Intel® HD Graphics OpenSource PRM

Volume 4 Part 2: Subsystem and Cores – Message Gateway, URB, Video Motion, and ISA

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

July 2010

Revision 1.1

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Sandy Bridge chipset family, Havendale/Auburndale chipset family, Intel® 965 Express Chipset Family, Intel® G35 Express Chipset, and Intel® 965GMx Chipset Mobile Family Graphics Controller may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel and the Intel are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Contents

Revision History

Document Number	Revision Number	Description	Revision Date
IHD-OS-022810-R1V4PT2	1.0	First Release.	February 2010
IHD-OS-072810-R1V4PT2	1.1	Update	July 2010

1. Message Gateway

The Message Gateway shared function provides a mechanism for active thread-to-thread communication. Such thread-to-thread communication is based on direct register access. One thread, a **requester thread**, is capable of writing into the GRF register space of another thread, a **recipient thread**. Such direct register access between two threads in a multi-processor environment some time is referred to as **remote register access**. Remote register access may include read or write. GEN4 architecture supports **remote register write**, but not remote register read (natively). Message Gateway facilitates such remote register write via message passing. The requester thread sends a message to Message Gateway requesting a write to the recipient thread's GRF register space. Message Gateway sends a writeback message to the recipient thread to complete the register write on behave of the requester. The requester thread and the recipient thread may be on the same EU or on different EUs.

[**ILK**] Please see Thread Spawn Message Section of Media Chapter for child thread termination using Message Gateway messages with EOT bit set.

1.1 Messages

Message Gateway supports such thread-to-thread communication with the following three messages:

- **OpenGateway:** opens a gateway for a requester thread. Once a thread successfully opens its gateway, it can be a recipient thread to receive remote register write.
- **CloseGateway:** closes the gateway for a requester thread. Once a thread successfully closes its gateway, Message Gateway will block any future remote register writes to this thread.
- **ForwardMsg:** forwards a formatted message (remote register write) from a requester thread to a recipient thread.

1.1.1 Message Descriptor

The following message descriptor applies to all messages supported by Message Gateway.

Bit	Description
16:15	**Notify.** Send Notification Signal. **[Pre-DevSNB]:** When the low bit of this field is set, the recipient thread's notification counter is incremented. The high bit is not part of the shared function specific message descriptor. 00: No notify 01: Increment recipient thread's N0 notification counter 10: Increment recepient thread's N2 notification counter 11: Reserved This field is only valid for a ForwardMsg message. It is ignored for other messages.

Bit	Description
14	**AckReq.** Acknowledgment Required. When this bit is set, an acknowledgment return message is required. Message Gateway will send a writeback message containing the error code to the requester thread using the post destination register address. When this bit is not set, no writeback message is sent to the requesting thread by Message Gateway, even if an error occurs. This field is valid for OpenGateway, CloseGateway, and ForwardMsg messages. When this bit is set, post destination register must be valid and the response length must be 1. When this bit is not set, post destination register must be *null* and the response length must be 0. This bit cannot be set when EOT is set; otherwise, hardware behavior is undefined. 0 = No Acknowledgement is required. 1 = Acknowledgement is required.
13:3	Reserved: MBZ
2:0	**SubFuncID.** Identify the supported sub-functions by Message Gateway. Encodings are: 000 = **OpenGateway.** Open the gateway for the requester thread. 001 = **CloseGateway.** Close the gateway for the requester thread. 010 = **ForwardMsg.** Forward the formatted message to the recipient thread with the given offset from the recipient's register base. 011 = **GetTimeStamp [DevGT+]**. Read absolute and relative timestamps. 100 = **BarrierMsg.** Open the gateway for the requester thread. 101 = **UpdateGatewayState.** Close the gateway for the requester thread. Others are reserved [**Pre-DevSNB**] 011 and 1xx = Reserved

1.1.2 OpenGateway Message

The OpenGateway message opens a communication channel between the requesting thread and other threads. It specifies a key for other threads to access its gateway, as well as the GRF register range allowed to be written. The message consists of a single 256-bit message payload.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completion of the OpenGateway function. Only the least significant DWord in the post destination register is overwritten.

If the EOT is set for this message, Message Gateway will ignore this message; instead, it will close the gateway for the requesting thread regardless of the previous state of the gateway.

It is software's policy to determine how to generate the key.

1.1.2.1 Message Payload

DWord	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
M0.5	31:29	Reserved: MBZ
	28:21	**RegBase:** The register base address to be stored in the Message Gateway. It is used to compute the destination GRF register address from the offset field in ForwardMsg. RegBase contains 256-bit GRF aligned register address. Note 1: This field aligns with bits [28:21] of the Offset field of the message payload for ForwardMsg. Note 2: the most significant bit of this field must be zero. Format = U8 Range = [0,127]
	20:11	Reserved: MBZ
	10:8	**Gateway Size:** The range limit for messages through the Message Gateway. The maximal allowed Gateway Size is 32 GRF registers. 000: 1 GRF Register 001: 2 GRF Registers 010: 4 GRF Registers 011: 8 GRF Registers 100: 16 GRF Registers 101: 32 GRF Registers 110: 64 GRF Registers (**[DevCTG-B+]** only) 111: 128 GRF Registers (**[DevCTG-B+]** only)

DWord	Bit	Description
	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. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4	31:16	Reserved: MBZ
	15:0	Reserved: MBZ
M0.3:0		Ignored

1.1.2.2 Writeback Message

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000: *Successful*. No Error (Normal) 001*: Gateway Size Exceeded*. Attempt to open a gateway with a Gateway Size that is larger than 32 GRF registers (**[Pre-DevCTG-B]** only) 101: *Opcode Error.* Attempt to send a message which is not either open/close/forward other codes: Reserved

1.1.3 CloseGateway Message

The CloseGateway message closes a communication channel for the requesting thread that was previously opened with OpenGateway. Each thread is allowed to have only one open gateway at a time, thus no additional information in the message payload is required to close the gateway. The message consists of a single 256-bit message payload.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completion of the CloseGateway function. Only the least significant DWord in the post destination register is overwritten.

1.1.3.1 Message Payload

DWord	Bit	Description
M0.7:6		Ignored
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. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4:0		Ignored

1.1.3.2 Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000: *Successful*. No Error (Normal) 101: *Opcode Error.* Attempt to send a message which is not either open/close/forward other codes: Reserved

1.1.4 ForwardMsg Message

The ForwardMsg message gives the ability for a requester thread to write a **data segment** in the form of a byte, a dword, 2 dwords, or 4 dwords to a GRF register in a recipient thread. The message consists of a single 256-bit message payload, which contains the specially formatted data segment.

The ForwardMsg message utilizes a communication channel previously opened by the recipient thread. The recipient thread has communicated its EUID, TID, and key to the requester thread previously via some other mechanism. Generally, this is done through the thread spawn message from parent to child thread, allowing each child (requester) to then communicate with its parent through a gateway opened by the parent (recipient). The child could then use ForwardMsg message to communicate its own EUID, TID, and key back to the parent to enable bi-directional communication after opening its own gateway.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requester thread after completion of the ForwardMsg function. Only the least significant DWord in the post destination register is overwritten.

If the Notify bit in the message descriptor is set, a 'notification' is sent to the recipient thread in order to increment the recipient thread's notification counter. This allows multiple messages to be sent to the recipient without waking up the recipient thread. The last message, having this bit set, will then wake up the recipient thread.

1.1.4.1 Message Payload

DWord	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
M0.5	31:29	Reserved: MBZ
	28:16	**Offset:** It provides the destination register position in the recipient thread GRF register space as the offset from the RegBase stored in the recipient thread's gateway entry. The offset is in unit of byte, such that bits [28:21] is the 256-bit aligned register offset and bits [4:0] is the sub-register offset. The sub-register offset must be aligned to the Length field in bits [10:8]. The subfields of Offset are further illustrated as the following. Offset[28:21]: Register offset from the gateway base (Range [0, 127]: bit 12 MBZ) Offset[20:18]: DW offset Offset[17:16]: Byte offset (must be 00 for all DW length cases)
	15:11	Reserved: MBZ
	10:8	**Length:** The length of the data segment. 000: 1 byte 001: 1 word 010: 1 dword 011: 2 dwords 100: 4 dwords 101-111: Reserved

DWord	Bit	Description	
	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. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.	
M0.4	31:28	Ignored	
	27:24	**EUID:** The Execution Unit ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.	
	23:19	Ignored	
	18:16	**TID:** The Thread ID as part of the Recipient field is used to identify the recipient thread to whom the message is forwarded.	
	15:0	**Key** The key to match with the one stored in the recipient thread's entry in Message Gateway. [**DevSNB+**] Ignored	
M0.3	31:0	**Data Segment DWord 3:** valid only for the 4-DWord data segment length	
M0.2	31:0	**Data Segment DWord 2:** valid only for the 4-DWord data segment length	
M0.1	31:0	**Data Segment Dword 1:** valid only for the 2- and 4-DWord data segment lengths	
M0.0	31:24	**Data Segment Byte 0:** the same byte must be copied to all four positions within this DWord. Valid only for the 1-Byte data segment length.	**Data Segment Dword 0:** valid only for the 1-, 2- and 4-Dword data segment lengths
	23:16	**Data Segment Byte 0**	
	15:8	**Data Segment Byte 0**	
	7:0	**Data Segment Byte 0**	

1.1.4.2 Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000: *Successful*. No Error (Normal) 001*:* Reserved 010: *Gateway Closed*. Attempt to send a message through a closed gateway 011: *Key Mismatched*. [Pre-DevSNB] Attempt to send a message with a mismatching key 100: *Limit Exceeded*. [Pre-DevSNB] Attempt to send a message with offset beyond the gateway limit 101: *Opcode Error.* Attempt to send a message which is not either open/close/forward 110: *Invalid Message Size.* Attempt to forward a message with length greater than 4 DW 111: Reserved

1.1.4.3 Writeback Message to Recipient Thread

This message contains the byte or dwords data segment indicated in the message written to the GRF register offset indicated. Only the byte/dword(s) will be enabled, all other data in the GRF register is untouched.

1.1.4.4 Message Payload

DWord	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
M0.5	31	**Return to High GRF**: 0: the return 128-bit data goes to the first half of the destination GRF register 1: the return 128-bit data goes to the second half of the destination GRF register
	30:8	Reserved : MBZ
	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. This field is ignored by Message Gateway This field is only required for a thread that is created by a fixed function (therefore, not a child thread) and EOT bit is set for the message.
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

1.1.4.5 Writeback Message to Requester Thread

As the writeback message is only sent if the **AckReq** bit in the message descriptor is set, **AckReq** bit must be set for this message.

Only half of the destination GRF register is updated (via write-enables). The other half of the register is not changed. This is determined by the **Return to High GRF** control field.

Writeback Message if Return to High GRF is set to 0:

DWord	Bit	Description
W0.7:4		Reserved (not overwritten)
W0.3	31:0	**RelativeTimeLapHigh:** This field returns the MSBs of time lap for the relative clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
W0.2	31:20	**RelativeTimeLapLow:** This field returns the LSBs of time lap for the relative clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ
W0.1	31:0	**AbsoluteTimeLapHigh:** This field returns the MSBs of time lap for the absolute clock since the previous reset. This field represents 1.024 uSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
W0.0	31:20	**AbsoluteTimeLapLow:** This field returns the LSBs of time lap for the absolute clock since the previous reset. This field represents 1/4 nSec increment of the time stamp. Hardware handles the wraparound (over 64 bit boundary) of the timestamp. Format: U12
	19:0	Reserved : MBZ

Writeback Message if Return to High GRF is set to 1:

DWord	Bit	Description
W0.7	31:0	**RelativeTimeLapHigh**
W0.6	31:20	**RelativeTimeLapLow**
	19:0	Reserved : MBZ
W0.5	31:0	**AbsoluteTimeLapHigh**
W0.4	31:20	**AbsoluteTimeLapLow**
	19:0	Reserved : MBZ
W0.3:0		Reserved : MBZ

1.1.5 BarrierMsg Message

The BarrierMsg message gives the ability for multiple threads to synchronize their progress. This is useful when there Writeare data shared between threads. The message consists of a single 256-bit message payload.

Upon receiving one such message, Message Gateway increments the Barrier counter and mark the Barrier requester thread. There is no immediate response from the Message Gateway. When the counter value equates **Barrier Thread Count**, Message Gateway will send response back to all the Barrier requesters.

1.1.5.1 Message Payload

DWord	Bit	Description
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:28	Ignored
	27:24	**BarrierID**. This field indicates which one from the 16 Barrier States is updated. Format: U4 Note: this field location matches with that of R0 header.
	23:0	Ignored
M0.1	31:0	Ignored
M0.0	31:4	Ignored

1.1.5.2 Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000: *Successful*. No Error (Normal) 001*: Error*. Other encodings are reserved

1.1.5.3 Broadcast Writeback Message

When the count for a Barrier (identified by BarrierID) reaches Barrier.Count, Message Gateway broadcasts the following messages to all threads using the BarrierID.

This message contains one single byte written to the GRF register at the *RegBase* location. Only the byte will be enabled, all other data in the GRF register is untouched.

Note: due to the broadcasting nature of the writeback to multiple threads, a fixed 'relative to RegBase' location (with the offset hard coded to zero) is used here, instead of storing offset per request thread.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:8	Reserved (not overwritten)
	7:0	**Barrier.Byte** Format: U8

1.1.6 UpdateGatewayState Message

The UpdateGatewayState message gives the ability for a thread to change the internal state of the Message Gateway.

As Message Gateway may take multiple cycles to send writeback messages with Barrier Byte to multiple requesters, the update of Barrier Byte may be delayed by Message Gateway so that the Barrier Byte delivered to all requesters has the same value. In other words, hardware will block processing new messages when it is in the middle of performing a multi-clock task to avoid risk conditions.

1.1.6.1 Message Payload

DWord	Bit	Description
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:28	Ignored
	27:24	**BarrierID**. This field indicates which one from the 16 Barrier States is updated. Format: U4 Note: this field location matches with that in R0 header.
	23:0	Ignored
M0.1	31:0	Ignored
M0.0	31:20	Ignored
	7:0	**Barrier.Byte** This byte will replace the internal state of Barrier Byte of the Message Gateway. Barrier Byte is initialized initially by MEDIA_VFE_STATE command. Format: U8

1.1.6.2 Writeback Message to Requester Thread

The writeback message is only sent if the **AckReq** bit in the message descriptor is set.

DWord	Bit	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	**Shared Function ID:** Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	**Error Code** 000: *Successful*. No Error (Normal) other codes: Reserved

1.1.7 MMIOReadWrite Message

1.1.7.1 Message Payload

DWord	Bit	Description
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:1	Ignored
	0	**MMIO R/W:** 0 – MMIO Read – a response will be sent to the EU with read data 1 – MMIO Write – no response is sent to EU (unless acknowledge requested in sideband)
M0.2	31:28	Ignored
	22:2	**MMIO Address:** The MMIO DWord address to be accessed.
	1:0	Ignored
M0.1	31:0	Ignored
M0.0	31:4	Ignored

1.1.7.2 Writeback Message to Requester Thread (MMIO Read Only)

DWord	Bit	Description
R0.7	31:0	Ignored
R0.6	31:0	Ignored
R0.5	31:0	Ignored
R0.4	31:0	Ignored
R0.3	31:0	Ignored
R0.2	31:0	Ignored
R0.1	31:0	Ignored
R0.0	31:0	**MMIO Read Data**

2. Unified Return Buffer (URB)

The Unified Return Buffer (URB) is a general-purpose buffer used for sending data between different threads, and, in some cases, between threads and fixed-function units (or vice-versa). A thread accesses the URB by sending messages.

2.1 URB Size

[Pre-DevCTG-B]: The URB provides 16KB of storage, arranged as 512 256-bit *rows*. A row corresponds in size to an EU GRF register. Read/write access to the URB is generally supported on a row-granular basis.

[DevCTG-B]: The URB provides 24KB of storage, arranged as 768 256-bit *rows*. A row corresponds in size to an EU GRF register. Read/write access to the URB is generally supported on a row-granular basis.

[DevILK]: The URB provides 64kB of storage, arranged as 2048 256-bit *rows*. A row corresponds in size to an EU GRF register. Read/write access to the URB is generally supported on a row-granular basis.

A URB *entry* is a logical entity within the URB, referenced by an entry *handle* and comprised of some number of consecutive rows.

2.2 URB Access

The URB can be written by the following agents:

- Command Stream (CS) can write constant data into Constant URB Entries (CURBEs) as a result of processing CONSTANT_BUFFER commands.
- The Video Front End (VFE) fixed-function unit of the Media pipeline can write thread payload data in to its URB entries.
- The Vertex Fetch (VF) fixed-function unit of the 3D pipeline can write vertex data into its URB entries
- GEN4 threads can write data into URB entries via URB_WRITE messages sent to the URB shared function.

The URB can be read by the following agents:

- The Thread Dispatcher (TD) is the main source of URB reads. As a part of spawning a thread, pipeline fixed-functions provide the TD with a number of URB handles, read offsets, and lengths. The TD reads the specified data from the URB and provide that data in the thread payload pre-loaded into GRF registers.
- The Geometry Shader (GS) and Clipper (CLIP) fixed-function units of the 3D pipeline can read selected parts of URB entries to extract vertex data required by the pipeline.
- The Windower (WM) FF unit reads back depth coefficients from URB entries written by the Strip/Fan unit.

Note that neither the CPU nor EU threads can read the URB directly.

2.3 State

The URB function is stateless, with all information required to perform a function being passed in the write message.

See URB Allocation (*Graphics Processing Engine*) for a discussion of how the URB is divided amongst the various fixed functions.

2.4 Messages

There is only one type of message supported by the URB shared function: URB_WRITE. It is primarily used by a thread to write data in to an entry in the URB, as referenced by the passed handle. FF units of the 3D pipeline snoop these messages, and a side effect of the message may be some information being passed to the FF unit which spawned the thread.

This section documents the global aspects of the URB write messages. The actual data contained in the message differs for each fixed function – refer to *3D Pipeline* and the fixed-function chapters or details on 3D URB data formats, *Media* for media-specific URB data formats, and *Graphics Processing Engine* for details on Constant URB Entries (CURBEs).

[**DevILK+**]: The FF_SYNC message is added. See below.

Programming Notes:

- The **End of Thread** bit in the message descriptor may be set on URB messages only in threads dispatched by the geometry shader (GS), clipper, and strips and fans (SF) units.

2.4.1 Execution Mask

The Execution Mask specified in the ‘send’ instruction determines which DWords within each message register are written to the URB.

2.4.2 Message Descriptor

Bit	Description
19	**[DevILK+]: Header Present** This bit must be set to **one** for all URB messages. (**[Pre-DevILK]:** this bit is not part of the shared function specific message descriptor)
18:16	Ignored (**[Pre-DevILK]:** these bits are not part of the shared function specific message descriptor)

Bit	Description
15	**Complete** If clear, this signals that the URB entry(s) referenced by the handle(s) are not yet completely specified. This setting is used to perform partial writes to URB entries, as would be required when writing an entry larger than the maximum single message payload can accommodate. Only the final write would be marked “complete”. Partial writes may be unordered. If set, this signals that there will be no further writes (past this one) to the specific URB entry(s) by the thread. A snooping FF unit uses this to identify when the corresponding URB entry(s) are completely specified, at which point the FF unit can initiate further operations the entry(s) (either a readback, passing the handle(s) down the pipeline, or immediate deallocation if the entry is “unused”). This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. Programming Notes: • The following message descriptor fields are only valid when **Complete** is set: **Used** • The following message header fields are only valid when **Complete** is set: **Handle 0 PrimType**, **Handle 0 PrimStart**, **Handle 0 PrimEnd**.
14	**Used** If set, this signals that the URB entry(s) referenced by the handle(s) are valid outputs of the thread. In all likelihood this means that that entry(s) contains complete & valid data to be subject to further processing by the pipeline. If clear, this signals that the URB entry(s) referenced by the handle(s) are not valid outputs of the thread. Use of this setting will result in the handle(s) being immediately dereferenced by the owning FF unit. This setting is to be used by GS or CLIP threads to dereference handles it obtained (either in the initial thread payload or subsequent allocation writebacks) but subsequently determined were not required (e.g., the object was completely clipped out). Programming Notes: • Only GS and CLIP threads are permitted to utilize Used==0. All other threads are required (by design) to generate valid outputs in all cases. • This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. • This bit is only valid when **Complete** is set, i.e., it is ignored on partial writes.
13	**Allocate** If set, this requests that an additional destination URB entry be allocated to the thread by the spawning FF unit. The FF unit will return the handle to this URB entry via a message writeback operation in response to this message (see writeback format below). Therefore, threads must specify a writeback register in ‘send’ instructions issuing messages with this bit set. If clear, an additional allocation is not requested. Programming Notes: • This bit is strictly control information passed to snooping FF units. The URB shared function itself does not use this bit for any purpose. • This bit is valid on all URB_WRITE messages, e.g., it could be used to allocate a new handle on a partial write (**Complete** not set). • Only one Allocate request (per thread) can be outstanding. Upon requesting an allocation, the thread must wait for the handle to be returned (written back) before another allocation can be requested.
12	**Fast Composite Restriction Check Pass**

Bit	Description
	[DevCTG+]: If set on the end of thread message, this field indicates that the setup kernel portion of the fast composite restriction check has passed. This field is ignored for threads dispatched by units other than Strips and Fans. This field is also ignored unless at least one contiguous dispatch mode is enabled *and* at least one normal dispatch mode is enabled in WM_STATE. **[DevBW] and [DevCL]:** Ignored
11:10	**Swizzle Control.** This field is used to specify which "swizzle" operation is to be performed on the write data. It indirectly specifies whether one or two handles are valid. 00: URB_NOSWIZZLE The message data is to be written directly to a single URB entry (Handle 0). 01: URB_INTERLEAVED The message contains data to be written to two URB entries. The message data provided is interleaved such that the upper DWords (7:4) of each 256-bit unit contain data to be written to Handle 1, and the lower DWords (3:0) contain data to be written to Handle 0. The URB shared function will de-interleave this data and write the two separate data streams to the two entries using the single Offset value (see Offset below for more details). 10: URB_TRANSPOSE This message contains data that is to be "transposed" before being written to the URB. The transpose applied is tailored to the passing of data between the SF and WM stages – it is not a generic transpose. (See description below). Therefore, the assumption is that this mode will only be used by Setup threads, where the setup-result data is swizzled before being written to the URB in order to provide a more optimal format for use in a subsequent PS thread. (See Strip/Fan, Windower chapters). **[DevCTG+]:** If **Transposed URB Read Enable** (WM_STATE) is set, the Setup thread must use URB_NOSWIZZLE to write the coefficient data (it will be transposed whenever the URB is read). URB_TRANSPOSE must only be used when **Transposed URB Read Enable** is clear. See Programming Restrictions in the URB_TRANSPOSE subsection below. 11: Reserved
9:4	**Offset.** This field specifies a destination offset (in 256-bit units) from the start of the URB entry(s), as referenced by **URB Return Handle *n***, at which the data (if any) will be written. When URB_INTERLEAVED is used, this field provides a 256-bit granular offset applied to both URB entry destinations. When URB_TRANSPOSE is used, this field provides a 256-bit granular offset applied to the URB entry destination. The least significant bit of **Offset** must be zero.
3:0	**URB Opcode** 0: URB_WRITE 1: FF_SYNC [**DevILK+**] all other codes are Reserved

The following table lists the valid and invalid combinations of the Complete, Used, Allocate and EOT bits:

Complete	Used	Allocate	EOT	Valid?	Usage
0	d/c	0	0	Valid.	Normal partial-write or non-write of URB.
0	d/c	0	1	Valid only if any and all preceding URB entries have been marked as “complete” and there is no outstanding Allocate request.	Thread terminate w/ non-write of URB
0	d/c	1	0	Valid only if any and all preceding URB entries have been marked as “complete” and there is no outstanding Allocate request.	Non-write of URB with request for an additional handle.
0/1	d/c	1	1	Invalid. Thread must never terminate with an outstanding writeback request.	n/a
1	0	0/1	0	Valid	Dereference of URB entry without/with new allocation request.
1	0	0	1	Valid	Dereference of URB entry and thread termination.
1	1	0/1	0	Valid	Completion of URB entry output without/with new allocation request.
1	1	0	1	Valid	Completion of URB entry output and thread termination.

2.4.3 URB_WRITE

2.4.3.1 URB_WRITE Message Header

DWord	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
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
M0.3	31:0	Ignored
M0.2	31:27	Ignored
	26:16	[**DevILK+]: SONumPrimsWritten Increment Value**. This field contains the value by which the SO_NUM_PRIMS_WRITTEN statistics register will be incremented. [**Pre-DevILK**]: Ignored (SO_NUM_PRIMS_WRITTEN is incremented via SVBWrite messages to the DataPort).
	15:8	Ignored
	7	[**DevCTG-B**]: **Increment CL_INVOCATIONS**: If set, causes the CL_INVOCATIONS register to get incremented by 1 (if enabled). [Otherwise]: Ignored
	6:2	**Handle 0 PrimType.** This field associates a primitive type with the vertex written at Handle 0. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
	1	**Handle 0 PrimStart.** This field is used to indicate that the vertex written at Handle 0 is the first vertex of a primitive. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
	0	**Handle 0 PrimEnd.** This field is used to indicate that the vertex written at Handle 0 is the last vertex of a primitive. NOTE: This field is only defined when the GS or Clipper FF unit is the target FF unit. Otherwise it is Reserved:MBZ.
M0.1	31:16	**Handle ID 1.** This ID is assigned by the fixed function unit and links the work in channel 1 to a specific entry within the fixed function unit. This field is ignored unless **Swizzle Control** indicates Interleave mode.
	15:0	**URB Return Handle 1.** This is the URB handle where channel 1's results are to be placed. This field is ignored unless **Swizzle Control** indicates interleave mode.
M0.0	31:16	**Handle ID 0.** This ID is assigned by the fixed function unit and links the work in channel 0 to a specific entry within the fixed function unit.
	15:0	**URB Return Handle 0.** This is the URB handle where channel 0's results are to be placed.

2.4.3.2 URB_WRITE Message Payload

For the URB message, the message payload will be written to the URB entries indicated by the URB return handles in the message header.

While GS and CLIP threads will write one vertex at a time to the URB, the VS will write two interleaved vertices. The description of the URB write messages will refer to the per-vertex DWords described in the Vertex URB Entry Formats section of the *3D Overview* chapter.

Payload	Usage
URB_NOSWIZZLE	The message payload contains data to be written to a single URB entry (e.g., one Vertex URB entry). The **Swizzle Control** field of the message descriptor must be set to 'NoSwizzle'.
URB_INTERLEAVED	The message payload contains data to be written to two separate URB entries. The payload data is provided in a high/low interleaved fashion. The **Swizzle Control** field of the message descriptor must be set to 'Interleave'.
URB_TRANSPOSE	The message payload contains data that is to be transposed before being written to the URB. See the *Strip & Fan (SF) Unit* chapter for details on the source and destination data layouts and intended usage model.

2.4.3.2.1 URB_NOSWIZZLE

URB_NOSWIZZLE is used to simply write data into consecutive URB locations (no data swizzling or transposition applied).

Programming Notes:

- The URB function will ignore the Channel Enables associated with this message and write all channels into the URB.
- **[DevCTG+]:** When **Transposed URB Read Enable** (WM_STATE) is set, the Setup thread must use URB_NOSWIZZLE to write the coefficient data into the URB (it will be transposed whenever the URB is read).

When URB_NOSWIZZLE is used to write vertex data, the following table shows an example layout of a URB_NOSWIZZLE payload containing one (non-interleaved) vertex containing *n* pairs of 4-DWord vertex elements (where for the example, *n* is >2).

DWord	Bit	Description
M1.7	31:0	**Vertex Data [7]**
M1.6	31:0	**Vertex Data [6]**
M1.5	31:0	**Vertex Data [5]**
M1.4	31:0	**Vertex Data [4]**
M1.3	31:0	**Vertex Data [3]**
M1.2	31:0	**Vertex Data [2]**
M1.1	31:0	**Vertex Data [1]**
M1.0	31:0	**Vertex Data [0]**
M2.7	31:0	**Vertex Data [15]**
M2.6	31:0	**Vertex Data [14]**
M2.5	31:0	**Vertex Data [13]**
M2.4	31:0	**Vertex Data [12]**
M2.3	31:0	**Vertex Data [11]**
M2.2	31:0	**Vertex Data [10]**
M2.1	31:0	**Vertex Data [9]**
M2.0	31:0	**Vertex Data [8]**
Mn.7	31:0	**Vertex Data [8(n-2)+7]**
Mn.6	31:0	**Vertex Data [8(n-2)+6]**
Mn.5	31:0	**Vertex Data [8(n-2)+5]**
Mn.4	31:0	**Vertex Data [8(n-2)+4]**
Mn.3	31:0	**Vertex Data [8(n-2)+3]**
Mn.2	31:0	**Vertex Data [8(n-2)+2]**
Mn.1	31:0	**Vertex Data [8(n-2)+1]**
Mn.0	31:0	**Vertex Data [8(n-2)+0]**

2.4.3.2.2 URB_INTERLEAVED

The following table shows an example layout of a URB_INTERLEAVED payload containing two interleaved vertices, each containing *n* 4-DWord vertex elements (n>1).

Programming Restrictions:

- At least 256 bits per vertex (512 bits total, M1 & M2) must be written. Writing only 128 bits per vertex (256 bits total, M1 only) results in UNDEFINED operation.
- The URB function will use (not ignore) the Channel Enables associated with this message.

DWord	Bit	Description
M1.7	31:0	**Vertex 1 Data [3]**
M1.6	31:0	**Vertex 1 Data [2]**
M1.5	31:0	**Vertex 1 Data [1]**
M1.4	31:0	**Vertex 1 Data [0]**
M1.3	31:0	**Vertex 0 Data [3]**
M1.2	31:0	**Vertex 0 Data [2]**
M1.1	31:0	**Vertex 0 Data [1]**
M1.0	31:0	**Vertex 0 Data [0]**
M2.7	31:0	**Vertex 1 Data [7]**
M2.6	31:0	**Vertex 1 Data [6]**
M2.5	31:0	**Vertex 1 Data [5]**
M2.4	31:0	**Vertex 1 Data [4]**
M2.3	31:0	**Vertex 0 Data [7]**
M2.2	31:0	**Vertex 0 Data [6]**
M2.1	31:0	**Vertex 0 Data [5]**
M2.0	31:0	**Vertex 0 Data [4]**
Mn.7	31:0	**Vertex 1 Data [4(n-2)+3]**
Mn.6	31:0	**Vertex 1 Data [4(n-2)+2]**
Mn.5	31:0	**Vertex 1 Data [4(n-2)+1]**
Mn.4	31:0	**Vertex 1 Data [4(n-2)+0]**
Mn.3	31:0	**Vertex 0 Data [4(n-2)+3]**
Mn.2	31:0	**Vertex 0 Data [4(n-2)+2]**
Mn.1	31:0	**Vertex 0 Data [4(n-2)+1]**
Mn.0	31:0	**Vertex 0 Data [4(n-2)+0]**

2.4.3.2.3 URB_TRANSPOSE

The following table shows an example layout of a URB_TRANSPOSE payload and how the data is transposed and stored in the destination URB entry. Note that Source Row 0, Source Row 1, and implied row of all-zero, and Source Row 3 is transposed and stored in successive 4-DW locations in the destination. This is then repeated for the next 3 rows of the source payload.   For the intended usage model in the Setup thread, Source Row 0 would contain "Cx" coefficients for the first 8 attributes, Source Row 1 would contain "Cy" coefficients for the first 8 attributes, and Source Row 2 would contain "C0" coefficients for the first 8 attributes, then repeating for the next 8 attributes. Insertion of the implied all-zero row is required to align the Cx,Cy and C0 attributes into half-rows within the URB. This permits the used of the "LINE" instruction to initiate attribute interpolation in the subsequent PS thread.

Programming Notes:

- The message payload must contain a multiple of 3 Source Rows of data (excluding the message header).
- The URB function will ignore the Channel Enables associated with this message and write all channels into the URB.
- **[DevCTG+]:** When **Transposed URB Read Enable** (WM_STATE) is set, the Setup thread must use URB_NOSWIZZLE to write the coefficient data into the URB (it will be transposed whenever the URB is read).   URB_TRANSPOSE should only be used when **Transposed URB Read Enable** is clear.

Table 2-1. URB_TRANSPOSE Payload

DWord	Bit	Description
M1.0-7	31:0	**Source Row 0 (e.g., Cx coeffs for the 1st set of 8 attributes)**
M2.0-7	31:0	**Source Row 1 (e.g., Cy coeffs for the 1st set of 8 attributes)**
M3.0-7	31:0	**Source Row 2 (e.g., C0 coeffs for the 1st set of 8 attributes)**
M4.0-7	31:0	**Source Row 3 (e.g., Cx coeffs for the 2nd set of 8 attributes)**
M5.0-7	31:0	**Source Row 4 (e.g., Cy coeffs for the 2nd set of 8 attributes)**
M6.0-7	31:0	**Source Row 5 (e.g., C0 coeffs for the 2nd set of 8 attributes)**
...	31:0	**...**

Table 2-2.URB_TRANSPOSE URB Destination Layout

	URB DW							
URB Row	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
n+0	M3.1	0	M2.1	M1.1	M3.0	0	M2.0	M1.0
n+1	M3.3	0	M2.3	M1.3	M3.2	0	M2.2	M1.2
n+2	M3.5	0	M2.5	M1.5	M3.4	0	M2.4	M1.4
n+3	M3.7	0	M2.7	M1.7	M3.6	0	M2.6	M1.6
n+4	M6.1	0	M5.1	M4.1	M6.0	0	M5.0	M4.0
n+5	M6.3	0	M5.3	M4.3	M6.2	0	M5.2	M4.2
n+6	M6.5	0	M5.5	M4.5	M6.4	0	M5.4	M4.4
n+7	M6.7	0	M5.7	M4.7	M6.6	0	M5.6	M4.6
	...	...	...	...	...	...	...	...

2.4.3.3 Writeback Message for URB Entry Allocate

A writeback only occurs if the **Allocate** bit is set in the message descriptor. A single register is returned containing the URB Return Handle and Handle ID for the allocated handle in the low DWord is returned. All high DWords contain zero.

DWord	Bit	Description
W0.7:1		Reserved : MBZ
W0.0	31:16	**Handle ID.** This ID is assigned by the fixed function unit and links the thread to a specific entry within the fixed function unit.
	15:0	**URB Return Handle.** This is the URB handle where the thread's results are to be placed.

2.4.4 FF_SYNC Messages ([DevILK])

The FF_SYNC message is used to pass critical information between GS/Clip threads and the GS/Clip FF units, as well as providing Gs/Clip thread synchronization (ordering). GS threads report various counts resulting from running the GS and/or SO functions, prior to performing any output (to SOB buffers or to URB handles). Clip threads report only # of handles required. A message response (writeback) length of 1 GRF will be indicated on the 'send' instruction if the thread requires response data and/or synchronization. Refer to the GS/Clip stage chapter for details.

2.4.4.1 FF_SYNC Message Header

Dword	Bit	Description
M0.7	31:0	**Reserved**
M0.6	31:0	**Reserved**
M0.5	31:8	Ignored
	7:0	**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.
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:7	Ignored
M0.1	31:16	Ignored
	15:0	**(GS-only) NumGSPrimsGenerated**. The number of objects (e.g., triangles) generated by the GS function performed by the thread. If the GS function is not enabled, this field MBZ. Format: U16 Range: [0,1024]
M0.0	31:16	**(GS-only) NumSOVertsToWrite**. The number of (expanded-to-list) vertices generated by the SO function performed by the thread. This represents the number of vertices the thread will attempt to write to the SOB(s) in memory, once it obtains the SVBI(s) in the FF_SYNC writeback. Note that overflow may occur either (a) prior to the SVBI(s) are returned in the writeback or (b) in the process of this thread outputting to the SOBs. In either case, the thread needs to check for overflow once it receives the writeback, based on the returned SVBI(s) and the number of vertices it must attempt to output. If the SO function is not enabled, this field MBZ. Format: U16 Range: [0,3066] (1024-vertex tristrip = 1022 triangles = 3066 trilist vertices)
	15:0	**(GS-only) NumSOPrimsNeeded**. The number of objects (e.g., triangles within a trillist) generated by the SO function performed by the thread (exclusive of any SOB overflow). If the SO function is not enabled, this field MBZ. Format: U16 Range: [0,1024]

2.4.4.2 FF_SYNC Writeback Message

(Both GS & Clip): DWord W0.0 of the writeback data contains initial handle information. If **Handle Valid** is clear, the FF unit did not have a handle available to be allocated as the initial handle – the thread will need to use **the URB_WRITE message to obtain the initial handle. Otherwise the Handle ID and URB** Return Handle fields are valid and can be used to write the first VUE.

(GS-only) The writeback data contains the SVBI values to be used as starting write indices by the GS thread. It is the responsibility of the GS thread to perform SOB overflow processing. If the GS thread is not performing StreamOutput and was simply using the writeback to provide GS vertex output synchronization, the return data is to be ignored.

(Clip-only) Dwords W0.1-7 of the writeback data are ignored.

DWord	Bit	Description
W0.7: W0.5	31:0	Reserved

W0.4	31:0	**(GS-only) Streamed Vertex Buffer Index 3**
		This field represents the value of SVBI[3] that is to be used as the starting index for the GS thread. If the thread is not performing StreamOutput, this field is ignored.
		Format = U32
		Range = [0,2^{27}-1]

W0.3	31:0	**(GS-only) Streamed Vertex Buffer Index 2**
W0.2	31:0	**(GS-only) Streamed Vertex Buffer Index 1**
W0.1	31:0	**(GS-only) Streamed Vertex Buffer Index 0**
W0.0	31	**Handle Valid:** If set, the FF unit has provided an initial handle. The other fields in this DWord are valid. If clear, the FF unit did not have an initial handle to provide. The other fields in this DWord are ignored.
	30:24	Reserved
	23:16	**Handle ID.** This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. Format: Reserved for Implementation Use
	15:12	Reserved
	11:0	**URB Return Handle.** This is the initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the first destination URB entry. **[DevILK]** Format: U9 512 bit URB Handle Address

3. Execution Unit ISA

3.1 Introduction

3.1.1 Objective and Scope

The core of GEN architecture consists of an array of multi-threaded processors, also referred to as Execution Units (EU). This Instruction Set Architecture (ISA) document specifies the instructions executable on the EUs of the GEN architecture. It defines the data types in the GEN architecture. It includes the binary format (machine code) and ASCII format (native syntax) of each instruction. It also provides example usages of instructions and modes of instructions, and certain data formats. The programming guideline in appendix provides information to help developers to understand the usage of GEN ISA. However, it is not intended to be a comprehensive tutorial.

3.1.2 Terms and Acronyms

AIP
Application IP. 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.

ARF
Architecture Register File. It is 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.

B
Byte. As a numerical data type of 8 bits, B represents a signed byte integer. It is used to specify the type of an operand in an instruction.

BNF
Backus Naur Form, a formal notation to describe the syntax of a given language. The meta symbols of BNF include “**::=**”, “|”, and “< >”, where “::=” means “is defined as”; “|” means “or”; and angle brackets “<” and “>” are used to surround category names.

CR
Control Register. These read-write registers are used for thread mode control and exception handling for a thread.

D
Double word (DWord). As a fundamental data type, D or DW represents 4 bytes. It may be used to specify the type of an operand in an instruction.

EOT
End Of Thread. This is 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.

EU
Execution Unit. An EU is a multi-threaded processor within the GEN 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 GEN Core.

EUID	Execution Unit Identifier. The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU where a thread is located. A thread can be uniquely identified by the EUID and TID.
ExecSize	Execution Size.
Execution Size	Execution Size indicates the number of data elements processed by a GEN SIMD instruction. It is one GEN instruction field and can be changed at a per instruction level.
FLT_MAX	The magnitude of the maximum represent-able single-precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's.
fmax	Same as FLT_MAX.
GEN Core	Alternative name for an EU in the GEN multi-processor system.
GRF	General Register File. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.
ISA	Instruction Set Architecture. The GEN ISA describes the instructions supported by a GEN EU. A sequence of GEN instructions forms a thread executed on an EU.
JIT	Just-In-Time compiler
LSB	Least Significant Bit
Message	Messages are data packages transmitted from a thread to another thread, to another shared function or to another fixed function. Message passing is the primary communication mechanism of the GEN architecture.
MRF	Message Register File. This is the write-only register space, organized as an array of 256-bit registers, for a thread to communicate with shared functions or other threads.
MSB	Most Significant Bit
DQ	Double Quad word (DQword). As a fundamental data type, DQ represents 16 bytes.
POR	Plan Of Record
QW	Quad Word (QWord). As a fundamental data type, QW represents 8 bytes.
QQ	Quad Quad word (QQword). As a fundamental data type, QQ represents 32 bytes.
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 a 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*.
SIMD	Single Instruction Multiple Data. The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at the instruction level. It can also be used to describe the instructions in such an architecture.
SIP	System IP. 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 certain book-keeping functions and then jumps to SIP.
SR	State Register. The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP.

Thread

A thread is 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 the GEN system may be independent from each other or communicate with each other through the Message Gateway share function.

TID

Thread Identifier. The 2-bit field within a thread state register (SR0) that identifies which out of the four possible thread slots on the EU is executing that thread. A thread can be uniquely identified by the EUID and TID.

TS

Thread Spawner. TS is the second and the last fixed function stage of the media pipeline.

V

Immediate integer vector. As a numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4 bits 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.

VF

Immediate floating point vector. As 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.

W

Word. As a numerical data type of 16 bits, W represents a signed word integer. It is used to specify the type of an operand in an instruction.

URB

Unified Return Buffer. The on-chip memory managed/shared by GEN Fixed Functions. Threads use the URB to return data that will be consumed either by a Fixed Function or other threads.

UB

Unsigned Byte integer. A numerical data type of 8 bits. It may be used to specify the type of an operand in an instruction.

UD

Unsigned Double Word integer. A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.

UW

Unsigned Word integer. A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.

VFE

Video Front End. VFE is the first fixed function stage of the media pipeline.

3.1.3 Formats and Conventions

In order to conveniently (and without ambiguity) describe the register files with 256-bit wide registers that may contain various data types with different data element widths, it is important to use a consistent table format to represent the registers. Throughout this document, we will adopt the following table formats and conventions. When a register or a number is presented by a row, increasing order is always **from right to left** and then **top down** pictorially. In other words, for a bit field, the LSB to MSB is from right to left; for a byte sequence, the least significant byte to the most significant byte is also from right to left. This is consistent with the 'Little Endian' convention used by IA-32 machines. The following tables depict the layout formats for different data units.

7	6	5	4	3	2	1	0	**Bits**
								A Byte

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Bits**
Byte 1								Byte 0								**A Word**

31	24	23	16	15	8	7	0	**Bits**
Byte 4		Byte 2		Byte 1		Byte 0		**A DWord**

31	30	29	..																									3	2	1	0	**32 Bytes**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**16 Words**

7	6	5	4	3	2	1	0	**8 DWords**

7	6	5	4	3	2	1	0	**16 DWords**
15	14	13	12	11	10	9	8	

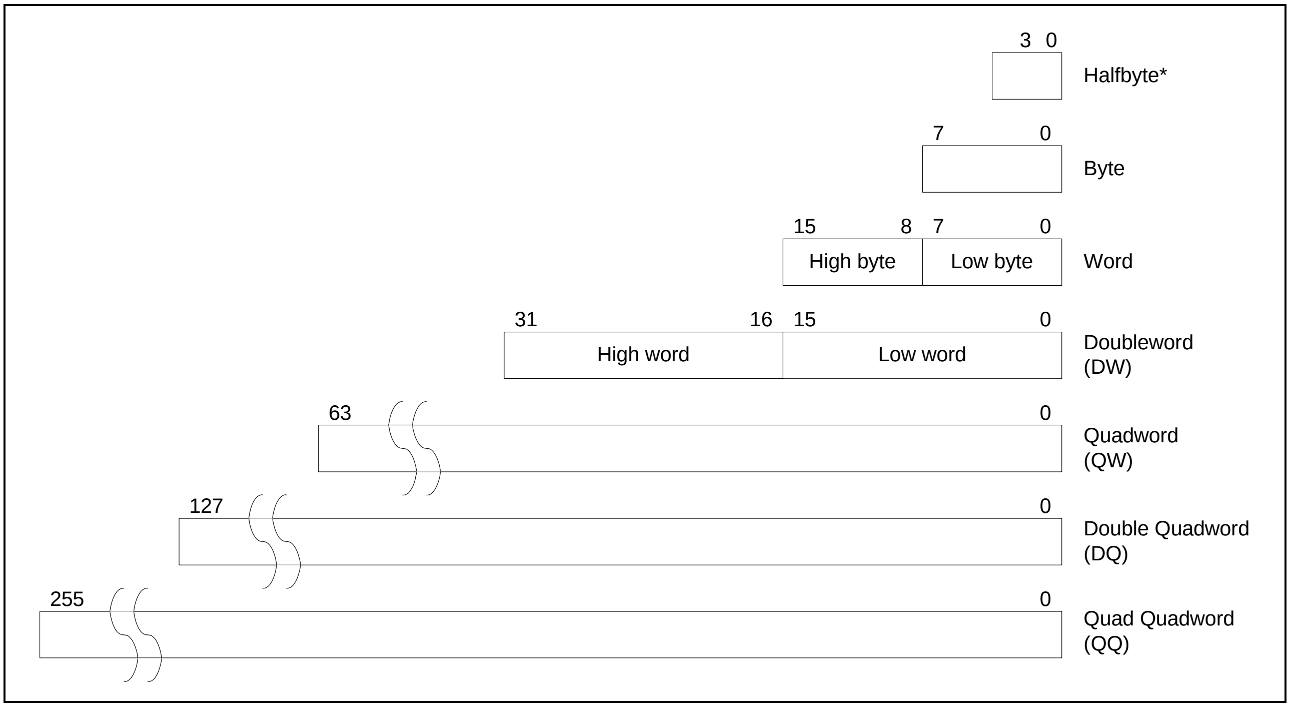
With this convention, we note that the execution channels are logically viewed as from right to left too, which is a little bit unconventional. However, as shown in the *GEN Execution Environment* Chapter, it matches with the bit order of the flag registers. This also impacts the view of a GRF register region, now the region origin is located at the upper-right corner and a region row is viewed from right to left.

4. EU Data Types

4.1 Fundamental Data Types

The fundamental data types in the GEN architecture are halfbyte, byte, word, doubleword (DW), quadword (QW), double quadword (DQ) and quad quadword (QQ). They are defined based on the number of bits of the data type, ranging from 4 bits to 256 bits. As shown in Figure 4-1, a halfbyte contains 4 bits, a byte contains 8 bits, a word contains two bytes, and a doubleword (dword) contains two words, and so on. Halfbyte is a special data type such that it cannot be accessed directly as standalone data element. It is only allowed as a subfield of the numerical data type of "packed signed halfbyte integer vector" described in the next section.

Figure 4-1. Fundamental data types



With the exception of halfbyte, the access of a data element to/from a GEN register or to/from memory must be aligned on the natural boundaries of the data type. The natural boundary for a word has an even-numbered address in unit of byte. The natural boundary for a doubleword has an address divisible by 4 bytes. Similarly, the natural boundary for a quadword, double quadword and quad quadword has an address divisible by 8, 16, and 32 bytes, respectively. Quadword, double quadword and quad quadword do not have corresponding numerical data type. Instead, they are used to describe a group (a vector) of numerical data elements of smaller size align to larger natural boundaries.

4.2 Numerical Data Types

The numerical data types defined in the GEN architecture include signed and unsigned integers and floating-point numbers (floats) of various numbers of bits. These numerical data types are pictorially illustrated in Figure 4-2 and Figure 4-3. **Error! Reference source not found.** details the notation, size and numerical range of each data type. The largest numerical data type has 32 bits.

Figure 4-2. Integer numerical data types

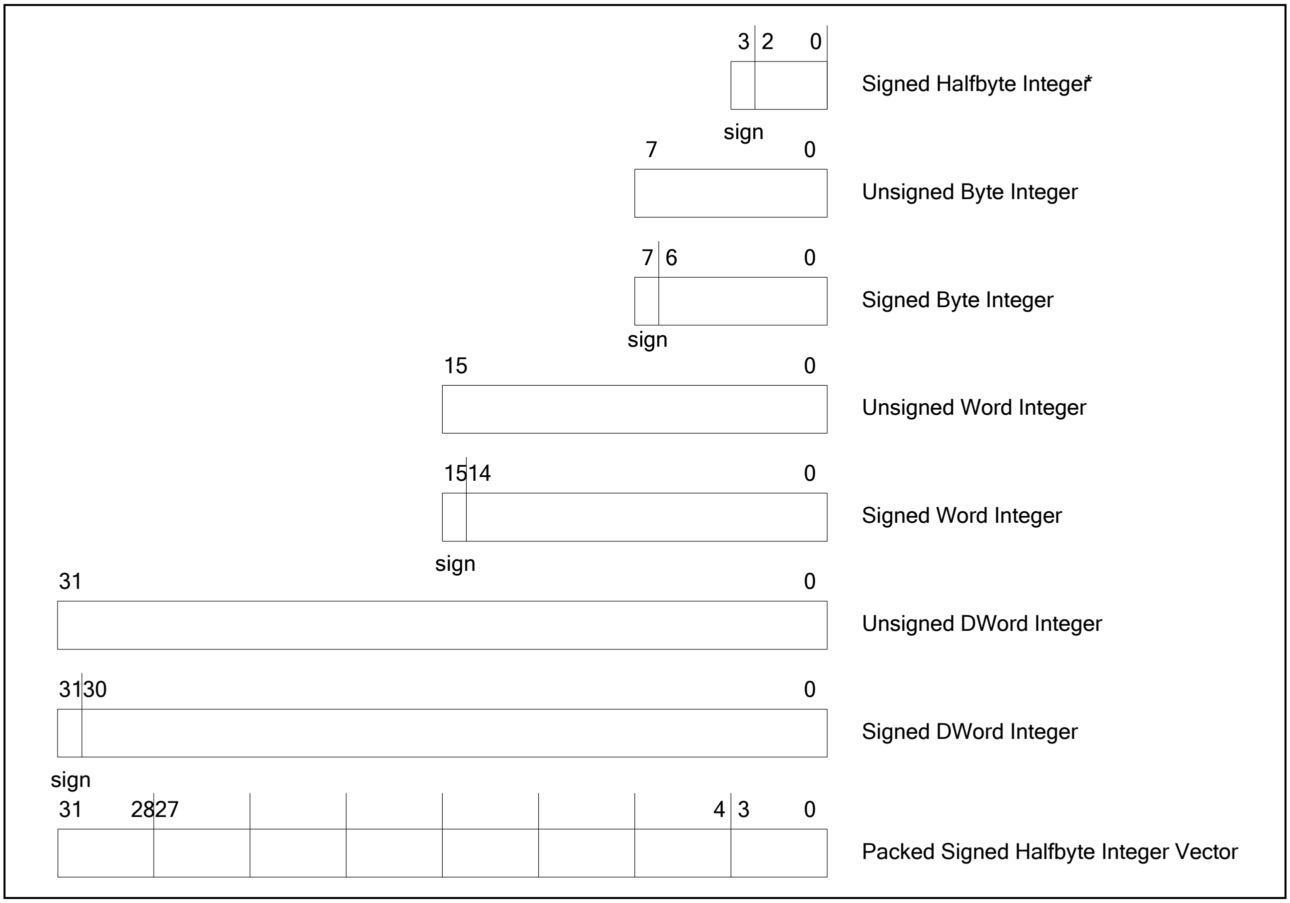
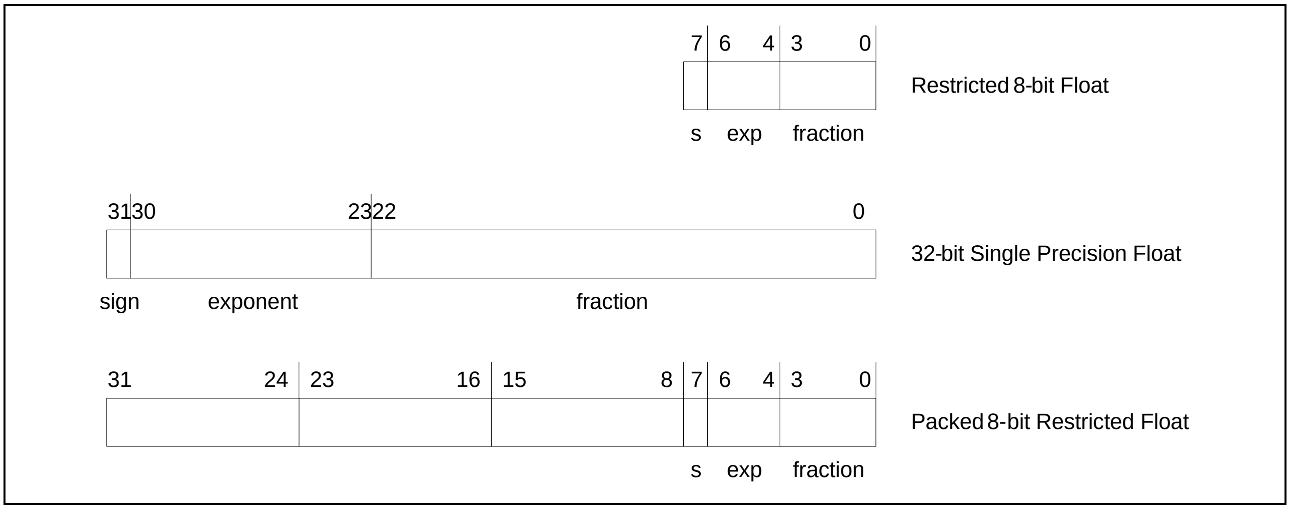


Figure 4-3. Floating point numerical data types



. Formats and ranges of numerical data types

Notation	Numerical Data Types	Fundamental Data Type	Range
UB	**Unsigned Byte Integer**	**Byte**	**[0, 255]**
B	**Signed Byte Integer**	**Byte**	**[-128, 127]**
UW	**Unsigned Word Integer**	**Word**	**[0, 65535]**
W	**Signed Word Integer**	**Word**	**[-32768, 32767]**
UD	**Unsigned Doubleword Integer**	**Doubleword**	$[0, 2^{32} - 1]$
D	**Signed Doubleword Integer**	**Doubleword**	$[-2^{31}, 2^{31} - 1]$
F	**Single Precision Float**	**Doubleword**	$[-(2-2^{-23})^{127} \ldots -2^{-149}, 0.0, 2^{-149} \ldots (2-2^{-23})^{127}]$
n/a	**Signed Halfbyte Integer**	**Halfbyte**	**[–8, 7]**
V	**Packed Signed Halfbyte Integer Vector**	**Doubleword**	**[–8, 7]**
n/a	**Restricted 8-bit Float**	**Byte**	**[–31…–0.125, 0, 0.125… 31]**
VF	**Packed Restricted Float Vector**	**Doubleword**	**[–31…–0.125, 0, 0.125… 31]**

4.2.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, a word or a doubleword. The range for an unsigned byte integer is from 0 to 255. The range for an unsigned word integer is from 0 to 65535. The range for an unsigned doubleword integer is from 0 to $2^{32} – 1$.

The short hand notation for an unsigned byte integer, an unsigned word integer, and an unsigned doubleword integer is **UB**, **UW**, **UD**, respectively.

4.2.2 Signed Integers

Signed integers are signed binary number in 2's compliment form contained in a halfbyte, a byte, a word or a doubleword. A signed halfbyte integer has a numerical range from –8 to 7 with the sign bit at bit 3. A signed byte integer has a range from –128 to 127 with the sign at bit 7. A signed word integer is has a range from -32768 to 32767 with the sign at bit 15. A signed doubleword integer has a range from $–2^{31}$ to $2^{31} – 1$ with the sign at bit 31.

The short hand notation for a signed byte integer, a signed word integer, and a signed doubleword integer is **B**, **W**, **D**, respectively.

4.2.3 Single Precision Floating-Point Numbers

The single precision floating point numbers is contained in a doubleword. Floating point format is as defined in IEEE Standard 754 for Binary Floating-Point Arithmetic. Maximal representable number is $(2–2^{-23})^{127}$ and the minimal number is $– (2–2^{-23})^{127}$. The smallest fractional negative number $–2^{-149}$ and the smallest fractional positive number is 2^{-149}. Value 0.0 has no fractional parts.

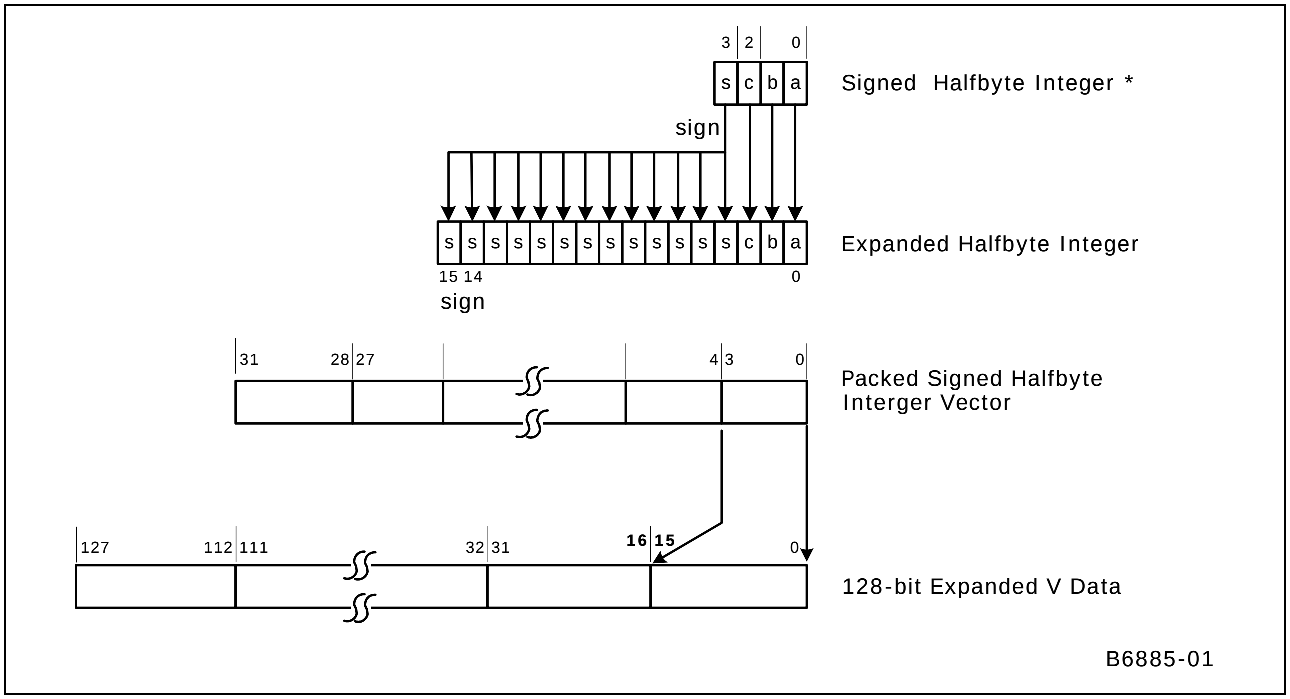
The short hand format notation for a single precision floating-point number is **F**.

4.2.4 Packed Signed Half-Byte Integer Vector

A packed signed halfbyte integer vector consists of 8 signed halfbyte integers contained in a doubleword. Each signed halfbyte integer element has a range from -8 to 7 with the sign on bit 3. This numerical data type is only used by an immediate source operand of doubleword in a GEN instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN hardware converts the 32-bit vector into 8-element signed word vector by sign extension. This is illustrated in Figure 4-4.

The short hand format notation for a packed signed half-byte vector is **V**.

Figure 4-4. Converting a packed half-byte vector to a 128-bit signed integer vector



3 2 0
s c b a
Signed Halfbyte Integer *
sign
s s s s s s s s s s s s s c b a
15 14 0
sign
Expanded Halfbyte Integer
31 28 27 4 3 0
Packed Signed Halfbyte Interger Vector
127 112 111 32 31 16 15 0
128-bit Expanded V Data
B6885-01

4.2.5 Packed 8-bit Restricted Float Vector

A packed restricted float vector consists of 4 8-bit restricted floats contained in a doubleword. Each restricted float has the sign at bit 7, a 3-bit coded exponent in bits 4 to 6, a 4-bit fraction in bits 0 to 3, and an implied integer 1. The exponent is in excess-3 format – having a bias of 3. Restricted float provides zero, positive/negative normalized numbers with a small range (3-bit exponent) and small precision (4-bit fraction). This numerical data type is only used by an immediate source operand of doubleword in a GEN instruction. It cannot be used for the destination operand, or a non-immediate source operand.

Figure 4-5 shows how to convert an 8-bit restricted float into a single precision float. Converting a 3-bit exponent with a bias of 3 to an 8-bit exponent with a bias of 127 is by adding 4, or equivalently copying bit 2 to bit 7 and putting the inverted bit 2 to bits 6:2. A special logic is also needed to take care of positive/negative zeros.

Figure 4-5. Conversion from a Restricted 8-bit Float to a Single-Precision Float

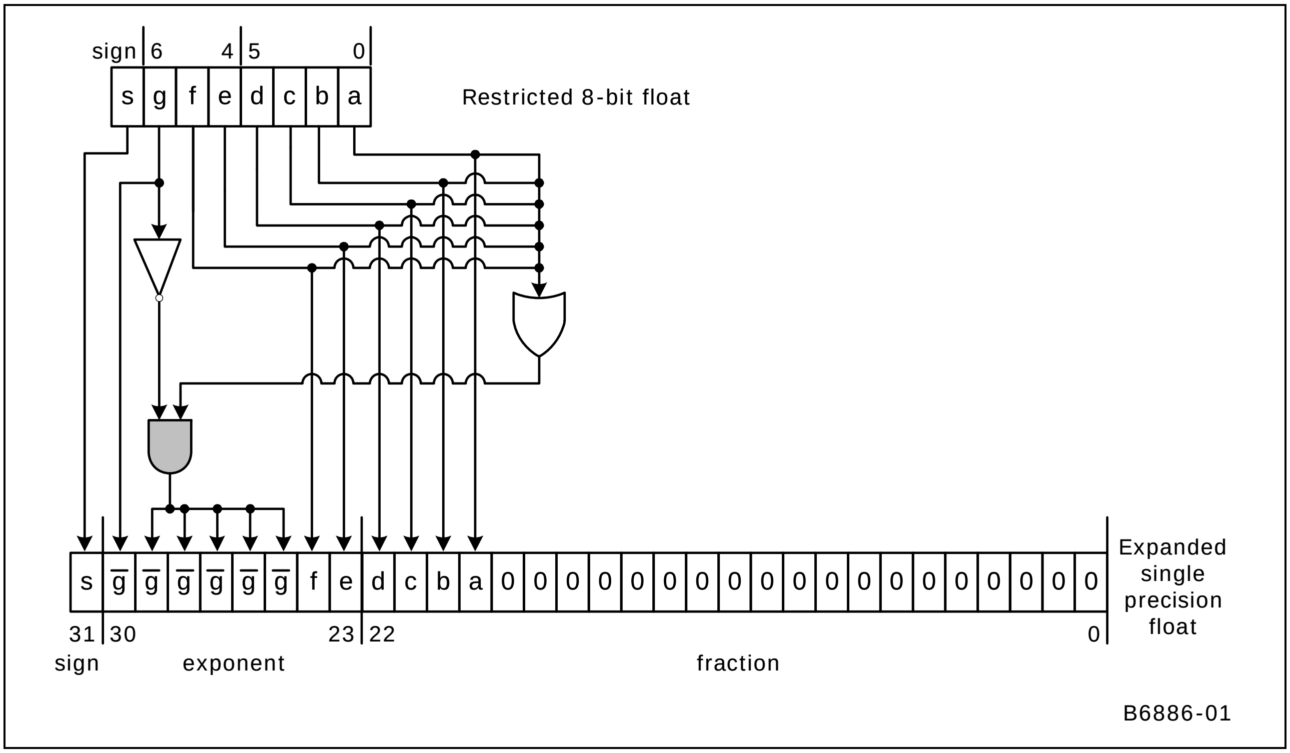


Table 4-1 shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinites, NaN and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range – the magnitude of the smallest non-zero number is 0.125.

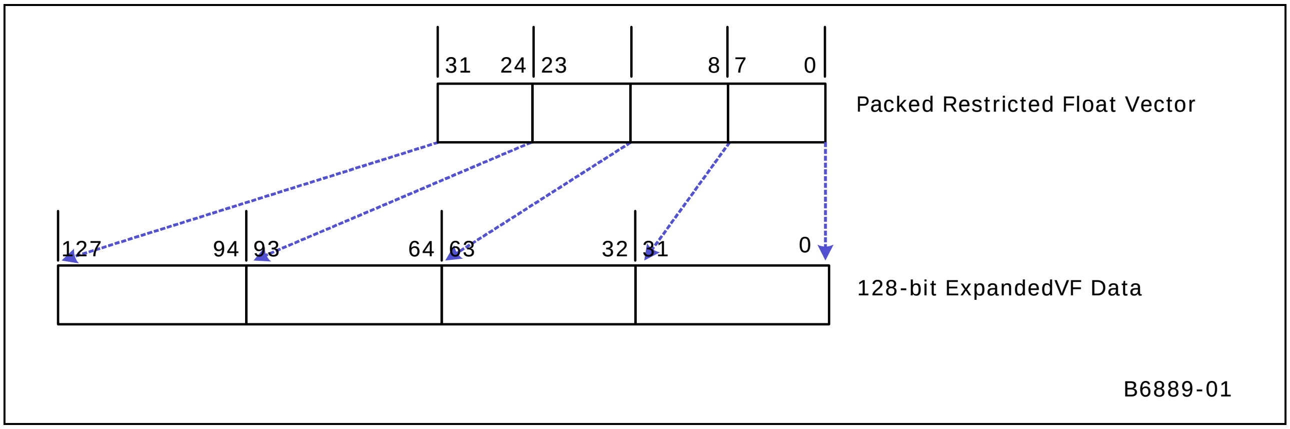
Table 4-1. Example of restricted 8-bit float numbers

Class	Restricted 8-bit Float				xtended 8-bit Exponent	Floating number in decimal
	Hex #	ign [7]	Exponent [6:4]	Fraction [3:0]		
sitive Normalized Float	70-0x7F		**1**	00 … 1111	00 00**11**	… 31
	60-0x6F		**0**	00 … 1111	00 00**10**	.. 15.5
	50-0x5F		**1**	00 … 1111	00 00**01**	.. 7.75
	40-0x4F		**0**	00 … 1111	00 00**00**	.. 3.875
	30-0x3F		**1**	00 … 1111	11 11**11**	.. 1.9375
	20-0x2F		**0**	00 … 1111	11 11**10**	… 0.96875
	10-0x1F		**1**	00 … 1111	11 11**01**	5 … 0.484375
	01-0x0F		**0**	0**1** … 1111	11 11**00**	25 … 0.2421875
	00		**0**	**00**	00 0000	+zero)
gative Normalized Float	F0-0xFF		**1**	00 … 1111	00 00**11**	… -31
	E0-0xEF		**0**	00 … 1111	00 00**10**	… -15.5
	D0-0xDF		**1**	00 … 1111	00 00**01**	… -7.75
	C0-0xCF		**0**	00 … 1111	00 00**00**	… -3.875
	B0-0xBF		**1**	00 … 1111	11 11**11**	… -1.9375
	A0-0xAF		**0**	00 … 1111	11 11**10**	5 … -0.96875
	90-0x9F		**1**	00 … 1111	11 11**01**	25 … -0.484375
	81-0x8F		**0**	0**1** … 1111	11 11**00**	125 … -0.2421875
	80		**0**	**00**	00 0000	(-zero)

Figure 4-6 shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The short hand format notation for a packed signed half-byte vector is **VF**.

Figure 4-6. Converting a Packed Restricted Float Vector to a 128-bit Float Vector



4.3 Floating Point Modes

GEN architecture supports two floating point operation modes, namely IEEE floating point mode (IEEE mode) and alternative floating point mode (ALT mode). Both modes follow mostly the requirements in IEEE-754 but with different deviations. The deviations will be described in details in later sections. The primary difference between these modes is on the handling of Infs, NaNs and denorms. The IEEE floating point mode may be used to support newer versions of 3D graphics API Shaders and the alternative floating point mode may be used to support early Shader versions.

These two modes are supported by all units that perform floating point computations, including GEN execution units, GEN shared functions like Extended Math, the Sampler and the Render Cache color calculator, and fixed functions like VF, Clipper, SF and WIZ. Host software sets floating point mode through the fixed function state descriptors for 3D pipeline and the interface descriptor for media pipeline. Therefore different modes may be associated with different threads running concurrently. Floating point mode control for EU and shared functions are based on the floating point mode field (bit 0) of *cr0* register.

4.3.1 IEEE Floating Point Mode

4.3.1.1 Partial Listing of Honored IEEE-754 Rules

Here is a summary of expected 32-bit floating point behaviors in GEN architecture. Refer to IEEE-754 for topics not mentioned.

- INF – INF = NaN
- 0 * (+/–)INF = NaN
- 1 / (+INF) = +0 and 1 / (–INF) = –0
    - (+/–)INF / (+/–)INF = NaN as A/B = A * (1/B)
- INV (+0) = RSQ (+0) = +INF, INV (–0) = RSQ (–0) = –INF, and SQRT (–0) = –0
- RSQ (–finite) = SQRT (–finite) = NaN
- LOG (+0) = LOG (–0) = –INF, LOG (–finite) = LOG (–INF) = NaN
- NaN (any OP) any-value = NaN with one exception for min/max mentioned below. Resulting NaN may have different bit pattern than the source NaN.

- Normal comparison with conditional modifier of EQ, SNB, GE, LT, LE, when either or both operands is NaN, returns FALSE. Normal comparison of NE, when either or both operands is NaN, returns TRUE.
  - Note: Normal comparison is either a **cmp** instruction or an instruction with conditional modifier
- Special comparison **cmpn** with conditional modifier of EQ, **SNB**, GE, LT, LE, when the second source operand is NaN, returns TRUE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns FALSE. **Cmpn** of NE, when the second source operand is NaN, returns FALSE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns TRUE.
  - This is used to support the proposed IEEE-754R rule on **min** or **max** operations. For which, if only one operand is NaN, min and max operations return the other operand as the result.
- Both normal and special comparisons of any non-NaN value against +/– INF return exact result according to the conditional modifier. This is because that infinities are exact representation in the sense that +INF = +INF and –INF = –INF.
  - NaN is unordered in the sense that NaN != NaN.
- IEEE-754 requires floating point operations to produce a result that is the nearest representable value to an infinitely precise result, known as "round to nearest even" (RTNE). 32-bit floating point operations must produce a result that is within 0.5 Unit-Last-Place (0.5 ULP) of the infinitely precise result. This applies to addition, subtraction, and multiplication.
- All arithmetic floating point instructions does Round To Nearest Even at the end of the computation, except the round instructions.

4.3.1.2 Complete Listing of Deviations or Additional Requirements vs. IEEE-754

For a result that cannot be represented precisely by the floating point format, GEN execution unit uses rounding toward zero (which is a bit-field truncation of the magnitude portion of a floating point data in sign-magnitude form)to nearest even to produce a result to the closest representable value. an infinitely precise result. This ends up with a result that is within 10.5 Unit-Last-Place (1(0.5 ULP) of the infinitely precise result.

- NaN input to an operation obviously always produces NaN on output, however the exact bit pattern of the NaN input to an operation obviously always produces NaN on output, however the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw “mov” instruction which does not alter data at all.)
- x*1.0f must always result in x (except denorm flushed and possible bit pattern change for NaN).
- x +/- 0.0f must always result in x (except denorm flushed and possible bit pattern change for NaN). But -0 + 0 = +0.
- Fused operations (such as mac, dp4, dp3, etc.) may produce intermediate results out of 32-bit float range, but whose final results would be within 32-bit float range if intermediate results were kept at greater precision. In this case, implementations are permitted to produce either the correct result, or else +/-INF. Thus, compatibility between a fused operation, such as “mac”, with the unfused equivalent, “mul” followed by “add” in this case, is not guaranteed.
  - As the accumulator registers have more precision than 32-bit float, any instruction with accumulator as a source/destination operand may produce a different result than that using GRF/MRF registers.
- API Shader divide operations are implemented as x*(1.0f/y). With the two-step method, x*(1.0f/y), the multiply and the divide each independently operate at the 32-bit floating point precision level (accuracy to 1 ULP).
- See the Type Conversion section below for rules on converting to/from float representations.

4.3.1.3 Comparison of Floating Point Numbers

The following tables (Table 4-2 through Table 4-7) detail the rules for floating point comparison. In the tables, “+/-Fin” stands for a positive or negative finite precision floating point number. Result is either a true (T) or false (FALSE or F). Each row corresponds to a fixed <src0> and each column corresponds to a fixed <src1>. When comparing two positive finite numbers (or two negative finite numbers), the result can be T or F depending on the values. Therefore, the corresponding fields in the following tables are marked as T/F.

Table 4-2. Results of “Greater-Than” Comparison – CMP.G

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+0	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+Fin	T	T	T	T	T	T	T/F	FALSE	FALSE
+inf	T	T	T	T	T	T	T	FALSE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-3. Results of “Less-Than” Comparison – CMP.L

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	FALSE
-Fin	FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
-0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-4. Results of “Equal-To” Comparison – CMP.E

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
-0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-5. Results of “Not-Equal-To” Comparison – CMP.NE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	T
-Fin	T	T/F	T	T	T	T	T	T	T
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
-0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin	T	T	T	T	T	T	T/F	T	T
+inf	T	T	T	T	T	T	T	FALSE	T
NaN	T	T	T	T	T	T	T	T	T

Table 4-6. Results of “Less-Than Or Equal-To” Comparison – CMP.LE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	FALSE
-Fin	FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm	FALSE	FALSE	T	T	T	T	T	T	FALSE
-0	FALSE	FALSE	T	T	T	T	T	T	FALSE
+0	FALSE	FALSE	T	T	T	T	T	T	FALSE
+denorm	FALSE	FALSE	T	T	T	T	T	T	FALSE
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-7. Results of “Greater-Than or Equal-To” Comparison – CMP.GE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm	T	T	T	T	T	T	FALSE	FALSE	FALSE
-0	T	T	T	T	T	T	FALSE	FALSE	FALSE
+0	T	T	T	T	T	T	FALSE	FALSE	FALSE
+denorm	T	T	T	T	T	T	FALSE	FALSE	FALSE
+Fin	T	T	T	T	T	T	T/F	FALSE	FALSE
+inf	T	T	T	T	T	T	T	T	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

4.3.1.4 Min/Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in GEN architecture to handle the difference of above mentioned floating point comparison and the rules on supporting MIN/MAX. To compute the MIN or MAX of two floating point numbers, if one of the numbers is NaN and the other one is not, MIN or MAX of the two numbers returns the one that is not NaN. When two numbers are NaN, MIN or MAX of the two numbers returns a NaN, which may not have the same binary form as any of the two numbers.

Mix and Max is achieved using conditional selects, i.e., SEL with ‘condition modifiers’ as :

Evaluations	GEN6 Instructions
MIN(src0, src1) = (src0 < src1) ? src0 : src1	sel.l.f0.0 dst src0 src1
MAX(src0, src1) = (src0 >= src1) ? src0 : src1	sel.ge.f0.0 dst src0 src1

Note even f0.0 is specified in the instruction, the flag register is not touched by this instruction.

The following tables (Table 4-8 through Table 4-13) detail the rules for this special compare-NaN operation for floating point numbers. Notice that excepting “Not-Equal-To” comparison-NaN, last columns in all other tables have ‘T’.

Table 4-8. Results of “Greater-Than” Comparison-NaN – CMPN.G

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	T
-Fin	FALSE	T/F	T	T	T	T	T	T	T
-denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
-0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-9. Results of “Less-Than” Comparison-NaN – CMPN.L

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	T	T	T	T	T	T	FALSE	FALSE	T
-0	T	T	T	T	T	T	FALSE	FALSE	T
+0	T	T	T	T	T	T	FALSE	FALSE	T
+denorm	T	T	T	T	T	T	FALSE	FALSE	T
+Fin	T	T	T	T	T	T	T/F	FALSE	T
+inf	T	T	T	T	T	T	T	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4-10. Results of “Equal-To” Comparison-NaN – CMPN.E

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
-0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+0	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+denorm	FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 4-11. Results of “Not-Equal-To” Comparison-NaN – CMPN.NE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	FALSE	T	T	T	T	T	T	T	**FALSE**
-Fin	T	T/F	T	T	T	T	T	T	**FALSE**
-denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
-0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+0	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+denorm	T	T	FALSE	FALSE	FALSE	FALSE	T	T	**FALSE**
+Fin	T	T	T	T	T	T	T/F	T	**FALSE**
+inf	T	T	T	T	T	T	T	FALSE	**FALSE**
NaN	T	T	T	T	T	T	T	T	**FALSE**

Table 4-12. Results of “Less-Than Or Equal-To” Comparison-NaN – CMPN.LE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	T
-Fin	FALSE	T/F	T	T	T	T	T	T	T
-denorm	FALSE	FALSE	T	T	T	T	T	T	T
-0	FALSE	FALSE	T	T	T	T	T	T	T
+0	FALSE	FALSE	T	T	T	T	T	T	T
+denorm	FALSE	FALSE	T	T	T	T	T	T	T
+Fin	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 4-13. Results of “Greater-Than or Equal-To” Comparison-NaN – CMPN.GE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin	T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm	T	T	T	T	T	T	FALSE	FALSE	T
-0	T	T	T	T	T	T	FALSE	FALSE	T
+0	T	T	T	T	T	T	FALSE	FALSE	T
+denorm	T	T	T	T	T	T	FALSE	FALSE	T
+Fin	T	T	T	T	T	T	T/F	FALSE	T
+inf	T	T	T	T	T	T	T	T	T
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

4.3.2 Alternative Floating Point Mode

The key characteristics of the alternative floating point mode is that NaN, Inf and denorm are not expected for an application to pass into the graphics pipeline, and the graphics hardware must not generate NaN, Inf or denorm as computation result. For example, a result that is larger than the maximum representable floating point number is expected to be flushed to the largest representable floating point number, i.e., +FLT_MAX. The FLT_MAX has an exponent of 0xFE and a mantissa of all one's, which is the same for IEEE floating point mode.

Here is the complete list of the differences of legacy graphics mode from the relaxed IEEE-754 floating point mode.

- Any +/- INF result must be flushed to +/- FLT_MAX, instead of being output as +/- INF.
- Extended mathematics functions of log(), rsq() and sqrt() take the absolute value of the sources before computation to avoid generating INF and NaN results.

Table 4-14 shows the support of these differences in various hardware units.

Table 4-14. Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	Sampler	RC
Any +/- INF result flushed to +/- FLT_MAX	Y	Y	Y	Y	Y	Y	Y	Y
Log, rsq, sqrt take abs() of sources	N/A	N/A	N/A	N/A	N/A	Y	N/A	N/A

Table 5-15 shows some of the desired or recommended alternative floating point mode behaviors that do not have hardware design impact. The reasons it is not necessary to utilize special hardware support for these items are also provided. "**Handling of NaNs, Infs and denorms is undefined. Applications should not pass in such values into the graphics pipeline.**"

Table 4-15. Dismissed legacy behaviors

Suggested IEEE-754 Deviations	Reason for Dismiss
Mov forces (+/-)INF to (+/-)FLT_MAX	(+/-)INF is never present as input
(+/-)INF – (+/-)INF = +/- FLT_MAX instead of NaN	(+/-)INF is never present as input
Denorm must be flushed to zero in all cases (including trivial mov and point sampling)	Denorm is never present as input
Anything*0=0 (including NaN*0=0 and INF*0=0)	NaN and INF are never present as input
Except propagated NaN, NaN is never generated	NaN is never present as input and GEN never generates NaN based on rules in the previous table
An input NaN gets propagated excepting (a)-(d)	NaN is never present as input
(a) Rcp (and rsq) of 0 yields FLT_MAX	N/A, as it is already covered by the general rule "Any +/- INF result flushed to +/- FLT_MAX"
(b) Sampler honors 0/0 = 0 as if (1/0)*0	There is no divide in Sampler
I Rcp (and rsq) of INF yields +/- 0	(+/-)INF is never present as input
(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0	There is no divide in Sampler

4.4 Type Conversion

4.4.1 Float to Integer

Converting from float to integer is based on rounding toward zero. If the floating point value is +0, -0, +Denorm, -Denorm, +NaN –r -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UD, D, UW, W, UB and B).

Input Format	Output Format					
F	UD	D	UW	W	UB	B
+/- Zero	00000000	00000000	00000000	00000000	00000000	00000000
+/- Denorm	00000000	00000000	00000000	00000000	00000000	00000000
NAN	00000000	00000000	00000000	00000000	00000000	00000000
-NAN	00000000	00000000	00000000	00000000	00000000	00000000
INF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-INF	00000000	80000000	00000000	00008000	00000000	00000080
$+2^{32}$ (*)	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-2^{32}-1 (*)	00000000	80000000	00000000	00008000	00000000	00000080

4.4.2 Integer to Integer with Same or Higher Precision

Converting an unsigned integer to a signed or an unsigned integer with higher precision is based on zero extension.

Converting an unsigned integer to a signed integer with the same precision is based on modular wrap-around. Without saturation, a larger than represent-able number becomes a negative number. With saturation, a larger than represent-able number is saturated to the largest positive represent-able number.

Converting a signed integer to a signed integer with higher precision is based on sign extension.

Converting a signed integer to an unsigned integer with higher precision is based on zero extension. Without saturation, a negative number becomes a large positive number with the sign bit wrapped-up. With saturation, a negative number is saturated to zero.

4.4.3 Integer to Integer with Lower Precision

Converting a signed or an unsigned integer to a signed or an unsigned integer with lower precision is based on bit truncation. Without saturation, only the lower bits are kept in the output regardless of the sign-ness of input and output. With saturation, a number that is outside the represent-able range is saturated to the closest represent-able value.

4.4.4 Integer to Float

Converting a signed or an unsigned integer to a single precision float number is to round to the closest representable float number. For any integer number with magnitude less than or equal to 24 bits, resulting float number is a precise representation of the input. However, if it is more than 24 bits, LSBs are truncated. This truncation is performed in sign-magnitude domain, thus, is equivalent to floating point rounding toward zero operation.

5. Execution Environment

5.1 Overview

GEN instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Supports for 3D graphics API (application program interface) Shader instructions are mostly native, meaning that GEN provides efficient execution for Shader programs. Depending on the Shader program operation modes (for example, a Vertex Shader may be executed on a base of a vertex-pair, while a Pixel Shader may be executed on a base of a 16-pixel group), translation from 3D graphics API Shader instruction streams into GEN native instructions may be required. In addition, there are many specific capabilities to accelerate media applications. The following list provides a summary of the GEN instruction set.

- GEN ISA support SIMD (single instruction multiple data) instructions. The number of data elements per instruction depends on the data type.
- GEN ISA supports SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- GEN ISA supports instruction level variable-width SIMD execution.
- GEN ISA supports conditional SIMD execution via destination mask, predication, and execution mask.
- GEN ISA supports instruction compaction.
- A GEN instruction may be executed in multiple cycles over a SIMD execution pipeline.
- Most GEN instructions have three operands. Some instructions have additional implied source and destination operands. Some instructions have explicit dual destinations.
- GEN ISA supports region-based register addressing.
- GEN ISA supports direct and indirect (indexed) register addressing.
- GEN instructions may have a scalar and vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

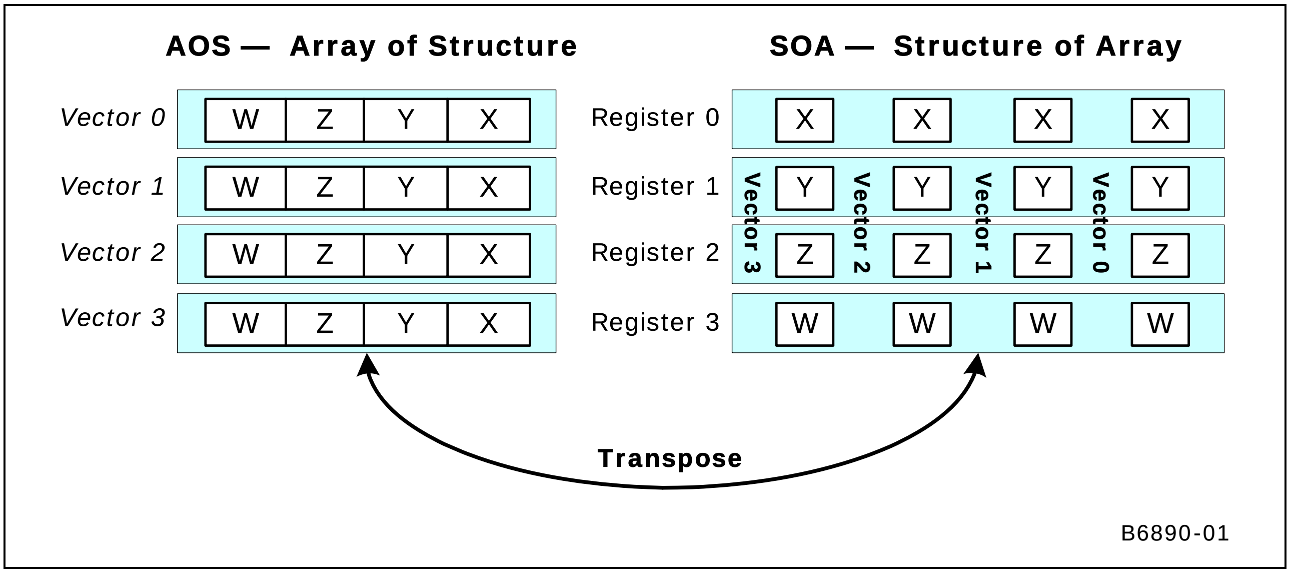
5.2 Primary Usage Models

In describing the usage models of GEN instruction set, it is inevitable to forward reference terminology, syntax and instructions detailed later in this specification. For clarity reasons, not all forward references will be provided in this section as well as subsequent sections. For example, reference to binary instruction fields such as ***Align1, Align16, Compr, SecHalf***, etc, can be found in the Instruction Summary chapter. And assembly instruction syntax can be found in the Instruction Summary chapter and Instruction Reference chapter.

5.2.1 AOS and SOA Data Structures

With the Align1 and Align16 access modes, GEN instruction set provides effective SIMD computation regardless whether data are arranged in array of structure (AOS) form or in structure of array (SOA) form. The AOS and SOA data structures are illustrated by the examples in Figure 5-1. The example shows two different ways of storing four vectors in four SIMD registers. For simplicity, data vector and SIMD register both have four data elements. The four data elements in a vector are denoted by X, Y, Z and W just as for a vertex in 3D geometry. The AOS structure stores one vector in a register and the next vector in another register. The SOA structure stores one data element of each vector in a register and the next element of each vector in the next register and so on. It is obvious in this case the two structures can be related by a matrix transpose operation.

Figure 5-1. AOS and SOA data structures



GEN 3D and media applications take advantage of such broad architecture support and use both AOS and SOA data arrangements.

- Vertices in 3D Geometry (Vertex Shader and Geometry Shader) are arranged in AOS structure and run on SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA structure and run on SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA structure, and occasionally in AOS structure with possible mixed mode of operations that use region-based addressing extensively.

These are preferred methods; alternative arrangements may also be possible. Shared function resources provide data transpose capability to support both modes of operations: The sampler has a transpose for sample reads, the data port has a transpose for render cache writes, and the URB unit has a transpose for URB writes.

The following 3D graphics API Shader instruction will be used in the following sections to illustrate various modes of operations:

add <dst>.xyz <src0>.yxzw <src1>.zwxy

This example is an SIMD instruction that takes two source operands <src0> and <src1>, performs addition operation (add), and store the additions to the destination operand <dst>. Each operand contains four floating point data elements. The data type is determined by the instruction opcode. This instruction also uses source swizzle modifier (.yxzw for <src0> and .zwxy for <src1> and destination mask modifier (.xyz). Please refer to programming specifications of 3D graphics API Shader instructions for more details.

A physical GRF register has 256 bits, which may be used to store 8 floating point data elements. For 3D graphics usage, the mode of operation is (loosely) termed after the data structure as SIMD*mxn*, where "*m*" is a numerical term describing the size of vector and "*n*" is the number of concurrent program flows executed in SIMD.

- Execution with AOS data structures
  - **SIMD4 (short for SIMD4x1)** stands for the mode of operation where a SIMD instruction operates on 4-element vectors stored packed in the registers. There is only one program flow.
  - **SIMD4x2** standards for the SIMD operation based on a pair of 4-element vectors stored in a register. There are effectively two programs running side by side with one vector per program.
- Execution with SOA data structures – also referred to as "channel serial" execution
  - **SIMD8 (short for SIMD1x8)** standards for the SIMD operation based on the SOA data structure where one register contains one data element (the same one) of 8 vectors. Effectively, there are 8 concurrent program flows.
  - **SIMD16 (short for SIMD1x16)** is a special term indicating the use of instruction compression whereas each compressed SIMD instruction operates on a pair of registers that contains one data element (the same one) of 16 vectors. SIMD16 has 16 concurrent program flows.

5.2.2 SIMD4 Mode of Operation

With a register mapping of <src0> to doublewords 0-3 of *r2*, <src1> to doublewords 4-7 of *r2* and <dst> to doublewords 0-3 of *r3*, the example 3D graphics API Shader instruction can be translated into the following GEN instruction:

add (4) r3<4>.xyz:f r2<4>.yzwx:f r2.4<4>.zwxy:f {NoMask}

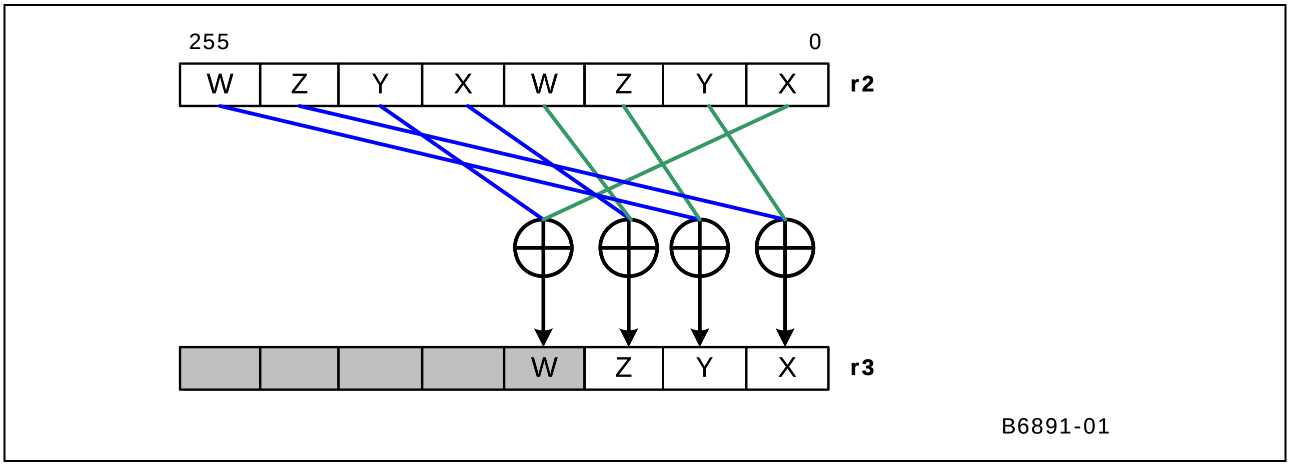
Without diving too much into the syntax definition of a GEN instruction, it is clear that a GEN instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar to the term SIMD Width used in the literature. Each operand is described by the register region parameters such as '<4>' and data type (e.g. "*:f*"). These will be detailed in Section 5.3. The instruction option field, {NoMask}, ensure that the execution occurs for the execution channels shown in the instruction, instead of, possibly, being masked out by the conditional masks of the thread (See Instruction Summary chapter for definition of *MaskCtrl* instruction field).

The operation of this GEN instruction is illustrated in Figure 5-2. In this example, both source operands share the same physical GRF register r2. The two are distinguished by the subregister number. The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 are not modified. In particular, doublewords 4-7 are unchanged as the execution size is 4; doubleword 3 is unchanged due to the destination mask setting.

In this mode of operation, there is only one program flow – any branch decision will be based on a scalar condition and apply to the whole vector of four elements. Option {*NoMask*} ensures that the instruction is not subject to the masks. In fact, most of the instructions in a thread should have {*NoMask*} set.

Even though the execution only performs four parallel add operations, the GEN instruction still executes in 2 cycles (with no useful computation in the second cycle).

Figure 5-2. A SIMD4 Example



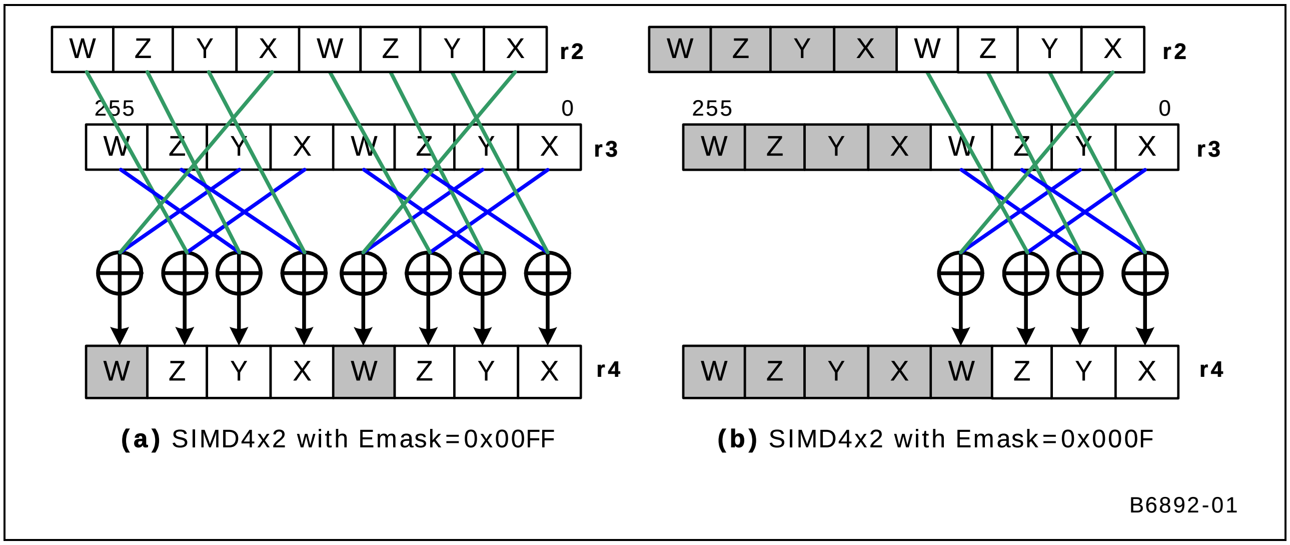
5.2.3 SIMD4x2 Mode of Operation

In this mode, two corresponding vectors from the two program flows fill a GEN physical register. With a register mapping of <src0> to r2, <src1> to r3 and <dst> to r4, the example 3D graphics API Shader instruction can be translated into the following GEN instruction:

add (8) r4<4>.xyz:f r2<4>.yxzw:f r3<4>.zwxy:f

This instruction is subject to the execution mask, which initiated from the dispatch mask. If both program flows are available (e.g. Vertex Shader executed with two active vertices), the dispatch mask is set to 0x00FF. The operation of this GEN instruction is illustrated in Figure 5-3 (a). The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 (doublewords 3 and 7) are unchanged due to the destination mask setting. If only one program flow is available (e.g. the same SIMD4x2 Vertex Shader with only one active vertex), the dispatch mask is set to 0x000F. The operation of the same instruction is shown in Figure 5-3 (b).

Figure 5-3. SIMD4x2 Examples with Different Emasks

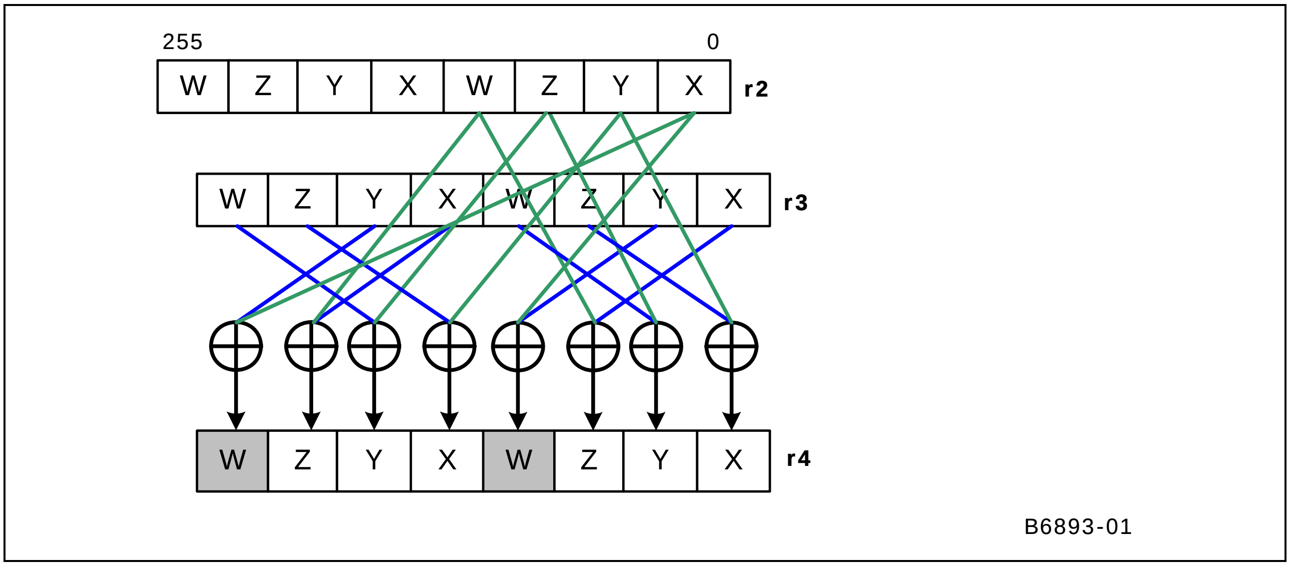


The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2, which is shared by the two program flows. The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

add (8) r4<4>.xyz:f r2<0>.yzwx:f r3<4>.zwxy:f

The only difference here is that the vertical stride of the first source is 0. The operation of this GEN instruction is illustrated in Figure 5-4.

Figure 5-4. A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows



5.2.4 SIMD16 Mode of Operation

With 16 concurrent program flows, one element of a vector would take two GRF registers. In this mode, two corresponding vectors from the two program flows fill a GEN physical register.

With the following register mappings,

<src0>:	*r2-r9 (with 16 X data elements in r2-r3, Y in r4-5, Z in r6-7 and W in r8-9),*
<src1>:	*r10-r17,*
<dst>:	*r18-r25,*

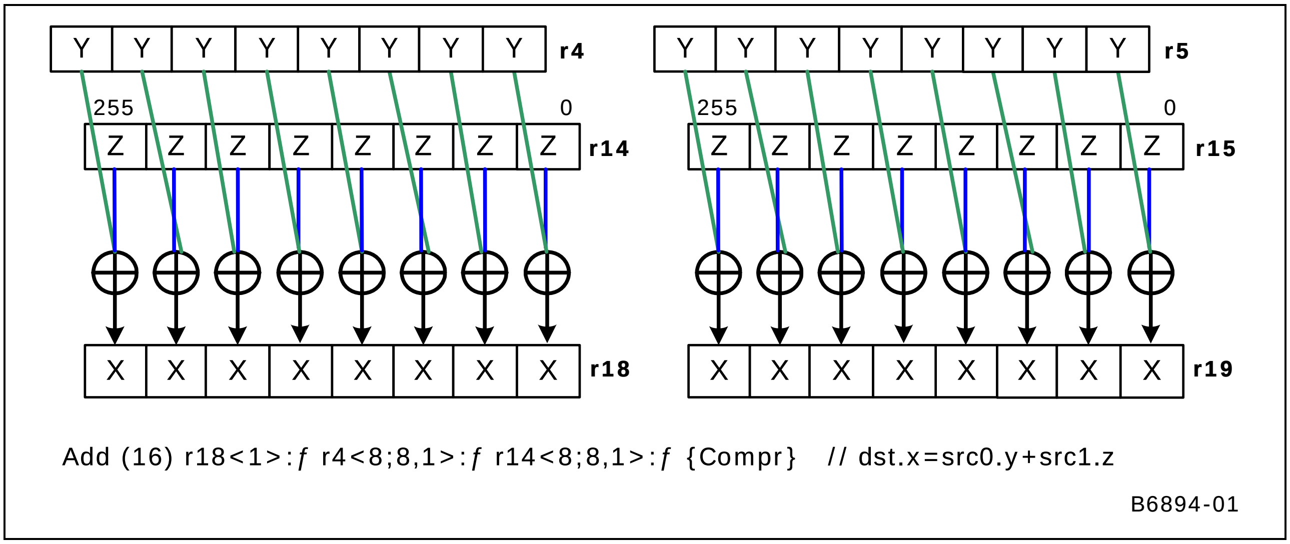
the example 3D graphics API Shader instruction can be translated into the following three GEN instructions:

```
add (16)   r18<1>:f   r4<8;8,1>:f   r14<8;8,1>:f        // dst.x = src0.y + src1.z
add (16)   r20<1>:f   r6<8;8,1>:f   r16<8;8,1>:f        // dst.y = src0.z + src1.w
add (16)   r22<1>:f   r8<8;8,1>:f   r10<8;8,1>:f        // dst.z = src0.w + src1.x
```

The three GEN instructions correspond to the three enabled destination masks As there is no output for the W elements of <dst>, no instruction is needed for that element. The first instruction inputs the Y elements of <src0> and the Z elements of <src1> and outputs the X elements of <dst>. The operation of this instruction is shown in Figure 5-5.

With the number of program flows more than one, the above instructions also subject to execution mask. The 16-bit dispatch mask is partitioned into four groups with four bits each. For Pixel Shader generated by the Windower, each 4-bit group corresponds to a 2x2 pixel subspan. If a subspan is not valid for a Pixel Shader instance, the corresponding 4-bit group in the dispatch mask is not set. Therefore, the same instructions can be used independent of the number of available subspans without creating bogus data in the subspans that are not valid.

Figure 5-5. A SIMD16 Example



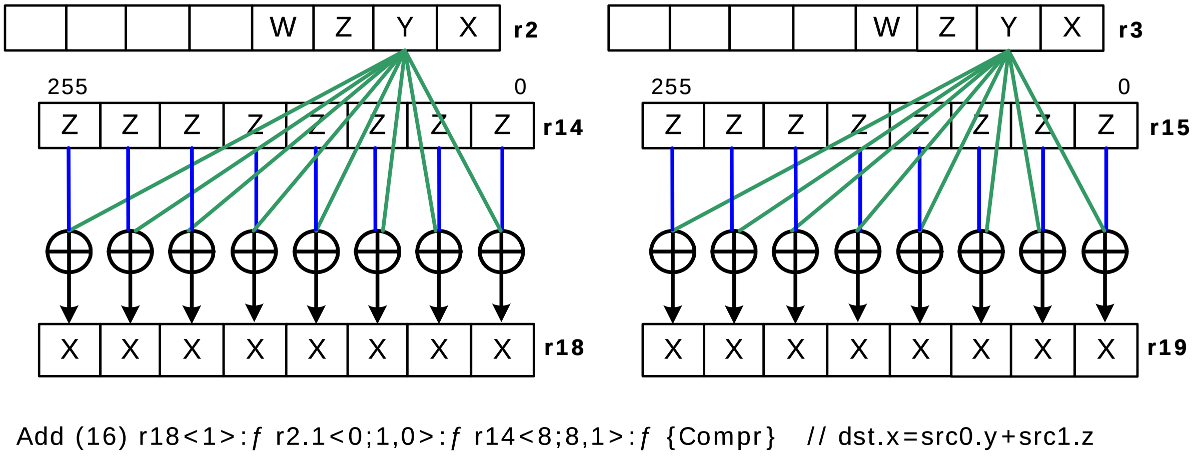
Add (16) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f {Compr}   // dst.x=src0.y+src1.z

Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS format). The example 3D graphics API Shader instruction can then be translated into the following GEN instruction:

```
add (16)   r18<1>:f   r2.1<0;1,0>:f   r14<8;8,1>:f   {Compr}        // dst.x = src0.y + src1.z
add (16)   r20<1>:f   r2.2<0;1,0>:f   r16<8;8,1>:f   {Compr}        // dst.y = src0.z + src1.w
add (16)   r22<1>:f   r2.3<0;1,0>:f   r10<8;8,1>:f   {Compr}        // dst.z = src0.w + src1.x
```

The register region of the first source operand represents a replicated scalar. The operation of the first GEN instruction is illustrated in Figure 5-6.

Figure 5-6. Another SIMD16 Example with an AOS Shared Constant



B6895-01

5.2.5 SIMD8 Mode of Operation

Each compressed instruction has two correspond uncompressed instructions. Taking the example instruction shown in Figure 5-6, it is equivalent to the following two instructions.

```
add (8)   r18<1>:f   r4<8;8,1>:f   r14<8;8,1>:f                    // dst.x[7:0] = src0.y + src1.z

add (8)   r19<1>:f   r5<8;8,1>:f   r15<8;8,1>:f   {SecHalf}        // dst.x[15:8] = src0.y + src1.z
```

Therefore, SIMD8 can be viewed as a special case for SIMD16.

There are other reasons that SIMD8 instructions may be used. Within a program with 16 concurrent program flows, some time SIMD8 instruction must be used due to architecture restrictions. For example, the address register a0 only have 8 elements, if an indirect GRF addressing is used, SIMD16 instructions are not allowed.

5.3 Registers and Register Regions

5.3.1 Register Files

GEN registers are grouped into different name spaces called register files. There are three different register files defined: General Register File, Message Register File, and Architecture Register File. In addition, immediate operands also have a unique encoding of the register file field, even though they come inline in the instruction word and do not have dedicated physical storages.

- General Register File (GRF): GRF contains general-purpose read-write registers.
- Message Register File (MRF): MRF contains special purpose registers used for message passing only. MRF registers are write-only.
- Architecture Register File (ARF): ARF contains all other architectural registers, including the address registers (a#), accumulators (*acc#*), flags (*f#*), masks (*mask#*), mask stack (*ms#*), mask stack depth (*msd#*), notification count (*n#*), instruction pointer (*ip*), etc. Null register (*null*) is also encoded as an ARF register.
- Immediate: Certain instructions take immediate terms as the source operands. These immediate terms have a distinct register file encoding.

Each thread executed in an EU has its own thread context, i.e. dedicated register space, which is not shared between threads executing on a common EU or on a different EU. In the rest of the Chapters, register access are in respect to a given thread.

5.3.2 GRF Registers

Number of Registers:	Various
Default Value:	None
Normal Access:	RW
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Index-ability:	Yes

Registers in the General Register File are the most commonly used read-write registers. During the execution of a thread, GRF registers are used to store the temporary data, and serve as the destination to receive data from shared function units (and some times from a fixed function unit). They are also used to store the input (initialization) data when a thread is created. By allowing fixed function hardware to initialize some portion of GRF registers during thread dispatch time, GEN architecture can achieve better parallelism. A thread's execution efficiency can also be improved as some data are already in the register to be executed upon. Besides these registers containing thread's payload, the rest of GRF registers of a thread are not initialized.

Table 5-1. Summary of GRF Registers

Register File	Register Name	Description
General Register File (GRF)	***r#***	General purpose read write registers

Each execution unit has a fixed size physical GRF register RAM. The GRF register RAM is shared by all threads on the EU. GRF space for a thread is allocated at thread dispatch time, allowing the amount of GRF space to adapt to the need of a given thread.

Mapping of a thread's GRF registers to the physical GRF RAM is through a translation table. Therefore, a thread's access to GRF is always through the 0-based logical view. For example, the GRF registers are *r0* through *r127*.

GRF registers can be accessed using region-based addressing at byte granularity (both read and write). A source operand must be contained within two adjacent physical registers. A destination operand must be contained within one physical register. GRF registers support direct addressing and register-indirect addressing. Register-indirect addressing uses the address registers (ARF registers a#) and an immediate address offset value.

When accessing (read and/or write) outside the GRF register range allocated for a given thread either through direct or indirect addressing, the result is unpredictable.

Table 5-2. GRF Registers Available in Device Hardware

Device	Physical Register Size	Allocation Granularity	Number per Thread	Number per EU
[DevSNB]	256 bits	Fixed allocation of 128 register	128 registers	640 registers

5.3.3 MRF Registers

Number of Registers:	Fixed
Default Value:	None
Normal Access:	WO
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Index-ability:	Yes

Registers in the Message Register File are used to store the header and payload for out-going messages from a thread to a shared function such as the Sampler. There are fixed number of MRF registers for each thread.

MRF registers are write-only, and therefore, can only be the destination operand of an instruction.

MRF registers support write-enable at byte granularity. When an MRF register is used as the current destination operand of the *send* instruction, only 256-bit register aligned access is supported.

When accessing (write) outside the MRF register range for a given thread, the result is unpredictable.

Table 5-3. Summary of MRF Registers

Register File	Register Name	Description
Message Register File (MRF)	***m#***	Special purpose output write-only registers

Table 5-4. MRF Registers Available in Device Hardware

Device	Physical Register Size	Number per Thread	Indirect Addressing?
[DevSNB]	256 bits	24 registers	Yes

Note for Programmers:. *Normal thread should access MRF starting at m1.*

5.3.4 ARF Registers

5.3.4.1 Overview

Besides GRF and MRF registers that are directly indicated by unique register file coding, all other registers belong to the general Architecture Register File (ARF). Encoding of architecture register types are based on the MSBs of the register number field, RegNum, in the instruction word. RegNum field has 8 bits. The 4 MSBs, RegNum[7:4], represent the architecture register type. This is summarized in Table 5-5**.**

Table 5-5. Summary of Architecture Registers [DevSNB+]

Register Type (RegNum [7:4])	Register Name	Register Count	Description
0000	***null***	*1*	Null register
0001	***a0.#***	*1*	Address register
0010	***acc#***	***2***	Accumulator register
0011	***f0.#***	*1*	Flag register
0100-0110	*reserved*		
0111	***sr0.#***	*1*	State register
1000	***cr0.#***	*1*	Control register
1001	***n#***	*2*	Notification count register
1010	***ip***	*1*	Instruction pointer register
1011	***tdr***	*1*	Thread dependency register
1100	***performance***	*1*	*Performace register*
1101-1111	*reserved*		

The remaining register number field RegNum[3:0] is used to identify the register number of a given architecture register type. Therefore, maximum number of registers for a given architecture register type is limited to 16. The subregister number field, SubRegNum, in instruction word has 5 bits. It is used for addressing subregister region for an architecture register supporting register subdivision. SubRegNum field is in unit of byte. Therefore, maximum number of bytes of an architecture register is limited to 32. Depending on alignment restriction of a register type, only certain encodings of SubRegNum field is applicable for an architecture register. The detailed definitions are provided in the following sections.

5.3.4.2 Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF and MRF. The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to control to generate the destination write mask at byte level.

A special restriction on region-based addressing for ARF is that the register region cannot cross register boundary. This rule in fact only applies to the accumulator as it is the only ARF register containing multiple registers (two).

Subregister fields of an ARF register may not all populated (indicated by the subregister indicated as reserved). Write to an unpopulated subregister will be dropped, there is no side effect. Read from an unpopulated subregister, if not specified, will return unpredictable data.

Some of ARF registers are read-only. Write to a read-only ARF register is dropped and there is no side effect.

5.3.4.3 Null Register

ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control:	N/A
Index-ability:	No

The null register is a special encoding for an operand that does not have physical map. It is primarily used in the instruction to indicate the non-existence of an operand. Write to the null register has no side effect. Read from the null register returns undefined result.

The null register can be used in the place when a source operand is absent. For example, for a single source operand instruction such as MOV, NOT, the second source operand <src1> must be a null register.

When the null register is used as the destination operand of an instruction, it indicates the computed results are not stored in any physical registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, update to the selected flag register also happens. In this case, the register region fields of the 'null' operand are valid.

Another example use is to use the null register as the posted destination of a *send* instruction for data write to indicate that there is no write completion acknowledgement required. In this case, however, the register region fields are still valid. The null register can also be the first source operand for a send instruction indicating the absent of the implied move. See *send* instruction for details.

5.3.4.4 Address Register

ARF Register Type Encoding (RegNum[7:4]):	0001b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	8
Element Size:	16 bits
Element Type:	UW or UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	N/A
Index-ability:	No

There are eight address subregisters forming an 8-element vector. Each address subregister contains 16 bits. Address subregisters can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the *send* instruction.

Table 5-6. Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***a0*** All other encodings are reserved.	When register a0 or subregisters in a0 is used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned. 00000 = **a0.0:uw** 00010 = **a0.1:uw** 00100 = **a0.2:uw** 00110 = **a0.3:uw** 01000 = **a0.4:uw** 01010 = **a0.5:uw** 01100 = **a0.6:uw** 01110 = **a0.7:uw** All other encodings are reserved. However, when register a0 or subregisters in a0 is an explicit source and/or destination register, other data types are allowed as long as the register region falls in the 128-bit range.

Table 5-7. Address Register Fields

Dword	Bits	Subfield Description
3	31:16	**Address subregister a0.7:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.6:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
2	31:16	**Address subregister a0.5:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.4:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
1	31:16	**Address subregister a0.3:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	**Address subregister a0.2:uw.** This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
0	31:16	**Address subregister a0.1:uw.** This field can be used for register-indirect register addressing or serve as message descriptor for *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For *send* instruction, it provides the higher 16 bits of <desc>. Format: U12 or U16.
	15:0	**Address subregister a0.0:uw.** This field can be used for register-indirect register addressing or serve as message descriptor for *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For *send* instruction, it provides the lower 16 bits of <desc>. Format: U12 or U16.

When used as a source or destination operand, the address subregisters can be accessed individually or as a group. In the following example, the first instruction moves all 8 address subregisters to the first half of GRF register r1, the second instruction replicates a0.4:uw as an unsigned word to the second half of r1, the third instruction moves the first 4 words in r1 into the first 4 address subregisters, and the fourth instruction replicates r1.4 as a unsigned word to the last 4 address subregisters.

```
mov (8) r1.0<1>:uw a0.0<8;8,1>:uw     // r1.n = a0.n for n = 0 to 7 in words

mov (8) r1.8<1>:uw a0.4<0;1,0>:uw     // r1.m = a0.4 for m = 8 to 15 in words

mov (4) a0.0<1>:uw r1.0<4;4,1>:uw     // a0.n = r1.n for n = 0 to 3 in words

mov (4) a0.4<1>:uw r1.4<0;1,0>:uw     // a0.m = r1.4 for m = 4 to 7 in words
```

When used as the register-indirect addressing for GRF registers, the address subregisters can be accessed also individually or in group. When accessed in group, the address subregisters must be group-aligned. For example, when two address subregisters are used for register indirect addressing, they must be aligned to even address subregisters. In the following example, the first instruction is legal. However, the second one is not. As ExecSize = 8 and the width of <src0> is 4, two address subregisters will be used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address subregisters are a0.2:uw and a0.3:uw. The two align to a dword group in the address register. However, the two address subregisters for the second instruction are a0.3:uw and a0.4:uw. They are not dword aligned in the address register and therefore violate the above mentioned alignment rule.

```
mov (8) r1.0<1>:d r[a0.2]<4,1>:d        // a0.2 and a0.3 is used for src1

mov (8) r1.0<1>:d r[a0.3]<4,1>:d                // ILLEGAL use of register indirect
```

Implementation restriction: *GEN ISA supports per channel indexing for a source operand. As there are only 8 sub-fields in the address register (to save hardware cost), the execution size of an instruction using per-channel indexing is limited to 8. Software may reload the address register and use compression control* ***SecHalf*** *to complete a 16-channel computation.*

Implementation restriction: *When used as the source operand <desc> for the send instruction, only the first dword subregister of a0 register is allowed (i.e. a0.0:ud, which can be viewed as the combination of a0.0:uw and a0.1:uw). In addition, it must be of UD type and in the following form <desc> = a0.0<0;1,0>:ud.*

Implementation restriction: *Elements a0.0 and a0.1 have 16 bits each, but the rest of elements (a0.2:uw through a0.7:uw) only have 12 bits populated each. 12-bit precision supports full indirect-addressing capability for the largest GRF register range. Software must observe the asymmetry of the implementation. When a0.0:uw and a0.1:uw are the source or destination, full 16-bit precision is preserved. However, when a0.2:uw to a0.7:uw are the destination, the higher 4 bits for each element will be dropped; when they are the source, hardware inserts zero to the higher 4 bits for each element.*

Performance Note: *There is only one scoreboard for the whole address register. When a write to some subregisters is in flight, hardware will stall any instruction writing to other subregisters. Software may use the destination dependency control {NoDDChk, NoDDClr} to improve performance in this case. Similarly, when a write to some subregisters is in flight, hardware will stall any instruction sourcing other subregisters until the write retires.*

5.3.4.5 Accumulator Registers

ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	2
Default Value:	None
Normal Access:	RW
Elements:	8 or 16
Element Size:	Various
Element Type:	Various
Access Granularity:	Word
Write Mask Granularity:	N/A
SecHalf Control:	Yes
Index-ability:	No

There are two accumulator registers, *acc0* and *acc1*. They can be accessed either as explicit source and/or destination registers or as implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 doublewords or 16 words of data elements. However, as shown in

Table 5-9, each data element may have higher precision with additional guard bits than that indicated by the numerical data type.

Table 5-8. Register and Subregister Numbers for Accumulate Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***acc0*** 0001 = ***acc1*** All other encodings are reserved.	Reserved: MBZ The accumulator subfields are individually addressable at word granularity. When an accumulator register is an explicit destination, it follows the rules for a destination register. If an accumulator is an explicit source operand, its register region must match with that of the destination register. The precision for Accumulator for floating point is the exact same as a regular GRF register.

The accumulators will be updated implicitly only if the ***AccWrCtrl*** is on for the current instruction. The accumulator Disable in control register cr0.0 allows software to over-write ***AccWrCtrl*** control for implicit accumulator update. . The write enable in word granularity for the instruction will be used to update the Accumulator, the data in the disabled channels will not be updated.

When an accumulator register is used as an implicit source or destination operand, it is acc0 by default. For a SIMD16 DW/float instruction, both acc0 and acc1 are used. If ***SecHalf*** is set, the implicit accumulator is then acc1.

It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). Result of such instruction is unpredictable.

For a SIMD16 DW/float instruction, if an accumulator register is used as an explicit source or destination operand, it must be acc0.

When an accumulator register is used as an explicit source operand, it must be the first source operand <src0>.

Whether the accumulator register is updated for a given instruction depends on several conditions: it can be an explicit destination operand, it can be an implicit destination if the ***AccWrCtrl*** is set in the instruction. Swizzling is not allowed when accumulator is used as implicit source or explicit source of an instruction

Implementation Precision Restriction: *As there are only 64 bits per channel in dword mode (D and UD), it is sufficient to store multiplication result of two dword operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The dword multiplication results will be 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.*

Implementation Precision Restriction: *As there are only 33 bits per channel in word mode (W and UW), it is sufficient to store multiplication result of two word operands with and without source modifier as the result is up to 33 bits. Integer is stored in accumulator in 2's compliment form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into risk of overflowing. When overflow occurs, only modular addition can generate correct result. But in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UDW (FFFFFFFF) with DW (00000001) results in (1FFFFFFFE). The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.*

Table 5-9. Accumulator Channel Precision

Data Type	# Channel	Bits / Channel	Description
F	8	32	When accumulator is used for float, it has the exact same precision as any GRF register
D (UD)	8	64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword integer values. The data are always stored in accumulator in 2's compliment form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting mul, mach and shr (or mov). [Open: may mention negating a UD may result in unpredictable numbers.]
W (UW)	16	33	When the internal execution data type is doubleword integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's compliment form with 33 bits total. This supports single instruction multiplication of two word source in W and/or UW format.
B (UB)	N/A	N/A	Not supported data type.

5.3.4.6 Control Register

ARF Register Type Encoding (RegNum[7:4]):	1000b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	4
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

The Control register is a read-write register. It contains four 32-bit subregisters that can be accessed individually.

Subregister *cr0.0*:*ud* contains normal operation control fields such as the floating point mode and the accumulator disable. It also contains the master exception status/control field that allows software to switch back to the application thread from the system routine.

Subregister *cr0.1*:*ud* contains the mask and status/control fields for all exceptions. The exception fields are arranged in significance-decreasing order from MSB to LSB. This allows the system routine to use *lzd* instruction to find the high priority exceptions and handles them first. As each exception is mapped to a single bit, other exception priority order may be implemented by software. System routine may choose to handle one exception at a time, by handle the exception detected by a *lzd* instruction and return to application thread. Or it may choose to handle all the concurrent exceptions, by looping through the exception fields until all outstanding exceptions are handled before returning back to the application thread.

Exception enable bits (bits 15:0 in *cr0.1*:*ud*) control whether an exception will cause hardware to jump to system routine or not. Exception status and control bits (bits 31:16 in *cr*0.1:*ud*) indicate which exceptions have occurred and are used for system routine to clear the exception. Even if a given exception is disabled, the corresponding exception status and control bit still reflects the status whether an exception event has occurred or not.

cr0.2:*ud* contains the **Application IP (AIP)** indicating the current thread IP when an exception occurs.

cr0.3:*ud* is reserved. Writing to this subregister is dropped; result of reading from this subregister is unpredictable.

Fields in Control registers also refer to a virtual register called **System IP (SIP)**. SIP is the virtual register holding the global System IP, which is the initial instruction pointer for the system routine. There is only one SIP for the whole system. It is virtual only from a thread's point of view, as it is not visible (i.e. not readable and not writeable) to the thread software executed on a GEN EU. It can only be accessed indirectly by the hardware to response to exception events. Upon an exception, hardware performs some book keepings (e.g. saving the current IP into AIP) and then jumps to SIP. Upon finishing exception handling, the system routine may return back to the application by clearing the Master Exception Status and Control field in *cr*0, which will cause the hardware to load AIP to IP register. See STATE_SIP command for how to set SIP.

Table 5-10. Register and Subregister Numbers for Control Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***cr0*** All other encodings are reserved.	00000 = **cr0.0:ud.** It contains general thread control fields 00100 = **cr0.1:ud.** It contains exception status and control 01000 = **cr0.2:ud.** It contains AIP.10100 **(reserved)** All other encodings are reserved.

Table 5-11. Control Register Fields

DWord	Bits	Subfield Description
0 (cr0.0:ud)	31	**Master Exception State and Control.** This field is the master state and control for all exceptions. Reading a 0 indicates that the thread is in normal operation state and a 1 means the thread is in exception handle state. Upon an exception event, hardware sets this field to 1 and switch to SIP. Writing a 1 to this field has no effect. Writing a 0 to this field also has no effect if the previous value is 0. In both cases, the field keeps the previous value. If the previous value of this field is 1, software writing a 0 causes the thread to return to AIP. This transition is automatic – software does not have to move AIP to IP. The value of this field then stays as 0. This field is initialized to 0. 0 = Indicate that the thread is in normal state 1 = Indicate that the thread is in exception state
	30:16	Reserved: MBZ
	15	**Breakpoint Suppress.** This field specifies whether breakpoint exception is suppressed or not. This field is normally set by software and cleared by hardware. If Master Exception Status and Control field is 1, this field is ignored by hardware. If Master Exception Status and Control field is 0 (i.e. not in system routine) and Breakpoint is enabled: If this field is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This field is provided to prevent infinite loop of jumping to the system routine on a breakpoint condition. The system routine must set this bit (and also clear the corresponding status and control field) before returning to the application thread. This field has no effect when Breakpoint Enable bits is cleared. This field is initialized to 0. 0 = Breakpoint exception is not suppressed 1 = Breakpoint exception is suppressed
	14:9	Reserved : MBZ
	7	Reserved : MBZ

DWord	Bits	Subfield Description
	3	**Vector Mask Enable (VME).** This field indicates DMask or Vmask should be used by EU for execution. This field is set by the Thread Dispatch.. 0 : Use Dispath Mask (DMASK) 1 : Use Vector Mask (VMASK)
	2	**Single Program Flow (SPF).** Specifies whether the thread has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). This field affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This field is initialized by the Thread Dispatch. 0: Multiple Program Flows 1: Single Program Flow **Programming Restriction:** The *fork* instruction MUST be used to toggle SPF in the middle of a program to bring the PcIPs to the ExIP. Program is not allowed to write directly into this bit. **Power Optimization:** If an entire shader doesn't do SIMD branching, driver can set the SPF to 1 to save power in HW.
	1	**Accumulator Disable.** This field controls the update of the accumulator by the instruction field *AccWrCtrl*. If this field is cleared, the accumulator is updated for all instructions with *AccWrCtrl* enabled. If set, the accumulator is disabled for all update operations, maintaining its value prior to being disabled. Setting this field has no effect if the accumulator is the explicit destination operand for an instruction. This field is initialized to 0. 0 = Enable accumulator update 1 = Disable accumulator update *Usage Notes:* *This control bit is primarily designed for the System Routine. That routine is not expected to use the accumulator, though it may need to use instructions which include implicit update of the accumulator. In order to use those instructions within the System Routine, but still preserving the accumulator contents upon return to the application kernel, the System Routine would either (a) save and restore the accumulator, or (b) prevent the accumulator from being unintentionally modified. This control bit has been added for the latter method.* *Software has the option to limit the setting of this control bit strictly within the System Routine. If, by convention, this bit is clear within application kernels, the System Routine can simply set the bit upon entry and clear it prior to returning control to the application kernel. This usage model would not necessarily require cr0.0 to be saved/restored in the System Routine. However, if by convention application kernels are permitted to set this bit, then the System Routine would be required to preserve the content of this bit.*
	0	**Floating Point Mode (FPMode).** This field specifies whether the current floating point operation mode is in IEEE standard mode or the alternative mode. It is used to control the floating operation of the Execution Unit. It is also forwarded on the message sideband for all out-going messages, for example, to control the floating point mode of the Extended Math unit or the Sampler unit. Software may modify this field to dynamically switch between the two floating point modes. This field is initialized by the Thread Dispatch. 0 = IEEE floating point mode 1 = Alternative floating point mode

DWord	Bits	Subfield Description
	30	**External Halt Exception Status and Control.** This field indicates the External Halt exception. It is set by EU hardware upon receiving the broadcast External Halt signal. System routine should reset this field before returning to application routine in order to avoid infinite loop. This bit may be set or cleared by software. This field is initialized to 0.
	29	**Software Exception Control.** This is the control field of software exception. Setting this field to 1 in application routine will cause an exception. Clearing this field in application routine has no effect. Upon entering system routine, the hardware maintains this field as one to signify software exception. System routine should reset this field before returning to application routine. This field may be set or cleared by software. This field is initialized to 0.
	28	**Illegal Opcode Exception Status.** This field, when set, indicates illegal opcode exception. The exception handle routine normally does not return back to the application thread upon an illegal opcode exception. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	27:24	Reserved: MBZ
	23	**Preemption Exception Status.** This field, when set, indicates a preemption exception, which can be active even if the preemption exception is disabled, allowing polling of this bit at specific points in the kernel rather than allowing an exception to occur at any instruction. This field is initialized to 0.
	22:16	Reserved: MBZ
	15	**Breakpoint Enable.** Specifies whether breakpoint exception is enabled or not. This field is initialized by the Thread Dispatcher. Format = ENABLED 0 = Disabled 1 = Enabled
	13	**Software Exception Enable.** This field enables or disables the software exception. Enabling or disabling this field may allow host software to turn on/off certain features (such as profiling) without changing the kernel program. This field is initialized by the Thread Dispatcher. Format = ENABLED
	12	**Illegal Opcode Exception Enable.** This field specifies whether illegal opcode exception is enabled or not. Illegal opcode exception includes illegal opcode and undefined opcode, caused by bad program or run time data corruption. This field is initialized by the Thread Dispatcher. Software should normally set it in the interface descriptor. Even though the mechanism is provided to disable illegal opcode exception, it should be used with extreme caution. Format = ENABLED
	11	Reserved: MBZ

<table>
<tr><th>DWord</th><th>Bits</th><th>Subfield Description</th></tr>
<tr><td></td><td>10</td><td><b>Preemption Exception Enable.</b> Specifies whether preemption exception is enabled or not.
This field is initialized to zero on thread dispatch.
Format = ENABLED
0 = Disabled
1 = Enabled</td></tr>
<tr><td></td><td>9:0</td><td>Reserved: MBZ</td></tr>
<tr><td>2
(cr0.2:ud)</td><td>31:3</td><td><b>Application IP (AIP).</b> This is the register storing the instruction pointer before an exception is handled. Upon an exception, hardware automatically saves the current IP into the AIP register, and then sets the <b>Master Exception State and Control</b> field to 1, which forces a switch to the System IP (SIP). AIP register may contain either the pointer to the instruction that causes the exception or the one after (such as mask stack overflow/underflow exceptions). This is shown in the following table, where IP is the instruction which generated the exception.
<table>
<tr><th>Exception Type</th><th>AIP Value</th></tr>
<tr><td>Breakpoint</td><td>IP</td></tr>
<tr><td>External Halt</td><td>n/a [1]</td></tr>
<tr><td>Software Exception</td><td>IP + 1</td></tr>
<tr><td>Illegal Opcode</td><td>IP</td></tr>
</table>
(1) External Halt exception is asynchronous and not associated with an instruction.
When the system routine changes the Master Exception State and Control field from 1 to 0. Hardware restores IP from this register. This field is writable allowing returning IP to be altered after an exception handle.</td></tr>
<tr><td></td><td>2:0</td><td>Reserved : MBZ</td></tr>
</table>

Implementation Restriction on Register Access: *When the control register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to* ***'switch'*** *for an instruction that uses control register as an explicit operand. This is important as the control register is an implicit source for most instructions. For example, fields like FPMode and Accumulator Disable control the arithmetic and/or logic instructions. Therefore, if the instruction updating the control register doesn't set 'switch', subsequent instructions may have indeterministic results.*

5.3.4.7 Notification Registers

ARF Register Type Encoding (RegNum[7:4]):	1001b
Number of Registers:	3
Default Value:	No
Normal Access:	RO
Elements:	3
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

There are three notification registers (*n0.0:ud*, *n0.1:ud, and n0.2:ud*) used by the *wait* instruction. These registers are read-only and can be accessed in 32-bit granularity.

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximal allowable of concurrent threads in the system. Therefore, the range of the thread-to-thread notification count in n0, is larger than the maximum number of threads computed by EUID * TID.

When directly accessed, this register is read-only. If the value is none zero, the only way to alter the value is to use the wait instruction to decrement the value until zero is reach. A wait instruction on a zero notification subregister will cause the thread to stall, waiting for a notification signal from outside targeting to the same subregister. See wait instruction for details.

Implementation Restrictions: The notification registers are initialized to 0 after hardware/software reset. However, it is not reset at thread dispatch time.

Table 5-12. Register and Subregister Numbers for Notification Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***n0*** All other encodings are reserved.	00000 = ***n0.0:ud*** 00100 = ***n0.1:ud*** 01000 **=** ***n0.2:ud*** All other encodings are reserved.

Table 5-13. Fields of Notification Register n0

DWord	Bits	Subfield Description
0	31:16	Reserved: MBZ
	15:0	**Thread to Thread Notification Count.** This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U16

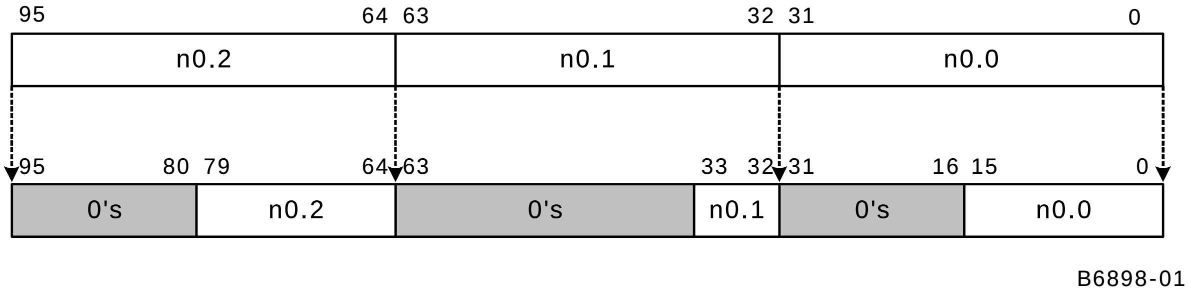
Table 5-14. Fields of Notification Register n1

DWord	Bits	Subfield Description
0	31:1	Reserved : MBZ

Table 5-15. Fields of Notification Register n2

DWord	Bits	Subfield Description
0	31:16	Reserved: MBZ
	15:0	**Thread to Thread Notification Count.** This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U16

Table 5-16. Format of the Notification Register



5.3.4.8 IP Register

ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

The ip register can be accessed as a 32-bit quantity. It is a read-write register, containing the current instruction pointer, which is relative to the **Generate State Base Address**. Reading this register returns the instruction pointer of the current instruction. The 3 LSBs are always read as zero. Writing this register forces the program flow to jump to the new address. When it is written, the 3 LSBs are dropped by hardware.

Table 5-17. Register and Subregister Numbers for IP Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ***ip*** All other encodings are reserved.	00000 = **ip:ud** All other encodings are reserved.

Table 5-18. IP Register Fields

DWord	Bits	Subfield Description
0	31:3	**Ip.** Specifies the current instruction pointer. This pointer is relative to the **General State Base Address**.
	2:0	Reserved : MBZ

5.3.4.9 TDR Register

ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	8
Default Value:	No
Normal Access:	RO/CW
Elements:	8
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	No
Index-ability:	No

There are 8 thread dependency registers (tdr0.0:uw, tdr0.1:uw, tdr0.2:uw, tdr0.3:uw, tdr0.4:uw, tdr0.5:uw, tdr0.6:uw, and tdr0.7:uw) used by HW for the *sendc* instruction. These registers are read-only and can be accessed in 16-bit granularity.

When accessed explicitly, each thread dependency register has FFTID in the lower 8 bits, bits8 to bits14 are forced to zero by HW. Bit[15] is the valid bit, which indicate whether the current thread has a dependency on the dependency thread stored in this thread dependency register..

The thread dependency registers are read only, the valids can only be set with a thread dispatch, and it will reset by the broadcasting end of thread messages after a thread retired. The FFTID's can only be changed with a therad dispatch. Any write into any of the TDR register will clear the valid bit for the particular TDR if the write enable is true, the FFTID portion is strictly read only.

Table 5-19. Register and Subregister Numbers for TDR Register

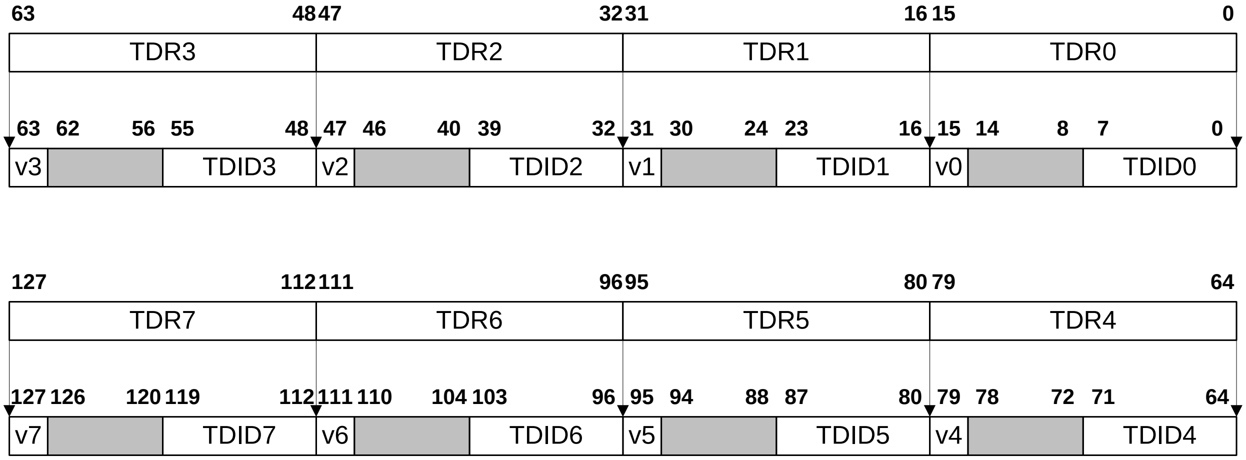
RegNum[3:0]	SubRegNum[4:0]
1011 = ***tdr0*** All other encodings are reserved.	00000 = **tdr0.0:uw** 00001 = **tdr0.1:uw** 00010 = **tdr0.2:uw** 00011 = **tdr0.3:uw** 00100 = **tdr0.4:uw** 00101 = **tdr0.5:uw** 00110 = **tdr0.6:uw** 00111 = **tdr0.7:uw** All other encodings are reserved.

Table 5-20. Fields of TDR Registers

DWord	Bits	Subfield Description
3	31	**Valid7.** This field indicates if the thread specified by FFTID7 is still in-flight.
	30:24	Reserved: MBZ
	23:16	**FFTID7.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
	15	**Valid6.** This field indicates if the thread specified by FFTID6 is still in-flight.
	14:8	Reserved: MBZ
	7:0	**FFTID6.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
2	31	**Valid5.** This field indicates if the thread specified by FFTID5 is still in-flight.
	30:24	Reserved: MBZ
	23:16	**FFTID5.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
	15	**Valid4.** This field indicates if the thread specified by FFTID4 is still in-flight.
	14:8	Reserved: MBZ
	7:0	**FFTID4.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
1	31	**Valid3.** This field indicates if the thread specified by FFTID3 is still in-flight.
	30:24	Reserved: MBZ
	23:16	**FFTID3.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
	15	**Valid2.** This field indicates if the thread specified by FFTID2 is still in-flight.
	14:8	Reserved: MBZ
	7:0	**FFTID2.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
0	31	**Valid1.** This field indicates if the thread specified by FFTID1 is still in-flight.
	30:24	Reserved: MBZ

DWord	Bits	Subfield Description
	23:16	**FFTID1.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8
	15	**Valid0.** This field indicates if the thread specified by FFTID0 is still in-flight.
	14:8	Reserved: MBZ
	7:0	**FFTID0.** This field is the FFTID of the third thread which the current thread is depended on. It can be changed by the end of thead broadcasting messages. Format: U8

Table 5-21. Format of the Thread Dependency Register



5.3.5 Region Parameters

Unlike conventional SIMD architectures where an N-bit wide SIMD instruction can only operate on N-bit aligned SIMD data registers, a region-based register addressing scheme is employed in GEN architecture. The region-based register addressing capability significantly improves the SIMD computation efficiency by providing per-instruction-based multiple data gathering from register file. This avoids instruction overhead to perform data pack, unpack, and shuffling, which has been observed on other SIMD architectures. One benefit of such capability is allowing various kinds of 3D Graphics API Shader compute models to run efficiently on GEN. Another benefit is allowing high throughput of media applications, which tend to operate on byte or word data elements.

This can be illustrated by the example shown in Figure 5-7 and Figure 5-8. As shown in Figure 5-7, a sequence of SIMD instruction is executed on a conventional load/store based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first level of loop unrolling. As shown, there is a second level of loop for the task. Before a given SIMD compute instruction, *Process (i)*, can proceed, there might be a load, a data rearrange and a data unpack (and conversion) instruction to load and prepare the input data. After the compute instruction is complete, it might also require pack, re-arrange and store instructions, to format and save the same to memory. At the loop, other scalar computations such as loop count and address generation may be needed. For the same program, when the data can fit in the large GEN GRF register file, the outer loop may be unrolled for GEN. Here one or a few block loads (using *send* instruction) may be sufficient to move the working set into GRF. Then the data shuffle can be combined with the processing operation with region-based addressing capability. Per operand float type and mixed data type operation may also allow GEN to combine data conditioning operations with computing operations. These techniques in GEN architecture help to achieve high compute efficiency and throughput for graphics and media applications.

Figure 5-7. Conventional SIMD Instruction Sequence

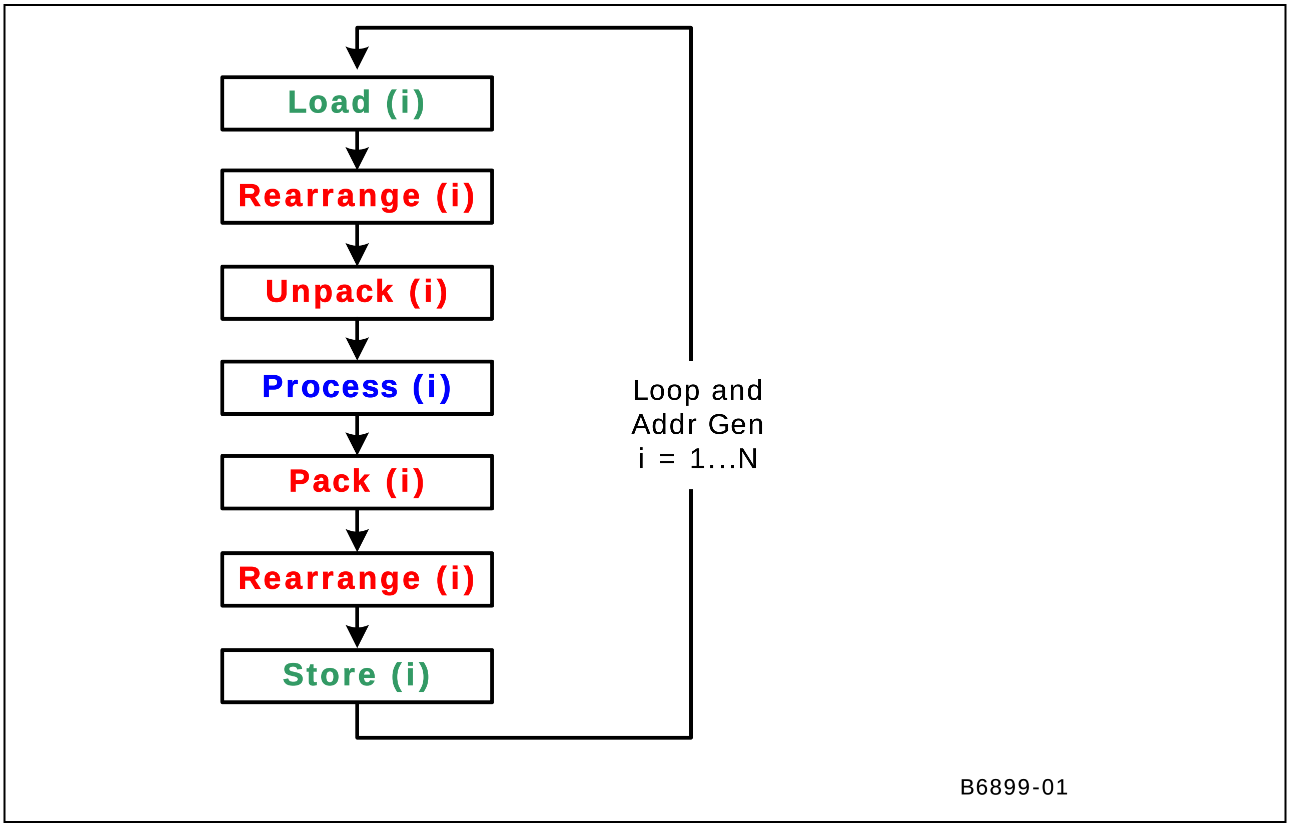
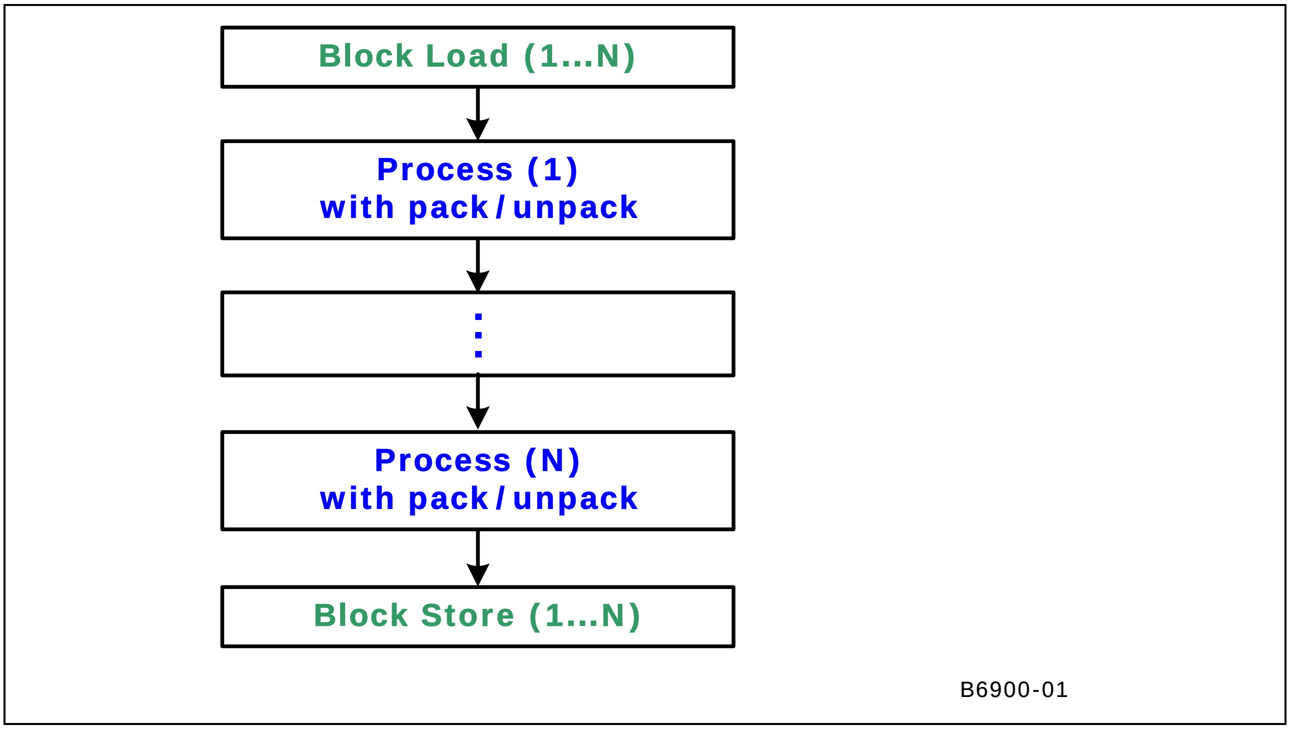


Figure 5-8. GEN SIMD Instruction Sequence for the Same Program



In a GEN instruction, each operand defines a region in the register file. A region may contain multiple data elements. Each data element is assigned to an execution channel in the EU. The total number of data elements of a region is called the **size** of the region, or the size of the operand. The number of execution channels is called the **execution size** (***ExecSize***), which is specified in the instruction word. ExecSize determines the size of region for source and destination operands in an instruction.

- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
    - Exception of this rule is present for the integer reduction instructions (such as sad2 and sada2) where the destination area is smaller than the source area.

Regions are **generalized 2-dimensional** (2D) arrays in row-major order. The first dimension is named the **horizontal** dimension (data elements within a row) and the second dimension is termed the **vertical** dimension (data elements in a column). Here, horizontal/vertical and row/column are just symbolic notations. When the GRF or MRF registers are viewed as a row-major 2D array of memory, such a notation normally matches well with the geometric locations of the data elements of an operand. However, as the register region is fully described by the parameters discussed below, the data elements of a register region may not form a regular rectangular shape. For example, Vertical Stride parameter is allowed to be smaller than Horizontal Stride, making the rows of a register region interleave with each other. It should also note that the meanings of horizontal/vertical here is different than that used for the flag control in Section **Error! Reference source not found.**.

Specifically, a region-based description of a source operand can take the following format

RegFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type

Parameters are as the follows.

- Register Region Origin (*RegFile*, *RegNum* and *SubRegNum*): This set of parameters, including the register file, *RegFile*, the register number, *RegNum*, and the subregister number, *SubRegNum*, describes the register region origin, which is the location of the first data element of the operand. *RegNum* is in unit of 256-bit and *SubRegNum* is in unit of the data element size.

- Width (*Width*): *Width* specifies the number of data elements along the horizontal dimension, or the number of data elements of a row.

- Horizontal Stride (*HorzStride*): *HorzStride* specifies the step size between two adjacent data elements within a row. It is in unit of data element size, which is determined by the data element *Type*.

- Vertical Stride (*VertStride*): *VertStride* specifies the step size between two adjacent data elements along the vertical dimension (or the step size between two rows). It is again in unit of data element size, which is determined by the data element *Type*.

- Data Element Type (*Type*): *Type* specifies numerical data type (float, word, byte, etc.) of the data elements. All data elements within a region must have the same type.

In GEN, both GRF and MRF register files consist of a sequence of 256-bit physical registers. When viewing the register file (GRF for example) as a sequence of 256-bit aligned physical registers, *RegNum* field provides the physical register number, thus for the name. *SubRegNum* provides the sub-field addressing within a physical register. However, when viewing the register file as a byte addressable memory array, (*RegNum* and *SubRegNum*) is just a byte address within the register file with *SubRegNum* providing the lower 5 bits and *RegNum* providing the higher bits.

The execution size is specified for each instruction by the parameter *ExecSize*. The size of the vertical dimension is *ExecSize/Width*, based on the rule that the size of regions must equal to the execution size.

Figure 5-9 depicts the register region description. The example shows a register region of *r4.1<16;8,2>:w*, where the shaded fields denote the data elements in the region and the numbers in these fields are the execution channel assignments. The register region assumes that an *ExecSize* of 16 is set for the instruction. Each data element is a word (as noted by the type field "*:w*"). The origin of the region is at the second word of r4, denoted by *r4.1*. Each row of the region has 8 data elements (words) that are 2 data elements (words) apart. The distance between two rows is 16 words. Note that the region shown is for illustration purpose only. It does not represent a typical usage model nor a performance optimized mode.

Figure 5-9. An example of a register region (*r4.1<16;8,2>:w*) with 16 elements

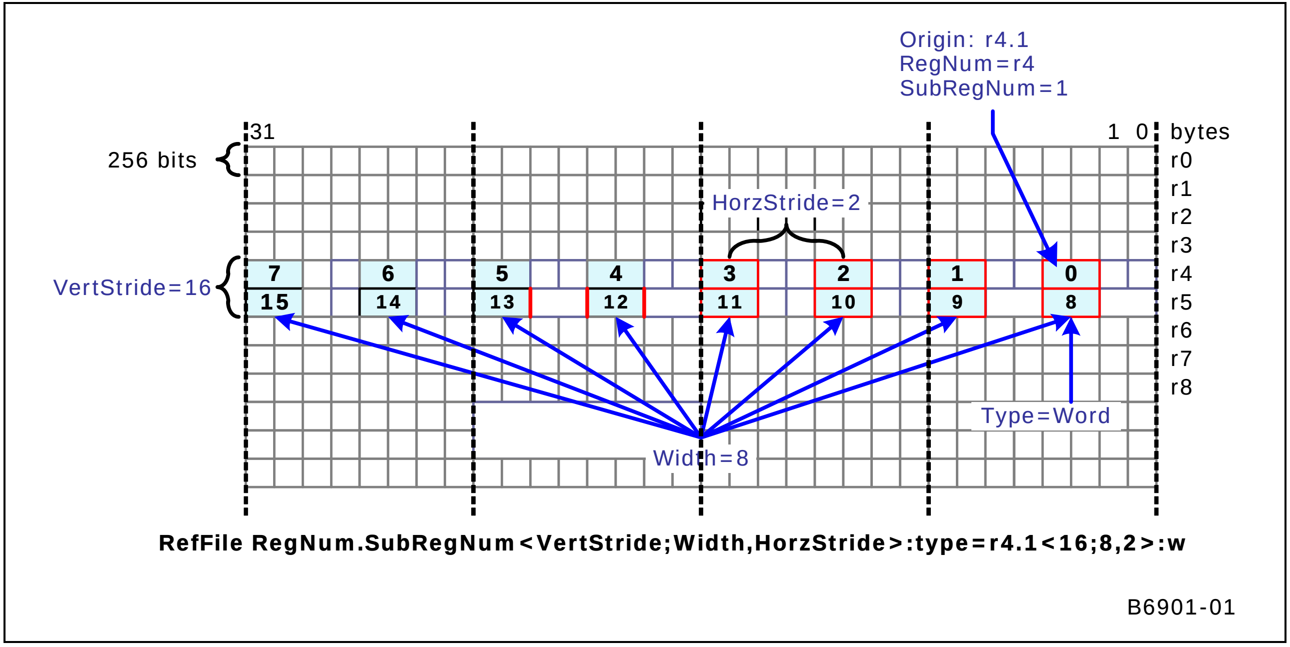
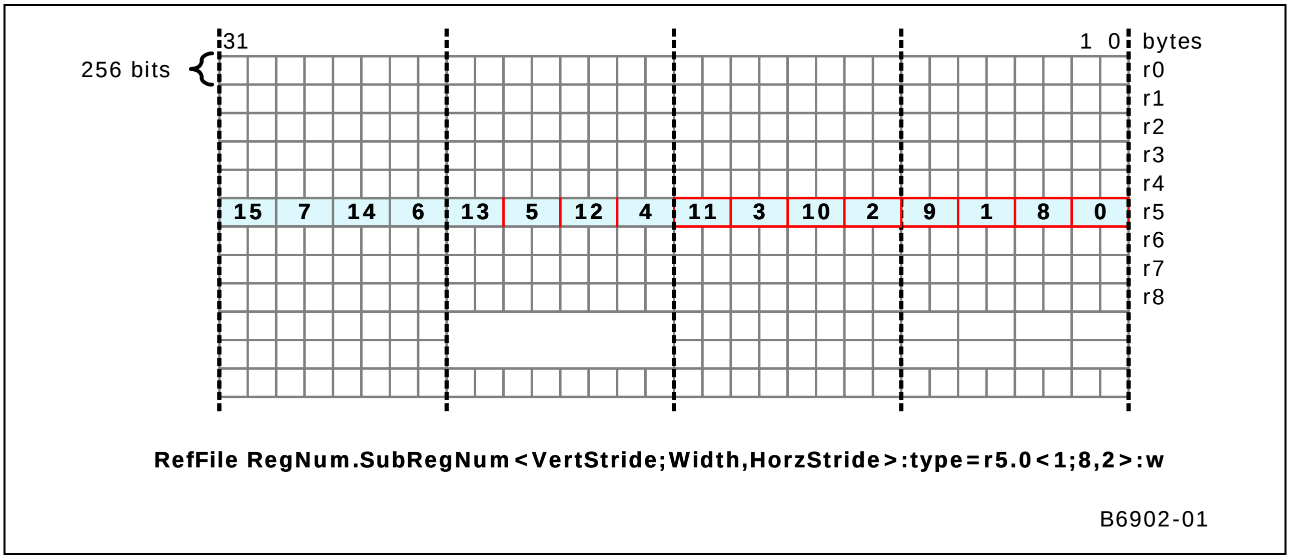


Figure 5-10 shows another example where the rows are interleaved. The region, having word data elements, starts at location r5.0:w. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location 5.2:w. And there are 8 elements per row. VertStride, the distance between two rows, is only 1 word, which is less than HorzStride. Therefore, the first element of the second row (channel number 8) is at r5.1:w, just next to channel number 0. It is clear from the picture that the two rows are interleaved.

By varying the region parameters, reader may construct other configurations. The next section provides more details on the region-based register addressing. However, there are restrictions imposed by hardware implementation, which can be found in the later sections of this chapter.

Figure 5-10. A 16-element register region with interleaved rows (*r5.0<1;8,2>:w*)


31
1 0
bytes
256 bits
r0
r1
r2
r3
r4
r5
r6
r7
r8
15
7
14
6
13
5
12
4
11
3
10
2
9
1
8
0
RefFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type=r5.0<1;8,2>:w
B6902-01

Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.

```
// Input:          Type: ub | b | uw | w | ud | d | f | v
//                 RegNum: In unit of 256-bit register
//                 SubRegNum: In unit of data element size
//                 ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output:         Address[0:ExecSize-1] for execution channels


int ElementSize = (Type==“b”||Type==“ub”) ? 1 : (Type==“w”|Type==“uw”) ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum<<5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
        int Offset = RowBase;
        for (int x=0; x<Width; x++) {
                Address [Channel++] = Offset;
                Offset += HorzStride*ElementSize;
        }
        RowBase += VertStride * ElementSize;
}
```

As *HorzStride* and *VertStride* are specified independently (note that *VertStride* might be smaller than or equal to *HorzStride*), the region may take various shapes from a replicated scalar, a replicated vector, a vector of replicated scalars, a sliding window, to a non-overlapped 2D array.

A region-based description of a destination operand can take the following simplified format

RegFile RegNum.SubRegNum<HorzStride>:type

The destination operand is only allowed to have a 1 dimensional region. The Register Region Origin and Type are the same as for a source operand. The total number of elements is given by *ExecSize*. However, only *HorzStride* is required to describe the 1D region, not *VertStride* and *Width*.

As a source register region may across multiple physical GRF register, an instruction with such source operands may take more than two execution cycles to gather source data elements for execution. The destination register region is restricted to be within a physical GRF register. In other words, destination scatter writes over multiple physical registers are not supported.

5.3.6 Region Addressing Modes

There are two different register addressing modes: Direct register addressing and register-indirect register addressing. Depending on the register region description, the register-indirect register addressing mode can be further divided into three usages: 1x1 index region where only the origin of register region is provided by the address register, Vx1 index region where the offset of each row of the register region is provided by an address register, VxH index region where the offset of each data element is provided by an address register.

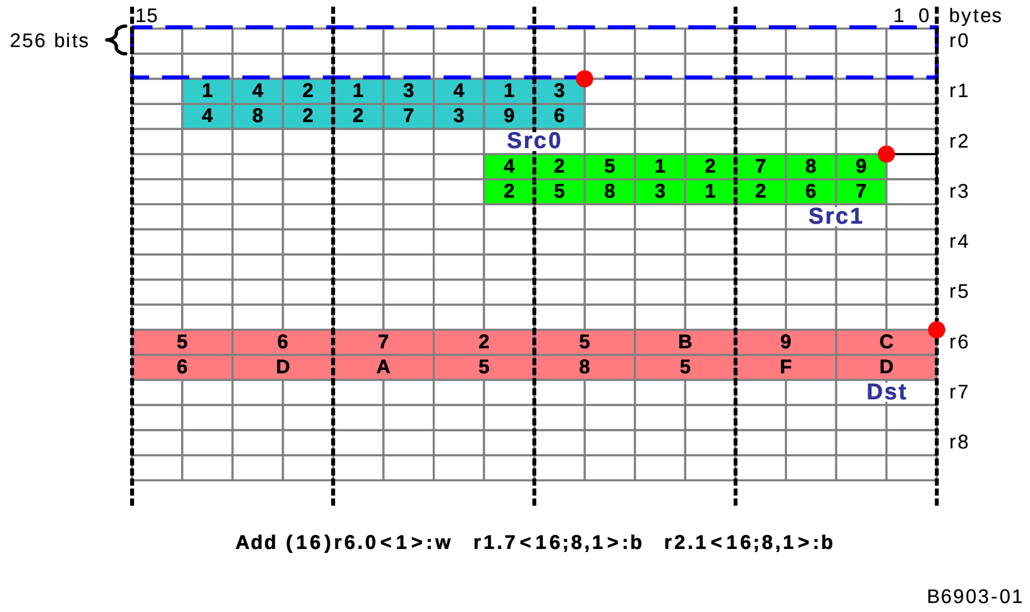
5.3.6.1 Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Figure 5-11 and Figure 5-12 are two examples of direct register addressing.

For the example in Figure 5-11, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical stride of 16 bytes for *Src0* and *Src1* indicates that the first element and the 9'th element are 16 bytes apart in the register file. Note that *Src0* falls into the 256-bit physical GRF register starting at r1.0, but Src1 crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have C = 3+9.

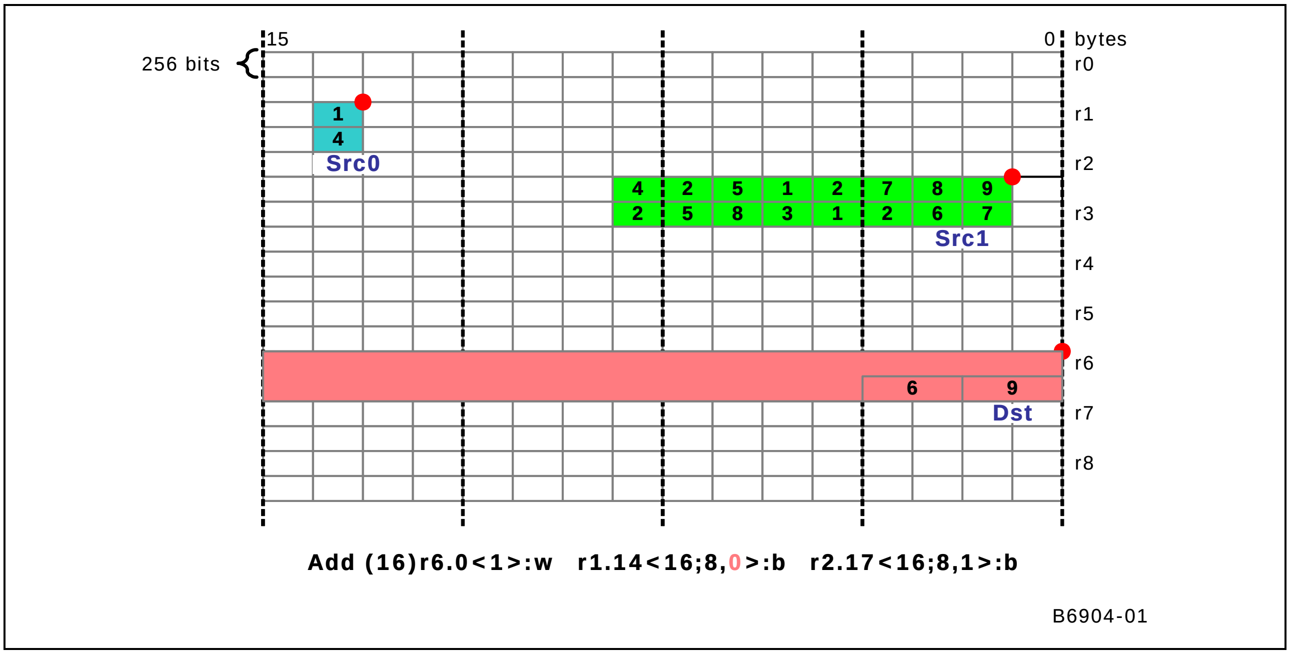
Figure 5-11. A region description example in direct register addressing



Add (16)r6.0<1>:w r1.7<16;8,1>:b r2.1<16;8,1>:b

For the example in Figure 5-12, the sizes of areas of *Src0* and *Src1* are the same, but *Src0* contains a vector of replicated scalars. With HorzStride = 0 and Width = 8, the first row of 8 elements in Src0 is a replication of the byte at r1.14. Comparing *ExecSize* of 16 to Width of 8 indicates that there is a second row of 8 elements in *Src0*. With VertStride = 16, the second row in *Src0* is a replication of the byte at r1.20 (20 = 14+16). Effectively, the 16 data elements of *Src0* are {1,1,1,1,1,1,1,1, 4,4,4,4,4,4,4,4}.

Figure 5-12. A region description example in direct register addressing with <src0> as a vector of replicated scalars

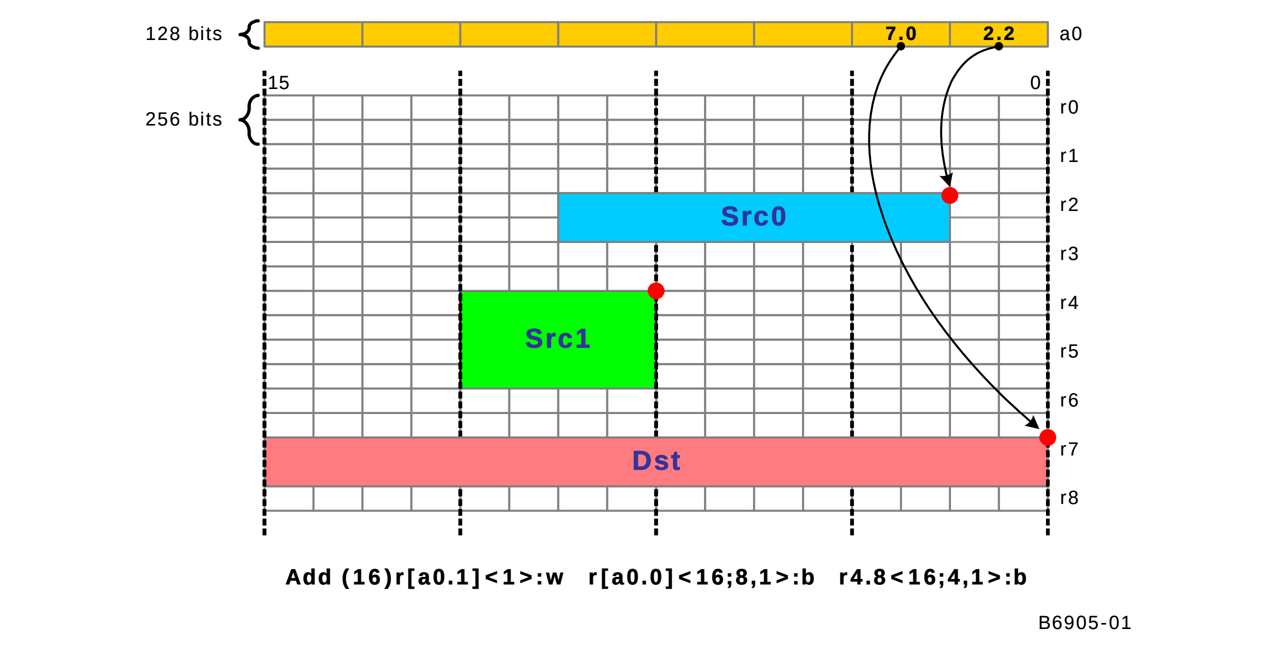


5.3.6.2 Register-indirect Register Addressing with a 1x1 Index Region

In the register-indirect register addressing mode with 1x1 index region, the region origin is provided by the content of the address register, the rest of region parameters are provided by the fields in the instruction word.

Figure 5-13 depicts an example for this addressing mode. For example, the present of full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.

Figure 5-13. An example illustrating register-indirect register addressing mode with a 1x1 index region

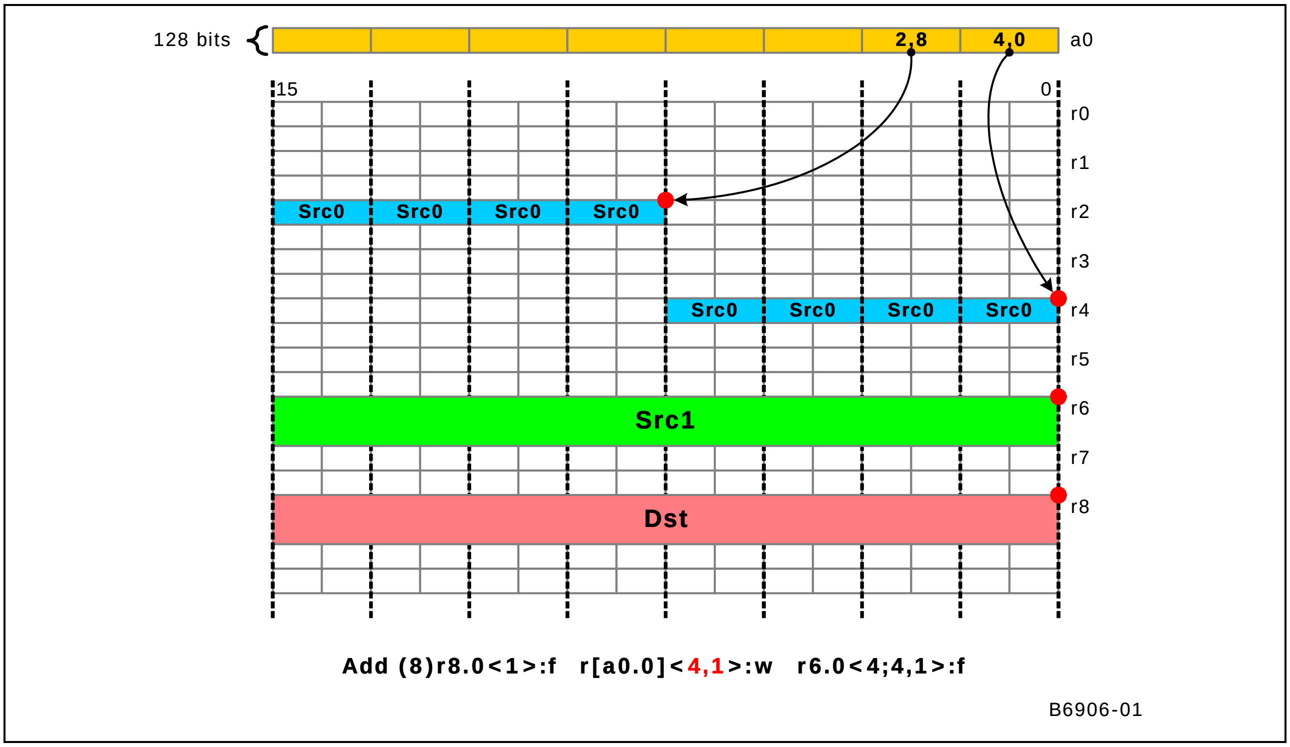


5.3.6.3 Register-indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided, the following contiguous address registers are for the following rows. The total number of address registers used is inferred by parameters *ExecSize* and *Width.*

An example is provided in Figure 5-14. The assembly syntax notion of a register region without vertical stride, <4,1>, corresponding to the special encoding of vertical stride of 0xF in the instruction word, indicates the VxH or Vx1 mode of indirect register addressing. In this case, the origin for each row of Src0 is provided by the address register. As ExecSize/Width = 2, there are two address registers a0.0 and a0.1, each pointing to a row of 4 data elements.

Figure 5-14. An example illustrating register-indirect-register addressing mode with a Vx1 index region (Src0)

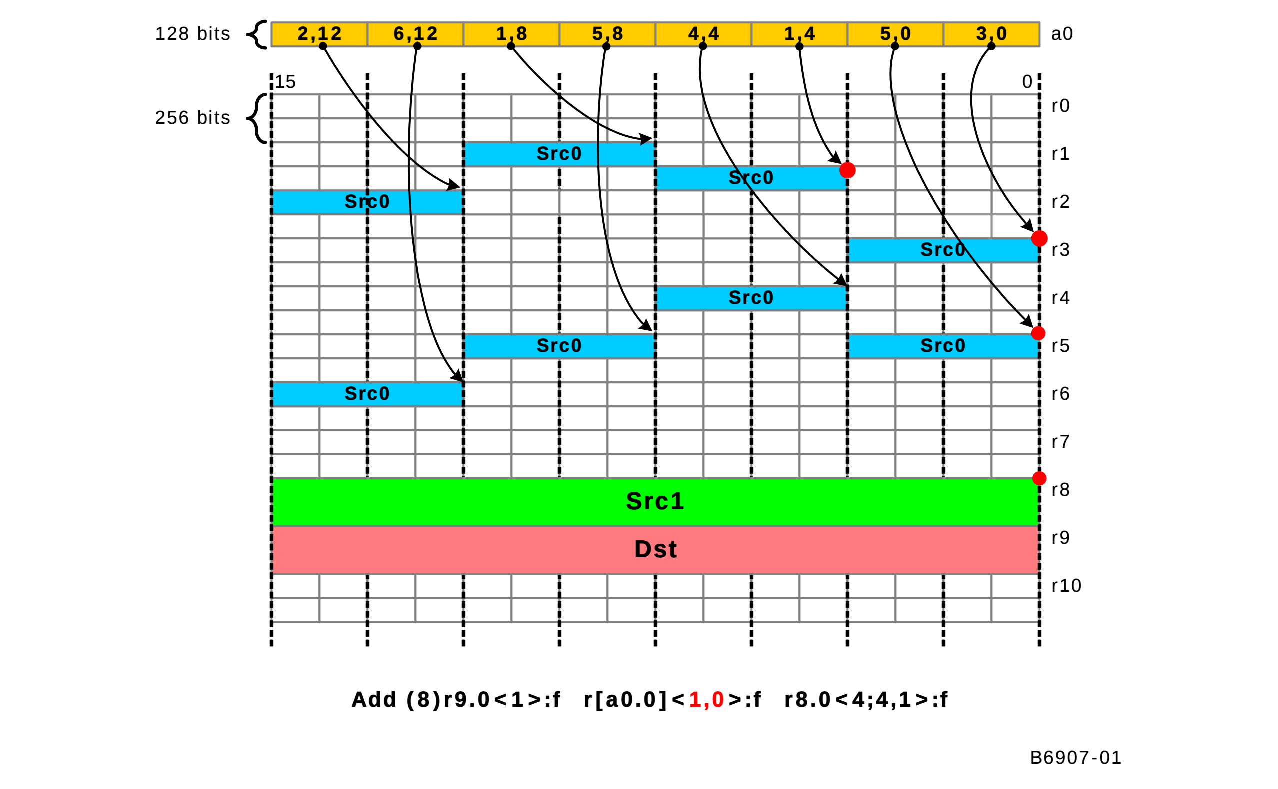


5.3.6.4 Register-indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When *Width* of the region is 1, the number of address registers used equals *ExecSize*.

An example is provided in Figure 5-15. The absent of vertical stride in the region description <1,0> with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As ExecSize/Width = 8, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.

Figure 5-15. An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



5.3.7 Access Modes

There are two basic GEN register access modes controlled by a single bit instruction subfield called Access Mode.

- 16-byte Aligned Access Mode (**align16**): In this mode, the origins of all operands (sources and destination), whether it is by direct addressing or register-indirect addressing, are 16-byte aligned. For example a row in the region description starts at 16-bype aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
  - The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.
- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region register descriptions are supported, however, source swizzle or destination mask are not supported.
  - The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.

5.3.8 Execution Data Type

GEN architecture supports instructions with mixed data types. The internal hardware computation is performed using the execution data type. When an instruction has only one source operand or has two source operands of the same data type, the execution data type is the same as that of the source. When an instruction has two source operands of different types, an execution data type is determined and one of the source operands will be converted to the execution type before the computation is performed. The execution type is independent of the destination data type. When the destination data type is different from the execution data type, a type conversion is performed on the intermediate compute results before the results are written into the destination register. Such a destination type conversion doesn't apply to accumulator registers, implicitly or explicitly. Therefore, accumulator type cannot differ from the execution data type.

Determination of the execution data type for two sources of different data types obeys the following rules

- Instuction with mixed float and integer type sources is not allowed.
- Else if any source is a dword, the execution data type is signed dword integer (D)
- Else execution data type is signed word integer (W)

Note that when the execution data type is an integer, it is always a signed integer. This doesn't affect the functional correctness of the instruction as extra precisions are carried within the hardware, including the accumulator. See Instruction Reference Chapter for detailed description for each instruction.

The following Instruction can have integer souce(s) and float destination, all the other instructions can only be all float or all integer for source(s) and destination.

- *MOV, ADD, MUL, MAC, MAD, LINE*

The *MOV* instruction is the only instruction can convert an float to integer.

5.3.9 Register Region Restrictions

The following register region rules apply to the GEN implementation. Rules and restrictions for compressed instructions can be found in the Instruction Compression section.

1. *ExecSize* must be equal to or less than the maximum execution size supported for the operand type. As shown in **Table 5-22**, the maximum execution size is determined by the largest operand type of the sources and destination of the instruction.
2. The mapping of data elements within the region of a source operand is in row-major order and is determined by the region description of the source operand, plus *ExecSize* and destination region description.
3. *ExecSize* must be equal to or greater than *Width*.
4. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.
5. If *ExecSize* = *Width* but *HorzStride* = 0, there is no restriction on *VertStride*.
6. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
7. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be set to zero.
8. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.
9. Destination region cannot cross the 256-bit register boundary.
   9.1. Exception to this rule is for a SIMD16 DW/Float instruction where the destination region covers exactly **two adjacent** 256-bit physical registers.
10. Destination region alignment rule.

10.1. With the exception on ‘raw move’ described in rule #10.3 and the exception on byte destination in rule #10.5, all destination data elements must be aligned to the size for the execution data type of the instruction. For example, if one of the source operands is in dword mode (a float, a signed or unsigned dword integer), the execution data type will be either float or signed dword integer. Therefore, the destination data elements must be dword aligned. This rule has the following two implications:

10.1.1. The destination sub-register must be aligned to the size of the execution data type.

10.1.2. If *ExecSize* is greater than 1, *dst.HorzStride**sizeof*(dst.Type)* must be equal to or greater than the size of the execution data type.

10.2. If *ExecSize* is 1, *dst.HorzStride* must not be 0. Note that this is relaxed from rule 10.1.2. Also note that this rule for destination horizontal stride is different from that for source as stated in rule #7.

10.3. When destination type is byte (UB or B), only a ‘raw move’ using ***mov*** instruction supports packed byte destination register region: *dst.HorzStride* = 1 and *dst.type* = (UB or B). This packed byte destination region is not allowed for any other instructions, including a ‘raw move’ using ***sel*** instruction. This is because ***sel*** instruction is based on word or dword wide execution channels.

10.4. When an instruction has a source region that spans two physical registers and destination register contained in one register, one of the followings must be true:

10.4.1. Destination region is entirely contained in the lower oword of a physical register,

10.4.2. Destination region is entirely contained in the upper oword of a physical register, or

10.4.3. Destination elements are evenly split between the two owords of a physical register.

10.5. When an instruction has a source region that spans two physical registers, the destination spans two physical registers, and the destionation elements are evenly split between the two physical registers.Then each destination register must be entirely derived from one source register.

10.6. Relaxed alignment rule for byte destination. When destination type is byte (UB or B), destination data elements can be either aligned to the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case, the destination data bytes can be either all in the even byte locations or all in the odd byte locations. This rule has the following two implications:

10.6.1. The destination sub-register must be either aligned to the size of the execution data type or one byte higher off the execution data type.

10.6.2. If *ExecSize* is greater than 1, *dst.HorzStride**sizeof*(dst.Type)* must be equal to or greater than the size of the execution data type. This is the same as that in #10.1.2.

11. In Align1 access mode, a source region must be within **two adjacent** 256-bit physical registers.

12. Rules on register-indirect register access:

12.1. An indexed source1 can only have a 1x1 indexed register region – only single index mode is allowed for a source 1.

12.2. An indexed destination can only have a 1x1 indexed register region – only single index mode is allowed for a destination operand.

12.3. Data elements referenced by a single index within a source region cannot cross 256-bit physical register boundary. This applies to register region with a single index or with multiple indices.

12.3.1. A register region with multiple indices may access multiple physical registers as long as data elements associated with each index follow the above-mentioned rule. For example instruction “mov (16) r0.0:uw r[a0.0]<2,2>:uw” is allowed. This is a source gathering instruction whereas the source operand may potentially tough 8 different physical GRF registers.

Table 5-22. Execution size in device hardware

Device	Native GEN Instructions		
Max Operand Size	DWORD	WORD	BYTE
[DevSNB]	16	32	32

Table 5-23. Indirect source addressing support available in device hardware

Device	Indirect Source 0	Indirect Source 1
[DevSNB]	Yes	Yes

5.3.9.1 Examples

Some examples are provided here to illustrate the cases when the register region restrictions are violated. It is provided as informative material to help understanding these restrictions.

Example 1: The following instructions are illegal as they violate rule #10.1, as the destination is not aligned to the execution data type.

```
mov (1) r0.1<1>:b r2.0:w  // dst.SubReg must be even

mov (2) r0.0<1>:b r2.0:w  // dst.HorzStride must be >= 2

mov (2) r0.0<2>:b r2.0:d  // dst.HorzStride must be >= 4

mov (2) r0.0<2>:b r2.0:f  // dst.HorzStride must be >= 4

mov (1) r0.2<1>:b r2.0:d  // dst.SubReg must be dword aligned
```

Example 2: This instruction is illegal as it violates rule #10.1.2, as when *ExecSize* = 1, *dst.HorzStride* cannot be zero.

```
mov(1) r0.0<0>:b r0.0:d
```

Example 3: This instruction is illegal as it violates rule #**Error! Reference source not found.**, as the source contains one row of 2 elements that spans physical register r2 and r3.

```
mov (2)  r1.0:d r2.7<2;2;1>:d
```

5.3.9.2 Different Raw Moves

Definition of Raw Move: Raw move is an operation that moves data elements from source to destination without altering the bit fields of the data elements. It must use one of the move instructions such as ***mov, sel, movi***. Arithmetic instruction that results in unaltered bit fields of the data elements are not treated as raw move. A raw move may subject to the execution channel enables by using prediction or being present in multi-channel branch code segment. Type conversion by definition cannot be used in a raw move. Therefore, source and destination operands must be of the identical data type. For example, if both source and destination are float, for an arithmetic instruction, denorm will be flushed to zero. However, for a raw move, denorm will be preserved.

Definition of Byte Raw Move: As the minimal execution channel type is word, when the destination stride is greater than one byte, each data element of the source can be mapped to one execution channel. This is referred to as Byte Raw Move. Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. Byte Raw move subjects to execution channel enables.

Definition of Packed-Byte Raw Move: As the minimal execution channel type is word, when the destination stride is equal to one byte, two data elements of the source are mapped to one execution channel. This is referred to as Packed-Byte Raw Move. Packed-Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. However, as the data elements are not mapped to execution channels, undefined results may occur if Packed-Byte Raw Move is mixed with execution channel enables. So for Packed-Byte Raw Move, ***WECtrl*** should be used when there are un-enabled channels within the execution size of the instruction.

5.3.10 Destination Operand Description

5.3.10.1 Destination Region Parameters

Based on the above restrictions, a subset of register region parameters are sufficient to describe the destination operand:

- Destination Register Origin
  - Destination Register Number and Destination Subregister Number for direct register addressing mode
  - A Scalar Destination Register Index for register-indirect-register addressing mode
- Destination Register 'Region' – Note that destination register region does not have full region description parameters
  - Destination Horizontal Stride

5.4 SIMD Execution Control

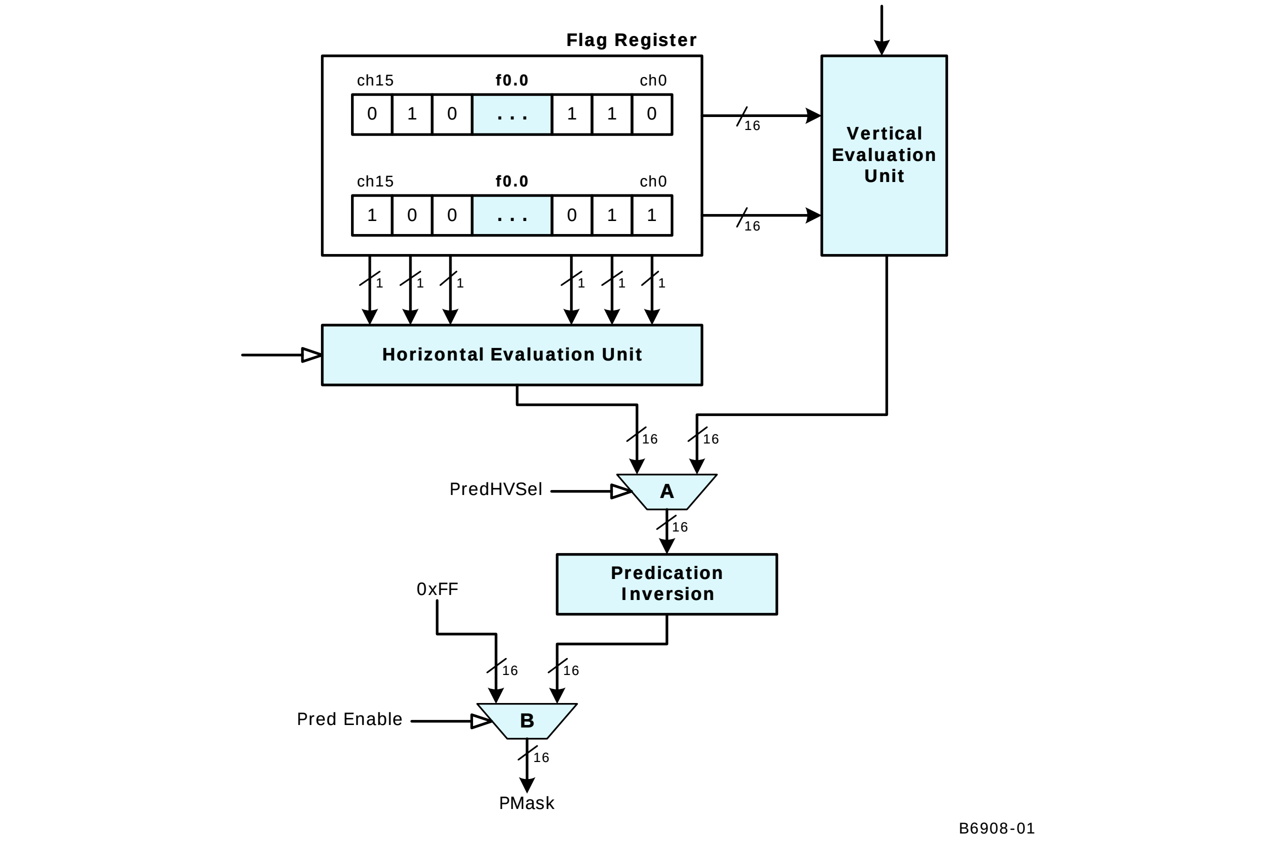
5.4.1 Predication

Predication is the conditional SIMD channel selection for execution on a per instruction basis. It is an efficient way of dynamic SIMD channel enabling without paying branch instruction overhead. When predication is enabled for an instruction, a Predicate Mask (PMask), which contains 16-bit channel enables, is generated internally in EU. Note that PMask is not a software visible register. It is provided here to explain how SIMD execution control works. PMask generation is based on the Predication Control (*PredCtrl*) field, Predication Inversion (*PredInv*) field and the flag source register in the instruction word. See Instruction Summary chapter for definition of these fields.

Figure 5-16 shows the block diagram of the hardware logic to generate PMask. PMask is generated based on combinatory logic operation of the bits in the flag register. Instruction field *PredCtrl* controls the horizontal evaluation unit and vertical evaluation unit. MUX A in the figure selects whether horizontally-evaluated results or vertically-evaluated results are sent to the Predication Invertion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.

Figure 5-16. Generation of predication mask



5.4.2 No Predication

When PredCtrl field of a given instruction is set to 0 (“no predication”), it indicates that no predication is applied to this instruction. Effectively, the resulting PMask is all 1’s. This is shown by the 2:1 multiplexer B controlled by the Pred Enable signal in Figure 5-16. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.

5.4.3 Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is ‘no combination’ – the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```
If (PredCtrl == “Sequential flag channel mapping”) {
        For (ch=0; ch<16; ch++)
                PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}
```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of ‘.x’, ‘.y’, ‘.z’ and ‘.w’ select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = ‘.x’ means that channel 0 in each group is replicated.
- OR combination: PredCtrl of ‘.any4h’ means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of ‘.all4h’ means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```
If (Access Mode == Align16) {
        For (ch = 0; ch < 16; ch += 4)
                Switch (PredCtrl) {
                Case ‘.x’:         bTmp = f0.#[ch]; break;
                Case ‘.y’:         bTmp = f0.#[ch+1]; break;
                Case ‘.z’:         bTmp = f0.#[ch+2]; break;
                Case ‘.w’:         bTmp = f0.#[ch+3]; break;
                Case ‘.any4h’:     bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
                Case ‘.all4h’:     bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
                }
                bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
                PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
        }
}
```

In **Align1** access mode, horizontal combination is based on AND combination '.any#h' and OR combination '.all#h' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.

```
If (Access Mode == Align1) {
        Switch (PredCtrl) {
                Case '.any2h':      groupSize = 2; <op> = '|'; break;
                Case '.all2h':      groupSize = 2; <op> = '&'; break;
                Case '.any4h':      groupSize = 4; <op> = '|'; break;
                Case '.all4h':      groupSize = 4; <op> = '&'; break;
                Case '.any8h':      groupSize = 8; <op> = '|'; break;
                Case '.all8h':      groupSize = 8; <op> = '&'; break;
                Case '.any16h':     groupSize = 16; <op> = '|'; break;
                Case '.all16h':     groupSize = 16; <op> = '&'; break;
        }
        For (ch = 0; ch < 16; ch += groupSize) {
                For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)
                        bTmp = bTmp <op> f0.#[ch+inc];
                For (inc = 0; inc < groupSize; inc ++)
                        PMask[ch+inc] = bTmp;
        }
}
```

5.4.4 Predication with Vertical Combination

Predication with vertical combination uses both flag subregister as inputs. The AND or OR combination is across the subregisters on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {
        For (ch = 0; ch < 16; ch ++) {
                If (PredCtrl == 'any2v')
                        PMask[ch] = f0.0[ch] | f0.1[ch]
                Else If (PredCtrl == 'any2h')
                        PMask[ch] = f0.0[ch] & f0.1[ch]
        }
}
```

5.5 Instruction Compaction

5.5.1 Motivation and Expected Usage

Instruction Compaction is used to reduce the memory footprint of the shaders. It relys on predefined tables to compact instructions if they has patterns matching the entries inside the compaction table.

Because of the limited size of the compaction table in HW, not all instructions will be compacted by Jitter, HW receives both compacted and noncompacted instructions during execution.

Jitter compacts the instruction using the same compaction table inside HW, Jitter need to calculate any branch/jump offset after instruction compaction is done.

5.5.2 Hardware Behavior

Upon receiving an instruction with the bit[29] CompactCtrl bit set, HW uses the 5 indexes inside the compacted instructions to lookup the compaction table, then uses the table output to reconstruct the full size instruction.

If any source of the compacted instruction is immediate, only 13bits of the immediate value is encoded in the compacted instruction, HW sign extends bit[12] all the way for the entire immediate DWord.

Table 5-24. GEN Compacted Instruction Format

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	Description	Bits in 128bits Format	Description (Imm. Src0 or Src1)	Bits in 128bits Format (Imm. Src0 or Src1)
1	8	63	56	8	*Src1 RegNum*	*[108:101]*	***Imm[23:16] Imm[7:0]***	***[119:112] [103:96]***
	8	55	48	8	*Src0 RegNum*	*[76:69]*	*Src0 RegNum*	*[76:69]*
	8	47	40	8	*Dst RegNum*	*[60:53]*	*Dst RegNum*	*[60:53]*
	5	39	35	5	*Src1Index[4:0]*	*[120:109]*	***Src1Index[4:0]***	***[127:120] [111:104]***
	3	34	32	3	*Src0Index[4:0]*	*[88:77]*	*Src0Index[4:0]*	*[88:77]*
0	2	31	30	2				
	1	29	29	1	*CmptCtrl*	*[29]*	*CmptCtrl*	*[29]*
	1	28	28	1	*FlagSubRegNum*	*[89]*	*FlagSubRegNum*	*[89]*
	4	27	24	4	*CondModifier*	*[27:24]*	*CondModifier*	*[27:24]*
	1	23	23	1	*AccWrCtrl*	*[28]*	*AccWrCtrl*	*[28]*
	5	22	18	5	*SubRegIndex[4:0]*	*[100:96] [68:64] [52:48]*	*SubRegIndex[4:0]*	*[100:96] [68:64] [52:48]*
	5	17	13	5	*DataTypeIndex[4:0]*	*[63:61] [46:32]*	*DataTypeIndex[4:0]*	*[63:61] [46:32]*
	5	12	8	5	*ControlIndex[4:0]*	*[31] [23:8]*	*ControlIndex[4:0]*	*[31] [23:8]*
	1	7	7	1	*DebugCtrl*	*[30]*	*DebugCtrl*	*[30]*
	7	6	0	7	*Opcode*	*[6:0]*	*Opcode*	*[6:0]*

Definitions of Fields in the Compact Instruction

Bits	Description
63:56	**Bits [108:101] Source1 register number** forms bits [108:101], the source 1 register number field. If immediate source is used, this field forms [103:96] of the 128-bit instruction word.
55:48	**Bits [76:69] Source0 register number** This field, after unpacking, forms bits [76:69], the source 0 register number field, of the 128-bit instruction word.
47:40	**Bits [60:53] Destination register number** This field, after unpacking, forms bits [60:53], the destination register number field, of the 128-bit instruction word.
39:35	**Src1Index** The 5-bit index for source 1. The 12-bit table-look-up result forms bits [120:109], the source 1 register region fields, of the 128-bit instruction word if immediate source is used, this field forms [108:104] of the 128-bit instruction word. Bit[39] is replicated to [127:109] of the 128-bit instruction word.
34:30	**Src0Index** The 5-bit index for source 0. The 12-bit table-look-up result forms bits [88:77], the source 0 register region fields, of the 128-bit instruction word.

Bits	Description
29	**CompactCtrl – Compaction Control** This field indicates whether the instruction is in the 64-bit compaction form. When this bit is set (bit 29 of DW0), the instruction length is only 64-bit.. The bit location is fixed in both 128-bit and 64-bit instruction forms. 0 = 128-bit form (normal) 1 = 64-bit compaction form
27:24	**Bits [27:24] – CondModifier** This field, after unpacking, is bits [27:24] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.
23	**AccWrCtrl** – **Implicit Accumulator Write Enable** This field, after unpacking, is bit[28] of the 128-bit instruction word.
22:18	**SubRegIndex** The 5-bit index for sub-register fields. The 15-bit table-look-up result forms bits [100:96], [68,64] and [52,48] of the 128-bit instruction word.
17:13	**DataTypeIndex** The 5-bit index for data type fields. The 18-bit table-look-up result forms bits [63:61] and [46, 32] of the 128-bit instruction word.
6:0	**Bits [6:0] – Opcode** This field, after unpacking, is bits [6:0] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.

5.5.3 Rules and Restrictions

In order to reduce the hardware complexity, the following rules and restrictions apply to the compressed instruction:

- Any branch/jump offset need to be based on the physical instruction size, after compaction.
- Any branch/jump instruction with immediate offset larger than 13bits should not be compacted.

5.6 End of Thread

There is no special instruction opcode (such as an END instruction) to cause the thread to terminate execution. Instead, the end of thread is signified by a *send* instruction with the end-of-thread (EOT) sideband bit set. Upon executing a *send* instruction with EOT set, the EU stops on the thread. Upon observing an EOT signal on the output message bus, the Thread Dispatcher makes the thread's resource available. If a thread uses pre-allocated resource managed by a fixed function, such as URB handles and scratch memory, some fixed function protocol also requires the thread to terminate with the message header phase to carry the information in order for the fixed function to release the pre-allocated resource.

EU hardware guarantees that if a terminated thread has in-flight read messages or loads at the time of 'end' that their writebacks will not interfere with either other threads in the system or new threads loaded in the system in the future.

More details can be found in the *send* instruction description in Instruction Reference chapter.

5.7 Creating Conditional Flags

FPU will output 2 sets of conditional signals, 1 set will be generated from before the adder outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. 1 set will be generated from the final results after clamping and re-normalizing/format conversion logic, and we will call this the post conditional signals. The post conditional signals are used for fusing the compare instruction. *The flags generated from the post conditional signals should be equivalent to the flags generated by a separate CMP instruction after the current arithmetic instruction.*

The pre conditional signals will be used to generated flags for CMP/CMPN instructions only, this logically does the compare of the 2 input sources. The post conditional signals will be used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

CMPN with both sources are NaN is a don't care case since this doesn't impact the MIN/MAX operations.

The pre conditional signals include the following:

- **pre_sign** bit: this bit reflects the sign of the computed result directly from the adders, without going through any kind clamping, normalizing, or format conversion logic.
- **pre_zero** bit: this bit reflects whether the value of the adder results are zero, again this should be obtained before any kind clamping, normalizing, or format conversion logic.

The post conditional signals include the following:

- **post_sign** bit: this bit reflects the sign of the final result after all the clamping, normalizing, or format conversion logic.
- **post_zero** bit: this bit reflects whether the value of the adder results are zero after all the clamping, normalizing, or format conversion logic.
- **OF** bit: this bit reflects whether an overflow occured in any of the compution of the current instruction, including clamping, re-normalizing, and format conversion.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0.

- **NS0** bit: The NaN bit indicates whether source 0 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0.

- **NS1** bit: The NaN bit indicates whether source 1 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0. For an operation with one source operand, this bit is also set to 0. This bit is only used for the comparison instruction CMPN, which is specifically provided to emulate MIN/MAX operations. For any other instructions, this bit is undefined.

Flag Generation for CMP instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, and **.le**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
‘**.e**’	Equal-to	**(pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
‘**.ne**’	Not-Equal-to	**!(pre_zero & !(NS0 | NS1)).** This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier ‘**.e**’.
‘**.g**’	Greater-than	**(!pre_sign & !pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
‘**.ge**’	Greater-than-or-equal-to	**((!pre_sign | pre_zero) & !(NS0 | NS1)).** This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
‘**.l**’	Less-than	**(pre_sign & !pre_zero & !(NS0 | NS1)).** This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
‘**.le**’	Less-than-or-equal-to	**((pre_sign | pre_zero) & !(NS0 | NS1)).** This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.

Flag Generation for CMPN instructions (The supported Conditional Modifiers are **ge**, and **.l**)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
‘**.ge**’	Greater-than-or-equal-to	**(!pre_sign | pre_zero | (NS1 & (Opcode==CMPN | OPcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)).** This conditional modifier tests whether source0 is greater than or equal to source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.
‘**.l**’	Less-than	**((pre_sign & !pre_zero) | (NS1 & (Opcode==CMPN | Opcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)).** This conditional modifier tests whether source0 is less than source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.

Flag Generation for All Arithmetic Instructions other than CMP/CMPN instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, **.le**, **.r**, **.o**, and **.u**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
'**.e**'	Equal-to	**(post_zero & !NC).** This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
'**.ne**'	Not-Equal-to	**!(post_zero & !NC).** This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier '**.e**'.
'**.g**'	Greater-than	**(!post_sign & !post_zero & !NC).** This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.ge**'	Greater-than-or-equal-to	**((!post_sign | post_zero) & !NC).** This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.l**'	Less-than	**(post_sign & !post_zero & !NC).** This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.le**'	Less-than-or-equal-to	**((post_sign | post_zero) & !NC).** This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
'**.o**'	Overflow	**(OF).** This conditional modifier tests whether the computed result causes overflow – the computed result is outside the range of the destination data type. All other internal conditional signals are ignored.
'**.u**'	Unordered	**(NC).** This conditional modifier tests whether the computed result is a NaN (unordered). All other internal conditional signals are ignored.

5.8 Destination Hazard

GEN architecture has built-in hardware to avoid destination hazard.

Destination Hazard stands for the risk condition when multiple operations are trying to write to the same destination and the result of the destination may be ambiguous. This may or may not happen on GEN for two instructions with the same destination, or with destinations that have overlapped register region, depending on the ordering of the arrival of destination results. Let's consider two instructions in a thread with potential destination hazard. There may be other instruction between them as long as there is no instruction sourcing the same destination. Using register scoreboards, GEN hardware automatically takes care of the destination hazard by not issuing the second instruction until the destination scoreboard is cleared. However, for certain cases, in fact for most cases, such destination hazard indicated by the register scoreboard is false, causing unnecessary delay of instruction issuing. This may result in lower performance. The destination dependency control field in the instruction word {*NoDDClr*, *NoDDhk*} allows software to selectively override such hardware destination dependency mechanism. Such performance optimization hooks must be used with extreme caution. When it is not 100% certainty that it is a false destination hazard, programmer should reply on hardware to result the dependency.

As the destination dependency control field does not apply to *send* instruction, there is only one condition that a programmer may use the {*NoDDClr, NoDDChk*} capability.

- If none of the two instructions is *send*, there CANNOT be any destination hazard. This is because instructions within a thread are dispatched in order (single-issued) and the execution pipeline is in-order and has a fixed latency.

5.9 Non-present Operands

Some instructions do not have two source operands and one destination operand. If an operand is not present for an instruction the operand field in the binary instruction must be filed with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand <src0>. In this case, the second source operand <src1> must be set to null and also with the same type as the first source operand <src0>. It is a special case when <src0> is an immediate, as an immediate <src0> uses DW3 of the instruction word, which is normally used by <src1>. In this case, <src1> must be programmed with register file ARF and the same data type as <src0>.

5.10 Instruction Prefetch

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 instructions (128 bytes) beyond the end of the kernel program – possibly into the next memory page.   Although these instructions will not be executed, software must account for the prefetch in order to avoid invalid page access faults.   One possible (though inefficient) solution would be to pad the end of all kernel programs with 8 NOOP instructions. A more efficient approach would be to ensure that the page after all kernel programs is at least valid (even if mapped to a dummy page). Note that the **General State Access Upper Bound** field of the STATE_BASE_ADDRESS command can be used to prevent memory accesses past the end of the General State heap (where kernel programs must reside).

6. Exceptions

6.1 Introduction

The GEN Architecture defines a basic exception handling mechanism for several exception cases. This mechanism supports both normal operations such as extensions of the mask-stack depth, was well as illegal conditions.

The following exception-types are supported:

Type	Trigger / Source	Sync/Async Recognition
MaskStack Overflow / Underflow	Hardware	Synchronous (w/ special case for 'do'; see **Error! Reference source not found.**)
Software Exception	Thread code	Synchronous
Breakpoint	A bit in the instruction word	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver, and delivered to an EU as part of a new-thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's CR0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception-type recognition across hardware threads. The exception enables in the CR0.1 register are r/w, and thus can be enabled/disabled via software at anytime during thread execution.

The exception handling mechanism relies on the "system routine", a single subroutine which provides common exception handling for all threads on all EUs in the system. This system routine is defined per-context and is identified via a 32b System-IP (SIP) register in context state. At the time of each context switch, the appropriate SIP for that context is loaded into each EU, allowing each context to have custom implementation of exception handling routines if so desired.

6.2 Exception-Related Architectural Registers

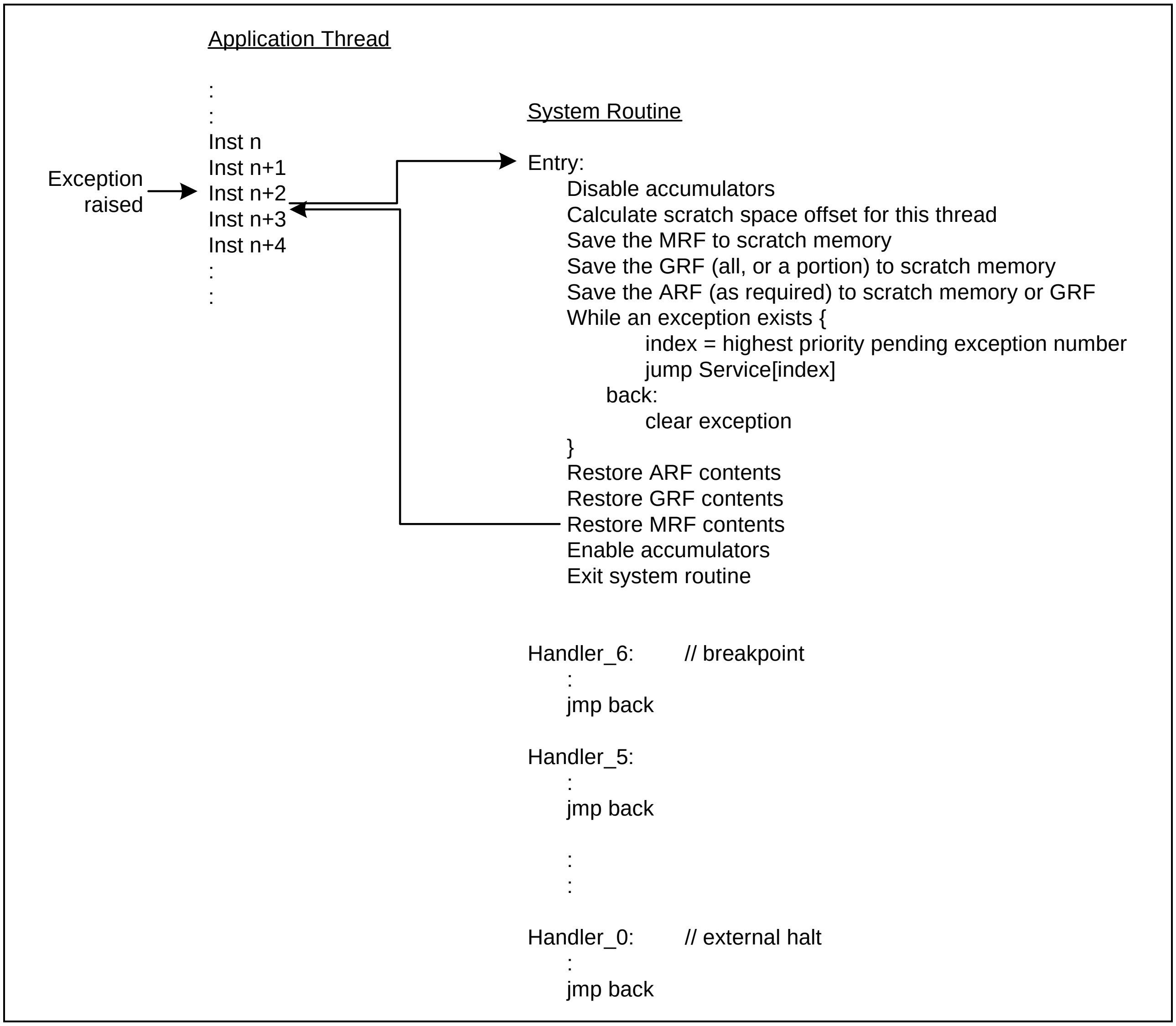
Exception-related registers are defined in architectural register CR0.0 through CR0.2. These registers are instantiated on a per-thread basis providing each hardware thread with unique control over exception recognition and handling. The registers provide the capability to mask exception types, determine the type of raised exception, provide storage the return address, and control exiting from the system routine back to the application thread.

Many of the bits in these registers are manipulated by both hardware and software. In all cases, the read/write operations by hardware and software occur at exclusive times in a thread's lifetime, thus there is no need for an atomic R-M-W operation when accessing these registers.

6.3 System Routine

6.3.1 General Flow of the System Routine

The following diagram illustrates the basic flow of exception handling and structure of the system routine.



```
Application Thread

:
:
Inst n
Inst n+1
Inst n+2
Inst n+3
Inst n+4
:
:
```

```
System Routine

Entry:
    Disable accumulators
    Calculate scratch space offset for this thread
    Save the MRF to scratch memory
    Save the GRF (all, or a portion) to scratch memory
    Save the ARF (as required) to scratch memory or GRF
    While an exception exists {
            index = highest priority pending exception number
            jump Service[index]
        back:
            clear exception
    }
    Restore ARF contents
    Restore GRF contents
    Restore MRF contents
    Enable accumulators
    Exit system routine


Handler_6:      // breakpoint
    :
    jmp back

Handler_5:
    :
    jmp back

    :
    :

Handler_0:      // external halt
    :
    jmp back
```

6.3.2 Invoking the System Routine

The system routine is invoked in response to a raised exception. Once an exception is raised, no further instructions from the application thread will be issued until the system routine has executed and returned control back to the application thread.

After a exception is recognized by hardware, the EU saves the thread's IP into its AIP register (CR0.2), an then moves the system routine offset, SIP, into the thread's IP register. At this point the next instruction to issue for that thread will be the first instruction of the system routine.

The system routine maintains the same execution priority, GRF and MRF register space, and thread state as that of the application thread from which it was invoked. Due to the assuming the same priority, there may be significant absolute time between exception being raised and the actual invocation of the system routine, as other higher priority threads within the EU continue to execute. From a thread's perspective, once an exception is recognized, the next instruction to issue is from the system routine.

At the time of system routine invocation, there may still be outstanding registers in-flight from the application thread. Depending on the instruction sequence in the system routine, an in-flight register may be referenced by the system routine and cause a register-in-flight dependency. These dependencies are honored by the system routine and may cause the system routine to be suspended until such time that the register retires.

Exception processing is non-nested within an system routine. If a future exception is detected while executing the system routine, the exception is latched into CR0.1, but does not cause a nested re-invocation of the system routine. The exception recognition hardware recognizes only one outstanding exception of each type; i.e. once a specific exception type is detected and latched in CR0.1, and until the exception is cleared, any further exception of that type will be lost.

Accumulators are not natively preserved across the system routine. To make sure the accumulators are in the identical state once control is returned to the application thread, the system routine must either set the Accumulator Disable bit of CR0.0 prior to using any instruction which modifies an accumulator, or manually save/restore the accumulators (to GRF registers or system thread scratch memory) around the system routine. Saving/restoring accumulators, including their extended precision bits, can be accomplished by a short series of mov's and shifts of the accumulator register. Also note the state of the Accumulator Disable bit itself must be preserved unless, by convention, the driver software limits its manipulation to only the system routine.

Further, upon system routine entry, the execution-related masks (Continue, Loop, If, and Active masks, contained in the Mask Register) will remain set as they were in the application thread. Thus only a subset of channels may be active for execution. To enable execution on all channels, the system routine may choose to use the instruction option 'NoMask', or may choose to set the mask registers to the desired value so long as it saves/restores the original masks upon system routine entry/exit.

Similarly there is no hardware mechanism to preserve flags, mask-stacks, or other architectural registers across the system routine. The system routine must ensure that these values are preserved (see Section 6.3.7 for related discussion).

6.3.3 Returning to the Application Thread

Prior to returning control to the application thread, the system routine should clear the proper Exception Status and Control bit in CR1. Failure to do so will force the thread's execution to re-enter the system routine prior to any further instructions being executed from that application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset prior to exit.)

The system routine may choose to loop under a single invocation of the system routine until all pending exceptions are serviced, or may choose to service exceptions one at a time (a simpler solution, but less efficient).

The system routine is exited, and control returned to the application thread, via a write to the Master Exception State and Control bit of CR0.0. Upon clearing this bit, the value of the AIP architectural register (CR0.2) is restored to the thread's IP register and, with no further exceptions pending, execution resumes that address. The system routine must follow any write to Master Exception State and Control bit with at least one simd-16 'nop' instruction to allow control to transition. Throughout the system routine, the AIP register maintains its value at the time the exception was raised unless directly modified by the system routine. (See the AIP register definition for specifics on the IP value saved to AIP).

6.3.4 System-IP (SIP)

The System IP (SIP) is a 16B-aligned 32b offset of the first instruction of the system routine, relative to the General State Base Address. It is set via the STATE_IP command to the command streamer. The upper 28b of the 32b address is automatically delivered to all GEN EUs.

When the system routine is invoked, the application thread's current IP is first saved into the AIP field of the thread's architectural register CR0.2. The SIP address is then loaded into the thread's IP register and execution continues within the system routine. Thus each invocation of the system routine has a common entry point at the first instruction of the system routine. Upon system routine completion, the value held in AIP is returned to IP and execution continues on the application thread at the place where the exception was recognized.

6.3.5 System Routine Register Space

The system routine uses the same GRF and MRF space at the thread which invoked it. As such all of the calling thread's registers and their contents are visible to the system routine. Further, the system routine must only use r0..r15 of the GRF, as a minimal thread may have requested and been allocated this few. If the system routine requires more registers than this, the driver should establish a higher minimum allocation to all threads. It should also be noted that the system routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

Only one 32b GRF location, R0.4, is reserved for system routine usage. This is sufficient to allow the system routine to calculate the appropriate offset of its private scratch memory in the larger system-scratch memory space (as dictated by binding table entry 254). The offset is left as a driver convention, but likely based on a combination of Thread and EU IDs (see example system handler in section 6.3.6). Other than the reserved R0.4 register field, there is no explicit GRF register space dedicated to the system routine, and any GRF needs must be accomplished via: (a) convention between the system routine an application thread, or (b) the system routine temporarily spilling the thread's GRF register contents to scratch memory, and restoration prior to system routine exit.

No persistent storage is natively allocated to the system routine, although a driver implementation may choose to carve out a piece of system scratch memory though it own convention.

Any parameter passing to the system routine (for use by s/w exceptions) is performed via the GRF based on a system-thread/application-thread convention.

6.3.6 System-Scratch Memory Space

There is a single unified system-scratch memory space per context shared by all EUs. It is anticipated that block is further partitioned into a unique scratch sub-space per-thread via convention implemented in the system routine, with a each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread is based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architectural register SR0.0.

Per_Thread_Block_Size = System_Scratch_Block_Size / (EU_Count * Thread_Per_EU);

Offset = (SR0.0.EID * Threads_Per_EU + SR0.0.TID) * Per_Thread_Block_Size;

where in GEN...
Threads_Per_EU = 4
EU_Count = 8
System_Scratch_Block_Size is a driver choice

Access to the system-scratch memory is performed through the Data Port via linear single-register or block-based read/write messages. The driver may choose to use any binding table index for system-scratch surface description. As a practical matter, the same index is expected to be used across all binding tables, as the index is typically hard coded in dataport messages used within the system routine coupled with the fact that a single system instance routine is used for all threads. Read/write messages to the Data Port contain the address of the binding table (provided in R0 of all threads) and an offset, from which the Data Port calculates the final target address.

It is expected that the system-memory block is allocated by the driver at context-create time and remains persistent at a constant memory address throughout the context's lifetime.

6.3.7 Conditional Instructions Within System Routines

It is expected that most, if not all, control flow with in the system routine is scalar in nature. If so, the system routine should set SPF (Single Program Flow, CR0.0) to enable scalar branching. In this mode, conditional/loop instructions do not update the mask-stacks and therefore do not have restrictions on their use nor require the save/restore of hardware mask-stack registers.

If SIMD branching is desired within the system routine, special considerations must be taken. Upon entry to the system routine, the depth of the mask-stacks is unknown at that point, and may be near-full. If so, a subsequent conditional instruction and its associated mask 'push' may cause a stack overflow. This would generate an exception-within-the-system-routine, an unsupported occurrence. To prevent this, if the system routine uses SIMD conditional instructions, it must save the mask-stacks prior to the first SIMD conditional instruction, and restore them after the last SIMD conditional instruction. As a general solution, it may be easiest simply to implement the save/restore as part of the entry/exit code sequence, using an available GRF register-pair as storage location. Once saved, the stacks should be reset to their empty condition, namely depth = 0 and top-of-stack = 0xFFFFFFFF.

6.3.8 Messages in System Routines

The system routine uses the same MRF space as the thread on whose behalf the system routine was invoked. To allow the thread to resume with the same state as prior to the system routine invocation, the thread's MRF contents must be preserved across a system routine invocation. If the system routine requires MRF space for messages, it must manually save and restore the MRF locations which it uses.

Note that the MRF can only be used as an instruction's destination register, not a source. Therefore there is no option to save the MRF to the GRF. Thus the system routine should save the MRF contents to its dedicated scratch space. By convention it is recommended that MRF register m0 be reserved for system-thread use. This allows the system routine enough space to construct an initial Data Port write message starting at m0 without corrupting any MRF registers, facilitating a complete save/restore of the MRF by the system-thread.

6.3.9 Use of 'NoDDClr'

The GEN instruction word defines an instruction option 'NoDDClr' which overrides the native register dependency clearing mechanism of the typical instruction. When specified, 'NoDDClr' does not clear, at register writeback time, the dependency placed on the destination register of the instruction. Use of this mechanism may provided increased performance when the kernel can guarantee no dependency issues between instructions, but may cause issues with exception handling in some circumstances as discussed here.

Typically 'NoDDClr' is used in an instruction series to enable a sequence of writes to sub-fields of a GRF register without paying a dependency penalty on each instruction. In this case, 'NoDDClr' and 'NoDDChk' are used across an instruction sequence to allow the first instruction to set the destination dependency, interior instructions to write to the GRF register w/o dependency checks, and the last instruction clear the dependency. (This sequence is referred to as a 'NoDDClr' code block going forward). By only allowing the last instruction to clear the dependency, program execution is prevented from going beyond a certain point until all writes of that sequence are known to retire.

The problem arises should an exception be raised within a 'NoDDClr' code block. In this case, there exists the potential for the system routine to hang while attempting to save/restore the code blocks destination register, as the outstanding dependency on that register will not clear until the final instruction of the block is executed – sometime after the system thread returns. Should the system routine attempt to use that register, the system routine will hang waiting on a dependency to clear from an instruction which has not yet been issued.

This is a known condition and will in some cases not allow the full GRF contents to be externally visible in system routine scratch space during a break or halt exception. To minimize the number of cases of such, guidelines are provided below for consideration. (Note that these are general guidelines, some of which can be alleviated through careful coding and register usage conventions and restrictions.)

- 'NoDDClr' code blocks should only be used where absolutely necessary.
- Instructions which may generate exceptions should not be placed within 'NoDDClr' blocks. This includes most conditional branch instructions (if, do, while, ...) as well as breakpoints explicitly in the instruction stream.
- If possible, use 'NoDDClr' on registers high in the thread's register allocation (e.g. r120), thus even if a system routine hang occurs, as much of the GRF is visible as possible. (Note this would also require the system routine to update the progress of the GRF dump, perhaps with each GRF block written, or to initialize the system routine's scratch space to a known value, to be able to distinguish valid/locations from unwritten locations).

Also a driver implementation may consider a “disable-NoDDclr” option in which jitted code does not use the ‘NoDDClr’ capability. In this case, there is no change to the code that is jitted other than removal of the ‘NoDDClr’ instruction option. The code executes as normal, but with a higher number of thread switches in what would have been a NoDDClr code block.

6.4 Exception Descriptions

6.4.1 ‘Illegal’ opcode

The GEN ISA defines a single ‘illegal’ opcode. The byte value of the ‘illegal’ opcode is selected to be 0x00 due to it being a likely byte-value encountered by a wayward instruction pointer value. The ‘illegal’ instruction raises an exception prior to issue and operates as a ‘nop’ when issued down the execution pipeline. (Specifically, the opcode acts a ‘nop’, although other non-opcode instruction attributes still apply).

6.4.2 Undefined opcode

All undefined opcodes in the 8b opcode space are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing it to the defined ‘illegal’ opcode. The offending instruction, should it eventually be issued down the execution unit’s pipeline, generates an ‘illegal opcode’ exception as described in section 6.4.1. Note that the memory location of the offending opcode remains modified and may be queried if desired to determine its original value.

6.4.3 Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered through of the Software Exception Set and Clear bit of CR0.1. Sub-function determination and parameter passing into and out-of the exception handler is left to convention between the system-thread and application-thread. The thread’s AIP instruction pointer is incremented prior to system-routine entry, therefore causing execution to resume at the subsequent application-thread instruction when the system routine is exited.

6.4.4 Breakpoint

A single-stepping capability may be implemented by leaving the “Breakpoint Exception Status and Control” set, and clearing the Breakpoint Suppress field prior to system routine exit. This combination causes the instruction associated with the breakpoint to be reissued, this time with the breakpoint suppressed, and then re-entry to the system routine prior to the subsequent instruction due to the lingering breakpoint exception that remained un-cleared.

6.4.5 External Halt

A ‘halt’ exception may occur upon direction manipulation of a MMIO bit by driver software. The halt exception is sent to all EUs simultaneously (although no guarantee is made as to recognition in identical clocks). An EU recognizes this condition internally by generating an External Halt exception. A likely implementation of a handling routine would dump the thread’s state to programmer-visible memory (such as the system routine’s scratch space) for inspection purposes. Although generally recognized within a few clocks, there is no specification as to the latency between triggering the Halt condition and it being recognized by an EU.

6.5 Events Which Do Not Generate Exceptions

The following conditions are either not recognized or do not generate an exception.

Illegal Instruction Format

This includes malformed instructions in which the opcode is legal, but the source or destination operands, or instruction attributes are not compliant with the instruction specification. There is no direct hardware support to detect these cases and the outcome of issuing a malformed instruction is undefined. Note that GEN does not support self-modifying code, therefore the driver has an opportunity to detect such cases before the thread is placed in service.

Malformed Message

A messages contents, destination registers, lengths, and descriptors are not interpreted in anyway by the execution units. Errors in specifying any of these fields do not raise exceptions in the execution unit but may be detected and reported by the shared functions.

GRF Register Out-of-Bounds

Unique GRF storage is allocated to each thread which, at a minimum, satisfies that the register requirements specified in the thread's declaration. References to GRF register numbers beyond that called for in the thread's declaration do not generate exceptions. Depending on implementation, out-of-bounds register numbers may be remapped to r0..r15, although this functionality should not be relied upon by the thread. The hardware guarantees the isolation of each threads register space, thus there is no possibility of direct register manipulation from an out-of-bounds register access.

MRF Register Out-of-Bounds

A fixed amount of MRF register space is allocated for each thread, namely m0 through m23. References to MRF registers beyond m23 do not generate exceptions. Depending on implementation details, out-of-bounds register numbers may alias to in-bounds register numbers, although this functionality should not be relied upon by the thread.

Hung Thread

There is no hardware mechanism in the execution units to detect a hung thread, and should it occur, the thread remains hung indefinitely. It is the expectation that one or more hung threads will eventually cause the driver to recognize a context timeout and take appropriate recovery action.

Instruction Fetch Out of Bounds

The GEN EUs implement a full 32b instruction address range (with the 4 lsb's don't care), making it possible for a thread to attempt to jump to any 16B aligned offset in the 32b address space. The EU itself does not provide any type of address checking on its instruction request stream sent to the memory/cache hierarchy, although various memory address related error conditions are reported through the Memory Interface Registers (specifically "Page Table Error Register").

FPU Math Errors

The EU's floating point units have defined behavior for traditional floating point errors and do not generate exceptions. Therefore there is no support for signaling FPU math errors as exceptions.

Destination Register Overflow

Depending on source operand contents, destination register size, and operation being performed, overflows may occur in the EU’s pipeline. These are not flagged as exceptions and software must explicitly check the overflow bit in the thread’s architectural register if overflow is a concern.

6.6 System Handler Example

The following code sequence illustrates some concepts of the system routine. It is intended to be just a shell, without getting into the specifics of each exception handler. The example frees enough MRF and GRF space to get the routine started, then jumps to the handler for the specific exception. Many other implementations are also valid, including single exception servicing (as opposed to looping) per invocation, and saving only the GRF or MRF space required by the exception being serviced.

```
#define ACC_DISABLE_MASK         0xFFFFFFFD
      #define MASTER_EXCP_MASK   0x7FFFFFFF
      #define SYSROUTINE_SCRATCH_BLKSIZE  16384          //for
example

      // --- SharedFunc IDs ---
      #define DPR                0x04000000
      #define DPW                0x05000000

      // --- message lengths ---
      #define ML5                0x00500000
      #define ML9                0x00900000

      // --- response lengths ---
      #define RL0                0x00000000
      #define RL4                0x00040000
      #define RL8                0x00080000

      // --- dataport block sizes ---
      #define BS1_LOW            0x0000
      #define BS1_HIGH           0x0100
      #define BS2                0x0200
      #define BS4                0x0300

      // --- Scratch Layout ---
      #define SCR_OFFSET_MRF     0
      #define SCR_OFFSET_GRF     512      // + 16 reg
      #define SCR_OFFSET_ARF     512 + 4096    // + 16 + 128
reg
```

```
      // --- Write Dataport constants ---
      // target=dcache, type= oword_block_wr,
binding_tbl_offset=0
      #define DPW                0x000

      // --- Read Dataport constants ---
      // target=dcache, type= oword_block_rd,
binding_tbl_offset=0
      #define DPR                0x000

Sys_Entry:

      // --- calc scratch offset for this thread into r0.4 ---
      shr   (1) r0.4 sr0.0:uw 6           {NoMask}
      add   (1) r0.4 r0.4 sr0.0:ub             {NoMask}
      mul   (1) r0.4 r0.4 SYSROUTINE_SCRATCH_BLKSIZE
      {NoMask}

      // --- setup m0 w/ block offset
      mov   (8) m0 r0                     {NoMask}

      // --- save mrf 7...0; (may choose to save the whole mrf)
      add   (1) m0.2 r0.4 SCR_OFFSET_MRF            {NoMask}
      send  (8) null m0 null DPW|ML9|RL0            {NoMask}

      // --- save mrf 8...15; (optional; req’ed if sys-routine
stays w/in mrf7-0)
      mov   (8) m7 r0                     {NoMask}
      add   (1) m7.2 r0.4 (SCR_OFFSET_MRF + 256)    {NoMask}
      send  (8) null m7 null DPW|ML9|RL0            {NoMask}

      // --- save r0..r1 to system scratch ---
      // --- (Note: done as a single register to guarantee
external
      // ---  visibility — see “Use of ‘NoDDClr’” in Excpetions
Bspec chapter
      mov  (16)  m1 r0                    {NoMask}
      send (8)  m0 null null DPW|ML2|RL0            {NoMask}

      // --- save r2..r3 to free some room
      mov  (16) m3 r2                     {NoMask}
      add  (1)  m0.2 r0.4 SCR_OFFSET_GRF + 64
      {NoMask}
      send (8)  m0 null null DPW|ML4|RL0            {NoMask}
      // --- save r4..r7 to free some room (optional, depending
on needs)
      mov  (16) m8 r4                     {NoMask}
      mov  (16) m10 r6                    {NoMask}
      add  (1)  m7.2 r0.4 (SCR_OFFSET_GRF + 128)    {NoMask}
      send (8)  m7 null null DPW|ML5|RL0            {NoMask}

      // --- save r8..r11 to free some room (optional,
depending on needs)
      mov  (16) m1 r8                     {NoMask}
      mov  (16) m3 r10                    {NoMask}
      add  (1)  m0.2 r0.4 (SCR_OFFSET_GRF + 256)    {NoMask}
      send (8)  m0 null null DPW|ML5|RL0            {NoMask}

      // --- save r12..r15 to free some room    (optional,
depending on needs)
      mov  (16) m8 r12                    {NoMask}
      mov  (16) m10 r14                   {NoMask}
      add  (1)  m7.2 r0.4 (SCR_OFFSET_GRF + 384)    {NoMask}
      send (8)  m7 null null DPW|ML5|RL0            {NoMask}

      // --- save ARF registers (optional, depending on use) --
-
      // flags, maskstacks, others...

      // --- save f0.0 ---
       mov (1)  r1.0:uw f0.0              {NoMask}

Next: // --- exceptions pending? If not, exit ---
      cmp.e (1) f0.0 cr0.4:uw 0:uw             {NoMask}
      (f0.0) mov (1) IP EXIT              {NoMask}

      // --- find highest priority exception ---
      lzd  (1) r1.1:uw cr0.4:uw           {NoMask}

      // --- jumptable to service routine ---
      jmpi (1)  r1.1:uw                   {NoMask}
      mov  (1)  IP CRService_0            {NoMask}
      mov  (1)  IP CRService_1            {NoMask}
      mov  (1)  IP CRService_2            {NoMask}
      // :
      // :
      // :
      mov  (1)  IP CRService_15           {NoMask}
      mov  (1)  IP Next
Service_0:
      // clear exception from CR0.1
      // perform service routine
      // jump to exit (or if looping on exceptions, go to next
loop)

      // :
      // :

Service_15:
      // clear exception from CR0.1
      // perform service routine
      // jump to exit (or if looping on exceptions, go to next
loop)

Exit:
      // --- restore f0.0 ---

      // --- restore ARF registers (as required) ---
      // flags, maskstacks, others...

      // --- restore r12..r15 ---
      // --- restore r8..r11 ---
      // --- restore r4..r7 ---
      // --- restore r0..r3 ---

      // --- restore m8..m15 ---
      // --- restore m0..m7 ---

      // --- clear Master Exception State bit in CR0.0
      and  (1)  cr0.0 cr0.0 MASTER_EXCP_MASK
      nop  (16)
```

Below is a code sequence to programmatically clear the GRF scoreboard in the case of a timeout waiting on a register that may never return:

```
          // At this point, all we know is we have a hung thread.
To get around
          // any hung dependency, we can walk the GRF using
NoDDChk, using execution mask
          // of f0 = 0 so we don’t touch the register contents.

Clear_Dep:
               mov f0  0x00
          (f0) mov r0 0x00 {NoDDChk}
          (f0) mov r1 0x00 {NoDDChk}
          (f0) mov r2 0x00 {NoDDChk}
               ...
               ...
          (f0) mov r127 0x00 {NoDDChk}

          // GRF scoreboard now cleared.
```

7. Instruction Set Summary

7.1 Instruction Set Characteristics

7.1.1 SIMD Instructions and SIMD Width

GEN instructions are SIMD (single instruction multiple data) instructions. The number of data elements per instruction, or the execution size, depends on the data type. For example, the execution size for GEN instructions operating on 256-bit wide vectors can be up to 8 for 32-bit data types, and be up to 16 for 16-bit data. The maximum execution size for GEN instructions for 8-bit data types is also limited to 16.

An instruction compression mode is supported for 32-bit instructions (including mixed 32-bit and 16-bit data computation). A compressed GEN instruction works on twice as many SIMD data as that for a non-compressed GEN instruction. Non-compressed instructions are also referred to as ‘native’ instructions. A compressed instruction is converted into two native instructions by the instruction dispatcher in the EU.

GEN instructions are executed on a narrower SIMD execution pipeline. Therefore, GEN native instructions take multiple execution cycles to complete. See **Error! Reference source not found.** for parameters for difference device hardware.

7.1.2 Instruction Operands and Register Regions

Majority of GEN instructions may have up to three operands, two sources and one destination. Each operand is able to address a register region. Source operands support negate and absolute modifier and channel swizzle, and the destination operand supports channel mask.

Dual destination instructions are also supported (four-operand instructions in a general sense): One case is for the implied destination – flag register, where the conditional modifiers and the predicate modifiers may apply. Another case is the message header creation (implied move or implied assembling of the header) in the *send* instruction.

Each execution channel contains an accumulator that is wider than the input data to support back-to-back accumulation operations with increased precision. The added precision (see accumulator register description in Execution Environment chapter) determines the maximum number of accumulations before possible overflow. The accumulator can be pre-loaded through the use of ***mov***. It can also be pre-loaded by arithmetic instructions such as ***add***, ***mul***, since the result of these instructions can go to the accumulator. The accumulator registers are per thread and therefore safe for thread switching.

Register access can be direct or register-indirect. Register-indirect register access uses address registers plus an immediate offset term to compute the register addresses, and only applies to the first source operand (src0) and/or the destination operand.

There is one address register that contains 8 sub-registers. Each sub-register contains a 16-bit unsigned value. The leading two sub-registers form a special doubleword that can be used as the descriptor for the send instruction.

Source operand can also be immediate value (also referred to as inline constants). For instructions with two source operands, only the second operand <src1> is allowed to be immediate. For instructions with only one source operand, the source operand <src0> is used and it can be an immediate.

An immediate source operand can be a scalar value of specified type up to 32-bit wide, which is replicated to create a vector with length of Execution Size. An immediate operand can also be a special 32-bit vector with 8 elements each of 4-bit signed integer value, or a 32-bit vector with 4 elements each of 8-bit restricted float value.

7.1.3 Instruction Execution

It is implied that all instructions operate across all channels of data unless otherwise specified either via destination mask, predication, execution mask (caused by SIMD branch and loop instructions), or execution size.

Instruction execution size can be specified per instruction, from scalar (*ExecSize* = 1) up to the maximal execution size supported for the data type, with the restriction that execution size can only be in power of 2.

7.2 Instruction Machine Formats

This section shows the machine formats of the GEN instruction set. The instructions in GEN architecture have fixed length of 128 bits. Out of the 128 bits, there are 120 bits in use, and the remaining bits are reserved for future extensions. One instruction consists of instruction fields that control various stages of execution of the instruction. These fields are roughly groups into the 4 doublewords as the following.

- Instruction Operation Doubleword (DW0) contains the opcode and other general instruction control fields.
- Instruction Destination Doubleword (DW1) contains the destination operand (<dst>) and the register file and type of source operands.
- Instruction Source-0 Doubleword (DW2) contains the first source operand (<src0>) and flag register number
- Instruction Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>).

The following table depicts the details of the organization of fields in the 128-bit instruction word based on the Addressing Mode and Access Mode of an instruction. Definitions for individual instruction fields are provided in the following sections.

The ***send*** instruction is shown in the talbe as it has some unique instruction fields. For example, the message descriptor (plus EOT) occupies the whole DW3, and the immediate destination register overlaps with the Conditional Modifier field. The rest of fields in DW0-3 follows the definition on the left, depending on Addressing Mode and Access Mode of the ***send*** instruction.

The ***math*** and ***conditional branch*** instruction are also shown in the table as they have some unique instruction fields.

Not shown is for immediate operands. When an immediate source is present in an instruction, it always occupies the whole DW3 with a 32-bit value.

Support for indirect addressing for <src1>, as shown by the gray areas in Table 4-2 is device dependent. See Table 5-23 (Indirect source addressing support available in device hardware) in ISA Execution Environment for details.

GEN5P75 Instruction Format [DevILK]

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	AddrMode = Direct		AddrMode = Indirect		SEND		MATH	Branch (2offsets)	Branch (1offset)	Imm Src
					AccessMode = Align16	AccessMode = Align1	AccessMode = Align16	AccessMode = Align1	MsgDesc Imm	MsgDesc Reg				
	1	127	127	1					EOT					
	2	126	125	2										
	4	124	121	4										
	4	120	117	4	Src1.VertStride									
	1	116	116	1										
	2	115	114	2		Src1.Width		Src1.Width						
	2	113	112	2	Src1.ChanSel[7:4]	Src1.HorzStride	Src1.ChanSel[7:4]	Src1.HorzStride				UIP[15:0]		
	1	111	111	1	Src1.AddrMode									
	2	110	109	2	Src1.SrcMod									
	3	108	106	3			Src1.AddrSubRegNum							
	5	105	101	5	Src1.RegNum [7:0]									
	1	100	100	1	Src1.SubRegNum [4]		Src1.AddrImm [9:4]							
3	4	99	96	4	Src1.ChanSel[3:0]	Src1.SubRegNum [4:0]	Src1.ChanSel[3:0]	Src1.AddrImm [9:0]	Imm[28:0]	Reg32	Same	JIP[15:0]	JIP[15:0]	Imm[31:0]
	5	95	91	5										
	1	**90**	**90**	**1**	**FlagRegNum**									
	1	**89**	**89**	**1**	**FlagSubRegNum**									
	4	88	85	4	Src0.VertStride									
	1	84	84	1										
	2	83	82	2		Src0.Width		Src0.Width						
	2	81	80	2	Src0.ChanSel[7:4]	Src0.HorzStride	Src0.ChanSel[7:4]	Src0.HorzStride						
	1	79	79	1	Src0.AddrMode									
	2	78	77	2	Src0.SrcMod									
	3	76	74	3			Src0.AddrSubRegNum							
	5	73	69	5	Src0.RegNum [7:0]									
	1	68	68	1	Src0.SubRegNum [4]		Src0.AddrImm [9:4]							
2	4	67	64	4	Src0.ChanSel[3:0]	Src0.SubRegNum [4:0]	Src0.ChanSel[3:0]	Src0.AddrImm [9:0]	Same		Same	Same	Same	Same
	1	63	63	1	Dst.AddrMode									
	2	62	61	2		Dst.HorzStride		Dst.HorzStride						
	3	60	58	3			Dst.AddrSubRegNum							
	5	57	53	5	Dst.RegNum [7:0]									
	1	52	52	1	Dst.SubRegNum [4]		Dst.AddrImm [9:4]							
	4	51	48	4	Dst.ChanEn[3:0]	Dst.SubRegNum [4:0]	Dst.ChanEn[3:0]	Dst.AddrImm [9:0]	Same		Same	Same	Same	Same
	1	47	47	1	NibCtrl									
	3	46	44	3	Src1.SrcType									
	2	43	42	2	Src1.RegFile									
	3	41	39	3	Src0.SrcType									
	2	38	37	2	Src0.RegFile									
	3	36	34	3	Dst.DstType									
1	2	33	32	2	Dst.RegFile				Same		Same	Same	Same	Same
	1	31	31	1	Saturate									
	1	30	30	1	DebugCtrl									
	1	29	29	1	CmptCtrl									
	1	28	28	1	AccWrCtrl				Same		Same	Same	Same	
	4	27	24	4	CondModifier				SFID[3:0]		FC[3:0]	MBZ	MBZ	
	3	23	21	3	ExecSize									
	1	20	20	1	PredInv									
	4	19	16	4	PredCtrl									
	2	15	14	2	ThreadCtrl									
	2	13	12	2	QtrCtrl									
	2	11	10	2	DepCtrl									
	1	9	9	1	WECtrl									
	1	8	8	1	AccessMode				Same		Same	Same	Same	Same
	1	7	7	0	(reserved for Opcode)									
0	7	6	0	7	Opcode				Same		Same	Same	Same	Same

The 3-src instructions have the following restrictions compare to the 1-src/2-src instructions.

- The only supported instructions are: LRP, MAD, BFE, BFI2
- Only GRF register allowed for sources, and only GRF/MRF register allowed for destination
- Subregister number can only go down to DWord granularity.
- Must be Align16, uses Align16 style swizzling, with extra replication control. No other regioning support.

7.2.1 Common Instruction Fields

7.2.1.1 1-src and 2-src Instructions

As shown in Table 7-1, the meanings (encoding) of certain bit fields in the 128-bit instruction word varies depending on the values of other bit fields.

Table 7-1 provides the definition of common fields in the instruction word. The ‘Width’ column specifies the width of the field in bits. These common fields will be referred to later in describing the fields of different doublewords of the instruction. The definition for fields that have unique representation can be found in its corresponding doubleword of the instruction.

Table 7-1. Definitions of Common Instruction Fields

Field	Description	Width
CondModifier	**Conditional Modifier.** This field sets the flag register based on the internal conditional signals output from the execution pipe such as sign, zero, overflow and NaNs, etc. If this field is set to 0000, no flag registers are updated. Flag registers are not updated for instructions with embedded compares. This field may also be referred to as the *flag destination control* field. This field applies to all instructions except *send, sendc, and math*. 0000 = Do not modify the flag register (normal) 0001 = Zero or Equal (‘**.z**’ or ‘**.e**’) 0010 = Not Zero or Not Equal (‘**.nz**’ or ‘**.ne**’) 0011 = Greater-than (‘**.g**’) 0100 = Greater-than-or-equal (‘**.ge**’) 0101 = Less-than (‘**.l**’) 0110 = Less-than-or-equal (‘**.le**’) 0111 = Reserved 1000 = Overflow (‘**.o**’) 1001 = Unordered with Computed NaN (‘**.u**’) 1010 -1111 = Reserved	4
AddrMode	**Addressing Mode.** This field determines the addressing method of the operand. When it is cleared, the register address of the operand is directly provided by bits in the instruction word. It is called a direct register addressing mode. When it is set, the register address of the operand is computed based on the address register value and an address immediate field in the instruction word. This is referred to as a register-indirect register addressing mode. This field applies to the destination operand and the first source operand, <src0>. Support for <src1> is device dependent. See Table XX (Indirect source addressing support available in device hardware) in ISA Execution Environment for details. 0 = “Direct”. Direct register addressing 1 = “Register-Indirect” (or in short “Indirect”). Register-indirect register addressing	1

Field	Description	Width
RegNum	**Register Number.** This field provides the register number for the operand. For GRF or MRF register operand, it provides the portion of register address aligning to 256-bit. For an ARF register operand, this field is encoded such that MSBs identify the architecture register type and LSBs provide its register number. This field together with the corresponding *SubRegNum* field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [12:5] of the byte address, while *SubRegNum* field provides bits [4:0]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U8, if *RegFile* = **GRF**. 0x00 to 0x7F = Register number in the range of [0, 127] 0x80 to 0xFF = Reserved Format = U8, if *RegFile* = **MRF**. 0x00 to 0x0F = Register number in the range of [0, 15] 0x10 to 0xFF = Reserved Format = 8-bit encoding, if *RegFile* = **ARF**. This field is used to encode the architecture register as well as providing the register number. See GEN Execution Environment chapter for details.	8
SubRegNum	**Sub-Register Number.** This field provides the sub-register number for the operand. For GRF or MRF register operand, it provides the byte address within a 256-bit register. For an ARF register operand, this field also provides the sub-register number according to special encoding for the given architecture register. This field together with the corresponding *RegNum* field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [4:0] of the byte address, while *RegNum* field provides bits [12:5]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U5, if *RegFile* = **GRF** or **MRF** 0x00 to 0x1F = Sub-Register number in the range of [0, 31] Format = 5-bit encoding, if *RegFile* = **ARF**. This field is used to encode the architecture register as well as providing the register number. See GEN Execution Environment chapter for details.	5

Field	Description	Width
AddrSubRegNum	**Address Sub-Register Number.** This field provides the sub-register number for the address register. The address register contains 8 sub-registers. The size of each sub-register is one word. The address register contains the register address of the operand, when the operand is in register-indirect addressing mode. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed. Format = U3 0x00 to 0x07 = Address Sub-Register number in the range of [0, 7]	3
AddrImm	**Address Immediate.** This field provides the immediate value in unit of byte to be added to the address register in order to compute the register address (byte-aligned region origin) for the operand. It is a 10-bit signed integer in 2's compliment form. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed. *Note: that the address immediate field may not be able to cover the whole GRF register range for a thread, as the maximum GRF register space for a thread is 4KB.* Format = S9 Valid range: [-512, 511]	10
SrcMod	**Source Modifier.** This field specifies the numerical modification to a source operand. The value of each data element of a source operand can optionally have its absolute value taken and/or its sign inverted prior to delivery to the execution pipe. The absolute value is prior to negate such that a guaranteed negative value can be produced. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 00 = No modification (normal) 01 = **"(abs)"**. Absolute 10 = **"–"**. Negate 11 = **"–(abs)"**. Negate of the absolute (forced negative value)	2

Field	Description	Width
VertStride	**Vertical Stride.** The field provides the vertical stride of the register region in unit of data elements for an operand. Encoding of this field provides values in power of 2, ranging from 0 to 32 elements. Larger values are not supported due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total). Special encoding 1111b (0xF) is only valid when the operand is in register-indirect addressing mode (*AddrMode* = 1). If this field is set to 0xF, one or more sub-registers of the address registers may be used to compute the addresses. Each address sub-register provides the origin for a row of data element. The number of address sub-registers used is determined by the division of *ExecSize* of the instruction by the *Width* fields of the operand. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. For **Align16** access mode, only encodings of 0000 and 0011 are allowed. Other codes are reserved. *Note 1: Vertical Stride larger than 32 is not allowed due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total).* *Note 2: In **Align16** access mode, as encoding 0xF is reserved, only single-index indirect addressing is supported.* *Note 3: If indirect address is supported for <src1>, encoding 0xF is reserved for <src1> – only single-index indirect addressing is supported.* 0000 = 0 Elements 0001 = 1 Element 0010 = 2 Elements 0011 = 4 Elements 0100 = 8 Elements 0101 = 16 Elements (applies to byte or word operand only) 0110 = 32 Elements (applies to byte operand only) 0111-1110 = Reserved 1111 = **VxH or Vx1 mode** (only valid for register-indirect addressing in **Align1** mode)	4
Width	**Width.** This field specifies the number of elements in the horizontal dimension of the region for a source operand. This field cannot exceed the *ExecSize* field of the instruction. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 000 = 1 Elements 001 = 2 Elements 010 = 4 Elements 011 = 8 Elements 100 = 16 Elements 101-111 = Reserved	3

<table>
<tr><th>Field</th><th>Description</th><th>Width</th></tr>
<tr><td><b><i>HorzStride</i></b></td><td><b>Horizontal Stride.</b> This field provides the distance in unit of data elements between two adjacent data elements within a row (horizontal) in the register region for the operand.
This field applies to both destination and source operands.
This field is not present for an immediate source operand.
00 = 0 Elements
01 = 1 Element
10 = 2 Elements
11 = 4 Elements</td><td>2</td></tr>
<tr><td><b><i>Imm32</i></b></td><td><b>32-bit Immediate.</b> The 32-bit immediate data field for the operand. It may contain any legal bit pattern for its associated type. Only one 32-bit immediate value may be present in an instruction, therefore binary operations only support <src1> as an immediate value.
The low order bits are directly used when fewer than 32-bits are needed to describe the desired type; the 32-bits are not coerced into the designated type.
For UW and W data types, programmer is required to replicate the lower word to the upper word of this field.
This field only applies to the last source operand.
Signed and unsigned byte integer data types are not supported for an immediate operand.
Valid ranges according to data type:
<table>
<tr><th>Immediate Data Type</th><th>Valid Range (inclusive)</th></tr>
<tr><td>F</td><td>$[0\ldots\pm1.0*2^{-128\ldots127}]$</td></tr>
<tr><td>UW</td><td>[0, 65535]</td></tr>
<tr><td>W</td><td>[-32768, 32767]</td></tr>
<tr><td>UD</td><td>$[0, 2^{32}-1]$</td></tr>
<tr><td>D</td><td>$[-2^{31}, 2^{31}-1]$</td></tr>
<tr><td>VF</td><td>[0, ±0.125…±31]</td></tr>
<tr><td>V</td><td>[-8, 7]</td></tr>
</table></td><td>32</td></tr>
<tr><td><b><i>ChanEn</i></b></td><td><b>Channel Enable.</b> Four channel enables are defined for controlling which channels will be written into the destination region. These channel mask bits are applied in a modulo-four manner to all <i>ExecSize</i> channels. There is 1-bit Channel Enable for each channel within the group of 4. If the bit is cleared, the write for the corresponding channel is disabled. If the bit is set, the write is enabled. Mnemonic for the bit being set for the group of 4 is “x”, “y”, “z”, and “w”, respectively, where “x” corresponds to Channel 0 in the group and “w” corresponds to channel 3 in the group.
This field only applies to destination operand.
This field is only present in <b>Align16</b> mode.
0 = Write Disabled
1 = Write Enabled (normal)</td><td>4</td></tr>
</table>

Field	Description	Width
ChanSel	**Channel Select.** This field controls the channel swizzle for a source operand. The normally sequential channel assignment can be altered by explicitly identifying neighboring data elements for each channel. Out of the 8-bit field, 2 bits are assigned for each channel within the group of 4. ChanSel[1:0], [3.2], [5.4] and [7,6] are for channel 0 (“x”), 1 (“y”), 2 (“z”), and 3 (“w”) in the group, respectively. For example with an execution size of 8, *r0.0**<4>.zywz:f*** would assign the channels as follows: Chan$_0$ = Data$_2$, Chan$_1$ = Data$_1$, Chan$_2$ = Data$_3$, Chan$_3$ = Data$_2$; Chan$_4$ = Data$_6$, Chan$_5$ = Data$_5$, Chan$_6$ = Data$_7$, Chan$_7$ = Data$_6$. This field only applies to source operand. This field is only present in **Align16** mode. It is not present for an immediate source operand. The 2-bit Channel Selection field for each channel within the group of 4 is defined as the following. 00 = **“x”**. Channel 0 is selected for the corresponding execution channel 01 = **“y”**. Channel 1 is selected for the corresponding execution channel 10 = **“z”**. Channel 2 is selected for the corresponding execution channel 11 = **“w”**. Channel 3 is selected for the corresponding execution channel	8
MsgDscpt31	**Message Description.** This field, containing 31-bit immediate values, provides the description of the message to be sent. This field only applies to the *send* instruction. It is not present for other instructions. The meaning of the field depends on the type of message as well as the message shared function target. Format: U31	31
EOT	**End of Thread.** This field controls the termination of the thread. For a *send* instruction, if this field is set, EU will terminate the thread and also set the EOT bit in the message sideband. This field only applies to the *send* instruction. It is not present for other instructions. 0 = The thread is not terminated 1 = **EOT**	1

7.2.1.2 3-src Instructions

The table in this section describes the encoding for the common fields for the 3-src instructions format.

Table 7-2. Definitions of Common Instruction Fields

Field	Description	Width
CondModifier	**Conditional Modifier.** This field sets the flag register based on the internal conditional signals output from the execution pipe such as sign, zero, overflow and NaNs, etc. If this field is set to 0000, no flag registers are updated. Flag registers are not updated for instructions with embedded compares. This field may also be referred to as the *flag destination control* field. This field applies to all instructions except *send, sendc, and math*. 0000 = Do not modify the flag register (normal) 0001 = Zero or Equal (‘**.z**’ or ‘**.e**’) 0010 = Not Zero or Not Equal (‘**.nz**’ or ‘**.ne**’) 0011 = Greater-than (‘**.g**’) 0100 = Greater-than-or-equal (‘**.ge**’) 0101 = Less-than (‘**.l**’) 0110 = Less-than-or-equal (‘**.le**’) 0111 = Reserved 1000 = Overflow (‘**.o**’) 1001 = Unordered with Computed NaN (‘**.u**’) 1010 -1111 = Reserved	4
RegNum	**Register Number.** This field provides the register number for the operand. For GRF or MRF register operand, it provides the portion of register address aligning to 256-bit. For an ARF register operand, this field is encoded such that MSBs identify the architecture register type and LSBs provide its register number. This field together with the corresponding *SubRegNum* field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [12:5] of the byte address, while *SubRegNum* field provides bits [4:0]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U8, if *RegFile* = **GRF**. 0x00 to 0x7F = Register number in the range of [0, 127] 0x80 to 0xFF = Reserved Format = U8, if *RegFile* = **MRF**. 0x00 to 0x0F = Register number in the range of [0, 15] 0x10 to 0xFF = Reserved	8
SubRegNum	**Sub-Register Number.** This field provides the sub-register number for the operand, it provides the **dword** address within a 256-bit register This field together with the corresponding *RegNum* field provides the dword aligned address for the origin of the register region. Specifically, this field provides bits [4:2] of the dword address, while *RegNum* field provides bits [12:5]. This field applies to the destination operand and the source operands.	3

Field	Description	Width
SrcMod	**Source Modifier.** This field specifies the numerical modification to a source operand. The value of each data element of a source operand can optionally have its absolute value taken and/or its sign inverted prior to delivery to the execution pipe. The absolute value is prior to negate such that a guaranteed negative value can be produced. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 00 = No modification (normal) 01 = **"(abs)"**. Absolute 10 = **"–"**. Negate 11 = **"–(abs)"**. Negate of the absolute (forced negative value)	2
ChanSel	**Channel Select.** This field controls the channel swizzle for a source operand. The normally sequential channel assignment can be altered by explicitly identifying neighboring data elements for each channel. Out of the 8-bit field, 2 bits are assigned for each channel within the group of 4. ChanSel[1:0], [3.2], [5.4] and [7,6] are for channel 0 ("x"), 1 ("y"), 2 ("z"), and 3 ("w") in the group, respectively. For example with an execution size of 8, *r0.0**<4>.zywz:f*** would assign the channels as follows: $Chan_0$ = $Data_2$, $Chan_1$ = $Data_1$, $Chan_2$ = $Data_3$, $Chan_3$ = $Data_2$; $Chan_4$ = $Data_6$, $Chan_5$ = $Data_5$, $Chan_6$ = $Data_7$, $Chan_7$ = $Data_6$. This field only applies to source operand. This field is only present in **Align16** mode. It is not present for an immediate source operand. The 2-bit Channel Selection field for each channel within the group of 4 is defined as the following. 00 = **"x"**. Channel 0 is selected for the corresponding execution channel 01 = **"y"**. Channel 1 is selected for the corresponding execution channel 10 = **"z"**. Channel 2 is selected for the corresponding execution channel 11 = **"w"**. Channel 3 is selected for the corresponding execution channel	8
RepCtrl	**Replicate Control.** This field controls the replication of the starting channel to all channels in the execution size. This field applies to all three source operands. 0 = No replication 1 = Replicate across all channels	1

7.2.2 Instruction Operation Doubleword (DW0)

Most fields in Instruction Operation Doubleword (DW0) apply to all instructions. Bit field [27:24] is one exception. It is *CondModifier* for most instructions but is *CurrDest.RegNum* field for the *send* instruction.

The descriptions in the table below are shared between the 1-src/2-src instructions and 3-src instructions.

Table 7-3. Definitions of Fields in Operation Doubleword (DW0)

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>31</td><td>**Saturate.** This field controls the destination saturation.
When it is set, output data to the destination register are saturated. The saturation operation depends on the destination data type. Saturation is the operation that converts any data that is outside the *saturation target range* for the data type to the closest representable value with the target range. If destination type is float, saturation target range is [0, 1]. For example, any positive number greater than 1 (including +INF) is saturated to 1 and any negative number (including –INF) is saturated to 0. A NaN is saturated to 0, For integer data types, the maximum range for the given numerical data type is the saturation target range.
When it is not set, output data to the destination register are not saturated. For example, a wrapped result (modular) is output to the destination for an overflowed integer data.
More details can be found in the Data Types chapter.
0 = No destination modification (normal)
1 = **“sat”**. Saturate the output
<table>
<tr><th>Destination Type</th><th>Saturation Target Range (inclusive)</th></tr>
<tr><td>Float (F)</td><td>[0.0, 1.0]</td></tr>
<tr><td>Byte (UB)</td><td>[0, 255]</td></tr>
<tr><td>Signed Byte (B)</td><td>[-128, 127]</td></tr>
<tr><td>Word (UW)</td><td>[0, 65535]</td></tr>
<tr><td>Signed Word (W)</td><td>[-32768, 32767]</td></tr>
<tr><td>Double Word (UD)</td><td>[0, 2^{32}-1]</td></tr>
<tr><td>Signed Double (D)</td><td>[-2^{31}, 2^{31}-1]</td></tr>
</table>
</td></tr>
<tr><td>29</td><td>Reserved: MBZ</td></tr>
<tr><td>28</td><td>**AccWrCtrl.** This field allows per instruction accumulator write control.
0 = don’t write result into accumulator
1 = **“AccWrCtrl”.** write result into accumulator, and destination</td></tr>
</table>

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>27:24</td><td>**CondModifier** or **CurrDst.RegNum[3:0]**
Definition of this bit field depends on whether the instruction is a *send/math* or not.
<table>
<tr><th>Opcode != '*send*'</th><th>Opcode = '*send*'</th></tr>
<tr><td>**CondModifier:**
This field sets the flag register based on the internal conditional signals output from the execution pipe.</td><td>**CurrDst.RegNum[3:0]**
This field sets the MRF register number for the current destination operand in the *send* instruction. No flag registers are updated for the *send* instruction. The 4-bit field provides full access of the 16 MRF registers.
(See Instruction Reference chapter for *CurrDst*.)</td></tr>
</table></td></tr>
<tr><td>23:21</td><td>**ExecSize – Execution Size.** This field determines the number of channels operating in parallel for this instruction. The size cannot exceed the maximum number of channels allowed for the given data type.
000 = 1 Channels (scalar operation)
001 = 2 Channels
010 = 4 Channels
011 = 8 Channels
100 = 16 Channels
101= 32 Channels
110-111 = Reserved</td></tr>
<tr><td>20</td><td>**PredInv – Predicate Inverse.** This field, together with *PredCtrl*, enables and controls the generation of the predication mask for the instruction. When it is set, the predication uses the inverse of the predication bits generated according to setting of Predicate Control. In other words, effect of PredInv happens after PredCtrl.
This field is ignored by hardware if Predicate Control is set to 0000 – there is no predication.
0 = **"+"**. Positive polarity of predication.
1 = **"–"**. Negative polarity of predication.</td></tr>
</table>

Bits	Description
19:16	**PredCtrl – Predicate Control.** This field, together with *PredInv*, enables and controls the generation of the predication mask for the instruction. It allows per-channel conditional execution of the instruction based on the content of the selected flag register. Encoding depends on the access mode. In **Align16** access mode, there are eight encodings (including no predication). All encodings are based on group-of-4 predicate bits, including channel sequential, replication swizzles and horizontal any|all operations. The same configuration is repeated for each group-of-4 execution channels. In **Align1** access mode, there are twelve encodings (including no predication). The encodings applies to all execution channels with explicit channel grouping from single channel up to group of 16 channels. Predicate Control in **Align16** access mode 0000 = No predication (normal) 0001 = Predication with sequential flag channel mapping 0010 = Predication with replication swizzle **'.x'** 0011 = Predication with replication swizzle **'.y'** 0100 = Predication with replication swizzle **'.z'** 0101 = Predication with replication swizzle **'.w'** 0110 = Predication with **'.any4h'** 0111 = Predication with **'.all4h'** 1000 -1111 = Reserved Predicate Control in **Align1** access mode 0000 = No predication (normal) 0001 = Predication with sequential flag channel mapping 0010 = Predication with **.anyv** (any from f0.0-f0.1 on the same channel) 0011 = Predication with **.allv** (all of f0.0-f0.1 on the same channel) 0100 = Predication with **.any2h** (any in group of 2 channels) 0101 = Predication with **.all2h** (all in group of 2 channels) 0110 = Predication with **.any4h** (any in group of 4 channels) 0111 = Predication with **.all4h** (all in group of 4 channels) 1000 = Predication with **.any8h** (any in group of 8 channels) 1001 = Predication with **.all8h** (all in group of 8 channels) 1010 = Predication with **.any16h** (any in group of 16 channels) 1011 = Predication with **.all16h** (all in group of 16 channels)

Bits	Description
15:14	**ThreadCtrl – Thread Control.** This field provides explicit control for thread switching. If this field is set to 00, it is up to the GEN execution units to manage thread switching. This is the normal operations mode. In this mode, for example, if the current instruction cannot proceed due to operand dependencies, EU switches to next available thread to fill the compute pipe. In another example, if the current instruction is ready to go, however, there is another thread with higher priority also has instruction ready, EU switches to that thread. If this field is set to **Switch**, a forced thread switch occurs after the current instruction is executed and before the next instruction. In addition, a long delay (longer than the execution pipe latency) for the current thread is introduced for the thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. If this field is set to **Atomic**, the next instruction will get highest priority in the thread arbitration for the exeuction pipelines. **Switch** is designed primarily as a safety feature in case there are race conditions for certain instructions. 00 = Normal Thread Control 10 = **“Switch”** 01 = **“Atomic”** 11 = Reserved
11:10	**DepCtrl – Destination Dependency Control.** This field selectively disables destination dependency check and clear for this instruction. When it is set to 00, normal destination dependency control is performed for the instruction – hardware checks for destination hazards to ensure data integrity. Specifically, destination register dependency check is conducted before the instruction is made ready for execution. After the instruction is executed, the destination register scoreboard will be cleared when the destination operands retire. When bit 10 is set (**NoDDClr**), the destination register scoreboard will NOT be cleared when the destination operands retire. When bit 11 is set (**NoDDChk**), hardware does not check for destination register dependency before the instruction is made ready for execution. **NoDDClr** and **NoDDChk** are not mutual exclusive. When this field is not all-zero, hardware does not protect against destination hazards for the instruction. This is typically used to assemble data in a fine grained fashion (e.g. matrix-vector compute with dot-product instructions), where the data integrity is guaranteed by software based on the intended usage of instruction sequences. 00 = Destination dependency checked and cleared (normal) 01 = **“NoDDClr”**. Destination dependency checked but not cleared 10 = **“NoDDChk”**. Destination dependency not checked but cleared 11 = **“NoDDClr, NoDDChk”**. Destination dependency not checked and not cleared
9	**WECtrl – Write Enable Control.** This field determines if the the per channel write enables are used to generate the final write enable. This field should be normally “0”. 0 = use normal write enables (normal) 1 = write all channels, except channels killed with predication control

Bits	Description
8	**AccessMode – Access Mode.** This field determines the operand access for the instruction. It applies to all source and destination operands. When it is cleared (**Align1**), the instruction uses byte-aligned addressing for source and destination operands. Source swizzle control and destination mask control are not supported. When it is set (**Align16**), the instruction uses 16-byte-aligned addressing for all source and destination operands. Source swizzle control and destination mask control are supported in this mode. 0 = **"Align1"** 1 = **"Align16"**
7	Reserved: MBZ (for future opcode extension)
6:0	**Opcode – Instruction Operation Code.** This field contains the instruction operation code. Each opcode is given a unique mnemonic. For example, opcode 0x01 is for a move operation. Mnemonic for this opcode is *mov*. See Section 7.3 for details of opcode encoding.

7.2.3 Instruction Destination Doubleword (DW1)

7.2.3.1 1-src and 2-src Instructions

Destination Doubleword (DW1) contains the register file and numeric type of all operands, as well as the register region parameters of the destination operand. Table 7-4 shows the field definition of the Instruction Destination Doubleword. Furthermore, the Destination Register Region is described in Table 7-5 through Table 7-8.

Table 7-4. Instruction Destination Doubleword

Bits	Description
31:16	**Destination Register Region.** This word contains the parameters describing the register region of the destination operand. Subfield definition depends on the AccessMode. Detailed descriptions can be found in Table 7-5 through Table 7-8. **Programming Notes:** Allthough *Dst.HorzStride* is a don't care for Align16, HW needs this to be programmed as "01".
15	Reserved: MBZ

Bits	Description
14:12	**Src1.SrcType – Source-1 Data Type.** This field specifies the numerical data type of the source operand <src1>. The bits of a source operand are interpreted as the identified numerical data type, rather than coerced into a type implied by the operator. Depending on *RegFile* field of the source operand, there are two different encoding for this field. If a source is a register operand, this field follows the Source Register Type Encoding. If a source is an immediate operand, this field follows the Source Immediate Type Encoding. Source Register Type Encoding is identical to that for Destination Type. Source Immediate Type Encoding differs in two areas. First, it does not support byte and unsigned numerical data types. Secondly, it has two 32-bit vector types – halfbyte integer vector (V) type and exponent-only float vector (VF) type. *Implementation Note 1: Both source operands, <src0> and <src1>, support immediate types, but only one immediate is allowed for a given instruction and it must be the last operand.* *Implementation Note 2: Halfbyte integer vector (v) type can only be used in instructions in packed-word execution mode. Therefore, in a two-source instruction where <src1> is of type :v, <src0> must be of type :b, :ub, :w, or :uw.* Source Register Type Encoding 000 = **“UD”**. **U**nsigned **D**oubleword integer 001 = **“D”**. Signed **D**oubleword integer 010 = **“UW”**. **U**nsigned **W**ord integer 011 = **“W”**. Signed **W**ord integer 100 = **“UB”**. **U**nsigned **B**yte integer 101 = **“B”**. Signed **B**yte integer 110 = Reserved **[DevGT]** 110 = **“DF”**. **D**ouble precision **F**loat (64-bit) **[DevIVB+]** Source Immediate Type Encoding: 000 = **“UD”** 001 = **“D”** 010 = **“UW”** 011 = **“W”** 100 = Reserved 101 = **“VF”**. 32-bit restricted **V**ector **F**loat 110 = **“V”**. 32-bit halfbyte integer **V**ector 111 = **“F”**
11:10	**Src1.RegFile – Source-1 Register File.** This field identifies the register file of source operand <src1>. 00 = **“ARF”**. Architecture Register File (**a**#, **acc**#, **f**#, **n**#, **null**, **ip**, etc.) 01 = **“GRF”**. General Register File (**r**#) 10 = **“MRF”**. Message Register File (**m**#) 11 = **“IMM”**. Immediate
9:7	**Src0.SrcType – Source-0 Data Type.** This field is the *SrcType* for <src0> operand. It has the same definitions as *Src1.SrcType*.

Bits	Description
6:5	**Src0.RegFile – Source-0 Register File.** This field is the *RegFile* for <src0> operand. It has the same definitions as *Src1.RegFile*.
4:2	**Dst.DstType – Destination Data Type.** This field specifies the numerical data type of the destination operand <dst>. The bits of the destination operand are interpreted as the identified numerical data type, rather than coerced into a type implied by the operator. For a *send* instruction, this field applies to the **CurrDst** – the current destination operand. Encoding: 000 = **"UD"**. **U**nsigned **D**oubleword integer 001 = **"D"**. Signed **D**oubleword integer 010 = **"UW"**. **U**nsigned **W**ord integer 011 = **"W"**. Signed **W**ord integer 100 = **"UB"**. **U**nsigned **B**yte integer 101 = **"B"**. Signed **B**yte integer 110 = Reserved 111 = **"F"**. Single precision **F**loat (32-bit)
1:0	**Dst.RegFile – Destination Register File.** This field identifies the register file of the destination operand <dst>. Note that it is obvious that immediate cannot be a destination operand. For a *send* instruction, this field applies to the **PostDst** – the post destination operand. Encoding: 00 = **"ARF"**. Architecture Register File (**a**#, **acc**#, **f**#, **n**#, **null**, **ip**, etc.) 01 = **"GRF"**. General Register File (**r**#) 10 = **"MRF"**. Message Register File (**m**#) 11 = **reserved**

The following tables describe the Destination Register Region based on the access mode and addressing mode.

Table 7-5. Destination Register Region in Direct + Align16 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. (See section 7.2.1 for definition of *AddrMode.*) For a *send* instruction, this field applies to **PostDst** – the post destination operand. Addressing mode for *CurrDst* (current destination operand) is fixed as Direct. (See Instruction Reference chapter for *CurrDst* and *PostDst*.)
14:13	Reserved: MBZ
12:5	**Dst.RegNum – Destination Register Number.** This field is the *RegNum* field for the destination operand. (See section 7.2.1 for definitions of *RegNum.*) For a *send* instruction, this field applies to **PostDst**.
4	**Dst.SubRegNum[4].** This is the 16-byte aligned sub-register address. (See section 7.2.1 for definitions of *SubRegNum*) For a *send* instruction, this field applies to **CurrDst**.
3:0	**Dst.ChanEn – Destination Channel Enable**. The channel enable field for the destination operand. (See section 7.2.1 for definitions of *ChanEn*) For a *send* instruction, this field applies to the **CurrDst**.

Table 7-6. Destination Register Region in Direct+Align1 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, it applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	**Dst.HorzStride – Destination Horizontal Stride.** This field is the *HorzStride* for the destination operand. For a *send* instruction, this field applies to **CurrDst**. **PostDst** only uses the register number.
12:5	**Dst.RegNum – Destination Register Number.** This field is the *RegNum* field for the destination operand. For a *send* instruction, this field applies to **PostDst**.
4:0	**Dst.SubRegNum – Destination Sub-Register Number.** This field is the **SubRegNum** for the destination operand. (See section 7.2.1 for definition of *SubRegNum*) For a *send* instruction, this field applies to **CurrDst**.

Table 7-7. Destination Register Region in Indirect+Align16 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, this field applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	Reserved: MBZ
12:10	**Dst.AddrSubRegNum – Destination Address Sub-Register Number.** This field is the *AddrSubRegNum* for the destination operand. (See section 7.2.1 for definition of *AddrSubRegNum.*) For a *send* instruction, this field applies to **PostDst**.
9:4	**Dst.AddrImm[9:4]** This is the half-register aligned *AddrImm* field for the destination operand. (See section 7.2.1 for definition of *AddrImm*) For a *send* instruction, this field applies to **PostDst**.
3:0	**Dst.ChanEn – Destination Channel Enable.** The channel enable field for the destination operand. For a *send* instruction, this field applies to the **CurrDst**.

Table 7-8. Destination Register Region in Indirect+Align1 mode

Bits	Description
15	**Dst.AddrMode – Destination Address Mode.** This field is the *AddrMode* for the destination operand. For a *send* instruction, this field applies to **PostDst**. Addressing mode for ***CurrDst*** is fixed as Direct.
14:13	**Dst.HorzStride – Destination Horizontal Stride** This field is the *HorzStride* for the destination operand. For a *send* instruction, this field applies to **CurrDst**. **PostDst** only uses the register number.
12:10	**Dst.AddrSubRegNum – Destination Address Sub-Register Number.** This field is the *AddrSubRegNum* for the destination operand. For a *send* instruction, this field applies to **PostDst**.
9:0	**Dst.AddrImm – Destination Address Immediate.** This field is the byte-aligned *AddrImm* for the destination operand. For a *send* instruction, this field applies to **PostDst**.

7.2.3.2 3-src Instructions

This section describes the field in DW1 of the 3-src instruction format.

Table 7-9. Instruction DW1

Bits	Description
31:24	**Destination Register Number.** This field contains the destination register number.
23:21	**Destination Subregister Number.** This field contains the destination subregister number.
20:17	**Destination Channel Enable.** Four channel enables are defined for controlling which channels will be written into the destination region. These channel mask bits are applied in a modulo-four manner to all *ExecSize* channels. There is 1-bit Channel Enable for each channel within the group of 4. If the bit is cleared, the write for the corresponding channel is disabled. If the bit is set, the write is enabled. Mnemonic for the bit being set for the group of 4 is “x”, “y”, “z”, and “w”, respectively, where “x” corresponds to Channel 0 in the group and “w” corresponds to channel 3 in the group. 0 = Write Disabled 1 = Write Enabled (normal)
12:10	Reserved: MBZ
9:8	**Source2 Modifier.** This field contains the modifier for source2. Refer to Table. 5-5 for the encoding.
7:6	**Source1 Modifier.** This field contains the modifier for source1. Refer to Table. 5-5 for the encoding.
5:4	**Source0 Modifier.** This field contains the modifier for source0. Refer to Table. 5-5 for the encoding.
3	Reserved: MBZ
1	**Flag Subregister Number.** This field contains the flag subregister number for instructions with non-zero Conditional Modifier.

7.2.4 Instruction Source-0 Doubleword (DW2)

7.2.4.1 1-src and 2-src Instructions

Instruction Source-0 Doubleword (DW2) contains the first source operand and also flag register number.

- Table 7-10 shows the field definition for Direct Addressing with Align16.
- Table 7-11 shows the field definition for Direct Addressing with Align1.

Table 7-12 shows the field definition for Indirect Addressing with Align16.

Table 7-13 shows the field definition for Indirect Addressing with Align1.

Table 7-10. Instruction Source-0 Doubleword in Direct+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand. There are two sub-registers in the flag register. Each sub-register contains 16 flag bits. The selected flag sub-register is the source for predication if predication is enabled for the instruction. It is the destination to store conditional flag bits if conditional modifier is enabled for the instruction. The same flag sub-register can be both the predication source and conditional destination, if both predication and conditional modifier are enabled.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This field is the *VertStride* for <src0> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	**Src0.ChanSel[7:4]** This is bits [7:4] of the *ChanSel* field for <src0> operand. (See section 7.2.1 for definition of *ChanSel*).It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This field is the *AddrMode* for <src0> operand. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This field is the *SrcMod* for source operand <src0>. (See section 7.2.1 for definition of *SrcMod*)It is ignored if <src0> is an immediate operand.
12:5	**Src0.RegNum – Source-0 Register Number** This is the *RegNum* field for source operand <src0>. (See section 7.2.1 for definition of *RegNum.*) It is ignored if <src0> is an immediate operand.
4	**Src0.SubRegNum[4]** This is the 16-byte aligned sub-register address for source operand <src0>. (See section 7.2.1 for definition of *SubRegNum*) It is ignored if <src0> is an immediate operand.

Bits	Description
3:0	**Src0.ChanEn – Source-0 Channel Enable** This is the *ChanEn* field for source operand <src0>. (See section 7.2.1 for definitions of *ChanEn*) It is ignored if <src0> is an immediate operand.

Table 7-11. Instruction Source-0 Doubleword in Direct+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride** This is the *VertStride* field for <src0> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20:18	**Src0.Width.** This is the *Width* field for source operand <src0>. (See section 7.2.1 for definition of *Width*) It is ignored if <src0> is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand <src0>. (See section 7.2.1 for definition of *HorzStride*) It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:5	**Src0.RegNum – Source-0 Register Number.** This is the *RegNum* field for source operand <src0>. (See section 7.2.1 for definition of *RegNum.*) It is ignored if <src0> is an immediate operand.
4:0	**Src0.SubRegNum – Source-0 Sub-Register Number.** This is the *SubRegNum* field for source operand <src0>. (See section 7.2.1 for definition of *SubRegNum*) It is ignored if <src0> is an immediate operand.

Table 7-12. Instruction Source-0 Doubleword in Indirect+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This is the *VertStride* field for <src0> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	**Src0.ChanSel[7:4] – Source-0 Channel Select.** This is bits [7:4] of the *ChanSel* field for <src0> operand. (See section 7.2.1 for definition of *ChanSel*). It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:10	**Src0.AddrSubRegNum – Source-0 Address Sub-Register Number.** This is the *AddrSubRegNum* field for source operand <src0>. (See section 7.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src0> is an immediate operand.
9:4	**Src0.AddrImm[9:4] – Source-0 Address Immediate.** This contains the half-register aligned *AddrImm* field ((bits [9:4]) for <src0>. (See section 7.2.1 for definition of *AddrImm*) It is ignored if <src0> is an immediate operand.
3:0	**Src0.ChanEn – Source-0 Channel Enable** . This is the *ChanEn* field for source operand <src0>. (See section 7.2.1 for definitions of *ChanEn*) It is ignored if <src0> is an immediate operand.

Table 7-13. Instruction Source-0 Doubleword in Indirect+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	**FlagSubRegNum – Flag Sub-Register Number.** This field specifies the sub-register number for a flag register operand.
24:21	**Src0.VertStride – Source-0 Vertical Stride.** This is the *VertStride* field for <src0> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src0> is an immediate operand.
20:18	**Src0.Width.** This is the *Width* field for source operand <src0>. (See section 7.2.1 for definition of *Width*) It is ignored if <src0> is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand <src0>. (See section 7.2.1 for definition of *HorzStride*) It is ignored if <src0> is an immediate operand.
15	**Src0.AddrMode – Source-0 Address Mode.** This is the *AddrMode* for source operand <src0>. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src0> is an immediate operand.
14:13	**Src0.SrcMod – Source-0 Source Modifier.** This is the *SrcMod* field for source operand <src0>. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src0> is an immediate operand.
12:10	**Src0.AddrSubRegNum – Source-0 Address Sub-Register Number.** This is the *AddrSubRegNum* field for source operand <src0>. (See section 7.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src0> is an immediate operand.
9:0	**Src0.AddrImm – Source-0 Address Immediate.** This is the byte aligned *AddrImm* field for <src0>. (See section 7.2.1 for definition of *AddrImm*) It is ignored if <src0> is an immediate operand.

7.2.4.2 3-src Instructions

This section describes the field in DW2 and DW3 of the 3-src instruction format.

Table 7-14. Instruction DW2 and DW3

DW	Bits	Description
DW3	31:29	Reserved: MBZ
	28:21	**Source2 Register Number.** This field contains the register number for source2.
	20:18	**Source2 Subregister Number.** This field contains the subregister number for source2.
	17:10	**Source2 Swizzle.** This field contains the swizzle control for source2. Refer to Table.5-5 for encoding.
	9:9	**Source2 Replication Control.** This field controls replication for source2. Refer to Table.5-5 for encoding.
	8:8	Reserved: MBZ
	7:0	**Source1 Register Number.** This field contains the register number for source1.
DW2	31:29	**Source1 Subregister Number.** This field contains the subregister number for source1.
	28:21	**Source1 Swizzle.** This field contains the swizzle control for source1. Refer to Table.5-5 for encoding.
	20:20	**Source1 Replication Control.** This field controls replication for source1. Refer to Table.5-5 for encoding.
	19:19	Reserved: MBZ
	18:11	**Source0 Register Number.** This field contains the register number for source0.
	10:8	**Source0 Subregister Number.** This field contains the subregister number for source0.
	7:1	**Source0 Swizzle.** This field contains the swizzle control for source0. Refer to Table.5-5 for encoding.
	0:0	**Source0 Replication Control.** This field controls replication for source0. Refer to Table.5-5 for encoding.

7.2.5 Instruction Source-1 Doubleword (DW3)

Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>). Table 7-15 and Table 7-16 define the fields in this doubleword with the following exceptions:

- If <src0> is an immediate operand, this doubleword contains **imm32** for <src0>.
- If <src1> is an immediate operand, this doubleword contains **imm32** for <src1>.
- If the instruction is a send, bit 31 of this doubleword contains **EOT** field.
    - If <src1> is immediate, the remaining 31 bits in this doubleword is **MsgDescpt31**.
    - If <src1> is a register, <src1> must be a0.0. The rest of this doubleword will be configured accordingly.
- If indirect address is supported for <src1>, Table 7-17 and Table 7-18 define the fields in DW3 for indirectly addressed <src1> in Align16 and Align1 modes.

Table 7-15. Instruction Source-1 Doubleword in Direct + Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride.** This field is the *VertStride* for <src1> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20	Reserved: MBZ
19:16	**Src1.ChanSel[7:4]** This contains bits [7:6] of the *ChanSel* field for <src1> operand. (See section 7.2.1 for definition of *ChanSel*) It is ignored if <src1> is an immediate operand.
15	Reserved: MBZ
14:13	**Src1.SrcMod – Source-1 Source Modifier.** This field is the *SrcMod* for <src1> operand. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.
12:5	**Src1.RegNum.** This field is the *RegNum* field for <src1> operand. (See section 7.2.1 for definition of *RegNum.*) It is ignored if <src1> is an immediate operand.
4	**Src1.SubRegNum[4].** This field is bit [4] of the *SubRegNum* field for <src1>. (See section 7.2.1 for definition of *SubRegNum*) It is ignored if <src1> is an immediate operand.
3:0	**Src1.ChanEn – Source-1 Channel Enable.** It is the channel enable field for <src1>. (See section 7.2.1 for definitions of *ChanEn*)It is ignored if <src1> is an immediate operand.

Table 7-16. Instruction Source-1 Doubleword in Direct + Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride.** This field is the *VertStride* for <src1> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20:18	**Src1.Width.** This is the *Width* field for source operand <src1>. (See section 7.2.1 for definition of *Width*) It is ignored if <src1> is an immediate operand.
17:16	**Src1.HorzStride.** This is the *HorzStride* field for source operand <src1>. (See section 7.2.1 for definition of *HorzStride*) It is ignored if <src1> is an immediate operand.
15	Reserved: MBZ
14:13	**Src1.SrcMod – Source-1 Source Modifier.** This field is the *SrcMod* for <src1> operand. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.
12:5	**Src1.RegNum – Source-1 Register Number.** This is the *RegNum* field for source operand <src1>. (See section 7.2.1 for definition of *RegNum.*) It is ignored if <src1> is an immediate operand.
4:0	**Src1.SubRegNum – Source-1 Sub-Register Number.** This is the *SubRegNum* field for source operand <src1>. (See section 7.2.1 for definition of *SubRegNum*) It is ignored if <src1> is an immediate operand.

Table 7-17. Instruction Source-1 Doubleword in Indirect+Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride** This is the *VertStride* field for <src1> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20	Reserved: MBZ
19:16	**Src1.ChanSel[7:4] – Source-1 Channel Select** This is bits [7:4] of the *ChanSel* field for <src1> operand. (See section 7.2.1 for definition of *ChanSel*). It is ignored if <src1> is an immediate operand.
15	**Src1.AddrMode – Source-1 Address Mode** This is the *AddrMode* for source operand <src1>. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src1> is an immediate operand.
14:13	**Src1.SrcMod – Source-1 Source Modifier** This is the *SrcMod* field for source operand <src1>. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.

Bits	Description
12:10	**Src1.AddrSubRegNum – Source-1 Address Sub-Register Number** This is the *AddrSubRegNum* field for source operand <src1>. (See section 7.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src1> is an immediate operand.
9:4	**Src1.AddrImm[9:4] – Source-1 Address Immediate** This contains the half-register aligned *AddrImm* field ((bits [9:4]) for <src1>. (See section 7.2.1 for definition of *AddrImm*) It is ignored if <src1> is an immediate operand.
3:0	**Src1.ChanEn – Source-1 Channel Enable** This is the *ChanEn* field for source operand <src1>. (See section 7.2.1 for definitions of *ChanEn*) It is ignored if <src1> is an immediate operand.

Table 7-18. Instruction Source-1 Doubleword in Indirect+Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	**Src1.VertStride – Source-1 Vertical Stride** This is the *VertStride* field for <src1> operand. (See section 7.2.1 for definition of *VertStride*) It is ignored if <src1> is an immediate operand.
20:18	**Src1.Width** This is the *Width* field for source operand <src1>. (See section 7.2.1 for definition of *Width*) It is ignored if <src1> is an immediate operand.
17:16	**Src1.HorzStride** This is the *HorzStride* field for source operand <src1>. (See section 7.2.1 for definition of *HorzStride*) It is ignored if <src1> is an immediate operand.
15	**Src1.AddrMode – Source-1 Address Mode** This is the *AddrMode* for source operand <src1>. (See section 7.2.1 for definition of *AddrMode*) It is ignored if <src1> is an immediate operand.
14:13	**Src1.SrcMod – Source-1 Source Modifier** This is the *SrcMod* field for source operand <src1>. (See section 7.2.1 for definition of *SrcMod*) It is ignored if <src1> is an immediate operand.
12:10	**Src1.AddrSubRegNum – Source-1 Address Sub-Register Number** This is the *AddrSubRegNum* field for source operand <src1>. (See section 7.2.1 for definition of *AddrSubRegNum.*) It is ignored if <src1> is an immediate operand.
9:0	**Src1.AddrImm – Source-1 Address Immediate** This is the byte aligned *AddrImm* field for <src1>. (See section 7.2.1 for definition of *AddrImm*) It is ignored if <src1> is an immediate operand.

7.3 Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

There are total of 48 opcodes defined. These opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxb are reserved.

7.3.1 Move and Logic Instructions

This instruction group has an opcode format of 00xxxxxb.

- The opcodes for move instructions (***mov***, ***sel*** and ***movi***) share the common 5 MSBs in the form of 00000xxb.
- The opcodes for logic instructions (***not***, ***and***, ***or***, and ***xor***) share the common 5 MSBs in the form of 00001xxb.
- The opcodes for shift instructions (***shr***, ***shl***, and ***asr***) share the common 4 MSBs in the form of 0001xxxb. Bit 2 indicates arithmetic or logic shift (0 = logic, 1 = arithmic). Bit 1 is always 0 (which is reserved for future extension to support rotation shift as 0 = shift, 1 = rotate). Bit 0 indicates the shift direction (0 = right, 1 = left).
- The opcodes for compare instructions (***cmp*** and ***cmpn***) share the common 6 MSBs in the form of 001000xb. Bit 0 indicates whether it is a normal compare, *cmp*, or a special compare-NaN, *cmpn*.
- This group of instructions does not implicitly update the accumulators.
- Instruction compression applies to this group.

Table 7-19. Move and Logic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
1	0x01	***mov***	Component-wise move	1	1
2	0x02	***sel***	Component-wise selective move based on predication	2	1
3	0x03	***movi***	Fast component-wise indexed move	1	1
4	0x04	***not***	Component-wise one’s compliment (bitwise not)	1	1
5	0x05	***and***	Component-wise logical AND (bitwise and)	2	1
6	0x06	***or***	Component-wise logical OR (bitwise or)	2	1
7	0x07	***xor***	Component-wise logical XOR (bitwise xor)	2	1
8	0x08	***shr***	Component-wise logical shift right	2	1
9	0x09	***shl***	Component-wise logical shift left	2	1
10-11	0x0A-0x0B	*Reserved*			
12	0x0C	***asr***	Component-wise arithmetic shift right	2	1
13-15	0x0D-0x0F	*Reserved*			
16	0x10	***cmp***	Component-wise compare, store condition code in destination	2	1
17	0x11	***cmpn***	Component-wise compare-NaN, store condition code in destination	2	1
18	0x12	*Reserved*			
21-22	0x15-0x16	*Reserved*			
20-31	0x12-0x1F	*Reserved*			

7.3.2 Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 7-20. Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	***jmpi***	Jump indexed	1	0
34	***if***	*If*	0/2	0	***if***
36	0x24	***else***	Else	1	0
37	0x25	***endif***	End if	0	0
38	0x26	***case***	Case – Inside Switch block	0/2	0
39	0x27	***while***	While	1	0
40	0x28	***break***	Break	1	0
41	0x29	***cont***	Continue	1	0
42	0x2A	***halt***	Halt	1	0
43	0x2B	*Reserved*			
44	0x2C	***call***	Subroutine call	1	1
45	0x2D	***return***	Subroutine return	1	1
46	0x2E	***fork***	go into 16pixel execution mode from 32pixel execution mode	1	0
47	0x2F	*Reserved*			

7.3.3 Miscellaneous Instructions

This instruction group has an opcode format of 011xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 7-21. Miscellaneous Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
48	0x30	***wait***	Wait for (external) notification	1	0
49	0x31	***send***	Send	1	1
50	0x32	***sendc***	Conditional Send (based on TDR)	1	1
51-55	0x33- 0x37	*Reserved*			
56	0x38	***math***	Math functions for extended math pipeline	1/2	1/2
57-63	0x39- 0x3F	*Reserved*			

7.3.4 Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

- The opcode for round instructions (***rndu***, ***rndd***, ***rnde***, and ***rndz***) share the common 5 MSBs in the form of 10001xxb, with the lower 2 bits indicate the type of round.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
    - Some instructions such as ***frc***, ***lzd***, etc, perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 7-22. Parallel Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
64	0x40	***add***	Component-wise addition	2	1
65	0x41	***mul***	Component-wise multiply	2	1
66	0x42	***avg***	Component-wise average of the two source operands	2	1
67	0x43	***frc***	Component-wise floating point truncate-to-minus-infinity fraction	1	1
68	0x44	***rndu***	Component-wise floating point rounding up (ceiling)	1	1
69	0x45	***rndd***	Component-wise floating point rounding down (floor)	1	1
70	0x46	***rnde***	Component-wise floating point rounding toward nearest even	1	1
71	0x47	***rndz***	Component-wise floating point rounding toward zero	1	1
72	0x48	***mac***	Component-wise multiply accumulate	2	1
73	0x49	***mach***	multiply accumulate high	2	1
74	0x4A	***lzd***	leading zero detection	1	1

7.3.5 Vector Arithmetic Instructions

- This instruction group has an opcode format of 101xxxxb.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
    - Some instructions such as ***dp4-dp2***, etc, perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 7-23. Vector Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
80	0x50	***sad2***	2-wide sum of absolute difference	2	1
81	0x51	***sada2***	2-wide sad accumulate	2	1
82-83	0x52-0x53	*reserved*			
84	0x54	***dp4***	4-wide dot product for 4-vector	2	1
85	0x55	***dph***	4-wide homogenous dot product for 4-vector	2	1
86	0x56	***dp3***	3-wide dot product for 4-vector	2	1
87	0x57	***dp2***	2-wide dot product for 4-vector	2	1
88	0x58	*reserved*			
89	0x59	***line***	Component-wise line equation computation (a multiply-add)	2	1
90	0x5A	***pln***	Component-wise floating point plane equation computation (a multiply-multiply-add)	2(3)	1
91	0x5B	***mad***	Component-wise floating point mad computation (a multiple-add)	3	1
92	0x5C	***lrp***	Component-wise floating point lrp computation (blend)	3	1
93-95	0x5D-0x5F	*reserved*			

7.3.6 Special Instructions

There are two special instructions, namely, *nop* (opcode = 0x7E) and *illegal* (opcode = 0x**00**).

- *Nop* instruction may be used for instruction padding in memory between two normal instructions to force alignment or to introduce instruction execution delay. Currently, there is no need for between-instruction padding.
- *Illegal* instruction may be used for instruction padding in memory outside the normal instruction sequence such as before or after the kernel program as well as between subroutines.
- *Nop* and *illegal* instructions do not have source operands or destination operand. Therefore, they do not implicitly update the accumulator register. They cannot be compressed.

Table 7-24. Special Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
0	0x00	***illegal***	Illegal instruction	0	0
96-124	0x60-0x7C	*Reserved*			
125	0x7D	***nenop***	Non-executed No-op	0	0
126	0x7E	***nop***	No-op	0	0
127	0x7F	*Reserved*	(may be used as an extension code)		

7.4 Native Instruction BNF

The Backus-Naur Form (BNF) grammar identifies the assembly language syntax, which is native to the hardware. It does not include intelligent defaults, assembler pragmas, etc.

7.4.1 Instruction Groups

```
<Instruction>               ::=  <UnaryInstruction>
                              |  <BinaryAccInstruction>
                              |  <BinaryInstruction>
                              |  <TriInstruction>
                              |  <JumpInstruction>
                              |  <BranchLoopInstruction>
                              |  <ElseInstruction>
                              |  <BreakInstruction>
                              |  <MaskControlInstruction>
                              |  <SyncInstruction>
                              |  <SpecialInstruction>

<UnaryInstruction>          ::=  <Predicate> <UnaryInst> <ExecSize> <Dst> <SrcAccImm> <InstOptions>
<UnaryInst>                 ::=  <UnaryOp> <ConditionalModifier> <Saturate>
<UnaryOp>                   ::=  "mov" | "frc" | "rndu" | "rndd" | "rnde" | "rndz" | "not" | "lzd"

<BinaryInstruction>         ::=  <Predicate> <BinaryInst> <ExecSize> <Dst> <Src> <SrcImm> <InstOptions>
<BinaryInst>                ::=  <BinaryOp> <ConditionalModifier> <Saturate>
<BinaryOp>                  ::=  "mul" | "mac" | "mach" | "line"
                              |  "sad2" | "sada2" | "dp4" | "dph" | "dp3" | "dp2"

<BinaryAccInstruction>      ::=  <Predicate> <BinaryAccInst> <ExecSize> <Dst> <SrcAcc> <SrcImm>
                              <InstrOptions>
<BinaryAccInst>             ::=  <BinaryAccOp> <ConditionalModifier> <Saturate>
<BinaryAccOp>               ::=  "avg" | "add" | "sel"
                              |  "and" | "or" | "xor"
                              |  "shr" | "shl" | "asr"
                              |  "cmp" | "cmpn"

<TriInstruction>            ::=  <Predicate> <TriInst> <ExecSize> <PostDst> <CurrDst> <TriSrc> <MsgDesc>
                              <InstOptions>
<TriInst>                   ::=  <TriOp> <ConditionalModifier> <Saturate>
<TriOp>                     ::=  "send"

<JumpInstruction>           ::=  <JumpOp> <RelativeLocation2>
<JumpOp>                    ::=  "jmpi"

<BranchLoopInstruction>     ::=  <Predicate> <BranchLoopOp> < RelativeLocation>
<BranchLoopOp>              ::=  "if" | "iff" | "while"

<ElseInstruction>           ::=  <ElseOp> < RelativeLocation>
```

```
<ElseOp>                 ::=   "else"

<BreakInstruction>       ::=   <Predicate> <BreakOp> <LocationStackCtrl>
<BreakOp>                ::=   "break" | "cont" | "halt"

<SyncInstruction>        ::=   <Predicate> <SyncOp> <NotifyReg>
<SyncOp>                 ::=   "wait"

<SpecialInstruction>     ::=   "do" | "endif" |"nop" | "illegal"
```

7.4.2 Destination Register

```
<Dst>                    ::=   <DstOperand>
                           |   <DstOperandEx>

<DstOperand>             ::=   <DstReg> <DstRegion> <WriteMask> <DstType>
<DstOperandEx>           ::=   <AccReg> <DstRegion> <DstType>
                           |   <FlagReg> <DstRegion> <DstType>
                           |   <AddrReg> <DstRegion> <DstType>
                           |   <MaskReg> <DstRegion> <DstType>
                           |   <MaskStackReg>
                           |   <ControlReg>
                           |   <IPReg>
                           |   <NullReg>

<DstReg>                 ::=   <DirectGenReg> | <IndirectGenReg>
                           |   <DirectMsgReg> | <IndirectMsgReg>

<PostDst>                ::=   <PostDstReg> <DstRegion> <WriteMask> <DstType>
                           |   <NullReg>

<PostDstReg>             ::=   <DirectGenReg> | <IndirectGenReg>

<CurrDst>                ::=   <DirectAlignedMsgReg>
```

7.4.3 Source Register

Source with Accumulator Access and with Immediate

```
<SrcAccImm>               ::=   <SrcAcc>
                             |  <Imm32> <SrcImmType>

<SrcAcc>                  ::=   <DirectSrcAccOperand>
                             |  <IndirectSrcOperand>

<DirectSrcAccOperand>     ::=   <DirectSrcOperand>
                             |  <SrcArcOperandEx>
                             |  <AccReg> <SrcType>

<SrcArcOperandEx>         ::=   <FlagReg> <Region> <SrcType>
                             |  <AddrReg> <Region> <SrcType>
                             |  <MaskReg> <Region> <SrcType> [Pre-DevSNB]
                             |  <MaskStackReg> [Pre-DevSNB]
                             |  <ControlReg>
                             |  <StateReg>
                             |  <NotifyReg>
                             |  <IPReg>
                             |  <NullReg>

<IndirectSrcOperand>      ::=   <SrcModifier> <IndirectGenReg> <IndirectRegion> <Swizzle > <SrcType>
```

Source without Accumulator Access

```
<Src>                     ::=   <DirectSrcOperand>
                             |  <IndirectSrcOperand>

< DirectSrcOperand >      ::=   <SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType>
                             |  <SrcArcOperandEx>

<TriSrc>                  ::=   <SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType>
                             |  <NullReg>

<MsgDesc>                 ::=   <ImmDesc>
                             |  <Reg32>
<Reg32>                   ::=   <DirectGenReg> <Region> <SrcType>
```

Source without Accumulator Access or IP Access

```
<SrcImm>                  ::=   <DirectSrcOperand>
                             |  <Imm32> <SrcImmType>
```

7.4.4 Address Registers

<AddrParam>	::=	<AddrReg> <ImmAddrOffset>
<ImmAddrOffset>	::=	“”
	|	“**,**” <ImmAddrNum>

7.4.5 Register Files and Register Numbers

<DirectGenReg>	::=	<GenRegFile> <GenRegNum> <GenSubRegNum>
<IndirectGenReg>	::=	<GenRegFile> “**[**“ <AddrParam> “**]**”
<GenRegFile>	::=	“**r**”
<GenRegNum>	:: =	“**0**”…“**127**”
<GenSubRegNum>	:: =	“”
	|	“**.0**”…“**.7**”
	|	“**.0**”…“**.15**”
	|	“**.0**”…“**.31**”
<DirectMsgReg>	::=	<DirectAlignedMsgReg> <MsgSubRegNum>
<DirectAlignedMsgReg>	::=	<MsgRegFile> <MsgRegNum>
<IndirectMsgReg>	::=	<MsgRegFile> “**[**“ <AddrParam> “**]**”
<MsgRegFile>	::=	“**m**”
<MsgRegNum>	:: =	“**0**”…“**15**”
<MsgSubRegNum>	:: =	<GenSubRegNum>
<AddrReg>	::=	<AddrRegFile> <AddrSubRegNum>
<AddrRegFile>	::=	“**a0**”
<AddrSubRegNum>	:: =	“”
	|	“**.0**” … “**.7**”
<AccReg>	::=	“**acc**” <AccRegNum><AccSubRegNum>
<AccRegNum>	:: =	“**0**” | “**1**”
<AccSubRegNum>	:: =	<GenSubRegNum>
<FlagReg>	::=	“**f0**” <FlagSubRegNum>
<FlagSubRegNum>	:: =	“”
	|	“**.0**”…“**.1**”

~~These are Pre-devGT and should not be in this spec.~~

~~[Pre-DevGT] <MaskReg>~~	~~::=~~	~~“**Mask0**” <MaskSubRegNum>~~
	~~|~~	~~“**AMask**” | “**IMask**” | “**LMask**” | “**CMask**”~~
~~[Pre-DevGT] <MaskSubRegNum>~~	~~:: =~~	~~“”~~
	~~|~~	~~“**.0**” … “**.3**”~~
~~[Pre-DevGT] <MaskStackReg>~~	~~::=~~	~~“**ms0**” <MaskStackSubRegNum>~~
	~~|~~	~~“**ims**” | “**lms**”~~
~~[Pre-DevGT] <MaskStackSubRegNum>~~	~~:: =~~	~~“”~~

		~~| “**.0**” | “**.16**”~~
~~[Pre-DevGT] <MaskStackDepthReg>~~	~~::=~~	~~“**MSD0**” <MaskStackDepthSubRegNum>~~
		~~| “**IMSD**” | “**LMSD**”~~
~~[Pre-DevGT] <MaskStackDepthSubRegNum>~~	~~:: =~~	~~“”~~
		~~| “**.0**” … “**.1**”~~

<NotifyReg>	::=	“**n**” <NotifyRegNum>
[Pre-DevGT] <NotifyRegNum>	:: =	“**0**”...“**1**”

<NotifyReg>	::=	“**n**” <NotifyRegNum>
<NotifyRegNum>	:: =	“**0**”...“**2**”
<StateReg>	::=	“**sr0**” <StateSubRegNum>
<StateSubRegNum>	:: =	“**.0**”... “**.1**”
<ControlReg>	::=	“**cr0**” <ControlSubRegNum>
<ControlSubRegNum>	:: =	“**.0**” ...“**.2**”
<IPReg>	::=	“**ip**”
<NullReg>	::=	“**null**”

7.4.6 Relative Location and Stack Control

<RelativeLocation>	::=	<imm16>
<RelativeLocation2>	::=	<imm32> | <reg32>
<LocationStackCtrl>	::=	<imm32>

7.4.7 Regions

<DstRegion>	::=	“**<**” <HorzStride> “**>**”
<IndirectRegion>	::=	<Region> | <RegionWH> | <RegionV>
<Region>	::=	“**<**” <VertStride> “**;**” <Width> “**,**” <HorzStride> “**>**”
<RegionWH>	::=	“**<**” <Width> “**,**” <HorzStride> “**>**”
<RegionV>	::=	“**<**”<VertStride> “**>**”
<VertStride>	::=	“**0**” | “**1**” | “**2**” | “**4**” | “**8**” | “**16**” | “**32**”
<Width>	::=	“**1**” | “**2**” | “**4**” | “**8**” | “**16**”
<HorzStride>	::=	“**0**” | “**1**” | “**2**” | “**4**”

7.4.8 Types

<SrcType>	::=	“**:f**” | “**:ud**” | “**:d**” | “**:uw**” | “**:w**” | “**:ub**” | “**:b**”
<SrcImmType>	::=	<SrcType> | “**:v**” | “**:vf**”
<DstType>	::=	<SrcType>

7.4.9 Write Mask

<WriteMask>	::=	“”
	|	“.” “**x**” | “.” “**y**” | “.” “**z**” | “.” “**w**”
	|	“.” “**xy**” | “.” “**xz**” | “.” “**xw**” | “.” “**yz**” | “.” “**yw**” | “.” “**zw**”
	|	“.” “**xyz**” | “.” “**xyw**” | “.” “**xzw**” | “.” “**yzw**”
	|	“.” “**xyzw**”

7.4.10 Swizzle Control

<Swizzle> ::= “”

| “.” <ChanSel>

| “.” <ChanSel> <ChanSel> <ChanSel> <ChanSel>

<ChanSel> ::= “**x**” | “**y**” | “**z**” | “**w**”

7.4.11 Immediate Values

<ImmAddrNum> ::= “**-512**”… “**511**”

<Imm32> ::= “**0.0**”… “**±1.0*2**$^{-128…127}$” | “**0**”…”**2**^{32}**-1**” | “**-2**^{31}”…”**2**^{31}**-1**”

<Imm16> ::= “**0**”…”**2**^{16}**-1**” | “**-2**^{15}”…”**2**^{15}**-1**”

<ImmDesc> ::= “**0**”…”**2**^{32}**-1**”

7.4.12 Predication and Modifiers

Instruction Predication

<Predicate> ::= “”

| “**(**” <PredState> <FlagReg> <PredCntrl> “**)**”

<PredState> ::= “”

| “**+**”

| “**-**“

<PredCntrl> ::= “”

| “**.x**” | “**.y**” | “**.z**” | “**.w**”

| “**.any2h**” | “**.all2h**”

| “**.any4h**” | “**.all4h**”

| “**.any8h**” | “**.all8h**”

| “**.any16h**” | “**.all16h**”

| “**.anyv**” | “**.allv**”

Source Modification

<SrcModifier> ::= “”

| “**-**”

| “**(abs)**”

| “**-**” “**(abs)**”

Instruction Modification

<ConditionalModifier> ::= “”

| <CondMod> “**.** ” <FlagReg>

<CondMod> ::= “**.z**” | “**.e**” | “**.nz**” | “**.ne**” | “**.g**” | “**.ge**” | “**.l**” | “**.le**” | “**.o**” | “**.r**” | “**.u**”

```
<Saturate>                  ::=   “”
                              |   “.sat”
```

Execution Size

```
<ExecSize>                  ::=   “(“ <NumChannels> “)”
<NumChannels>               ::=   “1” | “2” | “4” | “8” | “16” | “32”
```

7.4.13 Instruction Options

```
<InstOptions>               ::=   “”
                              |   “{” <InstOption> “}”
                              |   “{” <InstOption> <InstOptionEx> “}”

<InstOptionEx>              ::=   “”
                              |   “,” <InstOption> <InstOptionEx>

<InstOption>                ::=   <AccessMode>
                              |   <ComprCtrl>
                              |   <ThreadCtrl>
                              |   <DependencyCtrl>
                              |   <MaskCtrl> [Pre-DevSNB]
                              |   <SendCtrl>
                              |   <AccWrCtrl> [DevSNB+]
                              |   <WECtrl> [DevSNB+]

<AccessMode>                ::=   “Align1” | “Align16”
<ComprCtrl>                 ::=   “SecHalf” | “Compr”
<ThreadCtrl>                ::=   “Switch”
<DependencyCtrl>            ::=   “NoDDChk” | “NoDDClr”
<MaskCtrl>                  ::=   “NoMask”
<SendCtrl>                  ::=   “EOT”
```

Note for Assembler: Compression control “Compr” has a direct map to the binary instruction word. It may be omitted as long as the Assembler is able to determine whether an instruction is in compressed mode or not based on the execution size and the mode of operation.

7.5 Deprecated Features

7.5.1 Defeatured Instructions

The following instructions are removed from GEN implementation mainly due to implementation cost/schedule reasons. They are candidates for future generations.

- Sum of Absolute Difference 4 (sad4)
- Sum of Absolute Difference Accumulate 4 (sada4)
- Add Accumulate (aac)
- Min (min)
- Max (max)
- Next (next)
- Swizzle (swz)
- Dot Product Accumulate 2 (dpa2)
- Rotation Shift Left (rsl)
- Rotation Shift Right (rsr)

7.5.2 Others

The following features are also deprecated from GEN implementation.

- Restricted 16-bit Half Floating-Point Numbers

8. Instruction Set Reference

This chapter describes the functions of GEN instructions. Each GEN instruction is given a different page and the pages are sorted in alphabetical order according to assembly language mnemonic.

8.1 Conventions

8.1.1 Pseudo Code Format

The instructions are explained in the following pseudo-code format that resembles the GEN assembly instruction format.

```
[(<pred>)] opcode (<exec_size>) <dst> <src0> [<src1>]
```

Square bracket “[ ]” is used to signify that the field is optional. Saturation modifier and instruction options are omitted for simplicity.

8.1.2 General Macros and Definitions

INST_BYTE_COUNT is defined as a constant of 16 bytes.

```
#define INST_BYTE_COUNT    16   // byte count of instruction word
```

Function *floor()* converts a floating point value to an integral floating point value. For a given floating point value, from its closest two integral float values, function *floor()* returns the one that is closer to the negative infinity. For example, *floor*(1.3f) = 1.0f, and *floor*(-1.3f) = -2.0f.

```
float floor(float g)
{
     return maximum( any integral float f: f <= g)
}
```

Function *Condition()* takes the conditional signals {SN, ZR, OF, IN, NC} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data.

```
Bool Condition(result, cmod)
{
}
```

Function *ConditionNaN()* takes the conditional signals {SN, ZR, OF, IN, NC, NS} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data. The only difference between *Condition()* and *ConditionNaN()* is that *ConditionNaN()* uses the NS (NaN of the second source) signal.

```
Bool ConditionNaN(result, cmod)
{
}
```

Function *Jump()* jumps the instruction sequence from the current instruction location by *InstCount* number of instructions. If *InstCount* is a positive number, it jumps forward; if *InstCount* is a negative number, it jumps backward; if *InstCount* is zero, it is effectively an infinite loop on the current instruction.

```
void Jump(int InstCount)
{
      IP = IP + (InstCount * INST_BYTE_COUNT)
}
```

8.2 Evaluate Write Enable

The WrEn should be evaluated as below.

```
if (WECtrl == 1) {
      for (n =0; n < exec_size; n++) {
            WrEn[n] = 1;
      }
} else {
      for (n =0; n < exec_size; n++) {
            if (PcIP[n] == ExIP) {
                  WrEn[n] = 1;
            } else {
                  WrEn[n] = 0;
            }
      }
}

if (PredCtrl != “0000”) {
      for (n =0; n < exec_size; n++) {
            WrEn[n] = WrEn[n] & PMask[n];
      }
}
```

8.3 Instruction Description

The rest of the chapter contains the description of GEN instructions.

8.3.1 add – Addition

Opcode	Instruction	Description
64 (0x40)	add <dst> <src0> <src1>	Component-wise addition of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
●	●	●	●	[FLT]	[FLT]
●	●	●	●	[INT]	[INT]
●	●	●	●	[INT]	[FLT]

Format:

```
[(<pred>)] add[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] add[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] add[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] + src1.chan[n];
    }
}
```

Description:

The *add* instruction performs component-wise addition of <src0> and <src1> and stores the results in <dst>.

Addition of two floating point numbers follows rules in Table 8-1 (or Table 8-2), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-1. Floating point addition of A (column) and B (row) in IEEE mode

	–inf	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+inf**	**NaN**
–inf	–inf	–inf	–inf	–inf	–inf	–inf	–inf	NaN	NaN
–finite	–inf	*	A	A	A	A	**	+inf	NaN
–denorm	–inf	B	–0	–0	+0	+0	B	+inf	NaN
–0	–inf	B	–0	–0	+0	+0	B	+inf	NaN
+0	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+denorm	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+finite	–inf	**	A	A	A	A	***	+inf	NaN
+inf	NaN	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

*	Result can be { –finite}
**	Result can be {–finite, –0, +0, +finite}
***	Result can be { +finite}

Table 8-2. Floating point addition of A (column) and B (row) in ALT mode

	– fmax	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+ fmax**	********
–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–finite	+0	
–finite	–fmax	*	A	A	A	A	**	+fmax	
–denorm	–fmax	B	–0	–0	+0	+0	B	+fmax	
–0	–fmax	B	–0	–0	+0	+0	B	+fmax	
+0	–fmax	B	+0	+0	+0	+0	B	+fmax	
+denorm	–fmax	B	+0	+0	+0	+0	B	+fmax	
+finite	–finite	**	A	A	A	A	***	+fmax	
+fmax	+0	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	
********									

Notes:

*	Result can be { –fmax, –finite}
**	Result can be {–finite, –0, +0, +finite}
***	Result can be { +fmax, +finite}
****	Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

8.3.2 and – Logical And

Opcode	Instruction	Description
5 (0x05)	and <dst> <src0> <src1>	Performing component-wise logic AND of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] and[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] and[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] and[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] & src1.chan[n];
    }
}
```

Description:

The *and* instruction performs component-wise logic AND operation between <src0> and <src1> and stores the results in <dst>. Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction only applies to integer data types. The behavior is undefined if any operand is float.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.

- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

8.3.3 asr – Arithmetic Shift Right

Opcode	Instruction	Description
12 (0x0C)	asr <dst> <src0> <src1>	Performing component-wise arithmetic right shift of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] asr[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
     if (WrEn.channel[n] == 1) {
          if (src0.chan[n] >= 0) {
               dst.chan[n] = src0.chan[n] >> src1.chan[n];
          } else {
               int maskLSB = pow(2, src1.chan[n]) – 1;
               if (maskLSB & src0.chan[n] == 0) {
                    dst.chan[n] = sign(src0.chan[n]) *
                         ((abs)src0.chan[n] >> src1.chan[n]);
               } else {
                dst.chan[n] = sign(src0.chan[n]) *
                     ((abs)src0.chan[n] >> src1.chan[n])–1;
            }
        }
    }
}
```

Description:

The *asr* instruction performs component-wise arithmetic right shift of <src0> and storing the results in <dst>. Arithmetic right shift performs sign-extension by repeating the MSB of each data channel of <src0>. The amount of bit shift is provided by <src1>. Only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The rest of MSBs of <src1> data channels are ignored.

Operands for this instruction can be signed or unsigned integers, but cannot be floating point type. 5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction is effectively a power-of-2 integer divide with truncate in 2’s compliment form. Truncate in 2’s compliment form is also known as downward rounding – closest integer that is smaller than or equal to the result. For example, regardless of the bit shift amount in <src1>, the result of arithmetic right-shift of -1 (<src0>) is always -1.

Restrictions:

This instruction does not work with float type operands.

8.3.4 avg – Average

Opcode	Instruction	Description
66 (0x42)	avg <dst> <src0> <src1>	Component-wise averaging of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] avg[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = (src0.chan[n] + src1.chan[n] + 1) >> 1;
      }
}
```

Description:

The *avg* instruction performs component-wise integer average of <src0> and <src1> and stores the results in <dst>. An integer average uses integer upward rounding. It is equivalent to increment one to the addition of <src0> and <src1> and then apply an arithmetic right shift to this intermediate value.

Restrictions:

This instruction only applies to integer data types. The behavior is undefined if any operand is float.

Description:

The *break* instruction is used to early-out from the inner most loop, or early-out from the inner swtich block.

When used in a loop, upon execution, the *break* instruction terminates the loop for all execution channels enabled by PMask. This is performed by updating the per channel IP to the <UIP>. In case of all the enabled channels hit the *break* instruction, a jump will be performed to the instruction based <JIP>. <UIP> should be the offset to the end of the inner most conditional or loop block, <JIP> should be the offset to the first instruction after the loop block. In case of the *break* instruction directly under the loop, the <JIP> and the <UIP> will be the same.

When used in a switch block, predication is not allowed. When executed, the *break* instruction terminates the current enabled channels for the rest of the switch code block. In case of all the channels hit the *break* instruction, a jump will be performed to the instruction based on <JIP>. <JIP> should be the offset to the first instruction after the switch block. <JIP> and <UIP> must be the same when *break* is used inside a switch block.

The following table describes the 2 16-bit instruction pointer offset. Both the <JIP> and <UIP> are signed 16-bit numbers, added to IP pre-increment In GEN binary, <JIP> and <UIP> are at location <src1> and must be of type W (signed word integer).

Bit	Description
31:16	**UIP (Update Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2’s compliment
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2’s compliment

If SPF is ON, none of the PcIP is updated.

8.3.5 break – Break [DevGT+]

Opcode	Instruction	Description
40 (0x28)	Break <JIP><UIP>	Terminating enabled execution channels and conditionally breaking out from the inner most loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] break (<exec_size>) <JIP> <UIP>
```

Syntax:

```
[(<pred>)] break (<exec_size>) imm16 imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.channel[n] == 1) {
        if (PMask[n] == 1) {
            PcIP[n] = IP + <UIP>;
        }
        else {
            PcIP[n] = IP+1;
        }
    }
}
if (PcIP != (IP + 1)) { //all channels
    Jump(IP + <JIP>);
}
```

Description:

The *break* instruction is used to early-out from the inner most loop, or early-out from the inner swtich block.

When used in a loop, upon execution, the *break* instruction terminates the loop for all execution channels enabled by PMask. This is performed by updating the per channel IP to the <UIP>. In case of all the enabled channels hit the *break* instruction, a jump will be performed to the instruction based <JIP>. <UIP> should be the offset to the end of the inner most conditional or loop block, <JIP> should be the offset to the first instruction after the loop block. In case of the *break* instruction directly under the loop, the <JIP> and the <UIP> will be the same.

When used in a switch block, predication is not allowed. When executed, the *break* instruction terminates the current enabled channels for the rest of the switch code block. In case of all the channels hit the *break* instruction, a jump will be performed to the instruction based on <JIP>. <JIP> should be the offset to the first instruction after the switch block. <JIP> and <UIP> must be the same when *break* is used inside a switch block.

The following table describes the 2 16-bit instruction pointer offset. Both the <JIP> and <UIP> are signed 16-bit numbers, added to IP pre-increment In GEN binary, <JIP> and <UIP> are at location <src1> and must be of type W (signed word integer).

Bit	Description
31:16	**UIP (Update Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

need to add detail for SPF.

Restrictions:

Instruction compression is not allowed.

8.3.6 case – Case [DevGT+]

Opcode	Instruction	Description
38 (0x26)	case <dst> <src0> <src1> <JIP>	Signifying the start of an case block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•		•			

Format:

```
[(<pred>)] case (<exec_size>) null null null <JIP>
           case[.<cmod>] (<exec_size>) null <src0> <src1> <JIP>
```

Syntax:

```
[(<pred>)] case (<exec_size>) null null null imm16
           case[.<cmod>] (<exec_size>) null reg reg imm16
           case[.<cmod>] (<exec_size>) null reg imm32 imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.channel[n] == 1) {
        if (<cmod> == 0) { // no embedded compare
            if (PMask.channel[n] == 0) {
                PcIP[n] = IP + <JIP>;
            }
            else
            {
                PcIP[n] = IP + 1;
            }
        }
        else { // with embedded compare
            if (cmod.channel[n] == 0) {
                PcIP[n] = IP + <JIP>;
            }
            else
            {
```

```
                        PcIP[n] = IP + 1;
                  }
            }
      }
}
if (<cmod> == 0) { // no embedded compare
      if (PcIP != (IP+1)) { // all channels false
            Jump(IP + <JIP>);
      }
}
else { // with embedded compare
      if (PcIP != (IP+1)) { // all channels false
            Jump(IP + <JIP>);
      }
}
```

Description:

The *case* instruction starts an case/break code block. It restricts execution within the conditional block to only those channels that were enabled via either the predicate control or the condition from <cmod>.

Each *case* instruction must have a matching *break* instruction.

If all channels are inactive (for the case/break block), a jump is performed of the relative distance as specified in the instruction. This jump must be to right after the matching *break* instruction when present, or otherwise to the end of switch code block.

The following table describes the 16-bit exit code <JIP>. <JIP> must be an immediate operand, whereas it is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN instruction binary, <JIP> is at location <dst> and must be of type W (signed word integer).

Bit	Description
15:0	**JIP (Jump Instruction Count).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

need to add detail for SPF.

Restrictions:

Instruction compression is not allowed.

To use embedded compares, the predicate control field for this instruction must be zero, and the conditional modifier field must be none zero.

To use predicated *case* instruction, the conditional modifier field must be zero.

The *case* code block must end with a *break* instruction.

8.3.7 cmp – Compare

Opcode	Instruction	Description
16 (0x10)	cmp.<cmod> <dst> <src0> <src1>	Component-wise comparison of <src0> and <src1> according to conditional modifier in <cmod> and storing the results in flag register in <cmod> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] cmp[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] cmp[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] cmp[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      bitMask[n] = 0;
      if (WrEn.chan[n] == 1) {
            results[n] = src0.chan[n] - src1.chan[n];
            bitMask[n] = Condition(results[n]);
            dst.chan[n][0] = bitMask[n];
      }
}
flag# = bitMask;
```

Description:

The *cmp* instruction performs component-wise comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional code (excluding NS signal) based on the conditional modifier, and storing the conditional bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it is not null, for the enabled channels, the LSB of the result in the destination channel contains the flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by ‘and-ing’ an 0x01 constant).

If <exec_size> is equal or less than 8, when ‘*SecHalf*’ option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmp* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

8.3.8 cmpn – Compare NaN

Opcode	Instruction	Description
17 (0x11)	cmpn.<cmod> <dst> <src0> <src1>	Performing component-wise special NaN comparison of <src0> and <src1> according to conditional modifier in <cmod> and storing the results in flag register in <cmod> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] cmpn[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] cmpn[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] cmpn[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    bitMask[n] = 0;
    if (EMask.chan[n] == 1) {
        results[n] = src0.chan[n] - src1.chan[n];
        bitMask[n] = ConditionNaN(results[n]);
        dst.chan[n][0] = bitMask[n];
    }
}
flag# = bitMask;
```

Description:

The *cmpn* instruction performs component-wise special-NaN comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional signals including NS based on the conditional modifier, and storing the conditional flag bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it is not null, for the enabled channels, the LSB of the result in the destination channel contains the flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by ‘and-ing’ an 0x01 constant).

If <exec_size> is equal or less than 8, when ‘*SecHalf*’ option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmpn* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

This instruction is similar to *cmp*. The only difference is that if the second source operand <src1> is a NaN, the result for any conditional modifier except **.nz** is true (see details in Section **Error! Reference source not found.**).

For integer operands, *cmpn* and *cmp* are identical.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

8.3.9 cont – Continue [DevGT+]

Opcode	Instruction	Description
41 (0x29)	cont <JIP><UIP>	Temporally disabling enabled execution channels for the remainder of the inner most loop and conditionally jumping to the last instruction (while) of the loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] cont (<exec_size>) <JIP> <UIP>
```

Syntax:

```
[(<pred>)] cont (<exec_size>) imm16 imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.channel[n] == 1) {
            if (PMask[n] == 1) {
                  PcIP[n] = IP + <UIP>;
            }
            else {
                  PcIP[n] = IP + 1;
            }
      }
}
if (PcIP != (IP+1)) { //all channel true
      Jump(IP + <JIP>);
}
```

Description:

The *cont* instruction disables execution for the subset of channels for the remainder of the current loop iteration. Channels remain disabled until right before the *while* instuction or right before the condition check code block for the *while* instruction. In case of all enabled channels hit this instruction, a jump is made a distance of <JIP> where execution continues.

The following table describes the two 16-bit exit code, <JIP> and <UIP>. The field are signed 16-bit numbers, added to IP pre-increment. The <UIP> should always point to the loop's associated 'while' instruction, and the <JIP> should point to the last instruction of the inner most conditional block if the *cont* instruction is inside a conditional block. In case of the *break* instruction directly under the loop, the <JIP> and the <UIP> will be the same. In GEN binary, <JIP> and <UIP> are at location <src1> and must be of type W (signed eword integer).

Bit	Description
31:16	**UIP (Update Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

need to add detail for SPF.

(1) PMask here is for all the channels enabled for the *cont* instruction.

Restrictions:

Instruction compression is not allowed.

8.3.10 do – Do

Opcode	Instruction	Description
38 (0x26)	do	Updating mask and mask stack to enter a do-while loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
do
```

Syntax:

```
do
```

Pseudocode:

```
Evaluate(EMask);
LStack.push(LMask);
LStack.push(CMask);
LMask = EMask;
CMask = EMask;
```

Description:

The *do* instruction indicates the start of a do-while block. Each *do* must have a matching *while* instruction. Execution of the *do* instruction causes the LMask and CMask (in that order) to be saved to the LStack for preservation and eventual restoration upon completion of the do-while block.

This instruction is equivalent to two *msave* instructions (in the order of "*msave lstack lmask*" and "*msave lstack cmask*"). It is an efficient construct for a do-while block.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

SPF effectively turns this instruction into a nop, as LMask and CMask should be coherent with EMask. It may be used as instruction filler for code readability keeping in mind that a nop wastes an instruction cycle.

Restrictions:

Predication is not allowed. Instruction compression does not apply to this instruction.

Execution size is ignored for this instruction.

8.3.11 p2 – Dot Product 2

Opcode	Instruction	Description
87 (0x57)	Dp2 <dst> <src0> <src1>	Performing two-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] dp2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg imm32
Pseudocode: Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
     fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1];
     if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
     if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
     if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
     if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp2* instruction performs a two-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every third and fourth elements of <src0> (post-source-swizzle if present) are not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of four channels is 0.0f, the sign of the result correctly reflects the input data of the first two channels of the group of four.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation.

Horizontal stride must be 1.

[Pre-DevSNB]The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

8.3.12 dp3 – Dot Product 3

Opcode	Instruction	Description
86 (0x56)	dp3 <dst> <src0> <src1>	Performing three-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] dp3[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
     fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1]
          + src0.chan[n+2] * src1.chan[n+2];
     if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
     if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
     if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
     if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp3* instruction performs a three-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every fourth element of <src0> (post-source-swizzle if present) is not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of 4 channels is 0.0f, the sign of the result correctly reflects the input data of the first three channels of the group of four.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation.

Horizontal stride must be 1.

[Pre-DevSNB]The results are NOT stored in the accumulator register. This instruction does implicitly update accumulator register, however with undefined values.

[DevSNB+]The results can be stored in the accumulator register, the channels are wirrten the same way as the <dst> register with replication.

8.3.13 dp4 – Dot Product 4

Opcode	Instruction	Description
84 (0x54)	dp4 <dst> <src0> <src1>	Performing four-wide dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] dp4[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
     fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1]
          + src0.chan[n+2] * src1.chan[n+2]
          + src0.chan[n+3] * src1.chan[n+3];
     if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
     if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
     if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
     if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dp4* instruction performs a four-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation.

Horizontal stride must be 1.

[Pre-DevSNB]The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

8.3.14 dph –Dot Product Homogeneous

Opcode	Instruction	Description
85 (0x55)	dph <dst> <src0> <src1>	Performing four-wide homogeneous dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] dph[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
      fTmp = src0.chan[n] * src1.chan[n]
            + src0.chan[n+1] * src1.chan[n+1]
            + src0.chan[n+2] * src1.chan[n+2]
            + src1.chan[n+3];
      if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
      if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
      if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
      if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}
```

Description:

The *dph* instruction performs a four-wide homogeneous dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to dp4 except that every fourth element of <src0> (post-source-swizzle if present) is forced to 1.0f.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation.

Horizontal stride must be 1.

[Pre-DevSNB]The results are NOT stored in the accumulator register. This instruction does implicitely update accumulator register, however with undefined values.

8.3.15 else – Else [DevGT+]

Opcode	Instruction	Description
36 (0x24)	else <JIP>	An optional statement within an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
else (<exec_size>) <JIP>
```

Syntax:

```
else (<exec_size>) imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.channel[n] == 1) {
            PcIP[n] = IP + <JIP>;
      }
}
if (PcIP != (IP+1)) { // for all channels
      Jump(IP + <JIP>);
}
```

Description:

The *else* instruction is an optional statement within an *if/else/endif* block of code. It restricts execution within the else/endif portion to the opposite set of channels enabled under the if/else portion. Channels which were inactive prior to entering the *if/endif* block remain inactive throughout the entire block.

All enabled channels upon arriving the *else* instruction will be redirected to the matching *endif*. If all channels are redirected (by *else* or before *else*), a relative jump is performed to the location specified by <JIP>. The jump target should be the the matching *endif* instruction for that conditional block.

The following table describes the 16-bit <JIP>. In GEN binary, <JIP> is at location <dst> and must be of type W (signed word integer). <JIP> must be an immediate operand, it is a signed 16-bit number and is intended to be forward referencing. This value is added to IP pre-increment.

Bit	Description
15:0	**JIP (Jump Instruction Count).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

Restrictions:

Instruction compression is not allowed.

Predication is not allowed.

8.3.16 endif – End-If

Opcode	Instruction	Description
37 (0x24)	endif <JIP>	Restoring execution to those data channels that were active prior to the if/else/endif block. Jump to next hop point if all channels are disabled.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
endif <JIP>
```

Syntax:

```
endif imm16
```

Pseudocode:

```
Evaluate(WrEn);
if (WrEn == 0) { // all channels false
    Jump(IP + <JIP>);
}
```

Description:

The *endif* instruction terminates an if/else/endif block of code. It restores the execution to these data channels that were active prior to the if/else/endif block.

The *endif* instruction is also used to hop out nested conditionals by jumping to the end of the next outer conditional block when all channels are disabled.

The following table describes the 16-bit jump target offset <JIP>. In GEN binary, <JIP> is at location <dst> and must be of type W (signed word integer). <JIP> must be an immediate operand.

Bit	Description
31:16	**Reserved:** MBZ
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

Opcode	Instruction	Description
20 (0x14)	f16to32 <dst> <src0>	Component-wise convert the half precision float in <src0> to single precision float and storing in <dst>.

Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
	•	•	•	•	[W]	[FLT]

Format:

```
[(<pred>)] f16to32[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] f16to32[.<cmod>] (<exec_size>) reg reg
[(<pred>)] f16to32[.<cmod>] (<exec_size>) reg imm16
```

Pseudocode:

```
      Evaluate(WrEn);
      for (n = 0; n < exec_size; n++) {
         if (WrEn.chan[n] == 1) {
            dst.chan[n] = convert half precision float to single precision
   float(src0.chan[n]);
         }
      }
```

Description:

The *f16to32* instruction converts the half precision float in <src0> to single precision float and storing in <dst>.

Since half precision float is not a defined type in Gen, the source data type for *f16to32* instruction must be Word.

Table n-n: Floating point coversion in IEEE mode

Half Precision Float	Single Precision Float
-inf	-inf
-finite	-finite
-denorm	-finite
-0	-0

+0	+0
+denorm	+finite
+finite	+finite
+inf	+inf
NaN	NaN

Input denorm should not be flushed.

Restrictions:

```
        if (src0 == VMask) {
            if (VMask.channel[n] == 0) {
                PcIP[n] = IP + <JIP>;
            }
            else {
                PcIP[n] = IP+1;
            }
        }
    }

    if (PcIP != (IP+1)) { // for all channels
        Jump(IP + <JIP>);
    }
}
```

Description:

The *fork* instruction starts a *fork* block of code. It restricts execution within the *fork* block to only those channels that were enabled via select register (dispatch mask or vector mask) and further qualified by the predicate control.

If all channels are inactive (for the *fork* block), a jump is performed of the relative distance as specified in the instruction.

The following table describes the 16-bit exit code <JIP>. <JIP> must be an immediate operand, whereas JumpIP is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN instruction binary, <JIP> is at location <dst> and must be of type W (signed word integer).

If the register specified is NULL, fork simply set SPF = 1.

The *fork* instruction should be used to turn SPF off in a middle of a program, it will bring the PcIPs the the current ExIP for continuing execution.

Bit	Description
15:0	**JIP (Jump Instruction Count).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

Restrictions:

Instruction is only allowed at global scope.

Src0 is limited to DMask.0, Dmask.1, Vmask.0, Vmask.1 and NULL.

- The *fork* instruction should only be used to enable 32 channel execution where per-channel branch is required. The SPF control must be set during thread dispatch time based on 32 channel dispatch. Within the shader, the fork instruction should only be used to switch between the 3 modes below:
  - 32 channel execution in SPF on mode, where no per-channel branch can be performed
  - lo-16 channel execution in SPF off mode, where per-channel branch on the low 16 channels can be performed, **QtrCtrl {H1} must be used in this case.**
  - hi-16 channel execution in SPF off mode, where per-channel branch on the high 16 channels can be performed, **QtrCtrl {H2} must be used in this case.**
- Qtr control is not allowed for the *fork* instruction.

8.3.17 frc – Fraction

Opcode	Instruction	Description
67 (0x43)	frc <dst> <src0>	Taking component-wise truncate-to-minus-infinity fraction operation of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[FLT]	[FLT]
•		•	•	[FLT]	[INT]

Format:

```
[(<pred>)] frc[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] frc[.<cmod>] (<exec_size>) reg reg
[(<pred>)] frc[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
     if (WrEn.chan[n] == 1) {
          dst.chan[n] = src0.chan[n] – floor(src0.chan[n]);
     }
}
```

Description:

The *frc* instruction computes, component-wise, the truncate-to-minus-infinity fractional values of <src0> and stores the results in <dst>. The results, in the range of [0.0, 1.0], are the fractional portion of the source data.

Source operand for this instruction must be of floating point type. This instruction can only operate on normalized floating source and therefore can not take accumulator as source or destination operand.

This instruction only applies to floating point operands.

Floating point fraction computation follows rules in Table 8-3 (or Table 8-4), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-3. Floating point fraction computation in IEEE mode

<src0>	**–inf**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+inf**	**NaN**
<dst>	NaN	*	+0	+0	+0	+0	*	NaN	NaN

Notes:

* Result is in the range of [+0, 1) – not including 1.

Table 8-4. Floating point fraction computation in ALT mode

<src0>	**– fmax**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+ fmax**	**
<dst>	+0	*	+0	+0	+0	+0	*	+0	▨

Notes:

* Result is in the range of [+0, 1) – not including 1.

** Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

Saturation modifier does not apply to this instruction.

This instruction can not take accumulator as source or destination operand as it can only operate on normalized floating source.

This instruction does implicitely update accumulator register when enabled, however with undefined values.

In case of the *halt* instruction not inside any conditional code block, the value of <JIP> and <UIP> should be the same. In case of the *halt* instruction inside conditional code block, the <UIP> should be the end of the program, and the <JIP> should be end of the most inner conditional code block.

The following table describes the 16-bit jump offsets. In GEN binary, <JIP> and <UIP> are at location <src1> and must be of type W (signed word integer). The <JIP> and <UIP> are added to IP pre-increment.

Bit	Description
31:16	**UIP (Update Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the channel. Format = S15. Signed integer in 2's compliment
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

If SPF is ON, none of the PcIP is updated. The UIP must be used to update the execution IP, the JIP is not used in this case.

Restrictions:

<dst> and <src0> must be NULL.jmpi – Jump Indexed

Opcode	Instruction	Description
32 (0x20)	jmpi <index>	Redirecting program execution to <index> instructions forward of the current post-incremented instruction pointer.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•					

Format:

```
[(<pred>)] jmpi (1) <exitcode> {NoMask}
```

Syntax:

```
[(<pred>)] jmpi (1) reg32 {NoMask}
[(<pred>)] jmpi (1) imm32 {NoMask}
```

Pseudocode:

```
Evaluate(WrEn);
if (WrEn != 0) {
     Jump(<exitcode.index> + 1); of // +1 if compacted, +2 if not.
}
```

Description:

The *jmpi* instruction redirects program execution to <exitcode.index> instructions forward of the current **post-incremented** instruction pointer. <exitcode.index> is treated a signed integer value, with positive integers or zero generating forward jumps, and negative integers generating backward jumps. An <exitcode.index> value of 0 means execution continues at the instruction immediately following the *jmpi* instruction, while an index value of -1 would imply an infinite loop.

Bit	Description
31:16	Reserved: MBZ
15:0	**index (Jump Index)** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

<exitcode> may be a scalar register or an immediate. The data type of <exitcode > must be D (signed doubleword integer). However, hardware only uses lower 16 bits of <exitcode>. The valid range of <exitcode.index> is [–32768, 32767]. Behavior for <exitcode.index> outside that range is undefined.

This instruction executes regardless of the calculated WrEn at the time of issue. – To reduce hardware complexity, instruction optional control {NoMask} must be set for this instruction. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

Execution size must be 1.

Predication is allowed to provide conditional jump with a scalar condition. As the execution size is 1, the first channel of PMASK (flags post prediction control and negate) is used to determine whether the jump is taken or not. If the condition is false, the jump is not taken and the IP immediately following will be executed next.

In GEN binary, <exitcode.index> is at location <src1>. IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

If SPF is ON, none of the PcIP is updated.

8.3.18 halt – Halt [DevGT+]

Opcode	Instruction	Description
42 (0x2A)	halt <JIP>	Temporarily suspending execution for all enabled execution channels.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

```
[(<pred>)] halt (<exec_size>) <JIP>
```

Syntax:

```
[(<pred>)] halt (<exec_size>) imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.channel[n] == 1) {
            PcIP[n] = IP + <JIP>;
      else {
            PcIP[n] = (IP+1);
      }
}
if (PcIP != (IP+1)) { // for all channels
      Jump(IP + <JIP>);
}
```

Description:

The *halt* instruction temporarily suspends execution for all enabled compute channels. The value of AMask is updated, with bits in positions of enabled channels set to '0'. If all the bits of the resultant AMask are cleared, a jump is made <inst_count> instructions away.

The *halt* instruction is also used inside subroutines as a 'return', utilizing AMask to keep track of which execution channels have returned and which to continue execution. Since there is no hardware mask stack for AMask, software must manually preserve the value of AMask around a subroutine call.

The following table describes the 16-bit jump offset <JIP>. In GEN4 binary, <JIP> is at location <src1> and must be of type W (signed word integer). The <JIP> is added to IP pre-increment.

Bit	Description
15:0	**JIP (Jump Instruction Count).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

8.3.19 if – If [DevGT+]

Opcode	Instruction	Description
34 (0x22)	if <dst> <src0> <src1> <JIP>	Signifying the start of an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•		•			

Format:

```
[(<pred>)] if (<exec_size>) null null null <JIP>
           if[.<cmod>] (<exec_size>) null <src0> <src1> <JIP>
```

Syntax:

```
[(<pred>)] if (<exec_size>) null null null imm16
           if[.<cmod>] (<exec_size>) null reg reg imm16
           if[.<cmod>] (<exec_size>) null reg imm32 imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.channel[n] == 1) {
            if (<cmod> == 0) { // no embedded compare
                  if (PMask.channel[n] == 0) {
                        PcIP[n] = IP + <JIP>;
```

```
                }
                else {
                    PcIP[n] = IP+1;
                }
            }
            else { // with embedded compare
                if (cmod.channel[n] == 0) {
                    PcIP[n] = IP + <JIP>;
                }
                else {
                    PcIP[n] = (IP+1);
                }
            }
        }
    }
    if (<cmod> == 0) { // no embedded compare
        if (PcIP != (IP+1)) { // for all channels
            Jump(IP + <JIP>);
        }
    }
    else { // with embedded compare
        if (PcIP != (IP+1)) { // for all channels
            Jump(IP + <JIP>);
        }
    }
```

Description:

The *if* instruction starts an if/endif or an if/else/endif block of code. It restricts execution within the conditional block to only those channels that were enabled via either the predicate control or the condition from <cmod>.

Each *if* instruction must have a matching *endif* instruction and may have up to one matching *else* instruction before *endif*.

If all channels are inactive (for the if/endif or if/else block), a jump is performed of the relative distance as specified in the instruction. This jump must be to right after the matching *else* instruction when present, or otherwise to the matching *endif* instruction of that conditional block.

The following table describes the 16-bit exit code <JIP>. <JIP> must be an immediate operand, whereas JumpIP is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN instruction binary, <JIP> is at location <dst> and must be of type W (signed word integer).

Bit	Description
15:0	**JIP (Jump Instruction Count).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

need to add detail for SPF.

Restrictions:

Instruction compression is not allowed.

To use embedded compares, the predicate control field for this instruction must be zero, and the conditional modifier field must be none zero.

To use predicated *if* instruction, the conditional modifier field must be zero.

8.3.20 line – Line

Opcode	Instruction	Description
89 (0x59)	Line <dst> <src0> <src1>	Computing a component-wise line equation (v = p*u+q) of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] line[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] line[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] line[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
for (n = 0; n < exec_size; n++) {
    dwP = src0.RegNum.SubRegNum[bits4:2]  // a DW aligned scalar
    dwQ = src0.RegNum.(SubRegNum[bit4]|0x8)     // 4-th component
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = dwP * src1.chan[n] + dwQ
    }
}
```

Description:

The *line* instruction computes a component-wise line equation ($v = p*u+q$ where u/v are vectors and p/q are scalars) of <src0> and <src1> and storing the results in <dst>. <src1> is the input vector *u*. <src0> provides input scalars *p* and *q*, where *p* is the scalar value based on the region description of <src0> and *q* is the scalar value implied from <src0> region. Specifically, *q* is the fourth component of the 4-tuple (128-bit aligned) that *p* belongs to.

Restrictions:

This is a specialized instruction that only support execution size of 8 or 16.

<src0> region must be a replicated scalar (with HorzStride = VertStride = 0).

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

In particular, <src0> must be float. <src1> may be float, byte or word integer. <src1> cannot be dword integer. <dst> may be float or integer of any size.

Source operands cannot be an accumulator register.

<src0> for *line* instruction has to have .0 or .4 as the subregister number.

8.3.21 lzd – Leading Zero Detection

Opcode	Instruction	Description
74 (0x4A)	lzd <dst> <src0>	Performing component-wise leading zero detection of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•		[INT]	[INT]

Format:

```
[(<pred>)] lzd[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] lzd[.<cmod>] (<exec_size>) reg reg
[(<pred>)] lzd[.<cmod>] (<exec_size>) reg reg
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            UD udScalar = src0.chan[n];
            UD cnt = 0;
            while ( (udScalar & (1<<31)) == 0 && cnt != 32) {
                  cnt ++;
                  udScalar = udScalar << 1;
            }
            dst.chan[n] = cnt;
      }
}
```

Description:

The *lzd* instruction counts component-wise the leading zeros from <src0> and storing the resulting counts in <dst>.

This instruction only work on unsigned dword source. Source operand may be a signed or unsigned. If it is a signed integer, source modifier (abs) must be used to convert the source into an unsigned integer type.

The destination operand must also be of unsigned dword type.

Restrictions:

The destination operand cannot be the accumulator.

This instruction does implicitely update accumulator register when enabled, however with undefined values.

8.3.22 lrp – Linear Interpolation

Opcode	Instruction	Description
92 (0x5c)	lrp <dst> <src0> <src1>	Computing a component-wise lrp equation (w = u*x+v*(1-x) of vectors (u, v, x) from <src0> and vector <src1> (and implied vector <src2>) and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
●		●	●	[FLT]	[FLT]

Format:

```
[(<pred>)] lrp[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] lrp[.<cmod>] (<exec_size>) reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] * src2.chan[n] + src1.chan[n] * (1 – src2.chan[n])
    }
}
```

Description:

The *lrp* instruction takes component-wise multiplication of <src0> and <src1>, and adds the result to the component-wise multiplication of <src2> and (1 - <src0>) , and then stores the final results in <dst>.

The *lrp* instruction uses the 3-source instruction format.

Restrictions:

This instruction only supports float source and destination.

Immediate source is not allowed for *lrp*.

The vertical stride is overloaded to 4 in HW for 3-src instructions.

The overflow conditional modifier is not allowed.

8.3.23 mac – Multiply Accumulate

Opcode	Instruction	Description
72 (0x48)	mac <dst> <src0> <src1>	Performing component-wise multiply accumulate of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] mac[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mac[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mac[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = src0.chan[n] * src1.chan[n] + acc0.chan[n]
      }
}
```

Description:

The *mac* instruction takes component-wise multiplication of <src0> and <src1>, adds the results with the corresponding accumulator values, and then stores the final results in <dst>.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Explicit source operands cannot be accumulator.

This instruction doesn’t support dword integers (D or UD).

8.3.24 mach – Multiply Accumulate High

Opcode	Instruction	Description
73 (0x49)	mach <dst> <src0> <src1>	Performing component-wise multiply accumulation of <src0>, <src1> and accumulator register, and returning the high dword of results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] mach[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mach[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mach[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      acc0.chan[n][63:0]
            = (src0.chan[n][31:16] * src1.chan[n][31:0])<<16
                  + acc0.chan[n][63:0];
      if (WrEn.channel[n] == 1) {
            dst.channel[n][31:0] = acc0.chan[n][63:32]
      }
}
```

Description:

The *mach* instruction performs dword integer multiply-accumulate operation and outputs the high dword (bits [63:32]). On a component by component basis, this instruction multiplies dwords in <src1> with the high words of dwords in <src0>, left-shifts the results by 16 bits, adds them with the corresponding accumulator values, and keeps the whole 64-bit results in the accumulator. It then stores the high dword (bits [63:32]) of the results in <dst>.

This instruction is intended to be used to emulate 32-bit dword integer multiplication by utilizing the large number of bits available in the accumulator. For example, the following three instructions perform vector multiplication of two 32-bit signed integer source from r2 and r3 and store the resulting vectors with high 32-bit in r4 and low 32-bit in r5.

```
mul  (8) acc0:d      r2.0<8;8,1>d    r3.0<8;8,1>:d

mach (8) rTemp<1>:d r2.0<8;8,1>d    r3.0<8;8,1>:d

mov  (8) r5.0<1>:d    rTemp:d // hi-32bits

mov  (8) r6.0<1>:d   acc0:d // lo-32bits
```

The MUL and MACH instruction should have all channels enabled. The first MOV should have channel enable from the destHI of IMUL, the second MOV should have the channel enable from the destLO of IMUL.

As *mach* is used to generate part of 64-bit dword integer results, saturation modifier should not be used. In fact, saturation modifier should not be used for any of these three instructions.

Source and destination operands must be dword integers. Source and destination must be of the same type, signed integer or unsigned integer.

- If <dst> is UD, <src0> and <src1> may be UD and/or D. However, if any of <src0> and <src1> is D, source modifier, (abs), must be present to convert it to match with <dst>.
- If <dst> is D, <src0> and <src1> must also be D. They cannot be UD as it may cause unexpected overflow because the computed results are limited to 64 bits.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Therefore, explicit source operands cannot be accumulator.

8.3.25 mad – Multiply Add

Opcode	Instruction	Description
91 (0x5B)	mad <dst> <src0> <src1> <src2>	Component-wise multiply add of <src0>, <src1>, and <src2> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] mad[.<cmod>] (<exec_size>) <dst> <src0> <src1> <src2>
```

Syntax:

```
[(<pred>)] mad[.<cmod>] (<exec_size>) reg reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
     if (WrEn.chan[n] == 1) {
          dst.chan[n] = src1.chan[n] * src2.chan[n] + src0.chan[n]
     }
}
```

Description:

The *mad* instruction follows the 3-src instruction format.

The *mad* instruction takes component-wise multiplication of <src1> and <src2>, adds the results with the corresponding <src0> values, and then stores the final results in <dst>.

Restrictions:

This instruction only supports float source and destination.

Immediate source is not allowed for *mad*.

The vertical stride is overloaded to 4 in HW for 3-src instructions.

8.3.26 movi – Move Indexed

Opcode	Instruction	Description
3 (0x3)	movi <dst> <src0>	Fast component-wise indexed move from <src0> to <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] movi[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] movi[.<cmod>] (<exec_size>) reg reg
```

Pseudocode:

```
Evaluate(WrEn)
srcregfile = regfile(src0)
srcreg = reg(address[0])
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        srcsubreg = subreg(address[n] + addr imm)
        dst.chan[n] = srcregfile.srcreg.srcsubreg
    }
}
```

Description:

The *movi* instruction performs a fast component-wise indexed move for subfields from <src0> to <dst>. The source operand must be an indirectly-addressed register. All channels of the source operand share the same register number, which is provided by the register field of the first address subregister, with a possible immediate address offset. The register fields of the subsequent address subregisters are ignored by hardware. The subregister number of a source channel is provided by the subregister field of the corresponding address subregister.

Destination register may be either a directly-addressed or an indirectly-addressed register.

This instruction effectively performs a subfield shuffling from one register to another. Up to eight subfields can be selected by an instruction.

Restrictions:

Source operand must be a GRF register.

Source and destination must be the same type.

This instruction does not implicitly update accumulator register.

Execution size must be less than 16 (as there are only 8 address registers)

Address register for source must be aligned to the base (a0.0).

Destination register (directly or indirectly addressed) must be half GRF aligned (i.e. 16-byte aligned).

Destination stride in unit of byte must equal to the source element size in unit of byte.

Align16 access mode is not allowed for MOVI.

Alll the index registers used in MOVI instruction must all point to the same GRF register.

MOVI must use 1x1 indirect regioning.

MOVI is always based on register offset zero no matter what the destination offset is. The destination offset is used in HW only to create channel enables. The first index register (a0.0) will always be used to select the first element of the destination start from offset zero. Each index register will be used to select 1 element if type is byte, each index register will be used to select 2 elements if type is word, and each index register will be used to select 4 elements if type is dword

Conditional Modifier is not allowed for this instruction.

HW Implementation Details:

The destination offset of the *movi* instruction is only used in HW to generate destination write enables. Each element of the destination is directly mapped to the index registers for the *movi* instruction.

For Byte *movi*, byte0 of the destination is selected by a0.0[4:0], byte1 is selected by a0.1[4:0], ..., and byte7 is selected by a0.7[4:0]. The rest of the bytes are undefined.

For Word *movi*, byte0 of the destination is selected by (a0.0[4:1] & “0”), byte1 is selected by (a0.0[4:1] & “1”), byte2 is selected by (a0.1[4:1] & “0”), byte3 is selected by (a0.1[4:1] & “1”), ..., and byte15 is selected by (a0.7[4:1] & “1”). The rest of the bytes are undefined.

For DWord or float *movi*, byte0 of the destination is selected by (a0.0[4:2] & “00”), byte1 is selected by (a0.0[4:2] & “01”), byte2 is selected by (a0.0[4:2] & “10”), byte3 is selected by (a0.0[4:2] & “11”), byte4 is selected by (a0.1[4:2] & “00”), byte5 is selected by (a0.1[4:2] & “01”), ..., byte31 is selected by (a0.7[4:2] & “11”).

For all 3 conditions above, a0.n’[4:0] = a0.n[4:0] + addr_imm[4:0].

8.3.27 mul – Multiply

Opcode	Instruction	Description
65 (0x41)	mul <dst> <src0> <src1>	Performing component-wise multiplication of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
[(<pred>)] mul[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] mul[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] mul[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = src0.chan[n] * src1.chan[n];
      }
}
```

Description:

The *mul* instruction performs component-wise multiplication of <src0> and <src1> and stores the results in <dst>.

When both <src0> and <src1> are of type D or UD, only the lower 16 bits of each element of <src0> are used. Accumulator maintains full 48-bit precision. The macro described in *mach* instruction should be used to obtain the full precision 64 bits multiplication results.

Multiplication of two floating point numbers follows rules in Table 8-5 (or Table 8-6), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-5. Floating point multiplication of A (column) and B (row) in IEEE mode

	–inf	**–finite**	**–1.0**	**–denorm**	**–0**	**+0**	**+denorm**	**+1.0**	**+finite**	**+inf**	**NaN**
–inf	+inf	+inf	+inf	NaN	NaN	NaN	NaN	–inf	–inf	–inf	NaN
–finite	+inf	*	–A	+0	+0	–0	–0	A	**	–inf	NaN
–1.0	+inf	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–inf	NaN
–denorm	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
–0	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
+0	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+denorm	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+1.0	–inf	B	–1.0	–0	–0	+0	+0	+1.0	B	+inf	NaN
+finite	–inf	**	–A	–0	–0	+0	+0	A	*	+inf	NaN
+inf	–inf	–inf	–inf	NaN	NaN	NaN	NaN	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Note:

* Result may be {+finite, +inf (overflow)}

** Result may be {–inf (overflow), –finite}

Table 8-6. Floating point multiplication of A (column) and B (row) in ALT mode

	– fmax	**–finite**	**–1.0**	**–denorm**	**–0**	**+0**	**+denorm**	**+1.0**	**+finite**	**+fmax**	*******
– fmax	+fmax	+fmax	+fmax	**–0**	**–0**	**+0**	**+0**	–fmax	–fmax	–fmax	
–finite	+fmax	*	–A	+0	+0	–0	–0	A	**	–fmax	
–1.0	+fmax	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–fmax	
–denorm	+0	+0	+0	+0	+0	–0	–0	–0	–0	–0	
–0	+0	+0	+0	+0	+0	–0	–0	–0	–0	–0	
+0	–0	–0	–0	–0	–0	+0	+0	+0	+0	+0	
+denorm	–0	–0	–0	–0	–0	+0	+0	+0	+0	+0	
+1.0	–fmax	B	–1.0	–0	–0	+0	+0	+1.0	B	+fmax	
+finite	–fmax	**	–A	–0	–0	+0	+0	A	*	+fmax	
+fmax	–fmax	–fmax	–fmax	–0	–0	+0	+0	+fmax	+fmax	+fmax	
*******											

Note:

* Result may be {+finite, +fmax (overflow)}

** Result may be {–fmax (overflow), –finite}

*** Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Source operands cannot be an accumulator register.

When operating on integers with at least one of the source being a dword type (signed or unsigned), the destination cannot be a float (implementation note: the data converter only looks at the lower 34 bits of the results).

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

When operating on integers with at least one of the source being a dword type (signed or unsigned), the Overflow and Sign flags are undefined. Therefore, conditional modifier and instruction operation '.sat' cannot be used.

When multiple a DW and a W, the W has to be on src0, and the DW has to be on src1.

8.3.28 nop – No Operation

Opcode	Instruction	Description
126 (0x7E)	nop	Issuing an dummy instruction and performing no operation.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
nop
```

Syntax:

```
nop
```

Pseudocode:

```
n/a
```

Description:

The *nop* instruction takes an instruction dispatch but performs no operation. It may be used for assembly patching in memory, or be used to insert an instruction delay in the program sequence.

The *nop* instruction takes no operands, no instruction modifier, no conditional modifier and no predication.

8.3.29 not – Logic Not

Opcode	Instruction	Description
4 (0x04)	not <dst> <src0>	Performing component-wise logic NOT of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] not[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] not[.<cmod>] (<exec_size>) reg reg
[(<pred>)] not[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
        if (WrEn.chan[n] == 1) {
                dst.chan[n] = !src0.chan[n]
        }
}
```

Description:

The *not* instruction performs logical NOT operation (or one’s compliment) of <src0> and storing the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

This instruction does not work with float type operands.

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not implicitely update accumulator register.

Accumulator is allowed to be the source of this instruction, but source modifier is not allowed for an accumulator source.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

8.3.30 or – Logic Or

Opcode	Instruction	Description
6 (0x06)	or <dst> <src0> <src1>	Performing component-wise logic OR of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] or[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] or[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] or[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] | src1.chan[n];
    }
}
```

Description:

The *or* instruction performs component-wise logic OR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.

Accumulator is allowed to be the source of this instruction, but source modifier is not allowed for an accumulator source.

This instruction does not implicitely update accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

8.3.31 pln – Plane

Opcode	Instruction	Description
90 (0x5A)	PLN <dst> <src0> <src1>	Computing a component-wise plane equation (w = p*u+q*v+r) of scalar (p, q, r) from <src0> and vector <src1> (and implied vector <src2>) and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] pln[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] pln[.<cmod>] (<exec_size>) reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
for (n = 0; n < exec_size; n++) {
     float dwP = src0.RegNum.SubRegNum[bits4:2]   // a dword aligned scalar
     float dwQ = src0.RegNum.(SubRegNum[bit4:2]|0x1)   // 2nd component
     float dwR = src0.RegNum.(SubRegNum[bit4:2]|0x3)   // 4th component
     src2 = src1.(RegNum + 1)           // Next GRF register
     if (WrEn.chan[n] == 1) {
          dst.chan[n] = dwP * src1.chan[n] + dwQ * src2.chan[n] + dwR
     }
}
```

Description:

The *pln* instruction computes a component-wise plane equation (*w = p*u+q*v+r* where *u/v/w* are vectors and *p/q/r* are scalars) of <src0> and <src1> and storing the results in <dst>. <src1> is the input vector *u*. The second input vector v is implied from <src1>, as the next adjacent GRF register. <src0> provides input scalars *p, q* and *r*, where *p* is the scalar value based on the region description of <src0> and *q* and *r* are the scalar values implied from <src0> region. Specifically, *q/r* is the second/fourth component of the 4-tuple (128-bit aligned) that *p* belongs to.

When *pln* instruction is used in SIMD16 form, the same input data channels *p/q/r* for <src0> are used for both SIMD8 instructions. However, as <src1> has two vectors *u/v*, where *v* is implied, the second SIMD8 instruction takes src1.(RegNum+2) as the second source operand.

Restrictions:

This is a specialized instruction that only support execution size of 8 or 16.

<src0> region must be a replicated scalar (with HorzStride = VertStride = 0).

<src0>, <src1>, <src2>, and <dst> may be float.

Source operands cannot be an accumulator register.

<src0> for *pln* instruction has to has .0 or .4 as the subregister number.

<src1> must be even register aligned.

8.3.32 rndd – Round Down

Opcode	Instruction	Description
69 (0x45)	rndd <dst> <src0>	Taking component-wise floating point downward rounding of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] rndd[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndd[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndd[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = floor(src0.chan[n]);
      }
}
```

Description:

The *rndd* instruction takes component-wise floating point downward rounding (to the integral float number closer to negative infinity) of <src0> and storing the rounded integral float results in <dst>. This is commonly referred to as the *floor()* function.

This instruction only applies to floating point source and destination operands.

Output data <dst> for floating point rounding-down follow rules in Table 8-7 (or Table 8-8), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-7. Floating point round-down in IEEE mode

<src0>	**–inf**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+inf**	**NaN**
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

Table 8-8. Floating point round-down in ALT mode

<src0>	**– fmax**	**–finite**	**–denorm**	**–0**	**+0**	**+denorm**	**+finite**	**+ fmax**	*******
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

*** Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.

8.3.33 rndu – Round Up

Opcode	Instruction	Description
68 (0x44)	rndu <dst> <src0>	Taking component-wise floating point upward rounding of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] rndu[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndu[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndu[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            if (src0.chan[n]-floor(src0.chan[n]) > 0.0f)
                  dst.chan[n] = floor(src0.chan[n]) + 1;
            else
                  dst.chan[n] = src0.chan[n];
      }
}
```

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of <src0>, commonly known as the *ceiling()* function.

This instruction only applies to floating point source and destination operands.

Output data <dst> for floating point rounding-up follow rules in **Error! Reference source not found.** (or **Error! Reference source not found.**), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-9. Floating point round-up in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.

Table 8-10. Floating point round-up in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	

Notes:

*	Result may be {–finite, –0}.
**	Result may be {+finite, +0}.
***	Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.
****	Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

8.3.34 rnde – Round to Even

Opcode	Instruction	Description
70 (0x46)	Rnde <dst> <src0>	Taking component-wise floating point round-to-even operations of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] rnde[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rnde[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rnde[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
     if (WrEn.chan[n] == 1) {
          dst.chan[n] = floor(src0.chan[n]);
          if (src0.chan[n]-floor(src0.chan[n]) > 0.5f) {
               dst.chan[n] = floor(src0.chan[n]) + 1;
          } else if (src0.chan[n]-floor(src0.chan[n]) < 0.5f) {
               dst.chan[n] = floor(src0.chan[n]);
          } else {
               if (dst.chan[n] is odd) {
                    dst.chan[n] = floor(src0.chan[n]) + 1;
               } else {
                    dst.chan[n] = floor(src0.chan[n]);
               }
          }
     }
}
```

Description:

The *rnde* instruction takes component-wise floating point round-to-even operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in <dst> to create the final round-to-even values to emulate the round-to-even operation, commonly known as the *round()* function. The final results are the one of the two integral float values that is nearer to the input values. If the neither possibility is nearer, the even alternative is chosen.

Output data <dst> for floating point rounding-to-even follow rules in Table 8-11 (or Table 8-12), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-11. Floating point round-to-even in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

*** Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise.

Table 8-12. Floating point round-to-even in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

*** Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise.

**** Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

8.3.35 rndz – Round to Zero

Opcode	Instruction	Description
71 (0x47)	rndz <dst> <src0>	Taking component-wise floating point round-to-zero operations of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT]	[FLT]

Format:

```
[(<pred>)] rndz[.<cmod>] (<exec_size>) <dst> <src0>
```

Syntax:

```
[(<pred>)] rndz[.<cmod>] (<exec_size>) reg reg
[(<pred>)] rndz[.<cmod>] (<exec_size>) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
        if (WrEn.chan[n] == 1) {
                dst.chan[n] = floor(src0.chan[n]);
                if (abs(src0.chan[n]) < abs(dst.chan[n])) {
                        dst.chan[n] = floor(src0.chan[n]) + 1;
                } else {
                        dst.chan[n] = floor(src0.chan[n]);
                }
        }
}
```

Description:

The *rndz* instruction takes component-wise floating point round-to-zero operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in <dst> to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the *truncate()* function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.

Output data <dst> for floating point rounding-to-zero follow rules in Table 8-13 (or Table 8-14), if the current floating point mode is IEEE mode (or ALT mode).

Table 8-13. Floating point round-to-zero in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

*** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.

Table 8-14. Floating point round-to-zero in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	

Notes:

* Result may be {–finite, –0}.

** Result may be {+finite, +0}.

*** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.

**** Result is undefined if <src0> is {–inf, +inf, NaN}.

Restrictions:

8.3.36 sad2 – Sum of Absolute Difference 2

Opcode	Instruction	Description
80 (0x50)	sad2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, and storing the scalar result to the first channel per 2-tuple in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] sad2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
     if (WrEn.chan[n] == 1) {
          dst.chan[n] = abs(src0.chan[n] - src1.chan[n])
               + abs(src0.chan[n+1] - src1.chan[n+1]);
}
```

Description:

The *sad2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1> and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision.

Destination register must have a stride of 2 bytes and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values.

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.

8.3.37 sada2 – Sum of Absolute Difference Accumulate 2

Opcode	Instruction	Description
81 (0x51)	sada2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, added to that from the accumulator, and storing the scalar result to the first channel per 2-tuple in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] sada2[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] sada2[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] sada2[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
    uwTmp = abs(src0.channel[n] - src1.channel[n])
        + abs(src0.channel[n+1] - src1.channel[n+1])
    if (WrEn.channel[n] == 1) {
        dst.channel[n] = uwTmp + acc[n]
    }
}
```

Description:

The *sada2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1>, adds the intermediate result with the accumulator value corresponding to the first channel, and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision. Higher precision (guide bits) stored in the accumulator allows multiple rounds (64 rounds) of sada2 instructions to be issued back to back without overflow the accumulator.

Destination register must have a stride of 2 words and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.

8.3.38 sel – Select

Opcode	Instruction	Description
2 (0x02)	(pred) sel <dst> <src0> <src1>	Component-wise selective move from <src0> or <src1> to <dst> based on predication or cmod result. . The sel instruction can not use accumulator as destination

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

```
(<pred>) sel (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
(<pred>) sel (<exec_size>) reg reg reg
(<pred>) sel (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn, NoPMask);

if (cmod == “0000”) { // no CMod
    Evaluate(PMask);
    for (n = 0; n < exec_size; n++) {
        if (WrEn.channel[n] == 1) {
            if (PMask.channel[n] == 1) {
                dst.channel[n] = src0.channel[n]
            } else {
                dst.channel[n] = src1.channel[n]
            }
        }
    }
    }
    else { // with CMod
         Evaluate(CMod);
         for (n = 0; n < exec_size; n++) {
              if (WrEn.channel[n] == 1) {
                   if (CMod.channel[n] == 1) {
                        dst.channel[n] = src0.channel[n]
                   } else {
                        dst.channel[n] = src1.channel[n]
                   }
              }
         }
    }
```

Description:

The *sel* instruction selectively moves the components in <src0> or <src1> into the channels of <dst> based on the predication. On a channel by channel basis, if the channel condition is true, data in <src0> is moved into <dst>; Otherwise, data in <src1> is moved into <dst>.

As the predication is used to select the two sources, it is not included in the evaluation of WrEn. <pred> is mandatory if <cmod> is "0000". If it is <omitted> and <cmod> is "0000", the results are unpredictable.

In case of <cmod> not equal to "0000", a compare will be performed and the result flag will be used for the *sel* instruction. <cmod> .ge and .l follow the CMPN rules, and all other <cmod> follow the CMP rules. <pred> is not allowed in this mode.

sel instruction with <cmod> .l should be used to emulate MIN instruction.

sel instruction with <cmod> .ge should be used to emulate MAX instruciton.

If any of the source of the *sel* instruction is NaN, the non-NaN source will be the result, in case both sources are NaN, the result will also be NaN. This only applies to *sel* instruction with .l and .ge conditional modifier. For the other conditional modifiers, src1 will be always be selected if either or both sources is/are NaN.

sel instruction with <cmod> will flush denorm to zero; *sel* instruction without <cmod> will retain denorm.

Format conversion is not allowed.

sel instruction can not use accumulator source.

Restrictions:

Destination channels cannot be on odd-byte sub-register locations. In other words, when destination is of byte type, destination horizontal stride cannot be 1. If destination horizontal stride is not 1, destination register region origin cannot be on an odd byte location. This is because that the conditional flag for execution channels that have minimal granularity of word are used by this instruction.

The sel instruction can not use accumulator as destination

8.3.39 send – Send Message

Table 8-17. Sideband Signals Associated with Each Message Sent to the Shared Function

Signal	Bits	Source
EOT	1	End of Thread: Sourced from the EOT bit in *send* instruction word
SFID	3	Shared Function Identifier: Sourced from the target function ID field in <ex_desc> of *send*
MLEN	4	Message Length: Sourced from the message length field in <desc> of *send*
RLEN	5	Response Length: Sourced from the response length field in <desc> of *send*
FC	19	Function Control: Sourced from the function control field in <desc> of *send*
REG	7	Destination Register: Sourced from the 256-bit register aligned register number of the <dest> field of *send*
CE	16	Channel Enable: Sourced from the write enable of *send*
CLEAR	1	Destination Register Clear: Source from the Destination Dependency Control field (inverse of NoDDClr) in *send* instruction word
FFID	4	Fixed Function Identifier: Sourced from the Fixed Function ID field in *sr0*
EUID	4	Execution Unit Identifier: Sourced from the EUID field in *sr0*
TID	2	Thread Identifier: Sourced from the TID field in *sr0*

Restrictions:

Software must obey the following rules in signaling the end of thread using the *send* instruction:

- The posted destination operand must be null.
  - No acknowledgement is allowed for the *send* instruction that signifies the end of thread. This is to avoid deadlock as the EU is expecting to free up the terminated thread’s resource.
- A thread must terminate with a *send* instruction with message to a shared function on the output message bus; therefore, it cannot terminate with a *send* instruction with message to the following shared functions: Sampler unit, NULL function
  - For example, a thread may terminate with a URB write message or a render cache write message.
- A root thread originated from the media (generic) pipeline must terminate with a *send* instruction with message to the Thread Spawner unit. A child thread should also terminate with a send to TS. Please refer to the Media Chapter for more detailed description.

The *send* instruction can not update accumulator registers.

Saturate is not supported for *send* instruction.

ThreadCtrl are not supported for *send* instruction.

The MRF register must be writen into for every *send* instruction, using the same MRF register for multiple *send* without updating it in between is not allowed.

Table 8-17. Sideband Signals Associated with Each Message Sent to the Shared Function

Signal	Bits	Source
EOT	1	End of Thread: Sourced from the EOT bit in *send* instruction word
SFID	3	Shared Function Identifier: Sourced from the target function ID field in <ex_desc> of *send*
MLEN	4	Message Length: Sourced from the message length field in <desc> of *send*
RLEN	5	Response Length: Sourced from the response length field in <desc> of *send*
FC	19	Function Control: Sourced from the function control field in <desc> of *send*
REG	7	Destination Register: Sourced from the 256-bit register aligned register number of the <dest> field of *send*
CE	16	Channel Enable: Sourced from the write enable of *send*
CLEAR	1	Destination Register Clear: Source from the Destination Dependency Control field (inverse of NoDDClr) in *send* instruction word
FFID	4	Fixed Function Identifier: Sourced from the Fixed Function ID field in *sr0*
EUID	4	Execution Unit Identifier: Sourced from the EUID field in *sr0*
TID	2	Thread Identifier: Sourced from the TID field in *sr0*

Restrictions:

Software must obey the following rules in signaling the end of thread using the *send* instruction:

- The posted destination operand must be null.
  - No acknowledgement is allowed for the *send* instruction that signifies the end of thread. This is to avoid deadlock as the EU is expecting to free up the terminated thread's resource.
- A thread must terminate with a *send* instruction with message to a shared function on the output message bus; therefore, it cannot terminate with a *send* instruction with message to the following shared functions: Sampler unit, NULL function
  - For example, a thread may terminate with a URB write message or a render cache write message.
- A root thread originated from the media (generic) pipeline must terminate with a *send* instruction with message to the Thread Spawner unit. A child thread should also terminate with a send to TS. Please refer to the Media Chapter for more detailed description.

The *send* instruction can not update accumulator registers.

Saturate is not supported for *send* instruction.

ThreadCtrl are not supported for *send* instruction.

The MRF register must be writen into for every *send* instruction, using the same MRF register for multiple *send* without updating it in between is not allowed.

8.3.40 sendc – Conditional Send Message

Opcode	Instruction	Description
49 (0x31)	sendc <dest> <src> <desc> <ex_dest>	Wait for the dependencies in the TDR register cleared, then send a message stored in MRF starting at <str> to a shared function identified by <ex_desc> along with control from <desc> with a GRF writeback location at <dest>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•					[FLT] [INT]

Format:

```
[(<pred>)] sendc (<exec_size>) <dest> <src> <desc> <exdesc>
```

Syntax:

```
[(<pred>)] sendc (<exec_size>) reg reg reg32a imm4
[(<pred>)] sendc (<exec_size>) reg reg imm32 imm4
```

Pseudocode:

```
if ((TDR[7]... || TDR[2] || TDR[1] || TDR[0]) == TRUE) {
    wait;
}
Evaluate(WrEn);
<MsgChEnable> = WrEn;
<SourceReg> = <src>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>, <desc>,
<ex_dest>);
```

Description:

The *sendc* instruction has the same behavior as the *send* instruction except the following.

The *sendc* instruction first check the dependent threads inside the Thread Dependency Register, there are up to 4 dependent threads in the TDR register. The *sendc* instruction will be executed only when all the dependent threads in the TDR register are retired.

Restrictions:

The *sendc* instruction has the same restrictions as the *send* instruction.

8.3.41 shl – Shift Left

Opcode	Instruction	Description
9 (0x09)	shl <dst> <src0> <src1>	Performing component-wise logic left shift of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] shl[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = src0.chan[n] << src1.chan[n]
      }
}
```

Description:

The *shl* instruction performs component-wise logical left shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1>, where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored. The results are NOT stored in the accumulator register.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel. <src0> and <dst> can be signed or unsigned integers and can be of different types. This instruction does not work with float type operands. Saturation modifier is only allowed when this instruction is in packed-word mode. Hardware detects overflow properly and use it to perform saturation operation on the output, as long as the shifted result is within 33 bits. Otherwise, the result is undefined.

Results of saturation in packed-dword mode are unpredicable.

Restrictions:

This instruction does not work with float type operands.

8.3.42 shr – Shift Right

Opcode	Instruction	Description
8 (0x08)	shr <dst> <src0> <src1>	Performing component-wise logic right shift of <src0> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	[INT]	[INT]

Format:

```
[(<pred>)] shr[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] shr[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] shr[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = src0.chan[n] >> src1.chan[n]
      }
}
```

Description:

The *shr* instruction performs component-wise logical right shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1> where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction only takes on unsigned sources. When <src0> contains unsigned integers, no source modifier is allowed. <src0> is only allowed to be signed integer if source modifier (abs) is used. *Note: for unsigned sources, the behavior of shr and asr are effectively the same.*

Restrictions:

This instruction does not work with float type operands.

8.3.43 wait – Wait Notification

Opcode	Instruction	Description
48 (0x30)	wait <nreg>	Waiting for notification on the notification register <nreg>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

```
wait (<exec_size>) <nreg>
```

Syntax:

```
wait (1) n#
```

Pseudocode:

```
n/a
```

Description:

The *wait* instruction evaluates the value of the notification count register <nreg>. If <nreg> is zero, the execution of the thread is stalled and the thread is put in 'wait_for_notification' state. If <nreg> is not zero (i.e., one or more notifications have been received), <nreg> is decremented by one and the thread continues executing on the next instruction. If a thread is in the 'wait_for_notification' state, when a notification arrives, the notification count register is incremented by one. As the notification count register becomes non-zero, the thread wakes up to continue execution and at the same time the notification register is decremented by one. If there was only one notification arrived, the notification register value becomes zero. However, during the above mentioned time period, it is possible that more notifications may arrive, making the notification register non-zero again.

When multiple notifications are received, software must use 'wait' instruction to decrement notification count register for each of the notifications.

Notification register n0:ud is for thread to thread communication (through message gateway shared function) and n1:ud for host to thread communication (through MMIO registers). See Message Gateway chapter for thread-thread communication.

Restrictions:

Only one source operand.

<src0> and <dst> must be n0 or n1pr n2, <src1> must be *null*.

Execution size must be 1 as the notification registers are scalar.

Predication is not allowed.

Implementation restriction: Two back-to-back *wait* instructions in a program (without any instruction in between) are not allowed. As a minimal, a *nop* has to be inserted between two *wait* instructions.

```
            }
            else { // with embedded compare
                  if (cmod.channel[n] == 1) {
                        PcIP[n] = IP + <JIP>;
                  }
                  else {
                        PcIP[n] = IP + 1;
                  }
            }
      }
}
if (<cmod> == 0) { // no embedded compare
      if (|PMask == 1) { // any enabled channel true
            Jump(IP + <JIP>);
      }
}
else { // with embedded compare
      if (|cmod == 1) { // any enabled channel true
            Jump(IP + <JIP>);
      }
}
```

Description:

The *while* instruction marks the end of a do-while block. The instruction first evaluates the loop termination condition for each channel based the current channel enables the predication flag specified in the instruction. If any channel has not terminated, a branch is taken to a destination address based specified in the instruction, and the loop continued for those channels. Otherwise, execution continue down to the next instruction.

The following table describes the 16-bit jump target offset <JIP>. <JIP> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction with the *do* label of the do-while block of code. It should be a negative number for the backward referencing. In GEN binary, <JIP> is at location <dst> and must be of type W (signed word integer).

Bit	Description
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

If SPF is ON, none of the PcIP is updated.

Restrictions:

To use embedded compares, the predicate control field for this instruction must be zero, and the conditional modifier field must be none zero. The destination must follow the rules below:

1. Must has the same element size as source0
2. Must have horizontal stride of 1

To use predicated *if* instruction, the conditional modifier field must be zero.

8.3.44 while – While [DevGT+]

Opcode	Instruction	Description
39 (0x27)	while if <dst> <src0> <src1> <JIP>	Marking the end of a do-while block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•		•			

Format:

```
[(<pred>)] while (<exec_size>) null null null <JIP>
           while (<exec_size>) null <src0> <src1> <JIP>
```

Syntax:

```
[(<pred>)] while (<exec_size>) null null null imm16
           while (<exec_size>) null reg reg imm16
           while (<exec_size>) null reg imm32 imm16
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.channel[n] == 1) {
            if (<cmod> == 0) { // no embedded compare
                  if (PMask.channel[n] == 1) {
                        PcIP[n] = IP + <JIP>;
                  }
            }
            else { // with embedded compare
                  if (cmod.channel[n] == 1) {
                        PcIP[n] = IP + <JIP>;
                  }
            }
      }
}
if (<cmod> == 0) { // no embedded compare
      if (|PMask == 1) { // any enabled channel true
            Jump(IP + <JIP>);
```

```
        }
}
else { // with embedded compare
        if (|cmod == 1) { // any enabled channel true
                Jump(IP + <JIP>);
        }
}
```

Description:

The *while* instruction marks the end of a do-while block. The instruction first evaluates the loop termination condition for each channel based the current channel enables the predication flag specified in the instruction. If any channel has not terminated, a branch is taken to a destination address based specified in the instruction, and the loop continued for those channels. Otherwise, execution continue down to the next instruction.

The following table describes the 16-bit jump target offset <JIP>. <JIP> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction with the *do* label of the do-while block of code. It should be a negative number for the backward referencing. In GEN binary, <JIP> is at location <dst> and must be of type W (signed word integer).

Bit	Description
15:0	**JIP (Jump Target Offset).** This field specifies the jump distance in number of 64bits data chunks if a jump is taken for the instruction. Format = S15. Signed integer in 2's compliment

need to add detail for SPF.

Restrictions:

Instruction compression is not allowed.

8.3.45 xor – Logic Xor

Opcode	Instruction	Description
7 (0x07)	xor <dst> <src0> <src1>	Performing component-wise logic XOR of <src0> and <src1> and storing the results in <dst>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	[INT]	[INT]

Format:

```
[(<pred>)] xor[.<cmod>] (<exec_size>) <dst> <src0> <src1>
```

Syntax:

```
[(<pred>)] xor[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] xor[.<cmod>] (<exec_size>) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
      if (WrEn.chan[n] == 1) {
            dst.chan[n] = src0.chan[n] ^ src1.chan[n];
      }
}
```

Description:

The *xor* instruction performs component-wise logic XOR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.

The results are NOT stored in the accumulator register.

Accumulator is allowed to be the source of this instruction, but source modifier is not allowed for an accumulator source.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have satuation instruction modifier.

9. EU Programming Guide

9.1 Assembler Pragmas

9.1.1 Declarations

A register or a register region can be declared as a symbol using the following form

.**declare** <symbol> **Base**=RegFile RegBase {.SubRegBase} **ElementSize**=ElementSize {**SrcRegion**=DefaultSrcRegion} {**DstRegion**=DefaultDstRegion} {**Type**=DefaultType}

The register file, the base of the register origin and the element size (in unit of bytes) are the mandatory parameters for a declared register region. Optionally, the base of the sub-register address, the default source region, the default destination region and the default type can be provided in the declaration for the symbol.

For immediate register addressing mode, the declared symbol can be used in the following Cartesian form

<symbol>(RegOff, SubRegOff) 🡐 RegNum = *RegBase* + **RegOff***;* SubRegNum = *SubRegBase* + **SubRegOff**

or in the following simplified row-aligned form

<symbol>(RegOff) 🡐 RegNum = *RegBase* + **RegOff;** SubRegNum = *SubRegBase*

For register-indirect-register-addressing mode, the declared symbol can be used to provide immediate address term in the following Cartesian form

<symbol>[IdxReg, RegOff, SubRegOff] 🡐 RegNum (byte-aligned) = [**IdxReg**] +(*RegBase* + **RegOff**)*32 + (*SubRegBase* + **SubRegOff**)**ElementSize*

or in the following simplified row-aligned form

<symbol>[IdxReg, RegOff] 🡐 RegNum (byte-aligned) = [**IdxReg**] +(*RegBase* + **RegOff**)*32

or in the form without the immediate address term

<symbol>[IdxReg] 🡐 RegNum (byte-aligned) = [**IdxReg**] + *RegBase*

9.1.2 Defaults and Defines

The default execution size is set according to the destination register type as the following

Destination Register Type	Default Execution Size
UB | B	(16)
UW | W	(16)
F | UD | D	(8)

The default execution size can be overwritten globally for all instructions using

.default_execution_size *(Execution_Size)*

or be set according the **destination** register type using

.default_execution_size_*Type* *(Execution_Size)*

The default register type can be set for all register files using

.default_register_type *Type*

or be set per register file using

.default_register_type_*RegFile* *Type*

The default **source** register region for all symbols can be set using

.default_source_register_region *<VirtStride; Width, HorzStride>*

or be set per register type using

.default_source_register_region_*type* *<VirtStride; Width, HorzStride>*

The default **destination** register region for all symbols can be set using

.default_destination_register_region *< HorzStride>*

or be set per register type using

.default_destination_register_region_*type* *< HorzStride>*

Finally, the precompiler supports the string replacement statement of .define in the following form

.define <symbol> Expression

Notes:

- **.declare** does not support nesting. In other words, each symbol in .declare must be self defined. This would allow the pre-processor to expand all symbols in one pass.
- **.define** does support nesting. Only string substitution is supported (currently).
- White space within square, angle and round brackets are allowed for easy source code alignment.

9.1.3 Example Pragma Usages

Example 1: Declaration for 8x4=32-Byte Regions:

The following symbol *Block* can be used to address any 8x4 byte region within the Cartisian system of a 16x8 byte GRF register area starting from r0.

```
                      // 32x4 Byte Array
Declaration           .declare Block Base=r0 ElementSize=1 Region=<32;8,1> Type=b
                      mov(32)    ?:b     r0.16<32;8,1>:b       // r0 xxxxxxxxxxxxxxxxooooooooxxxxxxxx
Fully-Expressed                                                // r1 xxxxxxxxxxxxxxxxooooooooxxxxxxxx
Instr                                                          // r2 xxxxxxxxxxxxxxxxooooooooxxxxxxxx
                                                               // r3 xxxxxxxxxxxxxxxxooooooooxxxxxxxx

                      Mov    ?:b    Block(0,16)                // (0,16): RegNum=0, SubRegNum=16
Short-handed Instr
```

Example 2: Declaration for 8x1 Float Regions:
The following symbol *Trans* can be used to address any 8x1 float region within the Cartisian system of a 8x4 float GRF register area starting from r5.

```
                      // 8x4 float Array starting at r5
Declaration           .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f
                      mov(8)     ?:f     r6.0<0;8,1>:f         // 2nd 16x1 Row of Trans. Matrix
Fully-Expressed                                                // r5 FFFFFFFF
Instr                                                          // r6 00000000
                                                               // r7 FFFFFFFF
                                                               // r8 FFFFFFFF

                      mov        ?:f     Trans(1)              // RegNum = 5+1 = 6
Short-handed Instr
```

Example 3: Declaration for 8x1 Float Regions with 1x1 Indirect Addressing:
Trans region defined (same as in the previous example) is used in conjunction with the address register.

```
                      //8x4 float data array and 16x1 word address array
Declaration           .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f
                      mov(8)     ?:f     r[a0.0,224]<0;8,1>:f
Fully-Expressed
Instr
                      mov        ?:f     Trans[a0.0,2]         // [a0.0 + 5*32 + 2*32]
Short-handed Instr
```

Example 4: Declaration with VxH Indirect Addressing:
The VxH register-indirect-register-addressing for Trans can be provided in the following short-hand form.

Declaration	`//8x4 float data array and word indices`
	`.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`

Fully-Expressed Instr	`mov(8)     ?:f     r[a0.0,224]<1,0>:f`
Short-handed Instr	`mov        ?:f     Trans[a0.0,2]<1,0>     // [a0.0+224] [a0.1+224] … [a0.7+224]`

Example 5: Declaration with Vx1 Indirect Addressing:

```
As width (4) is smaller than the execution region size (8), multiple indexed registers are used.
```

Declaration	`//8x4 float data array and word address array`
	`.declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f`

Fully-Expressed Instr	`mov(8)     ?:f     r[a0.0,244]<4,1>:f`
Short-handed Instr	`mov        ?:f     Trans[a0.0,2]<4,1>     // [a0.0+224] [a0.1+224]`

9.1.4 Assembly Programming Guideline

The following program skeleton illustrates the basic structure of a typical assembly program.

```
//  single line comment

/*
       block comment
*/

<preproc_directive>    // macros, include, etc.  Are global – handled by the pre-processor
<preproc_directive>    // applies to all code that follows in sequence

// ------------ some kernel ---------------------------
.kernel <kernel_name_string>  // [REQUIRED]
                              // ------- Register requirements --------
 .reg_count_total     <uint>  //  [REQUIRED] a more direct way to specify the exact parameters
require
 .reg_count_payload   <uint>  // [REQUIRED] rather than to have to indirectly do that by adding the
                              //   the payload and temps together to get the total (as is the case
now)
                              // Note: no more “reg-count-temp”

                              // -------------- Defaults ---------------
 <default…>                   // these should be specified per-kernel and have only kernel-scope
 <default…>                   // Same defaults as those already defined in the ISA doc, but just
 <default…>                   // moved within the kernel to make each kernel completely self-
sufficient
                              // and not impacted defaults of earlier kernels

                              // --------- Memory Requirements ---------
                              // [optional] memory block info (just a placeholder for now...)
 <MBDa>                       //     memory block descriptor a (TBD)
 <MBDb>                       //     memory block descriptor b (TBD)
 <MBDc>                       //     memory block descriptor c (TBD)
 <MBDd>                       //     memory block descriptor d (TBD)

                              // ---------------- Code  ----------------
 .code                        // [REQUIRED]
        <instruction>
        <instruction>
        <instruction>
    <LabelLine>               // labels are code-block scope
        <instruction>
        <instruction>

 .end_code                    // [REQUIRED]

.end_kernel                   // [REQUIRED]

// --------- next kernel -------------

// --------- next kernel -------------

// ...
```

9.2 Usage Examples

9.2.1 Vector Immediate

The immediate form of vector allows a constant vector to be in-lined in the instruction stream. An immediate vector is denoted by type v as *imm32:v*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. Each 4-bit subfield contains a signed integer value in 2's compliment form. Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following figure.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
V7	V6	V5	V4	V3	V2	V1	V0

9.2.1.1 Supporting OpenGL Vertex Shader Instruction SWZ

When an OpenGL Vertex Shader program is converted to run on GEN in Vertex Pair, i.e., two 4-wide vectors in parallel, the special OpenGL Shader instruction SWZ (Swizzle) needs to be emulated. OpenGL SWZ instruction uses an extended swizzle control field that, in addition to the 4-wide full swizzle control, also includes constant 0 and 1 replacement as well as per channel sign reversal. The later two are not supported by the GEN native instruction. The vector immediate can significantly reduce the overhead of emulating such OpenGL instruction.

Consider an OpenGL Shader instruction in the form of

```
SWZ     r1          r0.0-zx-1           // Expected results: r1.x = 0; r1.y = -r0.z; r1.z = r0.x; r1.w = -1
```

It can be emulated by the following three GEN instructions.

```
mul        (8)        r1.0<1>:f          r0.xzxz            0x1F111F11:v    // Constant vector of (1 -1 1 1 1 -1 1 1)

mov (1)  f0.0                   8b'10011001                          // Set flag & masked out channels y and z

(f0.0)mov(8) r1.0<1>:f     0x000F000F:v                         // Constant vector of (0 0 0 -1 0 0 0 -1)
```

In case that only 0, 1, -1 channel replacement is used and there is no signed swizzle, it may be emulated in two GEN instructions. This is illustrated by the following example:

OpenGL:

```
SWZ     r1          r0.0zx-1 // Expected results: r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1
```

GEN:

```
mov (1)  f0.0        8b'01100110                        // Set flag and masked out channels x and w

(f0.0)sel (8) r1.0<1>:f r0.yzxy        0x000F000F:v     // Constant vector of (0 0 0 -1 0 0 0 -1)
```

9.2.2 Destination Mask for DP4 and Destination Dependency Control

The following example demonstrates the use of destination mask mode of floating point dot-product instruction as well as the use of destination dependency control to improve performance (i.e., avoiding unnecessary thread switch due to possible false dependencies).

Consider a generic Vertex Shader macro of matrix-vector product that is implemented on GEN in the pair of 4-component vector mode. The equivalent Shader instructions are as the following.

```
dp4 r5.x r0 r4

dp4 r5.y r1 r4

dp4 r5.z r2 r4

dp4 r5.w r3 r4
```

With destination dependency control, the GEN instructions are as the following. The first instruction in the sequence checks for the destination dependency, but does not clear the dependency bit. The subsequent two instructions would do neither of them. The last instruction avoids checking the destination dependency, but at completion, it clears the destination scoreboard. It ensures that the content of the destination register is coherent, if any of the following instructions uses the same register as source.

```
dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f     {NoDDClr}

dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f     {NoDDClr, NoDDChk}

dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f     {NoDDClr, NoDDChk}

dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f     {NoDDChk}
```

Just as a comparison, IF GEN DP4 implies reduction at the destination; additional shifted moves are required to achieve the same results. The corresponding codes are as the following. The lower performance due to the additional three move instruction as well as added back-to-back dependencies shows that why we choose to implement the destination channel replication for floating point DP4.

```
dp4 (8) r5.0<1>.y:f     r1.0<4;4,1>:f     r4.0<4;4,1>:f

mov (1) r5.1<1>:f       r8.0<1;1,1>:f

dp4 (8) r5.0<1>.z:f     r2.0<4;4,1>:f     r4.0<4;4,1>:f

mov (1) r5.2<1>:f       r8.0<1;1,1>:f

dp4 (8) r5.0<1>.w:f     r3.0<4;4,1>:f     r4.0<4;4,1>:f

mov (1) r5.3<1>:f       r8.0<1;1,1>:f

dp4 (8) r5.0<1>.x:f     r0.0<4;4,1>:f     r4.0<4;4,1>:f
```

9.2.3 Null Register as the Destination

Null register can be used as the destination for most of the instructions. Here are some example usages.

- Null as destination for regular ALU instructions: As all ALU instructions can be configured to update the flag registers using the conditional modifiers, it is not necessary to have a destination register if the programmer only cares about the conditionals of the operation. In that case, a null in the destination operand field saves register space as well as one less dependency checking.

- Null as the destination for SEND/STOR instructions: for the send instruction that only send messages out to an external unit and does not require any return data or feedback, a null in the destination register field signifies the case.

    - One extension of such case is that even though the operation does not have any return values, a return phase with no payload but simply updating the scoreboard flag for a non-null register can provide a signaling mechanism between the thread and the target external unit. One application of this usage is to allow software to manage the coherency of shared memory resources such like the many caches in the system (particularly, valuable for read/write caches). This is not currently the POR for GEN though.

9.2.4 Use of LINE Instruction

LINE instruction is specifically designed to speed up floating point vector/matrix computation when a program operates in channel serial.

The following example demonstrates how to use LINE instruction to compute Line Equations for Pixel Shader. In this example, 2 sets of (Cx#, Cy#, Don't Care, C0#) 4-tuple coefficient vectors are stored in registers R1.

R1: Cx0 Cy0 DC Co0 Cx1 Cy1 DC Co1

8 sets of coordinate 2-D vectors (X, Y) are stored in R2 and R3 in the channel serial mode as

R2: X0 X1 … X7
R3: Y0 Y1 … Y7

The objective is to compute the following two line equations for each set of 2D coordinate and store the results in R4 and R5 as

R4: (X0*Cx0 + Y0*Cy0+Co0) ... (X7*Cx0 + Y7*Cy0+Co0)

R5: (X0*Cx1 + Y0*Cy1+Co1) ... (X7*Cx1 + Y7*Cy1+Co1)

Example 9-1. LINE Equations

```
//-------------------------------------------------------------------
// Example compute LINE equation in channel serial scenario
//-------------------------------------------------------------------

line (8) acc:f   r1<0;1,0>:f   r2<0;8,1>:f        // does acc = X# * Cx0 + Co0
mac  (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f   // does r4.# = Y# * Cy0 + acc.#

line (8) acc:f   r1<0;1,0>:f   r2<0;8,1>:f        // does acc = X# * Cx0 + Co0
mac  (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f   // does r4.# = Y# * Cy0 + acc.#
```

The next example is to compute homogeneous dot product for OpenGL pixel shader running in Channel Serial. In this example, an original OpenGL PS instruction is like

```
dph R2.x R0 R1
```

With register remapping, we can store the input coefficient vector R0 in original format in r0, but 8 sets of input coordinate vectors in channel serial format in r2, r3, r4 and r5, and the destination R2.x component in r6.

```
r0: Cx0 Cy0 Cz0 Co0 DC DC DC DC
r2: X0 X1 … X7
r3: Y0 Y1 … Y7
r4: Z0 Z1 … Z7
r5: W0 W1 … W7
```

The objective is to compute the following DPH equations and store the results in r6 as

```
R6: (X0*Cx0+Y0*Cy0+Z0*Cz0+Co0) ... (X7*Cx0+Y7*Cy0+Z7*Cz0+Co0)
```

Example 9-2. Homogeneous Dot Product in Channel Serial

```
//-------------------------------------------------------------------

// Example compute homogeneous dot product in channel serial scenario

//-------------------------------------------------------------------


line (8) acc:f   r0<0;1,0>:f   r2<0;8,1>:f              // does acc = X# * Cx0 + Co0

mac  (8) acc:f   r0.1<0;1,0>:f r3<0;8,1>:f              // does acc.# = Y# * Cy0 + acc.#

mac  (8) r6<1>:f r0.2<0;1,0>:f r4<0;8,1>:f              // does r6.# = Z# * Cz0 + acc.#
```

9.2.5 Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of Shader instructions.

9.2.5.1 Channel Enables for Extended Math Unit

The following example demonstrates how to use the SEND instruction to get service from the Extended Math unit.

Let’s consider COS instruction in the following form

[([!]p0.{select|any|all})] cos[_sat] dest[.mask], [-]src0[_abs][.swizzle]

For a SIMD4x2 VS implementation with the following register mappings

p0 → f0.0

src0 → r0

dest → r1

The equivalent GEN instruction is as the following

[([!]f0.0.{select|any4h|all4h})] SEND (8) r1[.mask]:f m0 [-][(abs)]r0[.swizzle]:f MATHBOX|COS[|SAT]

If the source swizzle is replication, the message description field can be modified to MATHBOX|COS|SCALAR to take advantage of the fast mode (scalar mode) supported by the Extended Math. The implied move of the SEND instruction is equivalent to the following instruction:

MOV (8) m0[.mask]:f [-][(abs)]r0.0[.swizzle]:f {NoMask}

For a SIMD16 PS implementation, the register mappings are as the followings

p0 → f0…f3 // in order of R, G, B, A

src0 → r0,r1; r2,r3; r4,r5; r6,r7

dest → r8,r9; r10,r11; r12,r13; r14,r15

```
send (8) r8:f m0   -(abs)r2:f MATHBOX|COS
send (8) r9:f m1   -(abs)r3:f MATHBOX|COS {SecHalf}     // use the second half of 8 flag bits
mov (16) r10:f     r8:f                                 // All destination color chan’s are same
mov (16) r12:f     r8:f                                 // MOV is faster than most MathBox func’s
mov (16) r14:f     r8:f                                 // These MOV’s are compressed instructions
```

Notice that instead of issuing Extended Math messages with the same input data, destination color channel replication is performed by the MOV instructions. This is faster for the thread for most cases as many Extended Math functions consume multiple cycles. This also conserves message bus bandwidth as well as the usage of the shared resource – Extended Math. The destination mask in the instruction indicates which of the r8 to r15 registers are updated. If the source swizzle is not replication, there will be 8 SEND instructions.

With predication on, if the predication modifier is p0.select, translation is to take the selected flag register f#. The other predication modifiers '.any' and '.all' are translated into '.any4v' and '.all4v', respectively. Notice that with predication on, it is not required to run all 4 pixels in a subspan in the same way, so no need to enforce .any4h/.any4v. The following example shows the instruction with predication (but without .select modifier).

```
(f0[.any4v|.all4v]) send (8) r8:f m0  -(abs)r2:f MATHBOX|COS

(f0[.any4v|.all4v]) send (8) r9:f m1  -(abs)r3:f MATHBOX|COS {SecHalf}

(f1[.any4v|.all4v]) mov (16) r10:f   r8:f            // All destination color chan's are same

(f2[.any4v|.all4v]) mov (16) r12:f   r8:f            // MOV is faster than most MathBox func's

(f3[.any4v|.all4v]) mov (16) r14:f   r8:f            // These MOV's are compressed instructions
```

The same instructions works also for predication with select component modifier. We simply replace f0 to f3 above by the selected flag register, say, f1. The modifier of any4h/all4v would also work.

9.2.5.2 Channel Enables for Scratch Memory

The following example demonstrates how to use the SEND instruction to get service from the Data Port for scratch memory access.

Let's consider general instruction that uses scratch memory as a source operand

```
[([!]p0.{select|any|all})] add dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

For a SIMD4x2 VS implementation with the following register mappings

p0 → f0

src0 → r0

src1 → s2 / r10

dest → r1

In this example, the scratch memory offset is provided by an immediate and a GRF register r10 is used as the intermediate GRF location for spill/fill of scratch buffer accesses. This arithmetic instruction is converted into a Data Port read followed by an arithmetic instruction.

```
mov (8) r3:d r0:d {NoMask}          // move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}        // s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask} // s2 for vertex 1

send (8) r10 m0 r3 DATAPORT|RC|READ_SIMD2

[([!]f0.{sel|any4h|all4h})] add (8) r1[.mask]:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r10[.swizzle]:f
```

So if scratch register is the source, there is no need to use the channel enable side band. This is also true for channel-serial PS cases.

Now, let's consider the case when a scratch register is the destination of an instruction.

p0 → f0

src0 → r0

src1 → r1

dest → s2 / r10

We have

```
add (8) m1:f [-][(abs)]r0[.swizzle]:f [-][(abs)]r1[.swizzle]:f

mov (8) r3:d r0:d {NoMask}          // move scratch base address to be assembled with offset values

mov (1) r3.0:d 2*32 {NoMask}        // s2 for vertex 0

mov (1) r3.1:d 2*32+16 {NoMask} // s2 for vertex 1
    [([!]f0.{sel|any4h|all4h})] send (8) null[.mask] m0 r3 DATAPORT|RC|WRITE_SIMD2
```

Notice that with a null as the posted destination register, we are able to transfer the [.mask] over the message channel enables. In many cases for scratch memory assess, a write-with-commit is required, therefore, the posted destination register could be r10.

Now, let’s consider the PS case when a scratch register is the destination of an instruction.

p0 → f0-f4

src0 → r0-r7

src1 → r8-r15

dest → s16-s23 / r16-r23

When predication is not on (or predication with swizzle control on), we have

```
add (16) m4:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m6:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m8:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

add (16) m10:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (8)  r3:d 0x76543210:v {NoMask}                 // ramp function

mul (16) acc0:d r3:d 16 {NoMask}            // ramp function

add (8)  acc0:d acc0:d 64 {NoMask,SecHalf}          // ramp function

add (16) m2:d acc0:d 2*256 {NoMask}                 // ramp function
    send (16) null m1 r3 DATAPORT|RC|WRITE_SIMD16
```

As there is no bit left from the unit specified descriptor field, the 4 bit mask must be put into the header field in m1, which requires at least two more instructions.

Alternatively, or for the case that predication without modifier is on, we can do a read-modify-write.

```
mov (8)  r3:d 0x76543210:v {NoMask}                    // ramp function

mul (16) acc0:d r3:d 16 {NoMask}              // ramp function

add (8)  acc0:d acc0:d 64 {NoMask,SecHalf}             // ramp function

add (16) m2:d acc0:d 2*256 {NoMask}                    // ramp function
    send (16) r16 m1 r3 DATAPORT|RC|READ_SIMD16 // read from scratch

// some of the following four instructions may be omitted based on [.mask] field

[([!]f0.{sel|any4v|all4v})] add (16) r16:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)]
r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r18:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)]
r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r20:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)]
r8/10/12/14_BasedOnSwizzle:f

[([!]f0.{sel|any4v|all4v})] add (16) r22:f [-][(abs)]r0/2/4/6_BasedOnSwizzle:f [-][(abs)]
r8/10/12/14_BasedOnSwizzle:f

mov (16) m4:f r16:f {NoMask}

mov (16) m6:f r18:f {NoMask}

mov (16) m8:f r20:f {NoMask}

mov (16) m10:f r22:f {NoMask}
    send (16) null  m1 null DATAPORT|RC|WRITE_SIMD16 {NoMask}     // write back to scratch
```

9.2.6 Flow Control Instructions

Unconditional branches are performed through direct manipulation of the 32-bit IP architectural register. For example:

```
mov (1) IP <memory_address>      // jump absolute
add (1) IP  IP  <byte_count>     // jump relative
```

Note that jump distances are specified in terms of bytes, as opposed to instruction counts in the case of *break*, *halt*, etc. To minimize confusion, an assembler-only instruction ‘jmp <inst_count>’, where <inst_count> is an immediate term, may be defined which takes an instruction count for a distance. The *jmp* pseudo-opcode can be mapped to an “add (1) ip ip <inst_count> * 16” instruction.

Also note that IP is always an instruction-sized aligned address (16 bytes), thus the 4 LSB’s are not maintained in the IP architectural register and should not be relied upon by software.

IP, when used as a source operand, reflects the memory address of the instruction in which it is used. The following are examples illustrating the use of IP:

```
     add (1) IP 4*16       // jumps to HERE_1
     add (1) IP 0x35       // jumps to HERE_1 (4 lsb’s don’t-care)
     <instruction>
     <instruction>
HERE_1:    <instruction>
HERE_2:    <instruction>
     <instruction>
     add (1) IP -2*16      // jumps to HERE_2
     ...
     add (1) IP 0          // infinite loop
     add (1) IP 0xF        // infinite loop
     ...
```

Note for Assembler: *The if/iff/else/while/break instructions identify relative addresses as the targets of an implicit jump associated with the instruction. These are optional in the assembly syntax as the jitter can determine the location of the matching instruction (e.g. matching endif instruction for a given if instruction).*

9.2.7 Execution Masking

9.2.7.1 Branching

Example 9-3. If / Else / EndIf

```
//------------------------------------------------------------------
// Example if/else/endif scenario
//             "if (r5==r4) ...else ... end-if"
//------------------------------------------------------------------
                  ...
                  cmp.e.f0 (8)  null r5 r4      // does r5 == r4?
                  (f0) if (8)   HERE_1                   // "if" part - save then update IMASK;
                                                                  //    or goto the 'else' if all false
                  ...
                  ...
HERE_1:                                                                     // now do the 'else' part
                  else (8) HERE_2                        // "else" part - invert IMASK
                                                                  //    or goto the 'endif' if all false
                  ...
                  ...
HERE_2:
                  endif                                  // "end-if" part – restore IMASK
                  ....                                   // and continue...
```

If it is known that the code has no nested conditionals, a predicate can be used for a lower overhead, more efficient if/else/endif. (One must consider the probability of all channels taking the same branch, and the number of instructions under the if/else blocks as to which conditional method, predicate or mask, is most efficient).

9.2.7.2 Fast-If

Below is an example of a fast-if instruction. For the ‘iff’ instruction, only and iff-endif construct is allowed, as opposed to a if-else-endif. Note that the target address for branching if all enabled channels fail is one instruction beyond the endif, as the ‘iff’ does not push and update the IMask unless the branch is taken for at least one execution channel.

Example 9-4. Fast If

```
//-------------------------------------------------------------------
// Example – Fast If
//          One instruction overhead conditional
//-------------------------------------------------------------------
                    ...
                    cmp.e.f0 (8)  null r5 r4          // any flag update
                    ...
          (f0)      iff (8) HERE_1                              // “fast-if” – only pushes IMask;
                                                                          //   if execution falls through,
                                                                          //   else go to HERE_1
                    ...
                    ...
                    endif                                       // “end-if” part – restores IMask
HERE_1:
                    ...                                        // and continue...
```

9.2.7.3 Cascade Branching

As there is no ‘elseif’ instruction, a C-like cascade branching such as if / elseif / else / endif, can be realized using the basic building blocks of if / else / endif as shown in the following example. Notice that two ‘endif’s’ are required in order to pop the IStack correctly.

Example 9-5 If / Elseif / Else / EndIf

```
//-----------------------------------------------------------------
// Example if/elseif/else/endif scenario
//           “if (r5==r4) ...elseif (r6>r7) else ... end-if”
//-----------------------------------------------------------------
                    ...
                    cmp.e.f0 (8)  null r5 r4      // does r5 == r4?
          (f0)      if (8) HERE_1                           // “if” part - save then update IMask;
                                                                      //    or go to the ‘else’ part if all false
                    ...
                    ...
HERE_1:                                                                         // now do the ‘else’ part
                    else (8) HERE_2                         // “else if” part - invert IMask
                                                                      //    or go to the ‘else’ part if all false
                    cmp.g.f0 (8)  null r6 r7      // is r6 > r7?
          (f0)      if (8) HERE_3                           // “if” part - save then update IMask;
                                                                      //    or go to the ‘else’ part if all false
                    ...
                    ...
HERE_3:                                                                         // now do the ‘else’ part
                    else (8) HERE_4                         // “else” part - invert IMask
                                                                      //    or go to the ‘end-if’ part if all false
                    ...
                    ...
HERE_4:
                    endif                                   // “end-if” part – restore IMask for elseif
HERE_2:
                    endif                                   // “end-if” part – restore IMask for if
                    ....
```

9.2.7.4 Compound Branches

Compound branches are supported through the ability logically combine flag registers for each intermediate result.

Example 9-6 Compound Branch

```
//-----------------------------------------------------------------
// Example:  “if (r0 > r1) OR (r2 <= r3)”
//-----------------------------------------------------------------
                    ...
                    cmp.g.f0 (8)  null r0:d  r1:d                 // r0 > r1?
                    cmp.le.f1 (8) null r2:d  r3:d                 // r2 <= r3?
                    or (1) f0:w f0:w  f1:w                        // combine f0 and f1
          (f0) if (8) HERE_1                                      // Can now do normal if/else
                    ...
                    ...
HERE_1:                       endif

                    ...
```

Example 9-7. Compound Branch Using 'Any' or 'All'

```
//-------------------------------------------------------------------
// Example:  assuming we're doing a channel-serial vector in r0-r3
//           We want to know if all components of the vector are > 0x80
//-------------------------------------------------------------------
                    ...
                    cmp.g.f0 (16)   null r0 0x80                // r0 > 0x80?
                    cmp.g.f1 (16)   null r1 0x80                // r1 > 0x80?
                    cmp.g.f2 (16)   null r2 0x80                // r0 > 0x80?
                    cmp.g.f3 (16)   null r3 0x80                // r1 > 0x80?
          (f0.all4v) if (16) HERE_1
                    ...
                    ...                           // code executed only for those channels
                    ...                           // where per-channel r0,r1,r2,r3 all > 0x80
                    ...
HERE_1:                       endif
                    ...                                         //  and continue...
```

9.2.7.5 Looping

Due to GEN's SIMD-16 architecture, it must support the case of up to 16 loops running in parallel. These must be handled as independent loops, each with its own loop-exit condition which could occur after a different number of loop iterations. To account for each channel's progress, a 16b loop-mask 'LMask' is defined with 1b associated to each execution channel. This mask keeps track of which channels remain active inside a loop block.

Basic Do-While Loop

Example 9-8 illustrates the most basic loop. Two operations must be accomplished before loop entry. (1) Prior to loop entry, there is some subset of enabled channels as dictated by the code sequence prior. In general, the active status of each channel is indicated in the virtual EMask any point in time. These active channels will become the channels over which the loop is run, and LMask must be initialized with the EMask value. (2) Since a given loop may be nested within another loop, the previous LMask & CMask must be saved to the LStack for later restoration upon loop completion. The 'msave' instruction performs both the save and update in a single instruction, and thus all loop-blocks should be fronted with a "msave LStack LMask" and "msave LStack CMask" operation.

Note that the LMask and CMask share the same mask-stack. Thus, CMask must always be a 1's-subset of the LMask for proper stack operation. This is the case if CMask is updated to LMask each pass through the loop (see Example 9-8) and through the 'break' instruction updating both masks.

Each pass through the loop, a loop terminating operation must be evaluated and stored in a flag register. This condition must be evaluated on a channel-by-channel basis as exemplified:

```
                    cmp.z.f0(8) null r2 d3                      // any operation that updates a flag
```

The result of this operation sets a bit per channel in the specified flag register, which is then used in the 'while' instruction. As loops are performed, channels may become disabled as their termination condition is met.

'While' termination is determined on a channel-by-channel basis by the logical AND of corresponding bit positions of AMask, CMask and the specified flag. If the result is '1' the channel remains enabled for the next pass of the loop; if '0' the channel is disabled until loop fall-through. The 'while' instruction causes the LMask to be updated with the latest result of enabled channels. If any channel remains enabled (LMask != ...000b), an additional pass through the loop is made. Once a channel is terminated for the loop operation, it remains terminated until the loop is complete for all channels.

Upon fall through, the ‘while’ instruction causes the previously saved LMask & CMask to be popped from the LStack, enabling execution on the same subset of channels enabled prior to loop entry (unless a channel had been otherwise terminate inside the loop via ‘halt’).

Example 9-8. Basic Loop Construct

```
//--------------------------------------------------
//          Example: Basic do-while loop structure
//--------------------------------------------------
                              ...
                              do                                     // save L/CMask & update
BEGIN_LOOP:
                              mov (1) CMask LMask     {NoMask}       // update CMask for this pass
                              ...
                              ...
                              <some flag update>
                     (<p>)    while (8)  BEGIN_LOOP                  // cond. branch
                                                                     //    + restores LMask on fall-thru
                              ...
```

Do-While Loop with Break

A loop may also be terminated for any channel via the ‘break’ instruction. The ‘break’ instruction causes the corresponding bit positions of enabled channels to be cleared in the LMask. If the updated LMask = ...000b, a branch is made to the specified instruction location. An example is shown below in which the ‘break’ is at the same conditional-nesting level as the terminating ‘while’. Its primary value may simply be to support a “do...break.. while (true)” –type structure for a more direct 1:1 translation from higher-level source code.

Example 9-9. Loop Construct With Non-Nested ‘Break’

```
//------------------------------------------------------------
//          Example: While-true loop
//------------------------------------------------------------
#define BrkCode(i,d)          (i << 16) + d

                              do                             // save L/CMask & update
BEGIN_LOOP:
                              mov (1) CMask LMask     {NoMask} // update CMask for this pass
                              ...
                              <some flag update>
                     (<p>)    break (8) BrkCode(0,HERE_1)    // Restores LMask when all
                                                             // channels complete loop.
                              ...
                              ...
                              while (8) BEGIN_LOOP           // while true
HERE_1:
                              ...
```

A break condition may occur from various levels of nested-ifs. This gives rise to the possibility that a the loop may terminate from within nested ‘if’s, and due to the jump inherent in the ‘break’ instruction, the associated ‘endif’s are not encountered to clean-up the IStack as nesting levels are exited.

Example 9-10 Loop Construct With ‘Break’ From Within Nested If’s

```
//------------------------------------------------------------
//          Example: General Loop Structure w/ break inside if’s
//------------------------------------------------------------
#define BrkCode(i,d)        (i << 16) + d

                            do                                      // save L/CMask & update
BEGIN_LOOP:
                            mov (1) CMask LMask    {NoMask} // update CMask for this pass
                            ...
                            if ...
                            if ...
                            if ...
                            ...
                    (<p>)   break (8) BrkCode(3,HERE_1)     // we’re 3 levels deep, so
                            ...
                            endif
                            endif
                            endif
                            ...
                    (<p>)   break (8) BrkCode(0,HERE_1)
                            ...
                            while (8) <flag_spec> BEGIN_LOOP        // cond. branch
                                                                    // + restores C/LMask on fall-thru
HERE_1:
```

Do-While Loop with Continue

A continue instruction ‘cont’ is provided skip to the next iteration of the loop. Because not all channels participating in the loop may be enabled at the time this instruction is executed, some channels may require continuation of the loop. A special mask ‘CMask’ is defined which accounts for channels temporarily disabled for the current loop pass.

Since loops may nested, the CMask must be saved and restored around a loop similar to LMask. Since the CMask value within a properly constructed loop is always a subset of the LMask, it can share the LStack for storage, so long as it is pushed after LMask as shown in Example 9-11. This save/restore operations are not required if the loop being entered does not have any occurrence of a continue instruction.

Example 9-11. Do-While with Continue

```
//------------------------------------------------------------
//          Example: General Loop Structure w/ basic break and cont.
//------------------------------------------------------------
#define ContCode(i,d)       (i << 16) + d

                            do                              // save L/CMask & update
BEGIN_LOOP:
                            mov (1) CMask EMask             // re-initialize CMask for this pass
                            ...
                            ...
                    (<p>) cont (8) ContCode(0,HERE_1)
                            ...
HERE_1:
                    (<p>)   while (8) BEGIN_LOOP            // cond. branch
                                                            //    + restores C/LMask on fall-thru
```

...

9.2.7.6 Indexed Jump

Example 9-12. Indexed Jump

```
                //-------------------------------------------------------------------
        // Code example shows the use of jmpi to perform a case statement
        // of any number of options in 3 jumps
        //-------------------------------------------------------------------
        .default_execution_size     8
        ...
        jmpi r0<0,1,0>                      // jump relative, based on r0.a.x
                                            // ----- Jump Table ------
        jmp HERE_0                          // redirect for case 0
        jmp HERE_1                          // redirect for case 1
        jmp HERE_2                          // redirect for case 2
        jmp HERE_3                          // redirect for case 3
        ...
HERE_0:                                     // ... case 0 ...
        ...
        jmp DONE
HERE_1:                                     // ... case 1 ...
        ...
        jmp DONE
HERE_2:                                     // ... case 2 ...
        ...
        jmp DONE
HERE_3:                                     // ... case 3 ...
        ...
DONE:
        ...                                 //  and continue...
```