, constant buffers and kernel programs. The contents of this memory block are referenced via offsets from the contents of the General State Base Address register. See *Graphics Processing Engine*.
Geometry Shader	GS	Fixed-function unit between the vertex shader and the clipper that (if enabled) dispatches "geometry shader" threads on its input primitives. Application-supplied geometry shaders normally expand each input primitive into several output primitives in order to perform 3D modeling algorithms such as fur/fins. See *Geometry Shader*.
Graphics Address		The GPE virtual address of some memory-resident object. This virtual address gets mapped by a GTT or PGTT to a physical memory address. Note that many memory-resident objects are referenced not with Graphics Addresses, but instead with offsets from a "base address register".
Graphics Processing Engine	GPE	Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer.
Guardband	GB	Region that may be clipped against to make sure objects do not exceed the limitations of the renderer's coordinate space.
Horizontal Stride	HorzStride	The distance in element-sized units between adjacent elements of a Gen4 region-based GRF access.

Term	Abbr.	Definition
Immediate floating point vector	VF	A numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.
Immediate integer vector	V	A numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4-bit each. The 4-bit integer element is in 2's compliment form. It may be used to specify the type of an immediate operand in an instruction.
Index Buffer	IB	Buffer in memory containing vertex indices.
In-loop Deblocking Filter	ILDB	The deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe.
Instance		In the context of the VF unit, an instance is one of a sequence of sets of similar primitive data. Each set has identical vertex data but may have unique instance data that differentiates it from other sets in the sequence.
Instruction	--	Data in memory directing an EU operation. Instructions are fetched from memory, stored in a cache and executed on one or more Gen4 cores. Not to be confused with commands which are fetched and parsed by the command streamer and dispatched down the 3D or Media pipeline.
Instruction Pointer	IP	The address (really an offset) of the instruction currently being fetched by an EU. Each EU has its own IP.
Instruction Set Architecture	ISA	The GEN4 ISA describes the instructions supported by a GEN4 EU.
Instruction State Cache	ISC	On-chip memory that holds recently-used instructions and state variable values.
Interface Descriptor	--	Media analog of a State Descriptor.
Intermediate Z	IZ	Completion of the Z (depth) test at the front end of the Windower/Masker unit when certain conditions are met (no alpha, no pixel-shader computed Z values, etc.)
Inverse Discrete Cosine Transform	IDCT	the stage in the video decoding pipe between IQ and MC
Inverse Quantization	IQ	A stage in the video decoding pipe between IS and IDCT.
Inverse Scan	IS	A stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for both MPEG-2 and WMV.
Jitter		Just-in-time compiler.
Kernel	--	A sequence of Gen4 instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to Gen4 instructions.
Least Significant Bit	LSB	
MathBox	--	See Extended Math Unit
Media	--	Term for operations such as video decode and encode that are normally performed by the Media pipeline.
Media Pipeline	--	Fixed function stages dedicated to media and "generic" processing, sometimes referred to as the generic pipeline.

Term	Abbr.	Definition
Message	--	Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of GEN4 architecture.
Message Gateway	--	Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline.
Message Register File	MRF	Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction.
Most Significant Bit	MSB	
Motion Compensation	MC	Part of the video decoding pipe.
Motion Picture Expert Group	MPEG	MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
Motion Vector Field Selection	MVFS	A four-bit field selecting reference fields for the motion vectors of the current macroblock.
Multi Render Targets	MRT	Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state.
Normalized Device Coordinates	NDC	Clip Space Coordinates that have been divided by the Clip Space "W" component.
Object	--	A single triangle, line or point.
Open GL	OGL	A Graphics API specification associated with Linux.
Parent Thread	--	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Pipeline Stage	--	A abstracted element of the 3D pipeline, providing functions performed by a combination of the corresponding hardware FF unit and the threads spawned by that FF unit.
Pipelined State Pointers	PSP	Pointers to state blocks in memory that are passed down the pipeline.
Pixel Shader	PS	Shader that is supplied by the application, translated by the jitter and is dispatched to the EU by the Windower (conceptually) once per pixel.
Point	--	A drawing object characterized only by position coordinates and width.
Primitive	--	Synonym for object: triangle, rectangle, line or point.
Primitive Topology	--	A composite primitive such as a triangle strip, or line list. Also includes the objects triangle, line and point as degenerate cases.
Provoking Vertex	--	The vertex of a primitive topology from which vertex attributes that are constant across the primitive are taken.
Quad Quad word (QQword)	QQ	A fundamental data type, QQ represents 32 bytes.
Quad Word (QWord)	QW	A fundamental data type, QW represents 8 bytes.
Rasterization		Conversion of an object represented by vertices into the set of pixels that make up the object.
Region-based addressing	--	Collective term for the register addressing modes available in the EU instruction set that permit discontiguous register data to be fetched and used as a single operand.

Term	Abbr.	Definition
Render Cache	RC	Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test.
Render Target	RT	A destination surface in memory where render results are written.
Render Target Array Index	--	Selector of which of several render targets the current operation is targeting.
Root Thread	--	A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Sampler	--	Shared function that samples textures and reads data from buffers on behalf of EU programs.
Scratch Space	--	Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc.
Shader	--	A Gen4 program that is supplied by the application in a high level shader language, and translated to Gen4 instructions by the jitter.
Shared Function	SF	Function unit that is shared by EUs. EUs send messages to shared functions; they consume the data and may return a result. The Sampler, Data Port and Extended Math unit are all shared functions.
Shared Function ID	SFID	Unique identifier used by kernels and shaders to target shared functions and to identify their returned messages.
Single Instruction Multiple Data	SIMD	The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at instruction level. It can also be used to describe the instructions in such architecture.
Source	--	Describes an input or read operand
Spawn	--	To initiate a thread for execution on an EU. Done by the thread spawner as well as most FF units in the 3D pipeline.
Sprite Point	--	Point object using full range texture coordinates. Points that are not sprite points use the texture coordinates of the point's center across the entire point object.
State Descriptor	--	Blocks in memory that describe the state associated with a particular FF, including its associated kernel pointer, kernel resource allowances, and a pointer to its surface state.
State Register	SR	The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP.
State Variable	SV	An individual state element that can be varied to change the way given primitives are rendered or media objects processed. On Gen4 state variables persist only in memory and are cached as needed by rendering/processing operations except for a small amount of non-pipelined state.
Stream Output	--	A term for writing the output of a FF unit directly to a memory buffer instead of, or in addition to, the output passing to the next FF unit in the pipeline. Currently only supported for the Geometry Shader (GS) FF unit.

Term	Abbr.	Definition
Strips and Fans	SF	Fixed function unit whose main function is to decompose primitive topologies such as strips and fans into primitives or objects.
Sub-Register		Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, *r2*, is 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*.
Subsystem	--	The Gen4 name given to the resources shared by the FF units, including shared functions and EUs.
Surface	--	A rendering operand or destination, including textures, buffers, and render targets.
Surface State	--	State associated with a render surface including
Surface State Base Pointer	--	Base address used when referencing binding table and surface state data.
Synchronized Root Thread	--	A root thread that is dispatched by TS upon a 'dispatch root thread' message.
System IP	SIP	There is one global System IP register for all the threads. From a thread's point of view, this is a virtual read only register. Upon an exception, hardware performs some bookkeeping and then jumps to SIP.
System Routine	--	Sequence of Gen4 instructions that handles exceptions. SIP is programmed to point to this routine, and all threads encountering an exception will call it.
Thread		An instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction – a send instruction with EOT that signals the thread termination. Threads in GEN4 system may be independent from each other or communicate with each other through Message Gateway share function.
Thread Dispatcher	TD	Functional unit that arbitrates thread initiation requests from Fixed Functions units and instantiates the threads on EUs.
Thread Identifier	TID	The field within a thread state register (SR0) that identifies which thread slots on an EU a thread occupies. A thread can be uniquely identified by the EUID and TID.
Thread Payload		Prior to a thread starting execution, some amount of data will be pre-loaded in to the thread's GRF (starting at r0). This data is typically a combination of control information provided by the spawning entity (FF Unit) and data read from the URB.
Thread Spawner	TS	The second and the last fixed function stage of the media pipeline that initiates new threads on behalf of generic/media processing.
Topology		See Primitive Topology.
Unified Return Buffer	URB	The on-chip memory managed/shared by GEN4 Fixed Functions in order for a thread to return data that will be consumed either by a Fixed Function or other threads.
Unsigned Byte integer	UB	A numerical data type of 8 bits.
Unsigned Double Word integer	UD	A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.
Unsigned Word integer	UW	A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.

Term	Abbr.	Definition
Unsynchronized Root Thread	--	A root thread that is automatically dispatched by TS.
URB Dereference	--	
URB Entry	UE	URB Entry: A logical entity stored in the URB (such as a vertex), referenced via a URB Handle.
URB Entry Allocation Size	--	Number of URB entries allocated to a Fixed Function unit.
URB Fence	Fence	Virtual, movable boundaries between the URB regions owned by each FF unit.
URB Handle	--	A unique identifier for a URB entry that is passed down a pipeline.
URB Reference	--	
Variable Length Decode	VLD	The first stage of the video decoding pipe that consists mainly of bit-wide operations. GEN4 supports hardware VLD acceleration in the VFE fixed function stage.
Vertex Buffer	VB	Buffer in memory containing vertex attributes.
Vertex Cache	VC	Cache of Vertex URB Entry (VUE) handles tagged with vertex indices.
Vertex Fetcher	VF	The first FF unit in the 3D pipeline responsible for fetching vertex data from memory. Sometimes referred to as the Vertex Formatter.
Vertex Header	--	Vertex data required for every vertex appearing at the beginning of a Vertex URB Entry.
Vertex ID	--	Unique ID for each vertex that can optionally be included in vertex attribute data sent down the pipeline and used by kernel/shader threads.
Vertex URB Entry	VUE	A URB entry that contains data for a specific vertex.
Vertical Stride	VertStride	The distance in element-sized units between 2 vertically-adjacent elements of a Gen4 region-based GRF access.
Video Front End	VFE	The first fixed function in the GEN4 generic pipeline; performs fixed-function media operations.
Viewport	VP	
Windower IZ	WIZ	Term for Windower/Masker that encapsulates its early (“intermediate”) depth test function.
Windower/Masker	WM	Fixed function triangle/line rasterizer.
Word	W	A numerical data type of 16 bits, W represents a signed word integer.

2. Subsystem Overview

2.1 Introduction

The Gen4 subsystem consists of an array of *execution units* (*EUs*, sometimes referred to as an arrray of *cores*) along with a set of *shared functions* outside the EUs that the EUs leverage for I/O and for complex computations. Programmers access the Gen4 Subsystem via the 3D or Media pipelines.

EUs are general-purpose programmable cores that support a rich instruction set that has been optimized to support various 3D API shader languages as well as media functions (primarily video) processing.

Shared functions are hardware units which serve to provide specialized supplemental functionality for the EUs. A shared function is implemented where the demand for a given specialized function is insufficient to justify the costs on a per-EU basis. Instead a single instantiation of that specialized function is implemented as a stand-alone entity outside the EUs and shared amongst the EUs.

Invocation of the shared functionality is performed via a communication mechanism call a "message". A message is a small, self-contained packet of information created by a kernel and directed to specific shared function. The message is defined by sequential series of MRF registers which hold message operands, a destination shared function ID, a function-specific encoding of the desired operation to be performed, and a destination GRF register to which any writeback response is to be directed. Messages are dispatched to the shared function under software control via the 'send' instruction. This instruction identifies the contents of the message and the GRF register location(s) to direct any response.

The message construction and delivery mechanisms are general in their definition and capable of supporting a wide variety of shared functions.

2.2 Subsystem Topology

The subsystem is organized as an array of EUs, and a set of functions that are shared among all of the EUs. (The EU array is further divided into rows with each row having its own first level instruction cache and Extended Math shared function, though this aspect of the implemented topology is not exposed to software). The Sampler, DataPort, URB and Message Gateway functions are shared among the entire array of EUs.

2.3 Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on. Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple EU cores each with multiple threads can generally be ignored by software. There are some exceptions to this rule: e.g., for

- thread-to-thread communication (see *Message Gateway*, *Media*)
- synchronization of thread output to memory buffers (see *Geometry Shader*).

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs. See the *Gen4 Core* volume for details such as EU registers and instruction set.

2.4 Thread Dispatching

When the 3D and Media pipelines send requests for thread initiation to the Subsystem, the thread Dispatcher receives the requests. The dispatcher performs such tasks as arbitrating between concurrent requests, assigning requested threads to hardware threads on EUs, allocating register space in each EU among multiple threads, and initializing a thread's registers with data from the fixed functions and from the URB. This operation is largely transparent to software.

2.5 Shared Functions

In general, a shared function has the ability to receive messages at its input, perform some specialized amount of work for each, and if required, generate output back to the message's originating execution unit (Message Gateway may generate output to a target execution unit specified by the message).

To uniquely identify shared functions, each is assigned a unique 4-bit identifier code called its 'Function ID'. This ID is specified in the 'send' instruction's 32b <desc> field of each message. Gen4 Function ID assignments are listed in the *Graphics Processing Engine* chapter of this specification.

Each shared function may support one or more related operations within itself. For example an Extended Math shared function may support operations such as reciprocal, sine, cosine, and/or others. These are generically referred to as sub-functions. The communication method as to which sub-function is desired is typically contained in the 16b 'function-control' field of the 'send' instruction <desc> field. Alternatively, a function may choose to define sub-function encodings in-band within message payload, or in the case of a single function shared-function, the function code may be implied. The architecture, in no way interprets the sub-function code and the actual implementation choice is left to the function itself.

The Shared Function units included in the Subsystem are as follows (refer to the chapters devoted to each of these functions):

- Extended Math function
- Sampling Engine function
- DataPort function
- Message Gateway function
- Unified Return Buffer (URB)
- Thread Spawner (TS)
- Null function

The **Extended Math** function acts as an extension of the math functions already available inside the EUs. Certain functions such as inverse, square root, exponentiation, etc., require significant hardware resources to implement and are used infrequently enough that it is inefficient to implement them separately in each EU. The EUs therefore send the operands for these operations along with the operation to be performed to the Extended Math function which computes and returns the result to the requesting EU.

The **Sampling Engine** acts a (read-only) I/O port on behalf of the EUs, translating texture coordinates (and/or structure references) to memory addresses, reading texels and/or other data from memory, and in the case of texels, combining and filtering them according to programmed state. The resulting pixel and/or other data are then returned to the requesting EU.

The **Data Port** function acts as another I/O port on behalf of the EUs. It is both a read and a write port, and the only way for the Graphics Processing Engine to write results (e.g., images) back to memory. The Data Port contains the render and depth caches which receive the newly rendered pixels and write them out to memory when necessary. They also permit previously rendered objects to be read back efficiently by the Graphics Processing Engine in order to blend them with other rendered objects and test for visibility of newly rendered objects. Finally, the Data Port also provides read access constant buffers (arrays of constants in memory.)

The **Message Gateway** allows a thread to communicate (send a message to) another thread. A key is used to connect the sender and receiver threads, and a simple gateway protocol is used to send messages. This is primarily intended for media where a parent/child thread model is sometimes used and requires parent and child threads to synchronize and efficiently share information. It is not intended to be used by 3D graphics rendering threads.

The **Unified Return Buffer** (URB) is a single set of registers that EU threads use to return result data for future fixed functions and their threads to make use of. Individual entries in the buffer are “owned” by a given fixed function but a mechanism is provided where other fixed functions (those that follow) can read the data placed there by another fixed function. The buffer is considered a “Shared Function” since EUs need to be able to write result data to it using messages. In general, EU threads write their final results either to memory via the Data Port or to the URB for re-use by subsequent EU threads or certain 3D pipeline fixed-function units (CLIP, GS).

The **Thread Spawner** (TS) is a Shared Function that acts as a conduit for dispatching kernel-software-generated threads, one thread can request another thread to be dispatched by sending a request to the TS. TS is unique as it is also a Fixed Function in the media pipeline for dispatching threads originated from Video Front End fixed function.

The **Null** shared function is supported to allow the broadcast of certain information (e.g, End Of Thread) without invoking any other operation or response.

2.6 Messages

Communication between the EUs and the shared functions and between the fixed function pipelines (which are not considered part of the "Subsystem") and the EUs is accomplished via packets of information called *messages*. Message transmission is requested via the 'send' instruction. Refer to the 'send' instruction definition in the *ISA Reference* chapter for details.

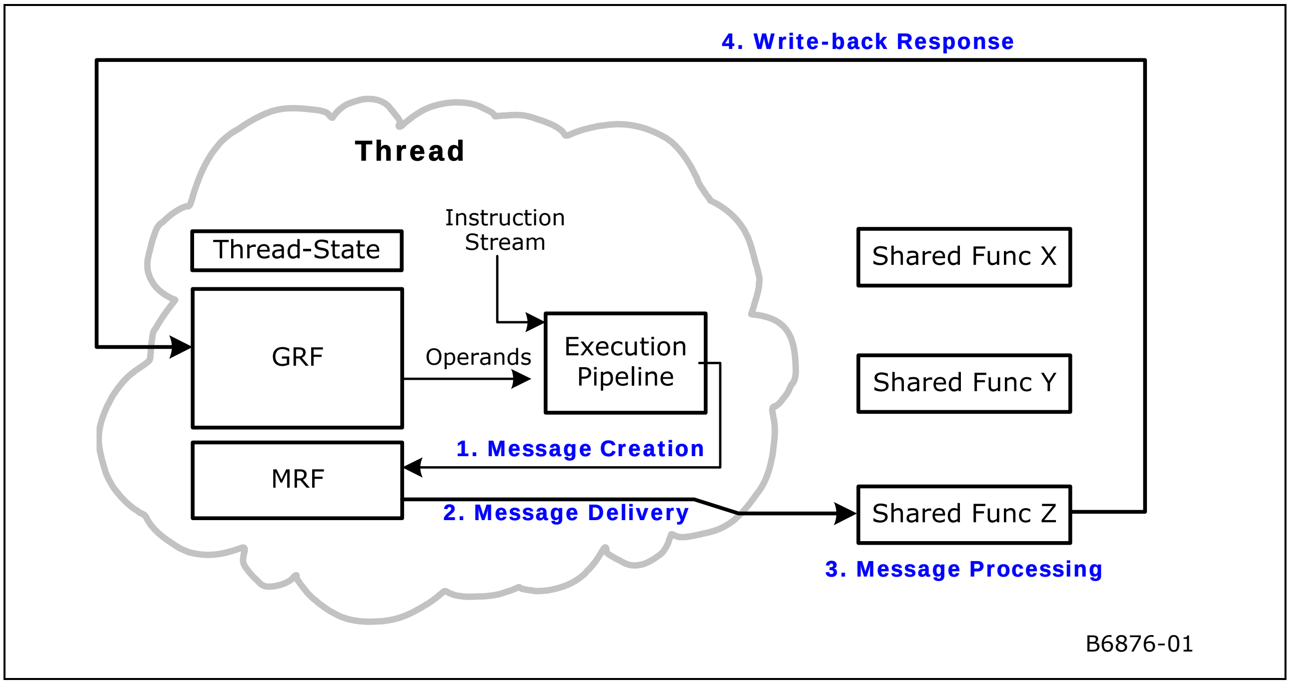
The information transmitted in a message falls into two categories:

- **Message Payload** data sourced from some number of registers (from 1 to 15 registers) in the Message Register File (MRF). The contents of the payload are dependent on the target function and specific function (etal), and may contain a header portion and/or data portion.
- Associated ("sideband") information provided by:
  - **Message Descriptor** specified with the 'send' instruction. Included in the message descriptor is control and routing information such as the target function ID, message payload length, response length, etc.
  - Additional information provided by the 'send' instruction, e.g., the starting destination register number, the execution mask (EMASK), etc.
  - A small subset of Thread State, such as the Thread ID, EUID, etc.

The software view of messages is shown in Figure 2-1. There are four basic phases to a message's lifetime as illustrated below:

1. Creation	The thread assembles the message payload into the Message Register File (MRF). This is done by a series of one or more instruction which specify a MRF register as the destination.
2. Delivery	The thread issues the message for delivery via the 'send' instruction. The 'send' instruction specifies the MRF register which is the first of a sequential register series which makes the data payload, the length of the message payload within the MRF, the destination shared function ID (SFID), and where in the GRF any response is to be directed. The messaging subsystem will enqueue the message for delivery and eventually route the message to the specified shared function.
3. Processing	The shared function receives the message and services it accordingly, as defined by the shared function definition.
4. Writeback	If called for, the shared function delivers an integral number of registers of data to the thread's GRF in response to the message.

Figure 2-1. Data Flow Associated With Messages



2.6.1 Message Register File (MRF)

Each thread has a dedicated MRF which is logically identical to the GRF: 256 bits wide per register, with word-wide addressability. There are 16 MRF registers, referred to as "m0".."m15". From a software perspective, the MRF is write-only and thus may only be used as a destination specifier. Limited register-region specifications are allowed so long as the region is contained within a single MRF register.

Each register of the MRF has an associated in-flight status, indicating the contents of the register is needed as part of a pending message, but has yet to be transmitted by the hardware. This bit is set at the time the message is enqueued for delivery via the 'send' instruction. Should a subsequent write to an in-flight register be attempted, the execution unit will temporarily suspend the thread's execution until the register's in-flight status is cleared (i.e., the message has been transmitted).

Register m0 is reserved for System Routine (exception handling) purposes, thus normal threads should construct their messages in m1..m15. The thread is free to start a message payload at any MRF register location, even to the point of having multiple messages under construction at the same time in non-overlapping spaces in the MRF. Further multiple messages over non-overlapping MRF space can be enqueued awaiting transmission at the same time. Regardless of actual hardware implementation, the thread should not assume that MRF addresses above m15 wrap to legal MRF registers.

2.6.2 Send Instruction

Messages are sent programmatically by the thread through the 'send' instruction. This instruction enqueues a message for delivery and marks as in-flight all MRF registers used for the message payload. It also allows for an optional implied move of one GRF register to a MRF register prior to the message being issued. This implied move allows for a higher message performance, eliminating the explicit 'mov' that would normally be required to move R0 to the lead MRF register of the message (as required by many message definitions).

A typical 'send' instruction is exemplified here (please see the ISA for a full instruction description). This example performs an implicit move from r0 to m3, then issues a message to the Extended Math unit, with a payload of 1 register starting at m3, and expecting 1 register in reply to be placed in r5.

```
send (16) r5 m3 r0 0x01110001
```

The execution unit guarantees that any prior instruction which wrote to a MRF register is guaranteed to have retired, and its result written to the destination MRF register in time for message transmission.

2.6.3 Creating and Sending a Message

A code snippet is listed below, showing a 4-register message (m3 to m6) whose response is directed to r30. Note that message construction does not have to occur in MRF register order.

```
...
mul (8)   m4    r20     r19
mov (8)   m6    r21
add (8)   m5    r29     r28
send (8)  r30   m3      r0   <desc>
...
```

Once a 'send' instruction is issued, the MRF registers used for its payload are marked as 'in-flight'. These registers remain in this state until the message is actually transmitted to the shared function and the register contents are no longer need. Any subsequent write to a MRF register which is in-flight results in a dependency and a thread switch until such time that the in-flight condition is cleared. An example is shown below in which the attempt to re-use m6 may result in a thread switch until message 1 is transmitted.

```
...
// --- message 1 ---
mul (8)   m4    r20     r19
mov (8)   m6    r21
add (8)   m5    r29     r28
send (8)  r30   m3      r0   <desc>
...

// --- message 2 ---
mov (8)   m6    r15     // thread switch until the
                        // previous msg is sent and
                        // m6 in-flight is cleared.
...
```

MRF registers of one message may be reused for a subsequent message without restriction. The in-flight check mechanism prevents a MRF register staged as part of a pending message from being altered while awaiting transmission. Further, a thread may rely on the contents of a MRF register being unaltered after message transmission. This allows the thread to quickly issue an identical or slightly altered message using the same MRF register set without having to re-construct the entire payload.

Although more than one message may be enqueued at any point in time, care must be taken by the programmer to ensure that each message's destination GRF register region, if any, does no over lap with that of another enqueued message. This condition is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles in the current implementation, the outcome in the GRF is indeterminate; It may be the result from the first message, or the result from the second message, or a mixture of data from both.

2.6.4 Message Payload Containing a Header

For most shared functions, the first register of the message payload contains the *header payload* of the message (or simply the *message header*). It contains the state fields (such as binding table pointer, sampler state pointer, etc.) following a consistent format structure. Consequently, the rest of the message payload is referred to as the *data payload*.

Messages to Extended Math do not have a header and only contain data payload. Those messages may be referred to as header-less messages. Messages to Gateway combine the header and data payloads in a single message register.

2.6.5 Writebacks

Some messages generate return data as dictated by the 'function-control' (opcode) field of the 'send' instruction (part of the <desc> field). The Gen4 execution unit and message passing infrastructure do not interpret this field in any way to determine if writeback data is to be expected. Instead explicit fields in the 'send' instruction to the execution unit the starting GRF register and count of returning data. The execution unit uses this information to set in-flight bits on those registers to prevent execution of any instruction which uses them as an operand until the register(s) is(are) eventually written in response to the message. If a message is not expected to return data, the 'send' instruction's writeback destination specifier (<post_dest>) must be set to 'null' and the response length field of <desc> must be 0 (see 'send' instruction for more details).

The writeback data, if called for, arrives as a series of register writes to the GRF at the location specified by the starting GRF register and length as specified in the 'send' instruction. As each register is written back to the GRF, its in-flight flag is cleared and it becomes available for use as an instruction operand. If a thread was suspended pending return of that register, the dependency is lifted and the thread is allowed to continue execution (assuming no other dependency for that thread remains outstanding).

2.6.6 Message Delivery Ordering Rules

All messages between a thread and an individual shared function are delivered in the ordered they were sent. Messages to different shared functions originating from a single thread may arrive at their respective shared functions out of order.

The writebacks of various messages from the shared functions may return in any order. Further individual destination registers resulting from a single message may return out of order, potentially allowing execution to continue before the entire response has returned (depending on the dependency chain inherent in the thread).

2.6.7 Execution Mask and Messages

The Gen4 Architecture defines an Execution Mask (EMask) for each instruction issued. This 16b bit-field identifies which SIMD computation channels are enabled for that instruction. Since the ‘send’ instruction is inherently scalar, the EMask is ignored as far as instruction dispatch is concerned. Further the execution size has no impact on the size of the ‘send' instruction’s implicit move (it is always 1 register regardless of specified execution size).

The 16b EMask is forwarded with the message to the destination shared function to indicate which SIMD channels were enabled at the time of the ‘send’. A shared function may interpret or ignore this field as dictated by the functionality it exposes. For instance, the Extended Math shared function observes this field and performs the specified operation only on the operands with enabled channels, while the DataPort writes to the render cache ignore this field completely, instead using the pixel mask included in-band in the message payload to indicate which channels carry valid data.

2.6.8 End-Of-Thread (EOT) Message

The final instruction of all threads must be a ‘send’ instruction which signals ‘End-Of-Thread’ (EOT). An EOT message is one in which the EOT bit is set in the ‘send’ instruction’s 32b <desc> field. When issuing instructions, the EU looks for an EOT message, and when issued, shuts down the thread from further execution and considers the thread completed.

Only a subset of the shared functions can be specified as the target function of an EOT message, as shown in the table below.

Target Shared Functions supporting EOT messages	*Target Shared Functions not supporting EOT messages*
DataPortWrite, URB, MessageGateway, ThreadSpawner	*DataPortRead, Sampler*

Both the fixed-functions and the thread dispatcher require EOT notification at the completion of each thread. The thread dispatcher and fixed functions in the 3D pipeline obtain EOT notification by snooping all message transmissions, regardless of the explicit destination, looking for messages which signal end-of-thread. The Thread Spawner in the media pipeline does not snoop for EOT. As it is also a shared function, all threads generated by Thread Spawner must send a message to Thread Spawner to explicity signal end-of-thread.

The thread dispatcher, upon detecting an end-of-thread message, updates its accounting of resource usage by that thread, and is free to issue a new thread to take the place of the ended thread. Fixed functions require end-of-thread notification to maintain accounting as to which threads it issued have completed and which remain outstanding, and their associated resources such as URB handles.

Unlike the thread dispatcher, fixed-functions discriminate end-of-thread messages, only acting upon those from threads which they originated, as indicated by the 4b fixed-function ID present in R0 of end-of-thread message payload. This 4b field is attached to the thread at new-thread dispatch time and is placed in its designated field in the R0 contents delivered to the GRF. Thus to satisfy the inclusion of the fixed-function ID, the typical end-of-thread message generally supplies R0 from the GRF as the first register of an end-of-thread message.

As an optimization, an end-of-thread message may be overload upon another "productive" message, saving the cost in execution and bandwidth of a dedicated end-of-thread message. Outside of the end-of-thread message, most threads issue a message just prior to their termination (for instance, a Dataport write to the framebuffer) so the overloaded end-of-thread is the common case. The requirement is that the message contains R0 from the GRF (to supply the fixed-function ID), and that destination shared function be either (a) the URB; (b) the Read or Write Dataport; or, (c) the Gateway, as these functions reside on the O-Bus. In the case where the last real message of a thread is to some other shared function, the thread must issue a separate message for the purposes of signaling end-of-thread to the "null" shared function.

2.6.9 Performance

The Gen4 Architecture imposes no requirement as to a shared function's latency or throughput. Due to this as well as factors such as message queuing, shared bus arbitration, implementation choices in bus bandwidth, and instantaneous demand for that function, the latency in delivering and obtaining a response to a message is non-deterministic. It is expected that a Gen4 implementation has some notion of fairness in transmission and servicing of messages so as to keep latency outliers to a minimum.

Other factors to consider with regard to performance:

- A thread may choose to have multiple messages under construction in non-overlapping registers the MRF at the same time.
- Multiple messages are allowed to be enqueued for transmission at the same time, so long as their MRF payload registers do not overlap.
- Messages may rely on the MRF registers being maintained across a send message, thus constructing subsequent messages overlaid on portions of a previous message,
- Software prefetching techniques may be beneficial for long latency data fetches (i.e. issue a load early in the thread for data that is required late in the thread).

2.6.10 Message Description Syntax

All message formats are defined in terms of DWords (32 bits). The message registers in all cases are 256 bits wide, or 8 DWords. The registers and DWords within the registers are named as follows, where n is the register number, and d is the DWord number from 0 to 7, from the least significant DWord at bits [31:0] within the 256-bit register to the most significant DWord at bits [255:224], respectively. For writeback messages, the register number indicates the offset from the specified starting destination register.

Dispatch Messages: **R**n.d

Dispatch messages are sent by the fixed functions to dispatch threads. See the fixed function chapters in the *3D and Media* volume.

SEND Instruction Messages: **M**n.d

These are the messages initiated by the thread via the SEND instruction to access shared functions. See the chapters on the shared functions later in this volume.

Writeback Messages: **W**n.d

These messages return data from the shared function to the GRF where it can be accessed by thread that initiated the message.

The bits within each DWord are given in the second column in each table.

2.6.11 Message Errors

Messages are constructed via software, and not all possible bit encodings are legal, thus there is the possibility that a message may be sent containing one or more errors in its descriptor or payload contents. There are two points of error detection in the message passing system: (a) the message delivery subsystem is capable of detecting bad FunctionIDs and some cases of bad message lengths; (b) the shared functions contain various error detection mechanisms which identify bad sub-function codes, bad message lengths, and other misc errors. The error detection capabilities are specific to each shared function. The execution unit hardware itself does not perform message validation prior to transmission.

In both cases, information regarding the erroneous message is captured and made visible through MMIO registers, and the driver notified via an interrupt mechanism. The set of possible errors is listed in Table 2-1 with the associated outcome. Please see the chapter on error handling for detailed information.

Table 2-1. Error Cases

Error	Outcome
Bad Shared Function ID	The message is discarded before reaching any shared function. If the message specified a destination, those registers will be marked as in-flight, and any future usage by the thread of those registers will cause a dependency which will never clear, resulting in a hung thread and eventual time-out.
Unknown opcode Incorrect message length	The destination shared function detects unknown opcodes (as specified in the 'send' instructions <desc> field), and known opcodes where the message payload is either too long or too short, and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through and interrupt mechanism (details tbd), then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.
Incorrect response length	Case: too few registers specified – the thread may proceed with execution prior to all the data returning from the shared function, resulting in the thread operating on bad data in the GRF. Case: too many registers specified – the message response does not clear all the registers of the destination. In this case, if the thread references any of the residual registers, it may hand and result in an eventual time-out.

Error	Outcome
Improper use of End-Of-Thread (EOT)	Any 'send' instruction which specifies EOT must have a 'null' destination register. The EU enforces this and, if detected, will not issue the 'send' instruction, resulting in a hung thread and an eventual time-out. The 'send' instruction specifies that EOT is only recognized if the <desc> field of the instruction is an immediate. Should a thread attempt to end a thread using a <desc> sourced from a register, the EOT bit will not be recognized. In this case, the thread will continue to execute beyond the intended end of thread, resulting in a wide range of error conditions.
Two outstanding messages using overlapping GRF destinations ranges	This is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles, the outcome in the GRF is indeterminate; may be the result from the first message, or the result from the second message, or a combination of both.

3. Shared Functions

This volume includes all the GEN4 shared function chapters (Sampler, DataPort, ExtendedMath, MessageGateway, URB), which are described in the following sections.

4. Sampling Engine

The Sampling Engine provides the capability of advanced sampling and filtering of surfaces in memory.

The sampling engine function is responsible for providing filtered texture values to the Gen4 Core in response to sampling engine messages.. The sampling engine uses SAMPLER_STATE to control filtering modes, address control modes, and other features of the sampling engine. A pointer to the sampler state is delivered with each message, and an index selects one of 16 states pointed to by the pointer. Some messages do not require SAMPLER_STATE. In addition, the sampling engine uses SURFACE_STATE to define the attributes of the surface being sampled. This includes the location, size, and format of the surface as well as other attributes.

Although data is commonly used for “texturing” of 3D surfaces, the data can be used for any purpose once returned to the execution core.

The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the Gen4 Core in order to complete the *sample* instruction.

Subfunction	Description
Texture Coordinate Processing	Any required operations are performed on the incoming pixel’s interpolated internal texture coordinates. These operations may include: cube map intersection.
Texel Address Generation	The Sampling Engine will determine the required set of texel samples (specific texel values from specific texture maps), as defined by the texture map parameters and filtering modes. This includes coordinate wrap/clamp/mirror control, mipmap LOD computation and sample and/or miplevel weighting factors to be used in the subsequent filtering operations.
Texel Fetch	The required texel samples will be read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.
Texture Palette Lookup	For streams which have “paletted” texture surface formats, this function uses the “index” values read from the texture map to look up texel color data from the texture palette.
Shadow Pre-Filter Compare	For shadow mapping, the texel samples are first compared to the 3rd (R) component of the pixel’s texture coordinate. The boolean results are used in the texture filter.
Texel Filtering	Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This “combination” ranges from simply passing through a “nearest” sample to blending the results of anisotropic filters performed on two mipmap levels. The output of this function is a single 4-component texel value.
Texel Color Gamma Linearization	Performs optional gamma decorrection on texel RGB (not A) values.

Subfunction	Description
Denoise/ Deinterlacer	Performs denoise and deinterlacing functions for video content (**[DevILK+]**)
8x8 Video Scaler	Performs scaling using an 8x8 filter (**[DevILK+]**)
Image Enhancement Filter / Video Signal Analysis	Image Enhancement functions for video content (**[DevILK+]**)

4.1 Texture Coordinate Processing

The Texture Coordinate Processing function of the Sampling Engine performs any operations on the texture coordinates that are required before physical addresses of texel samples can be generated.

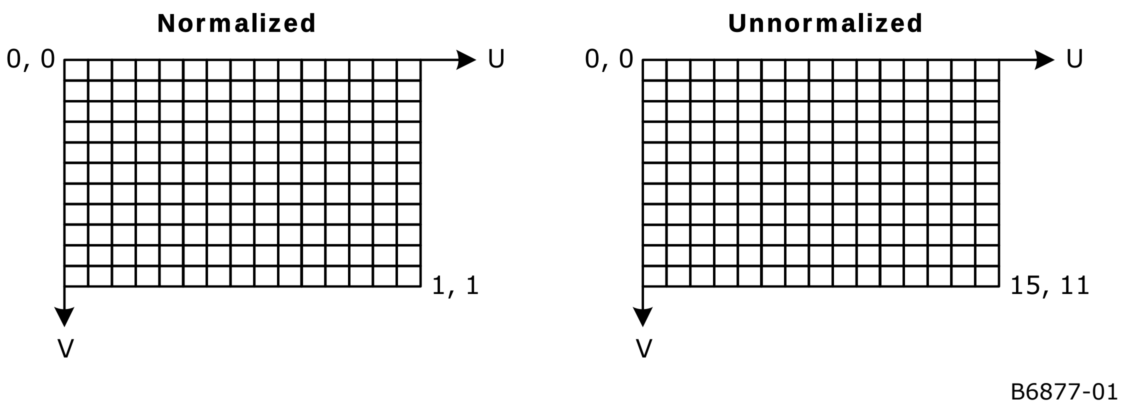
4.1.1 Texture Coordinate Normalization

A texture coordinate may have *normalized* or *unnormalized* values. In this function, unnormalized coordinates are normalized.

Normalized coordinates are specified in units relative to the map dimensions, where the origin is located at the upper/left edge of the upper left texel, and the value 1.0 coincides with the lower/right edge of the lower right texel . 3D rendering typically utilizes normalized coordinates.

Unnormalized coordinates are in units of texels and have not been divided (normalized) by the associated map's height or width. Here the origin is the located at the upper/left edge of the upper left texel of the base texture map. Unnormalized coordinates delivered to the sampling engine are only supported with the "ld" type messages.

Figure 4-1. Normalized vs. Unnormalized Texture Coordinates



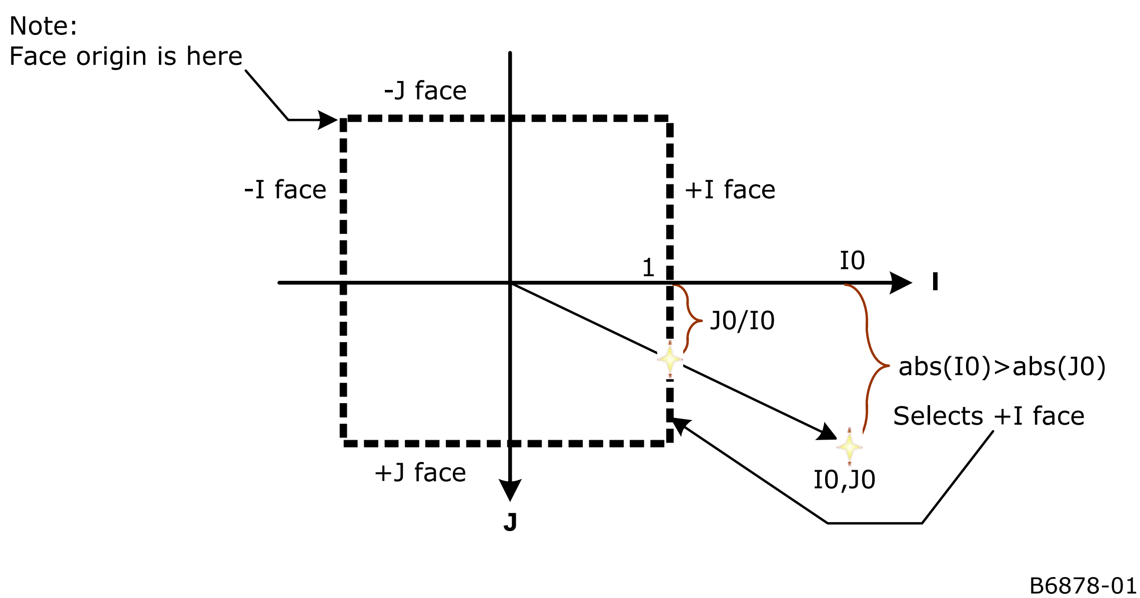
4.1.2 Texture Coordinate Computation

Cartesian (2D) and homogeneous (projected) texture coordinate values are projected from (interpolated) screen space back into texture coordinate space by dividing the pixel's S and T components by the Q component. This operation is done as part of the pixel shader kernel in the Gen4 Core.

Vector (cube map) texture coordinates are generated by first determining which of the 6 cube map faces (+X, +Y, +Z, -X, -Y, -Z) the vector intersects. The vector component (X, Y or Z) with the largest absolute value determines the proper (major) axis, and then the sign of that component is used to select between the two faces associated with that axis. The coordinates along the two minor axes are then divided by the coordinate of the major axis, and scaled and translated, to obtain the 2D texture coordinate ([0,1]) within the chosen face. Note that the coordinates delivered to the sampling engine must already have been divided by the component with the largest absolute value.

An illustration of this cube map coordinate computation, simplified to only two dimensions, is provided below:

Figure 4-2. Cube Map Coordinate Computation Example



4.2 Texel Address Generation

To better understand texture mapping, consider the mapping of each object (screen-space) pixel onto the textures images. In texture space, the pixel becomes some arbitrarily sized and aligned quadrilateral. Any given pixel of the object may "cover" multiple texels of the map, or only a fraction of one texel. For each pixel, the usual goal is to sample and filter the texture image in order to best represent the covered texel values, with a minimum of blurring or aliasing artifacts. Per-texture state variables are provided to allow the user to employ quality/performance/footprint tradeoffs in selecting how the particular texture is to be sampled.

The Texel Address Generation function of the Sampling Engine is responsible for determining how the texture maps are to be sampled. Outputs of this function include the number of texel samples to be taken, along with the physical addresses of the samples and the filter weights to be applied to the samples after they are read. This information is computed given the incoming texture coordinate and gradient values, and the relevant state variables associated with the sampler and surface. This function also applies the texture coordinate address controls when converting the sample texture coordinates to map addresses.

4.2.1 Level of Detail Computation (Mipmapping)

Due to the specification and processing of texture coordinates at object vertices, and the subsequent object warping due to a perspective projection, the texture image may become *magnified* (where a texel covers more than one pixel) or *minified* (a pixel covers more than one texel) as it is mapped to an object. In the case where an object pixel is found to cover multiple texels (texture minification), merely choosing one (e.g., the texel sample nearest to the pixel's texture coordinate) will likely result in severe aliasing artifacts.

Mipmapping and texture filtering are techniques employed to minimize the effect of undersampling these textures. With mipmapping, software provides *mipmap levels,* a series of pre-filtered texture maps of decreasing resolutions that are stored in a fixed (monolithic) format in memory. When mipmaps are provided and enabled, and an object pixel is found to cover multiple texels (e.g., when a textured object is located a significant distance from the viewer), the device will sample the mipmap level(s) offering a texel/pixel ratio as close to 1.0 as possible.

The device supports up to 14 mipmap levels per map surface, ranging from 8192 x 8192 texels to a 1 X 1 texel. Each successive level has ½ the resolution of the previous level in the U and V directions (to a minimum of 1 texel in either direction) until a 1x1 texture map is reached. The dimensions of mipmap levels need not be a power of 2.

Each mipmap level is associated with a *Level of Detail (LOD)* number. LOD is computed as the approximate, $\log_2$ measure of the ratio of texels per pixel. The highest resolution map is considered LOD 0. A larger LOD number corresponds to lower resolution mip level.

The *Sampler[]BaseMipLevel* state variable specifies the LOD value at which the minification filter vs. the magnification filter should be applied.

When the texture map is magnified (a texel covers more than one pixel), the base map (LOD 0) texture map is accessed, and the magnification mode selects between the nearest neighbor texel or bilinear interpolation of the 4 neighboring texels on the base (LOD 0) mipmap.

4.2.1.1 Base Level Of Detail (LOD)

The per-pixel LOD is computed in an implementation-dependent manner and approximates the $\log_2$ of the texel/pixel ratio at the given pixel. The computation is typically based on the differential texel-space distances associated with a one-pixel differential distance along the screen x- and y-axes. These texel-space distances are computed by evaluating neighboring pixel texture coordinates, these coordinates being in units of texels on the base MIP level (multiplied by the corresponding surface size in texels). The q coordinates represent the third dimension for 3D (volume) surfaces, this coordinate is a constant 0 for 2D surfaces.

The ideal LOD computation is included below.

$$LOD(x,y) = \log_2[\rho(x,y)]$$

where :

$$\rho(x,y) = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial q}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial q}{\partial y}\right)^2}\right\},$$

4.2.1.2 LOD Bias

A biasing offset can be applied to the computed LOD and used to artificially select a higher or lower miplevel and/or affect the weighting of the selected mipmap levels. Selecting a slightly higher mipmap level will trade off image blurring with possibly increased performance (due to better texture cache reuse). Lowering the LOD tends to sharpen the image, though at the expense of more texture aliasing artifacts.

The LOD bias is defined as sum of the *LODBias* state variable and the *pixLODBias* input from the input message (which can be non-zero only for sample_b messages). The application of LOD Bias is unconditional, therefore these variables must both be set to zero in order to prevent any undesired biasing.

Note that, while the LOD Bias is applied prior to clamping and min/mag determination and therefore can be used to control the min-vs-mag crossover point, its use has the undesired effect of actually changing the LOD used in texture filtering.

4.2.1.3 LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable. Enabling pre-clamping matches OpenGL semantics, while disabling it matches Direct3D.

After biasing and/or adjusting of the LOD , the computed LOD value is clamped to a range specified by the (integer and fractional bits of) *MinLOD* and *MaxLOD* state variables prior to use in Min/Mag Determination.

MaxLOD specifies the lowest resolution mip level (maximum LOD value) that can be accessed, even when lower resolution maps may be available. Note that this is the only parameter used to specify the number of valid mip levels that be can be accessed, i.e., there is no explicit “number of levels stored in memory” parameter associated with a mip-mapped texture. All mip levels from the base mip level map through the level specified by the integer bits of *MaxLOD* must be stored in memory, or operation is UNDEFINED.

MinLOD specifies the highest resolution mip level (minimum LOD value) that can be accessed, where LOD==0 corresponds to the base map. This value is primarily used to deny access to high-resolution mip levels that have been evicted from memory when memory availability is low.

MinLOD and *MaxLOD* have both integer and fractional bits. The fractional parts will limit the inter-level filter weighting of the highest or lowest (respectively) resolution map. For example if *MinLOD* is 4.5 and *MipFilter* is LINEAR, LOD 4 can contribute only up to 50% of the final texel color.

4.2.1.4 Min/Mag Determination

The biased and clamped LOD is used to determine whether the texture is being minified (scaled down) or magnified (scaled up).

The *BaseMipLevel* state variable is subtracted from the biased and clamped LOD. The *BaseMipLevel* state variable therefore has the effect of selecting the "base" mip level used to compute Min/Map Determination. (This was added to match OpenGL semantics). Setting *BaseMipLevel* to 0 has the effect of using the highest-resolution mip level as the base map.

If the biased and clamped LOD is non-positive, the texture is being magnified, and a single (high-resolution) miplevel will be sampled and filtered using the *MagFilter* state variable. At this point the computed LOD is reset to 0.0. Note that LOD Clamping can restrict access to high-resolution miplevels.

If the biased LOD is positive, the texture is being minified. In this case the *MipFilter* state variable specifies whether one or two mip levels are to be included in the texture filtering, and how that (or those) levels are to be determined as a function of the computed LOD.

4.2.1.5 LOD Computation Pseudocode

This section illustrates the LOD biasing and clamping computation in pseudocode, encompassing the steps described in the previous sections. The computation of the initial per-pixel LOD value *LOD* is not shown.

```
[if (sample_b)
        LOD += Bias + bias_parameter
else if (sample_l or ld)
        LOD = Bias + lod_parameter
else
        LOD += Bias

If (PreClamp)
        LOD = min(LOD, MaxLod)
        LOD = max(LOD, MinLod)

MagMode = (LOD - Base <= 0)
If (MagMode or MipFlt = None)
        LOD = 0
        LOD = min(LOD, ceil(MaxLod))
        LOD = max(LOD, floor(MinLod))
else if (MipFlt = Nearest)
        LOD = min(LOD, ceil(MaxLod))
        LOD = max(LOD, floor(MinLod))
        LOD = floor(LOD)
else    // MipFlt = Linear
        LOD = min(LOD, MaxLod)
        LOD = max(LOD, MinLod)
        TriBeta = frac(LOD)
        LOD0 = floor(LOD)
        LOD1 = LOD0 + 1

Lod += SurfMinLod
```

If Out_of_Bounds is true, LOD is set to zero and instead of sampling the surface the texels are replaced with zero in all channels, except for surface formats that don't contain alpha, for which the alpha channel is replaced with one. These texels then proceed through the rest of the pipeline.

4.2.2 Inter-Level Filtering Setup

The *MipFilter* state variable determines if and how texture mip maps are to be used and combined. The following table describes the various mip filter modes:

***MipFilter* Value**	**Description**
MIPFILTER_NONE	Mipmapping is DISABLED. Apply a single filter on the highest resolution map available (after LOD clamping).
MIPFILTER_NEAREST	Choose the nearest mipmap level and apply a single filter to it. Here the biased LOD will be rounded to the nearest integer to obtain the desired miplevel. LOD Clamping may further restrict this miplevel selection.
MIPFILTER_LINEAR	Apply a filter on the two closest mip levels and linear blend the results using the distance between the computed LOD and the level LODs as the blend factor. Again, LOD Clamping may further restrict the selection of miplevels (and the blend factor between them).

When minifying and MIPFILTER_NEAREST is selected, the computed LOD is rounded to the nearest mip level.

When minifying and MIPFILTER_LINEAR is selected, the fractional bits of the computed LOD are used to generate an inter-level blend factor. The LOD is then truncated. The mip level selected by the truncated LOD, and the next higher (lower resolution) mip level are determined.

Regardless of *MipFilter* and the min/mag determination, all computed LOD values (two for MIPFILTER_LINEAR, otherwise one) are then unconditionally clamped to the range specified by the (integer bits of) *MinLOD* and *MaxLOD* state variables.

4.2.3 Intra-Level Filtering Setup

Depending on whether the texture is being minified or magnified, the *MinFilter* or *MagFilter* state variable (respectively) is used to select the sampling filter to be used within a mip level (intra-level, as opposed to any inter-level filter). Note that for volume maps, this selection also applies to filtering between layers.

The processing at this stage is restricted to the selection of the filter type, computation of the number and texture map coordinates of the texture samples, and the computation of any required filter parameters. The filtering of the samples occurs later on in the Sampling Engine function.

The following table summarizes the intra-level filtering modes.

Sampler[]Min/MagFilter value	Description
MAPFILTER_NEAREST	Supported on all surface types. The texel nearest to the pixel's U,V,Q coordinate is read and output from the filter.
MAPFILTER_LINEAR	Not supported on buffer surfaces. The 2, 4, or 8 texels (depending on 1D, 2D/CUBE, or 3D surface, respectively) surrounding the pixel's U,V,Q coordinate are read and a linear filter is applied to produce a single filtered texel value.
MAPFILTER_ANISOTROPIC	Not supported on buffer or 3D surfaces. A projection of the pixel onto the texture map is generated and "subpixel" samples are taken along the major axis of the projection (center axis of the longer dimension). The outermost subpixels are weighted according to closeness to the edge of the projection, inner subpixels are weighted equally. Each subpixel samples a bilinear 2x2 of texels and the results are blended according to weights to produce a filtered texel value.
MAPFILTER_MONO	Supported only on 2D surfaces. This filter is only supported with the monochrome (MONO8) surface format. The monochrome texel block of the specified size surrounding the pixel is selected and filtered.

4.2.3.1 MAPFILTER_NEAREST

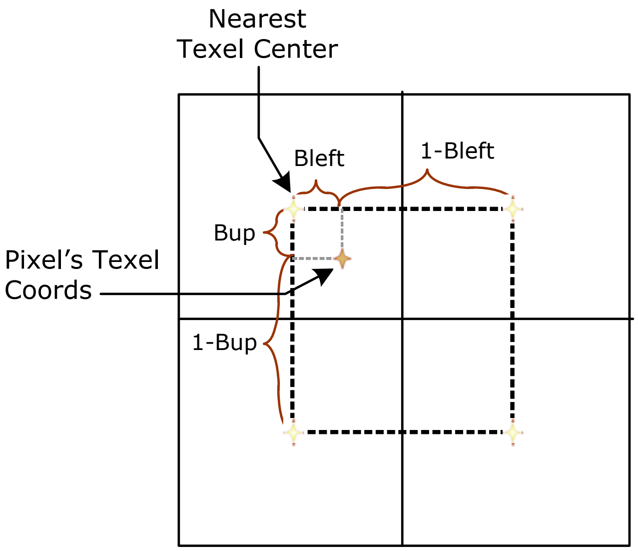
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level.

4.2.3.2 MAPFILTER_LINEAR

The following description indicates behavior of the MIPFILTER_LINEAR filter for 2D and CUBE surfaces. 1D and 3D surfaces follow a similar method but with a different number of dimensions available.

When the MAPFILTER_LINEAR filter is selected on a 2D surface, the 2x2 region of texels surrounding the pixel's texture coordinate are sampled and later bilinearly filtered.

Figure 4-3. Bilinear Filter Sampling



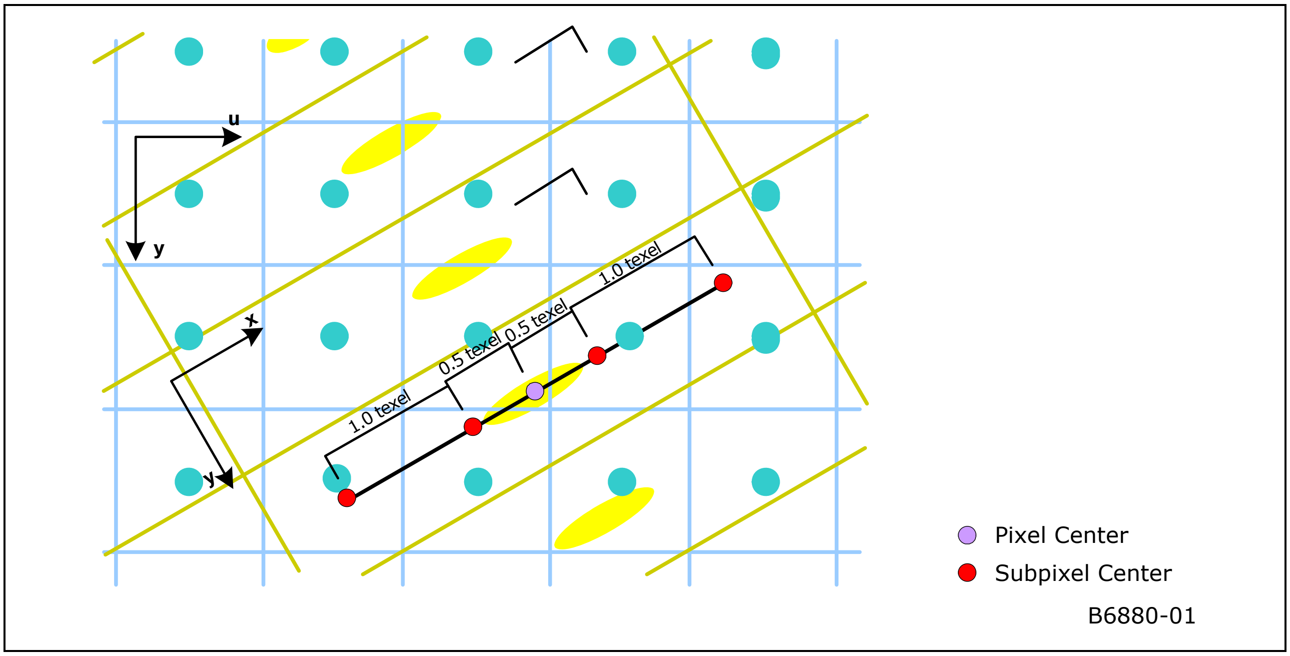
B6879-01

The four texels surrounding the pixel center are chosen for the bilinear filter. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

4.2.3.3 MAPFILTER_ANISOTROPIC

The MAPFILTER_ANISOTROPIC texture filter attempts to compensate for the anisotropic mapping of pixels into texture map space. A possibly non-square set of texel sample locations will be sampled and later filtered. The *MaxAnisotropy* state variable is used to select the maximum aspect ratio of the filter employed, up to 16:1.

The algorithm employed first computes the major and minor axes of the pixel projection onto the texture map. LOD is chosen based on the minor axis length in texel space. The anisotropic "ratio" is equal to the ratio between the major axis length and the minor axis length. The next larger even integer above the ratio determines the anisotropic number of "ways", which determines how many subpixels are chosen. A line along the major axis is determined, and "subpixels" are chosen along this line, spaced one texel apart, as shown in the diagram below. In this diagram, the texels are shown in light blue, and the pixels are in yellow.



Each subpixel samples a bilinear 2x2 around it just as if it was a single pixel. The result of each subpixel is then blended together using equal weights on all interior subpixels (not including the two endpoint subpixels). The endpoint subpixels have lesser weight, the value of which depends on how close the “ratio” is to the number of “ways”. This is done to ensure continuous behavior in animation.

4.2.3.4 MAPFILTER_MONO

When the MAPFILTER_MONO filter is selected, a block of monochrome texels surrounding the pixel sample location are read and filtered using the kernel described below. The size of this block is controlled by **Monochrome Filter Height** and **Width** (referred to here as N_v and N_u, respectively) state. Filters from 1x1 to 7x7 are supported (not necessarily square).

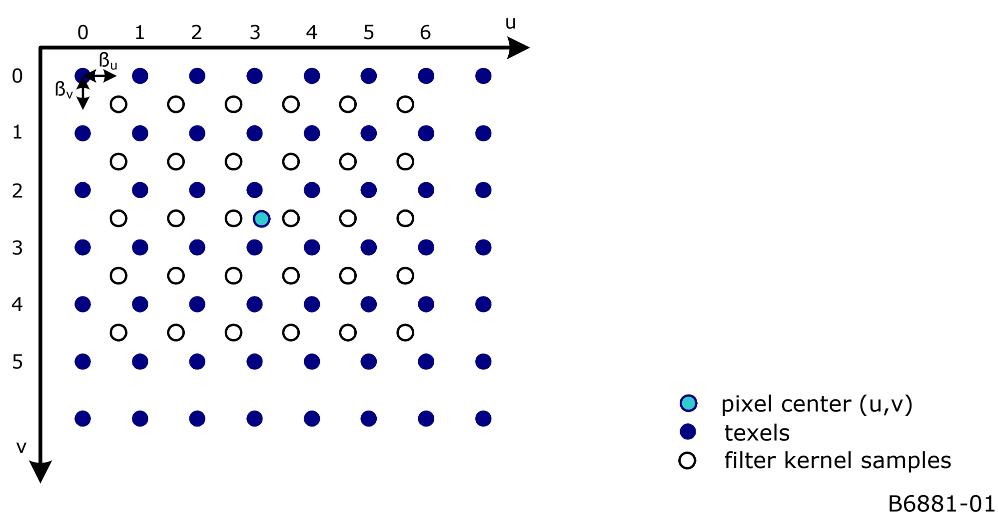
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f, v_f) relative to the pixel center at (u, v) is given by the following:

$$u_f = u - \frac{N_u}{2}$$

$$v_f = v - \frac{N_v}{2}$$

β_u and β_v are the fractional parts of u_f and v_f, respectively. The integer parts select the upper left texel for the kernel filter, given here as $T_{0,0}$.

Figure 4-4. Sampling Using MAPFILTER_MONO



The formula for the final filter output F is given by the following. Since this is a monochrome filter, each texel value (T) is a single bit, and the output F is an intensity value that is replicated across the color and alpha channels.

$$S = \frac{1}{N_u * N_v}$$

$$F = \left[(1-\beta_u)(1-\beta_v)\sum_{i=0}^{N_u-1}\sum_{j=0}^{N_v-1} T_{i,j} + \beta_u(1-\beta_v)\sum_{i=1}^{N_u}\sum_{j=0}^{N_v-1} T_{i,j} + (1-\beta_u)\beta_v \sum_{i=0}^{N_u-1}\sum_{j=1}^{N_v} T_{i,j} + \beta_u\beta_v \sum_{i=1}^{N_u}\sum_{j=1}^{N_v} T_{i,j} \right] * S$$

4.2.4 Texture Address Control

The *[TCX,TCY,TCZ]ControlMode* state variables control the access and/or generation of texel data when the specific texture coordinate component falls outside of the normalized texture map coordinate range [0,1).

Note: For **Wrap Shortest** mode, the setup kernel has already taken care of correctly interpolating the texture coordinates. Software will need to specify TEXCOORDMODE_WRAP mode for the sampler that is provided with wrap-shortest texture coordinates, or artifacts may be generated along map edges.

TC[X,Y,Z] Control	Operation
TEXCOORDMODE_CLAMP	Clamp to the texel value at the edge of the map.
TEXCOORDMODE_CLAMP_BORDER	Use the texture map's border color for any texel samples falling outside the map. The border color is specified via a pointer in SAMPLER_STATE.
TEXCOORDMODE_WRAP	Upon crossing an edge of the map, repeat at the other side of the map in the same dimension.
TEXCOORDMODE_CUBE	Only used for cube maps. Here texels from adjacent cube faces can be sampled along the edges of faces. This is considered the highest quality mode for cube environment maps.
TEXCOORDMODE_MIRROR	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.
TEXCOORDMODE_MIRROR_ONCE	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.

Separate controls are provided for texture TCX, TCY, TCZ coordinate components so, for example, the TCX coordinate can be wrapped while the TCY coordinate is clamped. Note that there are no controls provided for the TCW component as it is only used to scale the other 3 components before addressing modes are applied.

Maximum Wraps/Mirrors

The number of map wraps on a given object is limited to 32. Going beyond this limit is legal, but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the texel level (choosing the wrong texels for filtering).

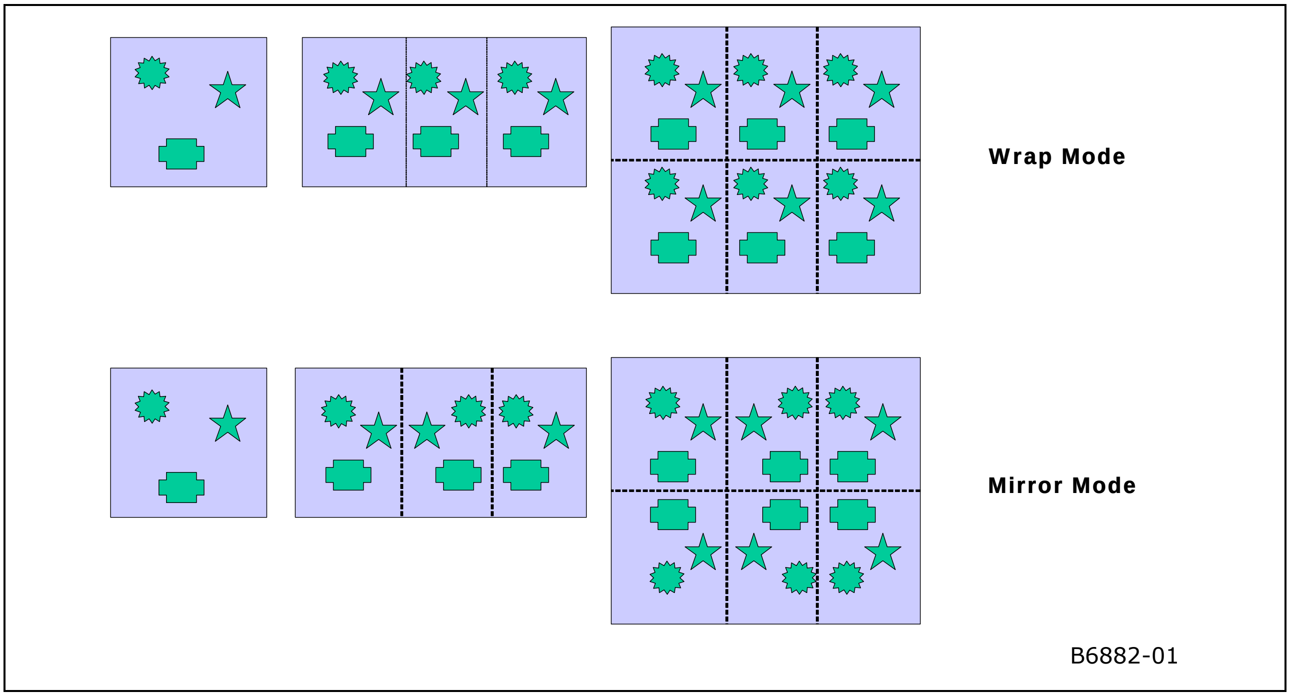
4.2.4.1 TEXCOORDMODE_WRAP Mode

In TEXCOORDMODE_WRAP addressing mode, the integer part of the texture coordinate is discarded, leaving only a fractional coordinate value. This results in the effect of the base map ([0,1)) being continuously repeated in all (axes-aligned) directions. Note that the interpolation between coordinate values 0.1 and 0.9 passes through 0.5 (as opposed to WrapShortest mode which interpolates through 0.0).

4.2.4.2 TEXCOORDMODE_MIRROR Mode

TEXCOORDMODE_MIRROR addressing mode is similar to Wrap mode, though here the base map is flipped at every integer junction. For example, for U values between 0 and 1, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on. The second row of pictures in the figure below indicate a map that is mirrored in one direction and then both directions. You can see that in the mirror mode every other integer map wrap the base map is mirrored in either direction.

Figure 4-5. Texture Wrap vs. Mirror Addressing Mode



4.2.4.3 TEXCOORDMODE_MIRROR_ONCE Mode

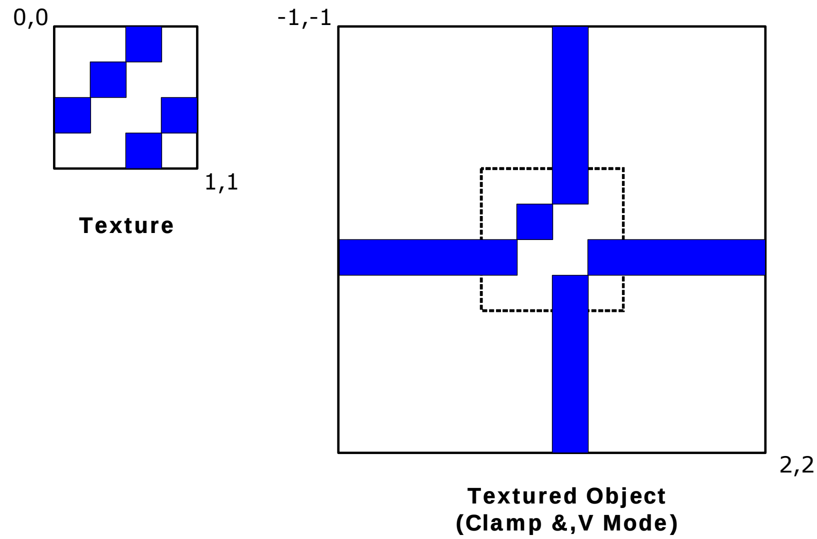
The TEXCOORDMODE_MIRROR_ONCE addressing mode is a combination of Mirror and Clamp modes. The absolute value of the texture coordinate component is first taken (thus mirroring about 0), and then the result is clamped to 1.0. The map is therefore mirrored once about the origin, and then clamped thereafter. This mode is used to reduce the storage required for symmetric maps.

4.2.4.4 TEXCOORDMODE_CLAMP Mode

The TEXCOORDMODE_CLAMP addressing mode repeats the “edge” texel when the texture coordinate extends outside the [0,1) range of the base texture map. This is contrasted to TEXCOORDMODE_CLAMPBORDER mode which defines a separate texel value for off-map samples. TEXCOORDMODE_CLAMP is also supported for cube maps, where texture samples will only be obtained from the intersecting face (even along edges).

The figure below illustrates the effect of clamp mode. The base texture map is shown, along with a texture mapped object with texture coordinates extending outside of the base map region.

Figure 4-6. Texture Clamp Mode



4.2.4.5 TEXCOORDMODE_CLAMPBORDER Mode

For non-cube map textures, TEXCOORDMODE_CLAMPBORDER addressing mode specifies that the texture map's border value *BorderColor* is to be used for any texel samples that fall outside of the base map. The border color is specified via a pointer in SAMPLER_STATE.

4.2.4.6 TEXCOORDMODE_CUBE Mode

For cube map textures TEXCOORDMODE_CUBE addressing mode can be set to allow inter-face filtering. When texel sample coordinates that extend beyond the selected cube face (e.g., due to intra-level filtering near a cube edge), the correct sample coordinates on the adjoining face will be computed. This will eliminate artifacts along the cube edges, though some artifacts at cube corners may still be present.

4.3 Texel Fetch

The Texel Fetch function of the Sampling Engine reads the texture map contents specified by the texture addresses associated with each texel sample. The texture data is read either directly from the memory-resident texture map, or from internal texture caches. The texture caches can be invalidated by the **Sampler Cache Invalidate** field of the MI_FLUSH instruction or via the **Read Cache Flush Enable** bit of PIPE_CONTROL. Except for consideration of coherency with CPU writes to textures and rendered textures, the texture cache does not affect the functional operation of the Sampling Engine pipeline.

When the surface format of a texture is defined as being a compressed surface, the Sampler will automatically decompress from the stored format into the appropriate [A]RGB values. The compressed texture storage formats and decompression algorithms can be found in the *Memory Data Formats* chapter. When the surface format of a texture is defined as being an index into the texture palette (format names includiong "Px"), the palette lookup of the index determines the appropriate RGB values.

4.3.1 Texel Chroma Keying

ChromaKey is a term used to describe a method of effectively removing or replacing a specific range of texel values from a map that is applied to a primitive, e.g., in order to define transparent regions in an RGB map. The Texel Chroma Keying function of the Sampling Engine pipeline conditionally tests texel samples against a "key" range, and takes certain actions if any texel samples are found to match the key.

4.3.1.1 Chroma Key Testing

ChromaKey refers to testing the texel sample components to see if they fall within a range of texel values, as defined by *ChromaKey[][High,Low]* state variables. If each component of a texel sample is found to lie within the respective (inclusive) range and ChromaKey is enabled, then an action will be taken to remove this contribution to the resulting texel stream output. Comparison is done separately on each of the channels and only if all 4 channels are within range the texel will be eliminated.

The Chroma Keying function is enabled on a per-sampler basis by the *ChromaKeyEnable* state variable.

The *ChromaKey[][High,Low]* state variables define the tested color range for a particular texture map.

4.3.1.2 Chroma Key Effects

There are two operations that can be performed to "remove" matching texel samples from the image. The *ChromaKeyEnable* state variable must first enable the chroma key function. The *ChromaKeyMode* state variable then specifies which operation to perform on a per-sampler basis.

The *ChromaKeyMode* state variable has the following two possible values:

KEYFILTER_KILL_ON_ANY_MATCH: Kill the pixel if any contributing texel sample matches the key

KEYFILTER_REPLACE_BLACK: Here the sample is replaced with (0,0,0,0). This matches the Direct3D COLORKEYBLENDENABLE functionality

The Kill Pixel operation has an effect on a pixel only if the associated sampler is referenced by a sample instruction in the pixel shader program. If the sampler is not referenced, the chroma key compare is not done and pixels cannot be killed based on it.

4.4 Shadow Prefilter Compare

When a *sample_c* message type is processed, a special shadow-mapping precomparison is performed on the texture sample values prior to filtering. Specifically, each texture sample value is compared to the "ref" component of the input message, using a compare function selected by *ShadowFunction*, and described in the table below. Note that only single-channel texel formats are supported for shadow mapping, and so there is no specific color channel on which the comparison occurs.

ShadowFunction	Result
PREFILTEROP_ALWAYS	0.0
PREFILTEROP_NEVER	1.0
PREFILTEROP_LESS	(texel < ref) ? 0.0 : 1.0
PREFILTEROP_EQUAL	(texel == ref) ? 0.0 : 1.0
PREFILTEROP_LEQUAL	(texel <= ref) ? 0.0 : 1.0
PREFILTEROP_GREATER	(texel > ref) ? 0.0 : 1.0
PREFILTEROP_NOTEQUAL	(texel != ref) ? 0.0 : 1.0
PREFILTEROP_GEQUAL	(texel >= ref) ? 0.0 : 1.0

The binary result of each comparison is fed into the subsequent texture filter operation (in place of the texel's value which would normally be used).

Software is responsible for programming the "ref" component of the input message such that it approximates the same distance metric programmed in the texture map (e.g., distance from a specific light to the object pixel). In this way, the comparison function can be used to generate "in shadow" status for each texture sample, and the filtering operation can be used to provide soft shadow edges.

Programming Notes:

- Refer to the Surface Formats table in section 4.10.2.1 for the specific surface formats that are supported with shadow mapping.

4.5 Texel Filtering

The Texel Filtering function of the Sampling Engine performs any required filtering of multiple texel values on and possibly between texture map layers and levels. The output of this function is a single texel color value.

The state variables *MinFilter*, *MagFilter*, and *MipFilter* are used to control the filtering of texel values. The *MipFilter* state variable specifies how many mipmap levels are included in the filter, and how the results of any filtering on these separate levels are combined to produce a final texel color. The *MinFilter* and *MagFilter* state variables specify how texel samples are filtered within a level.

4.6 Texel Color Gamma Linearization

This function is supported to allow pre-gamma-corrected texel RGB (not A) colors to be mapped back into linear (gamma=1.0) gamma space prior to (possible) blending with, and writing to the Color Buffer. This permits higher quality image blending by performing the blending on colors in linear gamma space.

This function is enabled on a per-texture basis by use of a surface format with "_SRGB" in its name. If enabled, the pre-filtered texel RGB color to be converted from gamma=2.4 space to gamma=1.0 space by applying a ^(1/2.4) = ^0.4167 exponential function.

4.7 Denoise/Deinterlacer [DevILK]

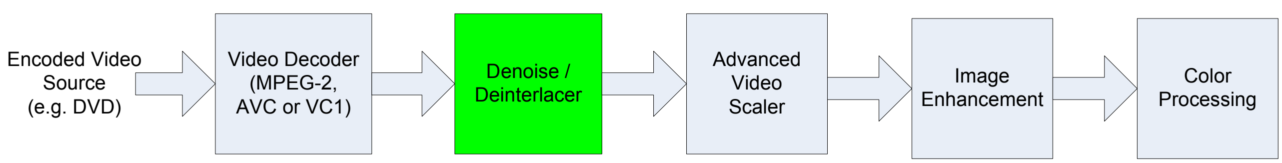
The Denoise/Deinterlacer function takes a 4:2:0 or 4:2:2 video stream and first apply a denoise filter to it and then deinterlace it.

The denoise filter is applied before the deinterlacer. The denoise filter detects and tries to minimize noise in the input field, while the deinterlacer takes a field consisting of every other lines converts a field into a frame. This block also gathers statistics for a global noise estimate made in software at the end of the frame which is used in following frames to tune the denoise filter and image enhancement filter.

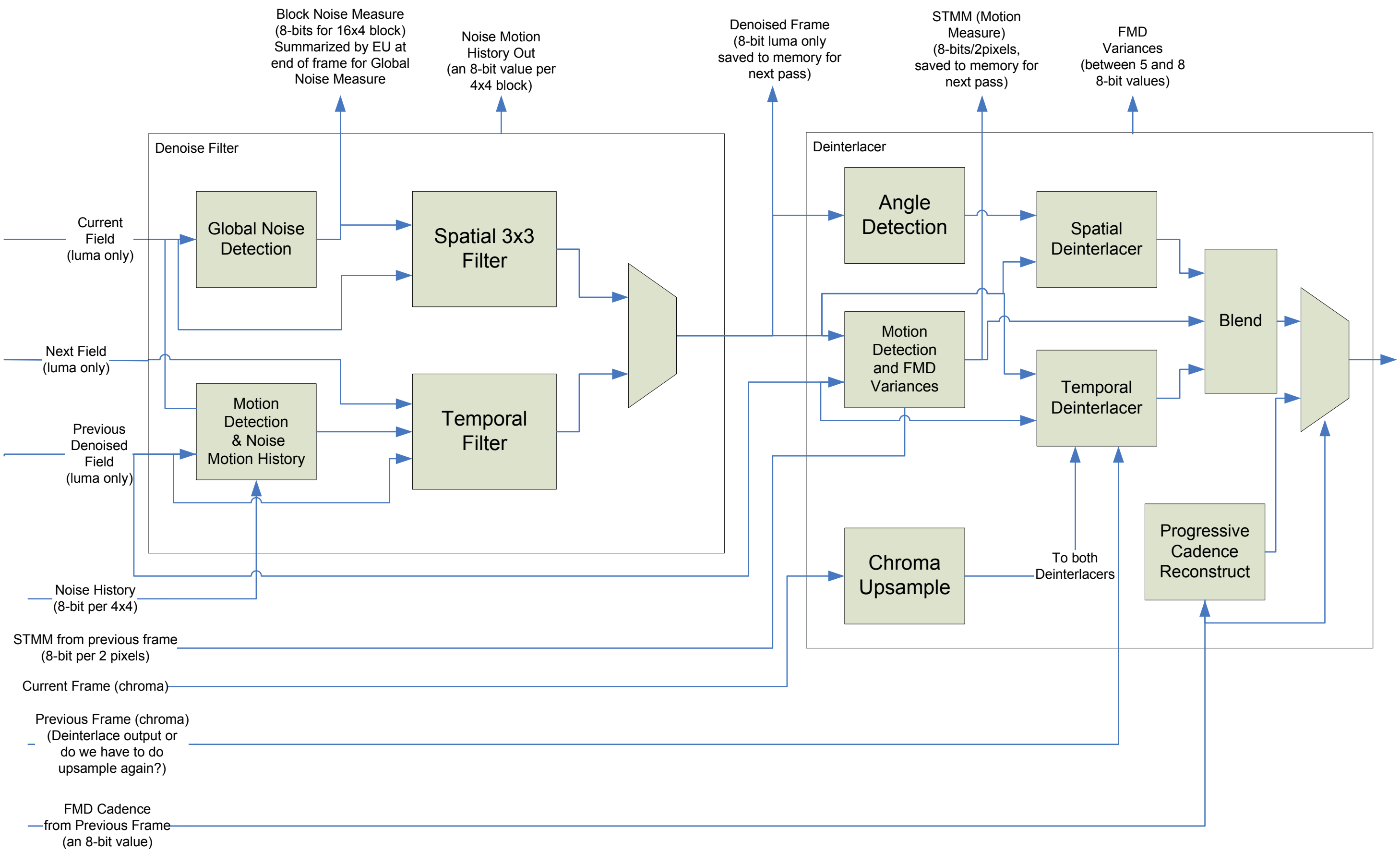
The deinterlacer takes the top and bottom fields of each frame and converts them into two individual frames. This block also gathers statistics for a film mode detector in software run at the end of the frame. If the film mode detector for the previous frame concludes that the input is progressive rather than interlaced then the fields will be put together in the best order rather than being interlaced.

4.7.1 Introduction

This diagram shows how the Denoise/Deinterlacer fits in with the other functions of the video pipe. This is only one possible usage model, other models are possible.



4.7.1.1 Block Diagram



4.7.1.2 Features

- **Denoise Filter** – detects noise and motion and filters the block with either a temporal filter when little motion is detected or a spatial filter. Noise estimates are kept between frames and blended together. Since the filter is before the deinterlacer it works on individual fields rather than frames. This usually improves the operation since the deinterlacer can take a single pixel of noise and spread it to an adjacent pixel, making it harder to remove. The denoise filter works the same whether deinterlacing or progressive cadence reconstruction is being done.

- **Block Noise Estimate** (BNE) – part of the Global Noise Estimate (GNE) algorithm, this estimates the noise over the entire block. The GNE will be calculated at the end of the frame by combining all the BNEs. The final GNE value is used to control the denoise filter for the next frame.

- **Film Mode Detection** (FMD) Variances – FMD determines if the input fields were created by sampling film and converting it to interlaced video. If so the deinterlacer is turned off in favor of reconstructing the frame from adjacent fields. Various sum-of-absolute differences are calcluated per block. The FMD algorithm is run at the end of the frame by looking at the variances of all blocks for both fields in the frame.

- **Deinterlacer** – Estimates how much motion is occuring across the fields. Low motion scenes are reconstructed by averaging pixels from fields from nearby times (temporal deinterlacer), while high motion scenes are reconstructed by interpolating pixels from nearby space (spatial deinterlacer).

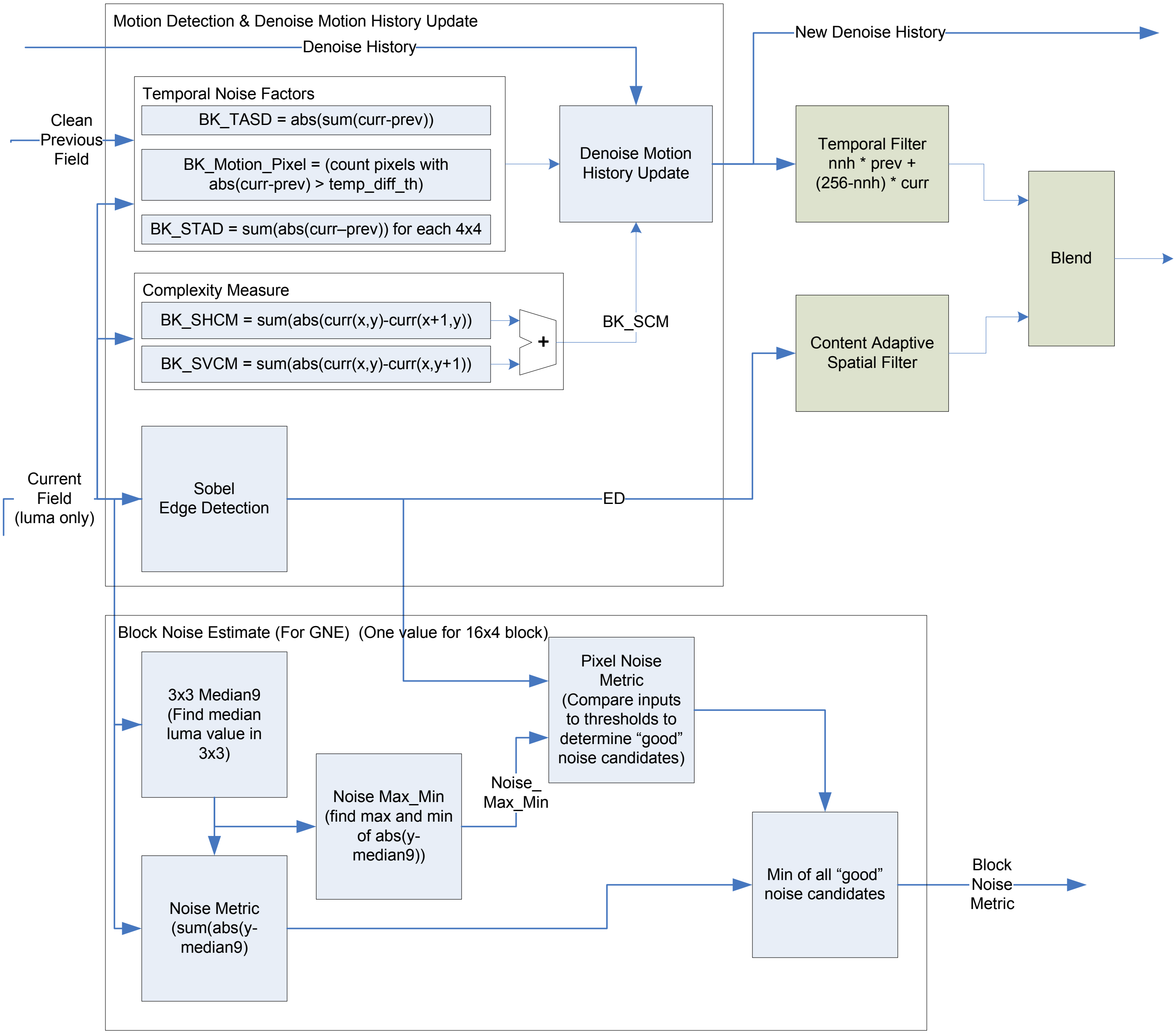
- **Progressive Cadence Reconstruction** – If the FMD for the previous frame determines that film was converted into interlaced video, then this block reconstructs the original frame by directly putting together adjacent fields.

- **Chroma Upsampling** – If the input is 4:2:0 then chroma will be doubled vertically to convert to 4:2:2. Chroma will then either go through it's own version of the deinterlacer or progressive cadence reconstruction.

The output for a 16x4 block is sent to the EU for further processing and writing to memory.

An alternate mode will be provided to send the Deinterlacer intermediate results to the EU to finish the calculation. The denoise filter output data will also be provided.

4.7.2 Denoise Algorithm



4.7.2.1 Motion Detection and Noise History Update

This block detection motion for the denoise filter, which it then combines with motion detected in the past in the same part of the screen. The Denoise History is both saved to memory and also used to control the temporal denoise filter.

The block calculates a number of values for updating the Denoise History. One value is calculated per 4x4 block (pixels from both fields, interleaved):

Block Sum of Temporal Absolute Difference:

$$\text{BK\_STAD} = \sum_{x=0}^{3}\sum_{y=0}^{3} abs(curr(x,y) - prev(x,y))$$

Where curr(x,y) and prev(x,y) are lumas from the current and previous field. The previous field should have already been run through the denoise filter.

Count of motion pixels: increment BK_Motion_Pixel for every pixel in the 4x4 for which: (abs(curr(x,y) – prev(x,y)) >= temporal_diff_th.

Absolute Sum of Temporal Difference sums the differences without the initial absolute value, so that random motions will tend to cancel out:

$$\text{BK\_TASD} = abs(\sum_{x=0}^{3}\sum_{y=0}^{3} (curr(x,y) - prev(x,y)))$$

Sum of Complexity Measure looks for differences in the spatial domain:

$$\text{BK\_SHCM} = \sum_{x=0}^{2}\sum_{y=0}^{3} abs(curr(x,y) - curr(x+1,y)) \quad \text{// sum of 12 pixel pairs}$$

$$\text{BK\_SVCM} = \sum_{x=0}^{3}\sum_{y=0}^{2} abs(curr(x,y) - curr(x,y+1)) \quad \text{// sum of 12 pixel pairs}$$

BK_SCM = BK_SHCM + BK_SVCM

Denoise Motion History Update (for an 8-bit motion history):

```
if (BK_STAD>=dnmh_stad_th) or (BK_Motion_Pixel > dnmh_mp_th) { // Motion Block
    motion_block = 1;
    if (denoise_history >= 128)
      new_denoise_history = denoise_history / 2;
    else
      new_denoise_history = 0;
} else {   // static block
    motion_block = 0;
    if  (denoise_history < 128)
      new_denoise_history = 128;
```

```
    else if (denoise_history < dnmh_history_max)
      new_denoise_history = denoise_history + dnmh_delta;  // default value 8
for delta
    else
      new_denoise_history = denoise_history;
    if ((BK_TASD > dnmh_tasd_th) and (BK_SCM < dnmh_scm_th))
      new_denoise_history = 128;
}
```

4.7.2.2 Temporal Filter

For each pixel we need to filter we look at the noise history for the associated 4x4.

```
temporal_denoised = (new_denoise_history * curr(x,y) + (256 –
      new_denoise_history) * prev(x,y) +128) >> 8
```

4.7.2.3 Edge Detection

Edge detection is done on every pixel by estimating a gradient on the 3x3 neighborhood of pixels in the current field. The calculation only uses a multiply of 2, so shifts and add are all that is needed. Currently only vertical and horizontal edges are detected, 45 degrees is a potential improvement.

```
Hrz Edge = abs(c(x-1,y-1) + 2*c(x,y-1) + c(x+1,y-1) – c(x-1,y+1) – 2*c(x,y+1) – c(x+1,y+1))
```

```
Vrt Edge = abs(c(x-1,y-1) + 2*c(x-1,y) + c(x-1,y+1) – c(x+1,y-1) – 2*c(x+1,y) – c(x+1,y+1))
```

The Hrz_Edge and Vrt_Edge are added together and if the sum is greather than dn_edge_th then an edge is detected:

```
ED = (Hrz_Edge + Vrt_Edge)  >> 3
```

4.7.2.4 Context Adaptive Spatial Filter

For each pixel in the local 3x3, compare it’s luma to the lumas of the pixel to be filtered. Each pixel for which the absolute difference is less than good_neighbor_th (see state variable in section 1.11.3.2) is marked as a “good neighbor”:

The filtered pixel is then equal to:

```
spatial_denoised = ∑ C1 * Good_neighbor luma / num_good_neighbors
```

The divide is implemented as a multiply by a table lookup:

```
spatial_denoised = ((∑Good_neighbor luma + (num_good_neighbors >>1)) *
gn_q_table[num_good_neighbors-1]) >> 11
```

Note: The number of good neighbors varies from 1 to 9 since the center pixel is always good. Gn_q_table provides the reciprocal:

```
gn_q_table[9] = {2048, 1024, 682, 512, 409, 341, 292, 256, 227};
```

If this pixel is not part of an edge then the spatial value is tested against the local median (TBD: any sharing of calculation for the Block Noise Estimate?):

```
if (ED < dn_edge_th && abs(block_median - pixel]) < temporal_diff_th)
      spatial_denoised = block_median;
```

4.7.2.5 Denoise Blend

The denoise blend combines the temporal and spatial denoise outputs.

First we check to see if the temporal is out of the local range, if so we use the average of the denoised and the local limit instead:

```
if (temporal_denoised >= block_max)

      temporal_denoised=(temporal_denoised+block_max)>>1;

if (temporal_denoised < block_min)

      temporal_denoised=(temporal_denoised+block_min)>>1;
```

Where block_max and block_min are the largest and smallest luma values in the local 3x3 (can be shared with BNE calculation).

Next we decide between using the spatial and temporal denoise output:

```
      t_diff  = abs(curr(x,y) - prev(x,y);
      if (t_diff < temporal_diff_th) {
            if (motion_block==1)
                  denoise_out = spatial_denoised;
            else {
                  if (t_diff < temp_diff_low)
                        denoise_out=temporal_denoised;
                  else {
                        denoise_out=
                          (spatial_denoised*(t_diff-temp_diff_low) +
                           temporal_denoised*(temporal_diff_th-t_diff)+
                           (temporal_diff_th-temp_diff_low)/2
                           ) * q_table[temporal_diff_th-temp_diff_low-1]) >> 10;
            }
      } else {
            denoise_out = spatial_denoised;
      }
```

Motion_block is defined in section 4.7.2.1 above. T_diff can be limited to 6-bits to minimize the multipler gates required in the blend. A divide is eliminated by providing the reciprocal of the divisor in the q_table which is defined:

```
q_table[16] = {1024,512,341,256,205,171,146,128,114,102,93,85,79,73,68,64}
```

The following restrictions also apply:
1) Temporal_diff_th – temp_diff_low is limited in the state variable definition to the range 16 to 1.
2) Since t_diff<temporal_diff_th; (t_diff – temp_diff_low) is less than 16

3) Since t_diff>=temp_diff_low; (temporal_diff_th-t_diff) is less then or equal to 16.

The precision needed for `spatial_denoised*(t_diff-temp_diff_low)` is 8-bit times 4-bits to produce 12-bits. The other multiply is 8 by 5 to produce 13-bits; the extra bit is needed for 16. The multiplier to implement the divide will be a 13-bit times the 11-bit number out of q_table, but this could be reduced by implementing a 13x9 bit multiplier with the top 2 bits controlling a mux since the only table entries that use them are 1024 and 512.

4.7.3 Block Noise Estimate (part of Global Noise Estimate)

The block noise estimate is a single number for the 16x4 block (DI enabled) or a 16x8 block (DN only). The block noise estimate for the entire frame is summed to get the global noise estimate. The algorithm uses these inputs calculated for each pixel:

- The Ede Detection from the denoise block is shared here, though BNE only needs numbers for the 16x4 block rather than the 32x8 that the denoise filter will be producing.
- median9 – the median of the 9 luma values for the 3x3 neighborhood pixels is used. Median5, the median of the pixels above/below/right/left/center may be satisfactory as a lower gate count solution. TBD: To see if this is OK 2/9/2007
- for each pixel luma "y" in 3x3: noise_metric = sum(y – median9)
- noise_min = min(abs(y-median9)) - min of all 9 ys in 3x3
- noise_max = max(abs(y-median9)) – max of all 9 ys in 3x3
- noise_min_max = noise_max(x,y) – noise_min(x,y)
- pixel_noise_metric = noise_metric if (ED(x,y) < bne_edge_th) and (noise_max_min(x,y) < bne_nn_th) // Same edge detection as in 4.7.2.3
- block_noise_estimate = min of all pixel_noise_metrics that pass the if test in the 16x4 (use 255 if no pixels pass the test)

If the block_noise_estimate is less than 255 then it is added to a sum gathered across the entire frame. The summation will need to be 23-bits wide to be able to sum 8-bit values for all 32,400 blocks in a 1920x1080 frame. In addition, there will be a count of the number of blocks in the sum. The data will be written to memory at the end of the frame. Two sets of counters are needed to support 2 simultaneous streams. The streams are distinguished by the dndi_stream_id state variable in the DI state.

The per block block_noise_estimate is also sent to the EU in the output message for possible use by the video encoder.

4.7.4 Deinterlacer Algorithm

The overall goal of the motion adaptive deinterlacer is to convert an interlaced video stream made of fields of alternating lines into a progressive video stream made of frames in which every line is provided.

If there is no motion in a scene, then the missing lines can be provided by looking at the previous or next fields, both of which have the missing lines. If there is a great deal of motion in the scene, then objects in the previous and next fields will have moved, so we can't use them for the missing pixels. Instead we have to interpolate from the neighboring lines to fill in the missing pixels. This can be thought of as interpolating in time if there is no motion and interpolating in space if there is motion.

This idea is implemented by creating a measure of motion on a per 2 pixel basis called the Spatial-Temporal Motion Measure (STMM). If this measure shows that there is little motion in an area around the pixels, then the missing pixels are created by averaging the pixel values from the previous and next frame. If the STMM shows that there is motion, then the missing pixels are filled in by interpolating from neighboring lines with the Spatial Deinterlacer (SDI). The two different ways to interpolate the missing pixels are blended for intermediate values of STMM to prevent sudden transitions.

The Deinterlacer uses two frames for reference. The current frame contains the field that we are deinterlacing. The reference frame is the closest frame in time to the field that we are deinterlacing – if we are working on the 1st field then it is the previous frame, if it is the 2nd field then it is the next frame.

4.7.4.1 Spatial-Temporal Motion Measure

This algorithm combines a complexity measure with a estimate of motion. This prevents high complexity scenes from incorrectly causing motion to be detected. It is calculated for a set of pixels 2 wide by 1 high.

Complexity is measured in the vertical and horizontal directions with the SVCM and SHCM. For each set of 2 pixels which need to be interpolated, a window of pixels is used that is 4 wide and 5 high - +/-1 pixel in X and +/- 2 pixels in Y. The pixels values are taken from both the current and previous field - for example, if we are deinterlacing the top field then lines y+2,y, and y-2 will come from the top field; while line y+1 and y-1 will come from the bottom field.

Spatial vertical complexity measure (SVCM) is a sum of all the differences in the vertical direction for a window around the current pixels. If we take x,y=0,0 as the left pixel of our 2x1 then:

$$\text{SVCM} = \sum_{x=0}^{1}\sum_{y=0}^{2} abs(c(x,y) - c(x,y-2))$$

Where c(x,y) is the luma value at that x,y location in the current frame. Note that we are skipping by 2 in the Y direction to ensure that the compares are only done with lines from the same field.

Spatial horizontal complexity measure (SHCM) is a sum of differences in the horizontal direction.

$$\text{SHCM} = \sum_{x=-1}^{1}\sum_{y=-1}^{y=1} abs(c(x,y) - c(x+1,y))$$

The vertical edge complexity measure (VECM) is a sum of difference in the horizontal direction similar to SHCM, but uses different pixels from the window.

$$\text{VECM} = \left(\left(\sum_{y=-2}^{y=2} abs(c(x,y) - c(x+1,y))\right) * \text{vecm\_mul}\right) >> 5$$

Temporal Difference Measure (TDM) is a measure of differences between pairs of fields with the same lines. It uses filtered versions of c(x,y) from the current frame and r(x,y) from the reference frame (either the previous or next frame).

The filter used is a cross filter which uses the pixels above, below, to the right and to the left of the needed pixel in the same field. When denoise filter is enabled, the filter input c(x,y) is a denoised pixel only if -2<=y<=6 for dndi_topfirst=1, and -3<=Y<=5 for dndi_topfirst=0. Note that r(x,y) is a denoised pixel regardless of y.

c'(x,y) = (6*c(x,y) + 3*c(x-1,y) + 3*c(x+1,y) + 2*c(x,y-2) + 2*c(x,y+2)) >> 4 (Done for both c(x,y) and r(x,y)

$$\text{TDM} = \sum_{x=-1}^{2}\sum_{y=-2}^{2} abs(c'(x,y) - r'(x,y))$$

STMM is then calculated by :

STMM = ((TDM >>tdm_shift1)<<tdm_shift2) / (SCM >> 4) + stmm_c2)

where SCM = max(0, SVCM+SHCM-VECM). Tdm_shift1 is used to quantize the STMM result, while Tdm_shift2 is used to set the STMM range. Tdm_shift1 can range from 4 to 6; since TDM has 13 bits this results in between 9 and 7 bits of precision. Tdm_shift2 can range from 6 to 8, producing a value between 17 and 13 bits, of which only 9-bits are non-zero. The divide can be implemented by a 8-bit reciprocal table followed by an 9 -bit x 8-bit multiply by the TDM value, which finally produces an output of 8-bits.

STMM is then smoothed with an exponential moving average with the STMM saved from the previous field:

```
if  (STMM > stmm_md_th)
     STMM2 = (stmm_trc1 * STMM_s + (256-stmm_trc1)*STMM) / 256
else
     STMM2 = (stmm_trc2 * STMM_s + (256-stmm_trc2)*STMM) / 256
```

with state variables stmm_trc1 (typical value 64), stmm_trc2 (typical value 192), and stmm_md_th.

This process prevent sudden changes in STMM, though STMM over a certain value uses a smaller smoothing constant (c1) which allows it to change faster. STMM2 is stored to memory to be read as STMM_s by the next frame.

One final step is used to prevent sudden drops in STMM in the horizontal direction – taking the maximum of the STMM on the right and left sides:

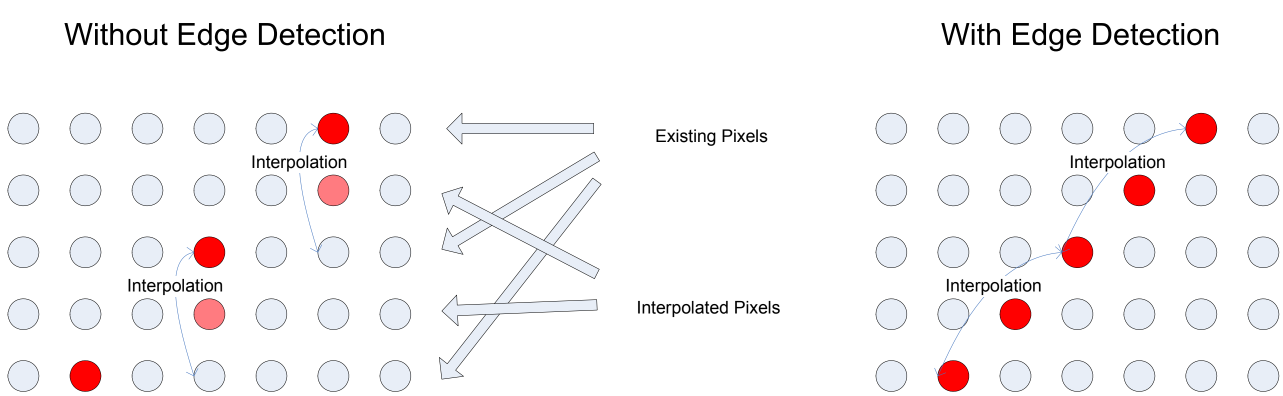
STMM3(x) = max (STMM2(x-2), STMM2(x), STMM2(x+2))

The resulting STMM3 will be used as a blending factor between the spatial and temporal deinterlacer.

4.7.4.2 Spatial Deinterlacer Angle Detection

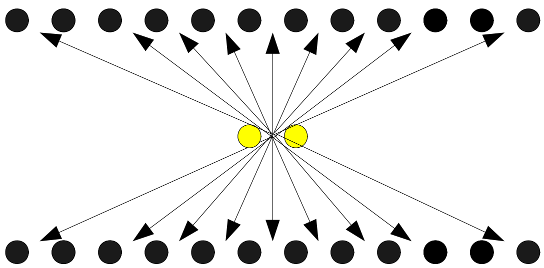
Deciding the best pixels to interpolate in the current field is the job of the spatial deinterlacer. The simplest method would be to interpolate directly from the pixels above and below the missing pixels, but this can look bad; edges and lines particularly look jagged with this solution.

A better solution is to detect the direction of edges in the pixel neighborhood and interpolate along the edge direction.



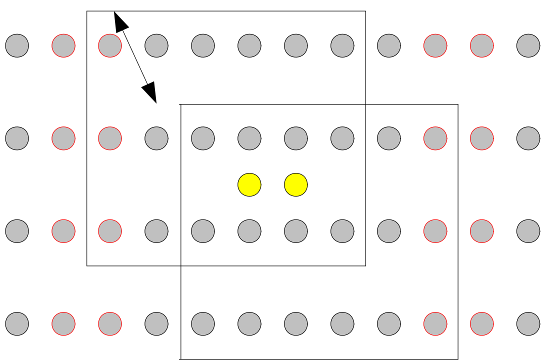
Edge detection is done per 2 pixels to lower the compute needed (may change in this implementation depending on quality). Edge detection is done by taking a window of pixels around the pixels of interest and comparing with a window offset in the direction being tested. The more simularity between the windows the more likely it is that the movement is in the direction of an edge.

We test 9 different directions to pick the best edge: vertical, +/-45°, +/-27°, +/-18° and +/-11 degrees. The window offset for 45° x+/-1, likewise the offset of 27° is x+/-2, 18° is x+/-3, and 11° is x+/-5. X+4 is not used because the gap between 18° and 11° is too small to make it worth checking.



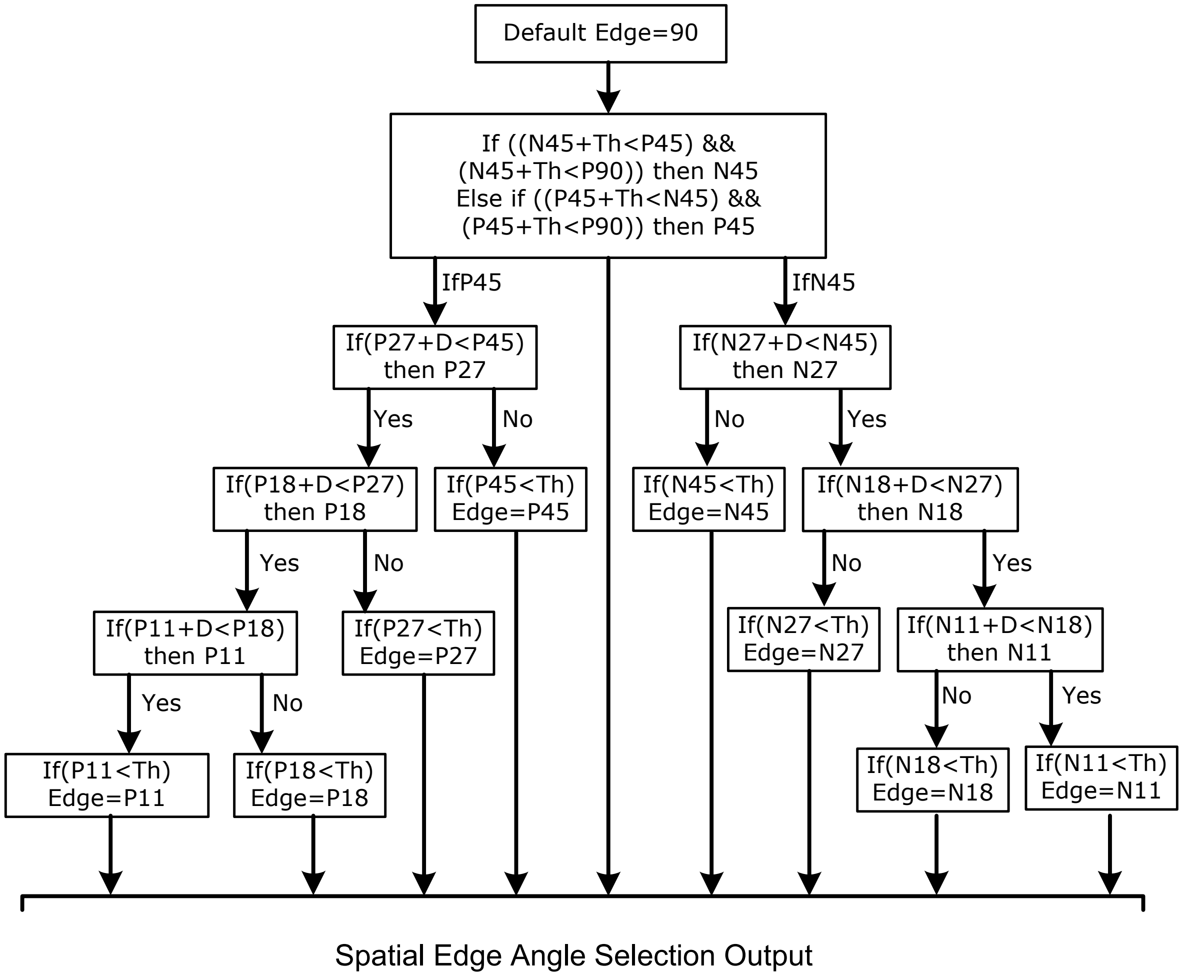
Use x,y=0,0 for the left pixel of the pair that we want to interpolate, and xoffset is the offset described in the above paragraph. The equation for each angle checked is:

$$\text{AngleCost\_6x3} = \sum_{x=-2}^{3} \sum_{y=-2,0,2} abs(n(x + xoffset, y + 1) - n(x - xoffset, y - 1))$$



The above picture illustrates the 45 degree angle compution – taking the sum-of-absolute differences of the two 6x3 blocks around the 2 pixels that need an angle estimated. Each block is offset by 1 in Y and X in opposite direction. The offset in X is larger for the other angles, of course. Angle detection requires up to 7 pixels (offset of 5 plus 2 to get all the pixels in the 6x3) on the right and left of the output block, requiring the input to the deinterlacer from the denoise to be 16 + 7 + 7, or 30 pixels.

Once we have all the angle values, the final decision is done by comparing them with each other. In the following diagram N45 indicates the AngleCost_6x3 for -45°, likewise P27 is the value for +27°, etc. Th and D are constants used to fine tune the algorithm.



B6783-02

Any missing arcs in the above diagram use the default edge of 90 degrees; for example if the lower left box has P11 >= Th then the default will be used.

4.7.4.2.1 Angle Robustness Check

Three special checks are made to eliminate incorrect angle detection.

Fallback Mode 1

Moving regions with fine details can confuse the angle detection. This fallback mode will detect fine details and fall back to 90 degrees if they are detected.

SUM_H1(x,y) = $\sum_{s=-2}^{3} abs(c(x+s,y)-c(x+s+1,y))$

This sum is similar to SHCM, but over a horizontal line of -2 to +3 only.

SUM_H2(x,y) = $\max_{s=-2,-1,\ldots,3}(abs(c(x-2,y)-c(x+s,y))+abs(c(x+s,y)-c(x+4,y))$

```
if (SUM_H1(y-1) + SUM_H1(y+1) > SUM_H2(y-1) + SUM_H2(y+1) + sdi_t1 &&

   SUM_H1(y-1) + SUM_H1(y+1) >= sdi_t2)          Then use 90 degree
```

The final decision for each pixel is done using the sums from above and below the current Y.

Fallback Mode 2

Sometimes the 6x3 angle detection window makes mistakes due to pixels on the edge of the window. Adding a check using a 2x1 window fixes these problems:

```
If( AngleCost_6x3(90 degree) + (AngleCost_2x1(90 degree)<<3) <
  AngleCost_6x3(best angle) + ((AngleCost_2x1(best angle) + sdi_angle2x1)<<3))
then use 90 degree
```

AngleCost_2x1 is the same as AngleCost_6x3 with a much smaller window:

AngleCost_2x1 = $\sum_{x=0}^{1} abs(n(x+xoffset,y+1)-n(x-xoffset,y-1))$

AngleCost_2x1 can be collected during the calculation of AngleCost_6x3.

Horizontal Median

One final step is used to prevent sudden angle changes – the angle detected for the pixel pair is compared to the angle detected for the pixels to the right and left and the median of the 3 is the angle finally used:

```
   angle_final(x) = median3(angle(x-2), angle(x), angle(x+2))
```

4.7.4.3 Spatial Deinterlacer Interpolation

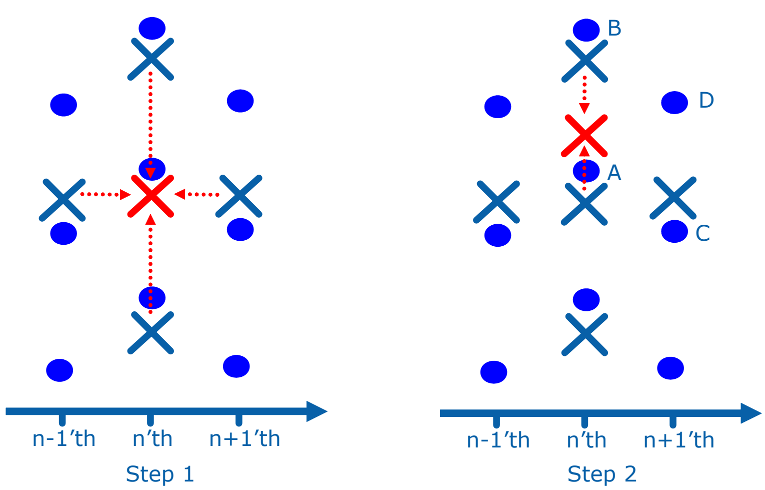
Once the best angle is picked, the interpolation is done on a per pixel basis. Both the chroma and luma need to be interpolated (see section 4.7.4.4 for chroma). Only 422 output is needed, so there will be a chroma pair for each 2 lumas. The interpolation itself is very simple: take a pixel from the line above and the line below along one of the 9 possible angles, and average the 8-bit luma and chroma values to get the result pixel. We will do 2 lumas per clock to get enough performance.

4.7.4.4 Chroma Up-Sampler

The DN/DI block supports 4:2:0, 4:1:1 and 4:2:2 inputs, but only outputs 4:2:2. For 4:2:0 and 4:1:1 the chroma needs to be up-sampled to 4:2:2 before interpolation.

The 4:2:0 input has chroma at ¼ the rate of the luma; ½ in the horizontal and ½ in the vertical directions. The output needs to be 4:2:2, where chroma is ½ the rate of luma; ½ the horizontal but the same in the vertical direction. Then chroma can be de-interlaced in the vertical direction. For luma we are working with 16x4 blocks, so for chroma we will have 8x2 in 4:2:0 and 8x4 in 4:2:2.

The 4:2:0 to 4:2:2 conversion requires doubling the chroma in the vertical direction to match the luma:



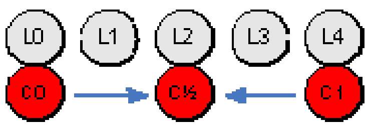
The chroma is doubled by a simple interpolation in both time and space. In the following equations, pixel locations are specified as u(field, x_location, y_location). Field=n would be from the current field, n-1 is from the previous field, and n+1 is from the next field. The Cr and Cb X and Y values are ½ the luma values to map to the smaller area.

```
temporal_cr = (cr(n-1,x,y) + cr(n+1,x,y)) / 2  // Simple average in time

spatial_cr = (cr(n,x,y-1) + cr(n,x,y+1)) / 2   // Simple average in vertical space

if (STMM3 < stmm_min)
    new_cr = temporal_cr
else if (STMM > stmm_max)
    new_cr = spatial_cr
else
   new_cr = ((STMM3 - stmm_min) * spatial_cr  + (stmm_max - STMM3) * temporal_cr) >>
stmm_shift
```

Note that this simple chroma interpolation is not correct, since the chroma sample position is ¼ of a pixel different between 420 and 422. The polyphase filter in the scaler will be used to correct this inprecision by modifying the filter coefficients in software.

For performance a single Cr and Cb has to be produce per clock in this stage to match the 2 pixel per clock performance goal.

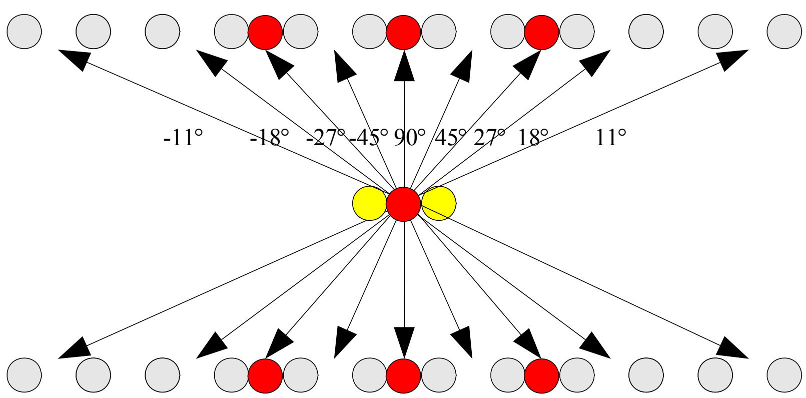
4:1:1 also has chroma at ¼ the rate of luma; ¼ in the horizontal direction and the same in the vertical direction. To convert to 4:2:2 we need to double the chroma horizontally. This will be done by averaging the chromas to the right and left to produce the new chroma.



The above diagram shows how the existing chroma values (both U and V) are averaged between C0 and C1 to produce the new C½. C0 is the chroma asociated with lumas L0 through L3, while C1 is associated with L4 through L7.

4.7.4.5 Chroma Deinterlace

The next step is to do the deinterlacing. Chroma uses the output of the luma angle decision, but reduces the number of angles. The actual spatial deinterlace algorithm is a little different for chroma, since there are only 1 chroma per 2 lumas: some of the chromas are missing and must be filled in.



The diagram shows the chromas used in red. Only 90°, -27° and 27° are directly available. The chromas for +/-45° are derived by a simple average of the 90° and 27° chromas. +/-18° and +/-11° both use the chroma for +/-27°.

4.7.4.5.1 Static Image Fallback Mode

This algorithm has a problem with static images – alternate fields use different luma angle detections and can select different angles, causing noticable flicker. Rather than calculating a separate set of angles for chroma, we instead will blend with STMM so that a static image will use 90 degrees.

```
if (STMM3 < stmm_min)
    chroma_sdi = chroma90degree
else if (STMM > stmm_max)
    chroma_sdi = chroma_3angle
else
    chroma_sdi = (chroma90degree * (stmm_max - STMM3) + chroma_3angle * (STMM3
      - stmm_min)) >> stmm_shift
```

4.7.4.6 Temporal Deinterlacer and Final Deinterlacer Blend

The temporal deinterlacer is a simple average between the previous and next field; when deinterlacing the 1st field of current the average will be between the 2nd field of previous and the 2nd field of current.

The interpolation between spatial and temporal:

```
if (STMM3 < stmm_min)
   deinterlace_out  = tdi;
else if (STMM3 > stmm_max)
   deinterlace_out = sdi;
else
  deinterlace_out = (sdi * (STMM3 - stmm_min)  + tdi * (stmm_max - STMM3)) >>
stmm_shift
```

4.7.4.7 Progressive Cadence Reconstruction

When the FMD for the previous frame indicates that a progressive mode is being used rather than interlaced, the luma and chroma will be taken from adjacent fields rather than spatially interpolated. The exact fields needed depend on state variables written to memory by a thread at the end of the previous frame. The thread will use the FMD variances written to memory via CSunit on the flush at the end of a frame.

Since we are deinterlacing 2 fields at a time – one from the previous frame and one from the current frame (see section 4.7.6.1) we will need a state variable which says how each one should be put together. In each case there are only two possibilities – either the field should be put together with the matching field in the same frame or it should be put together with the adjacent field in the other frame.

If we are deinterlacing the 2^{nd} field from frame N and the 1^{st} field from frame N+1, then the FMD decision (which is made on frame boundaries) will be from frame N-1.

Chroma is reconstructed the same as luma – only the first step of doubling chroma is done in the chroma upsampling block for the two needed fields.

4.7.4.8 Motion Search

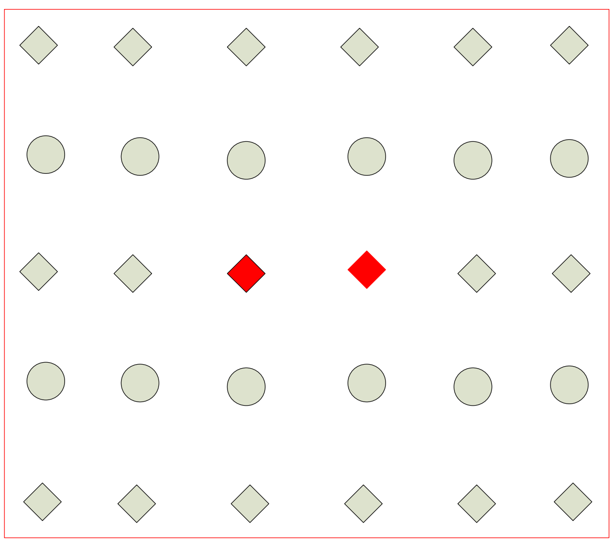
Motion will be estimated independently for each horizontal pair of pixels in the 16x4 block. The area around each pixel pair will be compared to areas in adjacent fields with different X/Y offsets. 16 different offsets, or motion vectors, will be examined in this order:

Y= -2, X = -1, 0, 1

Y = 0, X = -6, -5, -4, -3, -2, 2, 3, 4, 5, 6

Y = 2, X = -1, 0, 1

The area to be compared around the pixel pair is a 6 wide by 5 high window - 2 pixels on right and left and 2 lines above and below. The lines above and below are from both fields, so a total of 3 lines from the same field and 2 lines from the complement field are compared to lines in 2 fields from an adjacent frame.



The motion estimation equation for a pixel pair is:

$$SAD = \sum_{i=x-w}^{x+w+1} \sum_{j=y-h}^{y+h} \left| p_{ref}\left(i + M_x, j + M_y\right) - p_{curr}\left(i, j\right) \right|$$

$$\left(h = 2 \text{ and } w = 2\right)$$

Mx, My is the motion vector offset being tested, and x,y is the location of the leftmost pixel of the pair. The motion vector with the smallest SAD is kept as the best motion estimate; if two motion vectors have the same SAD then the last one tested will be kept.

4.7.4.9 Robustness Checks

The motion estimate output goes through 2 checks to make sure it is not an aberration – a smoothness check and a consistency check.

4.7.4.9.1 Consistency Check

The consistency check is done per pixel and makes sure that the pixels we are interpolating for MC have a lower delta than the ones that would be interpolated for spatial DI:

$$\left|P_{cur\_opp}(x - Edge, y-1) - P_{cur\_opp}(x + Edge, y+1)\right| > \left|P_{DI}(x,y) - P_{DI\_cur}(x,y)\right|$$
$$\&\&\left|P_{DI}(x,y) - P_{DI\_cur}(x,y)\right| < MC\_pixel\_consistency\_TH\,(default: 25)$$

Here Edge is the delta found by SDI which corresponds to the best angle.
MC_pixel_consistency_TH (U6)is a state parameter.

P_{DI_cur} is defined as: (same definition as in the motion compensation section)

- If(Mx%2 == 0 & & (My/2)%2 == 0)

$$P_{DI\_cur}(x,y) = P_{cur\_same}(x - M_x/2, y - M_y/2);$$

- If(Mx%2 == 1 & & (My/2)%2 == 0)

$$P_{DI\_cur}(x,y) = \begin{cases} \mathrm{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2)\right); \text{ if } (M_x \ge 0) \\ \mathrm{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2)\right); \text{ if } (M_x < 0) \end{cases}$$

- If(Mx%2 == 0 & & (My/2)%2 == 1)

$$P_{DI\_cur}(x,y) = \mathrm{AVG}\left(P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2, y - M_y/2 + 1)\right);$$

If(Mx%2 == 1 & & (My/2)%2 == 1)

$$P_{DI\_cur}(x,y) = \begin{cases} \mathrm{AVG}\begin{pmatrix} P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2 - 1), \\ P_{cur\_same}(x - M_x/2, y - M_y/2 + 1), P_{cur\_same}(x - M_x/2 - 1, y - M_y/2 + 1) \end{pmatrix}; if\,(M_x \ge 0) \\ \mathrm{AVG}\begin{pmatrix} P_{cur\_same}(x - M_x/2, y - M_y/2 - 1), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2 - 1), \\ P_{cur\_same}(x - M_x/2, y - M_y/2 + 1), P_{cur\_same}(x - M_x/2 + 1, y - M_y/2 + 1) \end{pmatrix}; if\,(M_x < 0) \end{cases}$$

4.7.4.9.2 Smoothness Check

The smoothness check compares the motion vector found for neighboring pixel pairs. The neighbors are different for different locations to make sure it stays within the local 4x4. Each pixel pair has 3 sets of comparison with neighbor pixel pair within the 4 by 4: 2 sets of X/Y comparisons for the vertical direction and one set of X/Y comparisons for the horizontal direction.

For lines 1 and 2 in the 16x4:

$$If\left(abs\left(MV_x(x,y) + MV_x(x,y+1)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) + MV_y(x,y+1)\right) <= smooth\_mv\_th$$
$$AND\left(abs\left(MV_x(x,y) - MV_x(x,y+2)\right)\right) <= smooth\_mv\_th$$
$$AND\ abs\left(MV_y(x,y) - MV_y(x,y+2)\right) <= smooth\_mv\_th$$

Where ***smooth_mv_th***(U2) is a state parameter.

This equation ensures that the pixel pair 1 and 2 lines below have motion vector X and Y components (MVx & MVy) that are within a threshold of the best motion vector for the current pixel pair. The compares with y+1 use "+" rather than "-" since they are comparing motion vectors in the opposite field, which have motion vectors pointing in the opposite direction, since they are using the current field as their reference. For example, if the current pixel has a motion vector of (4,2), the motion vector of x,y+1 would be the same if it is (-4,-2).

For lines 3 and 4 in the 16x4:

$$\begin{aligned}&If\left(abs\left(MV_x(x,y)+MV_x(x,y-1)\right)\right)<=smooth\_mv\_th\\&AND\ abs\left(MV_y(x,y)+MV_y(x,y-1)\right)<=smooth\_mv\_th\\&AND\left(abs\left(MV_x(x,y)-MV_x(x,y-2)\right)\right)<=smooth\_mv\_th\\&AND\ abs\left(MV_y(x,y)-MV_y(x,y-2)\right)<=smooth\_mv\_th\end{aligned}$$

For pixel pairs with the first pixel location x%4 == 0 (low X in the 4x4):

$$\begin{aligned}&If\left(abs\left(MV_x(x,y)-MV_x(x+2,y)\right)\right)<=smooth\_mv\_th\\&AND\ abs\left(MV_y(x,y)-MV_y(x+2,y)\right)<=smooth\_mv\_th\end{aligned}$$

For pixel pairs with the first pixel location x%4 != 0 (high X in 4x4):

$$\begin{aligned}&If\left(abs\left(MV_x(x,y)-MV_x(x-2,y)\right)\right)<=smooth\_mv\_th\\&AND\ abs\left(MV_y(x,y)-MV_y(x-2,y)\right)<=smooth\_mv\_th\end{aligned}$$

When all 3 comparisons pass the threshold, the smoothness check is passed.

4.7.4.10 Motion Comp

The MCDI output is an average done per pixel on pixels chosen from adjacent field.

There are 4 different equations depending on the motion vector (Mx, My):

If (Mx%2 ==0) && (My == 0) then $P_{DI}(x,y)=P_{ref\_same}(x+M_x/2,\ y+M_y/2);$

If (Mx%2 ==1) && (My == 0) then

$$P_{DI}(x,y)=\begin{cases}\mathrm{AVG}\left(P_{ref\_same}(x+M_x/2,\ y+M_y/2),P_{ref\_same}(x+M_x/2+1,\ y+M_y/2)\right);\ \text{if }(M_x\geq 0)\\\mathrm{AVG}\left(P_{ref\_same}(x+M_x/2,\ y+M_y/2),P_{ref\_same}(x+M_x/2-1,\ y+M_y/2)\right);\ \text{if }(M_x<0)\end{cases}$$

If (Mx%2==0) && abs(My) == 2 then

$$P_{DI}(x,y)=\mathrm{AVG}\left(P_{ref\_same}(x+M_x/2,\ y+M_y/2-1),P_{ref\_same}(x+M_x/2,\ y+M_y/2+1)\right);$$

If (Mx%2==1) & abs(My) == 2 then

$$P_{DI}(x,y)=\begin{cases}\mathrm{AVG}\begin{pmatrix}P_{ref\_same}(x+M_x/2,\ y+M_y/2-1), P_{ref\_same}(x+M_x/2+1,\ y+M_y/2-1),\\ P_{ref\_same}(x+M_x/2,\ y+M_y/2+1), P_{ref\_same}(x+M_x/2+1,\ y+M_y/2+1)\end{pmatrix}; if\ (M_x\geq 0)\\ \mathrm{AVG}\begin{pmatrix}P_{ref\_same}(x+M_x/2,\ y+M_y/2-1), P_{ref\_same}(x+M_x/2-1,\ y+M_y/2-1),\\ P_{ref\_same}(x+M_x/2,\ y+M_y/2+1), P_{ref\_same}(x+M_x/2-1,\ y+M_y/2+1)\end{pmatrix}; if\ (M_x<0)\end{cases}$$

For all these equations, if more vareties of My are used than -2,0,2 then we need to use (My/2)%2==0) instead of My==0, and (My/2)%2==1 instead of abs(My)==2.

4.7.4.11 Merge with TDI & SDI

The MADI equation used in Gen6 was:

```
if (STMM3 < stmm_min)
        deinterlace_out  = tdi;
else if (STMM3 > stmm_max)
        deinterlace_out = sdi;
Else
        deinterlace_out = blend(tdi, sdi)
```

Where STMM3 is a measure of the complexity of the scene and how much motion is in it.

The equation with MCDI is:

```
if (STMM3 < stmm_min)
        Deinterlace_out = tdi;
else if (STMM3 > stmm_max)
        deinterlace_out = DItemp;
else
        deinterlace_out = blend(tdi, DItemp)
```

Where DItemp is defined below:

Content Adaptive Thresholding:

We denote the best_ME_SAD as the minimal SAD value for the MV candidates. Best_ME_SAD and Best_SAD_Angle_cost are measured based on the block of pixels. The new control equation with MCDI is calculated per pixel:

If ((best_ME_SAD <= ***CAT_TH1***)

```
        If (Consistency check is passed && Smoothness check is passed)
                DItemp = MCDI;
        Else
                DItemp = sdi;
```

Else if (***CAT_TH1***<best_ME_SAD < CAT_TH2****30***) {

If (Consistency check is passed && Smoothness check is passed) AND

(SDI_angle =90 degree) AND

(best_ME_SAD + ***SAD_Tight_TH*30*** < Best_SAD_Angle_cost*2) AND

{(MCDI==median3(MCDI, $P_{curr\_seg}(x, y-1)$, $P_{curr\_seg}(x, y+1)$) ||

(Min[abs(MCDI - $P_{curr\_seg}(x, y-1)$), abs(MCDI - $P_{curr\_seg}(x, y+1)$)] <

NeighborPixel_TH)}

DItemp = MCDI;

Else

DItemp = sdi;

} Else

DItemp = sdi

Where ***CAT_TH1***(U2, default = 0), ***SAD_Tight_TH*** (U4, default=5) and ***NeighborPixel_TH***(U4, default=10) are state parameters. CAT_TH2 is a content adaptive value dependent on SCM. SCM = SHCM+SVCM from the spatial complexity measurement.

If (SCM < SCM_A)

CAT_TH2 = ***SAD_THA***;

Else if (SCM > SCM_B)

CAT_TH2 = ***SAD_THB***;

Else

CAT_TH2 = SCM / ***CAT_slope;***

Where ***CAT_slope*** (U4: default value 10). ***SAD_THA*** (U4, default 5) and ***SAD_THB*** (U4, default 10) are state parameters, and SCM_A and SCM_B are derived parameters:

SCM_A = ***CAT_slope * SAD_THA***; // 4-bit * 4-bit to produce 8-bit value

SCM_B = ***CAT_slope * SAD_THB***; // 4-bit * 4-bit to produce 8-bit value

4.7.5 Field Motion Detector

The Field Motion Detector is generated in either the EU or in the driver with a set of differences gathered across entire fields. It is used to detect when a non-interlaced source like a film has been converted to interlaced video – in this case there will be pairs of fields which can be put back together to make frames rather than interpolating. The variances for the block are sent to the CSunit to be summed across the entire frame. The CSunit will write the final values to memory on the flush at the end of the frame.

The blocks on the edge of the frame should not update the variances, because they tend to be noisy and not representative of the rest of the frame.

4.7.5.1 Simple Differences

The first set of variances are simply a sum of absolute pixel differences. The equations are done for every pixel with an even y coordinate:

variance[0] += Diff_cTpT = (c(x,y) – p(x,y)) ^ 2; – difference between pixels from the top fields of the current and previous frame.

variance[1] += Diff_cBpB = (c(x,y+1) – p(x,y+1)) ^ 2; – difference between pixels from the bottom fields of the current and previous frame.

variance[2] += Diff_cTcB = (c(x,y) – c(x,y+1)) ^ 2; – difference between pixels from the top field and bottom field in the current frame.

variance[3] += Diff_cTpB = (c(x,y) – p(x,y+1)) ^ 2; – difference between pixels from the top field of the current frame and bottom field of previous frame.

variance[4] += Diff_cBpT = (c(x,y+1) – p(x,y)) ^ 2; – difference between pixels from the bottom field of the current frame and top field of previous frame.

The variances summed for each 16x4 block are divided by 16 before adding them to the sum for the frame to make sure the frame-level sum fits in a 32-bit register.

4.7.5.2 Counter Variances

The rest of the variances are counters for variance conditions as described in the following code:

```
// Same field difference of the current frame
diff_cTcT = (c(x,y) - c(x,y+2)) ^ 2;
diff_cBcB = (c(x,y-1) - c(x,y+1)) ^ 2;

// Same field difference of the previous frame
diff_pTpT = (p(x,y) - p(x,y+2)) ^ 2;
diff_pBpB = (p(x,y-1) - p(x,y+1)) ^ 2;

// Same field vertical smoothness of the current frame
diff_cT = ABS(c(x,y) - c(x,y-2)) + ABS(c(x,y) - c(x,y+2)) -  ABS(c(x,y-2) + c(x,y+2));
diff_cB = ABS( c(x,y+1) - c(x,y-1) ) + ABS( c(x,y+1) - c(x,y+3) ) -
ABS( c(x,y-1) + c(x,y+3) );
```

```
if( diff_cTpT + diff_cBpB > fmd_tdiff ) {             // if moving pixels,

    // Fine tears for cadence detection except 2-2 detection
    if( diff_cTcB > diff_cTcT + diff_cBcB)            variance[5]++;
    else                                              variance[6]++;

    // Find tears for 2-2 cadence detection
    if( diff_cT < fmd_vdiff1 && diff_cB < fmd_vdiff1) {     // if fields are vertically
smooth,

variance[7]++;      // total moving pixels

// Find tears. (1st condition is to exclude very small variations)
if(diff_cTcB >=fmd_vdiff2 && diff_cTcB > diff_cTcT + diff_cBcB)    TEAR_1(x,y) = 1
if(diff_cTpB >=fmd_vdiff2 && diff_cTpB > diff_cTcT + diff_pBpB)    TEAR_2(x,y) = 1
if(diff_cBpT>=fmd_vdiff2 && diff_cBpT > diff_pTpT + diff_cBcB)     TEAR_3(x,y) = 1
    }
}
```

4.7.5.3 Tear Variances

The all 3 TEAR_N variables are compared to neighbors to eliminate strays:

```
if(TEAR_N(x-1,y) == 0 &&
   TEAR_N(x+1,y) == 0 &&
   TEAR_N(x,y-2) == 0 &&
   TEAR_N(x,y+2) == 0)            TEAR_N(x,y) = 0;    where N=1,2,3.

variance[8] = sum of TEAR_1(x,y)
variance[9] = sum of TEAR_2(x,y)
variance[10] = sum of TEAR_3(x,y)

if (variance[8] > variance[9] && variance[8] > variance[10])
            variance[7] = variance[8] = variance[9] = variance[10] = 0

if (variance[8] < fmd_thr_tear)    variance[8] = 0
if (variance[9] < fmd_thr_tear)    variance[9] = 0
if (variance[10] < fmd_thr_tear)  variance[10] = 0
```

The variances are summed for each block across the frame. The accumulators may require 24-bit adders if the differences are 8-bits and there can be 128 (horizontally) * 256 (vertically) of them. The sums are written to memory at the end of the frame.

Two sets of FMD variances are needed to support 2 simultaneous streams. The streams are distinguished by the dndi_stream_id state variable in the DI state.

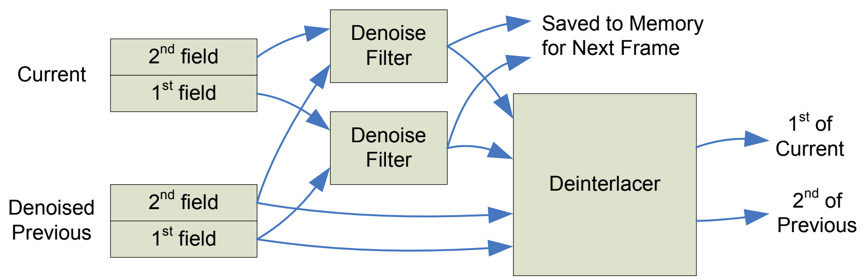
[DevILK] A-Stepping Erratum: TEAR_N compute doesn't follow the equation above. Two signals were missing, thus, it is incorrectly calculated as the following. Without the added protection of the N=-2 & N=4 collection of feature, the robustness of 2:2 detection suffers.

```
if(TEAR_N(x-1,y) == 0 &&
   TEAR_N(x+1,y) == 0 && )        TEAR_N(x,y) = 0;    where N=1,2,3.
```

4.7.6 Implementation Overview

4.7.6.1 Input and Output Frames

Two frames are needed to do deinterlacing, but for any two frames, two fields can be deinterlaced, doubling the output for the same input bandwidth. This also allows the denoise filter to only filter a frame once.



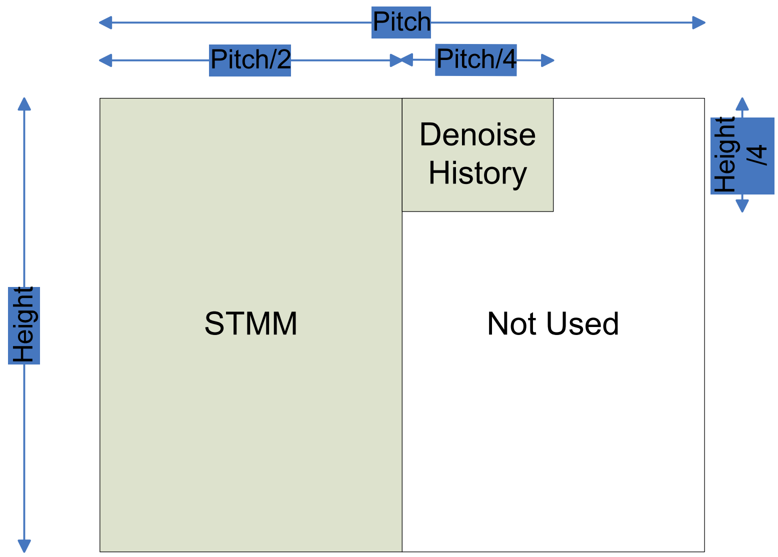
The above picture shows that two frames are read in, called current and previous. The two fields of the next frame are denoised using adjacent fields. The 2nd field of previous can be deinterlaced using current as the reference, and the 1st field of current can be deinterlaced using previous as reference.

Since we are producing 2 16x4 outputs, and the performance goal is to output 2 pixels per clock, we have 64 clocks to run 2 denoise filters and 2 deinterlacers.

The fields are referred to as 1st and 2nd because either the top or bottom field can be the first in the sequence depending on a state variable.

4.7.6.1.1 Statistics Surface Memory Format

The statistics memory page is used to store both STMM and Denoise history. The STMM and Denoise history are stored in separate areas addressed by a single base address pointer:



The STMM for any pixel pair is addressed by:

STMM_X = pixelX / 2

STMM_Y = pixelY

The Denoise History for any 4x4 block is addressed by

DH_X = Pitch/2 + pixelX/4

DH_Y = pixelY/4

Where the pixelX/Y comes from the address of the left pixel for STMM and the upper-left pixel for the Denoise History. The Pitch is from the surface state.

The read and write surfaces for each frame must be separate, since any individual block will not know if the neighbor blocks have been updated yet. This can be implemented as a ping-pong buffer pair with the write surface for each frame becoming the read surface for the next.

4.7.6.2 First Frame Special Case

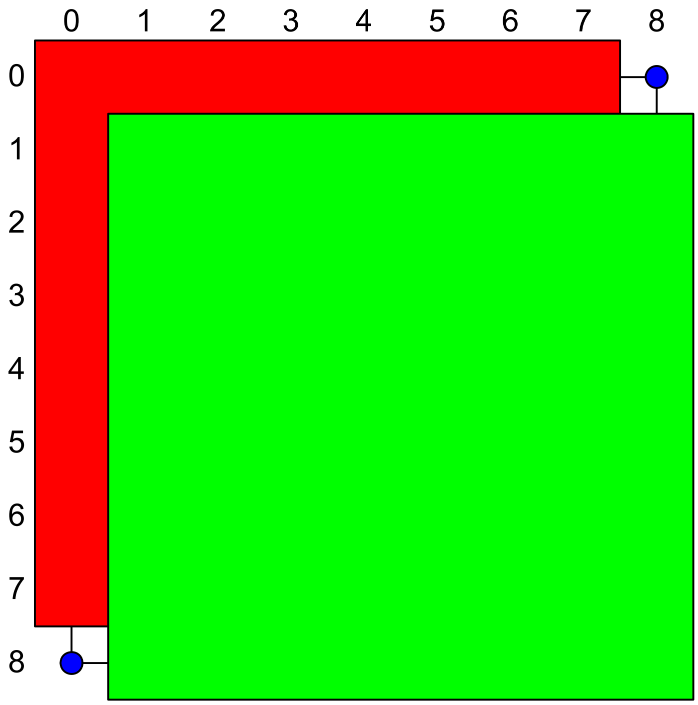
The first frame in the sequence is a special case for both denoise and deinterlace. Only data from the current frame address is read, the previous frame, clean previous, statistics and control addresses are ignored. Behavior for each function is as follows:

1) Denoise – The denoise filter needs to use the spatial filter, since there is no previous frame from which to do a temporal filter.
    a. The Denoise Motion History is not read.
    b. The blend between the temporal and spatial is forced to 100% spatial.
    c. The Denoise Motion History output values are written to mot_hist_init state variable.
2) BNE – The Block Noise Estimate only uses current frame values and so works normally.
3) Deinterlacer – Only the 1st field of the current frame frame is deinterlaced in this case – the 2nd of previous does not exist.
    a. The spatial deinterlacer is used to produce the output.
    b. The STMM input values are not read.
    c. The STMM output values are written as a the maximum 255 value so that the next frame is correctly told that spatial deinterlacing was used in this frame.
4) FMD – variances between the top and bottom of the current field should be output correctly. Variances that read from the previous field should indicate a maximum difference.
5) Progressive Cadence Reconstruction – the FMD input is not read, so always assume interlaced (is there ever a case where progressive should be assumed? If so maybe the control memory space should be used by the driver to indicate this).

4.8 Adaptive Video Scaler [DevILK+]

The adaptive video scaler consists of a pair of filters. The sharp filter is an 8x8 and the smooth filter is bilinear. The results of the two filters are alpha blended together using an alpha factor determined separately from an algorithm that examines the pixel values in the each vector.

There are a total of four different coefficient tables with two in each direction. For both directions is it possible to use either of the two tables that are assigned to it or use both at once with one table for the Y and the other table for the U/V. The coefficients are programmable by software and loaded via a new command streamer instruction. The coefficients are considered to be nonpipelined state, with a full pipeline flush being required before a new set of coefficients is loaded.



The above diagram shows two pixels (red and green) mapped onto a texture map, with the texel centers blue. The red/green boxes around the pixels indicate the area where the pixel would choose the same 8x8 footprint for its filter, while the large transparent box indicates the footprint for each pixel.

The u/v addresses for each pixel (in texel space) are as follows:

red pixel: u=3.3, v=3.3 (betau=0.3, betav=0.3)

green pixel: u=4.3, v=4.7 (betau=0.3, betav=0.7)

The integer u/v address of the upper left pixel of the footprint is a function of the pixel u/v address as follows:

u(UL) = floor(u(pix)) – 3

v(UL) = floor(v(pix)) – 3

When the 8x8 filter is selected, the 8x8 texel block surrounding the pixel sample point is selected. The blend factors "beta" (horizontal and vertical) are determined by the relative distance between the pixel center and the nearest 4 texels (2x2). The betas are first truncated to 5 bits (*i*).

The beta value is used to look up two sets of 8 coefficients, one set of 8 for horizontal (called K_h0..7), and one set of 8 for vertical (called K_v0..7).

4.8.1 Filtering Operations

There are two separate filters, sharp and smooth, which are blended in an adaptive manner.

4.8.1.1 Sharp Filter

The following formula is used to compute the filtered texture color for the sharp filter:

$$R0 = T00*K_h0 + T01*K_h1 + T02*K_h2 + T03*K_h3 + T04*K_h4 + T05*K_h5 + T06*K_h6 + T07*K_h7$$
$$R1 = T10*K_h0 + T11*K_h1 + T12*K_h2 + T13*K_h3 + T14*K_h4 + T15*K_h5 + T16*K_h6 + T17*K_h7$$
$$R2 = T20*K_h0 + T21*K_h1 + T22*K_h2 + T23*K_h3 + T24*K_h4 + T25*K_h5 + T26*K_h6 + T27*K_h7$$
$$R3 = T30*K_h0 + T31*K_h1 + T32*K_h2 + T33*K_h3 + T34*K_h4 + T35*K_h5 + T36*K_h6 + T37*K_h7$$
$$R4 = T40*K_h0 + T41*K_h1 + T42*K_h2 + T43*K_h3 + T44*K_h4 + T45*K_h5 + T46*K_h6 + T47*K_h7$$
$$R5 = T50*K_h0 + T51*K_h1 + T52*K_h2 + T53*K_h3 + T54*K_h4 + T55*K_h5 + T56*K_h6 + T57*K_h7$$
$$R6 = T60*K_h0 + T61*K_h1 + T62*K_h2 + T63*K_h3 + T64*K_h4 + T65*K_h5 + T66*K_h6 + T67*K_h7$$
$$R7 = T70*K_h0 + T71*K_h1 + T72*K_h2 + T73*K_h3 + T74*K_h4 + T75*K_h5 + T76*K_h6 + T77*K_h7$$
$$F' = R0*K_v0 + R1*K_v1 + R2*K_v2 + R3*K_v3 + R4*K_v4 + R5*K_v5 + R6*K_v6 + R7*K_v7$$
$$F\_sharp = \text{Clamp } F' \text{ to } [0.0, 1.0)$$

where:

- T*rc* is the texel color in row *r* ([0..3]) and column *c* ([0..3]) of the 8x8 array of neighboring texel colors
- F_sharp is the final output color of the sharp filter.

4.8.1.2 Smooth

The following formula is used to compute the filtered texture color for the smooth filter:

```
F_smooth = (T33 * (1-betaU) + T34 * betaU) * (1-betav) +  (T43 * (1-betaU) + T44 *
      betaU) * betav
```

4.8.1.3 Adaptive Filtering

The adaptive filter only supports RGB or YUV packed formats. For YUV formats, the alpha value is determined only by the Y channel (green), with this alpha value being applied to all three channels. For the RGB formats the alpha value is determined based on an average of all three channels with G having double the weight as the other channels.

Each horizontal or vertical filter has 8 texels input which feeds into an eight tap filter. On the center two there is a linear blend using the betaV. Then using the Y channel an adaptive part weight is calculated and the two filters are alpha blended. The adaptive part calculated on the Y channel is used on all three channels. Only the 8 MSBs are used in these calculations.

The adaptive part is done to classify a pixel as prone to ringing or not. This is done by analyzing the 8 Y samples from the interpolation window ($Wy_0 \ldots Wy_7$).

When the pixels are in an RGB format, Y is extracted from the RGB components in window W:

$$Wy_i = (Wr_i + 2*Wg_i + Wb_i)/4; \quad 0<=i<=7$$

There are 3 measurements on these samples that decide how to act. The result is a number between zero and one.

Analysis is performed on Y samples in 8 bit precision.

Measurement #1 –1st derivatives on center samples (minimum of 2 maximums).

```
maxDeriv4_a = max(|Wy3-Wy4|, |Wy2-Wy3|)
maxDeriv4_b = max(|Wy3-Wy4|, |Wy4-Wy5|)
maxDeriv4 = min(maxDeriv4_a, maxDeriv4_b)
```

Measurement #2 –2nd derivatives on center samples (minimum of 2 maximums).

```
Deriv1 = Wy2-Wy3; Derive2 = Wy3-Wy4; Deriv3 = Wy4-Wy5
Deriv2a = |Deriv1-Deriv2|
Deriv2b = |Deriv3-Deriv2|
Deriv2Avg = (Deriv2a + Deriv2b)/2
D4 = min(Deriv2Avg, maxDeriv4)
```

Measurement #3 – 1st derivative on all (8) Y samples.

$$maxDeriv8 = max(|Wy_m - Wy_{m+1}|);\ 0 \le m \le 6;$$

When *D4* is small enough and *maxDeriv8* is large enough then ringing can appear. So 2 alphas are calculated (one for *D4* and one for *maxDeriv8*), and the minimum of the two is used as the sharpness alpha. An alpha of 255 means the Polyphase scaler is used and an alpha of 0 means that the linear scaler is used.

$$D4Alpha = \begin{cases} D4 \le \mathbf{MaxDerivPoint4} & 0 \\ D4 \ge \mathbf{MaxDerivPoint4} + 2^{8-\mathbf{MaxDeriv4SlpBits}} & 255 \\ else & \left(D4 - \mathbf{MaxDerivPoint4}\right) \cdot 2^{8-\mathbf{MaxDeriv4SlpBits}} \end{cases}$$

$$D8Alpha = \begin{cases} maxDeriv8 \le \mathbf{MaxDerivPoint8} & 255 \\ maxDeriv8 \ge \mathbf{MaxDerivPoint8} + 2^{8-\mathbf{MaxDeriv8SlpBits}} & 0 \\ else & 255 - \left(\left(maxDeriv8 - \mathbf{MaxDerivPoint8}\right) \cdot 2^{8-\mathbf{MaxDeriv8Slp}}\right. \end{cases}$$

Note that multiplying by an exponent of 2 is implemented as bit shifts.

Calculate *SharpnessAlpha* (U0.8 precision):

```
SharpnessAlpha=max(D8alpha, D4Alpha)
if ((xDirection ? xAdaptiveBypass : YAdaptiveBypass) == 1) Then
       (SharpenessAlpha = SharpnessLevel)
```

$$Y_{out} = \begin{cases} SharpnessAlpha = 255 & Y_1 \\ else & Y_2 + \dfrac{128 + \left(Y - Y_1\right) \cdot SharpnessAlpha}{255} \end{cases}$$

The UV results are handled in the same manner.

4.9 Image Enhancement Filter and Video Signal Analysis [DevILK+]

The IEF module takes in the YUV 444 color space with 10 bit components.

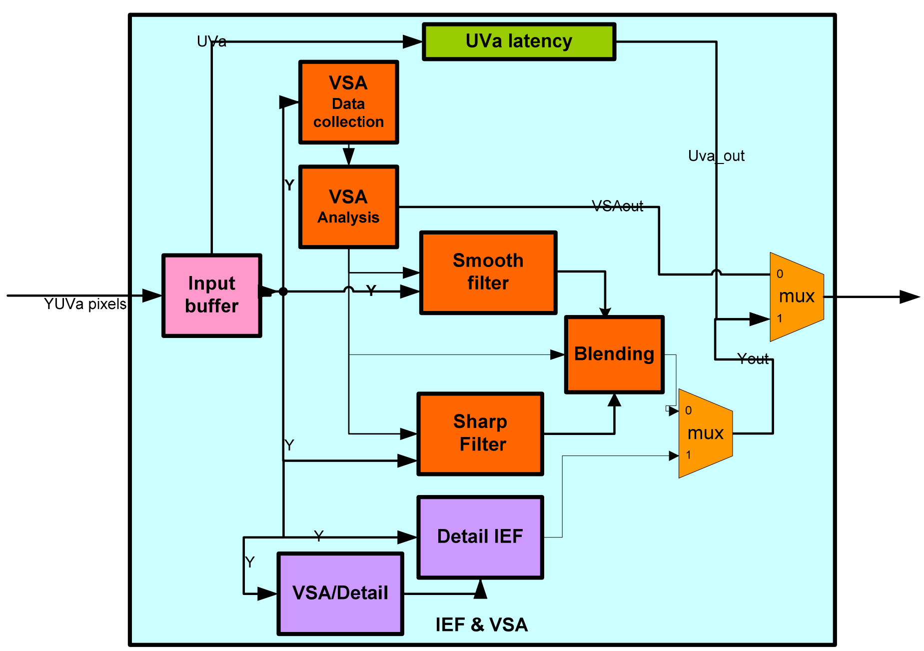
The IEF and VSA have 3 optional modes of operation: basic detail filter 3x3 mode, basic detail filter 5x5 mode and the combination mode. Detail Filter 3x3 mode which is a simple Sobel as VSA and 9 tap constant IEF. Detail Filter 5x5 mode which is a simple Sobel as VSA and 9 tap constant IEF on a sparse 5x5 environment. The combination mode is the full VSA mode and 25 tap filtering doing sharpening and/or smoothing. Either the detail filter mode or combination mode can be removed at synthesis.

VSA – Video Signal Analysis – analyzes the local Y environment of each pixel and outputs several values that describe its nature (smooth, detailed, sharpening). Those values will be used by the IEF to decide how the filter should be applied at each pixel location.

IEF – Image Enhancement Filter – The operations this filter performs are detail filter, smoothing and sharpening on the Y component, according to the VSA outputs.

The IEF throughput is 2 pixels per clock.

4.9.1 Block Diagram



1.10.2 Detail Filter Algorithm

4.9.1.1 VSA for Detail Filter

In the VSA for the detail flter mode, Sobel edge detection is used to set different weighting for detail filtering.

$$E\_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad E\_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The edge metric *(EM)* for the target pixel x is formulated as the convolution of the weighting with its 3x3 neighborhood *NH9(x)* as

ILK

*EM(x) = |NH9(x) * E_h| + |NH9(x) * E_v| // where the input is 10 bits, EM is 4 bits (*CLIP((|NH9(x) * E_h| + |NH9(x) * E_v|+8) >> 4 , 0 ,15))

If (EM(x) > **Strong_Edge_Threshold**) local_adjust = **Strong_Edge_Weight** // local_adjust is 3bits

Else if (EM(x) > **Weak_Edge_Threshold**) local_adjust = **Regular_Weight**

Else local_adjust = **Non_Edge_Weight**

The **Strong_Edge_Threshold, Weak_Edge_Threshold, Strong_Edge_Weight, Non_Edge_Weight** and **Regular_Weight** are the pipelined state variables to be specified by driver. **Strong_Edge_Threshold** & **Weak_Edge_Threshold** are 4-bit length variable for ILK.

Min and Max on the 3x3 neighborhood are found and diff3 = Max – Min is calculated. Similarly diff5 represents the difference calculated based on 5x5 neighborhood.

4.9.1.2 Detail IEF

In the mode of detail filter 3x3, the below 2-Dimensional formula is used to extract the high frequency component from the 3x3 neighborhood.

$$sigma(Xc)(2nd\_gradient) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

With the current pixel Xc with the 3x3 neighborhood below, the equation is

X1 X2 X3

X4 Xc X5

X6 X7 X8

Sigma(Xc)(2^{nd} Gradient) = 8 * Xc - (X1+X2+X3+X4+X5+X6+X7+X8) // 13 bits

In the mode of detail filter 5x5, the below 2-Dimensional formula is used to extract the high frequency component from the neighborhood.

$$sigma(Xc)(2nd\_gradient) = \begin{bmatrix} -1 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 8 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 0 & -1 \end{bmatrix}$$

The current pixel is Xc with the 5x5 neighborhood, the equation for 5x5 is

X0 X1 X2 X3 X4

X5 X6 X7 X8 X9

Xa Xb Xc Xd Xe

Xf Xg Xh Xi Xj

Xk Xl Xm Xn Xo

The basic equation is

*Sigma(Xc)(2nd_gradient)=8*Xc – (X0+X2+X4+Xa+Xe+Xk+Xm+Xo)* // 13 bits

The filter used here is the none-directional filter and so different coefficients can be applied to each of the outer 5x5 ring, where the middle pixel is subtracted from each pixel so the sum of the filter's coefficients is 0.

Clipping:

The clipping is utilized to limit the range of the calculated Sigma(Xc) to be among min_clip and max_clip.

```
min_clip = -1<< (5 + SrcPrecision - 8)

max_clip =  (1<< (5 + SrcPrecision - 8)) - 1

(SrcPrecision = 8 for 8-bit video, =10 for 10-bit video)

Thus, min_clip <= clipped(Sigma(Xc)) <= max_clip.
```

The **Gain_Factor** is the state variable specified by users, local adjust is the result of the VSA, diff3 is max-min in the 3x3 neighborhood. The equation below gives the delta from the original pixel:

Delta(Xc) = (clipped(sigma(Xc)) * gain_factor * local_adjust + 64) / (128*clipped(8+diff3)) (delta is 7 bits, and clipped(8+diff3) is between (0,255)

[ILK]

{In HW implementation.

Delta(Xc) = ((clipped(sigma(Xc)) * gain_factor * local_adjust + 64) * (m_DivTable[clipped(8+diff3)] +(1<< 7)) >> 8)}

The derived signal delta(Xc) is added to the pixel Xc for the filter result.

Clipping operation is then performed on the sum to ensure the output is within the allowable range of 0 - 1023.

4.9.2 Detail Filter Algorithm

4.9.2.1 VSA for Detail Filter

In the VSA for the detail flter mode, Sobel edge detection is used to set different weighting for detail filtering.

$$E\_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad E\_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The edge metric *(EM)* for the target pixel x is formulated as the convolution of the weighting with its 3x3 neighborhood *NH9(x)* as

The edge metric *(EM)* for the target pixel x is formulated as the convolution of the weighting with its 3x3 neighborhood *NH9(x)* as

ILK *EM(x) = |NH9(x) * E_h| + |NH9(x) * E_v| // where the input is 10 bits, EM is 4 bits (*CLIP((|NH9(x) * E_h| + |NH9(x) * E_v|+8) >> 4 , 0 ,15))

4.9.3 Cobination mode

4.9.3.1 VSA Analysis

In the VSA for the combo mode, the operation on the 5x5 neighborhood of the Y channel is assumed.

Diff (local contrast) is used as the main criteria. The local contrast result obtained from the diff criteria is fine tuned using global noise measure and other measurements from the VSA. Diff5 and diff3 are compared, because diff3 measures variability over a smaller region, it is multiplied by 3/2, the larger of the 2 is used as the basic parameter to estimate the smoothness strength. However if sharpness operation is performed the smaller of the 2 is used.

The mapping relation between filtering strength and the estimated variability is modeled using a piece wise linear (PWL) function to linearly interpolate the values among control points. The PWL parameters might vary depending on clip resolution, screen resolution, or other blocks in the video chain such as ACE. Using a PWL enables responding to specific clip features which will be measured by other modules (SW implemented).

8 points are used to divide the mapping range into 7 segments for PWL function. By default the value 0 is used as the Point 0 and the value 255 is used as the Point 7. Points 1 to 6 are specified by driver. Also, Slopes 0 to 6 and Bias 0 to 6 are specified by drivers. There are two sets of Point, Slope and Bias for the case of 3x3 and the case of 5x5. The pseudo code to implement PWL is as followed// (x[i],y[i]) and (x[i+1],y[i+1]) PWL(diff,PNT,BIAS5,SLP5)

```
PWL(diff,Point,Bias,Slope)

        if(Point[end] <= diff) //end =7 in this case

                i = end

        else

                find i  such that Point[i] <= diff <Point[i+1]

return Interpolation = MIN(MAX(((diff – Point[i]) * Slope[i])/8 + Bias[i]),0),255)
```

Gradient analysis

The gradient is defined to be derived based on 2x2 pixels. On a 5x5 neighborhood, there will be 16 (4x4) gredients for the overlapping 2x2 units. dx and dy are calculated using the below convolution masks

For dx

+1	-1
+1	-1

for dy

1	1
-1	-1

norm_grad = (abs(dx) + abs(dy)) is calculated on the 4x4 overlapping window.

And MaxNorm is the largest norm _grad in the 4x4 window.

Measurements of Multi-Ridge & Steepness

MR (multi ridge) is the ratio between the total of all norm_grad in the 4x4 window and the difference between minimum and maximum on the 5x5 window.

$$tot\_norm = \sum_{-2<i<2}^{-2<j<2} norm\_grad(i,j)$$

The total_norm is modified by the difference between minimum and maximum on the 5x5 window.

tot_norm -= 23* (max5 - min5)>>1; // zero if negative

MR = (5*(tot_norm / 8)) / (max5 - min5 +1) // 4 bit division

Dif5_mod = ((3*(max5 - min5))/8) +1

The norm is modified based on Dif5_mod

max_norm_mod= MAX(2*MaxNorm– Dif5_mod)/4,0) // 9.0u

Steepness = max_norm_mod/ Dif5_mod //4.0u. 4 bit division

Modify diff according to Global Noise Estimatiodenote

The GN1 is denoted as the Global Noise Estimation derived by software driver. The diff is modified based on the GN1 and the pixel intensity

modify_diff5 = diff5 – GN1

modify_diff3 = diff3 - (GN1 > 0? GN1: GN1/2),0))

diff = MAX(MIN(MAX(modify_diff5 , modify_diff3 + (modify_diff3)/2), 1),255) // 8.0u

if(diff > Pwl1_pnt3)

diff = MIN(modify_diff5 , modify_diff3 + (modify_diff3 >> 1)

The Weightings of Sharpening and smoothing strength

The weightings of sharpening and smoothening filter is based on the PWL conditioned on the modified diff.

Sharpening_strength = PWL(diff,PNT,BIAS5,SLP5) // 8.0u

Smoothing_strength = PWL(diff,PNT,BIAS3,SLP3) // 8.0u

And the sharpening weighting is further modified by the measurements of steepness and the multi-grid.

steepness = steepness   - MAX(8 -(diff3/2),0); // steepness disabled when diff is very low

Sharpening_strength = Sharpening_strength *(16 – MIN((MR – MR_Threshold)* MR_Boost + (steepness – Steepness_Threshold)* Steepness_Boost),15))/16 // 8.0u

Where MR_Threshold, MR_Boost, Steepness_Threshold and Steepness_Boost are the parameters specified by driver.

4.9.3.2 Sharpening Filtering

R5c	R5cx	R5x	R5cx	R5c
R5cx	R3c	R3x	R3c	R5cx
R5x	R3x		R3x	R5x
R5cx	R3c	R3x	R3c	R5cx
R5c	R5cx	R5x	R5cx	R5c

Non-directional IEF

The location of filter coefficients

The filter of the combinational mode is symmetric.

P(-2,-2)-P(0,0)	P(-1,-2)-P(0,0)	P(0,-2)-P(0,0)	P(1,-2)-P(0,0)	P(2,-2)-P(0,0)
P(-2,-1)-P(0,0)	P(-1,-1)-P(0,0)	P(0,-1)-P(0,0)	P(1,-1)-P(0,0)	P(2,-1)-P(0,0)
P(-2,0)-P(0,0)	P(-1,0)-P(0,0)		P(1,0)-P(0,0)	P(2,0)-P(0,0)
P(-2,1)-P(0,0)	P(-1,1)-P(0,0)	P(0,1)-P(0,0)	P(1,1)-P(0,0)	P(2,1)-P(0,0)
P(-2,2)-P(0,0)	P(-1,2)-P(0,0)	P(0,2)-P(0,0)	P(1,2)-P(0,0)	P(2,2)-P(0,0)

D(i,j) = P(i,j)-P(0,0) as the difference of the target (center) pixel, P(0,0), from the neighboring pixels, P(i,j), shown in the above figure.

Sharp = R5C*(D(2,0) + D(-2,0) + D(0,-2) + D(0,2)) +

R5X*(D(2,2) + D(-2,2) + D(2,-2) + D(-2,-2)) +

R5CX*(D(2,1) + D(-2,1) + D(1,-2) + D(-1,-2) + D(-2,-1) + D(2,-1) + D(1,2) + D(-1,2))

R5C, R5X and R5CX are the paramters specified by driver.

4.9.3.3 Smoothing Filter

Similar to the content adaptive spatial filter in Section 1.8.2.4, smoothing filter is using only neighboring pixels whose value is close to the center pixel value. Global noise is used as a threshold to decide if a pixel value is close to the center pixel. Only pixels whose distance from the center pixel is less than the global noise are used for smoothing.

For each pixel in the 3x3 neighborhood:

If(D(i,j) < GN1) D(i,j) = D(i,j)

Else D(i,j) = 0

The number of pixels that are not zeroed are counted for the coefficient R3C & R3X individually as NZC and NZX. The factor (NZC, NZX) is then multiplied by each coefficient depending on how many pixels it multiplies. The pseudo code to derive NZC and NZX are as follows.

```
NZX = 0

NZC = 0

For (-2 <= i, j <=2) {

    If (ABS(D(i,j) < GN1) {

        If (i==0 || j==0) NZC ++;

        Else NZX ++;

    }

}
```

Apply smoothing operation

```
Smooth = R3C*(D(1,0) + D(-1,0) + D(0,-1) + D(0,1)) * NZ[NZC] +

    R3X*( D(1,1) + D(-1,1) + D(1,-1) + D(-1,-1)) * NZ[NZX]  // 12.2u round 3 lsb, check for overflow
```

4.9.3.4 Filter Blending

Smoothing filter reduces the power of some or all of the frequencies in the image, while sharpening filter enhance some of the frequencies in the image. The output of filtering is based on the blending of both filterings.

```
Filtering  = -sharp_strength * Sharp + smooth_strength * Smooth // 11.0s round 10bits, check for overflows

Output_pixel = orginal_pixel + filtering // 10.0u
```

Limiting the Output Pixel

The limiter is applied to constrain the effect of overshoot and undershoot.

```
If (Output_pixel > max5)

    Output_pixel =( Output_pixel -  max5) * Maximum_Limiter + max5

    Output_pixel = MIN(max5 + Clip_Limiter + ((max5 - max3)*Limiter_Boost), Output_pixel);

else if(Output_pixel < max5)

    Output_pixel = min5 - ( min5 - Output_pixel)* Minimum_Limiter

    Output_pixel =  MAX(min5 - (Clip_Limiter + ((min5 - min3) *Limiter_Boost)), Output_pixel)
```

Maximum_Limiter, Minimum_Limiter, Limiter_Boost and Clip_Limiter are the parameters specified by driver.

4.10 State

4.10.1 BINDING_TABLE_STATE

The binding table binds surfaces to logical resource indices used by shaders and other compute engine kernels. It is stored as an array of up to 256 elements, each of which contains one dword as defined here. The start of each element is spaced one dword apart. The first element of the binding table is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31:5	**Surface State Pointer.** This 32-byte aligned address points to a surface state block. This pointer is relative to the **Surface State Base Address**. **[DevBW-A,B] Errata BWT007:** Surface State data pointed at by offsets from Surface State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ

4.10.2 SURFACE_STATE

The surface state is stored as individual elements, each with its own pointer in the binding table. Each surface state element is aligned to a 32-byte boundary.

Surface state defines the state needed for the following objects:

- texture maps (1D, 2D, 3D, cube) read by the sampling engine
- buffers read by the sampling engine
- constant buffers read by the data cache via the data port
- render targets read/written by the render cache via the data port
- streamed vertex buffer output written by the render cache via the data port
- media surfaces read from the texture cache or render cache via the data port
- media surfaces written to the render cache via the data port

4.10.2.1 For most messages

<table>
<tr><td rowspan="2">0</td><td>31:29</td><td>
Surface Type
Project: All
Format: U3 enumerated type FormatDesc
This field defines the type of the surface.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>SURFTYPE_1D</td><td>Defines a 1-dimensional map or array of maps</td><td>All</td></tr>
<tr><td>1h</td><td>SURFTYPE_2D</td><td>Defines a 2-dimensional map or array of maps</td><td>All</td></tr>
<tr><td>2h</td><td>SURFTYPE_3D</td><td>Defines a 3-dimensional (volumetric) map</td><td>All</td></tr>
<tr><td>3h</td><td>SURFTYPE_CUBE</td><td>Defines a cube map or array of cube maps</td><td>All</td></tr>
<tr><td>4h</td><td>SURFTYPE_BUFFER</td><td>Defines an element in a buffer</td><td>All</td></tr>
<tr><td>5h-6h</td><td>Reserved</td><td></td><td>All</td></tr>
<tr><td>7h</td><td>SURFTYPE_NULL</td><td>Defines a null surface</td><td>All</td></tr>
</table>
Programming Notes
A null surface will be used in instances where an actual surface is not bound. When a write message is generated to a null surface, no actual surface is written to. When a read message (including any sampling engine message) is generated to a null surface, the result is all zeros. All of the remaining fields in surface state are ignored for null surfaces, with the following exceptions:

[PreDevGT]: Width, Height, Depth, LOD, MIP Map Layout Mode, and Render Target View Extent fields must match the depth buffer’s corresponding state for all render target surfaces, including null.
Surface Format must be R8G8B8A8_UNORM.

The Surface Type of a surface used as a render target (accessed via the Data Port’s Render Target Write message) must be the same as the Surface Type of all other render targets and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless either the depth buffer or render targets are SURFTYPE_NULL.
</td></tr>
<tr><td>28</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>

<table>
<tr>
<td></td>
<td>27</td>
<td>
Data Return Format
Project: All
Format: U1 enumerated type FormatDesc
For Sampling Engine Surfaces, [DevBW] and [DevCL] only:
This field determines the format of the return data from the sampling engine to the compute engine, but only if the Data Return Format field in the message descriptor is set to FLOAT32. This field is ignored for surfaces used by other units.
For Other Surfaces:
This field is ignored.
For [DevCTG+] Sampling Engine surfaces, the state of this bit is effectively DATA_RETURN_FLOAT32 regardless of its programmed value.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>DATA_RETURN_FLOAT32</td><td>FLOAT32 data is returned</td><td>All</td></tr>
<tr><td>1h</td><td>DATA_RETURN_S1.14</td><td>S1.14 fixed point data is returned</td><td>[DevBW], [DevCL]</td></tr>
</table>
Programming Notes
The S1.14 return format is only legal for returning data from normalized (UNORM, or SNORM) map formats where all channels have <= 8 bits. It is not legal to use this format with any floating point or integer map format.
S1.14 return format is only used for SIMD16 and SIMD8 messages from the sampling engine. For SIMD4x2 messages, FLOAT32 format will be used for surfaces specifying S1.14 data return format.
Data returned in format S1.14 will be converted to FLOAT32 before reaching the GRF register, thus the state of this bit does not affect the kernel.
It is recommended that S1.14 format be used wherever it is legal, as the performance will generally be improved.
</td>
</tr>
</table>

	26:18	**Surface Format** Project: All Format: U9 FormatDesc Specifies the format of the surface or element within this surface. This field is ignored for all data port messages other than the render target message and streamed vertex buffer write message. Some forms of the media block messages use the surface format. Refer to the table in section 4.10.2.1 for the formats supported and their encodings. **Programming Notes** **Tile Walk** TILEWALK_YMAJOR is UNDEFINED for *render target* formats that have 128 bits-per-element (BPE). YUV (YCRCB) surfaces used as render targets can only be rendered to using 3DPRIM_RECTLIST with even X coordinates on all of its vertices, and the pixel shader cannot kill pixels. If **Number of Multisamples** is set to a value *other than* MULTISAMPLECOUNT_1, this field cannot be set to the following formats: • any format with greater than 64 bits per element • any compressed texture format (BC*) • any YCRCB* format **Errata** / **Description** / **Project** # / surfaces with FLOAT format are not supported. / [DevBW-A,B]
	17:14	**Color Buffer Component Write Disables** Project: [Pre-DevGT] Format: U4 bit mask of disables (0 or logical OR of any of the enumerated values) FormatDesc **For Render Target Surfaces:** This field contains a bitmask that controls the writing of individual color components into the Color Buffer. If a component is disabled (bit set) writes to the color buffer will not modify that component. If enabled (bit clear), that component can be overwritten. **For Other Surfaces:** this field is ignored. **Value** / **Name** / **Description** / **Project** 1000b / WRITEDISABLE_ALPHA / / All 0100b / WRITEDISABLE_RED / / All 0010b / WRITEDISABLE_GREEN / / All 0001b / WRITEDISABLE_BLUE / / All **Programming Notes** For YUV surfaces, this field must be set to 0000B (all channels enabled). **[DevCTG+]:** For render targets accessed with the Render Target UNORM Write message, this field is ignored (all component writes are enabled) **Errata** / **Description** / **Project** # / Desc / All

	13	**Color Blend Enable** Project: [Pre-DevGT] Format: Enable FormatDesc **For Render Target Surfaces:** Specifies that color blend is enabled for this particular render target. The Color Buffer Blend Enable state in COLOR_CALC_STATE provides global control over blending. See Color Buffer Blending (Windower) for details. **For Other Surfaces:** this field is ignored. **Errata #** | **Description** | **Project** This **Color Blend Enable** bit is not used, and acts as if it is ENABLED for each RenderTarget. Blending is enabled or disabled only a a global basis by the **Color Buffer Blend Enable** state variable in COLOR_CALC_STATE. | [DevBW-A,B]
	12	**Vertical Line Stride** Project: All Format: U1 in lines to skip between logically adjacent lines FormatDesc **For 2D Non-Array Surfaces accessed via the Sampling Engine or Data Port:** Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures. **For Other Surfaces:** Vertical Line Stride must be zero. **Programming Notes** This bit must not be set if the surface format is a compressed type (BCn*). If this bit is set on a sampling engine surface, texture addess control modes cannot be set to any mode other than TEXCOORDMODE_CLAMP and the mip mode filter must be set to MIPFILTER_NONE.
	11	**Vertical Line Stride Offset** Project: All Format: U1 in lines of initial offset (when Vertical Line Stride == 1) FormatDesc **For 2D Non-Array Surfaces accessed via the Sampling Engine or Data Port:** Specifies the offset of the initial line from the beginning of the buffer. Ignored when Vertical **Line Stride** is 0. **For Other Surfaces:** Vertical Line Stride Offset must be zero.

	10	**MIP Map Layout Mode** Project: All Format: U1 enumerated type FormatDesc **For 1D and 2D Surfaces and** **For Cube Surfaces** (ILK only): This field specifies which MIP map layout mode is used, whether the map for LOD 1 is stored to the right of the LOD 0 map, or stored below it. See Memory Data Formats for details on the specifics of each layout mode. **For Other Surfaces:** This field is reserved : MBZ

Value	Name	Description	Project
0h	MIPLAYOUT_BELOW		All
1h	MIPLAYOUT_RIGHT		All

Programming Notes

MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces

Errata	Description	Project
#	MIPLAYOUT_RIGHT is not supported with “ld” sampler message	[DevBW], [DevCL]
#	MIPLAYOUT_RIGHT is not supported with sample_c/sample_l_c/sample_b_c sampler messages.	[DevCL]

	9	**Cube Map Corner Mode** Project: All Format: U1 enumerated type FormatDesc **For Cube Surfaces accessed by the Sampling Engine:** When filtering at the corner of cube map one of the four texels does not exist. This field specifies if it gets replaced with the opposite corner texel or the average of all three that exist. **For Other Surfaces:** This field is Reserved : MBZ

Value	Name	Description	Project
0h	CUBE_REPLICATE		All
1h	CUBE_AVERAGE		[ILK]

Programming Notes

CUBE_AVERAGE may only be selected if all of the **Cube Face Enable** fields are equal to one.

[Pre-ILK]: Only CUBE_REPLICATE is supported.

ChromaKey Enable must not be set in CUBE_AVERAGE mode

	8	**Render Cache Read Write Mode** Project: All Format: U1 enumerated type FormatDesc **For Surfaces accessed via the Data Port to Render Cache:** This field specifies the way Render Cache treats a write request. If unset, Render Cache allocates a write-only cache line for a write miss. If set, Render Cache allocates a read-write cache line for a write miss. **For Surfaces accessed via the Sampling Engine or Data Port to Texture Cache or Data Cache:** This field is reserved : MBZ **Value** / **Name** / **Description** / **Project** 0h / / Allocating write-only cache for a write miss / All 1h / / Allocating read-write cache for a write miss / All **Programming Notes** This field is provided for performance optimization for Render Cache read/write accesses (from Gen4 EU's point of view). **Errata** / **Description** / **Project** # / This field must be set to 0h. / [DevBW-A,B]

<table>
<tr><td></td><td>7:6</td><td>
Media Boundary Pixel Mode
Project: All
Format: U2 enumerated type FormatDesc
For 2D Non-Array Surfaces accessed via the Data Port Media Block Read Message:
This field enables control of which rows are returned on vertical out-of-bounds reads using the Data Port Media Block Read Message. In the description below, frame mode refers to Vertical Line Stride = 0, field mode is Vertical Line Stride = 1 in which only the even or odd rows are addressable. The frame refers to the entire surface, while the field refers only to the even or odd rows within the surface. Refer to section 5.6.1.1 for more details.
For Other Surfaces:
Reserved : MBZ
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>NORMAL_MODE</td><td>the row returned on an out-of-bound access is the closest row in the frame or field. Rows from the opposite field are never returned.</td><td>All</td></tr>
<tr><td>1h</td><td>Reserved</td><td></td><td>All</td></tr>
<tr><td>2h</td><td>PROGRESSIVE_FRAME</td><td>the row returned on an out-of-bound access is the closest row in the frame, even if in field mode.</td><td>[DevCTG+]</td></tr>
<tr><td>3h</td><td>INTERLACED_FRAME</td><td>in field mode, the row returned on an out-of-bound access is the closest row in the field. In frame mode, even out-of-bound rows return the nearest even row while odd out-of-bound rows return the nearest odd row.</td><td>[DevCTG+]</td></tr>
</table>
Programming Notes
[DevBW] and [DevCL]: Only NORMAL_MODE is supported.
</td></tr>
</table>

	5:0	**Cube Face Enables** Project: All Format: U6 bit mask of enables FormatDesc **For SURFTYPE_CUBE Surfaces accessed via the Sampling Engine:** Bits 5:0 of this field enable the individual faces of a cube map. Enabling a face indicates that the face is present in the cube map, while disabling it indicates that that face is represented by the texture map's border color. Refer to Memory Data Formats for the correlation between faces and the cube map memory layout. Note that storage for disabled faces must be provided. **For other surfaces:** This field is reserved : MBZ

Value	Name	Description	Project
100000b		-X face	All
010000b		+X face	All
001000b		-Y face	All
000100b		+Y face	All
000010b		-Z face	All
000001b		+Z face	All

Programming Notes

When TEXCOORDMODE_CLAMP is used when accessing a cube map, this field must be programmed to 111111b (all faces enabled).

This field is ignored unless the **Surface Type** is SURFTYPE_CUBE.

1	31:0	**Surface Base Address** Project: All Format: GraphicsAddress[31:0] FormatDesc Specifies the byte-aligned base address of the surface. **Programming Notes** For SURFTYPE_BUFFER render targets, this field specifies the base address of first element of the surface. The surface is interpreted as a simple array of that single element type. The address must be naturally-aligned to the element size (e.g., a buffer containing R32G32B32A32_FLOAT elements must be 16-byte aligned). For SURFTYPE_BUFFER non-rendertarget surfaces, this field specifies the base address of the first element of the surface, computed in software by adding the surface base address to the byte offset of the element in the buffer. Mipmapped, cube and 3D sampling engine surfaces are stored in a “monolithic” (fixed) format, and only require a single address for the base texture. Linear depth buffer surface base addresses must be 64-byte aligned. Note that while render targets (color) can be SURFTYPE_BUFFER, depth buffers cannot. Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from **Surface Base Address** are tiled, **Surface Base Address** itself is not transformed using the tiling algorithm. **[DevCTG+]:** For tiled surfaces, the actual start of the surface can be offset from the **Surface Base Address** by the **X Offset** and **Y Offset** fields. Certain message types used to access surfaces have more stringent alignment requirements. Please refer to the specific message documentation for additional restrictions.

<table>
<tr>
<td>2</td>
<td>31:19</td>
<td>
Height
Project: All
Format: U13 FormatDesc
Range SURFTYPE_1D: must be zero
SURFTYPE_2D: height of surface – 1 (y/v dimension) [0,8191]
SURFTYPE_3D: height of surface – 1 (y/v dimension) [0,2047]
SURFTYPE_CUBE: height of surface – 1 (y/v dimension) [0,8191]
SURFTYPE_BUFFER: contains bits [19:7] of the number of entries in the buffer – 1 [0,8191]
This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. For buffers, this field specifies a portion of the buffer size.

Programming Notes
For buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27}. After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the Height, Width, and Depth fields as indicated, right-justified in each field. Unused upper bits must be set to zero.
If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frame
The Height of a render target must be the same as the Height of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).
<table>
<tr><th>Errata</th><th>Description</th><th>Project</th></tr>
<tr><td>#</td><td>The number of entries in a SURFTYPE_BUFFER is restricted to 2^27 – 1</td><td>[DevBW-A,B]</td></tr>
</table>
</td>
</tr>
</table>

	18:6	**Width** Project: All Format: U13 FormatDesc Range SURFTYPE_1D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_2D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_3D: width of surface – 1 (x/u dimension) [0,2047] SURFTYPE_CUBE: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_BUFFER: contains bits [6:0] of the number of entries in the buffer – 1 [0,127] This field specifies the width of the surface. If the surface is MIP-mapped, this field specifies the width of the base MIP level. The width is specified in units of pixels or texels. For buffers, this field specifies a portion of the buffer size. For surfaces accessed with the Media Block Read/Write message, this field is in units of DWords. **Programming Notes** For surface types other than SURFTYPE_BUFFER, the Width specified by this field must be less than or equal to the surface pitch (specified in bytes via the **Surface Pitch** field). For cube maps, Width must be set equal to the Height. For MONO8 textures, Width must be a multiple of 32 texels. The **Width** of a render target must be the same as the **Width** of the other render target(s) and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped). The **Width** of a render target with YUV surface format must be a multiple of 2.

	5:2	**MIP Count / LOD** Project: All Format: Sampling Engine Surfaces: U4 in (LOD units – 1) FormatDesc Render Target Surfaces: U4 in LOD units Range Sampling Engine Surfaces: [0,13] representing [1,14] MIP levels Render Target Surfaces: [0,13] representing LOD Other Surfaces: [0] **For Sampling Engine Surfaces:** This field indicates the number of MIP levels allowed to be accessed starting at **Surface Min LOD**, which must be less than or equal to the number of MIP levels actually stored in memory for this surface. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here. **For Render Target Surfaces:** This field defines the MIP level that is currently being rendered into. This is the absolute MIP level on the surface and is not relative to the **Surface Min LOD** field, which is ignored for render target surfaces. **For Other Surfaces:** This field is reserved : MBZ

Value	Name	Description	Project
0h	Disable	Desc	All
1h	Enable	Desc	All

Programming Notes

The **LOD** of a render target must be the same as the **LOD** of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).

For render targets with YUV surface formats, the **LOD** must be zero.

Errata	Description	Project
#	Desc	All

	1:0	**Render Target Rotation** Project: [DevCTG+] Format: U2 enumerated type FormatDesc

For Render Target Surfaces:

This field specifies the rotation of this render target surface when being written to memory.

For Other Surfaces:
This field is ignored.

[DevBW, DevCL]: Reserved : MBZ

Value	Name	Description	Project
0h	RTROTATE_0DEG	No rotation (0 degrees)	All
1h	RTROTATE_90DEG	Rotate by 90 degrees	All
2h	Reserved		All
3h	RTROTATE_270DEG	Rotate by 270 degrees	All

Programming Notes

Rotation is not supported for render targets of any type other than simple, non-mip-mapped, non-array 2D surfaces. The surface must be using tiled with X major.

Width and **Height** fields apply to the dimensions of the surface before rotation.

For 90 and 270 degree rotated surfaces, the **Height** (rather than the **Width**) must be less than or equal to the **Surface Pitch** (specified in bytes).

For 90 and 270 degree rotated surfaces, the actual **Height** and **Width** of the surface in pixels (not the field value which is decremented) must both be even.

Rotation is supported only for surfaces with the following surface formats: B5G6R5_UNORM, B5G6R5_UNORM_SRGB, R8G8B8[A|X]8_UNORM, R8G8B8[A|X]8_UNORM_SRGB, B8G8R8[A|X]8_UNORM, B8G8R8[A|X]8_UNORM_SRGB, B10G10R10[A|X]2_UNORM, B10G10R10A2_UNORM_SRGB, R10G10B10A2_UNORM, R10G10B10A2_UNORM_SRGB, R16G16B16A16_FLOAT, R16G16B16X16_FLOAT

3	31:21	**Depth** Project: All Format: U11 FormatDesc Range SURFTYPE_1D: number of array elements – 1 [0,511] SURFTYPE_2D: number of array elements – 1 [0,511] SURFTYPE_3D: depth of surface – 1 (z/r dimension) [0,2047] SURFTYPE_CUBE: number of array elements – 1 [see programming notes for range] SURFTYPE_BUFFER: contains bits [26:20] of the number of entries in the buffer – 1 [0,127] This field specifies the total number of levels for a volume texture or the number of array elements allowed to be accessed starting at the **Minimum Array Element** for arrayed surfaces. If the volume texture is MIP-mapped, this field specifies the depth of the base MIP level. For buffers, this field specifies a portion of the buffer size. **Programming Notes** The **Depth** of a render target must be the same as the **Depth** of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER). For SURFTYPE_CUBE: **[Pre-DevGT]:** for all cube surfaces, this field must be zero as cube arrays are not supported.
	20	**Reserved** Project: All Format: MBZ
	19:3	**Surface Pitch** Project: All Format: U17 pitch in (#Bytes – 1) FormatDesc Range For surfaces of type SURFTYPE_BUFFER: [0,2047] -> [1B, 2048B] For other linear surfaces: [0, 131071] -> [1B, 128KB] For X-tiled surface: [511, 131071] –> [512B, 128KB] = [1tile, 256 tiles] For Y-tiled surfaces: [127, 131071]->[128B,128KB] = [1 tile, 1024 tiles] This field specifies the surface pitch in (#Bytes - 1). For surfaces of type SURFTYPE_BUFFER, this field indicates the size of the structure. **Programming Notes** For linear *render target* surfaces, the pitch must be a multiple of the element size for non-YUV surface formats. Pitch must be a multiple of 2 * element size for YUV surface formats. For other linear surfaces, the pitch can be any multiple of bytes. For tiled surfaces, the pitch must be a multiple of the tile width.
	2	**Reserved** Project: All Format: MBZ

1

Tiled Surface

Project: All

Format: U1 enumerated type FormatDesc

This field specifies whether the surface is tiled.

Value	Name	Description	Project
0h	FALSE	Linear surface	All
1h	TRUE	Tiled surface	All

Programming Notes

Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory.

The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.

If **Surface Type** is SURFTYPE_BUFFER, this field must be FALSE (buffers are supported only in linear memory)

If the target cache via the Data Port is the Data Cache, this field must be disabled (zero). The data cache only supports access to linear memory.

If **Surface Type** is SURFTYPE_NULL, this field must be TRUE

0

Tile Walk

Project: All

Format: U1 enumerated type FormatDesc

This field specifies the type of memory tiling (XMajor or YMajor) employed to tile this surface. See *Memory Interface Functions* for details on memory tiling and restrictions.

Value	Name	Description	Project
0h	TILEWALK_XMAJOR	X major tiling	All
1h	TILEWALK_YMAJOR	Y major tiling	All

Programming Notes

Refer to *Memory Data Formats* for restrictions on *TileWalk* direction for the various buffer types. (Of particular interest is the fact that YMAJOR tiling is **not** supported for display/overlay buffers).

The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.

Use of TILEWALK_YMAJOR is UNDEFINED for render target formats that have 128 bits-per-element (BPE).

This field is ignored when the surface is linear.

<table>
<tr><td rowspan="2">4</td><td>31:28</td><td>Surface Min LOD
Project: All
Format: U4 in LOD units FormatDesc
Range [0,13]
For Sampling Engine Surfaces:
This field indicates the most detailed LOD that can be accessed as part of this surface. This field is added to the delivered LOD (sample_l, ld, or resinfo message types) before it is used to address the surface.
For Other Surfaces:
This field is ignored.

Programming Notes
This field must be zero if the Surface Format is MONO8
[DevBW-A,B]: this field must be zero</td></tr>
<tr><td>27:17</td><td>Minimum Array Element
Project: All
Format: U11 FormatDesc
Range 1D/2D/cube surfaces: [0,511]
3D surfaces: [0,2047]
For Sampling Engine and Render Target 1D and 2D Surfaces:
This field indicates the minimum array element that can be accessed as part of this surface. This field is added to the delivered array index before it is used to address the surface.
For Render Target 3D Surfaces:
This field indicates the minimum ‘R’ coordinate on the LOD currently being rendered to. This field is added to the delivered array index before it is used to address the surface.
For Sampling Engine Cube Surfaces:
This field must be set to zero.

<table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>#</td><td>This field must be zero.</td><td>[DevBW-A,B]</td></tr><tr><td>#</td><td>For sample_c/sample_b_c/sample_l_c instructions this field is ignored. If it is tiled surface and not at a 4k boundary it must be copied to a 4k aligned surface. Then for any case it must be pointed to by the Surface Base Address.</td><td>[DevBW-A,B,C,D], [DevCL-A,B]</td></tr></table></td></tr>
</table>

	16:8	**Render Target View Extent** Project: All Format: U9 FormatDesc Range [0,511] to indicate extent of [1,512] **For Render Target 3D Surfaces:** This field indicates the extent of the accessible ‘R’ coordinates minus 1 on the LOD currently being rendered to. **For Render Target 1D and 2D Surfaces:** This field must be set to the same value as the **Depth** field. **For Other Surfaces:** This field is ignored.
	7	**Reserved** Project: All Format: MBZ
	3	**Reserved** Project: All Format: MBZ
5	31:25	**X Offset** Project: [DevCTG+] Format: PixelOffset[8:2] FormatDesc Range TileX surfaces: [0,ceil(512/BytesPerElement)4] in multiples of 4 (low 2 bits missing) TileY surfaces: [0,ceil(128/BytesPerElement)-4] in multiples of 4 (low 2 bits missing) This field specifies the horizontal offset in pixels from the **Surface Base Address** to the start (origin) of the surface. This field effectively loosens the alignment restrictions on the origin of tiled surfaces. Previously, tiled surface origin was (by definition) located at the base address, and thus needed to satisfy the 4KB base address alignment restriction. Now the origin can be specified at a finer (4-wide x 2-high pixel) resolution. **Programming Notes** For linear surfaces, this field must be zero For surfaces accessed with the Data Port Media Block Read/Write message, the pixel size is assumed to be 32 bits in width For **Surface Format** with other than 8, 16, 32, 64, or 128 bits per pixel, this field must be zero. If **Render Target Rotation** is set to other than RTROTATE_0DEG, this field must be zero.

<table>
<tr>
<td></td>
<td>24</td>
<td>
Surface Vertical Alignment
Project: [DevGT+]
Format: U1 enumerated type FormatDesc
For Sampling Engine Uncompressed and Render Target Surfaces:
This field specifies the vertical alignment requirement for the surface. Refer to the “Memory Data Formats” chapter for details on how this field changes the layout of the surface in memory. This field applies to surface formats other than compressed formats.
For Other Surfaces:
This field is ignored.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>VALIGN_2</td><td>Vertical alignment factor j = 2</td><td>All</td></tr>
<tr><td>1h</td><td>VALIGN_4</td><td>Vertical alignment factor j = 4</td><td>All</td></tr>
</table>
Programming Notes
This field is intended to be set to VALIGN_4 if the surface was rendered as a depth buffer, for a multisampled (4x) render target, or for a multisampled (8x) render target, since these surfaces support only alignment of 4. Use of VALIGN_4 for other surfaces is supported, but increases memory usage.
</td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>
Y Offset
Project: [DevCTG+]
Format: RowOffset[4:1] FormatDesc
Range TileX surfaces: [0,6] in multiples of 2 (low bit missing)
TileY surfaces: [0,30] in multiples of 2 (low bit missing)
This field specifies the vertical offset in rows from the Surface Base Address to the start of the surface. (See additional description in the X Offset field)
Programming Notes
For linear surfaces, this field must be zero.
For render targets in which the Render Target Array Index is not zero, this field must be zero.
For Surface Format with other than 8, 16, 32, 64, or 128 bits per pixel, this field must be zero.
If Render Target Rotation is set to other than RTROTATE_0DEG, this field must be zero.
[ILK]: For surfaces accessed in field mode (Vertical Line Stride = 1 or equivalent Media Block Read/Write message override), this field must be set to a multiple of 4.
<table>
<tr><th>Errata</th><th>Description</th><th>Project</th></tr>
<tr><td>#</td><td>For surfaces accessed in field mode (Vertical Line Stride = 1 or equivalent Media Block Read/Write message override), the Y offset value must be divided by 2 when setting this field.</td><td>[DevCTG], [DevEL]</td></tr>
</table>
</td>
</tr>
<tr>
<td></td>
<td>15:0</td>
<td>Reserved Project: All Format: MBZ</td>
</tr>
</table>

4.10.2.1.1

4.10.2.1.2 Surface Formats

The following table indicates the supported surface formats and the 9-bit encoding for each. Note that some of these formats are used not only by the Sampling Engine, but also by the Data Port and the Vertex Fetch unit.

Support of each format and capability is as follows:

Y	supported on all products
Y*	supported only on **[DevCTG+]**
Y+	supported only on **[DevCTG-B+]**
Y~	supported only on **[ILK]**

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	Y~			Y	Y	Y	Y		000	R32G32B32A32_FLOAT	128**
Y				Y		Y	Y		001	R32G32B32A32_SINT	128**
Y				Y		Y	Y		002	R32G32B32A32_UINT	128**
						Y			003	R32G32B32A32_UNORM	128
						Y			004	R32G32B32A32_SNORM	128
						Y			005	R64G64_FLOAT	128
Y	Y~								006	R32G32B32X32_FLOAT	128
						Y			007	R32G32B32A32_SSCALED	128
						Y			008	R32G32B32A32_USCALED	128
Y	Y~					Y	Y		040	R32G32B32_FLOAT	96
Y						Y	Y		041	R32G32B32_SINT	96
Y						Y	Y		042	R32G32B32_UINT	96
						Y			043	R32G32B32_UNORM	96
						Y			044	R32G32B32_SNORM	96
						Y			045	R32G32B32_SSCALED	96
						Y			046	R32G32B32_USCALED	96
Y	Y			Y	Y+	Y		Y^	080	R16G16B16A16_UNORM	64
Y	Y			Y	Y^	Y			081	R16G16B16A16_SNORM	64
Y				Y		Y			082	R16G16B16A16_SINT	64

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y				Y		Y			083	R16G16B16A16_UINT	64
Y	Y			Y	Y	Y			084	R16G16B16A16_FLOAT	64
Y	Y~			Y	Y	Y	Y		085	R32G32_FLOAT	64
Y				Y		Y	Y		086	R32G32_SINT	64
Y				Y		Y	Y		087	R32G32_UINT	64
Y	Y~	Y							088	R32_FLOAT_X8X24_TYPELESS	64
Y									089	X32_TYPELESS_G8X24_UINT	64
Y	Y~								08A	L32A32_FLOAT	64
						Y			08B	R32G32_UNORM	64
						Y			08C	R32G32_SNORM	64
						Y			08D	R64_FLOAT	64
Y	Y								08E	R16G16B16X16_UNORM	64
Y	Y								08F	R16G16B16X16_FLOAT	64
Y	Y~								090	A32X32_FLOAT	64
Y	Y~								091	L32X32_FLOAT	64
Y	Y~								092	I32X32_FLOAT	64
						Y			093	R16G16B16A16_SSCALED	64
						Y			094	R16G16B16A16_USCALED	64
						Y			095	R32G32_SSCALED	64
						Y			096	R32G32_USCALED	64
Y	Y		Y	Y	Y	Y		Y^	0C0	B8G8R8A8_UNORM	32
Y	Y			Y	Y				0C1	B8G8R8A8_UNORM_SRGB	32
Y	Y			Y	Y	Y		Y^	0C2	R10G10B10A2_UNORM	32
Y	Y							Y^	0C3	R10G10B10A2_UNORM_SRGB	32
Y				Y		Y			0C4	R10G10B10A2_UINT	32
Y	Y					Y			0C5	R10G10B10_SNORM_A2_UNORM	32
Y	Y			Y	Y	Y		Y^	0C7	R8G8B8A8_UNORM	32
Y	Y			Y	Y			Y^	0C8	R8G8B8A8_UNORM_SRGB	32
Y	Y			Y	Y^	Y			0C9	R8G8B8A8_SNORM	32
Y				Y		Y			0CA	R8G8B8A8_SINT	32
Y				Y		Y			0CB	R8G8B8A8_UINT	32
Y	Y			Y	Y+	Y			0CC	R16G16_UNORM	32
Y	Y			Y	Y^	Y			0CD	R16G16_SNORM	32
Y				Y		Y			0CE	R16G16_SINT	32
Y				Y		Y			0CF	R16G16_UINT	32

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	Y			Y	Y	Y			0D0	R16G16_FLOAT	32
Y	Y			Y	Y			Y^	0D1	B10G10R10A2_UNORM	32
Y	Y			Y	Y			Y^	0D2	B10G10R10A2_UNORM_SRGB	32
Y	Y			Y	Y	Y			0D3	R11G11B10_FLOAT	32
Y				Y		Y	Y		0D6	R32_SINT	32
Y				Y		Y	Y		0D7	R32_UINT	32
Y	Y~	Y		Y	Y	Y	Y		0D8	R32_FLOAT	32
Y	Y~	Y							0D9	R24_UNORM_X8_TYPELESS	32
Y									0DA	X24_TYPELESS_G8_UINT	32
Y	Y								0DF	L16A16_UNORM	32
Y	Y~	Y							0E0	I24X8_UNORM	32
Y	Y~	Y							0E1	L24X8_UNORM	32
Y	Y~	Y							0E2	A24X8_UNORM	32
Y	Y~	Y							0E3	I32_FLOAT	32
Y	Y~	Y							0E4	L32_FLOAT	32
Y	Y~	Y							0E5	A32_FLOAT	32
Y	Y		Y					Y^	0E9	B8G8R8X8_UNORM	32
Y	Y								0EA	B8G8R8X8_UNORM_SRGB	32
Y	Y								0EB	R8G8B8X8_UNORM	32
Y	Y								0EC	R8G8B8X8_UNORM_SRGB	32
Y	Y								0ED	R9G9B9E5_SHAREDEXP	32
Y	Y								0EE	B10G10R10X2_UNORM	32
Y	Y								0F0	L16A16_FLOAT	32
						Y			0F1	R32_UNORM	32
						Y			0F2	R32_SNORM	32
						Y			0F3	R10G10B10X2_USCALED	32
						Y			0F4	R8G8B8A8_SSCALED	32
						Y			0F5	R8G8B8A8_USCALED	32
						Y			0F6	R16G16_SSCALED	32
						Y			0F7	R16G16_USCALED	32
						Y			0F8	R32_SSCALED	32
						Y			0F9	R32_USCALED	32
Y	Y		Y	Y	Y				100	B5G6R5_UNORM	16
Y	Y			Y	Y				101	B5G6R5_UNORM_SRGB	16
Y	Y		Y	Y	Y				102	B5G5R5A1_UNORM	16

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	Y			Y	Y				103	B5G5R5A1_UNORM_SRGB	16
Y	Y		Y	Y	Y				104	B4G4R4A4_UNORM	16
Y	Y			Y	Y				105	B4G4R4A4_UNORM_SRGB	16
Y	Y			Y	Y	Y			106	R8G8_UNORM	16
Y	Y		Y	Y	Y^	Y			107	R8G8_SNORM	16
Y				Y		Y			108	R8G8_SINT	16
Y				Y		Y			109	R8G8_UINT	16
Y	Y	Y		Y	Y+	Y		Y#	10A	R16_UNORM	16
Y	Y			Y	Y^	Y			10B	R16_SNORM	16
Y				Y		Y			10C	R16_SINT	16
Y				Y		Y			10D	R16_UINT	16
Y	Y			Y	Y	Y			10E	R16_FLOAT	16
Y~	Y~								10F	A8P8_UNORM [palette0]	16
Y~	Y~								110	A8P8_UNORM [palette1]	16
Y	Y	Y							111	I16_UNORM	16
Y	Y	Y							112	L16_UNORM	16
Y	Y	Y							113	A16_UNORM	16
Y	Y		Y						114	L8A8_UNORM	16
Y	Y	Y							115	I16_FLOAT	16
Y	Y	Y							116	L16_FLOAT	16
Y	Y	Y							117	A16_FLOAT	16
Y*	Y*								118	L8A8_UNORM_SRGB	16
Y	Y		Y						119	R5G5_SNORM_B6_UNORM	16
				Y	Y				11A	B5G5R5X1_UNORM	16
				Y	Y				11B	B5G5R5X1_UNORM_SRGB	16
						Y			11C	R8G8_SSCALED	16
						Y			11D	R8G8_USCALED	16
						Y			11E	R16_SSCALED	16
						Y			11F	R16_USCALED	16
Y~	Y~								122	P8A8_UNORM [palette0]	16
Y~	Y~								123	P8A8_UNORM [palette1]	16
Y	Y		Y*	Y	Y	Y			140	R8_UNORM	8
Y	Y			Y	Y^	Y			141	R8_SNORM	8
Y				Y		Y			142	R8_SINT	8
Y				Y		Y			143	R8_UINT	8

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y	Y		Y	Y	Y				144	A8_UNORM	8
Y	Y								145	I8_UNORM	8
Y	Y		Y						146	L8_UNORM	8
Y	Y								147	P4A4_UNORM [palette0]	8
Y	Y								148	A4P4_UNORM [palette0]	8
						Y			149	R8_SSCALED	8
						Y			14A	R8_USCALED	8
Y*	Y*								14B	P8_UNORM [palette0]	8
Y*	Y*								14C	L8_UNORM_SRGB	8
Y+	Y+								14D	P8_UNORM [palette1]	8
Y+	Y+								14E	P4A4_UNORM [palette1]	8
Y+	Y+								14F	A4P4_UNORM [palette1]	8
Y	Y								181	R1_UNORM/R1_UINT	1
Y	Y		Y	Y				Y^	182	YCRCB_NORMAL	0
Y	Y		Y	Y				Y^	183	YCRCB_SWAPUVY	0
Y*	Y*								184	P2_UNORM [palette0]	2
Y+	Y+								185	P2_UNORM [palette1]	2
Y	Y								189	BC4_UNORM	0
Y	Y								18A	BC5_UNORM	0
Y									18E	MONO8	1
Y	Y			Y				Y^	18F	YCRCB_SWAPUV	0
Y	Y			Y				Y^	190	YCRCB_SWAPY	0
Y	Y								192	FXT1	0
						Y			193	R8G8B8_UNORM	24
						Y			194	R8G8B8_SNORM	24
						Y			195	R8G8B8_SSCALED	24
						Y			196	R8G8B8_USCALED	24
						Y			197	R64G64B64A64_FLOAT	256
						Y			198	R64G64B64_FLOAT	192
Y	Y								199	BC4_SNORM	0
Y	Y								19A	BC5_SNORM	0
Y~	Y~					Y^			19B	R16G16B16_FLOAT	48
						Y			19C	R16G16B16_UNORM	48
						Y			19D	R16G16B16_SNORM	48
						Y			19E	R16G16B16_SSCALED	48

Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
						Y			19F	R16G16B16_USCALED	48
Y#	Y#								1A1	BC6H_SF16	0
Y#	Y#								1A2	BC7_UNORM	0
Y#	Y#								1A3	BC7_UNORM_SRGB	0
Y#	Y#								1A4	BC6H_UF16	0
									1FF	RAW	0

** Note: 128 BPE Formats cannot be Tiled Y when used as render targets

NOTE: "RAW" is supported only with buffers and structured buffers accessed via the untyped surface read/write and untyped atomic operation messages, which do not have a column in the table.

4.10.2.1.3 Sampler Output Channel Mapping

The following table indicates the mapping of the channels from the surface to the channels output from the sampling engine. Formats with all four channels (R/G/B/A) in their name map each surface channel to the corresponding output, thus those formats are not shown in this table.

Surface Format Name	R	G	B	A
R32G32B32X32_FLOAT	R	G	B	1.0
R32G32B32_FLOAT	R	G	B	1.0
R32G32B32_SINT	R	G	B	1.0
R32G32B32_UINT	R	G	B	1.0
R32G32_FLOAT	R	G	1.0	1.0
	R	G	0.0	1.0
R32G32_SINT	R	G	0.0	1.0
R32G32_UINT	R	G	0.0	1.0
R32_FLOAT_X8X24_TYPELESS	R	0.0	0.0	1.0
X32_TYPELESS_G8X24_UINT	0.0	G	0.0	1.0
L32A32_FLOAT	L	L	L	A
R16G16B16X16_UNORM	R	G	B	1.0
R16G16B16X16_FLOAT	R	G	B	1.0
A32X32_FLOAT	0.0	0.0	0.0	A
L32X32_FLOAT	L	L	L	1.0
I32X32_FLOAT	I	I	I	I

Surface Format Name	R	G	B	A
R16G16_UNORM	R	G	1.0	1.0
	R	G	0.0	1.0
R16G16_SNORM	R	G	1.0	1.0
	R	G	0.0	1.0
R16G16_SINT	R	G	0.0	1.0
R16G16_UINT	R	G	0.0	1.0
R16G16_FLOAT	R	G	1.0	1.0
	R	G	0.0	1.0
R11G11B10_FLOAT	R	G	B	1.0
R32_SINT	R	0.0	0.0	1.0
R32_UINT	R	0.0	0.0	1.0
R32_FLOAT	R	1.0	1.0	1.0
	R	0.0	0.0	1.0
R24_UNORM_X8_TYPELESS	R	0.0	0.0	1.0
X24_TYPELESS_G8_UINT	0.0	G	0.0	1.0
L16A16_UNORM	L	L	L	A
I24X8_UNORM	I	I	I	I
L24X8_UNORM	L	L	L	1.0
A24X8_UNORM	0.0	0.0	0.0	A
I32_FLOAT	I	I	I	I
L32_FLOAT	L	L	L	1.0
A32_FLOAT	0.0	0.0	0.0	A
B8G8R8X8_UNORM	R	G	B	1.0
B8G8R8X8_UNORM_SRGB	R	G	B	1.0
R8G8B8X8_UNORM	R	G	B	1.0
R8G8B8X8_UNORM_SRGB	R	G	B	1.0
R9G9B9E5_SHAREDEXP	R	G	B	1.0
B10G10R10X2_UNORM	R	G	B	1.0
L16A16_FLOAT	L	L	L	A
B5G6R5_UNORM	R	G	B	1.0
B5G6R5_UNORM_SRGB	R	G	B	1.0
R8G8_UNORM	R	G	1.0	1.0
	R	G	0.0	1.0
R8G8_SNORM	R	G	1.0	1.0
	R	G	0.0	1.0
R8G8_SINT	R	G	0.0	1.0
R8G8_UINT	R	G	0.0	1.0
R16_UNORM	R	0.0	0.0	1.0
R16_SNORM	R	0.0	0.0	1.0
R16_SINT	R	0.0	0.0	1.0
R16_UINT	R	0.0	0.0	1.0
R16_FLOAT	R	1.0	1.0	1.0
	R	0.0	0.0	1.0
I16_UNORM	I	I	I	I
L16_UNORM	L	L	L	1.0
A16_UNORM	0.0	0.0	0.0	A

Surface Format Name	R	G	B	A
L8A8_UNORM	L	L	L	A
I16_FLOAT	I	I	I	I
L16_FLOAT	L	L	L	1.0
A16_FLOAT	0.0	0.0	0.0	A
R5G5_SNORM_B6_UNORM	R	G	B	1.0
R8_UNORM	R	0.0	0.0	1.0
R8_SNORM	R	0.0	0.0	1.0
R8_SINT	R	0.0	0.0	1.0
R8_UINT	R	0.0	0.0	1.0
A8_UNORM	0.0	0.0	0.0	A
I8_UNORM	I	I	I	I
L8_UNORM	L	L	L	1.0
L8_UNORM_SRGB	L	L	L	1.0
R1_UNORM/R1_UINT	R	0.0	0.0	1.0
YCRCB_NORMAL	Cr	Y	Cb	1.0
YCRCB_SWAPUVY	Cr	Y	Cb	1.0
BC4_UNORM	R	0.0	0.0	1.0
BC5_UNORM	R	G	0.0	1.0
YCRCB_SWAPUV	Cr	Y	Cb	1.0
YCRCB_SWAPY	Cr	Y	Cb	1.0
BC4_SNORM	R	0.0	0.0	1.0
BC5_SNORM	R	G	0.0	1.0

4.10.2.2 For deinterlace, sample_8x8, and VME messages

[ILK] only. This state definition is used only by the *deinterlace* and *sample_8x8* sampling engine messages

DWord	Bit	Description
0	31:0	**Surface Base Address** Specifies the byte-aligned base address of the surface. For PLANAR surface formats, this address points to the Y (luma) plane, with the other plane(s) being specified via X/Y offsets. Programming Notes: • Tiled surface base addresses must be 4KB-aligned. Note that only the offsets from **Surface Base Address** are tiled, **Surface Base Address** itself is not transformed using the tiling algorithm. Format = Bits 31:0 of MI_Graphics_Address
1	31:19	**Height** This field specifies the height of the surface in units of pixels. For PLANAR surface formats, this field indicates the height of the Y (luma) plane. **Programming Notes:** • **Height** (field value + 1) must be a multiple of 2 for PLANAR_420 surfaces. Format = U13 Range = [0,8191] representing heights [1,8192]

DWord	Bit	Description
	18:6	**Width** This field specifies the width of the surface in units of pixels. For PLANAR surface formats, this field indicates the width of the Y (luma) plane. **Programming Notes:** • The Width specified by this field multiplied by the pixel size in bytes must be less than or equal to the surface pitch (specified in bytes via the **Surface Pitch** field). • **Width** (field value + 1) must be a multiple of 2 for PLANAR_420, PLANAR_422, and all YCRCB_* surfaces, and must be a multiple of 4 for PLANAR_411 surfaces. Format = U13 Range = [0,8191] representing widths [1,8192]
	5:2	Reserved : MBZ
	1:0	**Cr(V)/Cb(U) Pixel Offset V Direction** Specifies the distance to the U/V values with respect to the even numbered Y channels in the V direction Format = U0.2 **Programming Notes:** • This field is ignored for all formats except PLANAR_420_8
2	31:28	**Surface Format** Specifies the format of the surface. All of the Y and G channels will use table 0 and all of the Cr/Cb/R/B channels will use table 1. 0: YCRCB_NORMAL 1: YCRCB_SWAPUVY 2: YCRCB_SWAPUV 3: YCRCB_SWAPY 4: PLANAR_420_8 5: PLANAR_411_8 (*deinterlace* only) 6: PLANAR_422_8 (*deinterlace* only) 7: STMM_DN_STATISTICS (*deinterlace* only) 8: R10G10B10A2_UNORM (*sample_8x8* only) 9: R8G8B8A8_UNORM (*sample_8x8* only) 10: R8B8_UNORM (CrCb) (*sample_8x8* only) 11: R8_UNORM (Cr/Cb) (*sample_8x8* only) 12: Y8_UNORM 13-15 Reserved
	27	**Interleave Chroma** This field indicates that the chroma fields are interleaved in a single plane rather than stored as two separate planes. This field is only used for PLANAR surface formats. Format = Enable
	26	Reserved : MBZ

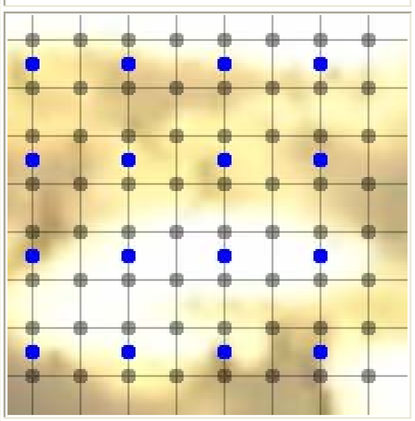
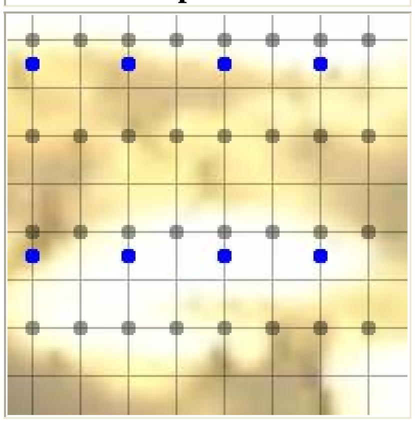
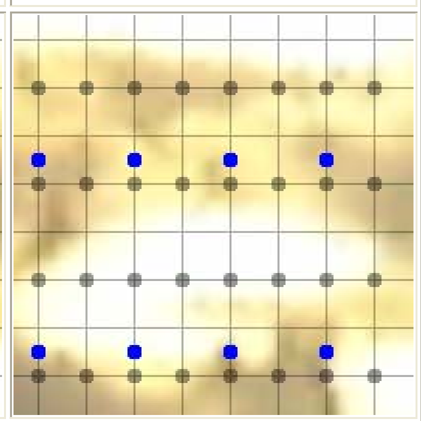
DWord	Bit	Description
	21:20	Reserved : MBZ
	19:3	**Surface Pitch** This field specifies the surface pitch in (#Bytes - 1). **Programming Notes:** • For tiled surfaces, the pitch must be a multiple of the tile width • For tiled surfaces, with **Half Pitch for Chroma** the pitch must be a multiple of the tile width x 2 • For non tiled surfaces with **Half Pitch for Chroma** pitch must be even • If **Half Pitch for Chroma** is set, this field must be a multiple of two tile widths for tiled surfaces, or a multiple of 2 bytes for linear surfaces. Format = U17 pitch in (Bytes - 1). For surfaces of type SURFTYPE_BUFFER: Range = [0,2047] -> [1B, 2048B] For other linear surfaces: Range = [0, 131071] -> [1B, 128KB] For X-tiled surface: Range = [511, 131071] –> [512B, 128KB] = [1tile, 256 tiles] For Y-tiled surfaces: Range = [127, 131071]->[128B,128KB] = [1 tile, 1024 tiles]
	2	**Half Pitch for Chroma** This field indicates that the chroma plane(s) will use a pitch equal to half the value specified in the **Surface Pitch** field. This field is only used for PLANAR surface formats. Format = Enable
	1	**Tiled Surface** This field specifies whether the surface is tiled. **Programming Notes:** • Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory. • The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit. • The tiled surfaces of current picture and reference picture should be declared as the identical type in VDI mode with the identical Height, Width and Format. Format = Boolean 1: TRUE: Tiled 0: FALSE: Linear

DWord	Bit	Description
	0	**Tile Walk** This field specifies the type of memory tiling (XMajor or YMajor) employed to tile this surface. See *Memory Interface Functions* for details on memory tiling and restrictions. This field is ignored when the surface is linear. **Programming Notes:** • The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit. Format = 3D_TileWalk 0: TILEWALK_XMAJOR 1: TILEWALK_YMAJOR
3	31:29	Reserved : MBZ
	28:16	**X Offset for U(Cb)** This field specifies the horizontal offset in pixels from the **Surface Base Address** to the start (origin) of the U(Cb) plane or the interleaved UV plane if **Interleave Chroma** is enabled. This field is only used for PLANAR surface formats. **Programming Notes:** • For PLANAR_420 and PLANAR_422 surface formats, this field must indicate an even number of pixels. Format = U13 Pixel Offset
	15:13	Reserved : MBZ
	12:0	**Y Offset for U(Cb)** This field specifies the veritical offset in rows from the **Surface Base Address** to the start (origin) of the U(Cb) plane or the interleaved UV plane if **Interleave Chroma** is enabled. This field is only used for PLANAR surface formats. **Programming Notes:** • This field must indicate an even number (bit 0 = 0). • If **Half Pitch for Chroma** is set this will be equal to 2*(height of Y surface) if U is **above** V or they are interleaved.. If not then it will be 2*(height of Y surface) + (Height of V surface) Format = U13 Row Offset
4	31:29	Reserved : MBZ
	28:16	**X Offset for V(Cr)** This field specifies the horizontal offset in pixels from the **Surface Base Address** to the start (origin) of the V(Cr) plane. This field is only used for PLANAR surface formats with **Interleave Chroma** disabled. **Programming Notes:** • For PLANAR_420 and PLANAR_422 surface formats, this field must indicate an even number of pixels. Format = U13 Pixel Offset
	15:13	Reserved : MBZ

DWord	Bit	Description
	12:0	**Y Offset for V(Cr)** This field specifies the veritical offset in rows from the **Surface Base Address** to the start (origin) of the V(Cr) plane. This field is only used for PLANAR surface formats with **Interleave Chroma** disabled. **Programming Notes:** This field must indicate an even number (bit 0 = 0). • If **Half Pitch for Chroma** is set this will be equal to 2*(height of Y surface) if V is **above** U or they are interleaved. If not then it will be 2*(height of Y surface) + (Height of U surface) Format = U13 Row Offset

Cr(V)/Cb(U) Pixel Offset V Direction

The position of Y is brown and the position of Cr(V)/Cb(U) is blue.

full frame	top field	bottom field
		
V Offset 0.5	V Offset 0.25	V Offset 0.75

4.10.3 SAMPLER_STATE

SAMPLER_STATE has three different formats, depending on the message type used. For **[ILK]**, all messages use the format described under "For most messages". For **[ILK]**, the sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

4.10.3.1 For most messages

4.10.3.1.1 [Pre-DevIVB]

SAMPLER_STATE		
Project: [Pre-DevIVB]		
This is the normal sampler state used by all messages that use SAMPLER_STATE except *sample_8x8* and *deinterlace*. The sampler state is stored as an array of up to 16 elements, each of which contains the dwords described here. The start of each element is spaced 4 dwords apart. The first element of the sampler state array is aligned to a 32-byte boundary.		
DWord	**Bit**	**Description**
0	31	**Sampler Disable** Project: All Format: Disable FormatDesc This field allows the sampler to be disabled. If disabled, all output channels will return 0.
	30	**Reserved** Project: All Format: MBZ
	28	**LOD PreClamp Enable** Project: All Format: U1 enumerated type FormatDesc When enabled, the computed LOD is clamped to [max,min] mip level *before* the mag-vs-min determination is performed. This is how the OpenGL API currently performs min/mag determination, and therefore it is expected that an OpenGL driver would need to set this bit. D3D drivers would not set this bit. **Value** / **Name** / **Description** / **Project** 0h / D3D / D3D Mode (LOD PreClamp disabled) / All 1h / OGL / OGL Mode (LOD PreClamp enabled) / All
	27	**Reserved** Project: All Format: MBZ
	26:22	**Base Mip Level** Project: All Format: U4.1 FormatDesc Range [0.0,13.0] Specifies which mip level is considered the "base" level when determining mag-vs-min filter and selecting the "base" mip level.

SAMPLER_STATE

	21:20	**Mip Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines if and how mip map levels are chosen and/or combined when texture filtering.

Value	Name	Description	Project
0h	MIPFILTER_NONE	Disable mip mapping – force use of the mipmap level corresponding to **Min LOD**.	All
1h	MIPFILTER_NEAREST	Nearest, Select the nearest mip map	All
2h	Reserved		All
3h	MIPFILTER_LINEAR	Linearly interpolate between nearest mip maps (combined with linear min/mag filters this is analogous to "Trilinear" filtering).	All

Programming Notes

MIPFILTER_LINEAR is not supported for surface formats that do not support "Sampling Engine Filtering" as indicated in the Surface Formats table unless using the sample_c message type.

SAMPLER_STATE

	Bits	Description
	19:17	**Mag Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines how texels are sampled/filtered when a texture is being "magnified" (enlarged). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension. **Value** / **Name** / **Description** / **Project** 0h / MAPFILTER_NEAREST / Sample the nearest texel / All 1h / MAPFILTER_LINEAR / Bilinearly filter the 4 nearest texels / All 2h / MAPFILTER_ANISOTROPIC / Perform an "anisotropic" filter on the chosen mip level / All 3h-5h / Reserved / / All 6h / MAPFILTER_MONO / Perform a monochrome convolution filter / All 7h / Reserved / / All **Programming Notes** Only MAPFILTER_NEAREST and MAPFILTER_LINEAR are supported for surfaces of type SURFTYPE_3D. Only MAPFILTER_NEAREST is supported for surface formats that do not support "Sampling Engine Filtering" as indicated in the Surface Formats table unless using the sample_c message type. MAPFILTER_MONO: Only CLAMP_BORDER texture addressing mode is supported. . Both **Mag Mode Filter** and **Min Mode Filter** must be programmed to MAPFILTER_MONO. **Mip Mode Filter** must be MIPFILTER_NONE. Only valid on surfaces with **Surface Format** MONO8 and with **Surface Type** SURFTYPE_2D. MAPFILTER_ANISOTROPIC may cause artifacts at cube edges if enabled for cube maps with the TEXCOORDMODE_CUBE addressing mode. MAPFILTER_ANISOTROPIC will be overridden to MAPFILTER_LINEAR when using a sample_l or sample_l_c message type or when **Force LOD to Zero** is set in the message header. **[DevBW, DevCL] Errata: Force LOD to Zero** will not cause MAPFILTER_ANISOTROPIC to get forced to MAPFILTER_LINEAR and instead it will have to be worked around using sample_l or sample_l_c.
	16:14	**Min Mode Filter** Project: All Format: U2 enumerated type FormatDesc This field determines how texels are sampled/filtered when a texture is being "minified" (shrunk). For volume maps, this filter mode selection also applies to the 3rd (inter-layer) dimension. See **Mag Mode Filter**

SAMPLER_STATE

	13:3	**Texture LOD Bias** Project: All Format: S4.6 2's complement FormatDesc Range [-16.0, 16.0) This field specifies the signed bias value added to the calculated texture map LOD prior to min-vs-mag determination and mip-level clamping. Assuming mipmapping is enabled, a positive LOD bias will result in a somewhat blurrier image (using less-detailed mip levels) and possibly higher performance, while a negative bias will result in a somewhat crisper image (using more-detailed mip levels) and may lower performance. **Programming Notes** There is <u>no</u> requirement or need to offset the LOD Bias in order to produce a correct LOD for texture filtering (as was required for correct bilinear and anisotropic filtering in some legacy devices).
	2:0	**Shadow Function** Project: All Format: U3 enumerated type FormatDesc This field is used for shadow mapping support via the sample_c message type, and specifies the specific comparison operation to be used. The comparison is between the texture sample red channel (except for alpha-only formats which use the alpha channel), and the "ref" value provided in the input message.

Value	Name	Description	Project
0h	PREFILTEROP_ALWAYS		All
1h	PREFILTEROP_NEVER		All
2h	PREFILTEROP_LESS		All
3h	PREFILTEROP_EQUAL		All
4h	PREFILTEROP_LEQUAL		All
5h	PREFILTEROP_GREATER		All
6h	PREFILTEROP_NOTEQUAL		All
7h	PREFILTEROP_GEQUAL		All

SAMPLER_STATE		
1	31:22	**Min LOD** Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0], where the upper limit is also bounded by the **Max LOD**. This field specifies the minimum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this maximum (resolution) mip clamping is applied. The integer bits of this field are used to control the "maximum" (highest resolution) mipmap level that may be accessed (where LOD 0 is the highest resolution map). The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. **Programming Notes** If **Min LOD** is greater than **Max LOD**, **Min LOD** takes precedence, i.e. the resulting LOD will always be **Min LOD**. This field must be zero if the **Min** or **Mag Mode Filter** is set to MAPFILTER_MONO
	21:12	**Max LOD** Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0] This field specifies the maximum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and before this minimum (resolution) mip clamping is applied. The integer bits of this field are used to control the "minimum" (lowest resolution) mipmap level that may be accessed. The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here. **Programming Notes** If **Min LOD** is greater than **Max LOD**, **Min LOD** takes precedence, i.e. the resulting LOD will always be **Min LOD**. **Errata** / **Description** / **Project** # / If the **Mip Mode Filter** is set to MIPFILTER_NEAREST and the fractional portion of **Max LOD** is < 0.5 but > 0.0, the LOD chosen is one too large. Zeroing the fractional portion of **Max LOD** in these cases gives the correct behavior as a software workaround. / [ILK]
	11:10	**Reserved** Project: All Format: MBZ

SAMPLER_STATE

	9	**Cube Surface Control Mode** Project: All Format: U1 enumerated type FormatDesc When sampling from a SURFTYPE_CUBE surface, this field controls whether the **TC* Address Control Mode** fields are interpreted as programmed or overridden to TEXCOORDMODE_CUBE.

Value	Name	Description	Project
0h	CUBECTRLMODE_PROGRAMMED		All
1h	CUBECTRLMODE_OVERRIDE		All

Errata	Description	Project
#	this field must be set to CUBECTRLMODE_PROGRAMMED	[DevBW-A,B], [DevCL-A]

SAMPLER_STATE

	Bits	Description
	8:6	**TCX Address Control Mode** Project: All Format: U3 enumerated type FormatDesc Controls how the 1st (TCX, aka U) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates "outside" the texture are handled (wrap/clamp/mirror). The setting of this field is subject to being overridden by the **Cube Surface Control Mode** field when sampling from a SURFTYPE_CUBE surface.

Value	Name	Description	Project
0h	TEXCOORDMODE_WRAP	Map is repeated in the U direction	All
1h	TEXCOORDMODE_MIRROR	Map is mirrored in the U direction	All
2h	TEXCOORDMODE_CLAMP	Map is clamped to the edges of the accessed map	All
3h	TEXCOORDMODE_CUBE	For cube-mapping, filtering in edges access adjacent map faces	All
4h	TEXCOORDMODE_CLAMP_BORDER	Map is infinitely extended with the border color	All
5h	TEXCOORDMODE_MIRROR_ONCE	Map is mirrored once about origin, then clamped	All
6h-7h	Reserved		All

Programming Notes

When using cube map texture coordinates, only TEXCOORDMODE_CLAMP and TEXCOORDMODE_CUBE settings are valid, and each TC component must have the same Address Control mode.

When TEXCOORDMODE_CLAMP is used when accessing a cube map, the map's **Cube Face Enable** field must be programmed to 111111b (all faces enabled).

MAPFILTER_MONO: Texture addressing modes must all be set to TEXCOORDMODE_CLAMP_BORDER. The **Border Color** is ignored in this mode, a constant value of 0 is used for border color. Software must pad the border texels within the map itself with 0.

TEXCOORDMODE_MIRROR and TEXCOORDMODE_MIRROR_ONCE cannot be used with the sample_unorm* message types.

<table>
<tr><th colspan="3">SAMPLER_STATE</th></tr>
<tr><td></td><td>5:3</td><td>TCY Address Control Mode
Project: All
Format: U3 enumerated type FormatDesc
Controls how the 2nd (TCY, aka V) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates “outside” the texture are handled (wrap/clamp/mirror).
See Address TCX Control Mode above for details
<table>
<tr><th>Errata</th><th>Description</th><th>Project</th></tr>
<tr><td>#</td><td>if this field is set to TEXCOORDMODE_CLAMP_BORDER and a 1D surface is sampled, incorrect blending with the border color in the vertical direction may occur.</td><td>[Pre-ILK]</td></tr>
</table></td></tr>
<tr><td></td><td>2:0</td><td>TCZ Address Control Mode
Project: All
Format: U3 enumerated type FormatDesc
Controls how the 3rd (TCZ) component of input texture coordinates are mapped to texture map addresses – specifically, how coordinates “outside” the texture are handled (wrap/clamp/mirror).
See Address TCX Control Mode above for details</td></tr>
<tr><td>2</td><td>31:5</td><td>Border Color Pointer
Project: All
Format: [Pre-DevGT]: GeneralStateOffset[31:5] FormatDes
[]
This field specifies the pointer to SAMPLER_BORDER_COLOR_STATE, which contains the “border” color to be used when accessing texels not contained within the texture map. This pointer is relative to the General State Base Address for [Pre-DevGT]</td></tr>
<tr><td></td><td>4:0</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td>3</td><td>31:29</td><td>Monochrome Filter Height
Project: [Pre-ILK]
Format: U3 FormatDesc
Range [1,7]
This field specifies the height of the monochrome filter. It is ignored if the monochrome filter is not enabled.
[ILK]: Reserved : MBZ (this field has been moved to 3DSTATE_MONOFILTER_SIZE)</td></tr>
<tr><td></td><td>28:26</td><td>Monochrome Filter Width
Project: All
Format: U3 FormatDesc
Range [1,7]
This field specifies the width of the monochrome filter. It is ignored if the monochrome filter is not enabled.
[ILK]: Reserved : MBZ (this field has been moved to 3DSTATE_MONOFILTER_SIZE)</td></tr>
</table>

<table>
<tr><th colspan="3">SAMPLER_STATE</th></tr>
<tr><td></td><td>25</td><td>ChromaKey Enable
Project: All
Format: Enable FormatDesc
This field enables the chroma key function.

Programming Notes
Supported only on a specific subset of surface formats. See section 4.10.2.1 “Surface Formats” for supported formats.
This field must be disabled if min or mag filter is MAPFILTER_MONO or MAPFILTER_ANISOTROPIC.
This field must be disabled if used with a surface of type SURFTYPE_3D.</td></tr>
<tr><td></td><td>24:23</td><td>ChromaKey Index
Project: All
Format: U2 FormatDesc
Range [0,3]
This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a “don’t care” unless ChromaKey Enable is ENABLED.</td></tr>
<tr><td></td><td>22</td><td>ChromaKey Mode
Project: All
Format: U1 enumerated type FormatDesc
This field specifies the behavior of the device in the event of a ChromaKey match. This field is ignored if ChromaKey is disabled.
KEYFILTER_KILL_ON_ANY_MATCH:
In this mode, if any contributing texel matches the chroma key, the corresponding pixel mask bit for that pixel is cleared. The result of this operation is observable only if the Killed Pixel Mask Return flag is set on the input message.
KEYFILTER_REPLACE_BLACK:
In this mode, each texel that matches the chroma key is replaced with (0,0,0,0) (black with alpha=0) prior to filtering. For YCrCb surface formats, the black value is A=0, R(Cr)=0x80, G(Y)=0x10, B(Cb)=0x80. This will tend to darken/fade edges of keyed regions. Note that the pixel pipeline must be programmed to use the resulting filtered texel value to gain the intended effect, e.g., handle the case of a totally keyed-out region (filtered texel alpha==0) through use of alpha test, etc.
<table>
<tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr>
<tr><td>0h</td><td>KEYFILTER_KILL_ON_ANY_MATCH</td><td></td><td>All</td></tr>
<tr><td>1h</td><td>KEYFILTER_REPLACE_BLACK</td><td></td><td>All</td></tr>
</table></td></tr>
</table>

SAMPLER_STATE		
	21:19	**Maximum Anisotropy** Project: All Format: U3 enumerated type FormatDesc This field clamps the maximum value of the anisotropy ratio used by the MAPFILTER_ANISOTROPIC filter (Min or Mag Mode Filter). **Value** / **Name** / **Description** / **Project** 0h / ANISORATIO_2 / At most a 2:1 aspect ratio filter is used / All 1h / ANISORATIO_4 / At most a 4:1 aspect ratio filter is used / All 2h / ANISORATIO_6 / At most a 6:1 aspect ratio filter is used / All 3h / ANISORATIO_8 / At most a 8:1 aspect ratio filter is used / All 4h / ANISORATIO_10 / At most a 10:1 aspect ratio filter is used / All 5h / ANISORATIO_12 / At most a 12:1 aspect ratio filter is used / All 6h / ANISORATIO_14 / At most a 14:1 aspect ratio filter is used / All 7h / ANISORATIO_16 / At most a 16:1 aspect ratio filter is used / All
	18:13	**Address Rounding Enable** Project: All Format: 6-bit mask of enables FormatDesc Controls whether the U/V/R texture address is rounded or truncated before being used to select texels to sample. Each bit provides independent control of rounding on one texture address dimension (U/V/R) in either mag or min filter mode. **Value** / **Name** / **Description** / **Project** 100000b / / U address mag filter / All 010000b / / U address min filter / All 001000b / / V address mag filter / All 000100b / / V address min filter / All 000010b / / R address mag filter / All 000001b / / R address min filter / All
	12:1	**Reserved** Project: All Format: MBZ

1.11.3.2 For sample_8x8 message

[DevILK] and [DevSNB] This state definition is used only by the *sample_8x8* message. This state is stored as an array of up to 4 elements, each of which contains the dwords described here. The start of each element is spaced 16 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-3 that selects which element is being used is multiplied by 4 to determine the **Sampler Index** in the message descriptor.

DWord	Bit	Description
0	31	**AVS Filter Type.** Defines the type of adaptive video scaler filter that will be enabled. 0: Adaptive 8-tap polyphase filter 1: Nearest filter
	30	Reserved : MBZ
	29	**IEF Bypass.** Causes IEF function to be bypassed, VSA will output neutral values.
	28	**IEF Filter Type** 0: Combo mode 1: Detail Filter
	27	**IEF Filter Size** 0: 3x3 1: 5x5 **Programming Notes:** • If **IEF Filter Type** is Advanced Filter, this field must be set to 5x5
	26:19	Reserved : MBZ
	18	**ChromaKey Enable.** This field enables chroma keying when accessing this particular texture map. **Programming Notes:** • For sample_8x8 instructions KEYFILTER_REPLACE_BLACK is assumed if chromakey is enabled. • For 10 bit formats only the 8 MSBs will be compared. Format = Enable
	17:16	**ChromaKey Index.** This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a "don't care" unless **ChromaKey Enable** is ENABLED. Format = U2 Range = [0,3]
	15:0	Reserved : MBZ
1	31:5	**Sampler 8x8 State Pointer.** This field specifies the pointer to the SAMPLER_8x8_STATE structure. This pointer is relative to the **General State Base Address** for **[ILK].** **Programming Notes:** • This field must be set to the same value in all sample_8x8 type SAMPLER_STATE instances applied to a given primitive. • **[ILK]:** MI_FLUSH with **State/Instruction Cache Invalidate** set is required between primitives that use different values of this field. (PIPE_CONTROL *cannot* be used as an alternative to MI_FLUSH). **[ILK]:** GeneralStateOffset[31:5]
	4:0	Reserved : MBZ
2	31:16	Reserved : MBZ
	15:8	**Global Noise Estimation.** Global noise estimation of previous frame from DI. Format = U8 (default = 22)

DWord	Bit	Description
	7:4	**Strong Edge Threshold.** If EM > **Strong Edge Threshold**, the basic VSA detects a strong edge. Format = U4 (default = 8)
	3:0	**Weak Edge Threshold.** If **Strong Edge Threshold** > EM > **Weak Edge Threshold**, the basic VSA detects a weak edge. Format = U4 (default = 1)
3	31	Reserved : MBZ
	30:28	**Strong Edge Weight.** Sharpening strength when a strong edge is found in basic VSA. Format = U3 (default = 7)
	27	Reserved : MBZ
	26:24	**Regular Weight.** Sharpening strength when a weak edge is found in basic VSA. Format = U3 (default = 2)
	23	Reserved : MBZ
	22:20	**Non Edge Weight.** Sharpening strength when no edge is found in basic VSA. Format = U3 (default = 1)
	19:14	**Gain Factor.** User control sharpening strength. Format = U6 (default = 40)
	13:11	Reserved : MBZ
	10:6	**R3c Coefficient.** IEF smoothing coefficient, see IEF map. Format = U0.5 (default = (59+2) >> 2)
	5	Reserved : MBZ
	4:0	**R3x Coefficient.** IEF smoothing coefficient, see IEF map. Format = U0.5 (default = ((25+2) >> 2)
4	31	Reserved : MBZ
	30:26	**R5c Coefficient.** IEF smoothing coefficient, see IEF map. Format = U0.5 (default = 3)
	25	Reserved : MBZ
	24:20	**R5cx Coefficient.** IEF smoothing coefficient, see IEF map. Format = U0.5 (default = 8)
	19	Reserved : MBZ
	18:14	**R5x Coefficient.** IEF smoothing coefficient, see IEF map. Format = U0.5 (default = 9)
	13:12	Reserved : MBZ
	11:8	**Steepness Threshold.** VSA uses steepness only when greater than this threshold. Format = U4 (default = 0)
	7	**Steepness Boost.** Used to increase effect of steepness. Format = Enable (default = 0)

DWord	Bit	Description
	6:3	**MR Threshold.** VSA uses MR only when greater than this threshold. Format = U4 (default = 5)
	2	**MR Boost.** Used to increase effect of MR. Format = Enable (default = 0)
	1:0	Reserved : MBZ
5	31:24	**PWL1 Point 4.** Point 4 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 26)
	23:16	**PWL1 Point 3.** Point 3 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 16)
	15:8	**PWL1 Point 2.** Point 2 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 12)
	7:0	**PWL1 Point 1.** Point 1 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 4)
6	31:24	**PWL1 R3 Bias 1.** Bias 1 for PWL of smoothing strength. Format = U8 (default = 98)
	23:16	**PWL1 R3 Bias 0.** Bias 0 for PWL of smoothing strength. Format = U8 (default = 127)
	15:8	**PWL1 Point 6.** Point 6 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 160)
	7:0	**PWL1 Point 5.** Point 5 for PWL of *both* sharpening and smoothing strength. Format = U8 (default = 40)
7	31:24	**PWL1 R3 Bias 5.** Bias 5 for PWL of smoothing strength. Format = U8 (default = 0)
	23:16	**PWL1 R3 Bias 4.** Bias 4 for PWL of smoothing strength. Format = U8 (default = 44)
	15:8	**PWL1 R3 Bias 3.** Bias 3 for PWL of smoothing strength. Format = U8 (default = 64)
	7:0	**PWL1 R3 Bias 2.** Bias 2 for PWL of smoothing strength. Format = U8 (default = 88)
8	31:24	**PWL1 R5 Bias 2.** Bias 2 for PWL of sharpening strength. Format = U8 (default = 32)
	23:16	**PWL1 R5 Bias 1.** Bias 1 for PWL of sharpening strength. Format = U8 (default = 32)
	15:8	**PWL1 R5 Bias 0.** Bias 0 for PWL of sharpening strength. Format = U8 (default = 3)
	7:0	**PWL1 R3 Bias 6.** Bi as 6 for PWL of smoothing strength. Format = U8 (default = 0)

DWord	Bit	Description
9	31:24	**PWL1 R5 Bias 6.** Bias 6 for PWL of sharpening strength. Format = U8 (default = 88)
	23:16	**PWL1 R5 Bias 5.** Bias 5 for PWL of sharpening strength. Format = U8 (default = 108)
	15:8	**PWL1 R5 Bias 4.** Bias 4 for PWL of sharpening strength. Format = U8 (default = 100)
	7:0	**PWL1 R5 Bias 3.** Bias 3 for PWL of sharpening strength. Format = U8 (default = 58)
10	31:24	**PWL1 R3 Slope 3.** Slope 3 for PWL of smoothing strength. Format = S3.4 2's complement (default = -32)
	23:16	**PWL1 R3 Slope 2.** Slope 2 for PWL of smoothing strength. Format = S3.4 2's complement (default = -96)
	15:8	**PWL1 R3 Slope 1.** Slope 1 for PWL of smoothing strength. Format = S3.4 2's complement (default = -20)
	7:0	**PWL1 R3 Slope 0.** Slope 0 for PWL of smoothing strength. Format = S3.4 2's complement (default = -116)
11	31:24	**PWL1 R5 Slope 0.** Slope 0 for PWL of sharpening strength. Format = S3.4 2's complement (default = 116)
	23:16	**PWL1 R3 Slope 6.** Slope 6 for PWL of smoothing strength. Format = S3.4 2's complement (default = 0)
	15:8	**PWL1 R3 Slope 5.** Slope 5 for PWL of smoothing strength. Format = S3.4 2's complement (default = 0)
	7:0	**PWL1 R3 Slope 4.** Slope 4 for PWL of smoothing strength. Format = S3.4 2's complement (default = -50)
12	31:24	**PWL1 R5 Slope 4.** Slope 4 for PWL of sharpening strength. Format = S3.4 2's complement (default = 9)
	23:16	**PWL1 R5 Slope 3.** Slope 3 for PWL of sharpening strength. Format = S3.4 2's complement (default = 67)
	15:8	**PWL1 R5 Slope 2.** Slope 2 for PWL of sharpening strength. Format = S3.4 2's complement (default = 104)
	7:0	**PWL1 R5 Slope 1.** Slope 1 for PWL of sharpening strength. Format = S3.4 2's complement (default = 0)
13	31:28	**Maximum Limiter.** Strength of overshoot limiter. Format = U0.4 (default = 11)
	27:24	**Minimum Limiter.** Strength of undershoot limiter. Format = U0.4 (default = 10)
	23:20	Reserved : MBZ

DWord	Bit	Description
	19:16	**Limiter Boost.** Used to increase limiter strength Format = U0.4 (default = 0)
	15:8	**PWL1 R5 Slope 6.** Slope 6 for PWL of sharpening strength. Format = S3.4 2's complement (default = -15)
	7:0	**PWL1 R5 Slope 5.** Slope 5 for PWL of sharpening strength. Format = S3.4 2's complement (default = -3)
14	31:18	Reserved : MBZ
	17:8	**Clip Limiter.** If extreme point is on the boundary of the neighborhood, adjust limiter's strength. Format = U10 (default = 130)
	7:0	Reserved : MBZ

4.10.3.2 For deinterlace message

[DevILK+] only. This state definition is used only by the *deinterlace* message. This state is stored as an array of up to 8 elements, each of which contains the dwords described here. The start of each element is spaced 8 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-7 that selects which element is being used is multiplied by 2 to determine the **Sampler Index** in the message descriptor.

DWord	Bit	Description
0	31:24	**Denoise STAD Threshold.** Threshold for denoise sum of temporal absolute differences. Format = U8
	23:16	**Denoise Maximum History.** Maximum allowed value for denoise history. Format = U8 Range = [128,240]
	15:8	**Denoise History Delta.** Amount that denoise_history is increased. Format = U8 Range = [0,15]
	7:0	**Denoise ASD Threshold.** Threshold for denoise absolute sum of differences. Format = U8 Range = [0,63]
1	31:30	Reserved : MBZ
	29:24	**Temporal Difference Threshold.** Format = U6 **Programming Notes:** ○ **Temporal Difference Threshold** – **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16.
	23:22	Reserved : MBZ

DWord	Bit	Description
	21:16	**Low Temporal Difference Threshold.** Format = U6 **Programming Notes:** ○ **Temporal Difference Threshold** – **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16.
	15:13	**STMM C2:** Bias for divisor in STMM equation. Format = U3 Range = [0,7] representing values [1,8]
	12:8	**Denoise Moving Pixel Threshold.** Threshold for number of moving pixels to declare a block to be moving. Format = U5 Range = [0,16]
	7:0	**Denoise Threshold for Sum of Complexity Measure.** Format = U8
2	31:24	**Good Neighbor Threshold.** Maximum difference from current pixel for neighboring pixels to be considered a good neighbor. Format = U8 Range = [0,63]
	23:16	**Denoise Edge Threshold.** Threshold for detecting an edge in denoise. Format = U8 Range = [0,15]
	15:8	**Block Noise Estimate Edge Threshold.** Threshold for detecting an edge in block noise estimate. Format = U8 Range = [0,15]
	7:0	**Block Noise Estimate Noise Threshold.** Threshold for noise maximum/minimum. Format = U8 Range = [0,31]
3	31	**STMM Blending Constant Select.** Format = U1 0: Use the blending constant for small values of STMM for stmm_md_th 1: Use the blending constant for large values of STMM for stmm_md_th
	30:24	**Blending constant across time for large values of STMM.** Format = U7
	23:16	**Blending constant across time for small values of STMM.** Format = U8
	15:14	Reserved : MBZ
	13:8	**Multiplier for VECM.** Determines the strength of the vertical edge complexity measure. Format = U6
	7:0	**Maximum STMM.** Largest allowed STMM in blending equations. Format = U8
4	31:24	**Minimum STMM.** Smallest allowed STMM in blending equations. Format = U8

DWord	Bit	Description
	23:22	**STMM Shift Down.** Amount to shift STMM down (quantize to fewer bits). Format = U2 0: Shift by 4 1: Shift by 5 2: Shift by 6 3: Reserved
	21:20	**STMM Shift Up.** Amount to shift STMM up (set range). Format = U2 0: Shift by 6 1: Shift by 7 2: Shift by 8 3: Reserved
	19:16	**STMM Output Shift.** Amount to shift output of STMM blend equation**.** **Programming Notes:** • The value of this field must satisfy the following equation: stmm_max – stmm_min = 2 ^ stmm_output_shift Format = U4 Range = [0,16]
	15:8	**SDI Threshold.** Threshold for angle detection in SDI algorithm. Format = U8
	7:0	**SDI Delta.** Delta value for angle detection in SDI algorithm. Format = U8
5	31:24	**SDI Fallback Mode 1 T1 Constant.** Format = U8
	23:16	**SDI Fallback Mode 1 T2 Constant.** Format = U8
	15:8	**SDI Fallback Mode 2 Constant (Angle2x1).** Format = U8
	7:0	**FMD Temporal Difference Threshold.** Format = U8
6	31:24	**FMD #1 Vertical Difference Threshold.** Format = U8
	23:16	**FMD #2 Vertical Difference Threshold.** Format = U8
	15:14	Reserved : MBZ
	13:8	**FMD Tear Threshold.** Format = U6
	7	Reserved : MBZ
	6	**Progressive DN.** Indicates that the denoise algorithm should assume progressive input when filtering neighboring pixels. **DI Enable** must be disabled when this field is enabled. Format = Enable 0: DN assumes interlaced video and filters alternate lines together 1: DN assumes progressive video and filters neighboring lines together

DWord	Bit	Description
	5	**DN/DI First Frame.** Indicates that this is the first frame of the stream, so previous clean is not available Format = Enable 0: Not first field; previous clean surface state is valid 1: First field; previous clean surface state is invalid
	4	**DN/DI Stream ID.** Distinguishes between the two simultaneous streams that are supported. Used to update the GNE and FMD counters for that stream. Format = U1
	3	**DN/DI Top First.** Indicates the top field is first in sequence, otherwise bottom is first Format = Enable 0 = Bottom field occurs first in sequence 1 = Top field occurs first in sequence
	2	**DI Partial.** If **DI Enable** and **DI Partial** are both enabled, the deinterlacer will output the partial VDI writeback message. Format = Enable 0: Output normal VDI writeback message (only if **DI Enable** is enabled also) 1: Output partial VDI writeback message (only if **DI Enable** is enabled also)
	1	**DI Enable.** Deinterlacer is bypassed if this is disabled: the output is the same as the input (same as a 2:2 cadence). FMD and STMM are not calculated and the values in the response message are 0. Format = Enable 0: Do not calculate DI 1: Calculate DI **Programming Notes:** ○ **DI Enable** and **DN Enable** cannot both be disabled.
	0	**DN Enable.** Denoise is bypassed if this is low – BNE is still calculated and output, but the denoised fields are not. VDI does not read in the denoised previous frame but uses the pointer for the original previous frame. Format = Enable 0: Do not denoise frame 1: Denoise frame **Programming Notes:** ○ **DI Enable** and **DN Enable** cannot both be disabled.
7	31:23	**Column Width Minus1** This field specifies the (column width-1) / stride in units of blocks (Each blocks has width 16 pixels). A column width * 16 that equals the width of the frame means the walker will walk to the end of the frame. Format = U9 Range = [0, 511] representing column widths [1 to 512] (interpret value as binary value + 1)
	31:19	Reserved : MBZ
	18	**VDI Walker Enable** Format = U1 0: Walker Disabled. Use XY generated by Driver. 1: Walker Enabled. Use XY generated by VDIunit.

DWord	Bit	Description
	17:16	**FMD for 2nd field of previous frame.** Format = U2 0: Deinterlace (not progressive output) 1: Put together with previous field in sequence (1st field of previous frame). 2: Put together with next field in sequence (1st field of current frame).
	15:10	Reserved : MBZ
	9:8	**FMD for 1st field of current frame.** Format = U2 0: Deinterlace (not progressive output). 1: Put together with previous field in sequence (2nd field of previous frame). 2: Put together with next field in sequence (2nd field of current frame).
	7:0	Reserved : MBZ

4.10.3.2.1 For Deinterlace Message (Gen7+)

This state definition is used only by the *deinterlace* message. This state is stored as an array of up to 8 elements, each of which contains the dwords described here. The start of each element is spaced 8 dwords apart. The first element of the array is aligned to a 32-byte boundary. The index with range 0-7 that selects which element is being used is multiplied by 2 to determine the **Sampler Index** in the message descriptor.

DWord	Bit	Description
0	31:24	**Denoise STAD Threshold.** Threshold for denoise sum of temporal absolute differences. Format = U8
	23:16	**Denoise Maximum History.** Maximum allowed value for denoise history. Format = U8 Range = [128,240]
	15	Reserved : MBZ
	14	**VDI Walker Frame Sharing Enable** For a GT2 system with 2 half-slices, this field controls how the frame is shared by the two deinterlacer walkers. 1 : The screen is shared by the two deinterlacers as controlled by the VDI Walker Y Stride 0 : There is only a single deinterlacer which must walk the entire frame. VDI Walker Y Stride is ignored.

DWord	Bit	Description
	13:12	**VDI Walker Y Stride** This field controls if the VDI walker skips pixels as it goes down the screen. This is used when a pair of VDI's are splitting the frame between them. The stride also implies the offset used by the 2nd half-slice. Format = U2 0 : Stride of 1 block (where a block is 4x4 when DI is enabled and 4x8 when DN only), offset for the 2nd half-slice is ½ the surface height. 1 : Stride of 2 blocks (every other row of blocks calculated by this VDI), offset for the 2nd half-slice is 1 block. 2 : Stride of 4 blocks (2 vertical blocks calclated by this VDI, then skip 2), offset for the 2nd half-slice is 2 blocks 3 : Stride of 8 blocks (4 vertical blocks calculated by this VDI, then skip 4), offset for the 2nd half-slice is 4 blocks.
	11:8	**dnmh_delta[3:0]** – Amount that denoise_history is increased Format = UINT4 Default = 8 **MAX: 15**
	7:0	**Denoise ASD Threshold.** Threshold for denoise absolute sum of differences. Format = U8 Range = [0,63]
1	31:30	Reserved : MBZ
	29:24	**Temporal Difference Threshold.** Format = U6 **Programming Notes:** ○ **Temporal Difference Threshold** – **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16.
	23:22	Reserved : MBZ
	21:16	**Low Temporal Difference Threshold.** Format = U6 **Programming Notes:** ○ **Temporal Difference Threshold** – **Low Temporal Difference Threshold** must be larger than 0 and less than or equal to 16.
	15:13	**STMM C2:** Bias for divisor in STMM equation. Format = U3 Range = [0,7] representing values [1,8]
	12:8	**Denoise Moving Pixel Threshold.** Threshold for number of moving pixels to declare a block to be moving. Format = U5 Range = [0,16]
	7:0	**Denoise Threshold for Sum of Complexity Measure.** Format = U8
2	31:30	Reserved : MBZ

DWord	Bit	Description
	29:24	**good_neighbor_th[5:0]** – Maximum difference from current pixel for neighboring pixels to be considered a good neighbor. Format = UINT6 Default = 4 (depending on GNE of previous frame) **MAX: 63**
	23:20	**CAT_slope_minus_1 –** Determines the slope of the Content Adaptive Threshold. +1 added internally to get CAT_slope. Format = U4 Default = 9 (CAT_slope value = 10)
	19:16	**SAD_Tight_th** Format = U4 Default = 5
	15:14	**smooth_mv_th** Format = U2
	13:12	Reserved : MBZ
	11:8	**bne_edge_th[3:0]** – Threshold for detecting an edge in block noise estimate. Format = UINT4 Default = 1 **MAX: 15**
	7:0	**Block Noise Estimate Noise Threshold.** Threshold for noise maximum/minimum. Format = U8 Range = [0,31]
3	31	**STMM Blending Constant Select.** Format = U1 0: Use the blending constant for small values of STMM for stmm_md_th 1: Use the blending constant for large values of STMM for stmm_md_th
	30:24	**Blending constant across time for large values of STMM.** Format = U7
	23:16	**Blending constant across time for small values of STMM.** Format = U8
	15:14	Reserved : MBZ
	13:8	**Multiplier for VECM.** Determines the strength of the vertical edge complexity measure. Format = U6
	7:0	**Maximum STMM.** Largest allowed STMM in blending equations. Format = U8
4	31:24	**Minimum STMM.** Smallest allowed STMM in blending equations. Format = U8

DWord	Bit	Description
	23:22	**STMM Shift Down.** Amount to shift STMM down (quantize to fewer bits). Format = U2 0: Shift by 4 1: Shift by 5 2: Shift by 6 3: Reserved
	21:20	**STMM Shift Up.** Amount to shift STMM up (set range). Format = U2 0: Shift by 6 1: Shift by 7 2: Shift by 8 3: Reserved
	19:16	**STMM Output Shift.** Amount to shift output of STMM blend equation**.** **Programming Notes:** • The value of this field must satisfy the following equation: stmm_max – stmm_min = 2 ^ stmm_output_shift Format = U4 Range = [0,16]
	15:8	**SDI Threshold.** Threshold for angle detection in SDI algorithm. Format = U8
	7:0	**SDI Delta.** Delta value for angle detection in SDI algorithm. Format = U8
5	31:24	**SDI Fallback Mode 1 T1 Constant.** Format = U8
	23:16	**SDI Fallback Mode 1 T2 Constant.** Format = U8
	15:8	**SDI Fallback Mode 2 Constant (Angle2x1).** Format = U8
	7:0	**FMD Temporal Difference Threshold.** Format = U8
6	31:24	**FMD #1 Vertical Difference Threshold.** Format = U8
	23:16	**FMD #2 Vertical Difference Threshold.** Format = U8
	15:14	**CAT_th1** Format = U2 Default = 0
	13:8	**FMD Tear Threshold.** Format = U6
	7	**MCDI Enable** – Use Motion Compensated Deinterlace algorithm. Ignored if DI Enable is off.

DWord	Bit	Description
	6	**Progressive DN.** Indicates that the denoise algorithm should assume progressive input when filtering neighboring pixels. **DI Enable** must be disabled when this field is enabled. Format = Enable 0: DN assumes interlaced video and filters alternate lines together 1: DN assumes progressive video and filters neighboring lines together
	5	**DN/DI First Frame.** Indicates that this is the first frame of the stream, so previous clean is not available Format = Enable 0: Not first field; previous clean surface state is valid 1: First field; previous clean surface state is invalid
	4	**DN/DI Stream ID.** Distinguishes between the two simultaneous streams that are supported. Used to update the GNE and FMD counters for that stream. Format = U1
	3	**DN/DI Top First.** Indicates the top field is first in sequence, otherwise bottom is first Format = Enable 0 = Bottom field occurs first in sequence 1 = Top field occurs first in sequence
	2	**DI Partial.** If **DI Enable** and **DI Partial** are both enabled, the deinterlacer will output the partial VDI writeback message. Format = Enable 0: Output normal VDI writeback message (only if **DI Enable** is enabled also) 1: Output partial VDI writeback message (only if **DI Enable** is enabled also)
	1	**DI Enable.** Deinterlacer is bypassed if this is disabled: the output is the same as the input (same as a 2:2 cadence). FMD and STMM are not calculated and the values in the response message are 0. Format = Enable 0: Do not calculate DI 1: Calculate DI **Programming Notes:** ○ **DI Enable** and **DN Enable** cannot both be disabled.
	0	**DN Enable.** Denoise is bypassed if this is low – BNE is still calculated and output, but the denoised fields are not. VDI does not read in the denoised previous frame but uses the pointer for the original previous frame. Format = Enable 0: Do not denoise frame 1: Denoise frame **Programming Notes:** ○ **DI Enable** and **DN Enable** cannot both be disabled.
7	31:23	**Column Width Minus1** This field specifies the (column width-1) / stride in units of blocks (Each blocks has width 16 pixels). A column width * 16 that equals the width of the frame means the walker will walk to the end of the frame. Format = U9 Range = [0, 511] representing column widths [1 to 512] (interpret value as binary value + 1)

DWord	Bit	Description
	22:19	**NeighborPixel_th** Format = U4 Default = 10
	18	**VDI Walker Enable** Format = U1 0: Walker Disabled. Use XY generated by Driver. 1: Walker Enabled. Use XY generated by VDIunit. Programming Note for Gen7+:When walker is enabled in a GT2 system, the MEDIA_OBJECT commands dispatching work to the VDI must use the Half-Slice Destination Select field to split the work between the two half-slices; the Half-Slice Destination Select must never be set to 00 (either half-slice).
	17:16	**FMD for 2nd field of previous frame.** Format = U2 0: Deinterlace (not progressive output) 1: Put together with previous field in sequence (1st field of previous frame). 2: Put together with next field in sequence (1st field of current frame).
	15:10	**MC_pixel_consistency_th** Format = U6 Default = 25
	9:8	**FMD for 1st field of current frame.** Format = U2 0: Deinterlace (not progressive output). 1: Put together with previous field in sequence (2nd field of previous frame). 2: Put together with next field in sequence (2nd field of current frame).
	7:4	**SAD_THB** Format = U4 Default = 10
	3:0	**SAD_THA** Format = U4 Default = 5

4.10.4 SAMPLER_8x8_STATE [DevILK+]

The 8x8 coefficients and other state used by the sample_8x8 message are stored as indirect state, pointed to by a field in SAMPLER_STATE. There are four different tables loaded using this structure (0X, 0Y, 1X, and 1Y). Each table is stored as an array of 17 elements, each with either 4 or 8 coefficients.

DWord	Bit	Description
0	31:24	**Table 0X Filter Coefficient[0,3]** Format = S1.6 in 2's complement format **[ILK]:** Range = [0.0, +2.0)

DWord	Bit	Description
	23:16	**Table 0X Filter Coefficient[0,2]** Format = S1.6 in 2's complement format Range = [-1, +1)
	15:8	**Table 0X Filter Coefficient[0,1]** Format = S1.6 in 2's complement format Range = [-2^{-1}, $+2^{-1}$) **Programming Notes:** • Must be zero if the format is R10G10B10A2_UNORM or R8G8B8A8_UNORM
	7:0	**Table 0X Filter Coefficient[0,0]** Format = S1.6 in 2's complement format Range = [-2^{-2}, $+2^{-2}$) **Programming Notes:** • Must be zero if the format is R10G10B10A2_UNORM or R8G8B8A8_UNORM
1	31:24	**Table 0X Filter Coefficient[0,7]** Format = S1.6 in 2's complement format Range = [-2^{-2}, $+2^{-2}$)
	23:16	**Table 0X Filter Coefficient[0,6]** Format = S1.6 in 2's complement format Range = [-2^{-1}, $+2^{-1}$)
	15:8	**Table 0X Filter Coefficient[0,5]** Format = S1.6 in 2's complement format Range = [-1, +1)
	7:0	**Table 0X Filter Coefficient[0,4]** Format = S1.6 in 2's complement format **[DevSNB]:** Range = [0.0, +2.0)
2:3		**Table 0Y Filter Coefficient[0,7:0]** This table has the same layout as Table 0X above.
4	31:24	**Table 1X Filter Coefficient[0,3]** Format = S1.6 in 2's complement format Range = [0.0, +2.0)
	23:16	**Table 1X Filter Coefficient[0,2]** Format = S1.6 in 2's complement format Range = [-1, +1)
	15:0	Reserved : MBZ
5	31:16	Reserved : MBZ
	15:8	**Table 1X Filter Coefficient[0,5]** Format = S1.6 in 2's complement format Range = [-1, +1)
	7:0	**Table 1X Filter Coefficient[0,4]** Format = S1.6 in 2's complement format Range = [0.0, +2.0)

DWord	Bit	Description
6:7		**Table 1Y Filter Coefficient[0,7:0]** This table has the same layout as Table 1X above.
8:15		**Filter Coefficient[1,7:0]**
16:23		**Filter Coefficient[2,7:0]**
…		
128:135		**Filter Coefficient[16,7:0]**
136	31:24	**Default Sharpness Level.** When adaptive scaling is off, determines the balance between sharp and smooth scalers. Format = U8 0: contribute 1 from the smooth scalar 255: contribute 1 from the sharp scalar
	23:16	**Max Derivative 4 Pixels.** Used in adaptive filtering to specify the lower boundary of the smooth 4 pixel area. Format = U8
	15:8	**Max Derivative 8 Pixels.** Used in adaptive filtering to specify the lower boundary of the smooth 8 pixel area. Format = U8
	7	Reserved : MBZ
	6:4	**Transition Area with 4 Pixels.** Used in adaptive filtering to specify the width of the transition area for the 4 pixel calculation. Format = U3
	3	Reserved : MBZ
	2:0	**Transition Area with 8 Pixels.** Used in adaptive filtering to specify the width of the transition area for the 8 pixel calculation. Format = U3
137	31:23	Reserved : MBZ
	22	**Bypass X Adaptive Filtering.** When disabled, the X direction will use **Default Sharpness Level** to blend between the smooth and sharp filters rather than the calculated value. Format = Disable 1: Disable X adaptive filtering 0: Enable X adaptive filtering
	21	**Bypass Y Adaptive Filtering.** When disabled the, Y direction will use Default Sharpness Level to blend between the smooth and sharp filters rather than the calculated value. Format = Disable 1: Disable X adaptive filtering 0: Enable X adaptive filtering
	20:0	Reserved : MBZ

4.10.5 SAMPLER_BORDER_COLOR_STATE

This structure is pointed to by a field in SAMPLER_STATE.

- For surface formats with one or more channels missing, the value from the border color is not used for the missing channels, resulting in these channels resulting in the overall default value (0 for colors and 1 for alpha) regardless of whether border color is chosen. The surface formats with “L” and “I” have special behavior with respect to the border color. The border color value used for the replicated channels (RGB for

“L” formats and RGBA for “I” formats) comes from the *red* channel of border color. In these cases, the green and blue channels, and also alpha for “I”, of the border color are ignored.

Programming Notes:

- The conditions under which this color is used depend on the **Surface Type** – 1D/2D/3D surfaces use the border color when the coordinates extend beyond the surface extent; cube surfaces use the border color for “empty” (disabled) faces.
- The border color itself is accessed through the texture cache hierarchy rather than the state cache hierarchy. Thus, if the border color is changed in memory, the texture cache must be invalidated and the state cache does not need to be invalidated.
- MAPFILTER_MONO: The border color is ignored. Border color is fixed at a value of 0 by hardware.

4.10.5.1 [DevILK+] and [DevSNB]

For [DevIILK], if border color is used, all formats must be provided. Hardware will choose the appropriate format based on **Surface Format**. The values represented by each format should be the same (other than being subject to range-based clamping and precision) to avoid unexpected behavior.

DWord	Bit	Description
0	31:24	**Border Color Alpha** Format = UNORM8
	23:16	**Border Color Blue** Format = UNORM8
	15:8	**Border Color Green** Format = UNORM8
	7:0	**Border Color Red** Format = UNORM8
1	31:24	**Border Color Alpha** Format = SNORM8
	23:16	**Border Color Blue** Format = SNORM8
	15:8	**Border Color Green** Format = SNORM8
	7:0	**Border Color Red** Format = SNORM8
2	31:0	**Border Color Red** Format = IEEE_FP
3	31:0	**Border Color Green** Format = IEEE_FP

DWord	Bit	Description
4	31:0	**Border Color Blue** Format = IEEE_FP
5	31:0	**Border Color Alpha** Format = IEEE_FP
6	31:16	**Border Color Green** Format = FLOAT_16
	15:0	**Border Color Red** Format = FLOAT_16
7	31:16	**Border Color Alpha** Format = FLOAT_16
	15:0	**Border Color Blue** Format = FLOAT_16
8	31:16	**Border Color Green** Format = UNORM16
	15:0	**Border Color Red** Format = UNORM16
9	31:16	**Border Color Alpha** Format = UNORM16
	15:0	**Border Color Blue** Format = UNORM16
10	31:16	**Border Color Green** Format = SNORM16
	15:0	**Border Color Red** Format = SNORM16
11	31:16	**Border Color Alpha** Format = SNORM16
	15:0	**Border Color Blue** Format = SNORM16

4.10.6 3DSTATE_CHROMA_KEY

3DSTATE_CHROMA_KEY

3DSTATE_CHROMA_KEY			
Project:	All	**Length Bias:**	2

The 3DSTATE_CHROMA_KEY instruction is used to program texture color/chroma-key key values. A table containing four set of values is supported. The **ChromaKey Index** sampler state variable is used to select which table entry is associated with the map. Texture chromakey functions are enabled and controlled via use of the **ChromaKey Enable** texture sampler state variable.

Texture Color Key (keying on a paletted texture index) is not supported.

DWord	Bit	Description
0	31:29	*Command Type* ***Default Value:*** ***3h*** ***GFXPIPE*** ***Format:*** ***OpCode***
	28:27	*Command SubType* ***Default Value:*** ***3h*** ***GFXPIPE_3D*** ***Format:*** ***OpCode***
	26:24	*3D Command Opcode* ***Default Value:*** ***1h*** ***3DSTATE*** ***Format:*** ***OpCode***
	23:16	*3D Command Sub Opcode* ***Default Value:*** ***04h*** ***3DSTATE_CHROMA_KEY*** ***Format:*** ***OpCode***
	15:8	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
	7:0	*DWord Length* ***Default Value:*** ***2h*** ***Excludes DWord (0,1)*** ***Format:*** ***=n*** ***Total Length - 2***
1	31:30	*ChromaKey Table Index* ***Project:*** ***All*** ***Format:*** ***U2*** ***index*** ***Range*** ***0..3*** ***Selects which entry in the ChromaKey table is to be loaded***
	29:0	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
2	31:0	*ChromaKey Low Value* ***This field specifies the "low" (minimum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range.*** ***See*** *ChromaKey High Value* ***for further format, programming info.***

3DSTATE_CHROMA_KEY		
3	31:0	*ChromaKey High Value* ***This field specifies the "high" (maximum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range.*** *Programming Notes* ***ChromaKey values are specified using 8-bit channels. When using surface formats with less than 8 bits per channel, the device will expand channels by replicating the required number of MSBs into the LSBs of each channel. Software must account for this conversion when it programs Chromakey Low/High Values (e.g., by performing the same replication).*** ***For channels that do not exist in the actual surface (e.g., Alpha channel for non-ARGB maps), software must explicitly program full range high/low values (High=FFh, Low=0h for formats using unsigned chroma key values, High=7Fh, Low=FFh for formats using sign magnitude chroma key values) in order to effectively remove the comparison of that field from the ChromaKey function.*** ***For channels in SNORM format in the surface format, the value in the high/low value for that channel is interpreted in* sign magnitude *format. Negative zero value is not supported (use positive zero instead). For channels with mixed UNORM/SNORM formats (i.e. R5G5_SNORM_B6_UNORM), the ChromaKey is programmed as if all channels are SNORM.*** ***YUV ChromaKey will use an interpolated chrominance value from the map for comparison to the chroma key values for those texels without chrominance due to downsampling. The chrominance value used is the average of values to the left and right of the texel in question.*** ***It is UNDEFINED to program any component of the ChromaKey High Value to be less than the corresponding component of ChromaKey Low Value.*** ***Format = interpreted according to associated texel format "class":*** ***Only the surface formats listed as supported for chroma key in the surface formats table can be used with this feature. Use of any other surface format with chroma key enabled is UNDEFINED.***

Surface Format	31:24	23:16	15:8	7:0
YCrCb formats	A	Cr	Y	Cb

4.10.7 3DSTATE_SAMPLER_PALETTE_LOAD0

3DSTATE_SAMPLER_PALETTE_LOAD0			
Project:	All	**Length Bias:**	2

The 3DSTATE_SAMPLER_PALETTE_LOAD0 instruction is used to load 24-bit ([DevBW], [DevCL]) or 32-bit ([DevCTG-A+]) values into the first texture palette. The texture palette is used whenever a texture with a paletted format (containing "Px [palette0]") is referenced by the sampler.

[DevBW] and [DevCL]: This instruction is used to load all or a subset of the 16 entries of the first palette. Partial loads always start from the first (index 0) entry.

[DevCTG-A+]: This instruction is used to load all or a subset of the 256 entries of the first palette. Partial loads always start from the first (index 0) entry.

DWord	Bit	Description
0	31:29	*Command Type* ***Default Value:*** ***3h*** ***GFXPIPE*** ***Format:*** ***OpCode***
	28:27	*Command SubType* ***Default Value:*** ***3h*** ***GFXPIPE_3D*** ***Format:*** ***OpCode***
	26:24	*3D Command Opcode* ***Default Value:*** ***1h*** ***3DSTATE*** ***Format:*** ***OpCode***
	23:16	*3D Command Sub Opcode* ***Default Value:*** ***02h*** ***3DSTATE_SAMPLER_PALETTE_LOAD0*** ***Format:*** ***OpCode***
	15:8	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
	7:0	*DWord Length* ***Default Value:*** ***0h*** ***Excludes DWord (0,1)*** ***Format:*** ***=n*** ***Total Length - 2***
1..n	31:24	*Palette Alpha[0:N-1]* ***Project:*** ***[DevCTG-A+]*** ***Alpha values loaded into the first N entries of the texture palette.*** ***Format = U8***
	23:0	*Palette Color[0:N-1]* ***Project:*** ***All*** ***Colors loaded into the first N entries of the texture color palette.*** ***Format = Bits 23:0 = U24 interpreted as RGB_888 color as follows:*** ***[23:16] Red*** ***[15:8] Green*** ***[7:0] Blue***

4.10.8 3DSTATE_SAMPLER_PALETTE_LOAD1 [DevCTG-B+]

3DSTATE_SAMPLER_PALETTE_LOAD1			
Project:	[DevCTG-B+]	**Length Bias:**	2

The 3DSTATE_SAMPLER_PALETTE_LOAD1 instruction is used to load 32-bit values into the second texture palette. The second texture palette is used whenever a texture with a paletted format (containing "Px...[palette1]") is referenced by the sampler.

This instruction is used to load all or a subset of the 256 entries of the second palette. Partial loads always start from the first (index 0) entry.

DWord	Bit	Description
0	31:29	*Command Type* ***Default Value:*** ***3h*** ***GFXPIPE*** ***Format:*** ***OpCode***
	28:27	*Command SubType* ***Default Value:*** ***3h*** ***GFXPIPE_3D*** ***Format:*** ***OpCode***
	26:24	*3D Command Opcode* ***Default Value:*** ***1h*** ***3DSTATE*** ***Format:*** ***OpCode***
	23:16	*3D Command Sub Opcode* ***Default Value:*** ***0Ch*** ***3DSTATE_SAMPLER_PALETTE_LOAD1*** ***Format:*** ***OpCode***
	15:8	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
	7:0	*DWord Length* ***Default Value:*** ***0h*** ***Excludes DWord (0,1)*** ***Format:*** ***=n*** ***Total Length - 2***
1..n	31:0	*Palette Color[0:N-1]* ***Project:*** ***All*** ***Colors loaded into the first N entries of the texture color palette.*** ***Format = Bits 31:0 = U32 interpreted as ARGB_8888 color as follows:*** ***[31:24] Alpha*** ***[23:16] Red*** ***[15:8] Green*** ***[7:0] Blue***

4.10.9 3DSTATE_MONOFILTER_SIZE [DevILK+]

3DSTATE_MONOFILTER_SIZE		
Project: [DevILK+]		**Length Bias:** 2
This state specifies the size of the filter which is used when filtering in MAPFILTER_MONO mode.		
DWord	**Bit**	**Description**
0	31:29	*Command Type* ***Default Value:*** ***3h*** ***GFXPIPE*** ***Format:*** ***OpCode***
	28:27	*Command SubType* ***Default Value:*** ***3h*** ***GFXPIPE_3D*** ***Format:*** ***OpCode***
	26:24	*3D Command Opcode* ***Default Value:*** ***1h*** ***3DSTATE_NONPIPELINED*** ***Format:*** ***OpCode***
	23:16	*3D Command Sub Opcode* ***Default Value:*** ***11h*** ***3DSTATE_MONOFILTER_SIZE*** ***Format:*** ***OpCode***
	15:8	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
	7:0	*DWord Length* ***Default Value:*** ***0h*** ***Excludes DWord (0,1)*** ***Format:*** ***=n*** ***Total Length - 2*** ***Project:*** ***All***
1	31:6	*Reserved* ***Project:*** ***All*** ***Format:*** ***MBZ***
	5:3	*Monochrome Filter Width* ***Project:*** ***All*** ***Format:*** ***U3*** ***FormatDesc*** ***Range*** ***[1,7]*** ***This field specifies the width of the monochrome filter. It is ignored if the monochrome filter is not enabled.***
	2:0	*Monochrome Filter Height* ***Project:*** ***All*** ***Format:*** ***U3*** ***FormatDesc*** ***Range*** ***[1,7]*** ***This field specifies the height of the monochrome filter. It is ignored if the monochrome filter is not enabled.***

4.11 Messages

Restrictions:

- Use of any message to the Sampling Engine function with the **End of Thread** bit set in the message descriptor is not allowed.
- **[DevBW-A,B,C0, DevCL-A0] Errata:** use of any Sampling Engine message in the same workload (between pipeline flushes) with any Data Port read messages utilizing the Sampler Cache or Data Cache is not allowed.

4.11.1 Initiating Message

Execution Mask

SIMD16. The 16-bit execution mask forms the valid pixel signals. This determines which pixels are sampled and results returned to the GRF registers. Samples for invalid pixels are not overwritten in the GRF. However, if LOD needs to be computed for a subspan based on the message type and MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

SIMD8. The lower 8 bits of the execution mask forms the valid pixel signals. If LOD needs to be computed based on MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

SIMD4x2. The lower 8 bits of the execution mask is interpreted in groups of four. If any of the high 4 bits are asserted, that sample is valid. If any of the low 4 bits are asserted, that sample is valid. The **Write Channel Mask** rather than the execution mask determines which channels are written back to the GRF.

SIMD32. The execution mask is ignored, all pixels are considered valid and all channels are returned regardless of the execution mask.

4.11.1.1 Message Descriptor

4.11.1.1.1 [DevBW] and [DevCL]

The following message descriptor applies to [DevBW] and [DevCL].

Bit	Description
15:14	**Message Type:** Specifies the type of message being sent, along with the message length (in the general message descriptor) Format = U2 Refer to the table in section 4.11.1.3 for encoding details.
13:12	**Data Return Format:** Specifies the format of the data returned to the requesting thread. 00: FLOAT32 – return a signed 32-bit IEEE Float to the thread. Required for all UNORM, SNORM, and FLOAT surface formats. Also required for all resinfo messages. 01: Reserved 10: UINT32 – return an unsigned 32-bit integer. Required for all UINT surface formats. 11: SINT32 – return a signed 32-bit 2's complement integer. Required for all SINT surface formats.
11:8	**Sampler Index:** Specifies the index into the sampler state table. Ignored for "ld" and "resinfo" type messages. Format = U4 Range = [0,15]
7:0	**Binding Table Index:** Specifies the index into the binding table. Format = U8 Range = [0,255]

4.11.1.1.2 [DevCTG]

The following message descriptor applies to [DevCTG]. The **Data Return Format** Field has been removed. The data return format used by the sampling engine depends on the **Surface Format** of the surface being sampled. UINT formats return UINT32, SINT formats return SINT32, and all other formats return FLOAT32. The resinfo instruction returns UINT32 only. If FLOAT32 is desired, the conversion must be done in the kernel.

Bit	Description
15:12	**Message Type:** Specifies the type of message being sent, along with the message length (in the general message descriptor) Format = U4 Refer to the table in section 4.11.1.3 for encoding details.
11:8	**Sampler Index:** Specifies the index into the sampler state table. Ignored for "ld" and "resinfo" type messages. Format = U4 Range = [0,15]
7:0	**Binding Table Index:** Specifies the index into the binding table. Format = U8 Range = [0,255]

4.11.1.1.3 [DevILK+]

The following message descriptor applies to [DevILK+]. Four more bits have been added to the message descriptor.

Bit	Description
19	**Header Present:** Specifies whether the message includes a header phase. If the header is not present (this field is zero), all of the fields normally contained in the header are assumed to be 0. Format = Enable
18	Reserved : MBZ
17:16	**SIMD Mode:** Specifies the SIMD mode of the message being sent. Format = U2 0 = SIMD4x2 1 = SIMD8 2 = SIMD16 3 = SIMD32/64
15:12	**Message Type:** Specifies the type of message being sent. Format = U4 Refer to the table in section 4.11.1.3.2 for encoding details.
11:8	**Sampler Index:** Specifies the index into the sampler state table. Ignored for "ld", "resinfo", and "sampleinfo" type messages. Format = U4 Range = [0,15] **Programming Notes:** • for the deinterlace message, this field must be a multiple of 2 (even) • for the sample_8x8 message, this field must be a multiple of 4
7:0	**Binding Table Index:** Specifies the index into the binding table. Format = U8 Range = [0,255]

4.11.1.2 Message Header

The message header for the sampling engine is the same regardless of the message type. If the header is not present (**[DevILK+]** only), behavior is as if the message was sent with all fields in the header set to zero (write channel masks are all enabled and offsets are zero).

DWord	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:5	**[Pre-DevILK]: Sampler State Pointer:** Specifies the 32-byte aligned pointer to the sampler state table. This field is ignored for "ld" and "resinfo" message types. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:5] **[DevILK+]:** Ignored
	4:0	Ignored
M0.2	31:17	Ignored
	16	**[Pre-DevILK]: Force LOD to Zero:** If this bit is enabled, the calculated LOD is replaced with zero. The LOD is replaced just before entering the pseudocode in section 4.2.1.5, therefore the LOD is still subject to bias, overriding by sample_l delivered LOD, and clamping. Format = Enable **[DevILK+]:** Ignored
	15	**Alpha Write Channel Mask:** Enables the alpha channel to be written back to the originating thread. 0: Alpha channel will be written back 1: Alpha channel will not be written back **Programming Notes:** • a message with all four channels masked is not allowed. • **[Pre-DevSNB]:** this field is ignored for the sample_unorm*. The write channel mask is generated from the message type itself. • this field is ignored for the deinterlace message. • this field must be set to zero for sample_8x8 in VSA mode.
	14	**Blue Write Channel Mask:** See Alpha Write Channel Mask
	13	**Green Write Channel Mask:** See Alpha Write Channel Mask
	12	**Red Write Channel Mask:** See Alpha Write Channel Mask
	11:8	**Reserved**
	7:4	**Reserved**
	3:0	**Reserved**

DWord	Bit	Description
M0.1	31:0	Ignored
M0.0	31:0	Ignored

4.11.1.3 Payload Parameter Definition

The table below shows all of the messages supported by the sampling engine. The message type field in the message descriptor in combination with the message length determines which message is being sent. The table defines all of the *parameters* sent for each message type. The position of the parameters in the payload is given in the section following corresponding to the *SIMD mode* given in the table.

All parameters are of type IEEE_Float, except those in the ld and resinfo instruction message types, which are of type S31. Any parameter indicated with a blank entry in the table is unused. A message register containing only unused parameters not included as part of the message. The response lengths given below assume all channels are unmasked. SIMD16 messages with masked channels will have reduced response length.

4.11.1.3.1 [Pre-DevILK]

[DevBW] and [DevCL] message type	[DevCTG+] message type	Message length	Response length	parameters 0	1	2	3	4	5	6	7	8	9	10	11	SIMD mode	API shader instruction
00	0000	3	8	u												SIMD16	sample
00	0000	5	8	u	v											SIMD16	sample
00	0000	7	8	u	v	r										SIMD16	sample
00	0000	4	4	u	v	r										SIMD8	sample
01	0001	4	5	u	v	r										SIMD8	sample+killpix
00	0000	9	8	u	v	r	bias									SIMD16	sample_b
01	0001	9	8	u	v	r	lod									SIMD16	sample_l
01	0001	2	1	u	v	r	lod									SIMD4x2	sample_l
10	0010	9	8	u	v	r	ref									SIMD16	sample_c
00	0000	2	1	u	v	r	ref									SIMD4x2	sample_c
00	0000	6	4	u	v	r	bias	ref								SIMD8	sample_b_c
01	0001	6	4	u	v	r	lod	ref								SIMD8	sample_l_c
01	0001	3	1	u	v	r	lod	ref								SIMD4x2	sample_l_c
11	0011	3	8	u												SIMD16	ld
11	0011	5	8	u	v											SIMD16	ld
11	0011	7	8	u	v	r										SIMD16	ld
11	0011	4	4	u	v	r										SIMD8	ld
11	0011	9	8	u	v	r	lod									SIMD16	ld
11	0011	2	1	u	v	r	lod									SIMD4x2	ld
10	0010	7	4	u	v	dudx		dudy								SIMD8	sample_g
10	0010	10	4	u	v	r	dudx		drdx	dudy		drdy				SIMD8	sample_g
10	0010	4	1	u	v	r		dudx		drdx		dudy		drdy		SIMD4x2	sample_g
10	0010	3	8	lod												SIMD16	resinfo
10	0010	2	1				lod									SIMD4x2	resinfo
N/A	0100	2	8	payload details in "SIMD32 Payload" section												SIMD32	sample_unorm
N/A	0101	2	4	payload details in "SIMD32 Payload" section												SIMD32	sample_unorm_RG
N/A	0110	2	5	payload details in "SIMD32 Payload" section												SIMD32	sample_unorm_RG +killpix

Note that the SIMD8 messages actually contain only eight pixels of data. For the sample_g messages, this is due to the message length constraint of 16 registers not allowing these messages of 16 pixels. The Jitter will need to send two messages to the sampler to get 16 pixels of data.

4.11.1.3.2 [DevILK+]

The table below shows all of the message types supported by the sampling engine. The **Message Type** field in the message descriptor determines which message is being sent. The **SIMD Mode** field determines the number of instances (i.e. pixels) and the formatting of the initiating and writeback messages. The **Header Present** field determines whether a header is delivered as the first phase of the message or the default header from R0 of the thread's dispatch is used. The **Message Length** field is used to vary the number of parameters sent with each message. Higher-numbered parameters are optional, and default to a value of 0 if not sent but needed for the surface being sampled.

The message lengths are computed as follows, where "N" is the number of parameters ("N" is rounded up to the next multiple of 4 for SIMD4x2), and "H" is 1 if the header is present, 0 otherwise. The maximum message length allowed to the sampler is 11. This would disallow sample_d, sample_b_c, and sample_l_c with a SIMD Mode of SIMD16.

SIMD Mode	*Message Length*
SIMD4x2	*H + (N/4)*
SIMD8	*H + N*
SIMD16	*H + (2*N)*

The response lengths are computed as follows:

SIMD Mode		Response Length
SIMD4x2		***1***
SIMD8	sample+killpix	***5***
	all other message types	***4***
SIMD16		***8 ****

* For SIMD16, phases in the response length are reduced by 2 for each channel that is masked.

SIMD16 messages with six or more parameters exceed the maximum message length allowed, in which case they are not supported. This includes some forms of sample_d, sample_b_c, and sample_l_c message types.

SIMD16 messages with six or more parameters exceed the maximum message length allowed, in which case they are not supported. This includes some forms of sample_d, sample_b_c, and sample_l_c message types.

SIMD4x2, SIMD8, and SIMD16 Messages:

Message Type	mnemonic	parameters									
		0	1	2	3	4	5	6	7	8	9
0000	sample	u	v	r	ai						
0001	sample_b	u	v	r	ai	bias					
0010	sample_l	u	v	r	ai	lod					
0011	sample_c	u	v	r	ai	ref					
0100	sample_d	u	v	r	ai	dudx	dudy			drdx	drdy
0101	sample_b_c	u	v	r	ai	ref	bias				
0110	sample_l_c	u	v	r	ai	ref	lod				
0111	ld	u	v	r	lod	si					
1000*	load4	u	v	r	ai						
1001*	LOD	u	v	r	ai						
1010	resinfo	lod									
1011*	sampleinfo										
1100	sample+killpix	u	v	r							

4.11.1.4 Message Types

The behavior of each message type is as follows:

Message Type	Description
sample ***sample2dms***	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels. One, two, or three parameters may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface cannot be MONO8 or any UINT or SINT format. • sample is not supported in SIMD4x2 mode.
sample+killpix	The surface is sampled as in the sample message type. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface cannot be MONO8 or any UINT or SINT format. • sample+killpix is supported only in SIMD8 mode.
sample_b	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels, then the value in the parameter is added to the LOD for each pixel. The LOD bias delivered in the bias parameter is restricted to a range of [-16.0, +16.0). Values outside this range produce undefined results. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface cannot be MONO8 or any UINT or SINT format. • sample_b is not supported in SIMD4x2 mode.
sample_l	The surface is sampled using the indicated sampler state. LOD is not computed, but instead is taken from the lod parameter. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The **Surface Format** of the associated surface cannot be a UINT or SINT format.

Message Type	Description
sample_c	The surface is sampled using the indicated sampler state. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. The ai parameter indicates the array index for a cube surface. The ref parameter specifies the reference value that is compared against the red channel of the sampled surface, and the texel is replaced with either white or black depending on the result of the comparison. The WGF sample_c_lz instruction is implemented by issuing the sample_c message with Force LOD to Zero enabled in the message header or by issuing the sample_l_c message with the LOD parameter set to zero. **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. • 1D and 2D arrays are not supported on **[Pre-*DevGT*]** (**Depth** of the associated surface must be 0). • The **Surface Format** of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. • With sample_c, MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed even for surface formats that are listed as not supporting filtering in the surface formats table. • Use of the SIMD4x2 form of sample_c without **Force LOD to Zero** enabled in the message header is not allowed, as it is not possible for the hardware to compute LOD for SIMD4x2 messages. For **[*ILK*]**, sample_c is not supported in SIMD4x2 mode. • Use of sample_c with SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, A32_FLOAT. • **[DevBW, DevCL] Errata:** When sample_c is used on a texture map with A16_FLOAT surface format, any value read in from the texture map that is a NaN will be treated like a + inf. • **[Pre-*ILK*] Errata:** When either the reference value or the source value from the texture map is NaN the compare value will be incorrectly replaced with 1.0 rather than 0.0 for Shadow Function of GEQUAL, GREATER, LEQUAL, or LESS.
sample_b_c	This is a combination of sample_b and sample_c. Both the LOD bias and reference values are delivered. All restrictions applying to both sample_b and sample_c must be honored.
sample_l_c	This is a combination of sample_l and sample_c. Both the LOD and reference values are delivered. All restrictions applying to both sample_l and sample_c must be honored. However, unlike sample_c, sample_l_c is allowed as a SIMD4x2 message. **Programming Notes:** • **[DevBW, DevCL] Errata:** SIMD4x2 sample_l_c is not allowed and must be worked around using SIMD8 sample_l_c.

<table>
<tr><th>Message Type</th><th>Description</th></tr>
<tr><td>sample_g

sample_d</td><td>The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The r coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The r coordinate contains the faceid, and the r gradients are ignored by hardware.

Programming Notes:
The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.
The Surface Format of the associated surface cannot be MONO8 or any UINT or SINT format.</td></tr>
<tr><td>resinfo</td><td>The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned as indicated in the table below. The format of the returned data is FLOAT32 for [Pre-DevCTG], and UINT32 for [DevCTG+]. The width, height, and depth may be shifted right, per pixel, by the LOD value provided in the lod parameter to give the dimensions of the specified mip level. The lod parameter is an unsigned 32-bit integer in this mode (note that sending a signed 32-bit integer always has the same effect, as negative values are out-of-range when interpreted as unsigned integers). The Sampler State Pointer and Sampler Index are ignored.

<table>
<tr><th>surface type</th><th>red</th><th>green</th><th>blue</th><th>alpha</th></tr>
<tr><td>SURFTYPE_3D</td><td>(Width+1)>>LOD</td><td>(Height+1)>>LOD</td><td>(Depth+1)>>LOD</td><td>MIPCount</td></tr>
<tr><td>SURFTYPE_CUBE</td><td>(Wdith+1)>>LOD</td><td>(Height+1)>>LOD</td><td>[Pre-DevGT]: 0
Depth+1 : 0</td><td>MIPCount</td></tr>
</table>

Programming Notes:
[DevBW-A,B] Errata: if lod is > 0xf it must be forced to 0xf.</td></tr>
</table>

Message Type	Description
ld **ld2dms** ***ld_mcs***	The surface is sampled using a default sampler state, indicated below. The parameter contains the LOD of the mip map to be sampled. The v and r channel may be ignored depending on the indicated surface type. All incoming values are unsigned 32-bit integers in this mode. The u, v, and r parameters contain integer texel addresses on the LOD indicated in the parameter. The Sampler State Pointer and Sampler Index are ignored. For *these* message types, the sampler state is defaulted as follows: • min, mag, and mip filter modes are "nearest" • all address control modes are "zero" (a special mode in which any texel off the map or outside the MIP range of the surface has a value of zero in all channels, except for surface formats without an alpha channel, which will return a value of one in the alpha channel) **Programming Notes:** • The **Surface Type** of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER *for the* ld *message.* • *The **Surface Type** of the associated surface must be SURFTYPE_2D for the* ld_mcs *and* ld2dms *messages.* • **[DevBW-A,B] Errata:** Only non-array (Depth = 0) SURFTYPE_1D and SURFTYPE_2D are supported with. • The **Surface Format** of the associated surface cannot be MONO8. • **[DevBW, DevCL] Errata:** For ld with SURFTYPE_BUFFER the lod channel MBZ. • **[Pre-ILK] Errata:** Surface formats with 8 bits per channel and no alpha channel will return zero in the alpha channel. • **[Pre-ILK] Errata:** For the SIMD8 or SIMD4x2 forms of this message, the v parameter must be set to zero for non-array SURFTYPE_1D, and r must be set to zero for all SURFTYPE_1D and array SURFTYPE_2D surfaces.
sample_unorm	**[DevCTG+] only:** *The surface* is *sampled using* the *indicated* sampler *state. 32 contiguous pixels in a 8-wide by 4-high arrangement are sampled. The **U** and **V addresses** for the upper left pixel is delivered in this message along with a **Delta U** and **Delta V** parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:* *U(x,y) = U(0,0) + DeltaU * x* *V(x,y) = V(0,0) + DeltaV * y* ***Programming Notes:*** • *The **Surface Type** of the associated surface must be SURFTYPE_2D* • *The **Surface Format** of the associated surface must be UNORM with <= 8 bits per channel* • *The **MIP Count**, **Depth**, **Surface Min LOD**, and **Min Array Element** of the associated surface must be 0* • *The **Min** and **Mag Mode Filter** must be MAPFILTER_NEAREST or MAPFILTER_LINEAR* • *The **Mip Mode Filter** must be MIPFILTER_NONE* • *The **TCX** and **TCY Address Control Mode** cannot be TEXCOORDMODE_CLAMP_BORDER* • ***DeltaU** * **Width** of the associated surface must be less than or equal to 3.0* • ***DeltaV** * **Height** of the associated surface must be less than or equal to 3.0*

Message Type	Description
sample_unorm_*RG*	***[DevCTG] to [ILK]* only:** This message is identical to the sample_unorm message except it *only* returns the *red and green* channels in the writeback message. All restrictions of the sample_unorm message apply to this message also.
sample_*unorm_RG* ***+killpix***	***[DevCTG] to [ILK] only:*** *This message is identical to the* sample_unorm_RG *message except it returns a kill pixel mask in addition to the red and green channels in the writeback message. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. All restrictions of the* sample_unorm *message apply to this message also.*
sample_8x8	***[ILK] only:*** *The surface is sampled using an optional 8x8 filter followed by an optional image enhancement filter, using state defined in SAMPLER_STATE and 3DSTATE_SAMPLE_8x8. The input can be one of three configurations. 64 contiguous pixels in an 8-wide by 8-high arrangement, 100 contiguous pixels in a 10-wide by 10-high arrangement, or 144 contiguous pixels in a 12-wide by 12-high arrangement. The address control mode behaves as clamp mode. The U and V addresses for the upper left pixel are delivered in this message along with a Delta U and Delta V parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:* $U(x,y) = U(0,0) + DeltaU * x + U\_2^{nd}\_Derivative * x * (x - 1)/2$ $V(x,y) = V(0,0) + DeltaV * y$ ***Programming Notes:*** • *The* ***Surface Type*** *of the associated surface must be SURFTYPE_2D* • *The* ***Surface Format*** *of the associated surface must be UNORM with <= 10 bits per channel* • ***DeltaV*** * ***Height*** *of the associated surface must be less than 16.0* • ***Map Width*** *must be >= 4* • ***[ILK]:*** *If* sample_8x8 *or* deinterlace *messages are used in a thread, software must ensure that the same thread or other threads that can concurrently be running do not use any other sampling engine messages.*
deinterlace	***[ILK*** ***] only:*** *The surface is deinterlaced and/or denoised, using state defined in SAMPLER_STATE. The U and V addresses for the upper left pixel are delivered in this message.* ***Programming Notes:*** • ***[ILK]:*** *If* sample_8x8 *or* deinterlace *messages are used in a thread, software must ensure that the same thread or other threads that can concurrently be running do not use any other sampling engine messages.*

Programming Notes:

- For surfaces of type SURFTYPE_CUBE, the sampling engine requires u, v, and r parameters that have already been divided by the absolute value of the parameter (u, v, or r) with the largest absolute value.

4.11.1.5 Parameter Types

sample*, LOD, and gather4 messages

For all of the sample*, LOD, and gather4 message types, all parameters are 32-bit floating point, except the 'mcs', 'offu', and 'offv' parameters. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Normalized values range from [0,1] across the surface, with values outside the surface behaving as specified by the **Address Control Mode** in that dimension. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension, with values outside the surface being clamped to the surface.

Surface Type	u	v	r	ai
SURFTYPE_1D	normalized 'x' coordinate	unnormalized array index	ignored	ignored
SURFTYPE_2D	normalized 'x' coordinate	normalized 'y' coordinate	unnormalized array index	ignored
SURFTYPE_3D	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	ignored
SURFTYPE_CUBE	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	unnormalized array index

mcs parameter [DevILK+]

The 'mcs' parameter delivers the multisample control data. The format of this parameter is always a 32-bit unsigned integer. Refer to the section titled "Multisampled Surface Behavior" for details on this parameter.

Ld* messages

For the ld message types, all parameters are 32-bit signed integers, except the 'mcs' parameter. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension. Input of any value outside of the range returns zero.

Surface Type	u	v	r
SURFTYPE_1D	unnormalized 'x' coordinate	unnormalized array index	ignored
SURFTYPE_2D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized array index
SURFTYPE_3D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized 'z' coordinate
SURFTYPE_BUFFER	unnormalized 'x' coordinate	ignored	ignored

4.11.1.6 SIMD16 Payload

The payload of a SIMD16 message provides addresses for the sampling engine to process 16 entities (examples of an entity are vertex and pixel). The number of parameters required to sample the surface depends on the state that the sampler/surface is in. Each parameter takes two message registers, with 8 entities, each a 32-bit floating point value, being placed in each register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this. For example, a 2D map using "sample_b" needs only u, v, and bias, but must send the r parameter as well.

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in section 4.11.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**

DWord	Bit	Description
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2.7	31:0	**Subspan 3, Pixel 3 (lower right) Parameter 0**
M2.6	31:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**
M2.5	31:0	**Subspan 3, Pixel 1 (upper right) Parameter 0**
M2.4	31:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**
M2.3	31:0	**Subspan 2, Pixel 3 (lower right) Parameter 0**
M2.2	31:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M2.1	31:0	**Subspan 2, Pixel 1 (upper right) Parameter 0**
M2.0	31:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M3 – Mn		Repeat packets 1 and 2 to cover all required parameters

4.11.1.7 SIMD8 Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels.

DWord	Bit	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in section 4.11.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 – Mn		Repeat packet 1 to cover all required parameters

4.11.1.8 SIMD4x2 Payload

DWord	Bit	Description
M1.7	31:0	**Sample 1 Parameter 3** Specifies the value of the pixel's parameter 3. The actual parameter that maps to parameter 3 is given in the table in section 4.11.1.3. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Sample 1 Parameter 2**
M1.5	31:0	**Sample 1 Parameter 1**
M1.4	31:0	**Sample 1 Parameter 0**
M1.3	31:0	**Sample 0 Parameter 3**
M1.2	31:0	**Sample 0 Parameter 2**
M1.1	31:0	**Sample 0 Parameter 1**
M1.0	31:0	**Sample 0 Parameter 0**
M2		Parameters 4-7 if present
M3		Parameters 8-11 if present

4.11.1.9 SIMD32/64 Payload

4.11.1.9.1 Pixel Shader [DevCTG+]

[DevCTG+] only

This position of **Delta U/V** in the pixel shader payload layout is to allow the register delivered in the pixel shader dispatch containing the coefficients for the texture coordinates to be left in their original position (Delta U = Cxs, Delta V = Cyt). The values for U and V are computed in the pixel shader into the unused positions in this register.

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	**Pixel 0 V Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,2046])
M1.5	31:0	**Delta V**: defines the difference in V for adjacent pixels in the Y direction. **Programming Notes**: • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than 16 for the sample_8x8 message type. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space
M1.4	31:0	Ignored
M1.3	31:0	Ignored
M1.2	31:0	**Pixel 0 U Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,4095])
M1.1	31:0	**[DevILK+]: U 2nd Derivative** Defines the change in the delta U for adjacent pixels in the X direction. **Programming Notes**: • This field is ignored for messages other than sample_8x8. Format = IEEE_Float in normalized space **[Pre-DevILK]:** Ignored

DWord	Bit	Description
M1.0	31:0	**Delta U**: defines the difference in U for adjacent pixels in the X direction. **Programming Notes**: • **Delta U** multiplied by **Width** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space

4.11.1.9.2 Media [DevILK]

4.11.1.9.3 Media [DevILK]

[ILK] only

The position of **Delta U** and **U 2nd Derivative** in the media payload layout is intended to make media kernels more efficient. Sending a message using the media payload layout behaves identically to the pixel shader payload layout other than the position of these two fields.

DWord	Bit	Description
M1.6	31:0	**Pixel 0 V Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,2046])
M1.5	31:0	**Delta V**: defines the difference in V for adjacent pixels in the Y direction. **Programming Notes**: • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • **Delta V** multiplied by **Height** in SURFACE_STATE must be less than 16 for the sample_8x8 message type. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space
M1.2	31:0	**Pixel 0 U Address** Format: sample_unorm* and sample_8x8: IEEE_Float in normalized space deinterlace: U32 (Range: [0,4095])
M1.1	31:0	**Delta U**: defines the difference in U for adjacent pixels in the X direction. **Programming Notes**: • **Delta U** multiplied by **Width** in SURFACE_STATE must be less than or equal to 3 for sample_unorm* message types. • This field is ignored for the deinterlace message type. Format = IEEE_Float in normalized space

DWord	Bit	Description
M1.0	31:0	**U 2nd Derivative** Defines the change in the delta U for adjacent pixels in the X direction. **Programming Notes**: • This field is ignored for messages other than sample_8x8. Format = IEEE_Float in normalized space

4.11.2 Writeback Message

Corresponding to the four input message definitions are four writeback messages. Each input message generates a corresponding writeback message of the same type (SIMD16, SIMD8, SIMD4x2, or SIMD32/64).

4.11.2.1 SIMD16

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3. The pixels written within each destination register is determined by the execution mask on the "send" instruction.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1.7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
W1.6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
W1.5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
W1.4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
W1.3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
W1.2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
W1.1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
W1.0	31:0	**Supspan 2, Pixel 0 (upper left) Red**

DWord	Bit	Description
W2		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W3		**Subspans 3 and 2 of Green:** See W1 definition for pixel locations
W4		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W5		**Subspans 3 and 2 of Blue:** See W1 definition for pixel locations
W6		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W7		**Subspans 3 and 2 of Alpha:** See W1 definition for pixel locations

4.11.2.2 SIMD8

This writeback message consists of four registers, or five in the case of sample+killpix. As opposed to the SIMD16 writeback message, channels that are masked in the write channel mask are not skipped, all four channels are always returned. The masked channels, however, are not overwritten in the destination register.

For the sample+killpix message types, an additional register (W4) is included after the last channel register.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written) **:** W4 is only delivered for the sample+killpix message type
W4.0	31:16	**Dispatch Pixel Mask:** This field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread.
	15:0	**Active Pixel Mask:** This field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. Since the SIMD8 message applies to only 8 pixels, only the low 8 bits within this field are used. The high 8 bits are always set to 1. **[DevBW, DevCL]** Errata: Active Pixel Mask needs to be ORed with the inverse of the EMask before it is ANDed with the DMask. Also if the sample instruction is within a conditional then the active pixel mask will be overwritten with the partial mask on each different sample instruction so this will have to be done for each instance of the sample instruction not just as the end.

4.11.2.3 SIMD4x2

A SIMD4x2 writeback message always consists of a single message register containing all four channels of each of the two "pixels" (called "samples" here, as they are not really pixels) of data. The write channel mask bits as well as the execution mask on the "send" instruction are used to determine which of the channels in the destination register are overwritten. If any of the four execution mask bits for a sample is asserted, that sample is considered to be active. The active channels in the write channel mask will be written in the destination register for that sample. If the sample is inactive (all four execution mask bits deasserted), none of the channels for that sample will be written in the destination register.

DWord	Bit	Description
W0.7	31:0	**Sample 1 Alpha:** Specifies the value of the pixel's alpha channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer. Format depends on the **Data Return Format** programmed for the surface being sampled.
W0.6	31:0	**Sample 1 Blue**
W0.5	31:0	**Sample 1 Green**
W0.4	31:0	**Sample 1 Red**
W0.3	31:0	**Sample 0 Alpha**
W0.2	31:0	**Sample 0 Blue**
W0.1	31:0	**Sample 0 Green**
W0.0	31:0	**Sample 0 Red**

4.11.2.4 SIMD32/64

4.11.2.4.1 Sample_unorm* [Pre-DevIVB]

[DevILK+] only

Pixels are numbered as follows:

0	***1***	***2***	***3***	***4***	***5***	***6***	***7***
8	***9***	***10***	***11***	***12***	***13***	***14***	***15***
16	***17***	***18***	***19***	***20***	***21***	***22***	***23***
24	***25***	***26***	***27***	***28***	***29***	***30***	***31***

DWord	Bit	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 7 & 6 Red**

DWord	Bit	Description
W0.4		**Pixel 5 & 4 Red**
W0.3		**Pixel 11 & 10 Red**
W0.2		**Pixel 9 & 8 Red**
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1.7		**Pixel 31 & 30 Red**
W1.6		**Pixel 29 & 28 Red**
W1.5		**Pixel 23 & 22 Red**
W1.4		**Pixel 21 & 20 Red**
W1.3		**Pixel 27 & 26 Red**
W1.2		**Pixel 25 & 24 Red**
W1.1		**Pixel 19 & 18 Red**
W1.0		**Pixel 17 & 16 Red**
W2.7:0		**Pixels 15:0 Green**
W3.7:0		**Pixels 31:16 Green**
W4.7:0		**Pixels 15:0 Blue** W4-W7 are not sent for the _RG versions of the sample_unorm message
W5.7:0		**Pixels 31:16 Blue** W4-W7 are not sent for the _RG versions of the sample_unorm message
W6.7:0		**Pixels 15:0 Alpha** W2 and W3 are not sent for the _RG versions of the sample_unorm message
W7.7:0		**Pixels 31:16 Alpha** W4-W7 are not sent for the _RG versions of the sample_unorm message

For the sample_unorm_RG+killpix and sample_unorm+killpix messages, an additional writeback phase is returned. For sample_unorm_RG+killpix, "n" is equal to 4, for sample_unorm+killpix, "n" depends on which channels are enabled for return, this register will immediately follow the first part of the writeback message.

DWord	Bit	Description
Wn.7:1		Reserved (not written)
Wn.0	31:0	**Active Pixel Mask:** This field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. The bits in this mask correspond to the pixels as follows: 0 1 4 5 16 17 20 21 2 3 6 7 18 19 22 23 8 9 12 13 24 25 28 29 10 11 14 15 26 27 30 31

4.11.2.4.2 sample_8x8 [PreDevIVB]

[DevILK+] only

The writeback message for sample_8x8 consists of up to 16 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in all four destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel.

Pixels are numbered as follows:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

DWord	Bit	Description
W0.7	31:16	**Pixel 15 Red** Format = 16-bit UNORM with an 8-bit range (the value FF00h maps to a real value of 1.0) Range = [0000h:FF00h]
	15:0	**Pixel 14 Red**
W0.6		**Pixel 13 & 12 Red**
W0.5		**Pixel 7 & 6 Red**
W0.4		**Pixel 5 & 4 Red**
W0.3		**Pixel 11 & 10 Red**
W0.2		**Pixel 9 & 8 Red**

DWord	Bit	Description
W0.1		**Pixel 3 & 2 Red**
W0.0		**Pixel 1 & 0 Red**
W1.7		**Pixel 31 & 30 Red**
W1.6		**Pixel 29 & 28 Red**
W1.5		**Pixel 23 & 22 Red**
W1.4		**Pixel 21 & 20 Red**
W1.3		**Pixel 27 & 26 Red**
W1.2		**Pixel 25 & 24 Red**
W1.1		**Pixel 19 & 18 Red**
W1.0		**Pixel 17 & 16 Red**
W2.7:0		**Pixels 15:0 Green**
W3.7:0		**Pixels 31:16 Green**
W4.7:0		**Pixels 15:0 Blue**
W5.7:0		**Pixels 31:16 Blue**
W6.7:0		**Pixels 15:0 Alpha**
W7.7:0		**Pixels 31:16 Alpha**
W8.7:0		**Pixels 47:32 Red**
W9.7:0		**Pixels 63:33 Red**
W10.7:0		**Pixels 47:32 Green**
W11.7:0		**Pixels 63:33 Green**
W12.7:0		**Pixels 47:32 Blue**
W13.7:0		**Pixels 63:33 Blue**
W14.7:0		**Pixels 47:32 Alpha**
W15.7:0		**Pixels 63:33 Alpha**

STMM block definition:

DWord	Bit	Description
Wr.7	31:24	**STMM (14,3)** Format = U8
	23:16	**STMM (12,3)**
	15:8	**STMM (10,3)**
	7:0	**STMM (8,3)**
Wr.6	31:0	**STMM (6:0,3)**
Wr.5	31:0	**STMM (14:8,2)**
Wr.4	31:0	**STMM (6:0,2)**
Wr.3	31:0	**STMM (14:8,1)**

DWord	Bit	Description
Wr.2	31:0	**STMM (6:0,1)**
Wr.1	31:0	**STMM (14:8,0)**
Wr.0	31:0	**STMM (6:0,0)**

Block Noise Estimate/Denoise History block definition: [prior to Gen6]

DWord	Bit	Description
Wq.7	31:0	Reserved : MBZ
Wq.6	31:0	Reserved : MBZ
Wq.5	31:0	Reserved : MBZ
Wq.4	31:0	Reserved : MBZ
Wq.3	31:0	Reserved : MBZ
Wq.2	31:0	Reserved : MBZ
Wq.1	31:8	Reserved : MBZ
Wq.1	7:0	**Block Noise Estimate** Format = U8
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen6 DI enabled]

DWord	Bit	Description
Wq.7	31:16	Y[15:0] – Location of 16x4
Wq.7	15:0	X[15:0] - Location of 16x4
Wq.6	31:24	STAD0 - Sum in time of absolute differences for 4x4 Format = U8 [STAD values are 0 if DN is disabled]
Wq.6	23:16	STAD1
Wq.6	15:8	STAD2
Wq.6	7:0	STAD3 (Ignore when both DN & DI are enabled)
Wq.5	31:24	SHCM0 - Sum horizontally of absolute differences for 4x4 Format = U8 [SHCM values are 0 if DN is disabled]
Wq.5	23:16	SHCM1
Wq.5	15:8	SHCM2
Wq.5	7:0	SHCM3 (Ignore when both DN & DI are enabled)

DWord	Bit	Description
Wq.4	31:24	SVCM0 Sum Vertically of absolute differences for 4x4 Format = U8 [SVCM values are 0 if DN is disabled]
Wq.4	23:16	SVCM1
Wq.4	15:8	SVCM2
Wq.4	7:0	SVCM3 (Ignore when both DN & DI are enabled)
Wq.3	31:16	Diff_cTpT - difference in top fields of current and previous frame Format = U16
Wq.3	15:0	Diff_cBpB - difference in bottom field of current and previous frame
Wq.2	31:16	Diff_cTcB - difference between top and bottom field in current frame.
Wq.2	15:0	Diff_cTpB - difference between current top and previous bottom
Wq.1	31:16	Diff_cBpT - difference between current bottom and previous top.
Wq.1	15:8	Motion_Count - number of pixels that are moving (different above a threshold) Format = U8
Wq.1	7:0	Block Noise Estimate for 16x4 (Valid only if DN is enabled)
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen6 DI disabled]

DWord	Bit	Description
Wq.7	31:16	Y[15:0] – Location of 16x4
Wq.7	15:0	X[15:0] - Location of 16x4
Wq.6	31:24	STAD0 - Sum in time of absolute differences for 4x8 Format = U8
Wq.6	23:16	STAD1
Wq.6	15:8	STAD2
Wq.6	7:0	STAD3
Wq.5	31:24	SHCM0 - Sum horizontally of absolute difference for 4x8
Wq.5	23:16	SHCM1
Wq.5	15:8	SHCM2
Wq.5	7:0	SHCM3

DWord	Bit	Description
Wq.4	31:24	SVCM0 Sum Vertically of absolute difference for 4x8
Wq.4	23:16	SVCM1
Wq.4	15:8	SVCM2
Wq.4	7:0	SVCM3
Wq.3	31:16	Reserved
Wq.3	15:0	Reserved
Wq.2	31:8	Reserved
Wq.2	7:0	Block Noise Estimate for 16x8
Wq.1	31:24	**Denoise History** for 4x4 at X = 15 to 12, Y = 7 to 4 Format = U8
Wq.1	23:16	**Denoise History** for 4x4 at X = 11 to 8, Y = 7 to 4
Wq.1	15:8	**Denoise History** for 4x4 at X = 7 to 4, Y = 7 to 4
Wq.1	7:0	**Denoise History** for 4x4 at X = 15 to 12, Y = 3 to 0
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen7 +] DI Enabled

DWord	Bit	Description
Wq.7	31:16	Y[15:0]
Wq.7	15:0	X[15:0]
Wq.6	31:16	STAD - Sum in time of absolute differences for 16x4 – value is 0 if DN disabled. Format = U16
Wq.6	15:0	SHCM - Sum horizontaly of absolute differences – value is 0 if DN is disabled. Format = U16
Wq.5	31:16	SVCM - Sum vertically of absolute differences – value is 0 if DN is disabled.. Format = U16
Wq.5	15:0	Diff_cTpT - sum of differences in top fields of current and previous frame Format = U16
Wq.4	31:16	Diff_cBpB - sum of differences in bottom field of current and previous frame Format = U16

DWord	Bit	Description
Wq.4	15:0	Diff_cTcB -sum of differences between top and bottom field in current frame. Format = U16
Wq.3	31:16	Diff_cTpB - sum of differences between current top and previous bottom Format = U16
Wq.3	15:0	Diff_cBpT - sum of differences between current bottom and previous top. Format = U16
Wq.2	31:0	Reserved
Wq.1	31:24	Tearing_Count - number of pixels that have (diff_cTcB > diff_cTcT + diff_cBcB) Format = U8
Wq.1	23:16	Fitting_Count - number of pixels that have (diff_cTcB<=diff_cTcT + diff_cBcB) Format = U8
Wq.1	15:8	Motion_Count - number of pixels that are moving (different above a threshold) Format = U8
Wq.1	7:0	Block Noise Estimate Format = U8
Wq.0	31:24	**Denoise History** for 4x4 at Y = 15 to 12, X = 3 to 0 Format = U8
Wq.0	23:16	**Denoise History** for 4x4 at Y = 11 to 8, X = 3 to 0
Wq.0	15:8	**Denoise History** for 4x4 at Y = 7 to 4, X = 3 to 0
Wq.0	7:0	**Denoise History** for 4x4 at Y = 3 to 0, X = 3 to 0

Block Noise Estimate/Denoise History block definition: [Gen7+] DI Disabled:

DWord	Bit	Description
Wq.7	31:16	Y[15:0]
Wq.7	15:0	X[15:0]
Wq.6	31:16	STAD - Sum in time of absolute differences for top 16x4 Format = U16
Wq.6	15:0	SHCM - Sum horizontaly of absolute differences for top 16x4 Format = U16
Wq.5	31:16	SVCM - Sum vertically of absolute differences for top 16x4 Format = U16
Wq.5	15:0	STAD - Sum in time of absolute differences for bottom 16x4 Format = U16

DWord	Bit	Description
Wq.4	31:16	SHCM - Sum horizontaly of absolute differences for bottom 16x4 Format = U16
Wq.4	15:0	SVCM - Sum vertically of absolute differences for bottom 16x4 Format = U16
Wq.3	31:0	Reserved
Wq.2	31:8	Reserved
Wq.2	7:0	Block Noise Estimate Format = U8
Wq.1	31:24	Denoise History for 4x4 at X = 15 to 12, Y = 7 to 4 Format = U8
Wq.1	23:16	Denoise History for 4x4 at X = 11 to 8, Y = 7 to 4
Wq.1	15:8	Denoise History for 4x4 at X = 7 to 4, Y = 7 to 4
Wq.1	7:0	Denoise History for 4x4 at X = 3 to 0, Y = 7 to 4
Wq.0	31:24	Denoise History for 4x4 at X = 15 to 12, Y = 3 to 0 Format = U8
Wq.0	23:16	Denoise History for 4x4 at X = 11 to 8, Y = 3 to 0
Wq.0	15:8	Denoise History for 4x4 at X = 7 to 4, Y = 3 to 0
Wq.0	7:0	Denoise History for 4x4 at X = 3 to 0, Y = 3 to 0

DI Enabled (Only)

This writeback message is returned when the DI Enable field in SAMPLER_STATE is enabled. The response length possibilities are:

- pre-Gen6 & DN Enabled: 12
- pre-Gen6 & DN Disabled: 9
- Gen6 & DN Enabled: 12
- Gen6 & DN Disabled: 10
- Gen7 & DN Enabled & surface_format == 4:2:2 packed: 12
- Gen7 & DN Enabled & surface_format != 4:2:2 packed: 11
- Gen7 & DN Disabled: 10

DWord	Bit	Description
W0		**Previous 2nd Field Deinterlaced Luma for Y=0,1** Refer to Luma block above for definition.

DWord	Bit	Description
W1		**Previous 2nd Field Deinterlaced Luma for Y=2,3**
W2		**Previous 2nd Field Deinterlaced Chroma for Y=0,1** Refer to Chroma block above for definition.
W3		**Previous 2nd Field Deinterlaced Chroma for Y=2,3**
W4		**Current 1st Field Deinterlaced Luma for Y=0,1**
W5		**Current 1st Field Deinterlaced Luma for Y=2,3**
W6		**Current 1st Field Deinterlaced Chroma for Y=0,1**
W7		**Current 1st Field Deinterlaced Chroma for Y=2,3**
W8		**STMM** Refer to STMM block above for definition.
W9		**Block Noise Estimate/Denoise History** Refer to Block Noise Estimate/Denoise History block above for definition. This register is only included if **DN Enable** is enabled for pre-Gen6. It is always included for Gen6+.
W10		**Current 2nd Field Luma for 16x2** This register is only included if **DN Enable** is enabled.
W11		Current 2nd Field Chroma This register is only included if DN Enable is enabled.

The denoised luma for both the current 1st and 2nd field needs to be written out, but only the 2nd field has a dedicated location. This is because the denoised data for the 1st field is in the deinterlaced output for the 1st field – Y=0 and Y=2 are the denoised data, while Y=1 and Y=3 either the deinterlaced lines or copied from the previous or current frame if progressive.

DI Disabled

This writeback message is returned when the **DI Enable** field in SAMPLER_STATE is disabled. The DN with DI disabled responses with a 16x8 block rather than a 16x4 with a response length of 9 for a 4:2:2 input format, or 5 for other formats. Two denoised luma and chroma fields are combined into an interleaved top/bottom format.

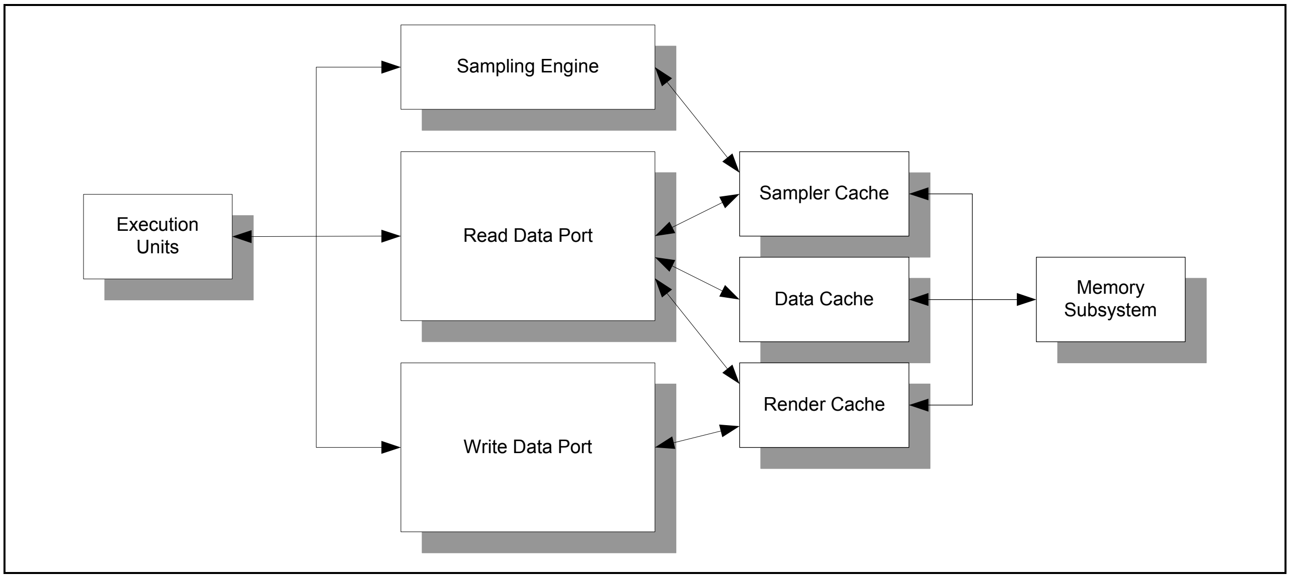
DWord	Bit	Description
W0		**Luma for Y=0 & 1** Refer to Luma block above for definition.
W1		**Luma for Y=2 & 3** Refer to Luma block above for definition, but add 2 to Y to get location
W2		**Luma for Y=4 & 5**
W3		**Luma for Y=6 & 7**

DWord	Bit	Description
W4		**Block Noise Estimate/Denoise History** Refer to Block Noise Estimate/Denoise History block above for definition.
W5		**Chroma for Y=0 & 1** Only sent if input surface format is 4:2:2
W6		**Chroma for Y=2 & 3** Only sent if input surface format is 4:2:2
W7		**Chroma for Y=4 & 5** Only sent if input surface format is 4:2:2
W8		**Chroma for Y=6 & 7** Only sent if input surface format is 4:2:2

5. Data Port

The Data Port provides all memory accesses for the Gen4 subsystem other than those provided by the sampling engine. These include render target writes, constant buffer reads, scratch space reads/writes, and media surface accesses.

[Pre-DevSNB]: The diagram below shows the two parts of the Data Port (Read and Write) and how they connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.



The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The read and write data ports are considered to be separate shared functions, each with its own shared function identifier.

5.1 Cache Agents

The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The three data ports are considered to be separate shared functions, each with its own shared function identifier.

5.2 Cache Agents

The data port allows access to memory via various caches. The choice of which cache to use for a given application is dictated by its restrictions, coherency issues, and how heavily that cache is used for other purposes.

[Pre-DevSNB]: The cache to use is selected by the **Target Cache** field of the read data port message descriptor. The write data port message descriptor does not have an equivalent field as it only supports writes to the render cache.

5.2.1 Render Cache

[Pre-DevIVB]: The render cache is the only cache that supports both reads and writes. All writes must use this cache. In addition, all reads to a surface that is also being written should use this cache to avoid expensive flushing that would be required for coherency. The render cache supports both linear and tiled memory.

The render cache is intended to be used for the following surfaces:

- 3D render target surfaces
- destination surfaces for media applications
- intermediate working surfaces for media applications
- scratch space buffers
- streamed vertex buffers

5.2.2 Data Cache [Pre-DevGT]

The data cache is a small, read-only cache that supports only linear memory. For 3D graphics, it is intended to be used only for constant buffers. For media and other generic applications, it may be used to load kernel constants such as filter coefficients as well as other linear data buffers such as compressed data buffer for HWMC.

In the hardware implementation on all of these devices, the data cache does not exist as a separate physical cache. It is mapped in hardware to the sampler cache.

5.3 Surfaces

The data elements accessed by the data port are called “surfaces”. There are two models used by the data port to access these surfaces: surface state model and stateless model.

5.3.1 Surface State Model

The data port uses the binding table to bind indices to surface state, using the same mechanism used by the sampling engine. The surface state model is used when a **Binding Table Index** (specified in the message descriptor) of less than 255 is specified. In this model, the **Binding Table Index** is used to index into the binding table, and the binding table entry contains a pointer to the SURFACE_STATE. SURFACE_STATE contains the parameters defining the surface to be accessed, including its location, format, and size.

This model is intended to be used for constant buffers, render target surfaces, and media surfaces.

5.3.2 Stateless Model

The stateless model is used when a **Binding Table Index** (specified in the message descriptor) of 255 is specified. In this model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = R32G32B32A32_FLOAT
- Vertical Line Stride = 0
- Surface Base Address = **General State Base Address** + **Immediate Base Address**
- Buffer Size = checked only against **General State Access Upper Bound**
- Surface Pitch = 16 bytes
- Utilize Fence = false
- Tiled = false

This model is primarily intended to be used for scratch space buffers.

5.4 Read/Write Ordering

[Pre-DevSNB]: Hardware does not guarantee ordering between read and write messages issued to the data port, even between messages issued by the same thread. If ordering is important, software must guarantee ordering. For a write followed by a read to the same location, the write must use a write commit, and wait for the write commit to return before issuing the read message. For a read followed by a write to the same location, software must wait for the read data to be returned before issuing the write message.

5.5 Accessing Buffers

There are four data port messages used to access buffers. Three of these are used for both constant buffers and scratch space buffers, the fourth is used by the geometry shader kernel to write to streamed vertex buffers. All of these messages support only buffers, and can use the surface state model as well as the stateless model.

The following table indicates the intended applications of each of the buffer messages.

Message	Applications
OWord Block Read/Write	• constant buffer reads of a single constant or multiple contiguous constants • scratch space reads/writes where the index for each pixel/vertex is the same • block constant reads, scratch memory reads/writes for media
OWord Dual Block Read/Write	• SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access) • SIMD4x2 scratch space reads/writes where the indices are different.
DWord Scattered Read/Write	• SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message) • SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message) • general purpose DWord scatter/gathering, used by media
Streamed Vertex Buffer Write	• geometry shader streaming vertex data out

These messages generally ignore the surface format field of the state and perform no format conversion. The exception is the Streamed Vertex Buffer Write, which uses the surface format field to determine only how many channels are to be written. The data contained in each channel is still not converted in any way.

5.6 Accessing Media Surfaces

The Media Block Read/Write message is intended to be used to access 2D media surfaces. The message specifies an X/Y coordinate into the 2D surface as input. Since this message only supports 2D surfaces, the stateless model cannot be used with this message.

5.6.1.1 Skin Tone Detection/Enhancement (STD/E)

The STD/E unit, composed of the Skin Tone Detection (STD) and Skin Tone Enhancement (STE) units, is part of color processing pipe located at the Render Cache Pixel Backend (RCBP).

The main goal of the STD/E is to reproduce the skin colors in a way that is more palatable to the observer, and by that to increase the sensed image quality. It may also pass indication of skin tones to the TCC and ACE.

The STD unit detects the skin like colors and passes a grade of skin tone color to the STE. The STE modify the saturation and Hue of the pixel. Both the STD and STE are per-pixel basis. The input pixels are required to be on the YUV space.

The skin tone detected factor will be recorded as a 5-bit number and it will be passed to the module of ACE and TCC to indicate the strength of skin tone likelihood.

5.6.1.1.1 STD

The STD operates on digital images in the YUV color space. In these space the skin-tone region is represented by the ellipse in the (U,V) subspace (chroma components), by a trapeze membership function in the Y direction (luma component) and by a piece-wise linear classifier in the (V,Y) subspace.

U,V data is transformed into Hue and Saturation space through shifting and rotation

Step 1: shift rectangle

```
U_center =  U - U_mid
V_center =  V - V_mid
```

Step 2: rotate rectangle

```
Sat = -(U_center * Cos - V_center * Sin)
Hue = -(U_center * Sin + V_center * Cos)
```

Where: Sin = ***Sin(α)***, and Cos = ***Cos(α).***

<u>Rectangle skin-tone measure determination</u>

Skin-tone detection is described by a continue score on the [0,1] range, where a level 0 means not a skin (SkinToneFactor = 0) , and a level 1 (SkinToneFactor = 1) means a full skin. In between, (0,1) region, we have partial skin-tone detection. This partial skin-tone detection is controlled by a margin parameter, which will be denoted by "***HS_margin***". The SkinToneFactor is expressed by 5 bits, and thus have values in the [0,31] range.

```
if( abs(Sat) <= SatMax && abs(Hue) <= HueMax)
  {
   if(HS_margin >= 5)
       {
         Sat_Factor = (Sat_max-abs(Sat)) / 2^(HS_margin - 5);
         Hue_Factor = (Hue_max-abs(Hue)) / 2^(HS_margin - 5);
       }
   else
       {
         Sat_Factor = (Sat_max-abs(Sat)) * 2^(HS_margin - 5);
         Hue_Factor = (Hue_max-abs(Hue)) * 2^(HS_margin - 5);
       } //end of if(HS_margin >= 5)
 }
else
{
         Sat_Factor = 0;
         Hue_Factor = 0;
} //end of if( abs(Sat) <= SatMax && abs(Hue) <= HueMax)


Sat_Factor = min(Sat_Factor,31);
Hue_Factor = min(Hue_Factor,31);


Rectagle_SkinToneFactor = min(Sat_Factor, Hue_Factor);
```

Rhombus skin tone detection determination

Similar to the rectangle skin-tone measure, a rhombus-margin (***Diamond_margin***) is introduced. This introduces a new rhombus region, inner to the original rhombus, in a similar happened with the rectangle. There are two regions such that: outside the original rhombus a SkinToneFactor = 0 (not a skin); inside the inner rhombus SkinToneFactor = 1 (full skin); in between 0 < SkinToneFactor < 1 indicating a partial skin-tone detection. As in the rectangle case, the SkinToneFactor is expressed by 5 bits, and thus have values in the [0,31] range.

A Diamond SkinToneFactor calculations algorithm is:

```
Dist = abs(Sat - Diamond_du) + Diamond_alpha(1/tan(β)) * abs(Hue - Diamond_dv);


                    //outside the diamond

if(Dist >= Diamond_TH)

{

  D_Factor = 0; //the point is out of the large rhombus

}

else if(Dist < (Diamond_TH - Diamond_margin))

  {

    D_Factor = 31; //the point is inside the inner rhombus

  }

   else  //the point is inbetween the outer and the inner rhombuses

   {

    if(Diamond_margin >= 5)

     {

      D_Factor = (Diamond_TH - Dist) / 2^(Diamond_margin - 5);

     }

     else

     {

      D_Factor = (Diamond_TH - Dist) * 2^(Diamond_margin - 5);

     } // end of if(Diamond_margin >= 5)

   } // if(D < (Diamond_TH - Diamond_margin))
```

```
Diamond_SkinToneFactor  = D_factor;
```

Finally the level of the skin-tone detection in the (U,V) subspace is given by:

UV_SkinToneFactor = min(Rectangle_SkinToneFactor, Diamond_SkinToneFactor);

Detection in Y direction

The detection based on the Y-values, is given by a piece-wise linear membership function, which is defined with 4 points ***(Y_point_x***) (x=1, 2, 3, and 4).

```
if(Y >= Y_Point_0 && in_Y < Point_1)
   Y_Factor = (Y - Y_Point_0) * Y_Slope_1;
else if(Y >= Point_1 && Y < Point_2)
        Y_Factor = 31;
     else if(Y >= Point_2  && Y < Point_3)
             Y_Factor = (Point_3 - Y) * Y_Slope_2 ;
          else
             Y_Factor = 0;
```

At the end of the process a double (min,max) clipping is applied:

```
Y_Factor = min(31,max(Y_Factor,0));
```

The final Skin-Tone detection is is given by:

```
SkinToneFactor = min(UV_SkinToneFactor, Y_factor);
```

Detection in the VY plane (3D-like DTD)

The operation of the detection in VY plane is particularly enabled by ***VY_STD_Enable*** bit

It is known that the application of a three-dimensional (3D) classifier in the (Y,U,V) space, instead of a two dimensional (2D) skin-tone detector in the (U,V) plane, is resulted in a better detection. Implementation complexity of the full 3D classifier is too high, and forces us to approximate the classifier by more simple, but useful methods. Skin-tone data distribution implies (it is almost convex, and has a predominate directions) that the 3D classifier could be approximated by the intersection of the three 2D classifiers in (U,V), (U,Y), and (V,Y) subspaces. The (U,V) subspace is the most important one it is already approximated by the ellipse, as was described previously. Our study implies that the (V,Y) subspace is the next most important one. Although the (U,Y) space carries the STD information, it is heavily redundant and has the reduced importance.

Thus the approximation of 3D classifier is an intersection of (U,V) and (V,Y) two-dimensional classifiers. The (V,Y) classifier is given by two piece-wise linear functions (PWLF), Each PWLF is composed of four straight segments. Each segment is described by the three parameters (Point, Slope and bias). Thus a single PWLF (lower or upper) is described by 12 parameters (4 points, 4 biases, 4 slopes).

The parameters of lower part are: 4 point ***PxL*** (x=0, 1, 2, 3), 4 bias ***BxL*** (x=0, 1, 2, 3) and 4 slope ***SxL*** (x=0, 1, 2, 3).

The parameters of upper part are: 4 point ***PxU*** (x=0, 1, 2, 3), 4 bias ***BxU*** (x=0, 1, 2, 3) and 4 slope ***SxU*** (x=0, 1, 2, 3).

There is Programming Restrictions to specify the parameters

- The points must be in the non-decreasing order: P0 <= P1 <= P2 <= P3.
- The parts must be continues on they ends. Thus the user:

  (a). must set: $P0_L = P0_U$ (continuity at the leftmost points).

  (b). must care for continuity at the rightmost points.

Margin for the detection in the VY plane (3D-like DTD)

Vertical margins of each part were introduced to provide a “soft” continuous detection over the classifier boundaries. There are two parameters defined

MarginVYL - the margin of the lower (blue) part.

MarginVYU - the margin of the upper (red) part.

Consider a pixel with coordinates $(Y,V) = (P2_L,V_1)$,. This pixel has a Y coordinate exactly as of the point $P2$, and a V coordinate equal V_1. For this pixel the detection relative to the Lower Part will be:

$$det_L = Max(Min((V_1 - B2_L) / MarginVYL, 0), 1)$$

The identical calculations are made for the Upper Line as well:

$$det_U = Max(Min((V_U - V_1) / MarginVYU, 0), 1)$$

Where:

det_L - is a detection relative to the Lower Part

det_U - is a detection relative to the Upper Part

V_U - is a V value of the Upper PWLF correspond to the $Y=P2_L$

B_U - is a V value of the Lower PWLF correspond to the $Y=P2_L$

The inverse operation of (1/ MarginVYL), and (1/ MarginVYU) is specified by the parameters INV***_MARGIN_VYL*** and ***INV_Margin_VYU***.

Both detections (det_L, det_U) are reduced to 5 bit representations, and the overal detection in the (V,Y)-plane is given by:

$$det\_VY = min(det_L, det_U)$$

The final Skin-Tone Detection is given by the minimum of the previously calculated STD in the (U,V)-plane (9), and the current one:

$$SkinToneFactor = min(SkinToneFactor, det\_VY)$$

This value is represented with 5 bits, and has a [0,31] range.

5.6.1.1.2 STE

The enhancement step is performed on the pixels which were detected as the skin-tone pixels only by the previous (STD) step. This step is divided into two sub-steps: saturation correction enhancemen and hue correction enhancement

STE – Saturation Correction Enhancement

The enhancement is performed by the transformation $Sat_{New} = F_{Sat}(Sat_{Old})$, which is realized by the piece-wise linear function (PWLF) with a 4 straight segments.

The parameters of this PWLF are:

Points:

SATP0 = -SatMax

SATPx (x=1,2,3) – defined by the user

SATP4 = SatMax

Biases:

SATB0 = -SatMax

SATBx (x=1,2,3) – defined by the user

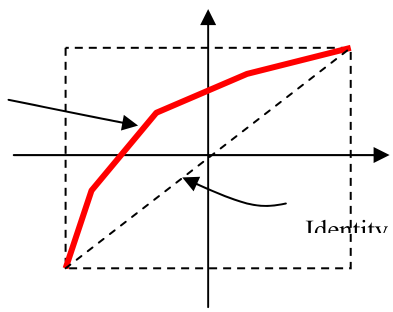
SATB4 = SatMax

Slopes:

SATSx (x=0,1,2,3) – defined by the user

There is Programming Restrictions to specify the parameters

- The point Sat = $-Sat_{Max}$ maps to itself: $(-Sat_{Max}) \rightarrow (-Sat_{Max})$.
- The point Sat = Sat_{Max} maps to itself: $(Sat_{Max}) \rightarrow (Sat_{Max})$.
- The correction function is continuous.
- The correction function is non-decreasing.



STE – Hue Correction Enhancement

The enhancement is performed by the transformation $Hue_{New} = F_{Sat}(Hue_{Old})$, which is realized by the piece-wise linear function (PWLF) with a 4 straight segments.

The parameters of this PWLF are:

Points:

HUEP0 = -HueMax

HUEPx (x=1,2,3) – defined by the user

HUEP4 = HueMax

Biases:

HUEB0 = -HueMax

HUEBx (x=1,2,3) – defined by the user

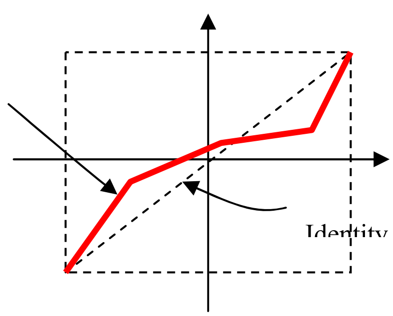
HUEB4 = HueMax

Slopes:

HUESx (x=0,1,2,3) – defined by the user

There is Programming Restrictions to specify the parameters

- The point Hue = -HueUE$_{Max}$ maps to itself: (-Hue$_{Max}$) → (-Hue$_{Max}$).
- The point Hue = Hue$_{Max}$ maps to itself: (Hue$_{Max}$) → (Hue$_{Max}$).
- The correction function is continuous.
- The correction function is non-decreasing.



STE – Skin Type Correction Enhancement

The operation of this mode is enabled by the control parameter ***Skin_types_enable***.

The Saturation and Hue enhancement processes are basic STE procedure. The advanced mode to adjust the enhacement based on the skin type define the second set of the Sat and the Hue enhancement parameters, which has an identical structure as the previous one (Points, Biases, Slopes) but having different values. We will refer one set of parameters to the Bright skin (Bs), and the other to the Dark skin (Ds). Each pixel is referred as belongs to the Bright, the Dark, or to the both skin types with a different membership values. The Dark/Bright skin classifier is defined by the two parameters: ***Skin_types_thesh***, and ***Skin_types_margin***. It works on the luma (Y) values.

The parameters related are

Points:

HUEPx_DARK (x=1,2,3) – defined by the user

SATPx_DARK (x=1,2,3) – defined by the user

Biases:

HUEBx_DARK (x=1,2,3) – defined by the user

SATBx_DARK (x=1,2,3) – defined by the user

Slopes:

HUESx_DARK (x=0,1,2,3) – defined by the user

SATSx_DARK (x=0,1,2,3) – defined by the user

For the luma value Y, we define

$$Y_A = \text{skinTypesThesh} - \text{skinTypesMargin}$$

$$Y_B = \text{skinTypesThesh} + \text{skinTypesMargin}$$

$$MV_{Dark} = \begin{cases} 1, & \text{if } Y < Y_A \\ 0, & \text{if } Y > Y_B \\ (Y_B - Y) / (2 * \text{skinTypesMargin}), & \text{if } Y_A <= Y <= Y_B \end{cases}$$

$$MV_{Bright} = 1 - mV_{Dark}$$

Where MV_{Dark} and MV_{Bright} are the membership value of the Dark and Bright skin (belongnes). *(Note: the membership values represent the "belongness" of the skin pixel to the different skin types).* Also, we mark that the inversee operation of 1/(2* Skin_types_margin) wil be specified by the parameter ***INV_skin_type_margin***.

In previous sections the procedure for the calculation of the Sat_{New} and Hue_{New} values was described. We calculate these values for the two skin types and thus get $Sat_{New\ B}$, $Hue_{New\ B}$, and $Sat_{New\ D}$, $Hue_{New\ D}$ values , where and subscribes "B" and "D" stands for the Bright and the Dark skin types, respectively. (In this case, the parameters with "_DARK" extension are used to work out $Sat_{New\ D}$ and $Hue_{New\ D}$, and the other set of the parameter could be reloaded with the parameters to work out $Sat_{New\ D}$, $Hue_{New\ D.}$)The final values of the enhanced pixel will be given by:

$$Sat_{New} = MV_{Dark} * Sat_{New\ D} + MV_{Bright} * Sat_{New\ B}$$

$$Hue_{New} = MV_{Dark} * Hue_{New\ D} + MV_{Bright} * Hue_{New\ B}$$

STE – (Sat, Hue) to (U, V) transformation

In prior session,, the (U,V) → (Sat,Hue) transformation was proceeded by the two steps: ***shift,*** and ***rotation***. Thus the backward transformation should be done in the inverse order: a ***rotation***, and then a ***shift***.

```
// Rotate back:
U_Center_New =  (Sat_New * Cos) + (Hue_New * Sin)
V_Center_New = -(Sat_New * Sin) + (Hue_New * Cos)

// Shift:
            U_New =  U_Center_New + U_mid
            V_New =  V_Center_New + V_mid
```

The (U_new, V_new) are the (Sat_{New}, Hue_{New}) values in transformed to the original (U,V) coordinates.

Let denote the original (U,V) values of the pixel by (U_in,V_in). Thus the difference between the corrected and the original values are:

DU = U_new – U_in

DV = V_new – V_in

The final correction must be depended by the *SkinToneFactor* value, and therefore DU, DV are corrected by:

DU = DU * STD_ Likelihood_Factor

DV = DV * STD_ Likelihood_Factor

Where:

$$STD\_Likelihood\_Factor = (SkinToneFactor / 32)$$

(Remember that the 0 <= SkinToneFactor <= 31).

After the DU and DV were corrected by the STD likelihood factor, the final (U,V) will be calculated by:

$$U = U\_in + DU$$

$$V = V\_in + DV$$

5.6.1.2 Adaptive Contrast Enhancement (ACE)

The Automatic Contrast Enhancement (ACE) is a part of the color processing pipe, which located at the render cache in the RCPB block.

The main goals of the ACE is to improve the overall contrast of the image, and emphasizing details when relevant (such as in dark areas).

The ACE algorithm analyzes the image, and consequently changes contrast of the image according to its characteristics. It works in YCbCr color space, where analysis and changes are performed over the Y component. The result of ACE is a 1d (1 dimension) look up table (1D LUT) operating on Y. The ACE follows the skin tone enhancement module in the pipe.

The ACE is receiving skin information from the STD block. When the frame includes skin the affect of the ACE is reduced in the skin area.

The ACE operation is divided into three stages:

1. Collecting information on Y and building the picture histogram. (Hardware)
2. Analysis on the collected data. (Software/Kernel)
3. Modification of the Y component. (Hardware)

The major steps of ACE can be divided into the following steps and depict in the below diagram.

1. Histogram calculation of the Y values.
2. Limiting extremely large histogram’s bins.
3. Calculate the Image’s gray level mean value (Ymean).
4. Calculate the Image’s “Dark Factor” by the Ymean and external transfer function.

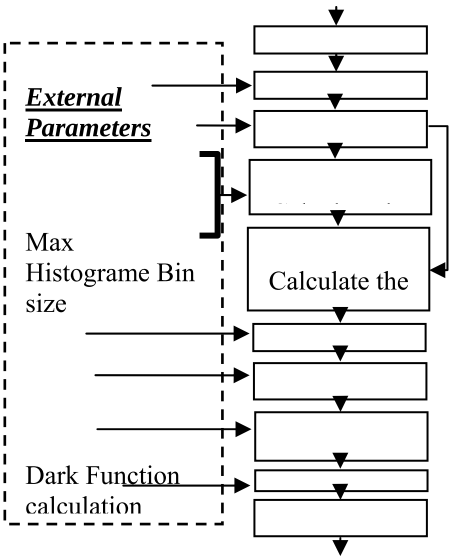
5. Find the PWLF anchor input and output points according to the "Portion Values" and the "Destination Points" of the Bright and the Dark images.
6. Find the PWLF anchor Input points by the blending of the Dark and Bright anchor input points, according to the Dark Factor calculated previously.
7. Find the PWLF anchor Output points by the blending of the Dark and Bright anchor output points, according to the Dark Factor calculated previously.
8. Limit Slopes between the anchor points. This stage's output is the current's image ACE PWLF.
9. "Soften" the ACE PWLF by blending I with the Identity Transformation.
10. Blend the current PWLF with the PWLF of the previous image (History blend).
11. Apply the final PWLF, and get the Yout values.

Note: Step 1 & step 11 are done in HW and steps 2-10 are done in software.

The main ACE goals are overall contrast improvement, and details emphasizing. ACE algorithm generates a Piece-wise Linear Function (PWLF), and the final gray values, Yout, are calculated by Yout = PWLF(Yin).

The HW compares the input pixels to the ***skin_threshold*** to determine if the target pixel is a skin pixel or not. It operates on all of the input pixels if the ***Full_image_histogram*** flag is defined. (to ignore the AOI flag). HW output the histogram of luma pixel value to VSC, and at VSC, the maximum and minimum value of luma pixels (Ymax, Ymin) ans the number of skin pixels is determined to be made available to the software development via MMIO register.

An eleven-segment (12 points) was established to implement PWLF via the state parameters (Points: ***Ymin, Y1-Y10, Ymax***, Bias: ***B1 – B10***, Slope: ***S0-S10***).


External Parameters
Max Histograme Bin size
Dark Function calculation
Calculate the

5.6.1.3 Total Color Control (TCC)

The TCC allows users to choose different grades of saturation for each of the six basic colors (Red, Green, Blue, Magenta, Yellow and Cyan) in order to custom the color scheme. The TCC algorithm operates on the UV-color components in the YUV color space. It operates in the pixel-wise mode, without considering any neighborhood information.

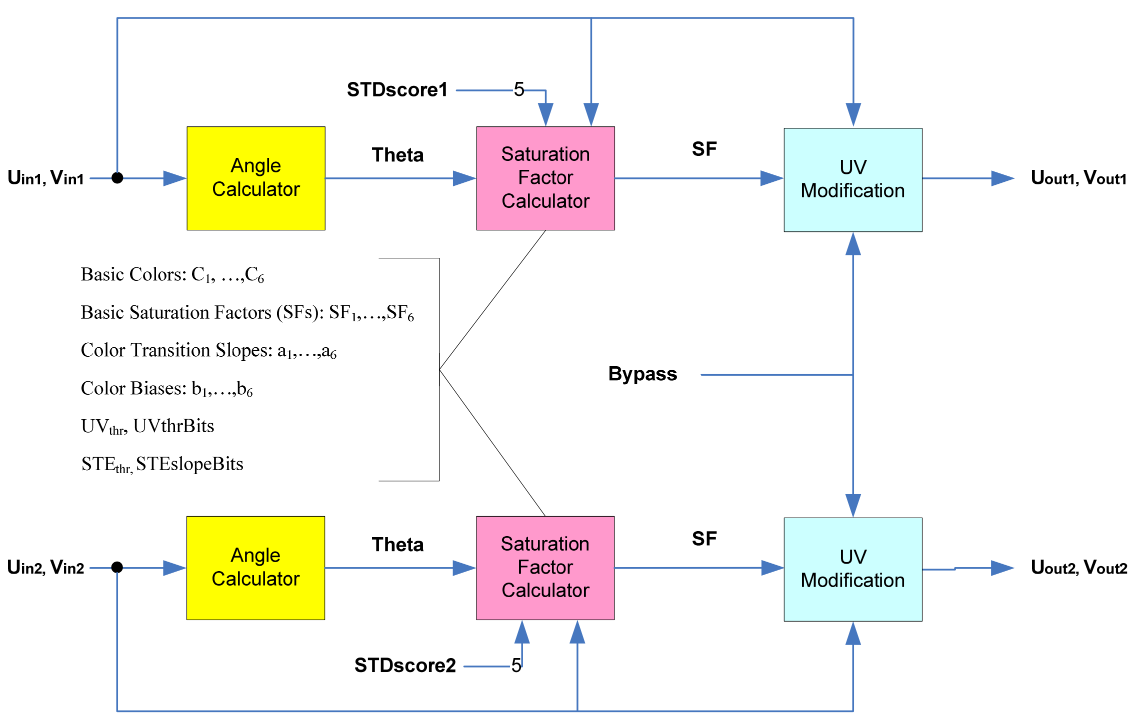
Its input is:

1. U,V color components (10 bit)
2. Skin-tone detection value (5 bit)
3. External control parameters

Its output is the new U, V values (10 bit).

The motivation to implement this block in HW is to reduce the power of the system and therefore the battery life.

The pixel TPT (throughput) is two pixels per clock. The pipeline works in YUV formats only – 10bit pixels. The TCC block is control by state only and does not require any memory access. The TCC block runs at the same frequency of the existing RCPBunit.



There are two paths in parallel to support the requirement of two pixels per clock. Valid out is a signal which high when the pixels are valid.

The TCC block includes three sub blocks.

Angle_calculator

This block receive pixel U and V and perform division of abs|v| by abs|u| using Divider ROM with pipeline.

The division result is used to calculated arctan of the V/U. This result is used to calculate the angle called θ, by using approximation equation. This angle is defined as a 10bit.

To simplify this calculation the "arctangent" function is approximated in the $[0,45]^o$ region by the second order polynomial:

$$\theta = \arctan(x) = -0.2880x^2 + 1.0797x - 0.005; \quad (0 <= x <= 1)$$

The resulted θ is given in radians with the maximal error of 0.005 rad. (0.286 deg.) This approximation is calculated by the minimizing the mean squared error (mse) between the actual "arctan" function, and its polynomial approximation, and thus represents the optimal mse-approximation in the [0,π/4] region. The θ for the all regions is calculated by:

$$\theta = \begin{cases} \theta_{0.25\pi}; & \text{for region I, } (0 <= x <= 1), \\ \pi/2 - \theta_{0.25\pi}; & \text{for region II, } (1 < (V/U) < \text{infinity}) \\ \pi/2 + \theta_{0.25\pi}; & \text{for region III, } (-\text{infinity} < (V/U) < -1) \\ \pi - \theta_{0.25\pi}; & \text{for region IV, } (-1 <= (V/U) < 0) \\ \pi + \theta_{0.25\pi}; & \text{for region V, } (0 <= (V/U) <= 1) \\ 3\pi/2 - \theta_{0.25\pi}; & \text{for region VI, } (1 < (V/U) < \text{infinity}) \\ 3\pi/2 + \theta_{0.25\pi}; & \text{for region VII, } (-\text{infinity} < (V/U) < -1) \\ 2\pi - \theta_{0.25\pi}; & \text{for region VIII, } (-1 <= (V/U) < 0) \end{cases}$$

Whereas x = (V/U), and the $\theta_{0.25\pi}$ is given by the above equation.

Saturation_Factor_Calculator

This block is using the angle θ, locate where it is in the color wheel, find the appropriate base colors and calculate the proportional distance from the adjacent base color. The result called α . Alpha (α) represent the distance from the two relevant base color.

Calculate the saturation by using the appropriate user parameters. The result is the Saturation factor. This block considering also the threshold and the maximum UV values, and considering also correction for gray colors to minimize the possible noise. In addition the saturation skipping doing saturation when the color is skin and doing alpha blending according the skin factor called STDscore.

This block requires several external parameters such:

BaseColor1,…, BaseColor6 – Six basic user defined colors.

SatFactor1,…, SatFactor6 – Six basic saturation change user defined factors.

ColorTransitSlope12, …..ColorTransit61 – Six calculation result of 1/(BaseColorX – BaseColorY)

ColorBias1,…, ColorBias6 – Six color bias.

STDscore – Skin-tone Detection score (from STD/E).

The result of SF is a number of 8bits.

There are four major steps to derive the saturation factor.

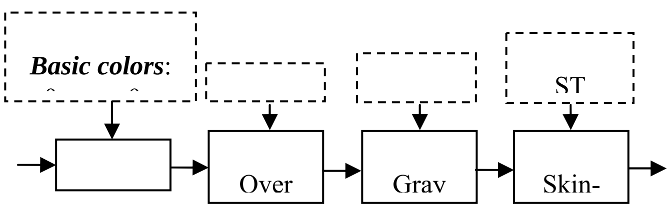


Figure. Calculation of the Saturation Factor (SF).

<u>The Interpolated Basic SFs_1</u>

With the calculated angle θ, which lies in the $[\theta_{Ci}, \theta_{Ci+1}]$ interval, the Interpolated Basic SFs_1 will be:

$SFs_1 = (1-\alpha)\, \textbf{\textit{SatFactor}}_{\boldsymbol{i}} + \alpha\, \textbf{\textit{SatFactor}}_{\boldsymbol{i+1}}$

Whereas α is calculated by:

$$\alpha = \text{Min}\{\text{Max}[(\theta - \textbf{\textit{BaseColor}}_{\boldsymbol{i}}) * \textbf{\textit{ColorTransitSlope}}_{\boldsymbol{i}} - \textbf{\textit{ColorBias}}_{\boldsymbol{i}}, 0], 1\}$$

Over Saturation Limiter SFs2

Over Saturation Limiter block is used to avoid saturation boosting of the already high saturated pixels. The SFs_2 is calculated by:

$$SFs_2 = \begin{cases} SFs_1, & \textit{for } (SF_1 <= 1) \\ 1 + (SFs_1 - 1)(\text{MaxColor} - UV_{max})/\text{MaxColor}, & \textit{for } (1 < SF_1 <= 2) \textit{ AND } (UV_{max} <= \textbf{\textit{UVMaxColor}}) \\ 1, & \textit{for } (UV_{max} > \textbf{\textit{UVMaxColor}}) \end{cases}$$

Where the $UV_{max} = \max(|U|,|V|)$, and ***UVMaxColor*** is an external parameter which in the case of YUV color space is equal to 448 in 10bit representation. ***The Inv_UVMaxColor*** was used for the inverse calculation of 1/UVMaxColor.

Note: *The last condition (UV_{max} > **UVMaxColor**) is associated with the illegal colors, and usually hasn't to appea (Can this be OK for wide gamut mapping?).*

GrayPixels Saturation LimiterSFs3

This block limits the saturation of the almost gray pixels. The reason for this limiter is to prevent the noise amplification by the Saturation increase process. The result of this block is:

$$SFs_3 = 1 + dSF * CLF$$

Where:

$$dSF = SFs_2 - 1;$$

And the CLF is called Color Limiting Factor and ranges from 0 to 1. The calculation of the CLF is given by:

$$CLF \begin{cases} = 1; & \textit{for } (SFs_2 <= 1) \textit{ AND } (\textit{any } UV_{max}) \\ = 0; & \textit{for } (UV_{max} <= \textbf{\textit{UV\_Threshold}}) \\ = (UV_{max} - \textbf{\textit{UV\_Threshold}}) / 2^{\textbf{\textit{UV\_Threshold\_Bits}}}; & \textit{for } (\textbf{\textit{UV\_Threshold}} < UV_{max} < (\textbf{\textit{UV\_Threshold}} + 2^{\textbf{\textit{UV\_Threshold\_Bits}}})) \end{cases}$$

Skin-tone Saturation LimiterSFs4

The last block effects TCC strength operation of the Skin-tone pixels. Uncontrolled enhancement of the skin pixels could lead to appearing of artifacts and to undesired results. The final SFs_4 is calculated by a linear blending:

$$SFs_4 = (128*STE_{factor} + (256 - STE_{factor})\ SFs_3\ ) / 256$$

Where the STE_{factor} is called Skin Tone factor and is calculated by:

$$diff = (STD_{score} - \textbf{\textit{STE\_Threshold}}) * 2^3$$

*Note: the STD_{score} (from STD) and the **STE_Threshold** are presented with 5 bits. The multiplication by 2^3 is in order to raise the "diff" to 8 bits.*

$$STE_{factor} = Min\ \{Max\ [(diff * 2^{STE\_SlopeBits}), 0], 255\}$$

The STD_{score} is a result of the Skin-tone Detection module. It is represented with 5 bits, where the values 0 and 31 mean no skin-tone, and full skin-tone detection, respectively. The STE_{factor} is given by 8 bits, where the value 256 represents the number 1.

It is evident that for the high values of STE_{factor} the resulted SFs_4 is close to 1, which means a weak TCC action of this pixel (SFs_4 = 1 actually means TCC is off).

UV Modification – The input pixels are multiple by the saturation factor. The results are the output pixels.

SF_{final} is the final saturation factor which actually resulted from the forth SFcalculation block:

$$SF_{final} = SFs_4$$

The calculation of the U_{new}, and V_{new} output values. They are calculated below:

$$U_{new} = U * SF_{final}$$

$$V_{new} = V * SF_{final}$$

Whereas (U,V) are the original input color components,

Because these pixels are represented in the unbiased form, which is the result of substraction of the value 512 from the original [U,V] values, the final [U_{out}, V_{out}] values are given by:

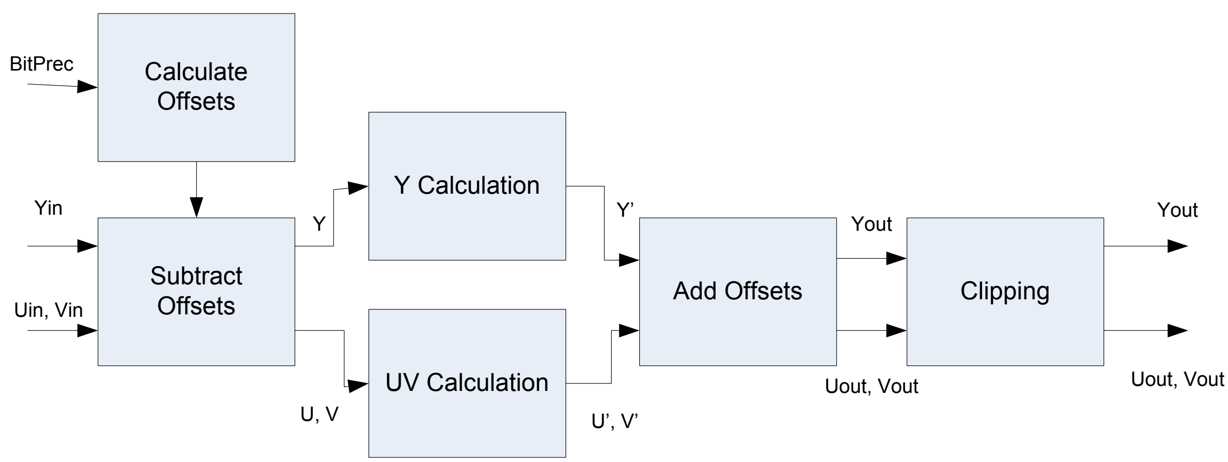
$$U_{out} = U_{new} + 512$$

$$V_{out} = V_{new} + 512$$

This is the final TCC output represented with 10 bits.

5.6.1.4 ProcAmp

The PROCAMP block modifies the brightness, contrast, hue and saturation of an image in YCbCr color space (or similar).



The algorithm itself uses 8-16 bits per color.

Y Processing: 256 is subtracted from the Y values to position the black level at zero. This removes the DC offset so that adjusting the contrast does not vary the black level. Since Y values may be less than 256, negative Y values should be supported at this point. Contrast is adjusted by multiplying the YUV pixel values by a constant. If U and V are adjusted, a color shift will result whenever the contrast is changed. The brightness property value is added (or subtracted) from the contrast adjusted Y values; this is done to avoid introducing a DC offset due to adjusting the contrast. Finally the value 64 is added to reposition the black level at 256. The equation for processing of Y values is:

Y' = ((Y-256) x C) + B + 256,

where C is the ***Contrast*** value and B is the ***Brightness*** value.

UV Processing: 2048 is first subtracted from both U and V values to position the range around zero. The hue property is implemented by mixing the U and V values together:

U' = (U-2048) x Cos(H) + (V-2048) x Sin(H)

V' = (V-2048) x Cos(H) – (U-2048) x Sin(H)

Where H represents the desired Hue angle; Saturation is adjusted by multiplying both U and V by a constant. Finally, the value 2048 is added to both U and V. The combined processing of Hue and Saturation on the UV data is:

U' = (((U-2048) x Cos(H) + (V-2048) x Sin(H)) x C x S) + 2048

V' = (((V-2048) x Cos(H) - (U-2048) x Sin(H)) x C x S) + 2048

Where C is the contrast, H is Hue angle and S is the Saturation and the combination of Cos(H)*C*S and Sin(H)*C*S is specified by parameters ***Cos_c_s*** and ***Sin_c_s.***

5.6.1.5 Color Space Conversion

The CSC block enables linear conversion between color spaces using vector shift, matrix multiplication, and additional shift.

The CSC algorithm is a linear coordinate transformation, comprising of the following stages:

- Shifting the input color coordinate.
- Multiply by 3*3 matrix
- Shifting the output color coordinate
- Formula representation of last 3 steps:

$$\begin{pmatrix} \text{vout\_1} \\ \text{vout\_2} \\ \text{vout\_3} \end{pmatrix} = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} * \begin{pmatrix} \text{vin\_1+v0\_1} \\ \text{vin\_2+v0\_2} \\ \text{vin\_3+v0\_3} \end{pmatrix} + \begin{pmatrix} \text{u0\_1} \\ \text{u0\_2} \\ \text{u0\_3} \end{pmatrix}$$

Where is

aij are the matrix elements, i.e., the transform coefficients: ***C0, C1, C2, C3, C4, C5, C6, C7, C8.***

vin_i is the input pixel color components

v0_i is the input offset vector, i.e., ***Offset_in_1***, ***Offset_in_2, Offset_in_3.***

u0_1_i is the output offset vector. i.e., ***Offset_out_1, Offset_out_2, Offset_out_3.***

Clipping the output to ensure each component is in allowed range.

The parameters ***YUV_IN*** is used to set input to be RGB format and ***YUV_OUT*** is uased to set output to be RGB format

Notes about Repacker:

There are two states to be used in the repacker: ***Alpha from State Select*** and ***color pipe alpha***. The last module in the IECP pipeline.

If ***Alpha from State Select*** is set, the Y, U ,V is packed with the information from ***color pipe alpha***, and then the data is sent out to RCPB.

Otherwise, “0” is inserted in the 4LSB (alpha) and the packed data is sent out to RCPB.

5.7 Accessing Render Targets

Render targets are the surfaces that the final results of pixel shaders are written to. The render targets support a large set of surface formats (refer to surface formats table in *Sampling Engine* for details) with hardware conversion from the format delivered by the thread. The render target message also causes numerous side effects, including potentially alpha test, depth test, stencil test, alpha blend (which normally causes a read of the render target), and other functions. These functions are covered in the *Windower* chapter as some of them (depth/stencil test) are also partially done in the Windower.

The render target write messages are specifically for the use of pixel shader threads that are spawned by the windower, and may not be used by any other threads. This is due to the pixel scoreboard side-effects that sending of this message entails. The pixel scoreboard ensures that incorrect ordering of reads and writes to the same pixel does not occur.

5.7.1 Single Source

The "normal" render target messages are single source. There are two forms, SIMD16 and SIMD8, intended for the equivalent-sized pixel shader threads. A single color (4 channels) is delivered for each of the 16 or 8 pixels in the message payload. Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

The pixel scoreboard bits corresponding to the dispatched pixel mask (or half of the mask in the case of SIMD8 messages) are cleared only if the **Last Render Target Select** bit is set in the message descriptor.

5.7.2 Dual Source [DevCL-B, DevCTG+]

Note: Dual Source messages are not supported in DevBW and DevCL-A devices.

The dual source render target messages only have SIMD8 forms due to maximum message length limitations. SIMD16 pixel shaders must send two of these messages to cover all of the pixels. Each message contains two colors (4 channels each) for each pixel in the message payload. In addition to the first source, the second source can be selected as a blend factor (BLENDFACTOR_*_SRC1_* options in the blend factor fields of COLOR_CALC_STATE or BLEND_STATE). Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

Each dual source message delivered will clear the corresponding pixel scoreboard bits if the **Last Render Target Select** bit in the message descriptor is set.

[Pre-DevSNB]: It is UNDEFINED to utilize a DualSource RT Write message when **Color Buffer Blend Enable** is DISABLED.

5.7.3 Replicate Data

The replicate data render target message is intended to be used for "fast clear" functionality in cases where the color data for each pixel is identical. This message performs better than the other messages due to its smaller message length. This message does not support depth, stencil, or antialias alpha data being sent with it. This message must target only tiled memory. Access of linear memory using this message type is UNDEFINED. The depth buffer can be cleared through the "early depth" function in conjunction with a pixel shader using this message. Refer to the *Windower* chapter for more details on the early depth function.

The pixel scoreboard bits corresponding to the dispatched pixel mask are cleared only if the **Last Render Target Select** bit is set in the message descriptor.

5.7.4 Multiple Render Targets (MRT)

Multiple render targets are supported with the single source and replicate data messages. Each render target is accessed with a separate Render Target Write message, each with a different surface indicated (different binding table index). The depth buffer is written only by the message(s) to the last render target, indicated by the **Last Render Target Select** bit set to clear the pixel scoreboard bits.

5.8 Flushing the Render Cache [Pre-DevSNB]

5.9 State

5.9.1 BINDING_TABLE_STATE

The data port uses the binding table to retrieve surface state. Refer to *Sampling Engine* for the definition of this state.

5.9.2 SURFACE_STATE

The data port uses the surface state for constant buffers, render targets, and media surfaces. Refer to *Sampling Engine* for the definition of this state.

5.10 Messages

5.10.1 Global Definitions

For data port messages, part of the message descriptor is used to determine the message type. This field is documented here. The remainder of the message descriptor is defined differently depending on the message type, and is documented in the section for the corresponding message.

[Pre-DevSNB]: The Data Port is actually two separate targets, **Data Port Read** and **Data Port Write**, each with its own target unit ID. Each target has its own set of message type encodings as shown below.

Restrictions:

- **[DevBW-A,B,C0, DevCL-A0] Errata:** use of any Sampling Engine message in the same workload (between pipeline flushes) with any Data Port read messages utilizing the Sampler Cache is not allowed.
- Data port messages may not have the **End of Thread** bit set in the message descriptor other than the following exeptions:
    - The Render Target Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in non-contiguous dispatch mode.
    - The Render Target UNORM Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in contiguous dispatch mode.

5.10.2 Data Port Messages

Most of the messages have an existing definition that is not expected to change. There are several new messages that are documented here.

Data Cache Data Port Message Summary

Message Type	Header Required	Shared Local Memory Support	Stateless Support	Address Modes	Vector Width
OWord Block Read	yes	no	yes	global	1
OWord Block Write	yes	no	yes	global	1
Unaligned OWord Block Read	yes	no	yes	global	1
OWord Dual Block Read	no	no	yes	global + offset	2
OWord Dual Block Write	no	no	yes	global + offset	2
DWord Scattered Read	no	no	yes	global + offset	8, 16
DWord Scattered Write	no	no	yes	global + offset	8, 16
Byte Scattered Read	no	yes	no	global + offset	8, 16
Byte Scattered Write	no	yes	no	global + offset	8, 16
Untyped Surface Read	no	yes	no	1D or 2D	2, 8, 16
Untyped Surface Write	no	yes	no	1D or 2D	8, 16
Untyped Atomic Operation	no	yes	no	1D or 2D	8, 16
Scratch Block Read	yes	no	yes (only)	Imm_Buf + offset	
Scratch Block Write	yes	no	yes (only)	Imm_Buf + offset	
Memory Fence	yes	N/A	N/A	N/A	N/A

"global" is the **Global Offset** in the message header (if header is not present, Global Offset is zero).
"imm_buf" is the Immediate Buffer Base Address provided in message header register M0.5.
"offset" is in the message payload, and is per-slot.
"handle" is the handle address in the message header.
"URBoffset" is the **Global Offset** field in the URB message descriptor.
"1D" and "2D" are the address payload.

Render Cache Data Port Message Summary

Message Type	Header Required	Address Modes	Vector Width
Media Block Read	yes	2D	1
Media Block Write	yes	2D	1
Render Target Write	no	2D + RTAI	8, 16
Typed Surface Read	yes	1D, 2D, 3D, 4D	8
Typed Surface Write	yes	1D, 2D, 3D, 4D	8
Typed Atomic Operation	yes	1D, 2D, 3D, 4D	8
Memory Fence	yes	N/A	N/A

"4D" address refers to U/V/R/LOD for mip-mapped surfaces
"2D + RTAI" address refers to a basic 2D address with render target array index for the third dimension

5.10.2.1 Message Descriptor

5.10.2.1.1 [DevBW] and [DevCL]

The following message descriptor definition applies to [DevBW] and [DevCL].

<table>
<tr><th>Bit</th><th colspan="3">Description</th></tr>
<tr><th colspan="2">DATA PORT READ TARGET</th><th colspan="2">DATA PORT WRITE TARGET</th></tr>
<tr><td>15:14</td><td>Target Cache
00: Data Cache
01: Render Cache
10: Sampler Cache
11: Reserved</td><td>15</td><td>Send Write Commit Message. Indicates that a write commit message will be sent back to the thread when the write has been committed. See section Error! Reference source not found. for more details.
Format = Enable</td></tr>
<tr><td>13:12</td><td>Read Message Type
00: OWord Block Read
01: OWord Dual Block Read
10: Media Block Read
11: DWord Scattered Read</td><td>14:12</td><td>Write Message Type
000: OWord Block Write
001: OWord Dual Block Write
010: Media Block Write
011: DWord Scattered Write
100: Render Target Write
101: Streamed Vertex Buffer Write
111: Flush Render Cache
All other encodings are reserved.</td></tr>
<tr><td>11:8</td><td colspan="3">Message Specific Control. Refer to the specific message section for the definition of these bits.</td></tr>
</table>

5.10.2.1.2 [DevILK]

The following message descriptor definition applies to [DevILK].

<table>
<tr><th>Bit</th><th colspan="3">Description</th></tr>
<tr><td>19</td><td colspan="3">Header Present
This bit must be set to one for all Data Port messages.</td></tr>
<tr><td>18:16</td><td colspan="3">Ignored</td></tr>
<tr><td colspan="2">DATA PORT READ TARGET</td><td colspan="2">DATA PORT WRITE TARGET</td></tr>
<tr><td>15:14</td><td>Target Cache
00: Data Cache
01: Render Cache
10: Sampler Cache
11: Sampler Cache Field Mode (This mode indicates that the Sample Cache is allocated with field cache lines. This mode is only allowed if the resulting Vertical Line Stride, from surface state or being overridden by this message, is 1. Thus, it can only be used for Media Block Read message from Sampler Cache.)</td><td>15</td><td>Send Write Commit Message. Indicates that a write commit message will be sent back to the thread when the write has been committed. See section Error! Reference source not found. for more details.
Format = Enable</td></tr>
<tr><td>13:11</td><td>Read Message Type
000: OWord Block Read
010: OWord Dual Block Read
100: Media Block Read
110: DWord Scattered Read
001: Render Target UNORM Read
011: AVC Loop Filter Read
All other encodings are reserved.</td><td>14:12</td><td>Write Message Type
000: OWord Block Write
001: OWord Dual Block Write
010: Media Block Write
011: DWord Scattered Write
100: Render Target Write
101: Streamed Vertex Buffer Write
110: Render Target UNORM Write
111: Flush Render Cache</td></tr>
<tr><td>10:8</td><td>Message Specific Control. Refer to the specific message section for the definition of these bits.</td><td>11:8</td><td>Message Specific Control. Refer to the specific message section for the definition of these bits.</td></tr>
<tr><td>7:0</td><td colspan="3">Binding Table Index. Specifies the index into the binding table for the specified surface. A binding table index of 255 indicates that a stateless model is to be used. Refer to section 5.3.2 for details on the stateless model.
[ILK] BindingTableIndex[3:0] cannot be "0000" for any Data Port Transactions when GS Enable bit is set in 3DSTATE_PIPELINED_POINTERS and GS Pass Through Enable in GS_STATE is cleared.
Format = U8
Range = [0,255]</td></tr>
</table>

5.10.2.2 Message Header

This header applies to the following data port messages:

- OWord Block Read/Write
- Unaligned OWord Block Read
- OWord Dual Block Read/Write
- DWord Scattered Read/Write

The header definitions for the other data port messages is in the section for each message.

DWord	Bit	Description
M0.5	31:10	**Immediate Buffer Base Address.** Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Ignored
	7:0	**Dispatch ID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	**Global Offset.** **[Pre-DevSNB]:** Specifies the global byte offset into the buffer. • For the OWord messages, this offset must be OWord aligned (bits 3:0 MBZ) • For the DWord messages, this offset must be DWord aligned (bits 1:0 MBZ) Format = U32 Range = [0,FFFFFFF0h] for OWord messages Range = [0,FFFFFFFCh] for DWord messages
M0.1	31:0	Ignored
M0.0	31:0	Ignored

5.10.2.3 Write Commit Writeback Message

The writeback message is only sent on Data Port Write messages if the **Send Write Commit Message** bit in the message descriptor is set. The destination register is not modified. Write messages without the **Send Write Commit Message** bit set will not return anything to the thread (response length is 0 and destination register is null).

DWord	Bit	Description
W0.7:0		Reserved

5.10.3 OWord Block Read/Write

This message takes one offset (Global Offset), and reads or writes 1, 2, 4, or 8 contiguous OWords starting at that offset.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be OWord aligned
- the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
- the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

- constant buffer reads of a single constant or multiple contiguous constants
- scratch space reads/writes where the index for each pixel/vertex is the same
- block constant reads, scratch memory reads/writes for media

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the first and third GRF registers returned (W0, W2) for read, or the first and third write registers sent (M1, M3). The high 8 bits are used similarly for the second and fourth (W1, W3 or M2, M4). For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

For the 1-OWord messages, only the low 8 bits of the execution mask are used. Either the low 4 bits or the high 4 bits, depending on the position of the OWord to be read or written, is used as the single group of four with behavior following that in the preceding paragraph. **[DevBW,DevCL] errata:** Execution mask bits outside of those corresponding to the OWord being read/written cannot be asserted.

The above behavior enables a SIMD16 thread to use the 8-OWord form of this message to access two channels (red and green) of a single scratch register across 16 pixels. A second message would access the other two channels (blue and alpha). The execution mask is used to ensure that data associated with inactive pixels are not overwritten.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

5.10.3.1 Message Descriptor

Bit	Description
13	**Invalidate After Read Enable** Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
12	Ignored (**[Pre-DevGT]:** this bit is part of the **Message Type** fields)
11	Ignored (**[DevCTG]:** this bit is part of the **Read Message Type** field for the read version of this message)
10:8	**Block Size.** Specifies the number of contiguous OWords to be read or written 000: 1 OWord, read into or written from the low 128 bits of the destination register 001: 1 OWord, read into or written from the high 128 bits of the destination register 010: 2 OWords 011: 4 OWords 100: 8 OWords all other encodings are reserved. **Programming Notes:** • The 6 OWord block size is valid only with **Data Port Constant Cache**.

5.10.3.2 Message Payload (Write)

For the write operation, the message payload consists of one, two, or four registers (not including the header) depending on the **Block Size** specified in the message. For the one-constant case, data is taken from either the high or low half of the payload register depending on the half selected in **Block Size**. In this case, the other half of the payload register is ignored.

The **Offset** referred to below is the **Global Offset** and is in units of OWords (discard low 4 bits for [Pre-DevGT]). The **OWord** array index is also in units of OWords.

DWord	Bit	Description
M1.7:4	127:0	**OWord[Offset + 1].** If the block size is 1 OWord to be written from the high 128 bits of the destination, OWord[Offset] will appear in this location
M1.3:0	127:0	**OWord[Offset]**
M2.7:4	127:0	**OWord[Offset+3]**
M2.3:0	127:0	**OWord[Offset+2]**
M3.7:4	127:0	**OWord[Offset+5]**
M3.3:0	127:0	**OWord[Offset+4]**
M4.7:4	127:0	**OWord[Offset+7]**
M4.3:0	127:0	**OWord[Offset+6]**

5.10.3.3 Writeback Message (Read)

For the read operation, the writeback message consists of one, two, three, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Offset** referred to below is the **Global Offset** and is in units of OWords (discard low 4 bits for [Pre-DevGT]). The **OWord** array index is also in units of OWords.

DWord	Bit	Description
W0.7:4	127:0	**OWord[Offset + 1].** If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord[Offset] will appear in this location
W0.3:0	127:0	**OWord[Offset]**
W1.7:4	127:0	**OWord[Offset+3]**
W1.3:0	127:0	**OWord[Offset+2]**
W2.7:4	127:0	**OWord[Offset+5]**
W2.3:0	127:0	**OWord[Offset+4]**
W3.7:4	127:0	**OWord[Offset+7]**
W3.3:0	127:0	**OWord[Offset+6]**

of the surface return 0.

5.10.3.4 Message Descriptor

Bit	Description
12:11	Ignored
10:8	**Block Size.** Specifies the number of contiguous OWords to be read 000: 1 OWord, read into the low 128 bits of the destination register 001: 1 OWord, read into the high 128 bits of the destination register 010: 2 OWords 011: 4 OWords 100: 8 OWords all other encodings are reserved.

5.10.3.5 Writeback Message (Read)

For the read operation, the writeback message consists of one, two, or four registers depending on the **Block Size** specified in the message. For the one-constant case, data is placed in either the high or low half of the returned register depending on the half selected in **Block Size**. In this case, the other half of the register is not changed.

The **Global Offset** is in units of **Bytes**, aligned to **DWord** (two LSBs set to zero). The **OWordX** array in units of OWord starts at Global Offset.

DWord	Bit	Description
W0.7:4	127:0	**OWord1 = *(&OWord0 + 1).** If the block size is 1 OWord to be loaded into the high 128 bits of the destination, OWord0 will appear in this location
W0.3:0	127:0	**OWord0 = Buffer[Global Offset]**
W1.7:4	127:0	**OWord3 = *(&OWord2 + 1)**
W1.3:0	127:0	**OWord2 = *(&OWord1 + 1)**
W2.7:4	127:0	**OWord5= *(&OWord4 + 1)**
W2.3:0	127:0	**OWord4 = *(&OWord3 + 1)**
W3.7:4	127:0	**OWord7 = *(&OWord6 + 1)**
W3.3:0	127:0	**OWord6 = *(&OWord5 + 1)**

5.10.4 OWord Dual Block Read/Write

This message takes two offsets, and reads or writes 1 or 4 contiguous OWords starting at each offset. The Global Offset is added to each of the specific offsets.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be OWord aligned
- the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
- the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

- SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access)
- SIMD4x2 scratch space reads/writes where the indices are different

Execution Mask. The low 8 bits of the execution mask are used to enable the 8 channels in the GRF registers returned for read, or each of the write registers sent. For reads, any mask bit asserted within a group of four will cause the entire OWord to be read and returned to the destination GRF register. For writes, each mask bit is considered for its corresponding DWord written to the destination surface.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

5.10.4.1 Message Descriptor

Bit	Description
13	**Invalidate After Read Enable** Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
12	Ignored (**[Pre-DevGT]:** this bit is part of the **Message Type** fields)
11:10	Ignored (**[DevCTG]:** bit 11 is part of the **Read Message Type** field for the read version of this message)
9:8	**Block Size:** Specifies the number of OWords in each block to be read or written 00: 1 OWord 10: 4 OWords all other encodings are reserved.

5.10.4.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored
M1.4	31:0	**Block Offset 1.** **[Pre-DevSNB]:** Specifies the byte offset of OWord Block 1 into the surface. Must be OWord aligned (bits 3:0 MBZ). Format = U32 Range = [0,FFFFFFF0h]
M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	**Block Offset 0**

5.10.4.3 Additional Message Payload (Write)

For the write operation, the message payload consists of one or four registers (not including the header or the first part of the payload) depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of OWords (discard low 4 bits for [Pre-DevSNB]). The **OWord** array index is also in units of OWords.

DWord	Bit	Description
M2.7:4	127:0	**OWord[Offset1]**
M2.3:0	127:0	**OWord[Offset0]**
M3.7:4	127:0	**OWord[Offset1+1]**

DWord	Bit	Description
M3.3:0	127:0	**OWord[Offset0+1]**
M4.7:4	127:0	**OWord[Offset1+2]**
M4.3:0	127:0	**OWord[Offset0+2]**
M4.7:4	127:0	**OWord[Offset1+3]**
M4.3:0	127:0	**OWord[Offset0+3]**

5.10.4.4 Writeback Message (Read)

For the read operation, the writeback message consists of one or four registers depending on the **Block Size** specified in the message.

The **Offset1/0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1/0** and is in units of OWords (discard low 4 bits for [Pre-DevSNB]). The **OWord** array index is also in units of OWords.

DWord	Bit	Description
W0.7:4	127:0	**OWord[Offset1]**
W0.3:0	127:0	**OWord[Offset0]**
W1.7:4	127:0	**OWord[Offset1+1]**
W1.3:0	127:0	**OWord[Offset0+1]**
W2.7:4	127:0	**OWord[Offset1+2]**
W2.3:0	127:0	**OWord[Offset0+2]**
W3.7:4	127:0	**OWord[Offset1+3]**
W3.3:0	127:0	**OWord[Offset0+3]**

5.10.5 Media Block Read/Write

The read form of this message enables a rectangular block of data samples to be read from the source surface and written into the GRF. The write form enables data from the GRF to be written to a rectangular block.

Restrictions:

- the only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Because of this, the stateless surface model is not supported with this message.
- the surface format is used to determine the pixel structure for boundary clamp, the raw data from the surface is returned to the thread without any format conversion nor filtering operation
- the target cache cannot be the data cache
- the surface base address must be 32-byte aligned
- When a surface is XMajor tiled, (**tile walk** field in the surface state is set to TILEWALK_XMAJOR), a memory area mapped through the Render Cache cannot be read and/or wrote in mixed frame and field modes. For example, if a memory location is first written with a zero Vertical Line Stride (frame mode), and later on (without render cache flush) read back using Vertical Line Stride of one (field mode), the read data stored in GRF are uncertain.

- The block width and offset should be aligned to the size of pixels stored in the surface. For a surface with 8bpp pixels for example, the block width and offset can be byte aligned. For a surface with 16bpp pixels, it is word aligned.
  - For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. dword aligned).
- The write form of message has the additional restriction that both **X Offset** and **Block Width** must be DWord aligned.
- **[DevBW, DevCL]** The read form of message also has the additional restriction that both **X Offset** and **Block Width** must be DWord aligned.
- **[DevBW-A] Erratum BWT001**: Surfaces being *read* with this message by the render cache must be tiled. Writes to linear surfaces are allowed.
- **[DevBW-A] Erratum**: A memory area mapped through the Render Cache cannot be read and/or wrote in mixed frame and field modes.
- When Color Processing is enabled for media write message. Render target must be tiled.

Applications:

- Block reads/writes for media

Execution Mask. The execution mask on the send instruction for this type of message is ignored. The data that is read or written is determined completely by the block parameters.

Out-of-Bounds Accesses. Reads outside of the surface results in the address being clamped to the nearest edge of the surface and the pixel in the position being returned. Writes outside of the surface are dropped and will not modify memory contents.

Determining the boundary pixel value depends on the surface format. Surface format definitions can be found in the Surface Formats Section of the Sampling Engine Chapter.

- For a surface with 8bpp pixels, the boundary byte is replicated. For example, for a boundary dword B0B1B2B3, to replicate the left boundary byte pixel, the out of bound dwords have the format of B0B0B0B0, and that for right boundary is B3B3B3B3.
  - This rule applies to all surface formats with BPE of 8. As the data port does not perform format conversion, the most likely used surface formats are R8_UINT and R8_SINT.
- For any other surfaces with 16bpp pixels, boundary pixel replication is on words. For example, for a boundary dword B0B1B2B3, to replicate the left boundary word pixel, the out of bound dwords have the format of B0B1B0B1, and that for right boundary is B2B3B2B3.
  - This rule applies to all surface formats with BPE of 16. As the data port does not perform format conversion, only the formats with integer data types may be useful in practice.
- For special surfaces with 16bpp pixels YUV422 packed format, there are two basic cases depending on the Y location: YUYV (surface format YCRCB_NORMAL) and UYVY (surface format YCRCB_SWAPY). Boundary handling for YVYU (surface format YCRCB_SWAPUV) is the same as that for YUYV. Similarly, boundary handling for VYUY (surface format YCRCB_SWAPUVY) is the same as that for UYVY. Note that these four surface formats have 16bpp pixels, even though the BPE fields are set to zero according to the table in the Surface Formats Section.
  - For a boundary dword Y0U0Y1V0, to replicate the left boundary, we get Y0U0**Y0**V0, and to replicate the right boundary, we get **Y1**U0Y1V0.
  - For a boundary dword U0Y0V0Y1, to replicate the left boundary, we get U0Y0V0**Y0**, and to replicate the right boundary, we get U0**Y1**V0Y1.
- For a surface with 32bpp pixels, the boundary dword pixel is replicated.

- This rule applies to all surface formats with BPE of 32. As the data port does not perform format conversion, some of the formats may not be useful in practice.

Hardware behavior for any other surface types is undefined.

When Color Processing Enable is set to 1 and the IECP output surface to be written is NV12 format (R16_UNORM surface format 0x10A, should be used if the output surface is NV12 format).

1. **NV12 surface state : The width of the surface should be always multiples of 4pixels. For 16bpp input message (422 8-bit) the width will always need to be in multiples of 8bytes and for 32bpp input message (422 16-bit or 444 8-bit) the width should be in multiples of 16bytes. Height should be in multiples of 2pixel high. (presently the MFX restriction is that width should be in multiples of 2pixels).**

    a. **y-offset of the media block write from the EU should be always even**

    b. **x-offset of the media block write from the EU should be in multiples of 4 pixel.**

2. **The media block dword write can have only the following combinations (for IECP when NV12 output format is used):**

    a. **8pixel wide for 422 8-bit mode**

    b. **4pixel wide for 422 8-bit mode**

    c. **4pixel wide for 422 16-bit**

    d. **4pixel wide for 444 8-bit.**

    e. **444 16-bit input format cannot be supported when the output format is NV12 (s/w should not use this combination).**

    f. **It has to be in multiples of 2pixel high for all above modes.**

3. **If 444-format is used then we use only the pixel_0 UV values of the 2x2 pixel and the rest are dropped and in case of 422-format the top UV values are used and the bottom UV values is dropped if the output format is NV12 format.**

4. **Assuming IECP messages will always have vertical stride = 0. (since this is only for pre-processing before the encoder).**

5.10.5.1 Message Descriptor

<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>13</td><td>Reserved: MBZ</td></tr>
<tr><td>12</td><td>Reserved : MBZ

[Pre-DevSNB]: this bit is part of the Message Type fields</td></tr>
<tr><td>11</td><td>Reserved : MBZ

[DevCTG,ILK]: this bit is part of the Read Message Type field for the read version of this message</td></tr>
<tr><td>10</td><td>Vertical Line Stride Override

Specifies whether the Vertical Line Stride and Vertical Line Stride Offset fields in the surface state should be replaced by bits 9 and 8 below.

If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:
<table>
<tr><td>Vertical Line Stride

(in surface state)</td><td>Override Vertical Line Stride</td><td>Derived 1-based surface height

(As a function of the 0-based Height in surface state)</td></tr>
<tr><td>0</td><td>0</td><td>Height + 1

(Normal)</td></tr>
<tr><td>0</td><td>1</td><td>(Height +1) / 2

Restriction: (Height + 1) must be an even number.</td></tr>
<tr><td>1</td><td>0</td><td>(Height + 1) * 2</td></tr>
<tr><td>1</td><td>1</td><td>Height + 1

(Normal)</td></tr>
</table>

For example, for a 720x480 standard resolution video buffer, if Vertical Line Stride in surface state is 0, i.e. a frame, Height (of the frame) should be 479. When accessing the bottom field of this frame video buffer, both Override Vertical Line Stride and Override Vertical Line Stride Offset will be set to 1, then the derived surface height (of the field) will be 240 ((Height + 1) / 2). In contrary, if Vertical Line Stride in surface state is 1 and Vertical Line Stride Offset in surface state is 0, the surface state represents the top field of the video buffer. In this case, Height (of the top field) should be programmed as 239. Accessing the bottom video field will use the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset set to 0) will result in a derived surface height of 480 ((Height + 1) * 2).

0 -- Use parameters in the surface state and ignore bits 9:8

1 -- Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset

[DevBW-A] Erratum: This field is ignored by hardware.</td></tr>
<tr><td>9</td><td>Override Vertical Line Stride

Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures.

Format = U1 in lines to skip between logically adjacent lines

[DevBW-A] Erratum: This field is ignored by hardware.</td></tr>
</table>

Bit	Description
13	Reserved: MBZ
8	**Override Vertical Line Stride Offset** Specifies the offset of the initial line from the beginning of the buffer. Ignored when **Override Vertical Line Stride** is 0. Format = U1 in lines of initial offset (when Vertical Line Stride == 1) **[DevBW-A] Erratum**: This field is ignored by hardware.

5.10.5.2 Message Header

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td rowspan="2">M0.5</td><td>31:8</td><td>Ignored</td></tr>
<tr><td>7:0</td><td>FFTID. This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.</td></tr>
<tr><td>M0.4</td><td>31:0</td><td>Ignored (reserved for hardware delivery of binding table pointer)</td></tr>
<tr><td colspan="3">The following M0.2 definition applies only if the Message Mode field is set to NORMAL:</td></tr>
<tr><td rowspan="4">M0.2</td><td>31:22</td><td>Ignored</td></tr>
<tr><td>21:16</td><td>Block Height. Height in rows of block being accessed.
Programming Notes:
The Block Height is restricted to the following maximum values depending on the Block Width:
<table>
<tr><th>Block Width (bytes)</th><th>Maximum Block Height (rows)</th></tr>
<tr><td>1-4</td><td>64</td></tr>
<tr><td>5-8</td><td>32</td></tr>
<tr><td>9-16</td><td>16</td></tr>
<tr><td>17-32</td><td>8</td></tr>
</table>
Format = U6
Range = [0,63] representing 1 to 64 rows</td></tr>
<tr><td>15:5</td><td>Ignored</td></tr>
<tr><td>4:0</td><td>Block Width. Width in bytes of the block being accessed.
Programming Notes:
Must be DWord aligned for the write form of the message.
[DevBW, DevCL] This field must also be DWord aligned for the read form of the message.
Format = U5
Range = [0,31] representing 1 to 32 Bytes</td></tr>
<tr><td>M0.1</td><td>31:0</td><td>Y offset. The Y offset of the upper left corner of the block into the surface.
Format = S31
Programming Notes:
If Message Mode is set to PIXEL_MASK, this field must be a multiple of 4</td></tr>
</table>

DWord	Bit	Description
M0.0	31:0	**X offset.** The X offset of the upper left corner of the block into the surface. Must be DWord aligned (Bits 1:0 MBZ) for the write form of the message. The **X offset** field defines the offset in the input message block. This may differ from the offset in the surface if Color Processing is enabled due to format conversion. **[DevBW, DevCL]** This field must also be DWord aligned for the read form of the message. Format = S31 **Programming Notes:** If **Message Mode** is set to PIXEL_MASK, this field must be a multiple of 32

5.10.5.3 Message Payload (Write)

DWord	Bit	Description
M1:n		**Write Data.** The format of the write data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**.

If **Color Processing Enable** is enabled, the write data is divided into pixels according to the **Message Format** field. The fields within each pixel are defined below. For the 4:2:2 modes, each pixel position includes channels for two pixels.

Message Format	31:24	23:16	15:8	7:0
YUV 4:2:2, 8 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 8 bits per channel	alpha (A)	luminance (Y)	Cb (U)	Cr (V)
	63:48	**47:32**	**31:16**	**15:0**
YUV 4:2:2, 16 bits per channel	Cr (V)	right pixel lum (Y1)	Cb (U)	left pixel lum (Y0)
YUV 4:4:4, 16 bits per channel	alpha (A)	Cr (V)	luminance (Y)	Cb (U)

5.10.5.4 Writeback Message (Read)

DWord	Bit	Description
W0:n		**Read Data.** The format of the read data depends on the **Block Height** and **Block Width**. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the **Block Width**.

5.10.6 DWord Scattered Read/Write

This message takes a set of offsets, and reads or writes 8 or 16 scattered DWords starting at each offset. The Global Offset is added to each of the specific offsets.

Programming Restrictions: Writes to overlapping addresses will have undefined write ordering.

For read messages with X/Y offsets that are outside the bounds of the surface, the address is clamped to the nearest edge of the surface. For write messages with X/Y offsets that are outside the bounds of the surface, the behavior is undefined.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned from the constant buffer to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.
- the surface cannot be tiled
- the surface base address must be DWord aligned
- the **Render Cache Read Write Mode** field in SURFACE_STATE must be set to read/write mode when using this message with the render cache in the surface state model
- the **Stateless Render Cache Read-Write Mode** field in the SVG_WORK_CTL register must be set to read/write mode when using this message with the render cache in the stateless model

Applications:

- SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message)
- SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message)
- general purpose DWord scatter/gathering, used by media

Execution Mask. Depending on the block size, either the low 8 bits or all 16 bits of the execution mask are used to determine which DWords are read into the destination GRF register (for read), or which DWords are written to the surface (for write).

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

5.10.6.1 Message Descriptor

Bit	Description
13	**Invalidate After Read Enable** Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
11:10	Ignored (**[DevCTG]:** bit 11 is part of the **Read Message Type** field for the read version of this message)

Bit	Description
13	**Invalidate After Read Enable** Enabling this field is intended for scratch and spill/fill, where the memory is used only by a single thread and thus does not need to be maintained after the thread completes. Format = Enable
9:8	**Block Size.** Specifies the number of DWords to be read or written 10: 8 DWords 11: 16 DWords All other encodings are reserved.

5.10.6.2 Message Payload

DWord	Bit	Description
M1.7	31:0	**Offset 7.** **[Pre-DevSNB]:** Specifies the byte offset of DWord 7 into the surface. Must be DWord aligned (bits 1:0 MBZ). Format = U32 Range = [0,FFFFFFFCh]
M1.6	31:0	**Offset 6**
M1.5	31:0	**Offset 5**
M1.4	31:0	**Offset 4**
M1.3	31:0	**Offset 3**
M1.2	31:0	**Offset 2**
M1.1	31:0	**Offset 1**
M1.0	31:0	**Offset 0**
M2.7	31:0	**Offset 15.** This message register is included only if the block size is 16 DWords.
M2.6	31:0	**Offset 14**
M2.5	31:0	**Offset 13**
M2.4	31:0	**Offset 12**
M2.3	31:0	**Offset 11**
M2.2	31:0	**Offset 10**
M2.1	31:0	**Offset 9**
M2.0	31:0	**Offset 8**

5.10.6.3 Additional Message Payload (Write)

For the write operation, either one or two additional registers (depending on the block size) of payload contain the data to be written.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords (discard low 2 bits for [Pre-DevSNB]). The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M3.7	31:0	**DWord[Offset7]**
M3.6	31:0	**DWord[Offset6]**
M3.5	31:0	**DWord[Offset5]**
M3.4	31:0	**DWord[Offset4]**
M3.3	31:0	**DWord[Offset3]**
M3.2	31:0	**DWord[Offset2]**
M3.1	31:0	**DWord[Offset1]**
M3.0	31:0	**DWord[Offset0]**
M4.7	31:0	**DWord[Offset15].** This message register is included only if the block size is 16 DWords
M4.6	31:0	**DWord[Offset14]**
M4.5	31:0	**DWord[Offset13]**
M4.4	31:0	**DWord[Offset12]**
M4.3	31:0	**DWord[Offset11]**
M4.2	31:0	**DWord[Offset10]**
M4.1	31:0	**DWord[Offset9]**
M4.0	31:0	**DWord[Offset8]**

5.10.6.4 Writeback Message (Read)

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords (discard low 2 bits for [Pre-DevSNB]). The **DWord** array index is also in units of DWords.

DWord	Bit	Description
W0.7	31:0	**DWord[Offset7]**
W0.6	31:0	**DWord[Offset6]**
W0.5	31:0	**DWord[Offset5]**
W0.4	31:0	**DWord[Offset4]**
W0.3	31:0	**DWord[Offset3]**
W0.2	31:0	**DWord[Offset2]**
W0.1	31:0	**DWord[Offset1]**
W0.0	31:0	**DWord[Offset0]**
W1.7	31:0	**DWord[Offset15].** This writeback message register is included only if the block size is 16 DWords.
W1.6	31:0	**DWord[Offset14]**
W1.5	31:0	**DWord[Offset13]**
W1.4	31:0	**DWord[Offset12]**
W1.3	31:0	**DWord[Offset11]**
W1.2	31:0	**DWord[Offset10]**
W1.1	31:0	**DWord[Offset9]**
W1.0	31:0	**DWord[Offset8]**

5.10.7 DWord Atomic write message [DevGT]

This message takes a set of offsets, and writes 8 scattered DWords starting at each offset. The Global Offset is added to each of the specific offsets. Although this is a write message, it has the read-data returning based on the atomic opcode.

For offsets that are outside the bounds of the surface, the corresponding DW is turned off in the hardware.

Hardware does not check for or optimize for cases where offsets are equal or contiguous, thus for optimal performance in these cases a different message may provide higher performance.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface format is ignored, data is returned to the GRF without format conversion.
- the surface pitch is ignored, the surface is treated as a 1-dimensional surface. An element size (pitch) of 16 bytes is used to determine the size of the buffer for out-of-bounds checking if using the surface state model.

- the surface cannot be tiled
- the surface base address must be DWord aligned

Applications:

- OpenCL compliant kernel and other GPGPU application can use this message to perform atomic operation.

Execution Mask. 8 dword enables are generated out of execution masks.

Out-of-Bounds Accesses. Reads to areas outside of the surface return 0. Writes to areas outside of the surface are dropped and will not modify memory contents.

5.10.7.1 Message Descriptor

Bit	Description
12	**Two-Source Message**. When this bit is set, there are two data-phases for two sources. Two-source message is used only for opcode "0111" and for all other opcodes this bit must be 0. When this bit is 0, M3 is not sent to the data-port.
11:8	**Atomic Operation Code: (Please refer to the table below)** **Unsupported opcodes:** **1101, 1110, 1111**

Opcode	**Operation**	**Return Value**
0000	ADD: new = old + src0	Old value
0001	SUB: new = old – src0	Old value
0010	INC : new = old+1	Old value
0011	DEC: new = old-1	Old value
0100	MIN: new = min(old, src0)	Old value
0101	MAX: new = max(old, src0)	Old value
0110	XCHG: new = src0	Old value
0111	CMPXCHG : new = (old==src0) ? src1 : old	Old value
1000	AND: new = old & src0	Old value
1001	OR: new = old | src0	Old value
1010	XOR: new = old ^ src0	Old value
1011	MIN_SINT: new = min(old, src0)	Old value(signed)
1100	MAX_SINT: new = max(old, src0)	Old value(signed)
		Old value

5.10.7.2 Message Payload

DWord	Bit	Description
M1.7	31:0	**Offset 7.** Specifies the DWord offset of DWord 7 into the surface. Format = U32 Range = [0,3FFFFFFFh]
M1.6	31:0	**Offset 6**
M1.5	31:0	**Offset 5**
M1.4	31:0	**Offset 4**
M1.3	31:0	**Offset 3**
M1.2	31:0	**Offset 2**
M1.1	31:0	**Offset 1**
M1.0	31:0	**Offset 0**

5.10.7.3 Source Payload

Either one or two additional registers (depending on **Two-Source Message**) of source payload contain the data to be used as source.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
M2.7	31:0	**DWord[Offset7] Src0**
M2.6	31:0	**DWord[Offset6] Src0**
M2.5	31:0	**DWord[Offset5] Src0**
M2.4	31:0	**DWord[Offset4] Src0**
M2.3	31:0	**DWord[Offset3] Src0**
M2.2	31:0	**DWord[Offset2] Src0**
M2.1	31:0	**DWord[Offset1] Src0**
M2.0	31:0	**DWord[Offset0] Src0**
M3.7	31:0	**DWord[Offset7] Src1**
M3.6	31:0	**DWord[Offset6] Src1**
M3.5	31:0	**DWord[Offset5] Src1**
M3.4	31:0	**DWord[Offset4] Src1**
M3.3	31:0	**DWord[Offset3] Src1**
M3.2	31:0	**DWord[Offset2] Src1**
M3.1	31:0	**DWord[Offset1] Src1**

DWord	Bit	Description
M3.0	31:0	**DWord[Offset0] Src1**

5.10.7.4 Writeback Message

For the read operation, the writeback message consists of either one or two registers depending on the block size.

The **Offsetn** referred to below is the **Global Offset** added to the corresponding **Offset n** and is in units of DWords. The **DWord** array index is also in units of DWords.

DWord	Bit	Description
W0.7	31:0	**DWord[Offset7]**
W0.6	31:0	**DWord[Offset6]**
W0.5	31:0	**DWord[Offset5]**
W0.4	31:0	**DWord[Offset4]**
W0.3	31:0	**DWord[Offset3]**
W0.2	31:0	**DWord[Offset2]**
W0.1	31:0	**DWord[Offset1]**
W0.0	31:0	**DWord[Offset0]**

5.10.7.5 Message Descriptor

Bit	Description
13	**Commit Enable** Specifies whether the commit is returned to the thread after the fence has been honored. Format = Enable
12:8	Ignored

5.10.7.6 Message Header

The fence messages consist of a single phase, and is completely ignored. The message length is always one.

DWord	Bit	Description
M0.7:0	31:0	Ignored

5.10.7.7 Writeback Message

The writeback message is only sent if **Commit Enable** in the message descriptor is set. The destination register is not modified. Memory fence messages without the **Commit Enable** set will not return anything to the thread (response length is 0 and destination register is null).

DWord	Bit	Description
W0.7:0		Reserved

5.10.8 Render Target Write

This message takes four subspans of pixels for write to a render target. Depending on parameters contained in the message and state, it may also perform a depth and stencil buffer write and/or a render target read for a color blend operation. Additional operations enabled in the Color Calculator state will also be initiated as a result of issuing this message (depth test, alpha test, logic ops, etc.). This message is intended only for use by pixel shader kernels for writing results to render targets.

Restrictions:

- All surface types are allowed.
- Dual Source messages are not supported on **DevBW** and **DevCL-A**
- For SURFTYPE_BUFFER and SURFTYPE_1D surfaces, only the X coordinate is used to index into the surface. The Y coordinate must be zero.
- For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, a **Render Target Array Index** is included in the input message to provide an additional coordinate. The **Render Target Array Index** must be zero for SURFTYPE_BUFFER.
- The surface format is restricted to the set supported as render target. If source/dest color blend is enabled, the surface format is further restricted to the set supported as alpha blend render target.
- **[Pre-DevGT]:** Only one pair of dual source messages is allowed per thread, as these messages implicitly clear the pixel scoreboard. In addition, a thread sending dual source messages is not allowed to send any other render target write messages.
- The last message sent to the render target by a thread must have the **End Of Thread** bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard.
- The stateless model cannot be used with this message (**Binding Table Index** cannot be 255).
- This message can only be issued from a kernel specified in WM_STATE or 3DSTATE_WM (pixel shader kernel), dispatched in non-contiguous mode. Any other kernel issuing this message will cause undefined behavior.
- **[Pre-DevCTG-B]:** The dual source message cannot be used if the **Antialias Alpha Present to Render Target** bit in the message header is enabled.
- **[Pre-DevCTG-B]:** The dual source message cannot be used if the **Alpha Test Enable** bit in COLOR_CALC_STATE is enabled.
- **[DevCTG+]:** The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG.
- This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1)

Execution Mask. The execution mask for render target messages is ignored. Control of which pixels are active is controlled by the **Pixel/Sample Enables** fields in the message header.

Out-of-Bounds Accesses. Accesses to pixels outside of the surface are dropped and will not modify memory contents. However, if the **Render Target Array Index** is out of bounds, it is set to zero and the surface write is not surpressed.

5.10.8.1 Subspan/Pixel to Slot Mapping

The following table indicates the mapping of subspans, pixels, and samples to slots in the pixel shader dispatch depending on the number of samples and message size. This table applies to all devices. Pixels are numbered as follows within a subspan:

0 = upper left

1 = upper right

2 = lower left

3 = lower right

sspi = Starting Sample Pair Index (from the message header)

Message Size	Num Samples	Slot Mapping	
SIMD16	1X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[0]
		Slot[7:4]	= Subspan[1].Pixel[3:0].Sample[0]
		Slot[11:8]	= Subspan[2].Pixel[3:0].Sample[0]
		Slot[15:12]	= Subspan[3].Pixel[3:0].Sample[0]
	4X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[0]
		Slot[7:4]	= Subspan[0].Pixel[3:0].Sample[1]
		Slot[11:8]	= Subspan[0].Pixel[3:0].Sample[2]
		Slot[15:12]	= Subspan[0].Pixel[3:0].Sample[3]
	8X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[2*sspi+0]
		Slot[7:4]	= Subspan[0].Pixel[3:0].Sample[2*sspi+1]
		Slot[11:8]	= Subspan[0].Pixel[3:0].Sample[2*sspi+2]
		Slot[15:12]	= Subspan[0].Pixel[3:0].Sample[2*sspi+3]
SIMD8	1X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[0]
		Slot[7:4]	= Subspan[1].Pixel[3:0].Sample[0]
	4X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[2*sspi+0]
		Slot[7:4]	= Subspan[0].Pixel[3:0].Sample[2*sspi+1]
	8X	Slot[3:0]	= Subspan[0].Pixel[3:0].Sample[2*sspi+0]
		Slot[7:4]	= Subspan[0].Pixel[3:0].Sample[2*sspi+1]

5.10.8.2 Message Descriptor

5.10.8.2.1 [Pre-DevGT]

Bit	Description
11	**Last Render Target Select.** This bit must be set on the last render target write message sent for each group of pixels. For single render target pixel shaders, this bit is set on all render target write messages. For multiple render target pixel shaders, this bit is set only on messages sent to the last render target.
10:8	**Message Type.** This field specifies the type of render target message. For the dual source messages, the low bit indicates which subspan channels to use for the X/Y addresses, stencil, and antialias alpha data. **Programming Notes:** • Replicated data (**Message Type** = 001) is only supported when accessing tiled memory. Using this Message Type to access linear (untiled) memory is UNDEFINED. • **[DevBW, DevCL-A] Errata:** Dual Source messages are not supported • **[DevCL-B]:** The SIMD8 dual source message using subspan 2 & 3 slots (encoding 011) is not supported 000: SIMD16 single source message 001: SIMD16 single source message with replicated data 010: SIMD8 dual source message, use subspan 0 & 1 slots 011: SIMD8 dual source message, use subspan 2 & 3 slots 100: SIMD8 single source message, use subspan 0 & 1 slots 101-111: Reserved

5.10.8.3 Message Header

The render target write message has a two-register message header.

5.10.8.3.1 [Pre-DevSNB]

DWord	Bit	Description
M0.5	31:8	Ignored
	7:0	**FFTID.** The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:6	**Color Calculator State Pointer.** Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:6]

DWord	Bit	Description
	5:0	Ignored
M0.0	31:16	**Dispatched Pixel Enables.** One bit per pixel indicating which pixels were originally enabled when the thread was dispatched. This field is only required for the end-of-thread message and on all dual-source messages. The **Dispatched Pixel Enables** *must be unmodified* from the ones sent when the pixel shader thread was initiated. If the **Dispatched Pixel Enables** are modified, behavior is undefined.
	15:0	**Pixel Enables.** One bit per pixel indicating which pixels are still lit based on kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer.
M1.7	31	Ignored
	30:27	**Viewport Index.** Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]
	26:16	**Render Target Array Index.** Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the "z" or "r" coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] SURFTYPE_BUFFER: must be zero. *face* *Render Target Array Index* ***+x*** ***0*** ***-x*** ***1*** ***+y*** ***2*** ***-y*** ***3*** ***+z*** ***4*** ***-z*** ***5*** Format = U11 The **Render Target Array Index** used by hardware for access to the Render Target is overridden with the **Minimum Array Element** defined in SURFACE_STATE if it is out of the range between **Minimum Array Element** and **Depth**. For cube surfaces, a depth value of 5 is used for this determination.
	15:0	**[DevCTG-B+]: Clipped Out Mask**. One bit per pixel indicating which pixels were discarded due to the kernel's Clip Distance test. For each bit set in this mask, the PS_INVOCATIONS statistics counter register will be decremented by one.. **[Pre-DevCTG-B**]: Ignored
M1.6	31	**Front/Back Facing Polygon.** Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0: Front Facing 1: Back Facing
	30	Ignored

DWord	Bit	Description
	29	**Source Depth Present to Render Target.** Indicates that source depth is included in the message. If **Destination Depth Present** is also set, the depth test and conditional write of the depth buffer must be performed. If **Destination Depth Present** is not set, no depth test is performed but the source depth value is conditionally written to the depth buffer. [ILK] Errata: This bit must be set if stencil test or write is enabled without any depth test or depth write (based on CC state) and if kill-pix (based on WM state) is enabled.
	28	**Destination Depth Present to Render Target.** Indicates that destination depth is included in the message, and that the depth test and conditional write of the depth buffer must be performed. It is not valid to have **Destination Depth Present** without **Source Depth Present**.
	27	**Destination Stencil Present to Render Target.** Indicates that destination stencil is included in the message, and that the stencil test and conditional write of the stencil buffer must be performed.
	26	**Antialias Alpha Present to Render Target.** Indicates that antialias alpha is included in the message, and that the antialias function must be performed.
	25:0	Ignored
M1.5	31:16	**Y3.** Y coordinate for upper-left pixel of subspan 3 Format = U16
	15:0	**X3.** X coordinate for upper-left pixel of subspan 3 Format = U16
M1.4	31:16	**Y2**
	15:0	**X2**
M1.3	31:16	**Y1**
	15:0	**X1**
M1.2	31:16	**Y0**
	15:0	**X0**
M1.1	31:0	Ignored
M1.0	31:0	Ignored

5.10.8.4 Header for SIMD8_IMAGE_WRITE][DevGT]

DWord	Bit	Description
M0.5	31:10	Ignored
	9:8	**Color Code:** This ID is assigned by the Windower unit and is used to track synchronizng events. Format: Reserved for HW Implementation Use.
	7:0	**FFTID.** The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)

DWord	Bit	Description
M0.3	31:0	Ignored
M0.2	31:3	Ignored
	2:0	**Render Target Index.** Specifies the render target index that will be used to select blend state from BLEND_STATE. Format = U3
M0.1	31:6	**Color Calculator State Pointer.** Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the **General State Base Address**. Format = GeneralStateOffset[31:6] **For SIMD8_IMAGE_WR message under normal GPGPU usage model, SW is recommended to program a dummy color-calc state such that all operations controlled by this state are disabled.**
	5:0	Ignored
M0.0	31	Ignored
	30:27	**Viewport Index.** Specifies the index of the viewport currently being used. Format = U4 Range = [0,15] SIMD8_IMAGE_WR message type this field is ignored by hardware.
	26:16	**Render Target Array Index.** Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the "z" or "r" coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] SURFTYPE_BUFFER: must be zero. **face** — **Render Target Array Index** +x 0 -x 1 +y 2 -y 3 +z 4 -z 5 Format = U11 The **Render Target Array Index** used by hardware for access to the Render Target is overridden with the **Minimum Array Element** defined in SURFACE_STATE if it is out of the range between **Minimum Array Element** and **Depth**. For cube surfaces, a depth value of 5 is used for this determination. For SMD8_IMAGE_WRITE : For SURFTYPE_2D, this field must be 0. For SURFTYPE_3D, this field may not be 0 for "Write-3D-Image" operation.

DWord	Bit	Description
	15:8	Ignored
	7:0	Pixel Maks for SIMD8 messages. 1: Pixel is enabled 0: Pixel is disabled , in this case the corresponding (x,y) should be ignored by hardware.
M1.7	31:16	**Y7: y-coordinate for pixel 7** Format = U16
	15:0	**X7: x-coordinate for pixel 7** Format = U16
M1.6	31:16	**Y6: y-coordinate for pixel 6** Format = U16
	15:0	**X6: x-coordinate for pixel 6** Format = U16
M1.5	31:16	**Y5: y-coordinate for pixel 5** Format = U16
	15:0	**X5: x-coordinate for pixel 5** Format = U16
M1.4	31:16	**Y4: y-coordinate for pixel 4** Format = U16
	15:0	**X4: x-coordinate for pixel 4** Format = U16
M1.3	31:16	**Y3: y-coordinate for pixel 3** Format = U16
	15:0	**X3: x-coordinate for pixel 3** Format = U16
M1.2	31:16	**Y2: y-coordinate for pixel 2** Format = U16
	15:0	**X2: x-coordinate for pixel 2** Format = U16
M1.1	31:16	**Y1: y-coordinate for pixel 1** Format = U16
	15:0	**X1: x-coordinate for pixel 1** Format = U16
M1.0	31:16	**Y0: y-coordinate for pixel 0** Format = U16
	15:0	**X0: x-coordinate for pixel 0** Format = U16

5.10.8.5 Stencil and Antialias Alpha Payload ([Pre-DevSNB] only)

The stencil and antialias alpha registers, if included, appears as message register 2 (M2), immediately following the header.

Note that the Antialias Alpha values are U0.4 for [DevBW,DevCL] and U0.8 for [DevCTG].

DWord	Bit	Description
		[DevCTG+]
M2.7	31:24	**Antialias Alpha for Subspan 3, Pixel 3 (lower right)** Format = U0.8 This register is only included if the **Antialias Alpha Present** or **Destination Stencil Present** bit is set.
	23:16	**Antialias Alpha for Subspan 3, Pixel 2 (lower left)**
	15:8	**Antialias Alpha for Subspan 3, Pixel 1 (upper right)**
	7:0	**Antialias Alpha for Subspan 3, Pixel 0 (upper left)**
M2.6	31:24	**Antialias Alpha for Subspan 2, Pixel 3 (lower right)**
	23:16	**Antialias Alpha for Subspan 2, Pixel 2 (lower left)**
	15:8	**Antialias Alpha for Subspan 2, Pixel 1 (upper right)**
	7:0	**Antialias Alpha for Subspan 2, Pixel 0 (upper left)**
M2.5	31:24	**Antialias Alpha for Subspan 1, Pixel 3 (lower right)**
	23:16	**Antialias Alpha for Subspan 1, Pixel 2 (lower left)**
	15:8	**Antialias Alpha for Subspan 1, Pixel 1 (upper right)**
	7:0	**Antialias Alpha for Subspan 1, Pixel 0 (upper left)**
M2.4	31:24	**Antialias Alpha for Subspan 0, Pixel 3 (lower right)**
	23:16	**Antialias Alpha for Subspan 0, Pixel 2 (lower left)**
	15:8	**Antialias Alpha for Subspan 0, Pixel 1 (upper right)**
	7:0	**Antialias Alpha for Subspan 0, Pixel 0 (upper left)**
		[DevBW,DevCL]
M2.7	31:28	**Antialias Alpha for Subspan 3, Pixel 3 (lower right)** Format = U0.4 This register is only included if the **Antialias Alpha Present** or **Destination Stencil Present** bit is set.
	27:24	**Antialias Alpha for Subspan 3, Pixel 2 (lower left)**
	23:20	**Antialias Alpha for Subspan 3, Pixel 1 (upper right)**
	19:16	**Antialias Alpha for Subspan 3, Pixel 0 (upper left)**
	15:12	**Antialias Alpha for Subspan 2, Pixel 3 (lower right)**
	11:8	**Antialias Alpha for Subspan 2, Pixel 2 (lower left)**
	7:4	**Antialias Alpha for Subspan 2, Pixel 1 (upper right)**
	3:0	**Antialias Alpha for Subspan 2, Pixel 0 (upper left)**
M2.6	31:28	**Antialias Alpha for Subspan 1, Pixel 3 (lower right)**
	27:24	**Antialias Alpha for Subspan 1, Pixel 2 (lower left)**
	23:20	**Antialias Alpha for Subspan 1, Pixel 1 (upper right)**

DWord	Bit	Description
	19:16	**Antialias Alpha for Subspan 1, Pixel 0 (upper left)**
	15:12	**Antialias Alpha for Subspan 0, Pixel 3 (lower right)**
	11:8	**Antialias Alpha for Subspan 0, Pixel 2 (lower left)**
	7:4	**Antialias Alpha for Subspan 0, Pixel 1 (upper right)**
	3:0	**Antialias Alpha for Subspan 0, Pixel 0 (upper left)**
M2.5:4		Reserved
M2.3	31:24	**Destination Stencil for Subspan 3, Pixel 3 (lower right)** Format = U8
	23:16	**Destination Stencil for Subspan 3, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 3, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 3, Pixel 0 (upper left)**
M2.2	31:24	**Destination Stencil for Subspan 2, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 2, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 2, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 2, Pixel 0 (upper left)**
M2.1	31:24	**Destination Stencil for Subspan 1, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 1, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 1, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 1, Pixel 0 (upper left)**
M2.0	31:24	**Destination Stencil for Subspan 0, Pixel 3 (lower right)**
	23:16	**Destination Stencil for Subspan 0, Pixel 2 (lower left)**
	15:8	**Destination Stencil for Subspan 0, Pixel 1 (upper right)**
	7:0	**Destination Stencil for Subspan 0, Pixel 0 (upper left)**

5.10.8.6 Color Payload: SIMD16 Single Source

5.10.8.6.1 [Pre-DevGT]

This payload is included if the Message Type is SIMD16 single source. The value of 'm' here is equal to 2 if both stencil and antialias alpha are not present, otherwise it is equal to 3.

DWord	Bit	Description
Mm.7	31:0	**Subspan 1, Pixel 3 (lower right) Red.** Specifies the value of the pixel's red channel. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
Mm.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
Mm.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**

DWord	Bit	Description
Mm.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
Mm.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
Mm.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
Mm.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
M(m+1)		**Subspans 1 and 0 of Green.** See Mm definition for pixel locations
M(m+2)		**Subspans 1 and 0 of Blue.** See Mm definition for pixel locations
M(m+3)		**Subspans 1 and 0 of Alpha** See Mm definition for pixel locations
M(m+4).7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
M(m+4).6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
M(m+4).5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
M(m+4).4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
M(m+4).3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
M(m+4).2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
M(m+4).1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
M(m+4).0	31:0	**Supspan 2, Pixel 0 (upper left) Red**
M(m+5)		**Subspans 3 and 2 of Green.** See M3 definition for pixel locations
M(m+6)		**Subspans 3 and 2 of Blue.** See M3 definition for pixel locations
M(m+7)		**Subspans 3 and 2 of Alpha.** See M3 definition for pixel locations

DWord	Bit	Description
Mm.7	31:0	**Slot 7 Red.** Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	**Slot 6 Red**
Mm.5	31:0	**Slot 5 Red**
Mm.4	31:0	**Slot 4 Red**
Mm.3	31:0	**Slot 3 Red**
Mm.2	31:0	**Slot 2 Red**
Mm.1	31:0	**Slot 1 Red**
Mm.0	31:0	**Slot 0 Red**
M(m+1).7	31:0	**Slot 15 Red**
M(m+1).6	31:0	**Slot 14 Red**

DWord	Bit	Description
M(m+1).5	31:0	**Slot 13 Red**
M(m+1).4	31:0	**Slot 12 Red**
M(m+1).3	31:0	**Slot 11 Red**
M(m+1).2	31:0	**Slot 10 Red**
M(m+1).1	31:0	**Slot 9 Red**
M(m+1).0	31:0	**Slot 8 Red**
M(m+2)		**Slot[7:0] Green.** See Mm definition for slot locations
M(m+3)		**Slot[15:8] Green.** See M(m+1) definition for slot locations
M(m+4)		**Slot[7:0] Blue.** See Mm definition for slot locations
M(m+5)		**Slot[15:8] Blue.** See M(m+1) definition for slot locations
M(m+6)		**Slot[7:0] Alpha.** See Mm definition for slot locations
M(m+7)		**Slot[15:8] Alpha.** See M(m+1) definition for slot locations

5.10.8.7 Color Payload: SIMD8 Single Source

This payload is included if the Message Type is SIMD8 single source or SIMD8 Image Write. For **[Pre-DevSNB]**, the value of 'm' here is equal to 2 if both stencil and antialias alpha are not present, otherwise it is equal to 3. .

DWord	Bit	Description
Mm.7	31:0	**Slot 7 Red.** Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	**Slot 6 Red**
Mm.5	31:0	**Slot 5 Red**
Mm.4	31:0	**Slot 4 Red**
Mm.3	31:0	**Slot 3 Red**
Mm.2	31:0	**Slot 2 Red**
Mm.1	31:0	**Slot 1 Red**
Mm.0	31:0	**Slot 0 Red**
M(m+1)		**Slot[7:0] Green.** See Mm definition for slot locations
M(m+2)		**Slot[7:0] Blue.** See Mm definition for slot locations
M(m+3)		**Slot[7:0] Alpha.** See Mm definition for slot locations

5.10.8.8 Color Payload: SIMD16 Replicated Data

This payload is included if the Message Type specifies single source message with replicated data. One set of R/G/B/A data is included in the message, and this data is replicated to all 16 pixels.

This message is legal with color data only. The registers for depth, stencil, and antialias alpha data cannot be included with this message, and the corresponding bits in the message header must indicate that these registers are not present.

For **[Pre-DevSNB]**, the value of 'm' here is equal to 2.

Programming Notes:

- This message is allowed only on tiled surfaces

DWord	Bit	Description
Mm.7:4	31:0	Reserved
Mm.3	31:0	**Alpha.** Specifies the value of all slots' alpha channel. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.2	31:0	**Blue**
Mm.1	31:0	**Green**
Mm.0	31:0	**Red**

5.10.8.9 Color Payload: SIMD8 Dual Source [DevCL-B], [DevCTG+]

This payload is included if the **Message Type** specifies dual source message. For **[Pre-DevSNB]**, the value of 'm' here is equal to 2 if both tencil and antialias alpha are not present, otherwise it is equal to 3. The dual source message contains only 2 subspans (8 pixels) due to limitations in message length.

DWord	Bit	Description
Mm.7	31:0	**Slot 7 Source 0 Red.** Specifies the value of the slot's red component. Format = IEEE Float, S31, or U32 depending on the **Surface Format** of the surface being accessed. SINT formats use S31, UINT formats use U32, and all other formats use Float.
Mm.6	31:0	**Slot 6 Source 0 Red**
Mm.5	31:0	**Slot 5 Source 0 Red**
Mm.4	31:0	**Slot 4 Source 0 Red**
Mm.3	31:0	**Slot 3 Source 0 Red**
Mm.2	31:0	**Slot 2 Source 0 Red**
Mm.1	31:0	**Slot 1 Source 0 Red**
Mm.0	31:0	**Slot 0 Source 0 Red**
M(m+1)		**Slot[7:0] Source 0 Green.** See Mm definition for slot locations
M(m+2)		**Slot[7:0] Source 0 Blue.** See Mm definition for slot locations
M(m+3)		**Slot[7:0] Source 0 Alpha.** See Mm definition for slot locations
M(m+4)		**Slot[7:0] Source 1 Red.** See Mm definition for slot locations
M(m+5)		**Slot[7:0] Source 1 Green.** See Mm definition for slot locations
M(m+6)		**Slot[7:0] Source 1 Blue.** See Mm definition for slot locations
M(m+7)		**Slot[7:0] Source 1 Alpha.** See Mm definition for slot locations

5.10.8.10 Depth Payload

The depth registers, if included, appear immediately following the color payload.

For the SIMD8 messages, only slot 7:0 data is sent, or only slot 15:8 depending on the **Message Type** encoding. Any complete message register containing ignored data cannot be delivered. Destination Depth is only supported for **[Pre-DevSNB]**.

DWord	Bit	Description
Mn.7	31:0	**Source Depth for Slot 7** Format = IEEE_Float This and the next register is only included if **Source Depth Present** bit is set.
Mn.6	31:0	**Source Depth for Slot 6**
Mn.5	31:0	**Source Depth for Slot 5**
Mn.4	31:0	**Source Depth for Slot 4**

DWord	Bit	Description
Mn.3	31:0	**Source Depth for Slot 3**
Mn.2	31:0	**Source Depth for Slot 2**
Mn.1	31:0	**Source Depth for Slot 1**
Mn.0	31:0	**Source Depth for Slot 0**
M(n+1).7	31:0	**Source Depth for Slot 15**
M(n+1).6	31:0	**Source Depth for Slot 14**
M(n+1).5	31:0	**Source Depth for Slot 13**
M(n+1).4	31:0	**Source Depth for Slot 12**
M(n+1).3	31:0	**Source Depth for Slot 11**
M(n+1).2	31:0	**Source Depth for Slot 10**
M(n+1).1	31:0	**Source Depth for Slot 9**
M(n+1).0	31:0	**Source Depth for Slot 8**
Mk.7	31:0	**Destination Depth for Slot 7 ([Pre-DevGT] only)** Format depends on depth buffer surface format. Software should not modify the destination depth fields from what was delivered in the thread payload. This and the next register is only included if **Destination Depth Present** bit is set.
Mk.6	31:0	**Destination Depth for Slot 6**
Mk.5	31:0	**Destination Depth for Slot 5**
Mk.4	31:0	**Destination Depth for Slot 4**
Mk.3	31:0	**Destination Depth for Slot 3**
Mk.2	31:0	**Destination Depth for Slot 2**
Mk.1	31:0	**Destination Depth for Slot 1**
Mk.0	31:0	**Destination Depth for Slot 0**
M(k+1).7	31:0	**Destination Depth for Slot 15**
M(k+1).6	31:0	**Destination Depth for Slot 14**
M(k+1).5	31:0	**Destination Depth for Slot 13**
M(k+1).4	31:0	**Destination Depth for Slot 12**
M(k+1).3	31:0	**Destination Depth for Slot 11**
M(k+1).2	31:0	**Destination Depth for Slot 10**
M(k+1).1	31:0	**Destination Depth for Slot 9**
M(k+1).0	31:0	**Destination Depth for Slot 8**

5.10.8.11 Message Sequencing Summary

5.10.8.11.1 [Pre-DevSNB]

This section summarizes the sequencing that occurs for each legal render target write message. All messages have the M0 and M1 header registers, thus they are not shown in the table. All cases not shown in this table are illegal.

Key:
s0, s1 = source 0, source 1
1/0 = subspan 1 & 0
3/2 = subspan 3 & 2
sZ = source depth
dZ = destination depth
sten = stencil & antialias alpha

Message Type	Source Depth Present	Dest Stencil Present or AA Alpha	Dest Depth Present	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
000	0	0	0	1/0R	1/0G	1/0B	1/0A	3/2R	3/2G	3/2B	3/2A					
001	0	0	0	RGBA												
010	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
011	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
100	0	0	0	R	G	B	A									
000	1	0	0	1/0R	1/0G	1/0B	1/0A	3/2R	3/2G	3/2B	3/2A	1/0sZ	3/2sZ			
010	1	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
011	1	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
100	1	0	0	R	G	B	A	sZ								
000	1	0	1	1/0R	1/0G	1/0B	1/0A	3/2R	3/2G	3/2B	3/2A	1/0sZ	3/2sZ	1/0dZ	3/2dZ	
010	1	0	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0dZ			
011	1	0	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2dZ			
100	1	0	1	R	G	B	A	sZ	dZ							
000	1	1	0	sten	1/0R	1/0G	1/0B	1/0A	3/2R	3/2G	3/2B	3/2A	1/0sZ	3/2sZ		
010	1	1	0	sten	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
011	1	1	0	sten	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
100	1	1	0	sten	R	G	B	A	sZ							
000	1	1	1	sten	1/0R	1/0G	1/0B	1/0A	3/2R	3/2G	3/2B	3/2A	1/0sZ	3/2sZ	1/0dZ	3/2dZ
100	1	1	1	sten	R	G	B	A	sZ	dZ						

5.10.8.11.2 [DevGT+]

This section summarizes the sequencing that occurs for each legal render target write message. All messages have the M0 and M1 header registers if the header is present. If the header is not present, all registers below are renumbered starting with M0 where M2 appears. All cases not shown in this table are illegal.

Key:
s0, s1 = source 0, source 1
1/0 = slots 15:8
3/2 = slots 7:0
sZ = source depth
oM = oMask

Message Type	oMask Present	Source Depth Present	Source 0 Alpha Present	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
000	0	0	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A					
000	0	0	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A			
000	0	1	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ			
000	0	1	1	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ	
000	1	0	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A				
000	1	0	1	1/0soA	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A		
000	1	1	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ		
000	1	1	1	1/0s0A	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ
001	0	0	0	RGBA												
001	1	0	0	oM	RGBA											
010	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
010	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
010	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
010	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
011	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
011	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
011	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
011	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
100	0	0	0	R	G	B	A									
100	0	0	1	s0A	R	G	B	A								
100	0	1	0	R	G	B	A	sZ								
100	0	1	1	s0A	R	G	B	A	sZ							
100	1	0	0	oM	R	G	B	A								
100	1	0	1	s0A	oM	R	G	B	A							
100	1	1	0	oM	R	G	B	A	sZ							
100	1	1	1	s0A	oM	R	G	B	A	sZ						

5.10.9 Render Target UNORM Read/Write [DevCTG] to [DevSNB]]

This message is supported on [DevCTG] to [DevSNBT] only.

This message reads from or writes to an 8x4 rectangular block of pixels in the render target.

Restrictions:

- the only **Surface Type** allowed is SURFTYPE_2D. Because of this, the stateless surface model is not supported with this message.
- the **Surface Format** must be R8G8B8A8_UNORM, B8G8R8A8_UNORM, R8G8B8X8_UNORM, or B8G8R8X8_UNORM. This is used to determine the pixel structure for boundary clamp, the raw data from the surface is returned to the thread without any format conversion nor filtering operation
- the **Surface Base Address** must be 32-byte aligned
- When a surface is XMajor tiled, (**Tile Walk** field in the surface state is set to TILEWALK_XMAJOR), a memory area mapped through the Render Cache cannot be read and/or written in mixed frame and field modes. For example, if a memory location is first written with a zero Vertical Line Stride (frame mode), and later on (without render cache flush) read back using Vertical Line Stride of one (field mode), the read data stored in GRF are uncertain.
- Unlike the normal "Render Target Write" message, no operations enabled by COLOR_CALC_STATE are supported (alpha blend, alpha test, depth test, stencil, test, logic ops, etc.). **[Pre-DevGT]:** Depth buffer operations are still possible if under conditions of "promoted depth" as described in the Windower chapter. Non-promoted and computed depth cases are not supported with this message.
- The **Target Cache** for the read message must be the Render Cache.
- **[Pre-DevSNB]:** If this message is issued from a windower dispatched thread, only one Render Target UNORM Write message is allowed in each 32-pixel dispatch thread, two are required in each 64-pixel dispatch thread. This is because the scoreboard is cleared whenever this message is issued.

Execution Mask. The execution mask on the send instruction for this type of message is ignored. The data that is written is determined by the **Pixel Mask**.

Out-of-Bounds Accesses. Writes outside of the surface result are dropped and do not modify memory contents. Reads outside of the surface return zero.

5.10.9.1 Message Descriptor

<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>12</td><td>Ignored ([Pre-DevGT]: this bit is part of the Message Type fields)</td></tr>
<tr><td>11</td><td>Ignored ([DevCTG]: this bit is part of the Read Message Type field for the read version of this message)</td></tr>
<tr><td>10</td><td>Vertical Line Stride Override

Specifies whether the Vertical Line Stride and Vertical Line Stride Offset fields in the surface state should be replaced by bits 9 and 8 below.

If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules:

<table>
<tr><th>Vertical Line Stride

(in surface state)</th><th>Override Vertical Line Stride</th><th>Derived 1-based surface height

(As a function of the 0-based Height in surface state)</th></tr>
<tr><td>0</td><td>0</td><td>Height + 1

(Normal)</td></tr>
<tr><td>0</td><td>1</td><td>(Height +1) / 2

Restriction: (Height + 1) must be an even number.</td></tr>
<tr><td>1</td><td>0</td><td>(Height + 1) * 2</td></tr>
<tr><td>1</td><td>1</td><td>Height + 1

(Normal)</td></tr>
</table>

For example, for a 720x480 standard resolution video buffer, if Vertical Line Stride in surface state is 0, i.e. a frame, Height (of the frame) should be 479. When accessing the bottom field of this frame video buffer, both Override Vertical Line Stride and Override Vertical Line Stride Offset will be set to 1, then the derived surface height (of the field) will be 240 ((Height + 1) / 2). In contrary, if Vertical Line Stride in surface state is 1 and Vertical Line Stride Offset in surface state is 0, the surface state represents the top field of the video buffer. In this case, Height (of the top field) should be programmed as 239. Accessing the bottom video field will use the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset set to 0) will result in a derived surface height of 480 ((Height + 1) * 2).

0 -- Use parameters in the surface state and ignore bits 9:8

1 -- Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset</td></tr>
<tr><td>9</td><td>Override Vertical Line Stride

Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures.

Format = U1 in lines to skip between logically adjacent lines</td></tr>
<tr><td>8</td><td>Override Vertical Line Stride Offset

Specifies the offset of the initial line from the beginning of the buffer. Ignored when Override Vertical Line Stride is 0.

Format = U1 in lines of initial offset (when Vertical Line Stride == 1)</td></tr>
</table>

5.10.9.2 Message Header

DWord	Bit	Description
M0.5	31:8	Ignored
	7:0	**Dispatch ID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	**Pixel Mask.** One bit per pixel indicating which pixels are lit, possibly impacted by kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer. This field is ignored by the read message, all pixels are always returned. The bits in this mask correspond to the pixels as follows: 0 1 4 5 16 17 20 21 2 3 6 7 18 19 22 23 8 9 12 13 24 25 28 29 10 11 14 15 26 27 30 31
M0.1	31:0	**Y offset.** The Y offset of the upper left corner of the block into the surface. Must be 4-row aligned (Bits 1:0 MBZ). Format = S31
M0.0	31:0	**X offset.** The X offset of the upper left corner of the block into the surface. This is a pixel offset assuming a 32-bit pixel. Must be 8-pixel aligned (Bits 2:0 MBZ). Format = S31

5.10.9.3 Message Payload (Write Message only)

The channels are defined as follows depending on surface format:

Channel	R8G8B8A8_UNORM R8G8B8X8_UNORM	B8G8R8A8_UNORM B8G8R8X8_UNORM
Channel 0	Red	Blue
Channel 1	Green	Green
Channel 2	Blue	Red
Channel 3	Alpha	Alpha

Pixels are numbered as follows:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

DWord	Bit	Description
M1.7	31:24	**Pixel 15 Channel 1** Format = 8-bit UNORM
	23:16	**Pixel 15 Channel 0**
	15:8	**Pixel 14 Channel 1**
	7:0	**Pixel 14 Channel 0**
M1.6		**Pixel 13 & 12 Channel 1/0**
M1.5		**Pixel 7 & 6 Channel 1/0**
M1.4		**Pixel 5 & 4 Channel 1/0**
M1.3		**Pixel 11 & 10 Channel 1/0**
M1.2		**Pixel 9 & 8 Channel 1/0**
M1.1		**Pixel 3 & 2 Channel 1/0**
M1.0		**Pixel 1 & 0 Channel 1/0**
M2.7		**Pixel 31 & 30 Channel 1/0**
M2.6		**Pixel 29 & 28 Channel 1/0**
M2.5		**Pixel 23 & 22 Channel 1/0**
M2.4		**Pixel 21 & 20 Channel 1/0**
M2.3		**Pixel 27 & 26 Channel 1/0**
M2.2		**Pixel 25 & 24 Channel 1/0**
M2.1		**Pixel 19 & 18 Channel 1/0**
M2.0		**Pixel 17 & 16 Channel 1/0**
M3.7:0		**Pixels 15:0 Channel 3/2**
M4.7:0		**Pixels 31:16 Channel 3/2**

5.10.9.4 Writeback Message (Read Message only)

DWord	Bit	Description
W0.7	31:24	**Pixel 15 Channel 1** Format = 8-bit UNORM
	23:16	**Pixel 15 Channel 0**
	15:8	**Pixel 14 Channel 1**
	7:0	**Pixel 14 Channel 0**
W0.6		**Pixel 13 & 12 Channel 1/0**
W0.5		**Pixel 7 & 6 Channel 1/0**
W0.4		**Pixel 5 & 4 Channel 1/0**
W0.3		**Pixel 11 & 10 Channel 1/0**
W0.2		**Pixel 9 & 8 Channel 1/0**
W0.1		**Pixel 3 & 2 Channel 1/0**
W0.0		**Pixel 1 & 0 Channel 1/0**
W1.7		**Pixel 31 & 30 Channel 1/0**
W1.6		**Pixel 29 & 28 Channel 1/0**
W1.5		**Pixel 23 & 22 Channel 1/0**
W1.4		**Pixel 21 & 20 Channel 1/0**
W1.3		**Pixel 27 & 26 Channel 1/0**
W1.2		**Pixel 25 & 24 Channel 1/0**
W1.1		**Pixel 19 & 18 Channel 1/0**
W1.0		**Pixel 17 & 16 Channel 1/0**
W2.7:0		**Pixels 15:0 Channel 3/2**
W3.7:0		**Pixels 31:16 Channel 3/2**

5.10.10 Streamed Vertex Buffer Write [Pre-DevIVB]

This message writes a single 4-tuple of data to a buffer, at a destination index specified in the message header.

Restrictions:

- surface types allowed are SURFTYPE_BUFFER and SURFTYPE_NULL
- surface formats allowed are indicated in the "Streamed Output Vertex Buffers" column of the Surface Formats table in the Sampling Engine chapter
- the surface cannot be tiled
- use of this message with the **End Of Thread** bit set in the message descriptor is not allowed as the Dispatch ID is not included in the message payload.
- the stateless model cannot be used with this message (**Binding Table Index** cannot be 255).
- Both the surface base address and surface pitch must be DWord aligned.

Execution Mask. The low 4 bits of the execution mask are used to enable the 4 channels of the write to the destination surface.

Out-of-Bounds Accesses. Writes to areas outside of the surface are dropped and will not modify memory contents.

5.10.10.1 Message Descriptor

Bit	Description
12	Ignored (**[Pre-DevGT]:** this bit is part of the **Write Message Type** field)
11	Ignored
10	**[DevCTG]**: **Increment SVBIs.** If set, increment **Streamed Vertex Buffer Index 0-3** **[DevBW,DevCL,ILK]:** Ignored
9	**[DevCTG]**: **Increment Num Prims Written.** If set, increment **SO_NUM_PRIMS_WRITTEN** statistics counter **[DevBW,DevCL,ILK]:** Ignored
8	**[DevCTG]**: **Increment Prim Storage Needed.** If set, increment **SO_PRIM_STORAGE_NEEDED** statistics counter **[DevBW,DevCL,ILK]:** Ignored

5.10.10.2 Message Payload

DWord	Bit	Description
M0.5	31:0	**Destination Index.** Specifies the index into the destination array where the data will be written Format = U32 Range = $[0,2^{27}-1]$
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	**A Data.** Data for the A channel of the destination Format = IEEE_Float, U32, or S31 matching the surface format of the target surface (no format conversion is performed by hardware)

DWord	Bit	Description
M0.2	31:0	**B Data.** Data for the B channel of the destination Format = IEEE_Float, U32, or S31 matching the surface format of the target surface (no format conversion is performed by hardware)
M0.1	31:0	**G Data.** Data for the G channel of the destination Format = IEEE_Float, U32, or S31 matching the surface format of the target surface (no format conversion is performed by hardware)
M0.0	31:0	**R Data.** Data for the R channel of the destination Format = IEEE_Float, U32, or S31 matching the surface format of the target surface (no format conversion is performed by hardware)

5.10.11 AVC Loop Filter Read [DevCTG] to [DevSNB]

This message enables a specially formed AVC Loop Filter control data block to read from the source surface, converted via table-look-up and expanded before being written into the GRF.

Restrictions:

- the only surface type allowed is SURFTYPE_BUFFER.
- the surface base address must be dword aligned
- **[DevBW, DevCL]** This message is not supported.

Applications:

- Specifically for AVC Loop Filter

Execution Mask. The execution mask on the send instruction for this type of message is ignored. The data that is read is determined completely by the message parameters.

Out-of-Bounds Accesses. Read outside of the surface returns zero.

The source surface contains an array of AVC-LF data structure, each corresponds to a macroblock. The AVC-LF data structure contains 16 dwords as shown in the following table.

DWord	Bit	Description
0	31:24	Reserved : MBZ
	23	**FilterTopMbEdgeFlag**
	22	**FilterLeftMbEdgeFlag**
	21	**FilterInternal4x4EdgesFlag**
	20	**FilterInternal8x8EdgesFlag**
	19	**FieldModeAboveMbFlag**
	18	**FieldModeLeftMbFlag**
	17	**FieldModeCurrentMbFlag**

DWord	Bit	Description
	16	**MbaffFrameFlag**
	15:8	**VertOrigin**
	7:0	**HorzOrigin**
1	31:30	**bS_h13**
	29:28	**bS_h12**
	27:26	**bS_h11**
	25:24	**bS_h10**
	23:22	**bS_v33**
	21:20	**bS_v23**
	19:18	**bS_v13**
	17:16	**bS_v03**
	15:14	**bS_v32**
	13:12	**bS_v22**
	11:10	**bS_v12**
	9:8	**bS_v02**
	7:6	**bS_v31**
	5:4	**bS_v21**
	3:2	**bS_v11**
	1:0	**bS_v01**
2	31:28	**bS_v30_0**
	17:24	**bS_v20_0**
	23:20	**bS_v10_0**
	19:16	**bS_v00_0**
	15:14	**bS_h33**
	13:12	**bS_h32**
	11:10	**bS_h31**
	9:8	**bS_h30**
	7:6	**bS_h23**
	5:4	**bS_h22**
	3:2	**bS_h21**
	1:0	**bS_h20**
3	31:28	**bS_h03_0**
	27:24	**bS_h02_0**
	23:20	**bS_h01_0**
	19:16	**bS_h00_0**
	15:12	**bS_v03**

DWord	Bit	Description
	11:8	**bS_v02**
	7:4	**bS_v01**
	3:0	**bS_v00**
4	31:24	**bIndexBinternal_Y** Internal index B for Y
	23:16	**bIndexBinternal_Y** Internal index A for Y
	15:12	**bS_h03_1**
	11:8	**bS_h02_1**
	7:4	**bS_h01_1**
	3:0	**bS_h00_1**
5	31:24	**bIndexBleft1_Y**
	23:16	**bIndexAleft1_Y**
	15:8	**bIndexBleft0_Y**
	7:0	**bIndexAleft0_Y**
6	31:24	**bIndexBtop1_Y**
	23:16	**bIndexAtop1_Y**
	15:8	**bIndexBtop0_Y**
	7:0	**bIndexAtop0_Y**
7	31:24	**bIndexBleft0_Cb**
	23:16	**bIndexAleft0_Cb**
	15:8	**bIndexBinternal_Cb**
	7:0	**bIndexAinternal_Cb**
8	31:24	**bIndexBtop0_Cb**
	23:16	**bIndexAtop0_Cb**
	15:8	**bIndexBleft1_Cb**
	7:0	**bIndexAleft1_Cb**
9	31:24	**bIndexBinternal_Cr**
	23:16	**bIndexAinternal_Cr**
	15:8	**bIndexBtop1_Cb**
	7:0	**bIndexAtop1_Cb**
10	31:24	**bIndexBleft1_Cr**
	23:16	**bIndexAleft1_Cr**
	15:8	**bIndexBleft0_Cr**
	7:0	**bIndexAleft0_Cr**
11	31:24	**bIndexBtop1_Cr**

DWord	Bit	Description
	23:16	**bIndexAtop1_Cr**
	15:8	**bIndexBtop0_Cr**
	7:0	**bIndexAtop0_Cr**
12	31:2	Reserved : MBZ
	1:0	**DisableDeblockingFilterIdc** This is the slice level signal provided as a hint for kernel performance tuning. It is supplied for cases where some slices in a frame have ILDB and some others don't have. In this case, ILDB kernel will be called for all macroblocks in a frame including the ones in the slice that disables ILDB. Setting this bit correctly will ensure that ILDB is not performed on MBs belonging to the slice which has disable deblocking set to 1. For example, kernel may check bit 0, if it is set to 1, no ILDB is performed on the macroblock. 00 - filterInternalEdgesFlag is set equal to 1 01 – disable all deblocking operation, no deblocking parameter syntax element is read; filterInternalEdgesFlag is set equal to 0 10 - macroblocks in different slices are considered not available; filterInternalEdgesFlag is set equal to 1 11 – Reserved (not defined in AVC)
13	31:0	Reserved : MBZ
14	31:0	Reserved : MBZ
15	31:0	Reserved : MBZ

5.10.11.1 Message Descriptor

Bit	Description
12:11	Ignored (**[DevCTG]:** these bits are part of the **Read Message Type** field)
10:8	Ignored

5.10.11.2 Message Header

DWord	Bit	Description
M0.5	31:8	Ignored
	7:0	**Dispatch ID.** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	**Global Offset.** Specifies the global byte offset into the buffer. • This offset must be OWord aligned (bits 3:0 MBZ) Format = U32 Range = [0,FFFFFFF0h]
M0.1	31:0	Ignored
M0.0	31:0	Ignored

5.10.11.3 Writeback Message

The writeback message is formed by the data port using the information from the stored surface and integrated lookup tables defining alpha, beta, tc0, and the edge control map.

Many of the fields are passed directly from the stored surface to the writeback message.

IndexA and IndexB index the following tables to populate the alpha and beta values. These tables are used for Y, Cr, and Cb. IndexTop0 values derive AlphaTop0 and BetaTop0, IndexTop1 values derive AlphaTop1 and BetaTop1, and likewise for the Left values.

Table 5-1.Derivation of offset dependent threshold variables α and β from indexA and indexB

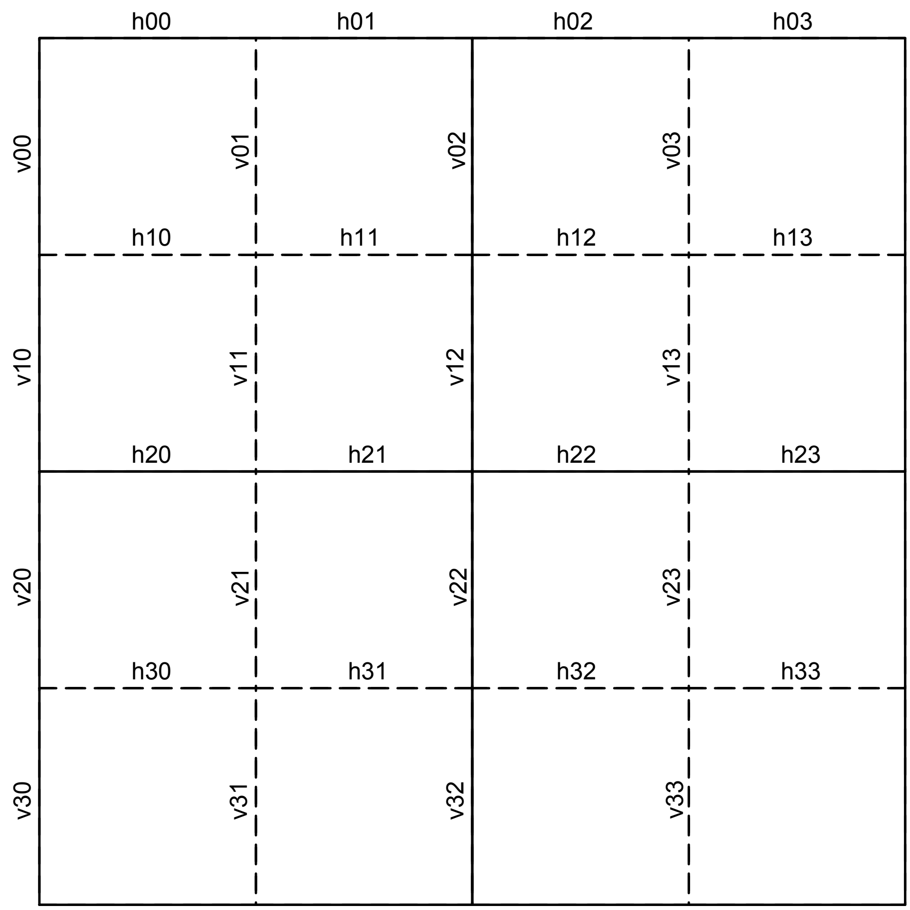
	indexA (for α) or indexB (for β)																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
α	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	5	6	7	8	9	10	12	13
β	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	3	3	3	3	4	4	4

Table 2-1. (Concluded) – Derivation of indexA and indexB from offset dependent threshold variables α and β

	indexA (for α) or indexB (for β)																									
	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
α	15	17	20	22	25	28	32	36	40	45	50	56	63	71	80	90	101	113	127	144	162	182	203	226	255	255
β	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18	18

For each block boundary, the data port must use the boundary strength values to derive tc0 and an edge control map. The following shows the layout of the boundary values in a Y block. Cr and Cb layout follows suit.

Figure 5-1. Boundary Values Layout in a Y Block



The boundary strengths are used in conjunction with indexA to derive tc0 values. The tables below show tc0 output as a function of the boundary strength (bS) and indexA. On external edges, the boundary strength may be 4. Under this condition, hardware should set the value of tc0 to 0.

For determination of tc0, use IndexA0 and external top and left boundary strength (0) values to derive bTc0 values with an index of _0_. During Mbaff mode, use IndexA1 and external top and left boundary strength (1) to derive bTc0 values with an index of _1_. The layout of the tc0 values in the macroblocks corresponds to Figure 5-1 in the same manner as the boundary strengths.

Table 5-2. Value of variable t_{C0} as a function of indexA and bS

	indexA																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
bS = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
bS = 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
bS = 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
bS = 4	tc0 set to 0																									

Table 2-2 (concluded) – Value of variable t_{C0} as a function of indexA and bS

	indexA																									
	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
bS = 1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13
bS = 2	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5	5	6	7	8	8	10	11	12	13	15	17
bS = 3	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13	14	16	18	20	23	25
bS = 4	tc0 set to 0																									

The boundary strengths also create the edge control maps in the writeback message. The internal boundaries require one control map set according to the boundary strength to drive the deblocking functionality. The external boundaries require two control maps set according to the boundary strength to enable deblocking and choose the deblocking algorithm. These control maps are shown in the tables below. Each edge's boundary strength has a corresponding edge control map (e.g. bS_v01 corresponds to EdgeCntlMap_v01).

Table 5-3. Boundary Strength Mapping to Edge Control Map: Internal Boundaries

bS	Internal boundary Edge Control Map	Description
00	0000	bS = 0, no de-blocking
01	1111	Perform de-blocking using bS < 4 algorithm
10	1111	Perform de-blocking using bS < 4 algorithm
11	1111	Perform de-blocking using bS < 4 algorithm

Table 5-4. Boundary Strength Mapping to Edge Control Map A: External Boundaries, Deblocking Enable

bS	External boundary Edge Control Map A	Description
0000	0000	bS = 0, no de-blocking
0001	1111	bS > 0, de-blocking the segment
0010	1111	bS > 0, de-blocking the segment
0011	1111	bS > 0, de-blocking the segment
0100	1111	bS > 0, de-blocking the segment

Table 5-5. Boundary Strength Mapping to Edge Control Map B: External Boundaries, Deblocking Algorithm

bS	External boundary Edge Control Map B	Description
0000	0000	(No deblocking, set algorithm to 0)
0001	0000	Perform de-blocking using bS < 4 algorithm
0010	0000	Perform de-blocking using bS < 4 algorithm
0011	0000	Perform de-blocking using bS < 4 algorithm
0100	1111	Perform de-blocking using bS = 4 algorithm

The following is the layout of the combined writeback message.

DWord	Bit	Description
W0.7	31:24	**bIndexBleft0_Cb**
	23:16	**bIndexAleft0_Cb**
	15:8	**bIndexBinternal_Cb**
	7:0	**bIndexAinternal_Cb**
W0.6	31:24	**bIndexBtop1_Y**
	23:16	**bIndexAtop1_Y**
	15:8	**bIndexBtop0_Y**
	7:0	**bIndexAtop0_Y**
W0.5	31:24	**bIndexBleft1_Y**
	23:16	**bIndexAleft1_Y**
	15:8	**bIndexBleft0_Y**
	7:0	**bIndexAleft0_Y**
W0.4	31:24	**bIndexBinternal_Y** Internal index B for Y

DWord	Bit	Description
	23:16	**bIndexAinternal_Y** Internal index A for Y
	15:12	**bS_h03_1**
	11:8	**bS_h02_1**
	7:4	**bS_h01_1**
	3:0	**bS_h00_1**
W0.3	31:28	**bS_h03_0**
	27:24	**bS_h02_0**
	23:20	**bS_h01_0**
	19:16	**bS_h00_0**
	15:12	**bS_v30_1**
	11:8	**bS_v20_1**
	7:4	**bS_v10_1**
	3:0	**bS_v00_1**
W0.2	31:28	**bS_v30_0**
	27:24	**bS_v20_0**
	23:20	**bS_v10_0**
	19:16	**bS_v00_0**
	15:8	**bbSinternalBotHorz**
	7:0	**bbSinternalMidHorz**
W0.1	31:30	**bS_h13**
	29:28	**bS_h12**
	27:26	**bS_h11**
	25:24	**bS_h10**
	23:22	**bS_v33**
	21:20	**bS_v23**
	19:18	**bS_v13**
	17:16	**bS_v03**
	15:14	**bS_v32**
	13:12	**bS_v22**
	11:10	**bS_v12**
	9:8	**bS_v02**
	7:6	**bS_v31**
	5:4	**bS_v21**
	3:2	**bS_v11**
	1:0	**bS_v01**

DWord	Bit	Description
W0.0	31:24	Reserved : MBZ
	23	**FilterTopMbEdgeFlag**
	22	**FilterLeftMbEdgeFlag**
	21	**FilterInternal4x4EdgesFlag**
	20	**FilterInternal8x8EdgesFlag**
	19	**FieldModeAboveMbFlag**
	18	**FieldModeLeftMbFlag**
	17	**FieldModeCurrentMbFlag**
	16	**MbaffFrameFlag**
	15:8	**VertOrigin**
	7:0	**HorzOrigin**
W1.7	31:0	Reserved : MBZ
W1.6	31:0	Reserved : MBZ
W1.5	31:0	Reserved : MBZ
W1.4	31:0	Reserved : MBZ
W1.3	31:24	**bIndexBtop1_Cr**
	23:16	**bIndexAtop1_Cr**
	15:8	**bIndexBtop0_Cr**
	7:0	**bIndexAtop0_Cr**
W1.2	31:24	**bIndexBleft1_Cr**
	23:16	**bIndexAleft1_Cr**
	15:8	**bIndexBleft0_Cr**
	7:0	**bIndexAleft0_Cr**
W1.1	31:24	**bIndexBinternal_Cr**
	23:16	**bIndexAinternal_Cr**
	15:8	**bIndexBtop1_Cb**
	7:0	**bIndexAtop1_Cb**
W1.0	31:24	**bIndexBtop0_Cb**
	23:16	**bIndexAtop0_Cb**
	15:8	**bIndexBleft1_Cb**
	7:0	**bIndexAleft1_Cb**
W2.7	31:28	**EdgeCntlMapB_h03_1** Used in Mbaff mode only
	27:24	**EdgeCntlMapB_h02_1** Used in Mbaff mode only

DWord	Bit	Description
	23:20	**EdgeCntlMapB_h01_1** Used in Mbaff mode only
	19:16	**EdgeCntlMapB_h00_1** Used in Mbaff mode only
	15:12	**EdgeCntlMapA_h03_1** Used in Mbaff mode only
	11:8	**EdgeCntlMapA_h02_1** Used in Mbaff mode only
	7:4	**EdgeCntlMapA_h01_1** Used in Mbaff mode only
	3:0	**EdgeCntlMapA_h00_1** Used in Mbaff mode only
W2.6	31:28	**EdgeCntlMapB_v30_1** Used in Mbaff mode only
	27:24	**EdgeCntlMapB_v20_1** Used in Mbaff mode only
	23:20	**EdgeCntlMapB_v01_1** Used in Mbaff mode only
	19:16	**EdgeCntlMapB_v00_1** Used in Mbaff mode only
	15:12	**EdgeCntlMapA_v30_1** Used in Mbaff mode only
	11:8	**EdgeCntlMapA_v20_1** Used in Mbaff mode only
	7:4	**EdgeCntlMapA_v10_1** Used in Mbaff mode only
	3:0	**EdgeCntlMapA_v00_1** Used in Mbaff mode only
W2.5	31:28	**EdgeCntlMapB_h03_0**
	27:24	**EdgeCntlMapB_h02_0**
	23:20	**EdgeCntlMapB_h01_0**
	19:16	**EdgeCntlMapB_h00_0**
	15:12	**EdgeCntlMapA_h03_0**
	11:8	**EdgeCntlMapA_h02_0**
	7:4	**EdgeCntlMapA_h01_0**
	3:0	**EdgeCntlMapA_h00_0**
W2.4	31:28	**EdgeCntlMapB_v30_0**

DWord	Bit	Description
	27:24	**EdgeCntlMapB_v20_0**
	23:20	**EdgeCntlMapB_v10_0**
	19:16	**EdgeCntlMapB_v00_0**
	15:12	**EdgeCntlMapA_v30_0**
	11:8	**EdgeCntlMapA_v20_0**
	7:4	**EdgeCntlMapA_v10_0**
	3:0	**EdgeCntlMapA_v00_0**
W2.3	31:0	Reserved : MBZ
W2.2	31:28	**EdgeCntlMap_h33**
	27:24	**EdgeCntlMap_h32**
	23:20	**EdgeCntlMap_h31**
	19:16	**EdgeCntlMap_h30**
	15:12	**EdgeCntlMap_h23**
	11:8	**EdgeCntlMap_h22**
	7:4	**EdgeCntlMap_h21**
	3:0	**EdgeCntlMap_h20**
W2.1	31:28	**EdgeCntlMap_h13**
	27:24	**EdgeCntlMap_h12**
	23:20	**EdgeCntlMap_h11**
	19:16	**EdgeCntlMap_h10**
	15:12	**EdgeCntlMap_v33**
	11:8	**EdgeCntlMap_v23**
	7:4	**EdgeCntlMap_v13**
	3:0	**EdgeCntlMap_v03**
W2.0	31:28	**EdgeCntlMap_v32**
	27:24	**EdgeCntlMap_v22**
	23:20	**EdgeCntlMap_v12**
	19:16	**EdgeCntlMap_v02**
	15:12	**EdgeCntlMap_v31**
	11:8	**EdgeCntlMap_v21**
	7:4	**EdgeCntlMap_v11**
	3:0	**EdgeCntlMap_v01**
W3.7	31:24	**bTc0_h33_0_Y**
	23:16	**bTc0_h32_0_Y**
	15:8	**bTc0_h31_0_Y**
	7:0	**bTc0_h30_0_Y**

DWord	Bit	Description
W3.6	31:24	**bTc0_h23_0_Y**
	23:16	**bTc0_h22_0_Y**
	15:8	**bTc0_h21_0_Y**
	7:0	**bTc0_h20_0_Y**
W3.5	31:24	**bTc0_h13_0_Y**
	23:16	**bTc0_h12_0_Y**
	15:8	**bTc0_h11_0_Y**
	7:0	**bTc0_h10_0_Y**
W3.4	31:24	**bTc0_h03_0_Y**
	23:16	**bTc0_h02_0_Y**
	15:8	**bTc0_h01_0_Y**
	7:0	**bTc0_h00_0_Y**
W3.3	31:24	**bTc0_v33_Y**
	23:16	**bTc0_v23_Y**
	15:8	**bTc0_v13_Y**
	7:0	**bTc0_v03_Y**
W3.2	31:24	**bTc0_v32_Y**
	23:16	**bTc0_v22_Y**
	15:8	**bTc0_v12_Y**
	7:0	**bTc0_v02_Y**
W3.1	31:24	**bTc0_v31_Y**
	23:16	**bTc0_v21_Y**
	15:8	**bTc0_v11_Y**
	7:0	**bTc0_v01_Y**
W3.0	31:24	**bTc0_v30_0_Y**
	23:16	**bTc0_v20_0_Y**
	15:8	**bTc0_v10_0_Y**
	7:0	**bTc0_v00_0_Y**
W4.7	31:24	**bTc0_h03_1_Y** Used in Mbaff mode only
	23:16	**bTc0_h02_1_Y** Used in Mbaff mode only
	15:8	**bTc0_h01_1_Y** Used in Mbaff mode only
	7:0	**bTc0_h00_1_Y** Used in Mbaff mode only

DWord	Bit	Description
W4.6	31:24	**bTc0_v30_1_Y** Used in Mbaff mode only
	23:16	**bTc0_v20_1_Y** Used in Mbaff mode only
	15:8	**bTc0_v10_1_Y** Used in Mbaff mode only
	7:0	**bTc0_v00_1_Y** Used in Mbaff mode only
W4.5	31:0	MBZ
W4.4	31:24	**bBetaTop1_Y**
	23:16	**bAlphaTop1_Y**
	15:8	**bBetaLeft1_Y**
	7:0	**bAlphaLeft1_Y**
W4.3	31:0	MBZ
W4.2	31:0	MBZ
W4.1	31:16	MBZ
	15:8	**bBetaInternal_Y**
	7:0	**bAlphaInternal_Y**
W4.0	31:24	**bBetaTop0_Y**
	23:16	**bAlphaTop0_Y**
	15:8	**bBetaLeft0_Y**
	7:0	**bAlphaLeft0_Y**
W5.7	31:24	**bTc0_h23_Cr**
	23:16	**bTc0_h22_Cr**
	15:8	**bTc0_h21_Cr**
	7:0	**bTc0_h20_Cr**
W5.6	31:24	**bTc0_h03_0_Cr**
	23:16	**bTc0_h02_0_Cr**
	15:8	**bTc0_h01_0_Cr**
	7:0	**bTc0_h00_0_Cr**
W5.5	31:24	**bTc0_v32_Cr**
	23:16	**bTc0_v22_Cr**
	15:8	**bTc0_v12_Cr**
	7:0	**bTc0_v02_Cr**
W5.4	31:24	**bTc0_v30_0_Cr**
	23:16	**bTc0_v20_0_Cr**

DWord	Bit	Description
	15:8	**bTc0_v10_0_Cr**
	7:0	**bTc0_v00_0_Cr**
W5.3	31:24	**bTc0_h23_Cb**
	23:16	**bTc0_h22_Cb**
	15:8	**bTc0_h21_Cb**
	7:0	**bTc0_h20_Cb**
W5.2	31:24	**bTc0_h03_0_Cb**
	23:16	**bTc0_h02_0_Cb**
	15:8	**bTc0_h01_0_Cb**
	7:0	**bTc0_h00_0_Cb**
W5.1	31:24	**bTc0_v32_Cb**
	23:16	**bTc0_v22_Cb**
	15:8	**bTc0_v12_Cb**
	7:0	**bTc0_v02_Cb**
W5.0	31:24	**bTc0_v30_0_Cb**
	23:16	**bTc0_v20_0_Cb**
	15:8	**bTc0_v10_0_Cb**
	7:0	**bTc0_v00_0_Cb**
W6.7	31:0	MBZ
W6.6	31:0	MBZ
W6.5	31:0	MBZ
W6.4	31:0	MBZ
W6.3	31:16	MBZ
	15:8	**bBetaInternal_Cr**
	7:0	**bAlphaInternal_Cr**
W6.2	31:24	**bBetaTop0_Cr**
	23:16	**bAlphaTop0_Cr**
	15:8	**bBetaLeft0_Cr**
	7:0	**bAlphaLeft0_Cr**
W6.1	31:16	MBZ
	15:8	**bBetaInternal_Cb**
	7:0	**bAlphaInternal_Cb**
W6.0	31:24	**bBetaTop0_Cb**
	23:16	**bAlphaTop0_Cb**
	15:8	**bBetaLeft0_Cb**
	7:0	**bAlphaLeft0_Cb**

DWord	Bit	Description
W7.7	31:24	**bTc0_h03_1_Cr**
	23:16	**bTc0_h02_1_Cr**
	15:8	**bTc0_h01_1_Cr**
	7:0	**bTc0_h00_1_Cr**
W7.6	31:24	**bTc0_v30_1_Cr**
	23:16	**bTc0_v20_1_Cr**
	15:8	**bTc0_v10_1_Cr**
	7:0	**bTc0_v00_1_Cr**
W7.5	31:0	MBZ
W7.4	31:24	**bBetaTop1_Cr**
	23:16	**bAlphaTop1_Cr**
	15:8	**bBetaLeft1_Cr**
	7:0	**bAlphaLeft1_Cr**
W7.3	31:24	**bTc0_h03_1_Cb**
	23:16	**bTc0_h02_1_Cb**
	15:8	**bTc0_h01_1_Cb**
	7:0	**bTc0_h00_1_Cb**
W7.2	31:24	**bTc0_v30_1_Cb**
	23:16	**bTc0_v20_1_Cb**
	15:8	**bTc0_v10_1_Cb**
	7:0	**bTc0_v00_1_Cb**
W7.1	31:0	MBZ
W7.0	31:24	**bBetaTop1_Cb**
	23:16	**bAlphaTop1_Cb**
	15:8	**bBetaLeft1_Cb**
	7:0	**bAlphaLeft1_Cb**

5.10.12 Flush Render Cache [Pre-DevSNB]

This message causes a flush of the render cache. The flush occurs in-order relative to message arrival at the write data port. It is not synchronized with messages to the read data port.

If the **Send Write Commit Message** bit in the message descriptor is set for this message, the writeback message is delivered after the cache flush has been completed.

5.10.12.1 Message Descriptor

Bit	Description
11:8	Ignored

5.10.12.2 Message Payload

DWord	Bit	Description
M0.5:0	31:0	Ignored

6. Extended Math

6.1 Messages

Restrictions:

- Use of any message to the Extended Math with the **End of Thread** bit set in the message descriptor is not allowed.
- The Extended Math supports vector operations up to 8 channels. It only looks at the lower 8 channel enables (execution mask bits), and ignores the higher 8.

6.1.1 Initiating Message

6.1.1.1 Message Descriptor

Bit	Description
19	**[DevILK]: Header Present** This bit must be set to **zero** for all Extended Math messages. **[Pre-DevILK]:** this bit is not part of the shared function specific message descriptor**.**
18:9	Reserved : MBZ **[Pre- DevILK]:** Bits 18:16 are not part of the shared function specific message descriptor.
7	**Source Structure.** This bit indicates whether the operation is based on vector inputs or scalar inputs. If this bit is not set, the Extended Math performs the indicated math function on a channel by channel basis. For an enabled channel, EM takes the input data from the corresponding channel and outputs the result in the same position. If this bit is set, EM performs the math function on a 4-channel group basis. If any of the 4 channels within a group is enabled, the data on the first channel (channel 0) is used as the input. The result is broadcasted to all enabled channels within the group. See section 6.1.1.2 below for more details. 0: vector structure 1: scalar structure
6	**Saturate Control** 0: no saturate 1: saturate result to [0,1] range (allowed only on floating point math functions)
5	**Precision.** This bit provides a hint whether the indicated math function is performed in full precision or partial precision. It is only valid for floating point math functions when the floating point mode is in alternative mode. It is ignored if the floating point mode is in IEEE754 mode. Floating point mode is selected via the **Floating Point Mode** bit in CR0. This bit is also ignored for integer math functions. See section for more details. 0: use full precision 1: use partial precision

Bit	Description
4	**Integer Type.** Determines the data type for both source and destination operands of the INT DIV functions. Ignored for other functions. 0: unsigned integer 1: signed integer
3:0	**Math Function.** For floating point math functions (1h to Ah), the floating point mode signal in the request message (originated from the Floating Point Mode bit in CR0) determines whether the operation is in IEEE754 floating point mode or in alternative floating point mode. Functions LOG and EXP are base 2. SIN, COS, SINCOS take inputs in radians. 0h: Reserved 1h: INV (reciprocal) 2h: LOG 3h: EXP 4h: SQRT 5h: RSQ 6h: SIN 7h: COS 8h: SINCOS 9h: Reserved Ah: POW Bh: INT DIV – return quotient and remainder Ch: INT DIV – return quotient only Dh: INT DIV – return remainder only Eh: Reserved Fh: Reserved

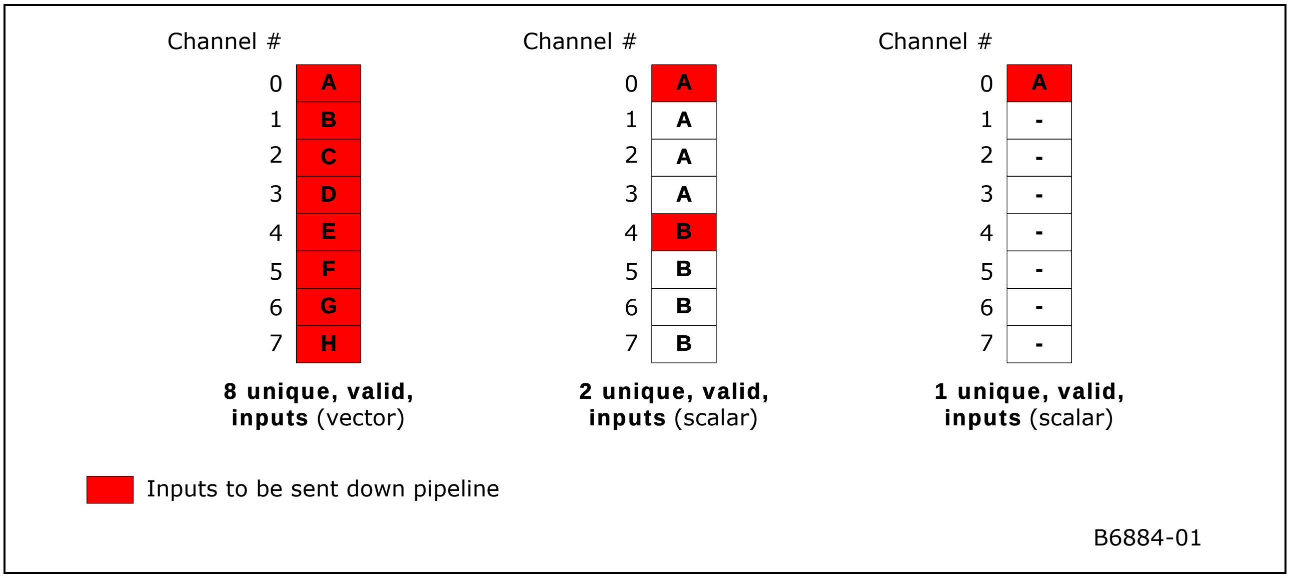
6.1.1.2 Scalar and Vector Mode

For a given request message, the Extended Math examines the 8-bit channel enable field and the Source Structure field in the message descriptor to determine which dwords contain valid inputs. There are two general cases that EM sees.

- **Vector mode**: The first case is when the Source Structure is a vector structure. In this vector mode, 8 input data channels contain 8 unique input values. The channel enable bits in the sideband determine which one of the 8 input values are valid and therefore need to be computed and outputted. It is possible that none of the channels are enabled, or all 8 channels are enabled, or anything in between. EM only sends the valid input values into the compute pipeline to achieve higher throughput. As the channel enable field is forwarded to the writeback message bus, only the resulting values with channel enable bit on are written back to the requesting thread's GRF register.

- **Scalar mode**: The second case is when the Source Structure is a scalar structure. In this scalar mode, there may be up to 2 unique input values present, one for each group of 4 channels. The 2 unique input values reside in the first channel of each group of 4, channel 0 and channel 4, specifically. The computed results of the two scalar inputs are replicated to the corresponding 4 channels. The sideband channel enable field determines which channels are enabled at the final output. It is obvious that as long as any bit out of a group of four channel-enable bits are set, the corresponding scalar data must be computed. Inversely, if all four channel enable bits in a group are zero, computation of the corresponding scalar is skipped.

A subset of the scalar mode is when there is only one valid input. In this case the channel enable field will show that one of the two groups of four does not contain valid data. These three cases are illustrated below:


Channel #
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 unique, valid, inputs (vector)
Channel #
0 A
1 A
2 A
3 A
4 B
5 B
6 B
7 B
2 unique, valid, inputs (scalar)
Channel #
0 A
1 -
2 -
3 -
4 -
5 -
6 -
7 -
1 unique, valid, inputs (scalar)
Inputs to be sent down pipeline
B6884-01

6.1.1.3 Message Payload

8 channel message:

All incoming messages are comprised of a single message register except the POW function and INT DIV, which consist of two message registers. The higher 8 bits are ignored by hardware. The lower 8 bits of the channel enables (execution mask) are used as the (dword) channel enables for the math function operation.

[DevCTG+] 16 channel message:

In additional to the 8 channel message type described above, 16 channel message type is also supported for all functions except POW and INT DIV which require two operands. A 16 channel message consists of two message registers. In this case, all 16 bits of channel enables are used, with the higher 8 bits as the enables for the corresponding operands (from 8 to 15).

Message registers for 8-channel message:

DWord	Bit	Description
M0.7	31:0	**Operand0[7].** The value of Operand0 for element 7 For the POW function, this operand is the base For the INT DIV functions, this operand is the denominator For all other functions, this operand is the single input operand Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
M0.6	31:0	**Operand0[6].** Refer to Operand0[7] above for the function of this operand.
M0.5	31:0	**Operand0[5].** Refer to Operand0[7] above for the function of this operand.
M0.4	31:0	**Operand0[4].** Refer to Operand0[7] above for the function of this operand.
M0.3	31:0	**Operand0[3].** Refer to Operand0[7] above for the function of this operand.
M0.2	31:0	**Operand0[2].** Refer to Operand0[7] above for the function of this operand.
M0.1	31:0	**Operand0[1].** Refer to Operand0[7] above for the function of this operand.
M0.0	31:0	**Operand0[0].** Refer to Operand0[7] above for the function of this operand.
M1.7	31:0	**Operand1[7].** The value of Operand1 for element 7 For the POW function, this operand is the power For the INT DIV functions, this operand is the numerator For all other functions, this data phase of the message is not present Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
M1.6	31:0	**Operand1[6].** Refer to Operand1[7] above for the function of this operand.
M1.5	31:0	**Operand1[5].** Refer to Operand1[7] above for the function of this operand.
M1.4	31:0	**Operand1[4].** Refer to Operand1[7] above for the function of this operand.
M1.3	31:0	**Operand1[3].** Refer to Operand1[7] above for the function of this operand.

DWord	Bit	Description
M1.2	31:0	**Operand1[2].** Refer to Operand1[7] above for the function of this operand.
M1.1	31:0	**Operand1[1].** Refer to Operand1[7] above for the function of this operand.
M1.0	31:0	**Operand1[0].** Refer to Operand1[7] above for the function of this operand.

[DevCTG+] Message registers for 16-channel message, which is not valid for POW and INT DIV:

DWord	Bit	Description
M0.7	31:0	**Operand0[7].** The value of Operand0 for element 7 This operand is the single input operand Format = IEEE Float or Alternative Float depending on floating point mode signal
M0.6	31:0	**Operand0[6].** Refer to Operand0[7] above for the function of this operand.
M0.5	31:0	**Operand0[5].** Refer to Operand0[7] above for the function of this operand.
M0.4	31:0	**Operand0[4].** Refer to Operand0[7] above for the function of this operand.
M0.3	31:0	**Operand0[3].** Refer to Operand0[7] above for the function of this operand.
M0.2	31:0	**Operand0[2].** Refer to Operand0[7] above for the function of this operand.
M0.1	31:0	**Operand0[1].** Refer to Operand0[7] above for the function of this operand.
M0.0	31:0	**Operand0[0].** Refer to Operand0[7] above for the function of this operand.
M1.7	31:0	**Operand1[15].** Refer to Operand0[7] above for the function of this operand.
M1.6	31:0	**Operand1[14].** Refer to Operand0[7] above for the function of this operand.
M1.5	31:0	**Operand1[13].** Refer to Operand0[7] above for the function of this operand.
M1.4	31:0	**Operand1[12].** Refer to Operand0[7] above for the function of this operand.
M1.3	31:0	**Operand1[11].** Refer to Operand0[7] above for the function of this operand.
M1.2	31:0	**Operand1[10].** Refer to Operand0[7] above for the function of this operand.
M1.1	31:0	**Operand1[9].** Refer to Operand0[7] above for the function of this operand.
M1.0	31:0	**Operand1[8].** Refer to Operand0[7] above for the function of this operand.

6.1.2 Writeback Message

Writeback messages for most EM functions contain a single GRF register. The exceptions to this rule are SINCOS and INT DIV. SINCOS returns two GRF registers, the first register contains the computed Sine of the inputs, and the second contains the computed Cosine values. INT DIV returns the quotient in the first GRF register and the remainder in the second GRF register. The two GRF registers are adjacent.

The lower 8 bits of the channel enables (execution mask) of the writeback bus are the same 8 (dword) channel enables of the request message. Because EM supports vector operations with a maximum of 8 channels, the higher 8 bits of the channel enables are set to 0. The same 16-bit channel enables are repeated for the second GRF register write, if present.

DWord	Bit	Description
W0.7	31:0	**Result0[7].** The value of Result0 for element 7 For the SINCOS function, this result is the sine For the INT DIV (return quotient and remainder) functions, this result is the quotient For all other functions, this result is the single output result Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
W0.6	31:0	**Result0[6]**
W0.5	31:0	**Result0[5]**
W0.4	31:0	**Result0[4]**
W0.3	31:0	**Result0[3]**
W0.2	31:0	**Result0[2]**
W0.1	31:0	**Result0[1]**
W0.0	31:0	**Result0[0]**
W1.7	31:0	**Result1[7].** The value of Result1 for element 7 For the SINCOS function, this result is the cosine For the INT DIV (return quotient and remainder) functions, this result is the remainder For all other functions, this data phase of the message is not present Format = S31 or U32 depending on **Integer Type** for INT DIV functions Format = IEEE Float or Alternative Float depending on floating point mode signal for all other functions
W1.6	31:0	**Result1[6]**
W1.5	31:0	**Result1[5]**
W1.4	31:0	**Result1[4]**
W1.3	31:0	**Result1[3]**
W1.2	31:0	**Result1[2]**
W1.1	31:0	**Result1[1]**
W1.0	31:0	**Result1[0]**

6.2 Performance

The Extended Math shared function unit supports extended math functions with up to 8 data channels per request. Computations for a vector request are performed channel by channel on a serial execution pipeline. Most functions require iterative computations. For example, SQRT takes three rounds of computation in the serial execution pipeline. The latency for each round is about 22 clocks. Trigonometric functions may take variable number of rounds depending on the input data. For certain math functions, the throughput with partial precision computation in alternative floating point mode is higher than the full precision computation. After computations for all channels of a request are completed, data vectors (of one or two phases) are assembled before the writeback message is sent back to the requesting thread.

The following table shows the number of rounds per element for each function type. The table may be used to estimate the utilization of the extended math unit and the minimal latency of the message.

Function	Throughput (rounds/element)	Note
INV	1	
LOG	Partial: 2 Full: 3	Computes Log base 2
SQRT	3	Implemented as: √x = x * 1/√x
RSQ	2	
EXP	Full: 4 Partial: 3	Both partial and full precision versions have the same throughput. Computes 2^x (anti-log)
POW	8	
SIN	Min: 5 Max: 12 Typical: 6	Trigonometric functions are the only ones with variable throughput. Throughput depends on the input data range. Input is in radians
COS	Same as SIN	Input is in radians
SINCOS	See SIN	The two-output-phase SINCOS function is implemented as back to back SIN and COS functions. Input is in radians
INT DIV	Quotient: 3 Remainder: 4	

To best utilize the extended math shared function, programmers should consider the following characteristics of the shared function:

- In vector mode, only the enabled channels consume computation rounds, while the disabled channels do not.
- In scalar mode, one data element is computed for a group of 4 channels if any of the 4 channels is enabled. If all 4 channels are disabled, no compute cycle is wasted for the group.

6.3 Function Reference

A math function may take one request message register (src0) or two request message registers (src0 and src1), and may output one writeback message register (dst0) or two writeback message registers (dst0 and dst1).

Vector mode or scalar mode is determined by the Source Structure field of message descriptor.

The operations is based on the channel enables as noted by EMask.

6.3.1 INV

Description Computes reciprocal of src0 (32-bit float format) and stores computed result in dest as a 32-bit float

Format: INV <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst0.channel[n] = 1 / src0.channel[srcCh]
        }
}
```

Precision: 1 ULP

Src->	+inf	+0 / +Denorm	- 0 / -Denorm	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN
Dest – ALT mode		+FLT_MAX	-FLT_MAX		NaN

6.3.2 LOG

Description: Computes Log_2 of Src0 and stores computed result in Dest. Both src0 and dest are 32-bit FP values

Format: LOG <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst0.channel[n] = Log2(src0.channel[srcCh])
        }
}
```

Precision: +/- 2-21 max relative error – Full precision
+ / - 2-10 max relative error- partial precision

Notes: In ALT mode log is computed as Log_2 (abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**-inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**-FLT_MAX**	**-FLT_MAX**		**+F**	**NaN**

6.3.3 EXP

Description: Computes 2^{src0} and stores computed result in Dest. Both src0 and dest are 32-bit FP values

Format: EXP <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst0.channel[n] = 2^(src0.channel[srcCh])
        }
}
```

Precision: + / - 2-21 max relative error – full precision
+/- 2-10 max relative error – partial precision

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**1**	**1**	**0**	**+F**	**NaN**
Dest – ALT mode		**1**	**1**		**+F**	**NaN**

6.3.4 SQRT

Description: Computes square-root of src0 and stores computed result in dest. Both src0 and dest are 32-bit FP values

Format: SQRT <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst0.channel[n] = sqrt(SRC0.channel[srcCh])
        }
}
```

$$dst0.channel[n] = \sqrt{SRC0.\text{channel}[\text{srcCh}]}$$

Precision: 1 ULP

Notes: In ALT mode SQRT is computed as SQRT(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**0**	**-0**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**0**	**0**		**+F**	**NaN**

6.3.5 RSQ

Description: Computes reciprocal square-root of src0 and stores computed result in dest. Both src0 and dest are 32-bit FP values

Format: RSQ <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst.channel[n] = 1/sqrt(SRC0.channel[n])
        }
}
```

$$dst.channel[n] = 1\Big/\sqrt{SRC0.\text{channel}[n]}$$

Precision: 1 ULP

Notes: In ALT mode RSQ is computed as RSQ(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+0**	**+inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**+FLT_MAX**	**+FLT_MAX**		**+F**	**NaN**

6.3.6 POW

Description: Computes abs(src0) raised to the src1 power and stores computed result in dst0. Src0, src1, and dst0 are 32-bit FP values. Src1 is always scalar value.

Format: POW <dst0> <src0> <src1>

Pseudocode:

for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {

$$dst0.channel[n] = 2^{src1 \cdot \log_2(abs(src0.channel[srcCh]))}$$

    }
}

Precision: 2^-15 relative error

IEEE Mode:

Src0->

Src1	abs(F > 1)	abs(F < 1)	abs(+F = = 1)	+inf	+0 / +Denorm	-Denorm / -0	-inf	NaN
+inf	+inf	0	NaN	+inf	0	0	+inf	NaN
+0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F			+inf	0	0	NaN	NaN

ALT Mode:

Src0->

Src1	+F	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
+inf							
+0 / Denorm	1		1	1		1	NaN
-0 / Denorm	1		1	1		1	NaN
-inf							
-F	+F		+FLT_MAX	+FLT_MAX		+F	NaN
NaN			NaN	NaN		NaN	NaN
+F	+F		0	0		+F	NaN

6.3.7 SIN

Description: Computes the sine of src0 (in radians) and stores computed result in dst0. Src0 and dst0 are 32-bit FP values.

Format: SIN <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst.channel[n] = Sin(src0.channel[srcCh])
        }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi
Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+0	-0		-1 to 1	NaN

6.3.8 COS

Description: Computes the cosine of src0 (in radians) and stores computed result in dst0. Src0 and dst0 are 32-bit FP values.

Format: SIN <dst0> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                dst.channel[n] = Cos(src0.channel[srcCh])
        }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi
Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of COS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+1	+1		-1 to 1	NaN

6.3.9 SINCOS

Description: Computes the sine of src0 (in radians) and stores computed result in dst0. Computes the cosine of src0 (in radians) and returns the result to dst1. Src0, dst0 and dst1 are 32-bit FP values.

Format: SINCOS <dst0> <dst1> <src0>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                if(dst0 != NULL){
                        dst0.channel[n] = Sin(src0.channel[srcCh])
                }
                if(dst1 != NULL){
                        dst1.channel[n] = Cos(src0.channel[srcCh])
                }
        }
}
```

Precision: Max absolute error of 0.0008 for the range of +/- 100 * pi.

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SINCOS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Notes: See individual Sin and Cos tables for error handling

6.3.10 INT DIV

Description: Computes src0 divided by src1 and returns an integer result to dst0. Src0, src1 and dst0 are 32-bit integers.

Format: INTDIV <dst0> <dst1> <src0> <src1>

Pseudocode:

```
for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
                if(dst0 != NULL){
                        dst0.channel[n] = quotient (src0.channel[srcCh] / src1.channel[srcCh])
                }
                if(dst1 != NULL){
                        dst1.channel[n] = remainder (src0.channel[srcCh] / src1.channel[srcCh])
                }
        }
}
```

Precision: 32-bit integer

For signed inputs, INT DIV behavior is illustrated by the table below:

Inputs:	Numerator	+	+	-	-
	Denominator	+	-	+	-
Outputs:	Quotient	+	-	-	+
	Remainder	+	+	-	-

IDIV	**SRC0**		
SRC1	**+ INT**	**- INT**	**0**
+ INT	+INT	-INT	0
- INT	-INT	+INT	0
0	Q:0x7FFF FFFF	Q: 0x8000 0000	Q:0x7FFF FFFF
	R:0x7FFF FFFF	R: 0x8000 0000	R:0x7FFF FFFF
UDIV	**SRC0**		
SRC1	**<> 0**	**0**	
<>0	UINT	0	
0	Q: 0xFFFF FFFF	Q: 0xFFFF FFFF	
	R: 0xFFFF FFFF	R: 0xFFFF FFFF	