Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM)
Volume 4 Part 3: Execution Unit ISA (Ivy Bridge)

For the 2012 Intel® Core™ Processor Family

May 2012

Revision 1.0

NOTICE:
This document contains information on products in the design phase of development, and Intel reserves the right to add or remove product features at any time, with or without changes to this open source documentation.

Doc Ref #: IHD-OS-V4 Pt 3 – 05 12

Creative Commons License

You are free to Share — to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

Contents

1. Introduction

This chapter contains these sections that introduce this volume.

- Introducing the Execution Unit
- EU Terms and Acronyms
- EU Changes by Processor Generation
- EU Notation

Subsequent chapters cover:

- EU Data Types
- Execution Environment
- Exceptions
- Instruction Set Summary
- Instruction Set Reference
- EU Programming Guide

The EU Programming Guide provides some useful examples and information but is not a complete or comprehensive programming guide.

1.1 Introducing the Execution Unit

This section introduces the Execution Unit (EU), a simple and capable processor within the GPU that supports graphics processing within the graphics pipelines, can do general purpose computing (GPGPU), and responds to exceptional conditions via the System Routine.

The EU provides parallelism at two levels: thread and data element. Multiple threads can execute on the EU; the number executing concurrently depends on the processor and is transparent to EU code. Each thread has its own registers (GRF and ARF, described below). Most EU instructions operate on arrays of data elements; the number of data elements is normally the *ExecSize (execution size)* or number of channels for the instruction. A *channel* is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels is independent of the number of physical ALUs or FPUs for a particular graphics processor.

EU native instructions are 128 bits (16 bytes) wide. Some combinations of instruction options can use compact instruction formats that are 64 bits (8 bytes) wide. Identifying instructions that can be compacted and creating the compact representations is done by software tools, including compilers and assemblers.

Data manipulation instructions have a destination operand (dst) and one, two, or three source operands (src0, src1, or src2). The instruction opcode determines the number of source operands. An instruction's last source operand can be an immediate value rather than a register.

Data read or written by a thread is generally in the thread's *GRF (General Register File)*, 128 general registers, each 32 bytes. A data element address within the GRF is denoted by a register number (r0 to r127) and a subregister number. In the instruction syntax, subregister numbers are in units of data element size. For example, a :d (Signed Doubleword Integer) element can be in subregister 0 to 7, corresponding to byte numbers in the instruction encoding of 0, 4, ... 28.

Note: The EU cannot directly read or write data in system memory.

Specialized registers used to implement the ISA are in a distinct per thread *Architecture Register File (ARF)*. Each such register or group of related registers has its own distinct name. For example, ip is the instruction pointer and f0 is a flags register. An ARF register can be a src0 or dst operand but not a src1 or src2 operand. There are restrictions on how particular ARF registers are accessed that should be understood before directly reading or writing those registers. See the ARF Registers section for more information.

The EU supports both integer and floating-point data types, as described in the Numeric Data Types section.

For EU flow control, each channel has its own per-channel instruction pointer (PcIP[n]) and only executes an instruction when IP == PcIP[n] and any other masks enable the channel. Most flow control instructions use signed offsets from the current instruction address to reference their targets. Unconditional branches are done using *mov* with IP as the destination. Flow control can also use SPF (Single Program Flow) mode to execute with a single instruction pointer (IP).

The EU ISA supports predication, masking, regioning, swizzling, some type conversions, source modification, saturation, accumulator updates, and flag updates as part of instruction execution:

- *Predication* creates a bit mask (*PMask*) to enable or disable channels for a particular instruction execution. Pmask is derived from flag register and subregister values using boolean formulas determined by the PredCtrl (Predicate Control) and PredInv (Predicate Inversion) instruction fields. See the Predication section.
- *Masking* is the overall process of determining which channels execute for a given instruction based on five factors:
  - Number of channels (only channels in [0, *ExecSize* - 1] can execute)
  - Execution mask (*EMask*)
  - Whether the channel is on the instruction (if not in Single Program Flow mode and MaskCtrl is not NoMask)
  - Predicate mask (*PMask*)
  - In Align16 mode, any enabling of channels using the Dst.ChanEn instruction field (if MaskCtrl is not NoMask).
- *Regioning* specifies an array of data elements contained in one or two registers, with options for scattering, interleaving, or repeating data elements in registers using width and stride values, subject to significant constraints. Regioning also includes access mode (Align1 or Align16) and addressing mode (Direct or Indirect). See the Registers and Register Regions section.
- *Swizzling* allows small scale reordering of data elements within groups of four at the input using the modulo 4 channel names x, y, z, and w. For example, a swizzle of .wzyx with an *ExecSize* of 8 reads execution channels 0 to 7 from these input channels: 3, 2, 1, 0, 7, 6, 5, and 4. Swizzling is only available in the Align16 access mode, described in the Execution Environment chapter.
- *Type Conversions* do any needed conversion from source data type to execution data type and from execution data type to destination data type. See Execution Data Type for more information. Each instruction description indicates what combinations of data types are supported.
- *Source Modification* modifies a source operand just before doing the requested operation. For a numeric operation, the choices are:
  - No modification (normal).
  - - indicating negation.
  - (abs) indicating absolute value.
  - -(abs) indicating a forced negative value.

Source modification logically occurs after any conversion from source data type to execution data type. Each instruction description indicates whether it supports source modification.

- *Saturation* clamps result values to the nearest value within a saturation range determined by the destination type. For a floating-point type, the saturation range is [0.0, 1.0]. For an integer type, the saturation range is the entire range for that type, for example [0, 65535] for the UW (Unsigned Word) type. Each instruction description indicates whether it supports saturation.
- *Accumulator Updates* optionally update the accumulator register or registers in the ARF with destination values as a side effect of instruction execution. The AccWrCtrl instruction field enables accumulator updates. The Accumulator Disable flag in control register 0 (cr0) can be used to disable accumulator updates, regardless of AccWrCtrl values; for example, this flag may be used in the System Routine.
- *Flag Updates* optionally update a flags register and subregister (f0.0, f0.1, f1.0, or f1.1) with conditional flags based on the CondModifier (Condition Modifier) instruction field. For example, a CondModifier of **.nz** (not zero) assigns flag bits based on whether result elements are not zero (1) or zero (0). Each instruction description indicates whether it supports the Condition Modifier and any restrictions on the values supported.

Note: The EU is not required to execute steps in its internal pipeline sequentially or in order, so long as it produces correct results.

The assembler syntax uses spaces between operands and encloses ExecSize and any predicate in parentheses. Instruction mnemonics, register names, conditional modifiers, predicate controls, and type designators use lowercase. Function names used with the math instruction are UPPERCASE.

```
( pred ) inst cmod sat ( exec_size ) dst src0 src1 { inst_opt, ... }
```

General register destination regions use the syntax r*m.n<HorzStride>*:type. General register directly addressed source regions use the syntax r*m.n<VertStride*;*Width*,*HorzStride*>:type. You need to understand more about register regioning to understand all of these terms.

The following example assembly language instruction adds two packed 16-element single-precision Float arrays in r4/r5 and r2/r3 writing results to r0/r1, only on those channels enabled by the predicate in f0.0 along with any other applicable masks.

```
(f0.0) add (16) r0.0<1>:f r2.0<8;8,1>:f r4.0<8;8,1>:f
```

1.2 EU Terms and Acronyms

This section provides three tables describing EU general terms and acronyms, EU data types, and EU selected ARF registers.

EU General Terms and Acronyms

Term	Description
ALT mode	A floating-point execution mode that maps +/- inf to +/- fmax, +/- denorm to +/-0, and NaN to +0 at the FPU inputs and never produces infinities, denormals, or NaN values as outputs. See IEEE mode.
ALU	Arithmetic Logic Unit. A functional block that performs integer arithmetic and logic operations, as distinct from instruction fetch and decode, floating-point operations (see FPU), or messaging.
AOS	Array Of Structures. Also see SOA.
ARF	Architecture Register File, a distinct register file containing registers used to implement specific ISA features. For example the Instruction Pointer and condition flags are in ARF registers. See GRF.
byte	An 8-bit value aligned on an 8-bit boundary and the basic unit of addressing. Bits within a byte are denoted 0 to 7 from LSB to MSB.

Term	Description
channel	A logical unit of SIMD data parallel execution within a thread and within the EU. The number of physical ALUs or FPUs is not directly related to the number of channels. Supports up to 16 channels. Supports up to 32 channels.
compact instruction	A 64-bit instruction encoded as described in the EU Compact Instructions section. Only some combinations of instruction parameters can be encoded as compact instructions. See native instruction.
compressed instruction	An instruction that writes to two destination registers. For example a SIMD16 instruction with Float operands can write channels 0 to 7 to one 32-byte general register and channels 8 to 15 to a second, consecutive 32-byte general register.
denorm	A very small but nonzero number in IEEE mode, with a magnitude less than the smallest normalized floating-point number representable in a particular floating-point format. Denormals lose precision as their values approach zero, called *gradual underflow*.
DWord	Doubleword. A 32-bit (4-byte) value aligned on a 32-bit (4-byte) boundary. Bits within a DWord are denoted 0 to 31 from LSB to MSB.
EOT	End of Thread. A flag set on a *send* or *sendc* instruction to terminate a thread's execution on the EU.
EU	Execution Unit. The single GPU unit described in this volume. This volume describes individual data parallel execution paths within a thread in the EU as *channels*. A few fields, like EUID, use EU to refer to a particular hardware resource used to implement the overall EU.
exception	An error or interrupt condition that arises during execution that may transfer control to the System Routine. Some exceptions can be disabled, preventing such transfers. As defined in this volume, some errors do not produce exceptions.
ExecSize	The number of execution channels for a particular instruction. Channels within that number are enabled or disabled by various masks.
floating-point	Numeric types that allow fractional values and often a wider range than integer types. The EU supports binary floating-point types including the single precision type and the double precision typedefined by the IEEE 754 standard.
GEN	GEN is sometimes used to refer to Intel's mainstream GPU architecture integrated with recent CPU generations.
GRF	General Register File, a distinct register file containing 128 general registers, r0 to r127. Each general register is 256 bits (32 bytes), can contain any type of data, and can be accessed with any valid combination of addressing mode, access mode, and region parameters. A general register is directly addressed using a register number and subregister number, or indirectly addressed using an address subregister (index register) and an address immediate offset.
IEEE mode	A floating-point execution mode that supports all the kinds of floating-point values described by the IEEE 754 standard: normalized finite nonzero binary floating-point numbers, signed zeros, signed infinities, signed denormals that are closer to zero than any normalized value but still nonzero, and NaN (not a number) values. See ALT mode.
index register	An address subregister when used for indirect addressing.
inf	Infinity, +inf or -inf, as a floating-point value in IEEE mode.
instruction	In this volume, *instruction* always refers to an EU instruction.
ISA	Instruction Set Architecture, processor aspects visible to programs and programmers and independent of a particular implementation, including data types, registers, memory access, addressing modes, exceptions, instruction encodings, and the instruction set itself. An ISA does not include instruction timing, hardware pipeline details, or the number of physical resources (ALUs, FPUs, instruction decoders) mapped to logical constructs (threads, channels). This volume also includes a recommended assembly language syntax, closely related to the ISA but logically distinct from it.
LSB	Least significant bit.
message	A data structure transmitted from a thread to another thread, to a shared function, or to a fixed function. Message passing is the primary communication mechanism of the GEN architecture.
MSB	Most significant bit.

Term	Description
NaN	Not a Number. A non-numeric value allowed in the standard single precision and double precision floating-point number formats. Quiet NaNs propagate through calculations and signaling NaNs cause exceptions. NaNs are not used in the ALT floating-point mode.
native instruction	A 128-bit instruction, the regular instruction format that allows all defined instruction parameters and options. Some instructions can also be encoded using a 64-bit compact instruction format.
OWord	Octword. A 128-bit (16-byte) value aligned on a 128-bit (16-byte) boundary. Bits within an OWord are denoted 0 to 127 from LSB to MSB. This term is used rarely and may be dropped from future versions of this volume.
packed	A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist. The immediate vector data types are all described as *Packed* because each such type packs several small data elements into a 32-bit immediate value.
QWord	Quadword. A 64-bit (8-byte) value aligned on a 64-bit (8-byte) boundary. Bits within a QWord are denoted 0 to 63 from LSB to MSB.
region	A collection of data locations in registers and subregisters for a source or destination operand. The associated regioning parameters allow regions to be arrays with various layouts.
register	Part of the directly accessible state of an EU program, such as a general register in the GRF or an architecture register in the ARF. Note that system memory is not directly accessible.
SIMD	Single Instruction Multiple Data. Each EU instruction can operate on multiple data elements in parallel, as specified by the instruction's ExecSize.
SIP	System Instruction Pointer, the starting IP value for the System Routine.
SOA	Structure of Arrays. Also see AOS.
SPF	Single Program Flow. A mode in which every execution channel uses the common instruction pointer, IP in the ip register. The SPF bit in the control register is 1 to enable SPF and 0 to disable it. If SPF is disabled, then each execution channel n has its own instruction pointer, PcIP[n] and each channel n is only eligible to execute, subject to other masking, when PcIP[n] == IP.
swizzle	Rearrange data elements within a vector. The EU supports modulo four swizzling of register source operands at the input in the Align16 access mode.
System Routine	A global EU exception handling routine. Any enabled exception from any EU thread transfers control to this routine.
thread	An instance of a program executing on the EU. The life cycle for a thread on the EU starts with the first instruction after being dispatched to the EU by the Thread Dispatcher and ends after executing a *send* or *sendc* instruction with EOT set, signaling thread termination. Threads can be independent or can communicate with each other via the Message Gateway shared function.
word	A 16-bit (2-byte) value aligned on a 16-bit (2-byte) boundary. Bits within a word are denoted 0 to 15 from LSB to MSB. *Word* has denoted a 16-bit unit for Intel processors since the 8086 and 8088 processors were introduced in 1978.

The next table lists all EU numeric data types. See the Numeric Data Types section for more information about each data type.

EU Numeric Data Types (Listed Alphabetically by Short Name)

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
B	:b	Signed Byte Integer	1	8	I	Signed integer in the range -128 to 127.
D	:d	Signed Doubleword Integer	4	32	I	Signed integer in the range -2^{31} to $2^{31} - 1$.
DF	:df	Double Float	8	64	F	Double precision floating-point number.
F	:f	Float	4	32	F	Single precision floating-point number.

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
UB	:ub	Unsigned Byte Integer	1	8	I	Unsigned integer in the range 0 to 255.
UD	:ud	Unsigned Doubleword Integer	4	32	I	Unsigned integer in the range 0 to 2^{32} - 1.
UV	:uv	Packed Unsigned Half Byte Integer Vector	4	32	I	Eight 4-bit unsigned integer values each in the range 0 to 15. Only used as an immediate value.
UW	:uw	Unsigned Word Integer	2	16	I	Unsigned integer in the range 0 to 65,535.
V	:v	Packed Signed Half Byte Integer Vector	4	32	I	Eight 4-bit signed integer values each in the range -8 to 7. Only used as an immediate value.
VF	:vf	Packed Restricted Float Vector	4	32	F	Four 8-bit restricted float values. Only used as an immediate value.
W	:w	Signed Word Integer	2	16	I	Signed integer in the range -32,768 to 32,767.

The next table lists the seven ARF registers that you should understand first, omitting several others. See the ARF Registers section for more information, including descriptions of additional registers not listed below.

EU Selected ARF Registers (Listed Alphabetically by Name)

Name	Assembler Syntax	Description
Accumulators	acc0, acc1	Data registers that can hold integer or floating-point values of various sizes. Many instructions can implicitly update accumulators with a copy of destination values, done by setting the AccWrCtrl instruction option. A few instructions, like *mac* (Multiply Accumulate), use the accumulators as an implicit source operand, useful for some iterative calculations.
Address Register	a0.*s*	Holds subregisters primarily used for indirect addressing. Each subregister is a 16-bit UW (Unsigned Word) value. For an indirectly addressed operand or element, the subregister value plus an AddrImm signed offset field determines the byte address (RegNum and SubRegNum) within the register file (GRF). There are 8 address subregisters.
Control Register	cr0.*s*	Contains bit fields for floating-point modes, flow control modes, and exception enable/disable. Also contains exception indicator flags and saves the AIP (Application Instruction Pointer) on transferring control to the System Routine to handle an exception.
Flags	f*r.s*	Used as the outputs for various channel conditional signals, such as equality/zero or overflow. Used as the inputs for predication. There are two 32-bit flags registers each containing two 16-bit subregisters.
Instruction Pointer (IP)	ip	References the current instruction in memory, as an unsigned offset from the General State Base Address. IP is the thread's overall instruction pointer. Each channel n can have its own instruction pointer (PcIP[n]). If not in Single Program Flow mode (SPF is 0) then only those channels where PcIP[n] == IP are eligible to execute the instruction, if enabled by all other applicable masks.
Null Register	null	Indicates a non-existent operand. Unused operands in the instruction format, like the unused second source operand field in a *mov* instruction, are encoded as null. For present source operands, reading a null source operand returns undefined values. For null destination operands, results are discarded but any implicit updates to accumulators or flags still occur.
State Register	sr0.*s*	Contains thread identification and scheduling fields, and mask fields for enabling or

Name	Assembler Syntax	Description
		disabling channels.

1.3 EU Changes by Processor Generation

This section describes how the EU changes for particular processor generations. Instruction compaction tables can differ for each generation, so that is not mentioned in these lists. Particular readers and audiences can see only certain content in this section. Errata and workarounds for particular generations, SKUs, or steppings are not included in these lists. Some small changes in instruction layouts are not included in these lists.

Ivy Bridge

These features or behaviors are added , continuing to later generations:

- The maximum *ExecSize* increases to 32, for byte or word operands.
- Increase the number of flag registers from one to two.
- Add the NibCtrl field, used with QtrCtrl to select groups of channels or flags.
- Add the DF (Double Float) data type, the first time an 8-byte data type is supported. DF only supports the IEEE floating-point mode and not the ALT floating-point mode.
- Add a shared source data type field and a destination data type field for instructions with three source operands, allowing F (Float), DF (Double Float), D (Signed Doubleword Integer), or UD (Unsigned Doubleword Integer) types to be specified.
- Add bit manipulation instructions: *bfi1*, *bfi2*, *bfrev*, *cbit*, *fbh*, and *fbl*.
- Add the integer *addc* (Add with Carry) and *subb* (Subtract with Borrow) instructions.
- Add the *brc* (Branch Converging) and *brd* (Branch Diverging) instructions.
- For the *cmp* and *cmpn* instructions, relax the accumulator restrictions.
- For the *sel* instruction, remove the accumulator restriction.
- Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0.

These features or behaviors are specific to and may not continue to later generations:

- Each DF (Double Float) operand uses an element size of 4 rather than 8 and all regioning parameters are twice what the values would be based on the true element size: *ExecSize*, *Width*, *HorzStride*, and *VertStride*. Each DF operand uses a pair of channels and all masking and swizzling should be adjusted appropriately.
- The *f16to32* and *f32to16* instructions convert between half-precision float and Float.
- The *mul* instruction limits integer multiplication involving DWords so that only the low 16 bits of src1 are used even if src1 is a DWord.
- The *sel* (Select) instruction does not support an *ExecSize* of 32.
- SIMD16 execution on DWords is not allowed when an accumulator is an explicit source or destination operand.

1.4 EU Notation

The `Courier New` font is used for code examples and for the Syntax, Format, and Pseudocode sections in the instruction reference.

The *italic* font style is used for instruction mnemonics outside of code (e.g., the *send* instruction), for syntactic production names, for key values in algorithms (*ExecSize*), and to emphasize a word or phrase. For example: When bit 10 is set, the destination register scoreboard is *not* cleared.

The **bold** font weight is used for the short name and long name of a bit field being described, for value names being defined, for syntactic terminals, for unnumbered subheadings, and for the terms Note, Erratum/Errata, or Workaround used to introduce a paragraph.

Bit field names and value names used where not being defined and not as syntactic terminals are in plain text.

Bit field values in hex use the 0x prefix. The BSpec currently uses the 0x prefix for hex in some parts and the h suffix for hex in other parts. For single bits, values appear as simply 0 or 1. For multi-bit binary values, the appropriate number of binary digits appears with a b suffix.

Instruction mnemonics are lowercase. Function names invoked using the *math* instruction are UPPERCASE. For example, SQRT.

Tables describing bit field layouts or registers proceed from most significant to least significant bits. Figures showing bit fields or registers show most significant bits on the left and least significant bits on the right.

Any bit, field, or register described as Reserved should be regarded as undefined and unpredictable. Such bits should be treated as follows:

- When testing values, do not depend on the state of reserved bits. Mask out or otherwise ignore such bits.
- Sometimes software must initialize reserved bits. For example, a compiler must write complete instruction values when creating an instruction stream, including reserved bits. In such cases, write reserved bits as zeros unless otherwise indicated.
- Do not use reserved bits as extra storage for software-defined values; put nothing in such bits.
- When saving state and restoring state, save and restore any reserved bits as well.
- Do not assume that reserved bits are invariant between explicit writes. Software should function even if reserved bits change in undefined and unpredictable ways.

Any value, encoding, or combination of values or encodings described as Reserved must not be used. The EU's behavior is undefined in this case.

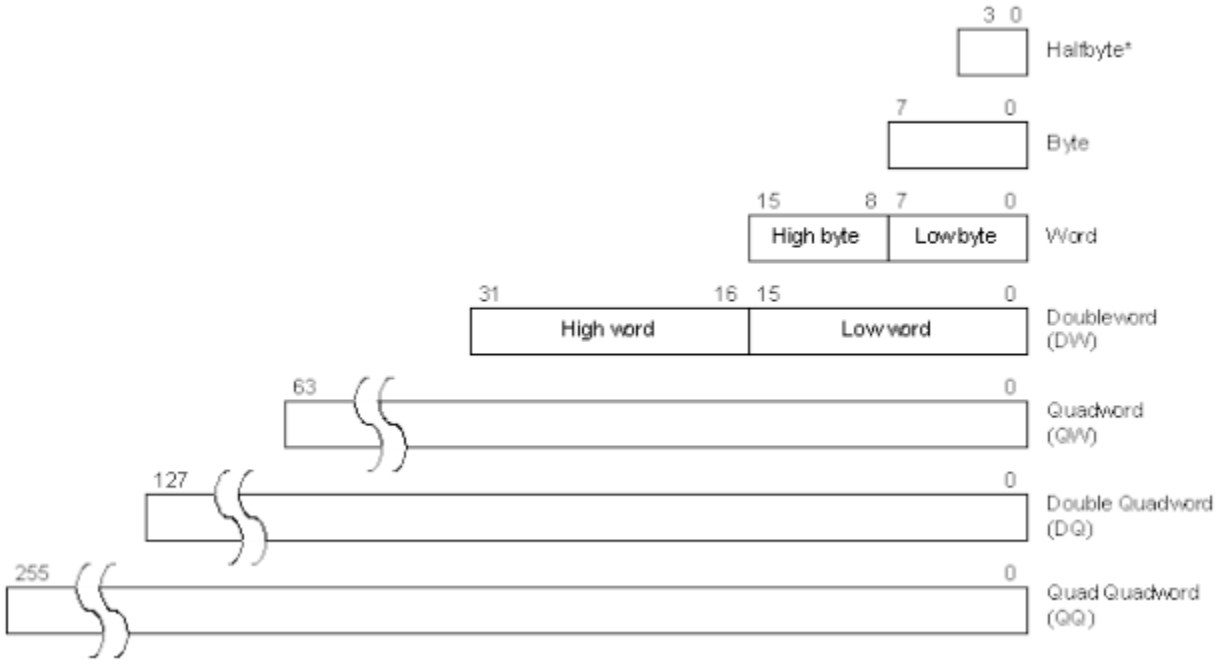
When a combination of instruction parameters or an EU state is described as producing undefined results or behavior, do not assume that undefined results or behavior are confined to specific instructions, operands, registers, or channels.

2. EU Data Types

2.1 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 *Fundamental Data Types*, 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 numeric data type of “packed signed halfbyte integer vector” described in the next section.

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 numeric data type. Instead, they are used to describe a group (a vector) of numeric data elements of smaller size align to larger natural boundaries.

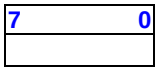
2.2 Numeric Data Types

The numeric data types defined in the GEN architecture include signed and unsigned integers and floating-point numbers (floats) of various sizes. These numeric data types are described below.

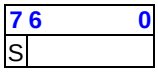
2.2.1 Integer Numeric Data Types

The Execution Unit supports the following integer data types. Signed integer types use two's complement representation for negative numbers.

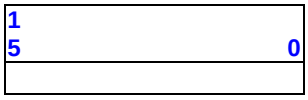
UB: Unsigned Byte, 8-bit Unsigned Integer



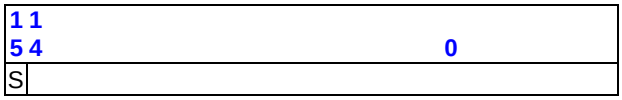
B: Byte, 8-bit Signed Integer



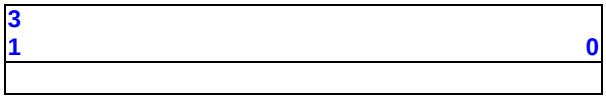
UW: Unsigned Word, 16-bit Unsigned Integer



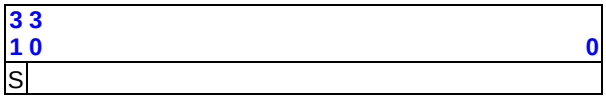
W: Word, 16-bit Signed Integer



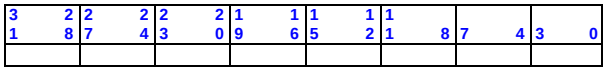
UD: Unsigned Doubleword, 32-bit Unsigned Integer



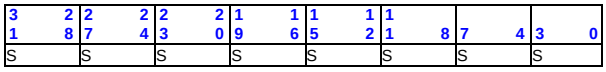
D: Doubleword, 32-bit Signed Integer



UV: Packed Unsigned Half-Byte Integer Vector, 8 x 4-Bit Unsigned Integer



V: Packed Signed Half-Byte Integer Vector, 8 x 4-Bit Signed Integer



The following table summarizes the EU integer data types.

Execution Unit Integer Data Types

Notation	Size in Bits	Name	Range
UB	8	Unsigned Byte Integer	[0, 255]
B	8	Signed Byte Integer	[-128, 127]
UW	16	Unsigned Word Integer	[0, 65535]
W	16	Signed Word Integer	[-32768, 32767]

Notation	Size in Bits	Name	Range
UD	32	Unsigned Doubleword Integer	[0, 2^{32} – 1]
D	32	Signed Doubleword Integer	[-2^{31}, 2^{31} – 1]
UV	32	Packed Unsigned Half-Byte Integer Vector	[0, 15] in each of eight 4-bit immediate vector elements.
V	32	Packed Signed Half-Byte Integer Vector	[-8, 7] in each of eight 4-bit immediate vector elements.

Restriction: Only a raw move using the *mov* instruction supports a packed byte destination register region. For information about raw moves, refer to the **Description** in *mov – Move*.

2.2.2 Floating-Point Numeric Data Types

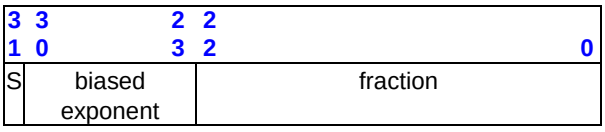
The Execution Unit supports the following floating-point data types. The Float and Double Float types use the single precision and double precision formats specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. In the ALT floating-point mode, representations for infinities, denorms, and NaNs within those formats are not used. The EU does not support the double extended precision (80-bit) floating-point format found in the x86/x87/Intel 64 floating-point registers. All floating-point formats are signed using signed magnitude representation (a distinct sign bit, separate from the magnitude information).

The F (Float) type supports both the ALT and IEEE floating-point modes, controlled by the Single Precision Floating-Point Mode bit in the Control Register.

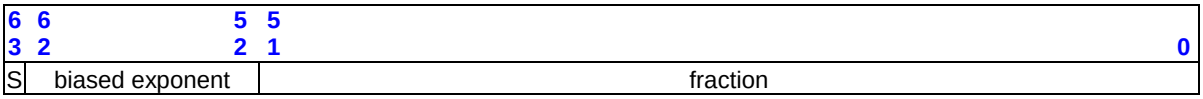
In IEEE mode, F calculations flush denormalized values to zero and gradual underflow is not supported.

The DF (Double Float) type only supports the IEEE floating-point mode. Whether DF calculations support denorms or flush denormalized values to zero is controlled by the Double Precision Denorm Mode bit in the Control Register.

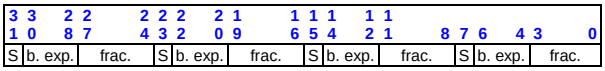
F: Float, 32-bit Single-Precision Floating-Point Number



DF: Double Float, 64-bit Double-Precision Floating-Point Number+



VF: Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number



The following table summarizes the EU floating-point data types.

Execution Unit Floating-Point Data Types

Notation	Size in Bits	Name	Range
F	32	Float	Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand:

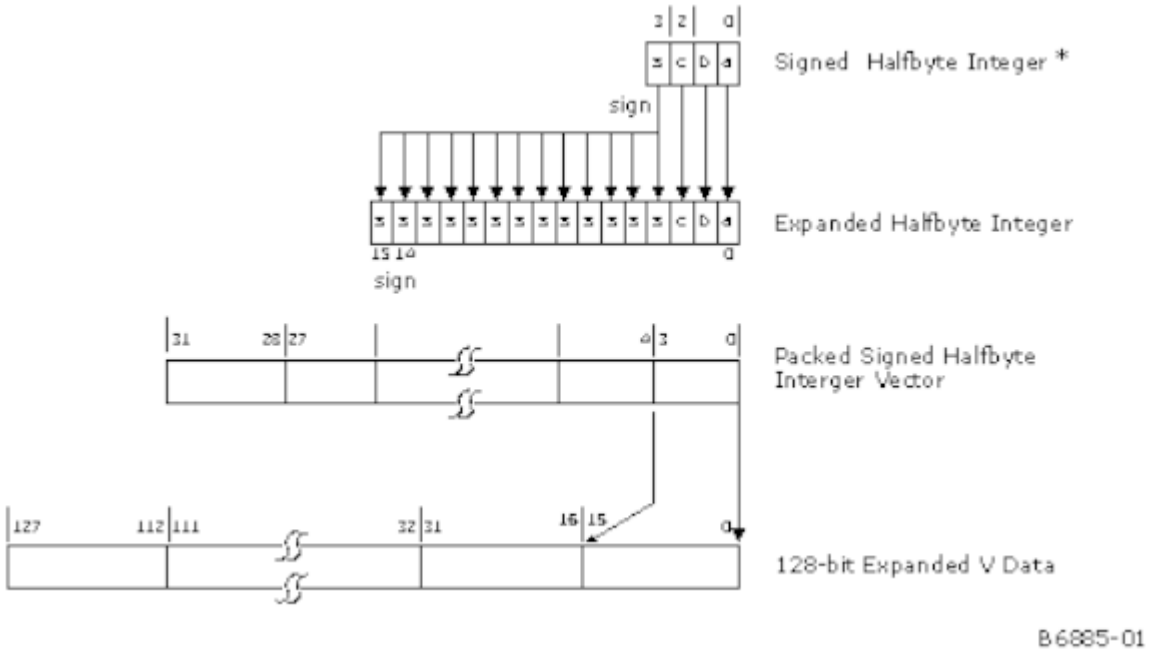
Notation	Size in Bits	Name	Range
			[–(2–2^{-23})^{127}…–2^{-149}, 0.0, 2^{-149}… (2–2^{-23})^{127}]
DF	64	Double Float	Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: [–(2–2^{-52})^{1023}…–2^{-1074}, 0.0, 2^{-1074}… (2–2^{-52})^{1023}]
VF	32	Packed Restricted Float Vector	Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit, 3 bits for the biased exponent (bias of 3), and 4 bits for the significand: [–31…–0.125, 0, 0.125… 31]

2.2.3 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 numeric 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 vector into an 8-element signed word vector by sign extension. This is illustrated in *Numeric Data Types*.

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

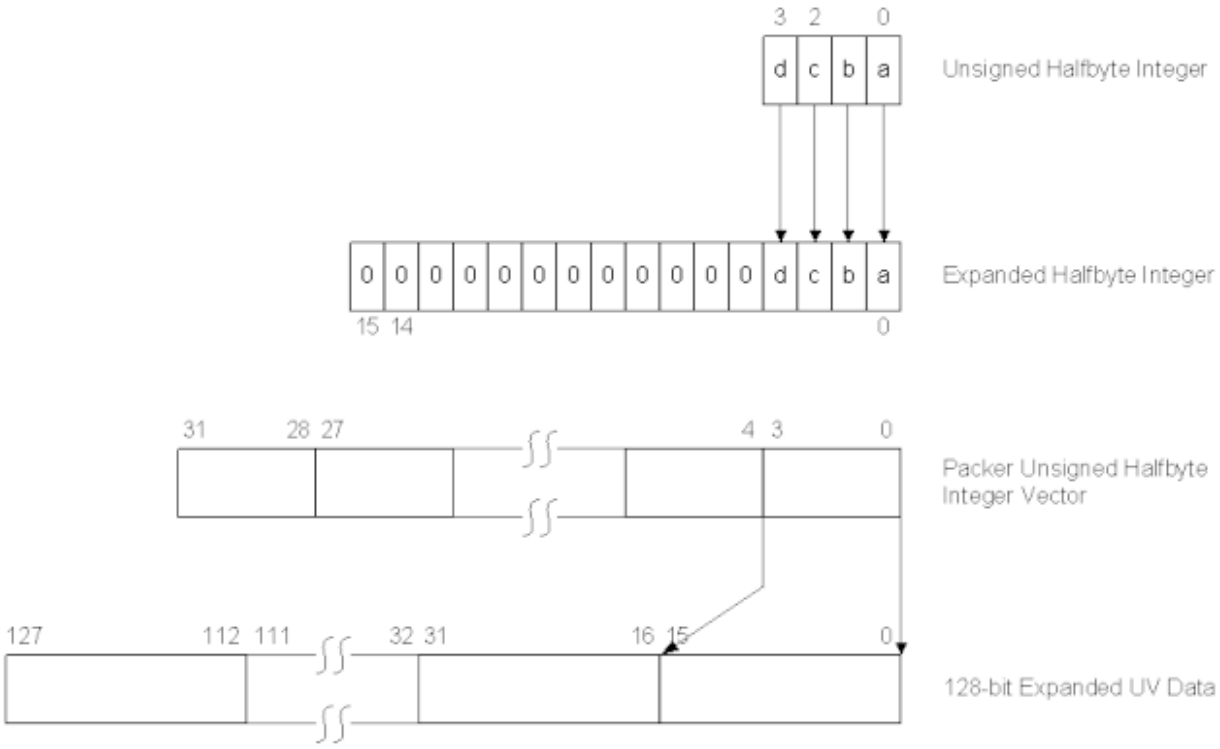
Converting a Packed Half-Byte Vector to a 128-bit Signed Integer Vector



2.2.4 Packed UnSigned Half-Byte Integer Vector

A packed unsigned halfbyte integer vector consists of 8 unsigned halfbyte integers contained in a doubleword. Each unsigned halfbyte integer element has a range from 0 to 15. This numeric 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 vector into an 8-element signed word vector.

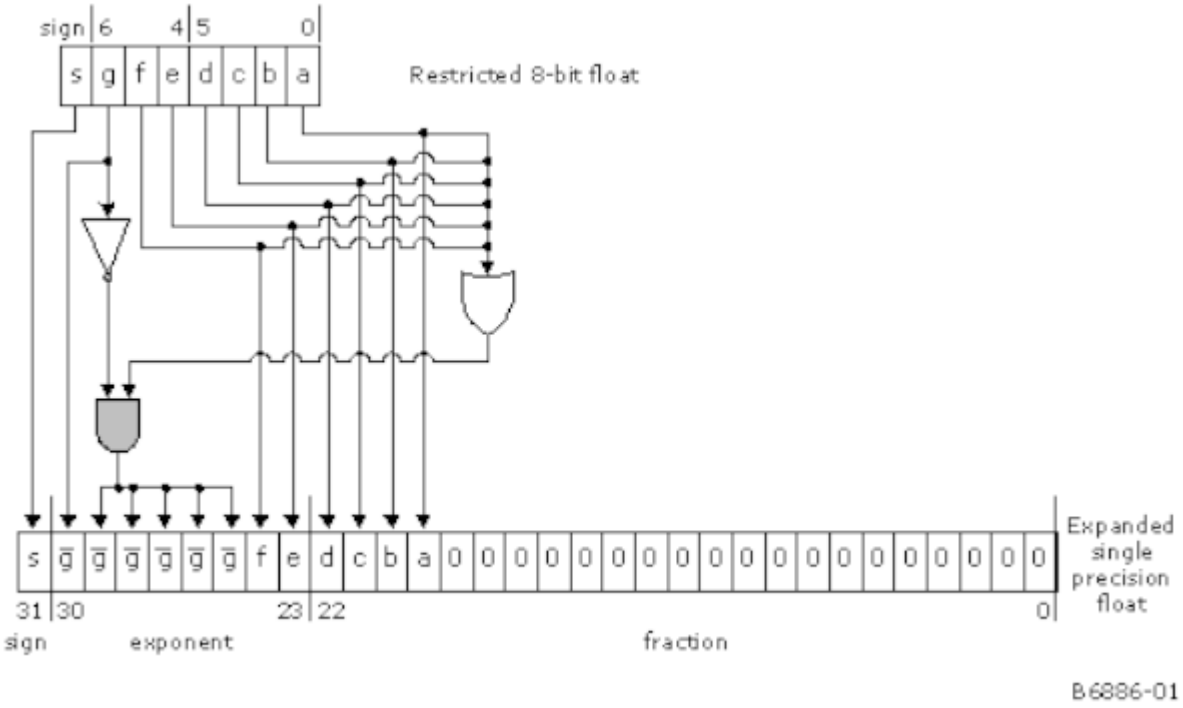


2.2.5 Packed Restricted Float Vector

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 numeric 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.

The following figure 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.

Conversion from a Restricted 8-bit Float to a Single-Precision Float



The following table 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.

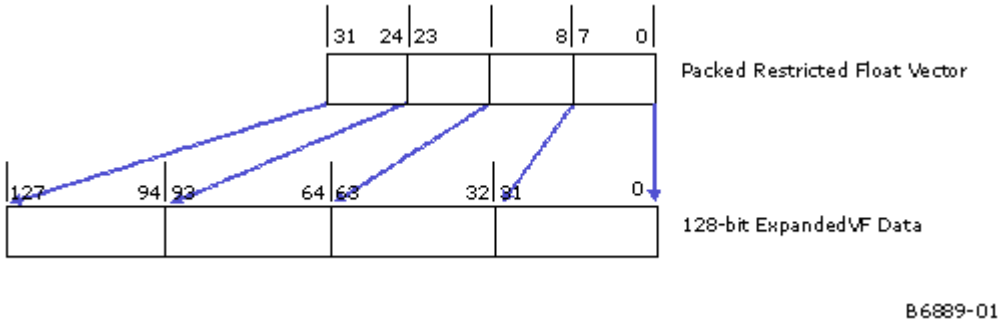
Examples of Restricted 8-bit Float Numbers

Class	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]	Extended 8-bit Exponent	Floating Number in Decimal
Positive Normalized Float	0x70-0x7F	0	111	0000 … 1111	1000 0011	16 … 31
	0x60-0x6F	0	110	0000 … 1111	1000 0010	8 … 15.5
	0x50-0x5F	0	101	0000 … 1111	1000 0001	4 … 7.75
	0x40-0x4F	0	100	0000 … 1111	1000 0000	2 … 3.875
	0x30-0x3F	0	011	0000 … 1111	0111 1111	1 … 1.9375
	0x20-0x2F	0	010	0000 … 1111	0111 1110	0.5 … 0.96875
	0x10-0x1F	0	001	0000 … 1111	0111 1101	0.25 … 0.484375
	0x01-0x0F	0	000	0001 … 1111	0111 1100	0.125 … 0.2421875
	0x00	0	000	0000	0000 0000	0 (+zero)
Negative Normalized Float	0xF0-0xFF	1	111	0000 … 1111	1000 0011	-16 … -31
	0xE0-0xEF	1	110	0000 … 1111	1000 0010	-8 … -15.5
	0xD0-0xDF	1	101	0000 … 1111	1000 0001	-4 … -7.75
	0xC0-0xCF	1	100	0000 … 1111	1000 0000	-2 … -3.875
	0xB0-0xBF	1	011	0000 … 1111	0111 1111	-1 … -1.9375
	0xA0-0xAF	1	010	0000 … 1111	0111 1110	-0.5 … -0.96875
	0x90-0x9F	1	001	0000 … 1111	0111 1101	-0.25 … -0.484375

Class	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]	Extended 8-bit Exponent	Floating Number in Decimal
	0x81-0x8F	1	000	0001 … 1111	0111 1100	-0.125 … -0.2421875
	0x80	1	000	0000	0000 0000	-0 (-zero)

The following figure shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The shorthand format notation for a packed signed half-byte vector is VF.



2.3 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.

2.3.1 IEEE Floating Point Mode

2.3.1.1 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, GT, 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, GT, 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.

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

For a result that cannot be represented precisely by the floating point format, the EU uses rounding to nearest or even to produce a result that is within 0.5 Unit-Last-Place(0.5 ULP) of the infinitely precise result.

The rounding mode is specified by the Rounding Mode field in the Control Register.

The EU can report floating point overflow and NaN into conditional flags. Hewever, there is no support for floating point exceptions, status bits, or traps.

] handle denorms as follows:

- Single precision (F, Float) denorms are flushed to sign-preserved zero on input and output of any floating-point mathematical operation.
- Double precision (DF, Double Float) denorms are kept or flushed in mathematical operations based on the Double Precision Denorm Mode in the Control Register.
- Denorms are not flushed for format conversions, irrespective of any denorm mode.
- Denorms are not flushed for raw *mov* operations. For information about raw *mov* operations, refer to the **Description** in *mov – Move*.

- Input denorms are not flushed for half precision to single precision floating-point conversion.

Other information regarding floating-point behaviors:

- NaN input to an operation 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 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 for rules on converting to and from float representations.

2.3.1.3 Comparison of Floating Point Numbers

The following tables 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 (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. When comparing two double float numbers, the result can be T or F depending on the values and the denorm mode (enabled/disabled). The corresponding fields in the following tables are marked T/F*.

Results of “Greater-Than” Comparison – CMP.

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

Results of "Less-Than" Comparison – CMP.L

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

Results of "Equal-To" Comparison – CMP.E

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

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	T/F*	T/F*	T/F*	T/F*	T	T	T
-0	T	T	T/F*	FALSE	FALSE	T/F*	T	T	T
+0	T	T	T/F*	FALSE	FALSE	T/F*	T	T	T
+denorm	T	T	T/F*	T/F*	T/F*	T/F*	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

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	F
-Fin	F	T/F	T	T	T	T	T	T	F
-denorm	F	F	T/F*	T/F*	T/F*	T/F*	T	T	F

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

Results of "Greater-Than or Equal-To" Comparison – CMP.GE

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

2.3.1.4 Min/Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in the 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 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 source1.

Min and Max is supported by conditional select.

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

The following tables 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'.

2.3.2 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., +fmax. The fmax has an exponent of 0xFE and a mantissa of all one's, which is the same for IEEE floating point mode.

Note that this mode is applicable ONLY to Single Precision Float datatype.

This also implies that ALT mode is not supported when Single precision datatype is involved in format conversion to double precision of half precision.

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 +/- fmax, 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.

Alternative Floating Point Mode shows the support of these differences in various hardware units.

Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	Sampler	RC
Any +/- INF result flushed to +/- fmax	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

Alternative Floating Point Mode shows some of the desired or recommended alternative floating point mode behaviors that do not have hardware design impact. The reasons of not needing special hardware support for these items are also provided. This is based on the compliance requirement that can be found in the DirectX 9 specification: "**Handling of NaNs, Infs, and denorms is undefined. Applications should not pass in such values into the graphics pipeline.**"

Dismissed legacy behaviors

Suggested IEEE-754 Deviations	Reason for Dismiss
Mov forces (+/-)INF to (+/-)fmax	(+/-)INF is never present as input
(+/-)INF – (+/-)INF = +/- fmax 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 fmax	N/A, as it is already covered by the general rule "Any +/- INF result flushed to +/- fmax"
(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

2.4 Type Conversion

2.4.1 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

Input Format	Output Format					
F	UD	D	UW	W	UB	B
+/- 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

2.4.2 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 sign 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.

2.4.3 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.

2.4.4 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, by default a “round to nearest even” is performed.

2.4.5 Double Precision Float to Single Precision Float

Converting a double precision floating-point number to a single precision floating-point number uses the round to zero rounding mode.

Double Precision Float	Single Precision Float
-inf	-inf
-finite	-finite/-denorm/-0

Double Precision Float	Single Precision Float
-denorm	-0
-0	-0
+0	+0
+denorm	+0
+finite	+finite/+denorm/+0
+inf	+inf
NaN	NaN

The upper Dword of every Qword will be written with undefined value when converting DF to F.

2.4.6 Single Precision Float to Double Precision Float

Converting a single precision floating-point number to a double precision floating-point number will produce a precise representation of the input.

Single Precision Float	Double Precision Float
-inf	-inf
-finite	-finite
-denorm	-finite
-0	-0
+0	+0
+denorm	+finite
+finite	+finite
+inf	+inf
NaN	NaN

3. Execution Environment

3.1 EU Overview

The GEN instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Support for 3D graphics API (Application Programming Interface) Shader instructions is mostly native, meaning that GEN efficiently executes Shader programs. Depending on 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 that accelerate media applications. The following feature list summarizes the GEN instruction set architecture:

- SIMD (single instruction multiple data) instructions. The maximum number of data elements per instruction depends on the data type.
- SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- Instruction level variable-width SIMD execution.
- Conditional SIMD execution via destination mask, predication, and execution mask.
- Instruction compaction.
- An instruction may executed in multiple cycles over a SIMD execution pipeline.
- Most GEN instructions have three operands. Some instructions have additional implied source or destination operands. Some instructions have explicit dual destinations.
- Region-based register addressing.
- Direct or indirect (indexed) register addressing.
- Scalar or 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).

CoIssue/Dual Issue:

The Gen7 generation of EU allows two instructions to be issued at the same time (sometimes referred to as “dual-issue” or more generally “co-issue”). The two instructions issued are always from different threads. The terms “FPU Pipe” and “EM Pipe” are the terms used when refering to the two simultaneous pipes. The Gen7 implementation dual-issue capability is limited to only the most popular instructions and source operand modes. Later generations of EU expand on this concept to allow more operations.

Description:

Opcodes: add, mov, mad, mul, cmp

- Datatype: single precision floats.
- Accessmode:
  - Align1:
    - No Scattering or Gathering data. This means data in source and destination registers are aligned and packed (data is contiguous in a register).

```
//Example:
```

```
// allowed, data is contiguous and source and destination regioning
map one to one.

mov (8) r10.0:f  r11.0<8;8,1>:f

// not allowed, data from source is strided and requires gathering to
write to destination

mov (8) r10.0:f  r11.0<4;4,2>:f

// not allowed, data from source is contiguous but not aligned with
destination. Destination register requires scattering

mov (8) r10.0<2>:w r11.0<8;8,1>:w

//not allowed, data from source is contiguous but destination is not
aligned to source

mov (8) r10.1:f r11.0<4;4,1>:f

// allowed. Source and destination have stride but are aligned

mov (4) r10.1:f r11.1<4;4,1>:f
```

        - A single precision float scalar is allowed.
    - Align16

- Addressmode: Direct Addressing
- Register File: GRF/NULL. No access to Accumulator.
- Condition modifiers supported only for cmp.

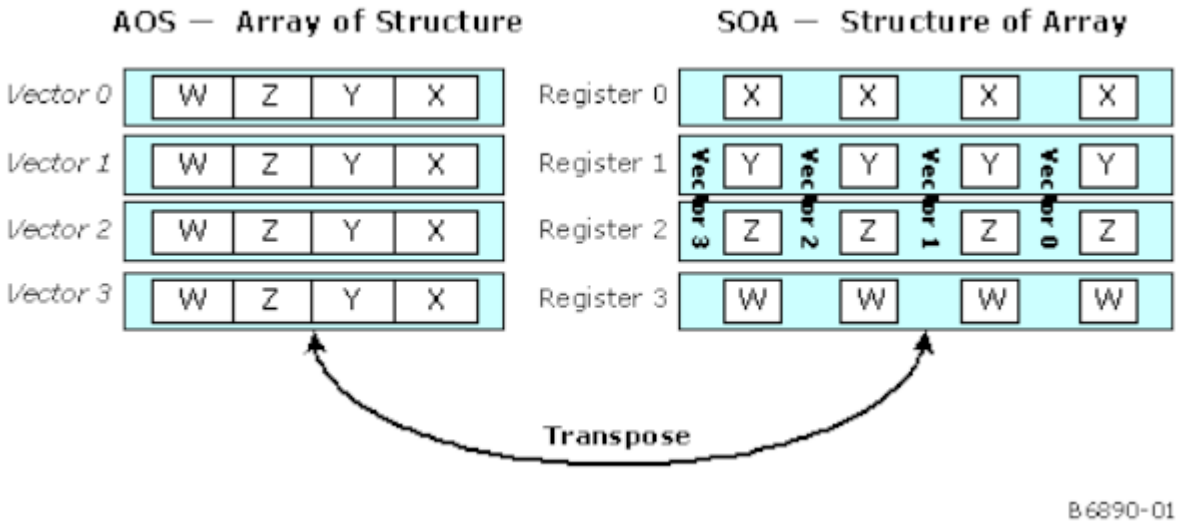
3.2 Primary Usage Models

In describing the usage models of the GEN instruction set, the following sections forward reference terminology, syntax, and instructions described later in this specification. For clarity reasons, not all forward references are explained at the point of reference. See the Instruction Set Summarychapter for information about instruction fields and syntax.

3.2.1 AOS and SOA Data Structures

With the Align1 and Align16 access modes, the GEN instruction set provides effective SIMD computation whether data is arranged in array of structures (AOS) form or in structure of arrays (SOA) form. The AOS and SOA data structures are illustrated by the examples in *AOS and SOA Data Structures*. The example shows two different ways of storing four vectors in four SIMD registers. For simplicity, the data vector and the 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. The two structures can be related by a matrix transpose operation.

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 form and use SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA form and use SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA form, and occasionally in AOS form with possibly mixed modes of operation that uses 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 is used in the following sections to illustrate various operation modes:

```
add dst.xyz src0.yxzw src1.zwxy
```

This example is a SIMD instruction that takes two source operands src0 and src1, adds them, and stores the result 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 swizzles (.yxzw for src0 and .zwxy for src1) and a destination mask (.xyz). Please refer to the programming specifications of 3D graphics API Shader instructions for more details.

A general register has 256 bits, which can store 8 floating point data elements. For 3D graphics, the mode of operation is (loosely) termed after the data structure as SIMD*m*x*n*, where *m* is the size of the vector and *n* is the number of concurrent program flows executed in SIMD.

Execution with AOS data structures:

- **SIMD4** (short for SIMD4x1) indicates that a SIMD instruction operates on 4-element vectors storedin registers. There is one program flow.

- **SIMD4x2** indicates that a SIMD instruction operates on a pair of 4-element vectors in registers. 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, mostly uses:

- **SIMD8** (short for SIMD1x8) indicates a SIMD instruction based on the SOA data structure where one register contains one data element (the same one) for each of 8 vectors. Effectively, there are 8 concurrent program flows.
- **SIMD16** (short for SIMD1x16) indicates that a SIMD instruction operates on a pair of registers that contain one data element (the same one) for each of 16 vectors. SIMD16 has 16 concurrent program flows.

3.2.2 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}

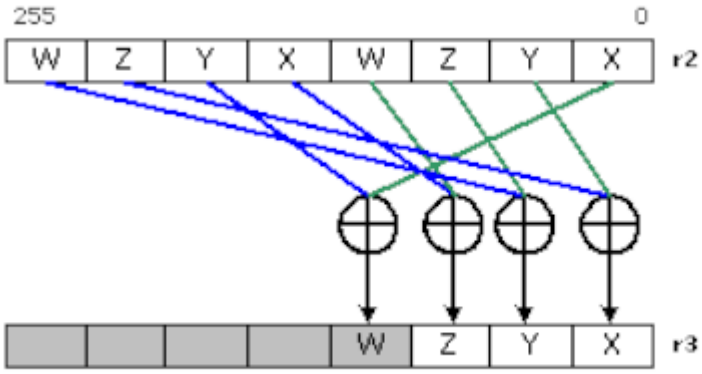
Without diving too much into the syntax definition of a GEN instruction, it is clear that a GEN instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar 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 the SIMD8 Mode of Operation section. 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 the following figure. 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).

A SIMD4 Example



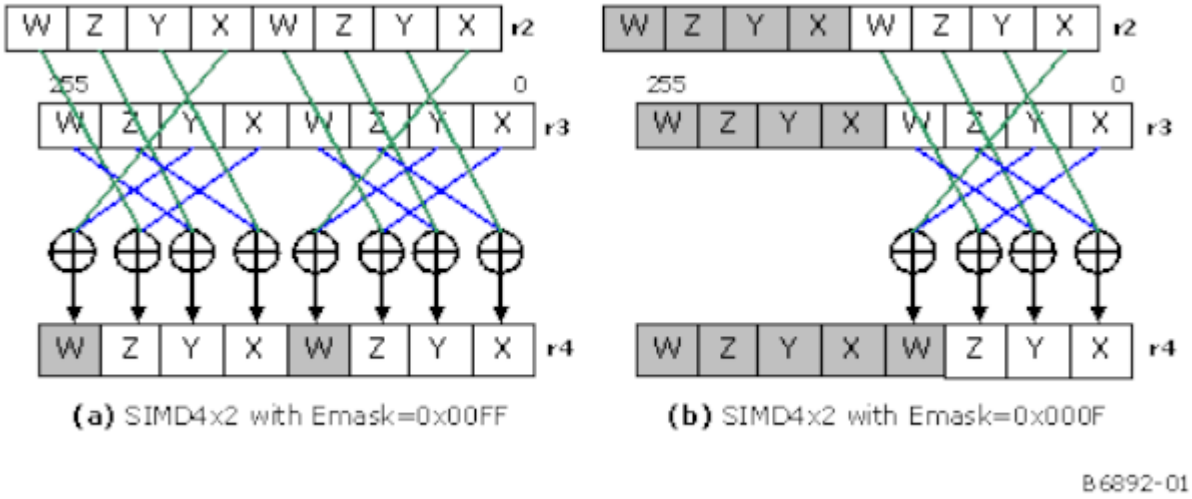
3.2.3 SIMD4x2 Mode of Operation

In this mode, two corresponding vectors from the two program flows fill a GEN 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 *SIMD4x2 Mode of Operation* (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 *SIMD4x2 Mode of Operation* (b).

SIMD4x2 Examples with Different Emasks



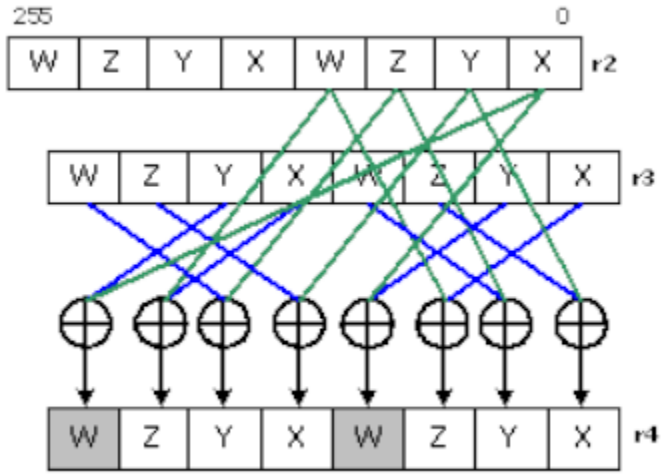
The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3

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 *SIMD4x2 Mode of Operation*.

A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows



3.2.4 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 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

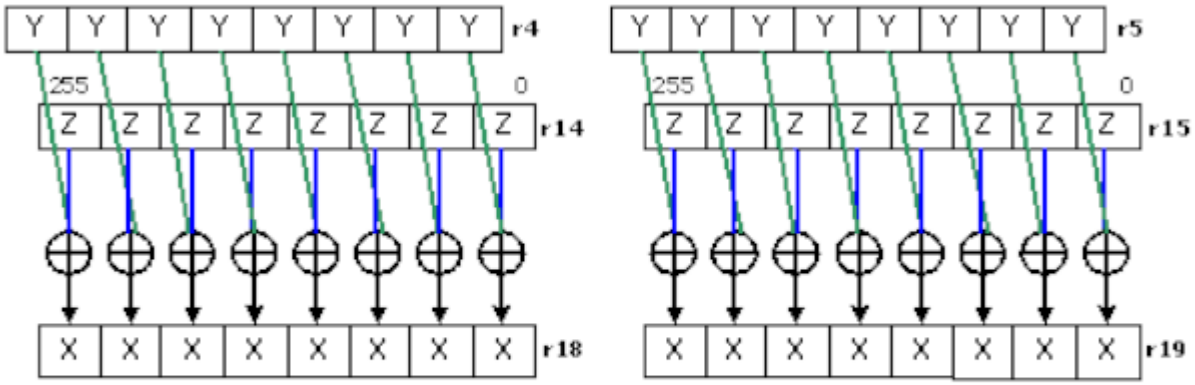
add (16) r22<1>:f r8<8;8,1>:f r10<8;8,1>:f // dst.z = src0.w + src1.x

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 *SIMD16 Mode of Operation*.

With more than one program flow, the above instructions are also subject to the 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.

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
```

B6894-01

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:

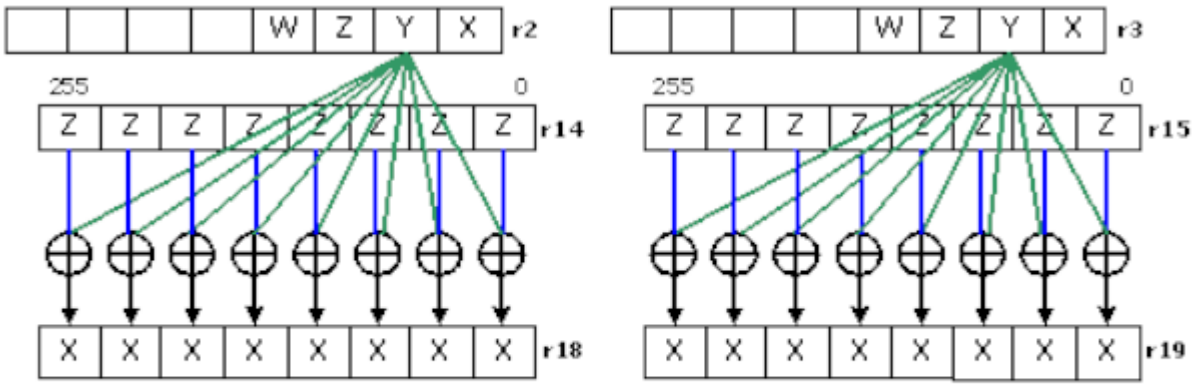
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr}// dst.x = src0.y + src1.z

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 *SIMD16 Mode of Operation*.

Another SIMD16 Example with an AOS Shared Constant




```
Add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr}  // dst.x=src0.y+src1.z
```

B6895-01

3.2.5 SIMD8 Mode of Operation

Each compressed instruction has two corresponding native instructions. Taking the example instruction shown in *SIMD16 Mode of Operation*, 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.

3.3 Registers and Register Regions

3.3.1 Register Files

GEN registers are grouped into different name spaces called register files. There are two register files, the General Register File and the Architecture Register File. A third encoding of some register file instruction fields indicates immediate operands within instructions rather than a register file.

- General Register File (GRF): The GRF contains general-purpose read-write registers.
- Architecture Register File (ARF): The ARF contains all architectural registers defined for specific purposes, including address registers (*a#*), accumulators (*acc#*), flags (*f#*), notification count (*n#*), instruction pointer (*ip*), null register (*null*), etc.
- Immediate: Certain instructions can take immediate source operands. A distinct register file field encoding indicates an immediate operand.

Each thread executed in an EU has its own thread context, a dedicated register space that is not shared between threads, whether executing on a common EU or on a different EU. In the rest of the chapters in this volume, register access is relative to a given thread.

3.3.2 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
Indexable?	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.

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. Each thread has a dedicated space of 128 register, 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 registers. A destination operand must be contained within one 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.

3.3.3 ARF Registers

3.3.3.1 ARF Registers Overview

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

Summary of Architecture Registers

Register Type (RegNum [7:4])	Register Name	Register Count	Description
0000b	*null*	1	Null register
0001b	*a0.#*	1	Address register
0010b	*acc#*	2	Accumulator register
0011b	*f#.#*	2	Flag register
0101b	*Reserved*		Reserved
0110b	*Reserved*		Reserved
0111b	*sr0.#*	1	State register
1000b	*cr0.#*	1	Control register
1001b	*n#*	2	Notification Count register
1010b	*ip*	1	Instruction Pointer register
1011b	*tdr*	1	Thread Dependency register
1100b	*tm0*	1	TimeStamp register
1101b	*Reserved*		Reserved

Register Type (RegNum [7:4])	Register Name	Register Count	Description
1110b	*Reserved*		Reserved

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

In general an ARF register can be dst (destination) or src0 (source 0, first source operand) for an instruction. Depending on the register and the instruction, other restrictions may apply.

3.3.3.2 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. 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 generate the destination write mask at byte level.

A special restriction on region-based addressing for ARF is that the register region cannot cross a register boundary.

Subregister fields of an ARF register may not all be populated (indicated by the subregister being indicated as reserved). Writes to unpopulated subregisters are dropped; there are no side effect. Reads from unpopulated subregisters, if not specified, return unpredictable data.

Some ARF registers are read-only. Writes to read-only ARF registers are dropped and there are no side effects.

3.3.3.3 Null Register

Null Register Summary

Attribute	Value
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
Indexable?	No

The null register is a special encoding for an operand that does not have a physical mapping. It is primarily used in instructions to indicate non-existent operands. Writing to the null register has no side effect. Reading from the null register returns an undefined result.

The null register can be used where a source operand is absent. For example, for a single source operand instruction such as MOV or 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 registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, updates to the selected flag register also occur. 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 no write completion acknowledgement is 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 the *send* instruction for details.

3.3.3.4 Address Register

Address Register Summary

Attribute	Value
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
Indexable?	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.

Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000b = a0 All other encodings are reserved.	When register a0 or subregisters in a0 are 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. 00000b = **a0.0:uw** 00010b = **a0.1:uw** 00100b = **a0.2:uw** 00110b = **a0.3:uw** 01000b = **a0.4:uw**

RegNum[3:0]	SubRegNum[4:0]
	01010b = **a0.5:uw** 01100b = **a0.6:uw** 01110b = **a0.7:uw** All other encodings are reserved. However, when register a0 or subregisters in a0 are explicit source and/or destination registers, other data types are allowed as long as the register region falls in the 128-bit range.

Address Register Fields

DWord	Bits	Description
7	31:16	**Address subregister a0.15:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.14:uw**. Follows the same format as **a0.2**.
6	31:16	**Address subregister a0.13:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.12:uw**. Follows the same format as **a0.2**.
5	31:16	**Address subregister a0.11:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.10:uw**. Follows the same format as **a0.2**.
4	31:16	**Address subregister a0.9:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.8:uw**. Follows the same format as **a0.2**.
3	31:16	**Address subregister a0.7:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.6:uw**. Follows the same format as **a0.2**.
2	31:16	**Address subregister a0.5:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.4:uw**. Follows the same format as **a0.2**.
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 a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *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 a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *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 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 next 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 individually or as a group. When accessed as a 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 are 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 are 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 the 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 high 4 bits for each element are dropped; when they are the source, hardware inserts zero to the high 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 stalls 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 stalls any instruction sourcing other subregisters until the write retires.

3.3.3.5 Accumulator Registers

Accumulator Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	2
Default Value:	None
Normal Access:	RW

Accumulator registers can be accessed either as explicit or implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 DWords or 16 Words of data elements. However, as described in the Implementation Precision Restriction notes below, each data element may have higher precision with added guard bits not indicated by the numeric data type.

Accumulator capabilities vary by data type, not just data size, as described in the Accumulator Channel Precision table below. For example, D and F are both 32-bit data types, but differ in accumulator support.

See the Accumulator Restrictions section for information about additional general accumulator restrictions and also accumulator restrictions for specific instructions.

There are two accumulator registers, acc0 and acc1.

Register and Subregister Numbers for Accumulator Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = acc0 0001b = acc1 All other encodings are reserved.	Reserved: MBZ.

- Accumulators are updated implicitly only if the AccWrCtrl bit is set in the instruction. The Accumulator Disable bit in control register cr0.0 allows software to disable the use of AccWrCtrl for implicit accumulator updates. The write enable in word granularity for the instruction is used to update the accumulator. Data in disabled channels is not updated.
- When an accumulator register is an implicit source or destination operand, hardware always uses acc0 by default and also uses acc1 if the execution size exceeds the number of elements in acc0. When implicit access to acc1 is required, QtrCtrl is used. Note that QtrCtrl can be used only if acc1 is accessible for a given data type. If acc1 is not accessible for a given data type, QtrCtrl defaults to acc0.

  acc0 and acc1 are supported for single-precision Float (F) only. Use QtrCtrl of Q2 or Q4 to access acc1.

  Examples:

```
// Updates acc0 and acc1 because it is SIMD16:
add (16) r10:f r11:f r12:f {AccWrEn}
// Updates acc0 because it is SIMD8:
add (8) r10:f r11:f r12:f {AccWrEn}
// Updates acc1. Implicit access to acc1 using QtrCtrl:
add (8) r10:f r11:f r12:f {AccWrEn, Q2}
// Updates acc1 for Half Floats using QtrCtrl:
add (16) r10:hf r11:hf r12:hf {AccWrEn, H2}
```

- It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). The result of such an instruction is unpredictable.
- Some processor generations or steppings limit SIMD16 Float operations, as follows:
  - SIMD16 execution on Floats is not allowed when an accumulator is an explicit source or destination operand.
- Accumulator registers may be accessed explicitly as src0 operands only.
- Swizzling is not allowed when an accumulator is used as an implicit source or an explicit source in an instruction.
- For any DWord operation, including DWord multiply, accumulator can store up to 8 channels of data, with only acc0 supported.
- When an accumulator register is an explicit destination, it follows the rules of a destination register. If an accumulator is an explicit source operand, its register region must match that of the destination register with the exception described below.

Implementation Precision Restriction: As there are only 64 bits per channel in DWord mode (D and UD), it is sufficient to store the 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 result is then 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 the multiplication result of two Word operands with and without source modifier as the result is up to 33 bits. Integers are stored in accumulator in 2's complement 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 a risk of overflowing. When overflow occurs, only modular addition can generate a 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 UD (FFFFFFFFh) with D (FFFFFFFFh) results in 1FFFFFFFEh. 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.

Accumulator Channel Precision

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
DF	acc0	4	64	When accumulator is used for Double Float, it has the exact same precision as any GRF register.
F	acc0/acc1	8	32	When accumulator is used for Float, it has the exact same precision as any GRF register.
D (UD)	acc0	8	33/64	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 complement 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 of *mul*, *mach*, and *shr* (or *mov*).
W (UW)	acc0	16	33	When the internal execution data type is word integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's complement form with 33 bits total. This supports single instruction multiplication of two word sources in W and/or UW format.
B (UB)	N/A	N/A	N/A	Not supported data type.

3.3.3.6 Flag Register

Flag Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0011b
Number of Registers:	2
Default Value:	None
Normal Access:	RW
Elements:	2
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Word
Write Mask Granularity:	Word

Attribute	Value
SecHalf Control?	Yes
Indexable?	No

There are two flag registers, f0 and f1.

Each flag register contains two 16-bit subregisters. Each flag bit corresponds to a data channel. Predication uses flag values to enable or disable channels. Conditional modifiers assign flag values. If an instruction uses both predication and conditional modifiers, both features use the same flag register or subregisters.

Flags can be split to halfs, quarters, or eighths using the QtrCtrl and NibCtrl instruction fields. Those fields affect the selection of flags for predication and conditional modifiers, but do not affect reading or writing flags as explicit instruction operands.

The values held in the individual bits of a flag register are the result of the most recent instruction with a conditional modifier and specifying that flag register. For example:

```
add.nz.f0.0 ...
```

Updates flag subregister f0.0 with the per-channel results of the not zero condition.

The flag register has per-bit write enables. When being updated as the secondary destination associated with a conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. Other bits in the flag subregister are unchanged.

Flag registers and subregisters can also be explicit source or destination operands.

The *sel* instruction does not update flags.

Note: When branching instructions are predicated, branching is evaluated on all channels enabled at dispatch. This means, the appropriate number of flag register bits must be initialized or used in predication depending on the execution mask (EMask). Uninitalized flags may result in undesired branching. For example, if using DMask as EMask and if all 32 channels of DMask are enabled, a SIMD8 kernel must initialize unused flag bits so that predication on branching is evaluated correctly.

Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000b = f0:ud	00000b = f*n*.0:uw
0001b = f1:ud	00010b = f*n*.1:uw
Other encodings are reserved.	Other encodings are reserved.

Reference an entire flag register as f0:ud or f1:ud. Reference the flag subregisters as f0.0:uw, f0.1:uw, f1.0:uw, and f1.1:uw.

3.3.3.7 State Register

State Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0111b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW

Attribute	Value
Elements:	4
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Byte
Write Mask Granularity:	N/A
SecHalf Control?	No
Indexable?	No

Register and Subregister Numbers for State Register

RegNum[3:0]	SubRegNum[4:0]
0000b = sr0 All other encodings are reserved.	Valid encoding range: 00000b – 01100b All other encodings are reserved.

State Register Fields

DWord	Bits	Description
0 (sr0.0:ud)	31:28	**Reserved**. MBZ.
	27:24	**FFID** (Fixed Function Identifier). Specifies which fixed function unit generates the current thread. This field is set at thread dispatch and is forwarded on the message bus for all out-going messages from this thread.
	23	**Priority Class**. This field, when set, indicates the thread belongs to the high priority class, which has higher scheduling priority over any thread with this field cleared. The priority field below determines the relative priority within the same priority class. This field is initialized by the thread dispatcher at thread dispatch time and stays unchanged throughout the life span of the thread. This field is forwarded on the message bus to the message bus arbiter for all out-going messages. It serves as a priority hint for the target shared function. See the Shared Function chapters for whether and how a shared function uses this priority hint. 0 = Low priority class. 1 = High priority class.
	22:19	**Reserved**. MBZ.
	18:16	**Priority**. This field is the relative aging priority of the thread. This field indicates the 'age' of this thread relative to other threads within the EU. No two threads in the same EU can have the same priority number (independent of the priority class value). Within the same priority class, an older thread (with a larger priority number) has higher schedule priority over a younger thread. This field is set to zero at a thread's dispatch. During a thread's run time, this field may or may not be incremented when a new thread is dispatched to the same EU. It is only incremented when another thread's priority number is incremented and reaches the same value. For example, if currently there is a thread with priority 0 on an EU, then dispatching a new thread to that EU causes the old thread's priority number to increment to 1. However, if the active thread (assuming for simplicity that there is only one) on an EU has a priority number 1 (or 2 or 3), then dispatching a new thread to this EU does not change the old thread's priority number. As threads on an EU may terminate in arbitrary order, the exact number for a thread depends on the dynamic execution of threads.
	15:8	**:** [15:13] Reserved. MBZ.

DWord	Bits	Description
		[12] HSID. HalfSlice Identifier for the EU. [11:8] EUID[3:0]. Execution Unit Identifier. The MSB of this field is the RowID.
	7:3	**Reserved**. MBZ.
	2:0	**TID** (The thread identifier). Specifies the thread slot that the current thread is assigned to. This field is set at thread dispatch.
1 (sr0.1:ud)	31:24	**FFTID** (Fixed Function Thread ID). There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.
	23:0	**Reserved**. MBZ.
1 (sr0.1:ud)	31:23	
2 (sr0.2:ud)	31:0	**Dispatch Mask** (DMask). This 32-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register. Format: U32
3 (sr0.3:ud)	31:0	**Vector Mask** (VMask). This 32-bit field contains, for each 4-bit group, the OR of the corresponding 4-bit group in the dispatch mask. This field is used by hardware to initialize the mask register. Format: U32

3.3.3.8 Control Register

Control Register Summary

Attribute	Value
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
Indexable?	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 arrangement allows the System Routine to use the *lzd* instruction to find the high priority exceptions and handle them first. As each exception is mapped to a single bit, another exception priority order may be implemented by software. The System Routine may choose to handle one exception at a time, by handling the exception detected

by an *lzd*instruction and returning to the 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 causes hardware to jump to the System Routine or not. Exception status and control bits (bits 31:16 in *cr0.1:ud*) indicate which exceptions have occurred, and are used by the system routine to clear the exception. Even if a given exception is disabled, the corresponding exception status and control bit still reflects its 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. Values written to this subregister are dropped; the result of reading from this subregister is unpredictable.

Fields in Control registers also reference 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 respond to exception events. Upon an exception, hardware performs some bookkeeping (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 *cr0*, which causes the hardware to load AIP to IP register. See the STATE_SIP command for how to set SIP.

Register and Subregister Numbers for Control Register

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

Control Register Fields

DWord	Bits	Description
0	31	**Master Exception State and Control.** This bit 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 bit to 1 and switches to SIP. Writing 1 to this bit has no effect. Writing 0 to this bit also has no effect if the previous value is 0. In both cases, the bit keeps the previous value. If the previous value of this bit 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 bit then stays as 0. This bit is initialized to 0. 0 = The thread is in normal state. 1 = The thread is in exception state.
	30:16	**Reserved**. MBZ.
	15	**Breakpoint Suppress.** This bit specifies whether breakpoint exception is suppressed or not. This bit is normally set by software and cleared by hardware. If Master Exception Status and Control bit is 1, this bit is ignored by hardware. If Master Exception Status and Control bit is 0 (i.e. not in System Routine) and Breakpoint is enabled: If this bit is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This bit is provided to prevent infinite loops of jumping to the System Routine on a breakpoint

DWord	Bits	Description
		condition. The System Routine must set this bit (and also clear the corresponding status and control bit) before returning to the application thread. This bit has no effect when Breakpoint Enable bits are cleared. This bit is initialized to 0. 0 = Breakpoint exception is not suppressed. 1 = Breakpoint exception is suppressed.
	14:10	**Reserved**. MBZ.
	7	Reserved.
	6	**Double Precision Denorm Mode**. This bit determines how denormal numbers are handled for the DF (Double Float) type. It is initialized by Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results.
	5:4	**Rounding Mode**. This field specifies the FPU rounding mode. It is initialized by Thread Dispatch. 00b = Round to Nearest or Even (RTNE) 01b = Round Up, toward +inf (RU) 10b = Round Down, toward -inf (RD) 11b = Round Toward Zero (RTZ)
	3	**Vector Mask Enable (VME)**. This bit indicates DMask or Vmask should be used by EU for execution. This bit is set by the Thread Dispatch. 0: Use Dispatch 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 bit affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This bit is initialized by the Thread Dispatch. 0: Multiple Program Flows 1: Single Program Flow **Programming Restrictions:** Only H1/Q1/N1 are allowed in SPF mode. Power Optimization: If an entire shader does not do SIMD branching, the driver can set the SPF bit to 1 to save power in HW.
	1	**Accumulator Disable**. This bit controls the update of the accumulator by the instruction field AccWrCtrl. If this bit 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 bit has no effect if the accumulator is the explicit destination operand for an instruction. This bit 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 that implicitly update the accumulator. To use

DWord	Bits	Description
		such instructions in the System Routine, but still preserve the accumulator contents on returning 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 to 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 before 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 is required to preserve the content of this bit.
	0	**Single Precision Floating Point Mode (FP Mode)**. This bit specifies whether the current single-precision floating-point operation mode is IEEE mode (IEEE Standard 754) or the ALT (alternative mode). This bit does not affect the floating-point mode used for other floating-point data types. This bit is also forwarded on the message sideband for all out-going messages, for example, to control the floating-point mode of the Sampler. Software may modify this bit to dynamically switch between the two floating-point modes. This bit is initialized by Thread Dispatch. 0 = IEEE floating-point mode for the F (Float) type. 1 = ALT (alternative) floating-point mode for the F (Float) type.
	30	**External Halt Exception Status and Control**. This bit indicates the External Halt exception. It is set by EU hardware on receiving the broadcast External Halt signal. The System Routine should reset this bit before returning to an application routine to avoid infinite loops. This bit may be set or cleared by software. This bit is initialized to 0.
	29	**Software Exception Control**. This bit is the control bit for software exceptions. Setting this bit to 1 in an application routine causes an exception. Clearing this bit in an application routine has no effect. Upon entering the system routine, the hardware maintains this bit as 1 to signify a software exception. The System Routine should reset this bit before returning to an application routine. This bit may be set or cleared by software. This bit is initialized to 0.
	28	**Illegal Opcode Exception Status**. This bit, when set, indicates an illegal opcode exception. The exception handler 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 an application routine leaving this bit set, it doesn't cause any exception. This bit should not be set by software or left set by the system routine to avoid confusion. This bit is initialized to 0.
	27	**Stack Overflow Exception Status**. This bit when set, indicates a stack overflow exception. The exception handler routine normally does not return back to the application thread upon a stack overflow exception. Leaving this bit set has no effect on hardware; if system software adversely returns to an application routine leaving this bit set, it doesn't cause any exception. This bit should not be set by software or left set by the system routine to avoid confusion. This bit is initialized to 0.
	26:24	**Reserved**
	23:16	**Reserved**. MBZ.
	15	**Breakpoint Enable**. Specifies whether the breakpoint exception is enabled or not. This bit is initialized by the Thread Dispatcher. Format = ENABLED:

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>0: Disabled
1: Enabled</td></tr>
<tr><td></td><td>13</td><td>Software Exception Enable. This bit enables or disables the software exception. Enabling or disabling this bit may allow host software to turn on/off certain features (such as profiling) without changing the kernel program.

This bit is initialized by the Thread Dispatcher.

Format = ENABLED:
0: Disabled
1: Enabled</td></tr>
<tr><td></td><td>12</td><td>Illegal Opcode Exception Enable. This bit specifies whether the illegal opcode exception is enabled or not. The Illegal opcode exception includes illegal opcodes and undefined opcodes, caused by bad programs or run-time data corruption.

This bit is initialized by the Thread Dispatcher.

Software should normally assign this bit in the interface descriptor. Even though this mechanism is provided to disable the illegal opcode exception, it should be used with extreme caution.

Format = ENABLED:
0: Disabled
1: Enabled</td></tr>
<tr><td></td><td>11</td><td>Stack Overflow Exception Enable. This bit specifies whether the stack overflow exception is enabled or not. The stack overflow exception includes an overflow or an underflow in the stack space allocated for the thread.

This bit is initialized by the Thread Dispatcher.

Software should normally assign this bit in the interface descriptor.

Format = ENABLED:
0: Disabled
1: Enabled</td></tr>
<tr><td></td><td>10:0</td><td>Reserved. MBZ.</td></tr>
<tr><td>2

(cr0.2:ud)</td><td>31:3</td><td>Application IP (AIP). 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 Master Exception State and Control field to 1, which forces a switch to the System IP (SIP). The AIP register may contain either the pointer to the instruction that causes the exception or the one after (such as masked stack overflow/underflow exceptions). This is shown in the following table, where IP is the instruction that 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 the returning IP to be altered after an exception is handled.</td></tr>
</table>

DWord	Bits	Description
	2:0	**Reserved**. MBZ.

Implementation Restriction on Register Access: When the control register is used as an explicit source and/or destination, hardware does not 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 undefined results.

3.3.3.9 Notification Registers

Notification Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1001b
Number of Registers:	3
Default Value:	No
Normal Access:	RO (RW – Context save/restore only)
Elements:	3
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	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, except under context restore, and can be accessed in 32-bit granularity. Write access to this register is allowed only when context is restored.

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximum allowed number 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.

There is only one bit for the host-to-thread notification count in n1.

When directly accessed, this register is read-only. If the value is non zero, the only way to alter the value is to use the wait instruction to decrement the value until zero is reached. A wait instruction on a zero notification subregister causes the thread to stall, waiting for a notification signal from outside targeting the same subregister. See the wait instruction for details.

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

Register and Subregister Numbers for Notification Registers

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

Notification Register 0 Fields

DWord	Bits	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 the WAIT instruction for details. Format: U16

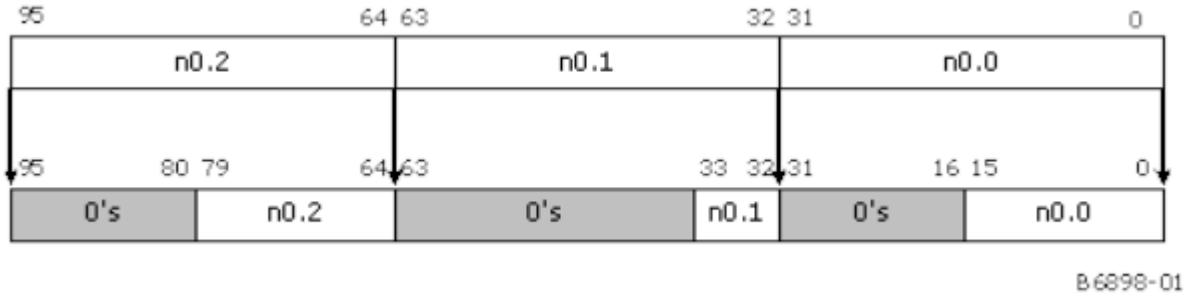
Notification Register 1 Fields

DWord	Bits	Description
0	31:1	**Reserved**. MBZ.

Notification Register 2 Fields

DWord	Bits	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 the WAIT instruction for details. Format: U16

Format of the Notification Register



3.3.3.10 IP Register

IP Register Summary

Attribute	Value
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
Indexable?	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 causes program flow to jump to the new address. When it is written, the 3 LSBs are dropped by hardware.

Register and Subregister Numbers for IP Register

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

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.

3.3.3.11 TDR Registers

TDR Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1011b
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
Indexable?	No

There are 8 thread dependency registers (tdr0.0:uw to 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, bits 8 to 14 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 are reset by 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 registers will clear the valid bit for the particular TDR if the write enable is true, the FFTID portion is strictly read only.

Register and Subregister Numbers for TDR Registers

RegNum[3:0]	SubRegNum[4:0]
1011b = tdr0 All other encodings are reserved.	00000b = **tdr0.0:uw** 00010b = **tdr0.1:uw** 00100b = **tdr0.2:uw** 00110b = **tdr0.3:uw** 01000b = **tdr0.4:uw** 01010b = **tdr0.5:uw** 01100b = **tdr0.6:uw** 01110b = **tdr0.7:uw** All other encodings are reserved.

TDR Registers Fields

DWord	Bits	Description
3	31	**Valid7**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
	15	**Valid6**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
2	31	**Valid5**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
	15	**Valid4**. This field indicates whether 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 that the current thread depends on. It can be

DWord	Bits	Description
		changed by the end of thread broadcasting messages. Format: U8
1	31	**Valid3**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
	15	**Valid2**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
0	31	**Valid1**. This field indicates whether the thread specified by FFTID1 is still in-flight.
	30:24	**Reserved**. MBZ
	23:16	**FFTID1**. This field is the FFTID of the third thread that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8
	15	**Valid0**. This field indicates whether 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 that the current thread depends on. It can be changed by the end of thread broadcasting messages. Format: U8

3.3.3.12 Performance Registers

Performance Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1100b
Number of Registers:	1
Default Value:	0h
Normal Access:	RO
Elements:	2
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Byte
Write Mask Granularity:	N/A
SecHalf Control?	No
Indexable?	No

Starting with, a block of ARF register space is allocated for per-thread performance information. Currently only a timestamp register is defined within this space, although it is anticipated that future performance-related information would be located here also.

Register and Subregister Numbers for Performance Registers

RegNum[3:0]	SubRegNum[4:0]
1100b = timestamp All other encodings are reserved.	Valid encoding range: 00000b – 00111b (in units of bytes) All other encodings are reserved.

Timestamp Register

This generation defines a new low latency timestamp source, “TM”, available as part of a thread's Architectural Register File (ARF). This is a is free running counter, 64b in size, and exposed to the ISA as individual 32b high ‘TmHigh’ and low ‘TmLow’ unsigned integer source operands. As part of the EU's register space, access to the timestamp has a low and deterministic latency and therefore can be used for intra-kernel high resolution performance profiling.

The TM counter is free running based on the EU’s clock and continues to increment across all time. Given a base EU clock frequency of 1.25 GHz and the counter’s 64b size, rollover of the lower 32b occurs approximately every 3.3 seconds, with the upper 32b value rollover measured as ~450 years. The TM count continues to increment during a thread's active/standby state transitions as well as context switches. It is read-only and not pre- or resettable under any software control, either kernel or driver, other than a full gfx reset. The 64b TM value is expected to be identical across all EUs of the system unless DOP clock gating is enabled.

The TM features provides a 1-bit indicator ‘TmEvent’ which identifies the occurrence of a time-impacting event such as context switch or frequency change since the last time any portion of the Timestamp register value was read by that thread. Software that uses the Timestamp capability should check this bit to identify when a relative time calculation may be suspect. To properly use this additional information, the instrumentation code should operate on the Timestamp register value as a whole (i.e. as an 8 dword register) so that the 64b time and this 1b value are captured simultaneously, as opposed to 32b portions, to eliminate a the chance of missing a TmEvent that might occur between accesses to 32b portions of this register.

Note: The Timestamp register is saved as part of thread state on context-save, but only ‘TmEvent’ is restored (and technically always restored to ‘1’ as a context switch had occurred).

Timestamp Register Fields

DWord	Bits	Description
7:3	31:0	**Undefined**.
2	31:29	**Undefined**.
	0	**TmEvent**. Indicates a discontinuous time-impacting event (e.g. context switch, frequency change) occurred since any portion of the Timestamp register was last read, thus making any relative duration calculation based on this counter suspect. This bit is reset at the time a new thread is loaded, and on each read of any portion of the ‘Timestamp’ register.
1	31:0	**TmHigh**. The upper 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32
0	31:0	**TmLow**. The lower 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32

3.3.4 Immediate

Two forms of immediate are provided as a source operand: scalar and vector.

The immediate field in a GEN instruction has 32 bits. For a word or an unsigned word immediate data, software must replicate the same 16-bit immediate value to both the lower word and the high word of the 32-bit immediate field in a GEN instruction.

For a scalar immediate, it can be of any of the specified numeric data types from a word to a dword. Byte and unsigned byte are not supported as the smallest internal type of the execution pipeline is word. These two numeric types are reserved for future extensions.

The immediate form of vector allows a constant vector to be in-lined in the instruction stream. Both integer and float immediate vectors are supported.

An immediate integer vector is denoted by type **v** or **uv** as *imm32:v* or *imm32:uv*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. *Refer to the Numeric DataType Section* for description of the packing of vector integers to a dword.

An immediate float vector is denoted by type **vf** as *imm32:vf*, where the 32-bit immediate field is partitioned into 4 8-bit subfields. *Refer to the Numeric DataType Section* for the description of the packing of vector floats to a dword.

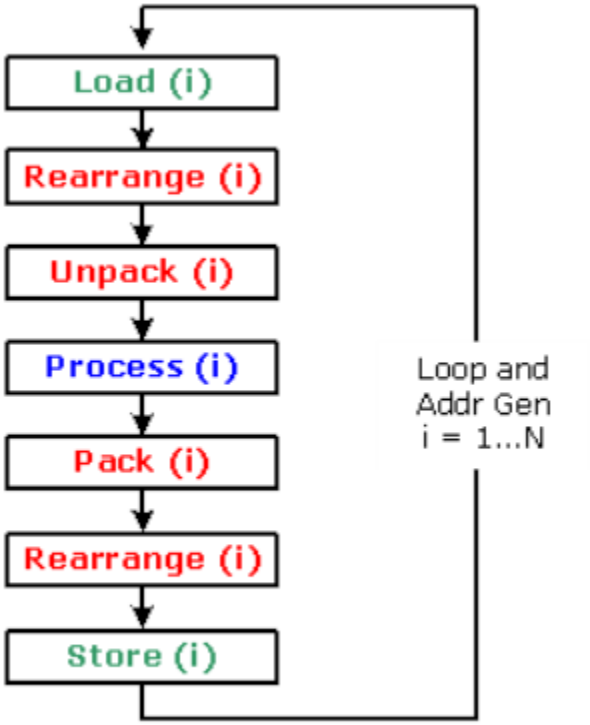
Restriction: *When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (**v**) and equivalent to a dword for an immediate float vector (**vf**).*

3.3.5 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.

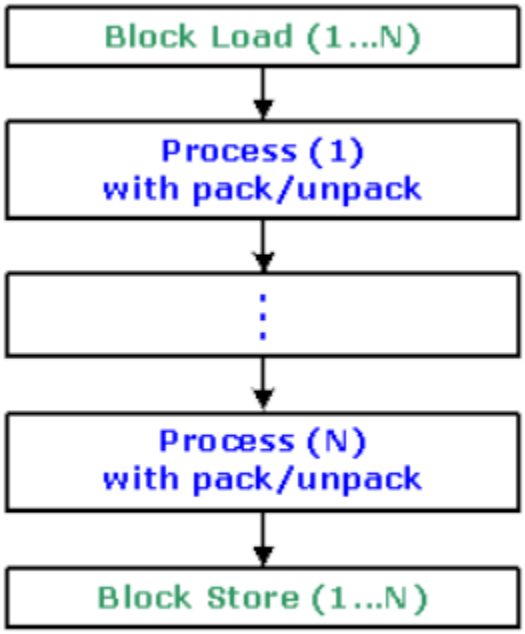
This can be illustrated by the example shown in *Region Parameters* and *Region Parameters*. As shown in *Region Parameters*, a sequence of SIMD instruction is executed on a conventional load/store based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first 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.

Conventional SIMD Instruction Sequence



B6899-01

GEN SIMD Instruction Sequence for the Same Program



B6900-01

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 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 the *Flag Registers* section.

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.

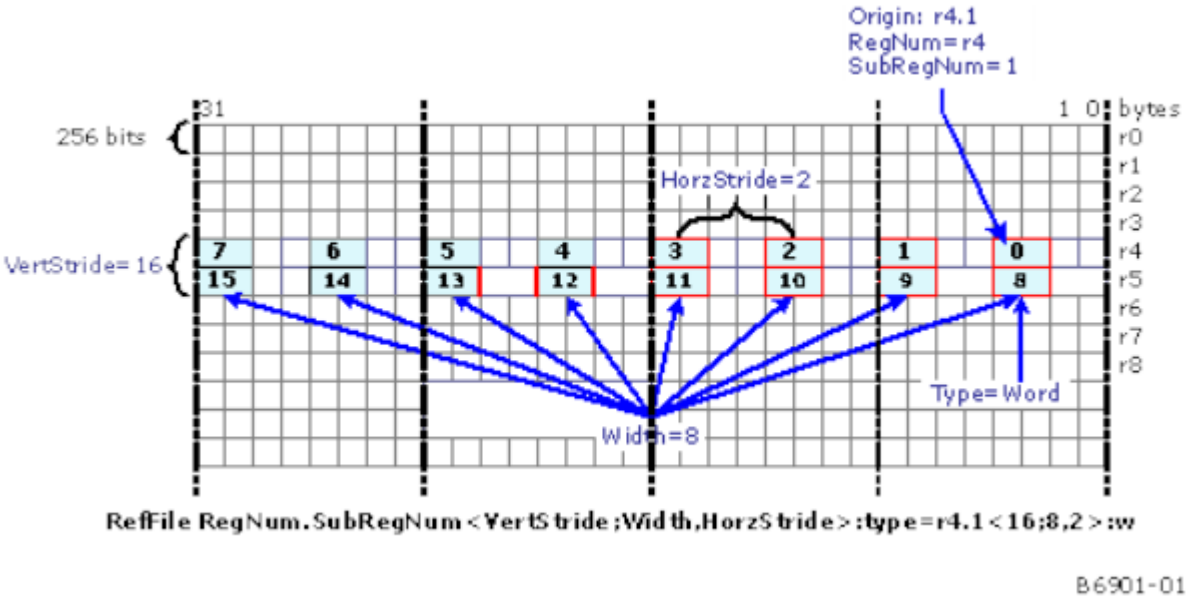
- Register Region Origin (*RegFile*, *RegNum* and *SubRegNum*): This set of parameters, including the register file, *RegFile*, the register number, *RegNum*, and the subregister number, *SubRegNum*, describes the register region origin, which is the location of the first data element of the operand. *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 numeric data type (float, word, byte, etc.) of the data elements. All data elements within a region must have the same type.

GRF register file consists of a sequence of 256-bit registers. When viewing the register file (GRF for example) as a sequence of 256-bit aligned registers, *RegNum* field provides the register number, thus for the name. *SubRegNum* provides the sub-field addressing within a 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.

Region Parameters 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.

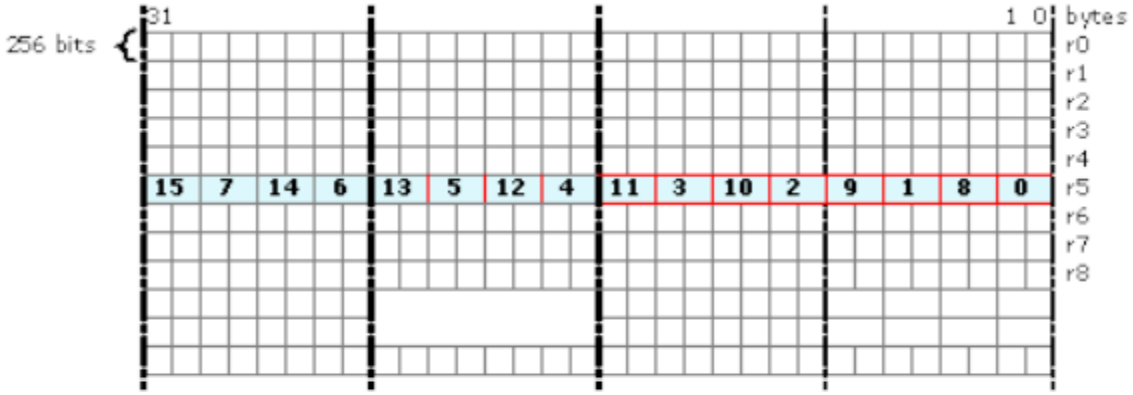
An example of a register region (*r4.1<16;8,2>:w*) with 16 elements



Region Parameters 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.

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



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 cross multiple physical GRF registers, 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 registers are not supported.

3.3.6 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.

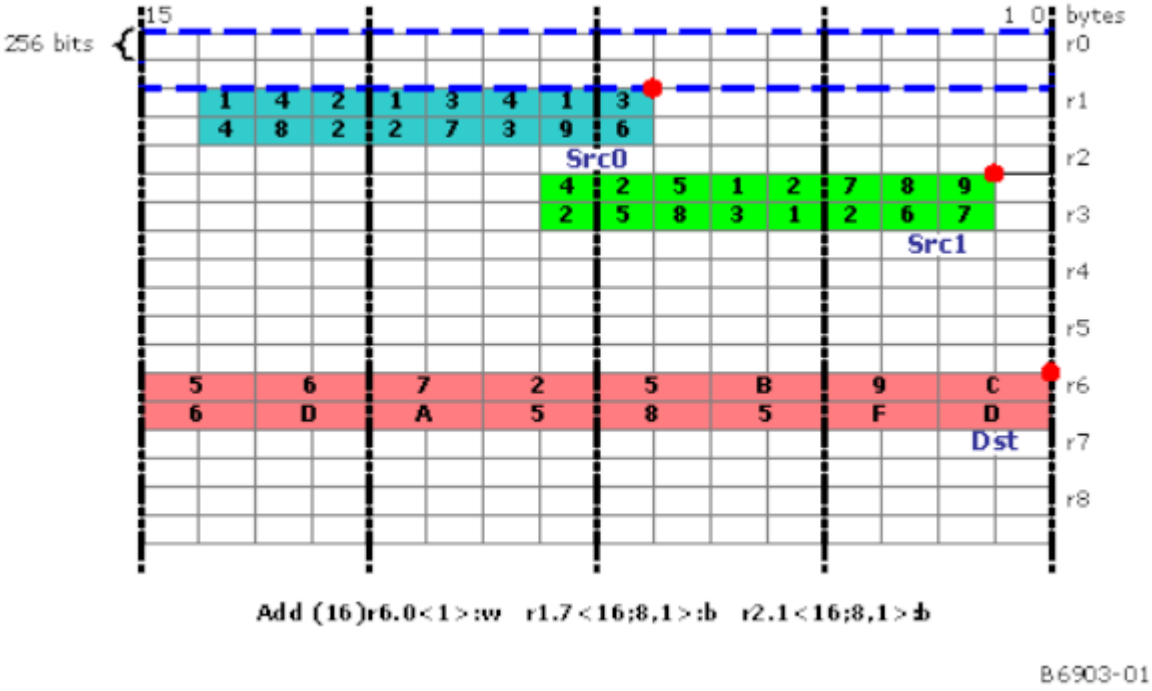
3.3.6.1 Direct Register Addressing

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

Direct Register Addressing and *Direct Register Addressing* are two examples of direct register addressing.

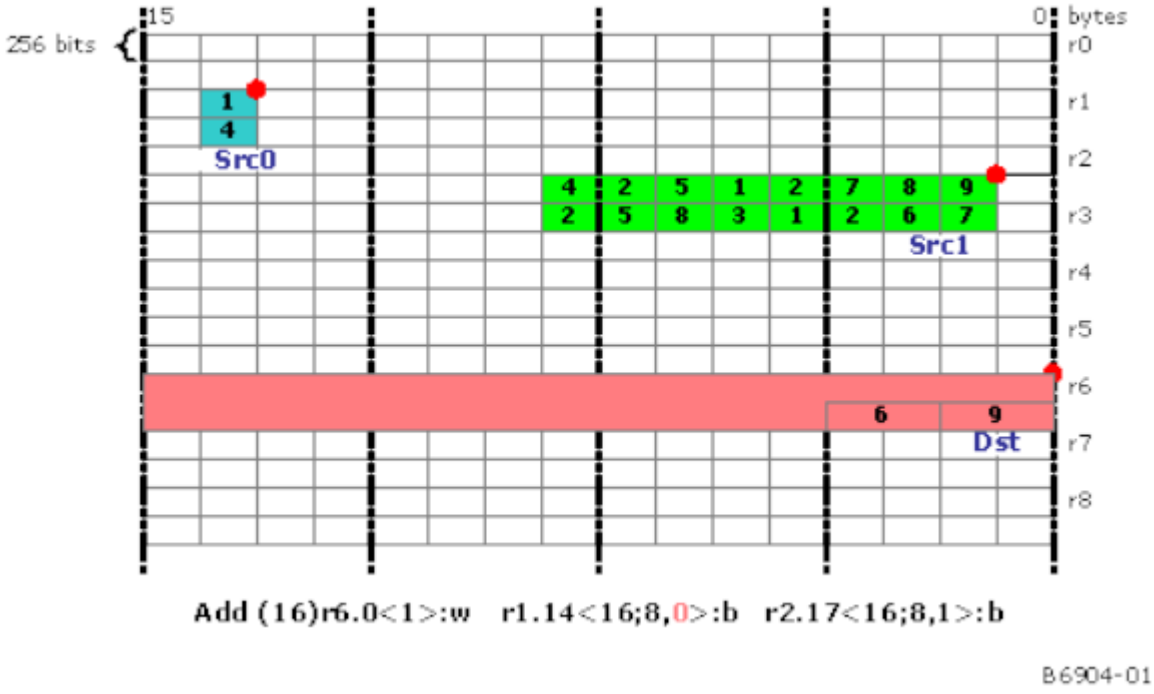
For the example in *Direct Register Addressing*, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical 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.

A region description example in direct register addressing



For the example in *Direct Register Addressing*, 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}.

A region description example in direct register addressing with src0 as a vector of replicated scalars

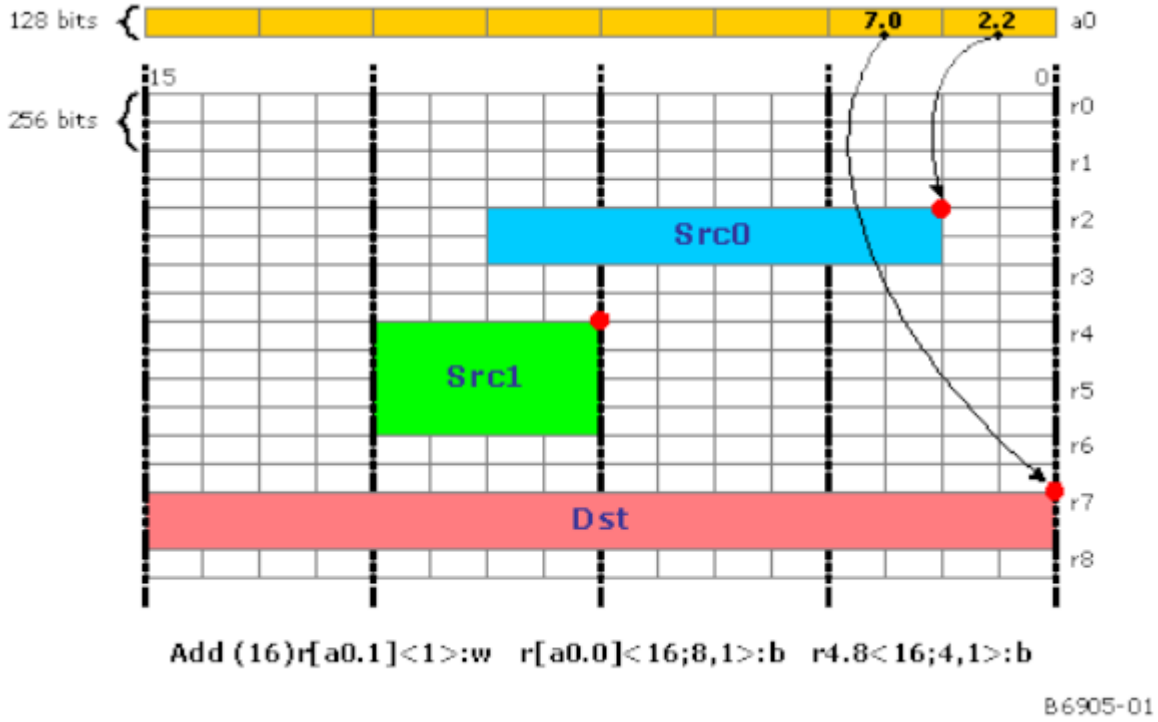


3.3.6.2 Register-Indirect Register Addressing with a 1x1 Index Region

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.

Register indirect Register Addressing with a 1x1 Index Region depicts an example for this addressing mode. For example, the presence of a full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.

An example illustrating register-indirect register addressing mode with a 1x1 index region



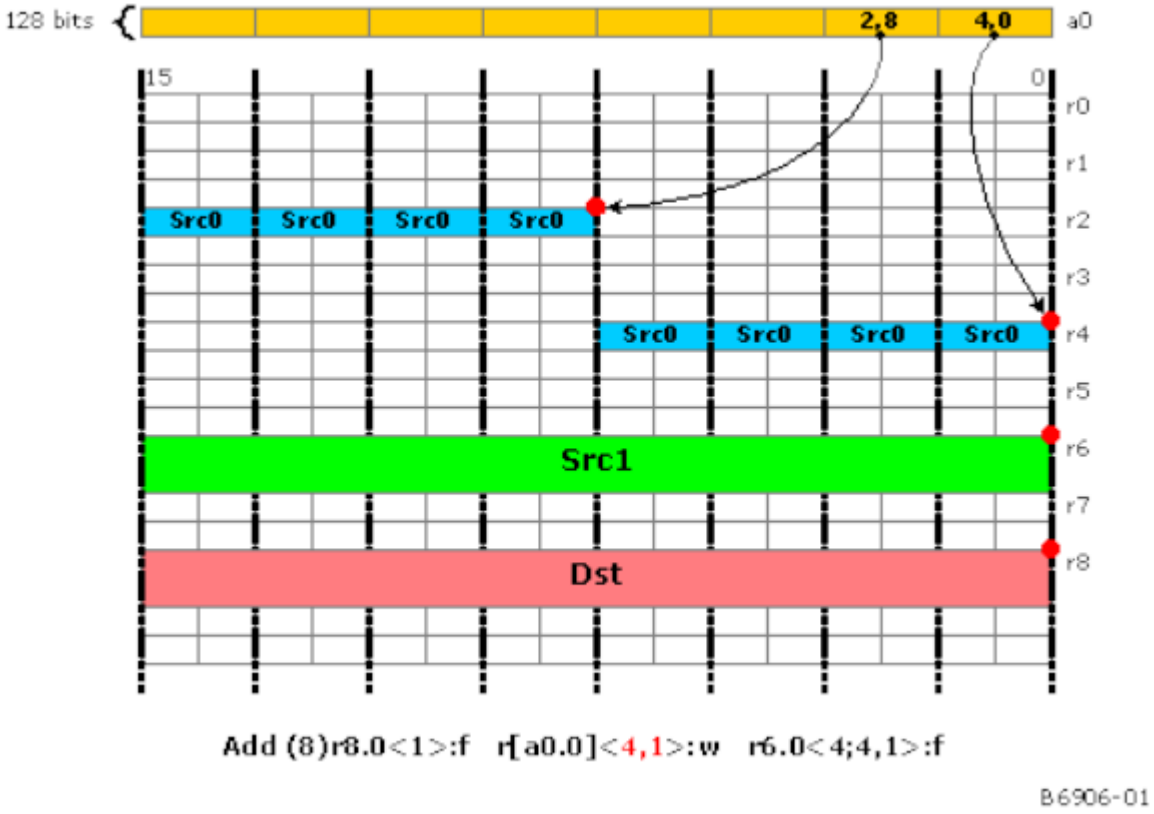
3.3.6.3 Register-Indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, the 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 and the following contiguous address registers are for the following rows. The total number of address registers used is inferred from the parameters *ExecSize* and *Width*.

Within the 16-bit address register, bits 15:5 determine RegNum and bits 4:0 determine SubRegNum.

An example is provided in *Register-Indirect Register Addressing with a Vx1 Index Region*. 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.

An example illustrating register-indirect-register addressing mode with a Vx1 index region (src0)

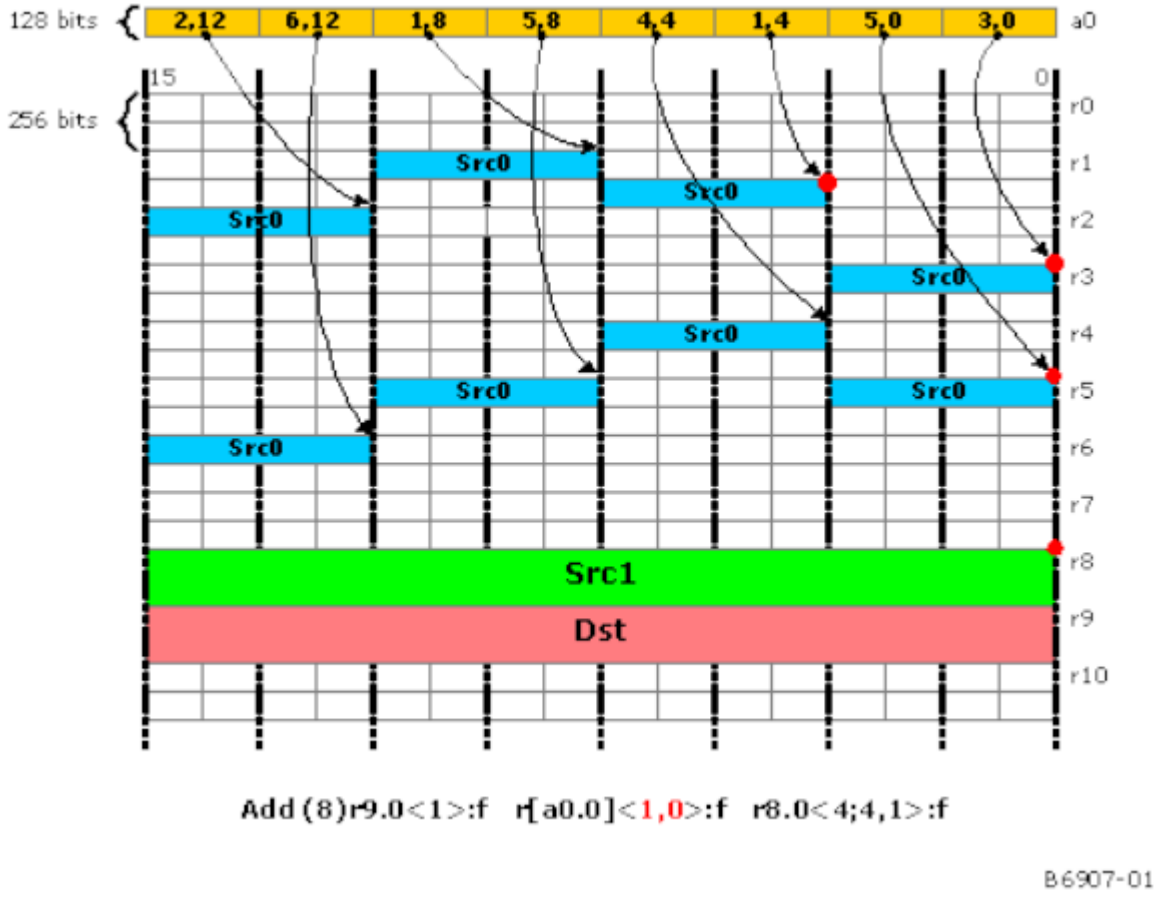


3.3.6.4 Register-indirect Register Addressing with a VxH Index Region

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 *Register indirect Register Addressing with a VxH Index Region*. 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.

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



3.3.7 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.

3.3.8 Execution Data Type

The GEN architecture carries out arithmetic and logical operations using a smaller set of data types than the variety supported as source or destination operands. These are the *execution data types*. A particular arithmetic or logical instruction has one execution data type, from those listed in the table.

Execution Data Types

Type	Description
W	Word. 16-bit signed integer.
D	Doubleword. 32-bit signed integer.
F	Float. 32-bit single precision floating-point number.
DF	Double Float. 64-bit double precision floating-point number.

The following rules explain the conversion of multiple source operand types, possibly a mix of different types, to one common execution type:

- For floating-point sources, all source operands must have the same floating-point type, with the exception that a two-source floating-point instruction can have Float as the src0 type and VF (Packed Restricted Float Vector) as the immediate src1 type.
- Mixing floating-point and integer source types is not allowed. Either all source types must be one floating-point type or all source types must be integer types.
- Unsigned integers are converted to signed integers.
- Byte (B) or Unsigned Byte (UB) values are converted to a Word or wider integer execution type.
- If source operands have different integer widths, use the widest width specified to choose the signed integer execution type.

Note that when the execution data type is an integer type, it is always a signed integer type. For integer execution types, extra precision is provided within the hardware, including the accumulators, so that conversions from unsigned to signed do not affect instruction correctness.

3.3.9 Register Region Restrictions

A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist.

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

1. **General Restrictions Based on Operand Types**

   There are these general restrictions based on operand types:

   A. Where *n* is the largest element size in bytes for any source or destination operand type, *ExecSize* * *n* must be <= 64.

   B. When the Execution Data Type is wider than the destination data type, the destination must be aligned as required by the wider execution data type and specify a *HorzStride* equal to the ratio

in sizes of the two data types. For example, a *mov* with a D source and B destination must use a 4-byte aligned destination and a *Dst.HorzStride* of 4.

2. **General Restrictions on Regioning Parameters**

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, the destination operand, and the *ExecSize*, with these restrictions:

A. *ExecSize* must be greater than or equal to *Width*.

B. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.

C. If *ExecSize* = *Width* and *HorzStride* = 0, there is no restriction on *VertStride*.

D. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.

E. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be 0. This defines a scalar.

F. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.

G. *Dst.HorzStride* must not be 0.

H. *VertStride* must be used to cross GRF register boundaries. This rule implies that elements within a '*Width*' cannot cross GRF boundaries.

3. **Region Alignment Rules**

A. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.

B. A destination cannot span more than 2 adjacent GRF registers.

C. When an instruction has a source region spanning two registers and a destination region contained in one register, one of the following must be true:

i. The destination region is entirely contained in the lower OWord of a register.

ii. The destination region is entirely contained in the upper OWord of a register.

iii. The destination elements are evenly split between the two OWords of a register.

D. When an instruction has a source region that spans two registers and the destination spans two registers, the destination elements must be evenly split between the two registers and each destination register must be entirely derived from one source register. **Note:** *In such cases, the regioning parameters must ensure that the offset from the two source registers is the same.*

The examples below illustrate the behavior of the cases permitted:

```
// Case (a) First 8 elements are from r12 to r10 and second from r13 to r11:
mov (16) r10.0<2>:w r12<16;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12<16;8,1>:w
mov (8) r11.0<2>:w r13<16;8,1>:w

// Case (b) First 8 elements from r12.8 to r10 and second from r13 to r11:
mov (16) r10.0<2>:w r12.8<16;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12.8<16;8,1>:w
mov (8) r11.0<2>:w r13.8<16;8,1>:w
```

The following examples indicate cases that are not allowed:

```
// Not allowed, because the source has 12 elements from r12 and 4 from r13:
mov (16) r10.0<2>:w r12.4<4;4,1>:w
```

```
// Not allowed, because the destination has 6 elements in r10 and 10 in r11:
mov (16) r10.2<2>:w r12<16;8,1>:w

// Not allowed, because the source has only one GRF register
// but the destination spans two registers:
mov (16) r10.0<2>:w r12.0<8;8,1>:w
```

E. When destination spans two registers, the source MUST span two registers. The exception to the above rule:
   i. When source is scalar, the source registers are not incremented.
   ii. When source is packed integer Word and destination is packed integer DWord, the source register is not incremented by the source sub register is incremented.

   The examples below illustrate the behavior of the cases permitted:

```
// Case (a) Scalar source:
mov (16) r10.0<2>:w r12.0<0;1,0>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<2>:w r12.0<0;1,0>:w
mov (8) r11.0<2>:w r12.0<0;1.0>:w

// Case (b) First 8 elements from r12 to r10 and second from r12.8 to r11:
mov (16) r10.0<1>:d r12<8;8,1>:w
// The above instruction behaves the same as the following two instructions:
mov (8) r10.0<1>:d r12<8;8,1>:w
mov (8) r11.0<1>:d r12.8<8;8,1>:w
```

F. With the exception of the two rules in “Special Cases for Byte Operations” below, 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 is either float or signed DWord integer. Therefore the destination data type must be DWord-aligned. This rule has the following two implications:
   i. The destination sub-register must be aligned to the size of the execution data type.
   ii. If *ExecSize* is greater than 1, *Dst.HorzStride* * sizeof(*Dst.DstType*) must be greater than or equal to the size of the execution data type.

4. **Special Cases for Byte Operations**

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

B. There is a relaxed alignment rule for byte destinations. When the destination type is byte (UB or B), destination data types can be aligned to either 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 two implications illustrated by this example:

```
// Example:
mov (8) r10.0<2>:b r11.0<8;8,1>:w
```

```
mov (8) r10.1<2>:b r11.0<8;8,1>:w

// Dst.HorzStride must be 2 in the above example so that the destination
// subregisters are aligned to the execution data type, which is :w.
// However, the offset may be .0 or .1.
// This special handling applies to byte destinations ONLY.
```

5. **Special Requirements for Handling Double Precision Data Types**

   A. In Align1 mode, all regioning parameters like stride, execution size, and width must use the syntax of a pair of packed floats. The offsets for these data types must be 64-bit aligned. The execution size and regioning parameters are in terms of floats.

```
// Example:
mov (8) r10.0<1>:df r11.0<8;8,1>:df
// The above instruction moves four double floats.
```

   B. In Align16 mode, all regioning parameters must use the syntax of a pair of packed floats, including channel selects and channel enables.

```
// Example:
mov (8) r10.0.xyzw:df r11.0.xyzw:df
// The above instruction moves four double floats. The .x picks the
// low 32 bits and the .y picks the high 32 bits of the double float.
```

   C. When using Align16 mode for conversion of data elements of different sizes, both source and destination must be one register each.

6. **Regioning Rules for Register Indirect Addressing**

   A. When the execution size and destination regioning parameters require two registers, each register is pointed to by adjacent index registers.

```
// Example:
mov (16) r[a0.0]:f r10:f
// The above instruction behaves the same as the following two instructions:
mov (8) r[a0.0]:f r10:f
mov (8) r[a0.1]:f r11:f
```

   B. When the destination requires two registers and the sources are indirect, the sources must use 1x1 regioning mode. In addition, the sources are assembled from two GRF registers each accessed by adjacent index registers in 1x1 regioning modes. The data for each destination GRF register is entirely derived from one source register.

      Note: There is an exception to this rule. When source is scalar, the indirect address is not incremented.

```
// Example:
// Case (a):
add (16) r[a0.0]:f r[a0.2]:f r[a0.4]:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]:f
add (8) r[a0.1]:f r[a0.3]:f r[a0.5]:f
// Each access, source and destination, is a 1x1 regioning access.
```

```
// Case (b):
add (16) r[a0.0]:f r[a0:2]:f r[a0.4]<0;1,0>:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]<0;1,0>:f
add (8) r[a0.1]:f r[a0.3]:f r[a0.4]<0;1,0>:f
// Note that the src1 indirect address does not change.
```

C. Indirect addressing on src1 must be a 1x1 indexed region mode.

D. When a Vx1 or a VxH addressing mode is used on src0, the destination must use ONLY one register.

E. Indirect addressing on the destination must be a 1x1 indexed region mode.

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

G. A register region with multiple indices may access multiple registers if the data elements associated with each index follow the above-mentioned rule.

```
// Example:
mov (16) r10.0:w r[a0.0]<2,2>:w
// This instruction gathers source elements pointed to by up to 8
// different physical GRF registers, where two elements accessed
// by each indirect access are from one physical GRF register.
```

7. **Special Restrictions**

A. In Align16 access mode, SIMD16 is not allowed for DW operations and SIMD8 is not allowed for DF operations.

B. When an instruction is SIMD32, the low 16 bits of the execution mask are applied for both halves of the SIMD32 instruction. If different execution mask channels are required, split the instruction into two SIMD16 instructions.

C. Instructions with condition modifiers must not use SIMD32.

D. All flow control (branching) instructions must use the Align1 access mode.

3.3.10 Destination Operand Description

3.3.10.1 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

3.4 SIMD Execution Control

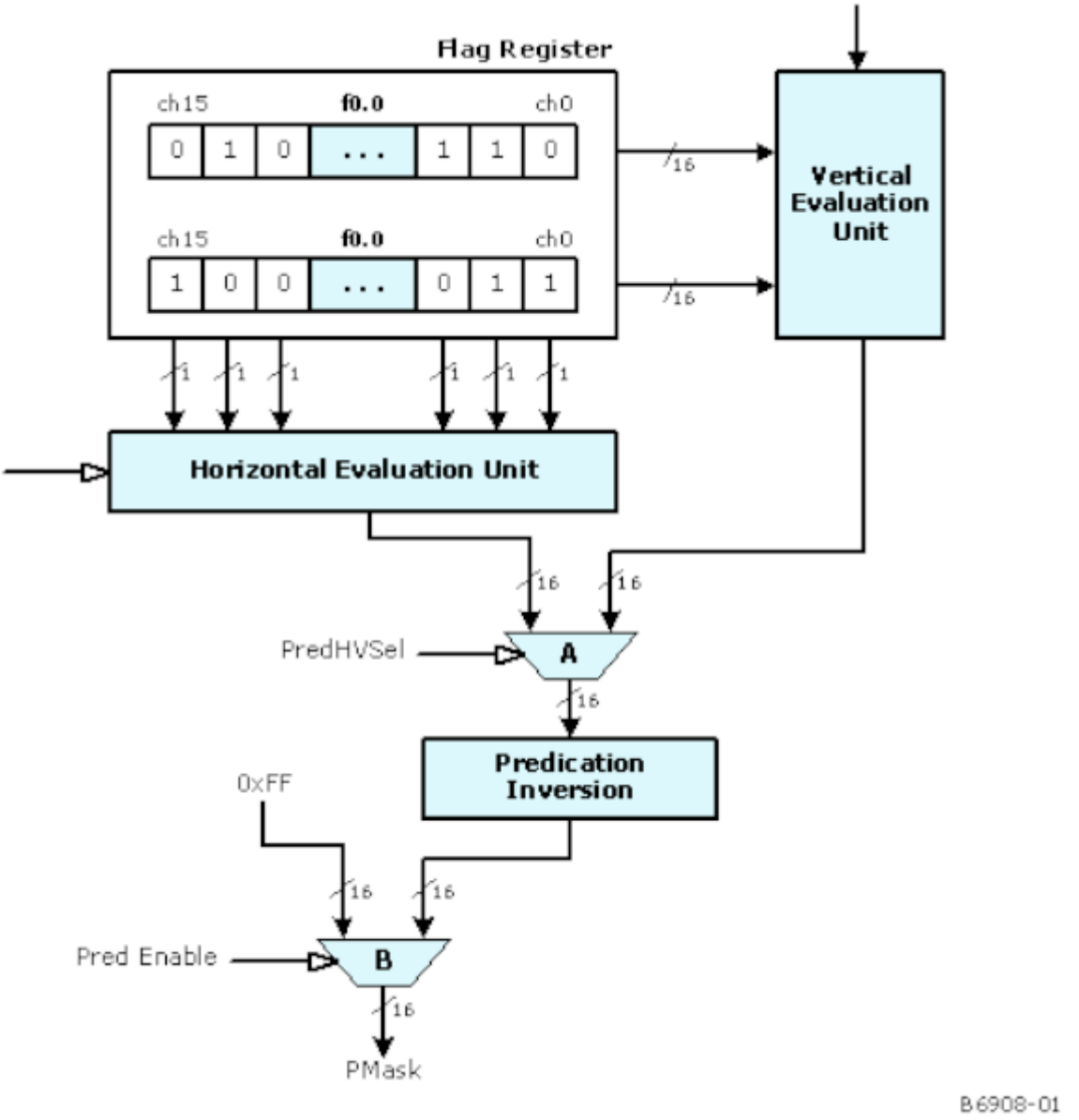
3.4.1 Predication

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.

Predication 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.

Generation of predication mask



3.4.2 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 *Predication*. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.

3.4.3 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;
    }
    }
```

3.4.4 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]
}
    }
```

3.5 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.

3.6 Assigning Conditional Flags

Instructions can output two sets of conditional signals, one set from before the outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. The second set is generated from the final results after clamping and re-normalizing/format conversion logic, and we call this the post conditional signals. The post conditional signals are used for fusing the DirectX compare instruction.
Note: 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 are used to generated flags for *cmp*/*cmpn* instructions only, this logically does the compare of the two input sources. The post conditional signals are used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

cmpn with both sources as NaNs is a don't care case as 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 before going through any kind of clamping, normalizing, or format conversion logic.
- **pre_zero** bit: This bit reflects whether the computed result is zero before any kind of 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 final result is zero after all the clamping, normalizing, or format conversion logic.
- **OF** bit: This bit reflects whether an overflow occured in any of the computation 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 0.
- **NS0** bit: The NaN Source 0 bit indicates whether src0 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 0.
- **NS1** bit: The NaN Source 1 bit indicates whether src1 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 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.
- Note that the bits generated at the output of a compute are before the **.sat**.

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 two sources are equal. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	**! (pre_zero & ! (NS0 | NS1)).** This conditional modifier test whether the two sources are equal. It takes exactly the reverse polarity as the modifier **.e**.
.g	Greater-than	**(! pre_sign & ! pre_zero & ! (NS0 | NS1)).** This conditional modifier tests whether src0 is greater than src1. 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 src0 is greater than or equal to src1. 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 src0 is less than src1. 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 src0 is less than or equal to src1. 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 | (NS1 & (Opcode ==** *cmpn* **| Opcode ==** *sel* **with CMod))) & ! (NS0 & (Opcode ==** *cmpn*)). This conditional modifier tests whether src0 is greater than or equal to src1. If src1 is a NaN (i.e. NS is true), the flag is forced to true.
.l	Less-than	**(pre_sign | (NS1 & (Opcode ==** *cmpn* **| Opcode ==** *sel* **with CMod))) & ! (NS0 & (Opcode ==** *cmpn*)). This conditional modifier tests whether src0 is less than src1. If src1 is a NaN (i.e. NS is true), the flag is forced to true.

Flag Generation for All Instructions Other than *cmp*/*cmpn* Instructions (The Supported Conditional Modifiers are .e, .ne, .g, .ge, .l, .le, .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 result is equal to zero. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	**! (post_zero & ! NC).** This conditional modifier test whether the result is not equal to zero. It takes exactly the reverse polarity as modifier **.e**.
.g	Greater-than	**(! post_sign & ! post_zero & ! NC).** This conditional modifier tests whether result is greater than zero. 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 result is greater than or equal to zero. 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 result is equal to zero. 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 result is equal to or less than zero. 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.

3.7 Destination Hazard

The 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 certain that it is a false destination hazard, the programmer should rely on hardware to resolve 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.

When a sequence of NoDDChk and NoDDClr are used, the last instruction that completes the scoreboard clear must have a non-zero execution mask. This means, if any kind of predication can change the execution mask or channel enable of the last instruction, the optimization must be avoided. This is to avoid instructions being shot down the pipeline when no writes are required.

Example:

(f0.0) mov r10.0 r11.0 {NoDDClr}

(-f0.0) mov r10.0 r11.0 {NoDDChk, NoDDClr}

In the above case, if predication can disable all writes to r10 for the second instructions, the instruction maybe shot down the pipeline resulting in un-deterministic behavior. Hence, This optimization must not be used in these cases.

3.8 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.

3.9 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).

4. Exceptions

The 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, as well as detecting some illegal conditions .

Exception Types

Type	Trigger / Source	Sync/Async Recognition
Software Exception	Thread code	Synchronous
Breakpoint	• A bit in the instruction word • Breakpoint IP match • Breakpoint Opcode match	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous
Context Save/Restore	Preemption Interrupt	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 the 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 enable bits in the cr0.1 register are read/write, and thus can be enabled/disabled via software at any time during thread execution.

The exception handling mechanism relies on the System Routine, a single subroutine that 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 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.

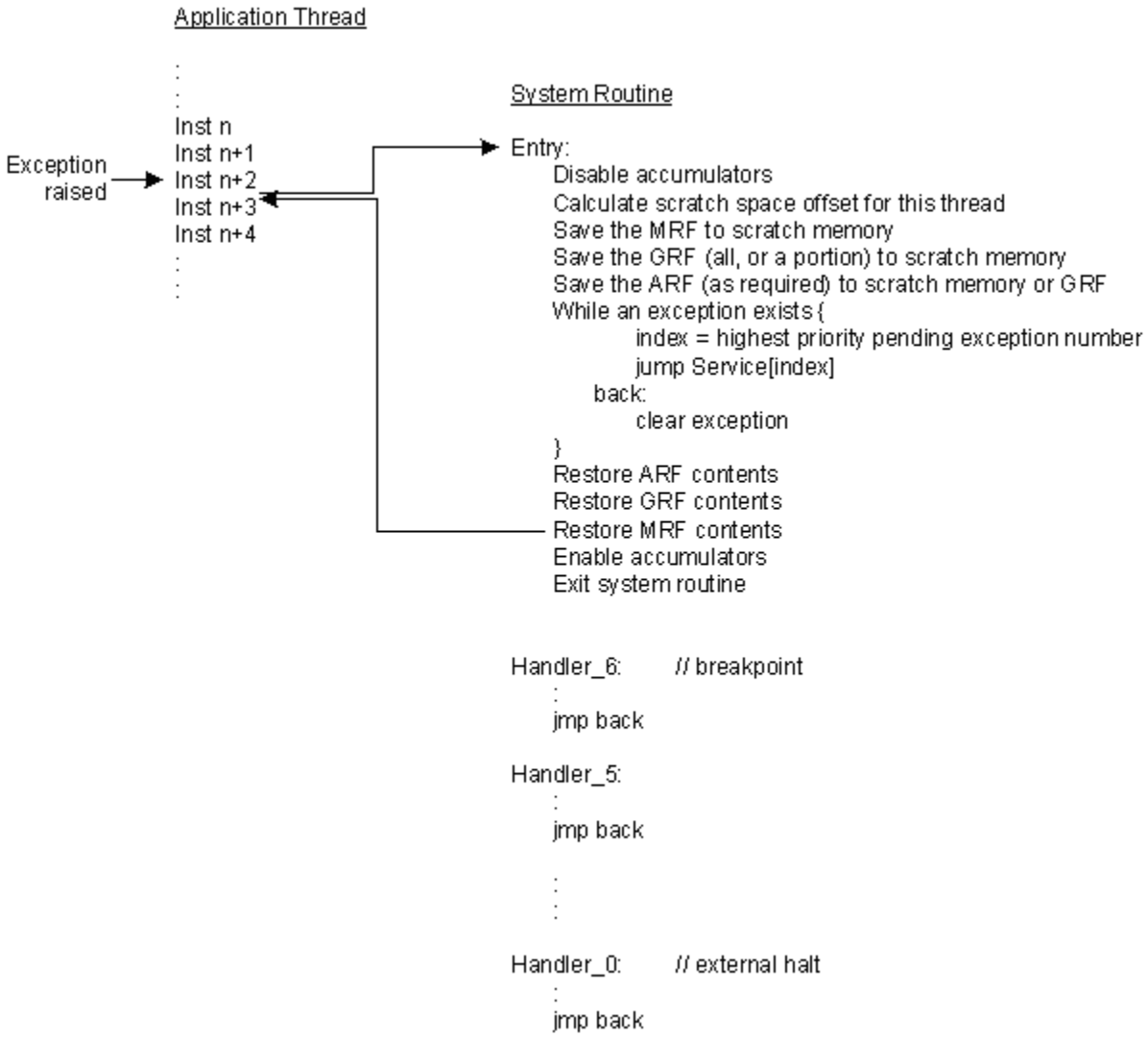
4.1 Exception-Related Architecture Registers

Exception-related registers are architecture registers 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, store 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 atomic read-modify-write operations when accessing these registers.

4.2 System Routine

The following diagram illustrates the basic flow of exception handling and the structure of the System Routine.



4.2.1 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 are issued until the System Routine has executed and returned control back to the application thread.

After an exception is recognized by hardware, the EU saves the thread's IP into its AIP register (cr0.2), and then moves the System Routine offset, SIP, into the thread's IP register. At this point the next instruction issued for that thread is the first instruction of the System Routine.

The System Routine maintains the same execution priority, GRF register space, and thread state as the application thread from which it is invoked. Due to assuming the same priority, there may be significant

absolute time between an exception being raised and invoking 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 issued 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 the register retires.

Exception processing is not nested within the 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 is 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 before using any instruction that modifies an accumulator, or save and restore the accumulators (using GRF registers or system thread scratch memory) around the System Routine. Saving and restoring accumulators, including their extended precision bits, can be accomplished by a short series of moves and shifts of the accumulator register. Also note that 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 architecture registers across the System Routine. The System Routine must ensure that these values are preserved (see the *Conditional Instructions Within the System Routine* section for related information).

4.2.2 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 cr0.1. Failure to do so forces the thread's execution to reenter the System Routine before any further instructions are executed from the application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset before exit.)

The System Routine may choose to loop within 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 in cr0.0. Upon clearing this bit, the value of AIP (cr0.2) is restored to the thread’s IP register and, with no further exceptions pending, execution resumes at that address. The System Routine must follow any write to the 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).

4.2.3 System IP (SIP)

The System IP (SIP) is the 16 byte-aligned offset of the first instruction of the System Routine, relative to the General State Base Address. SIP is assigned by the STATE_SIP command to the command streamer which updates SIP in the EU.

When the System Routine is invoked, the application thread's current IP is first saved into the AIP field of 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. Returning from the System Routine loads IP from AIP, continuing thread execution.

4.2.4 System Routine Register Space

The System Routine uses the same GRF space as the thread that invokes 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 for all threads.

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 32-bit GRF location, r0.4, is reserved for System Routine use. This location 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 is likely based on a combination of Thread and EU IDs (see the example system handler in the *System Scratch Memory Space*) section. 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 and application code, or (b) the System Routine temporarily spilling the thread's GRF register contents to scratch memory and restoring those contents before System Routine exit.

No persistent storage is automatically allocated to the System Routine, although a driver implementation may set aside part of system scratch memory for the System Routine.

Any parameter passing to the System Routine (for use by software exceptions) is done via the GRF based on system thread/application thread convention.

4.2.5 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 conventions implemented in the System Routine, with each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread can be based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architecture 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 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 Data Port messages used within the System Routine coupled with the fact that a single System 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 scratch memory space is allocated by the driver at context-create time and remains persistent at a constant memory address throughout a context's lifetime.

4.2.6 Conditional Instructions Within the System Routine

It is expected that most, if not all, control flow within 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 to simply implement the save/restore as part of the entry/exit code sequence, using an available GRF register pair as a storage location. Once saved, the stacks should be reset to their empty condition, namely depth = 0 and top of stack = 0xFFFFFFFF.

4.2.7 Use of NoDDClr

The instruction word defines an instruction option **NoDDClr** that 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 a 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 without dependency checks, and the last instruction to 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 if an exception is raised within a NoDDClr code block. In this case, there exists the potential for the System Routine to hang while attempting to save/restore a register used as a destination register by the NoDDCLr code block, as the outstanding dependency on that register will not clear until the final instruction of the NoDDCLr block is executed, sometime after the System Routine returns. Should the System Routine attempt to use that register, it hangs waiting on a dependency to be cleared by an instruction not yet issued.

Note: 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 avoid this condition, 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 that may generate exceptions should not be placed within NoDDClr blocks. This includes most conditional branch instructions (if, do, while, ...) .
- 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 that 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.

4.3 Exception Descriptions

This section describes conditions that can cause exceptions and transfer control to the System Routine.

4.3.1 Illegal Opcode

The ISA defines a single *illegal* opcode. The byte value of the *illegal* opcode is 0x00 due to it being a likely byte value encountered by a wayward instruction pointer value. The *illegal* instruction signals an exception if exception handling is enabled and invokes the system interrupt routine. If exception handling is NOT enabled, the illegal opcode is executed resulting in undetermined behavior including a system hang. Hardware decodes all legal opcodes supported. Any byte value that is not in the legal opcode list is decoded as an illegal opcode to trigger exception.

4.3.2 Undefined Opcodes

All undefined opcodes in the 8-bit opcode space (which includes instruction bit 7, reserved for future opcode expansion) are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing the opcode value within the pipeline 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 the section *Illegal Opcode*. The memory location of the offending opcode keeps its original value. That location can be queried to determine the opcode value.

4.3.3 Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered using 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 IP is incremented before saving AIP and entering the System Routine, causing execution to resume at the next application-thread instruction after returning from the System Routine.

4.3.4 Context Save and Restore

The System Routine is also used to save and restore the context of the Execution Unit. This feature is enabled in GPGPU workloads *only*.

When the execution engine receives a preemption or an interrupt, the application thread invokes the System Routine. The System Routine is invoked only when all in-flight registers have retired. The system routine is used to save all the state of the EU to memory. When the sequence is complete, the master exception control bit is cleared. This action stops all execution for the given thread and invalidates the thread. This means a new thread from a different context may be loaded. When the master exception control bit is cleared, software must ensure that all outstanding messages from the EU are dispatched out of the execution message pipeline. This is achieved by creating a dependency on the last send that is saving EU state. A dummy instruction before clearing the master exception control bit ensures that this is achieved.

The System Routine is also invoked on a context restore request. In this case a dummy thread is loaded into the EU which starts with the System Routine. This routine now restores the state of the EU. The restore sequence used in such a case should be consistent with the save sequence to ensure that state is restored correctly. After completing the restore sequence, the System Routine must clear the master exception control bit in the Control Register. This enables hardware to switch to the application thread which continues execution.

4.4 Events That Do Not Generate Exceptions

The conditions described in this section are either not recognized or do not generate an exception.

4.4.1 Illegal Instruction Format

This condition includes malformed instructions in which the opcode is legal, but the source or destination operands or other instruction attributes do not comply 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.

4.4.2 Malformed Message

A message's contents, destination registers, lengths, and descriptors are not interpreted in any way by the execution unit. Errors in specifying message fields do not raise exceptions in the EU but may be detected and reported by the shared functions .

4.4.3 GRF Register Out of Bounds

Unique GRF storage is allocated to each thread which, at a minimum, satisfies 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 the 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 thread's register space, thus there is no possibility of direct register manipulation via an out-of-bounds register access.

4.4.4 Hung Thread

There is no hardware mechanism in the EU to detect a hung thread and such a thread may remain hung indefinitely. It is expected that one or more hung threads will eventually cause the driver to recognize a context timeout and take appropriate recovery action.

4.4.5 Instruction Fetch Out of Bounds

The EU implements a full 32-bit instruction address range (with the 4 LSBs don't care), making it possible for a thread to attempt to jump to any 16-byte aligned offset in the 32-bit instruction address range. (Instruction addresses are offsets from the General State Base Address.) The EU does not provide any type of address checking on instruction fetch requests sent to the memory/cache hierarchy, although error conditions for memory addresses are reported via the Page Table Error Register and other memory interface registers.

4.4.6 FPU Math Errors

The EU's floating point units (FPUs) have defined behaviors for traditional floating point errors and do not generate exceptions. There is no support for signaling FPU math errors as exceptions.

4.4.7 Computational Overflow

Depending on source operand types and values, destination type, and the operation being performed, overflows may occur in the execution pipelines. Many instructions support the overflow (**.o**) conditional modifier that assigns flag bits based on whether or not an overflow occurs.

The EU never signals exceptions for overflows. Software must provide any overflow handling.

4.5 System Routine 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.

This example contains code for the message registers in the MRF. All message register and MRF references are specific to DevSNB. Other code in this example is useful for other processor generations.

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

// Shared function 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

// Data port block sizes:
#define BS1_LOW  0x0000
#define BS1_HIGH 0x0100
#define BS2      0x0200
#define BS4      0x0300

// Scratch Layout:
#define SCR_OFFSET_MRF 0           // MRF is specific to DevSNB.
#define SCR_OFFSET_GRF 512         // + 16 MRF registers
#define SCR_OFFSET_ARF 512 + 4096  // + 16 MRF + 128 GRF registers

// Write data port constants:
// target=dcache, type= oword_block_wr, binding_tbl_offset=0
#define DPW 0x000

// Read data port constants:
// target=dcache, type= oword_block_rd, binding_tbl_offset=0
#define DPR 0x000

Sys_Entry:  // Entry point to the System Routine.

// Disable accumulator for system routine:
and (1) cr0.0 cr0.0 ACC_DISABLE_MASK {NoMask}

// 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 with block offset:
mov (8) m0 r0{NoMask}

// Save MRF 7..0 (may choose to save the whole MRF). MRF is specific to DevSNB:
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). MRF is
specific to DevSNB.
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 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 selected ARF registers (optional, depending on use):
// flags, 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}

// Jump table 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 other ARF registers (as required).
// 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 case of a timeout waiting on a register that may never return.

At this point, all we know is we have a hung thread. We'd like to copy the GRF to scratch memory to make it visible , but there may be a register that is hung with an outstanding dependency. To get around any hung dependency, walk the GRF using NoDDChk, using an 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.
```

5. Instruction Set Summary

5.1 Instruction Set Characteristics

5.1.1 5nstruction Operands and Register Regions

Most 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*** or ***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. There are 8 address 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.

5.1.2 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.

5.2 Instruction Machine Formats

This section shows the machine formats of the instruction set. The instructions in the architecture have a fixed length of 128 bits in the native format. A compact format, discussed separately in this volume, can represent some instructions using 64 bits. Out of the 128 bits in the native format, 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. These fields are roughly grouped into the 4 DWords as follows:

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

Most instructions have 1 or 2 source operands and use a common instruction format. Within that format, there are variations based on AddrMode and AccessMode. There is a separate instruction format for a small number of instructions with 3 source operands. Send, math, and branching instructions have format variations described separately.

The 3-source instructions have the following restrictions:

- Only GRF registers can be sources, and only GRF registers can be the destination.
- Subregister numbers have DWord granularity.
- AccessMode is Align16, uses Align16-style swizzling, with extra replication control. There is no other regioning support.

The next two subsections describe the instruction formats for various processor generations using tables. The following diagrams provide another view of the same information. The first two diagrams are for native instructions with one or two source operands.

Instruction Format – 1-src and 2-src

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	AddrMode = Direct: AccessMode = Align16	AddrMode = Direct: AccessMode = Align1	AddrMode = Indirect: AccessMode = Align16	AddrMode = Indirect: AccessMode = Align1	SEND: MsgDesc Imm	SEND: MsgDesc Reg	MATH	Branch (2offsets)	Branch (1offset)	Imm Src	DWI
	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	Imm[63:0]
	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	Same
	1	47	47	1	NibCtrl										Same
	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	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]	MEI2	MEI2		
	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	Same
	1	7	7	0	(reserved for Opcode)										
0	7	6	0	7	Opcode				Same		Same	Same	Same	Same	Same

The next two diagrams are for instructions with three source operands.

Instruction Format – 3-src

DW #	Instr Bits	High Bit	Low Bit	Instr Bits	Description
	2	127	126	0	*reserved*
	8	125	118	8	Src2 Regnum
	3	117	115	3	Src2 Subregnum
	8	114	107	8	Src2 Swizzle
	1	106	106	1	Src2 RepCtrl
	1	105	105	0	*reserved*
	8	104	97	8	Src1 Regnum
	3	96	94	3	Src1 Subregnum
	8	93	86	8	Src1 Swizzle
	1	85	85	1	Src1 RepCtrl
	1	84	84	0	*reserved*
	8	83	76	8	Src0 Regnum
	3	75	73	3	Src0 Subregnum
	8	72	65	8	Src0 Swizzle
2,3	1	64	64	1	Src0 RepCtrl
	8	63	56	8	Dst Regnum
	3	55	53	3	Dst Subregnum
	4	52	49	4	Dst chan enable
	1	48	48	0	*reserved*
	1	47	47	1	*NibCtrl*
	1	46	46	0	*reserved*
	2	45	44	2	Dst Type
	2	43	42	2	Src Type
	2	41	40	2	Src2 Modifier
	2	39	38	2	Src1 Modifier
	2	37	36	2	Src0 Modifier
	1	35	35	0	*reserved*
	1	34	34	1	FlagRegNum
	1	33	33	1	Flag Sub Reg Num
1	1	32	32	1	*reserved*
	1	31	31	1	*Saturate*
	1	30	30	1	*DebugCtrl*
	1	29	29	1	*CmptCtrl*
	1	28	28	1	*AccWrCtrl*
	4	27	24	4	*CondModifier*
	3	23	21	3	*Exec Size*
	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	*WE Ctrl*
	1	8	8	1	*AccessMode*
	1	7	7	0	*(reserved for Opcode)*
0	7	6	0	7	*Opcode*

5.2.1 EU Instruction Formats

This section covers the layout of instruction fields, not changes in allowed field encodings from generation to generation. For example, adds new data types and instructions which add new allowed values for particular fields, but those changes do not change field positions or layout.

DWord 0, bits 31:0 of the 128-bit instruction, has the same format regardless of the number of source operands.

The following three tables cover the most common instruction format, for instructions with 1 or 2 source operands; then the format for the few instructions with 3 source operands; and finally format variations used by a few exceptional instructions.

Execution Unit Instruction Format for 1 or 2 Source Operands

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
Any Imm32 32-bit immediate operand uses bits 127:96, replacing the following fields.					
127:121	Reserved				
120:117	Src1.VertStride				
116	Varies based on AccessMode	Reserved	Src1.Width	Reserved	Src1.Width
115:114		Src1.ChanSel[7:4]		Src1.ChanSel[7:4]	
113:112				Src1.HorzStride	
111	Src1.AddrMode				
110:109	Src1.SrcMod				
108:106	Varies based on AddrMode and AccessMode	Src1.RegNum		Src1.AddrSubRegNum	
105:101				Src1.AddrImm[9:4]	Src1.AddrImm[9:0]
100					
99:96		Src1.ChanSel[3:0]	Src1.ChanSel[3:0]		
95:91	Reserved				
90	FlagRegNum				
89	FlagSubRegNum				
88:85	Src0.VertStride				
84	Varies based on AccessMode	Reserved	Src0.Width	Reserved	Src0.Width
83:82		Src0.ChanSel[7:4]		Src0.ChanSel[7:4]	
81:80				Src0.HorzStride	
79	Src0.AddrMode				
78:77	Src0.SrcMod				
76:74	Varies based on AddrMode and AccessMode	Src0.RegNum		Src0.AddrSubRegNum	
73:69				Src0.AddrImm[9:4]	Src0.AddrImm[9:0]
68					
67:64		Src0.ChanSel[3:0]	Src0.ChanSel[3:0]		
63	Dst.AddrMode				
62:61	Varies based on AccessMode	Reserved	Dst.HorzStride	Reserved	Dst.HorzStride
60:58	Varies based on AddrMode and AccessMode	Dst.RegNum		Dst.AddrSubRegNum	
57:53				Dst.AddrImm[9:4]	Dst.AddrImm[9:0]
52					

Bits	Description	AddrMode and AccessMode Variations			
		AddrMode = Direct		AddrMode = Indirect	
		Align16	Align1	Align16	Align1
51:48		Dst.ChanEn[3:0]	Dst.ChanEn[3:0]		
47	NibCtrl				
46:44	Src1.SrcType				
43:42	Src1.RegFile				
41:39	Src0.SrcType				
38:37	Src0.RegFile				
36:34	Dst.DstType				
33:32	Dst.RegFile				
31	Saturate				
29	CmptCtrl				
28	AccWrCtrl				
27:24	CondModifier				
23:21	ExecSize				
20	PredInv				
19:16	PredCtrl				
15:14	ThreadCtrl				
13:12	QtrCtrl				
11:10	DepCtrl				
9	MaskCtrl				
8	AccessMode				
7	Reserved (for future Opcode expansion)				
6:0	Opcode				

The 3-source operand instructions are:

bfe - Bit Field Extract
bfi2 - Bit Field Insert 2
lrp - Linear Interpolation
mad - Multiply Add

In the 3-source instruction format, the upper QWord contains three groups of 21 bits for the three source operands, where each group contains four fields in 20 bits and otherwise adjacent groups are separated by single reserved bits.

Execution Unit Instruction Format for 3 Source Operands

Bits	Description
127:126	Reserved
125:118	Src2.RegNum
117:115	Src2.SubRegNum
114:107	Src2.ChanSel
106	Src2.RepCtrl
105	Reserved
104:97	Src1.RegNum
96	Src1.SubRegNum[2]
95:94	Src1.SubRegNum[1:0]
93:86	Src1.ChanSel
85	Src1.RepCtrl
84	Reserved

Bits	Description
83:76	Src0.RegNum
75:73	Src0.SubRegNum
72:65	Src0.ChanSel
64	Src0.RepCtrl
63:56	Dst.RegNum
55:53	Dst.SubRegNum
52:49	Dst.ChanEnable
48	Reserved
47	NibCtrl
46	Reserved
45:44	DstType
43:42	SrcType
41:40	Src2.Modifier
39:38	Src1.Modifier
37:36	Src0.Modifier
35	Reserved
34	FlagRegNum
33	FlagSubRegNum
32	Reserved
31	Saturate
29	CmptCtrl
28	AccWrCtrl
27:24	CondModifier
23:21	ExecSize
20	PredInv
19:16	PredCtrl
15:14	ThreadCtrl
13:12	QtrCtrl
11:10	DepCtrl
9	MaskCtrl
8	AccessMode
7	Reserved (for future Opcode expansion)
6:0	Opcode

Specific instructions have different instruction formats as described below. These instructions include send / sendc, math, and branch instructions.

Execution Unit Instruction Format for Specific Instructions

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description		
		send / sendc	math	Branch Instructions
127	Reserved	EOT		UIP[15:0] (2-offset branches)
126:125				
124:121		Imm[28:0] / Reg32		
120:117				Src1.VertStride
116:112				Varies based on

Bits	Regular 1 or 2 Source Operands Description	Empty white areas mean Same, use the regular description		
		send / sendc	math	Branch Instructions
				AccessMode
111			Src1.AddrMode	JIP[15:0]
110:109			Src1.SrcMod	
108:96			Varies based on AddrMode and AccessMode	
95:91	Reserved			
90	FlagRegNum			
89	FlagSubRegNum			
88:85	Src0.VertStride			
84:80	Varies based on AccessMode			
79	Src0.AddrMode			
78:77	Src0.SrcMod			
76:64	Varies based on AddrMode and AccessMode			
63	Dst.AddrMode			Any branch instruction: Same as regular
62:61	Varies based on AccessMode			
60:48	Varies based on AddrMode and AccessMode			
47	NibCtrl			
46:44	Src1.SrcType			
43:42	Src1.RegFile			
41:39	Src0.SrcType			
38:37	Src0.RegFile			
36:34	Dst.DstType			
33:32	Dst.RegFile			
31	Saturate			
29	CmptCtrl			
28	AccWrCtrl			
27:24	CondModifier	SFID[3:0]	FC[3:0]	Any branch instruction: MBZ
23:21	ExecSize			
20	PredInv			
19:16	PredCtrl			
15:14	ThreadCtrl			
13:12	QtrCtrl			
11:10	DepCtrl			
9	MaskCtrl			
8	AccessMode			
7	Reserved (for future Opcode expansion)			
6:0	Opcode			

5.2.2 Common Instruction Fields

As shown in the table below, the meanings (encoding) of certain bit fields in the 128-bit native instruction format varies depending on the values of other bit fields.

Common Instruction Fields provides the definition of common fields in the native instruction format. The 'Width' column specifies the width of the field in bits. These common fields are referenced in describing the fields of different doublewords of the instruction. The definition for fields that have unique representations can be found in the sections for the corresponding instruction DWords.

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 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 (signed overflow) (**.o**) 1001 = Unordered with Computed NaN (**.u**) 1010 -1111 = Reserved	4
AddrMode	**Addressing Mode.** This field determines the addressing method of the operand. Normally the destination operand and each source operand each have a distinct addressing mode field. 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
RegNum	**Register Number.** This field provides the register number for the operand. For GRF 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	8

Field	Description	Width
	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 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.	
SubRegNum	**Sub-Register Number.** This field provides the sub-register number for the operand. For a GRF 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 the encoding defined 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 the *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. **Note:** The recommended instruction syntax uses subregister numbers within the GRF in units of actual data element size, corresponding to the data type used. For example for the F (Float) type, the assembler syntax uses subregister numbers 0 to 7, corresponding to subregister byte addresses of 0 to 28 in steps of 4, the element size. Format = U5, if *RegFile* = **GRF** 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
AddrSubRegNum	**Address Sub-Register Number.** This field provides the subregister number for the address register. The address register contains 8 sub-registers. The size of each subregister 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	3

Field	Description	Width
	the operand is directly addressed. An address subregister used for indirect addressing is often called an *index register*. Format = U3 0x0 to 0x7 = Address Sub-Register number in the range [0, 7]	
AddrImm	**Address Immediate.** This field provides the immediate value in units of bytes added to the address register to compute the register address (byte-aligned region origin) for the operand. It is a signed integer. 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 numeric modification of 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
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 of 0 or powers of 2, ranging from 1 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. *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.*	4

Field	Description	Width
	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)	
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
HorzStride	**Horizontal Stride.** 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	2
Imm32	**32-bit Immediate.** 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. See the Numeric Data Types section for information about data types and their ranges.	32

Field	Description	Width
ChanEn	**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. This field only applies to destination operand. This field is only present in **Align16** mode. 0 = Write Disabled 1 = Write Enabled (normal)	4
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 ***[Errata Astep]*** *Replicate Control is not supported. As a WorkAround, Break 3 source instructions to simd4 and use channel select when replicate control is required.*	1
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

Field	Description	Width
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

5.2.3 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 *SFID[3:0]* field for the *send* instruction.

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

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 represented 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 numeric 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.

000b = 1 channel (scalar operation)

001b = 2 channels

010b = 4 channels

011b = 8 channels

100b = 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>
<tr><td>19:16</td><td>
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.

See the Predication section for more informatio about predication.

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)
</td></tr>
</table>

Bits	Description
	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) 1100 = Predication with **.any32h** (any in group of 32 channels) 1101 = Predication with **.all32h** (all in group of 32 channels) 1110 -1111 = Reserved
15:14	**ThreadCtrl – Thread Control.** This field provides explicit control for thread switching. If this field is set to 00b, it is up to the GEN execution units to manage thread switching. This is the normal (and unnamed) mode. In this mode, for example, if the current instruction cannot proceed due to operand dependencies, the EU switches to the 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 that also has an instruction ready, the 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) is introduced for the current thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. **Switch** is designed primarily as a safety feature in case there are race conditions for certain instructions. If this field is set to **Atomic**, the next instruction gets highest priority in thread arbitration for the execution pipeline. 00b = Normal thread control 10b = **"Switch"**

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>01b = “Atomic”
11b = Reserved</td></tr>
<tr><td>13:12</td><td>QtrCtrl – Quarter Control. This field provides explicit control for ARF selection.
This filed combines with ExecSize determines which channels are used for the ARF registers.
Along with NibCtrl in DW1, 1/8 DMask/VMask and ARF can be selected.
<table>
<tr><th>QtrCtrl</th><th>NibCtrl</th><th>ExecSize</th><th>Description</th><th>BNF</th></tr>
<tr><td>00</td><td>x</td><td>8</td><td>use first quarter for DMask/VMask
use first half for everything else</td><td>1Q</td></tr>
<tr><td>01</td><td>x</td><td>8</td><td>use second quarter for DMask/VMask
use second half for everything else</td><td>2Q</td></tr>
<tr><td>10</td><td>x</td><td>8</td><td>use third quarter for DMask/VMask
use first half for everything else</td><td>3Q</td></tr>
<tr><td>11</td><td>x</td><td>8</td><td>use forth quarter for DMask/VMask
use second half for everything else</td><td>4Q</td></tr>
<tr><td>0x</td><td>x</td><td>16</td><td>use first half for DMask/VMask
use all channels for everything else</td><td>1H</td></tr>
<tr><td>1x</td><td>x</td><td>16</td><td>use second half for DMask/VMask
use all channels for everything else</td><td>2H</td></tr>
<tr><td>00</td><td>0</td><td>4</td><td>use first 1/8 for DMask/VMask and ARF</td><td>1N</td></tr>
<tr><td>00</td><td>1</td><td>4</td><td>use second 1/8 for DMask/VMask and ARF</td><td>2N</td></tr>
<tr><td>01</td><td>0</td><td>4</td><td>use third 1/8 for DMask/VMask and ARF</td><td>3N</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td>01</td><td>1</td><td>4</td><td>use fourth 1/8 for DMask/VMask and ARF</td><td>4N</td></tr>
<tr><td>10</td><td>0</td><td>4</td><td>use fifth 1/8 for DMask/VMask and ARF</td><td>5N</td></tr>
<tr><td>10</td><td>1</td><td>4</td><td>use sixth 1/8 for DMask/VMask and ARF</td><td>6N</td></tr>
<tr><td>11</td><td>0</td><td>4</td><td>use seventh 1/8 for DMask/VMask and ARF</td><td>7N</td></tr>
<tr><td>11</td><td>1</td><td>4</td><td>use eighth 1/8 for DMask/VMask and ARF</td><td>8N</td></tr>
</table>
2H is only allowed for SIMD16 instruction in Single Program Flow mode (SPF=1).

NibCtrl is only allowed for SIMD4 instructions with (DF) double precision source and/or destination.
</td></tr>
<tr><td>11:10</td><td>
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
</td></tr>
<tr><td>9</td><td>
MaskCtrl – Mask Control (formerly Write Enable Control). This field determines if 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. ChanEn is ignored in this case.

MaskCtrl = NoMask skips the check for PcIP[n] == ExIP before enabling a channel, as described in the Evaluate Write Enable section.
</td></tr>
<tr><td>8</td><td>
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.
</td></tr>
</table>

Bits	Description
	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*.

5.2.4 Instruction Destination Doubleword (DW1)

5.2.4.1 DW1 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. See the Region Parameters section and the sections following it for more information about those parameters.

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. See the Region Parameters section and the sections following it for more information about these parameters. **Programming Notes:** Allthough *Dst.HorzStride* is a don't care for Align16, HW needs this to be programmed as "01".
15	Reserved: MBZ
14:12	**Src1.SrcType – Source 1 Data Type.** This field specifies the numeric data type of the source operand src1. The bits of a source operand are interpreted as the identified numeric 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 numeric data types. Second, it has three packed vector types, the V, UV, and VF types. *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

Bits	Description
	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 110 = **“DF”**. **D**ouble precision **F**loat (64-bit) 111 = **“F”**. Single precision **F**loat (32-bit) Source Immediate Type Encoding: 000 = **“UD”** 001 = **“D”** 010 = **“UW”** 011 = **“W”** 100 **=** “UV”. 32-bit halfbyte **U**nsigned Integer **V**ector 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**#) 10 = **Reserved**. Reserved. Do not use this encoding. 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*.
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 numeric data type of the destination operand dst. The bits of the destination operand are interpreted as the identified numeric 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

Bits	Description
	011 = **"W"**. Signed **W**ord integer 100 = **"UB"**. **U**nsigned **B**yte integer 101 = **"B"**. Signed **B**yte integer 110 = Reserved 110 = **["DF"**] **D**ouble Precision **F**loat (64-bit) 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**#) 10 = **Reserved**. Reserved. Do not use this encoding. 11 = **reserved**

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

Destination Register Region in Direct + 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** – 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. For a *send* instruction, this field applies to **PostDst**.
4	**Dst.SubRegNum[4].** This is the 16-byte aligned sub-register address. For a *send* instruction, this field applies to **CurrDst**.
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**.

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.

Bits	Description
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.) **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field. For a *send* instruction, this field applies to **CurrDst**.

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. 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. 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**.

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**.

Bits	Description
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**.

5.2.4.2 DW1 3-src Instructions

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

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. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.
20:17	**Destination Channel Enable.** Four channel enables are defined for controlling which channels are 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. Mnemonics for the bit being set for the group of 4 are “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)
16:15	**Dst Type.** This field contains the data type for the destination. 00b = Single Precision Float 01b = DWord 10b = Unsigned DWord 11b = Double Precision Float
14:13	**Src Type.** This field contains the data type for all three sources. 00b = Single Precision Float 01b = DWord 10b = Unsigned DWord 11b = Double Precision Float
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.

Bits	Description
	Refer to Table 5-5 for the encoding.
3	Reserved: MBZ
2	**Flag Register Number.** This field contains the flag register number for instructions with a non-zero Conditional Modifier.
1	**Flag Subregister Number.** This field contains the flag subregister number for instructions with a non-zero Conditional Modifier.
0	**Reserved**

5.2.5 Instruction Source 0 Doubleword 2 (DW2)

5.2.5.1 DW2 1-src and 2-src Instructions

Instruction Source 0 Doubleword 2 (DW2) contains the first source operand and also flag register number.

- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Direct Addressing with Align16.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Direct Addressing with Align1.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Indirect Addressing with Align16.
- *Instruction Source 0 Doubleword 2 (DW2)* shows the field definition for Indirect Addressing with Align1.

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. 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.
15	**Src0.AddrMode – Source 0 Address Mode.** This field is the *AddrMode* for src0 operand. It is ignored if src0 is an immediate operand.

Bits	Description
14:13	**Src0.SrcMod – Source 0 Source Modifier.** This field is the *SrcMod* for source operand src0.
12:5	**Src0.RegNum – Source 0 Register Number** This is the *RegNum* field for source operand src0. 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. It is ignored if src0 is an immediate operand. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field. For example, using the F (Float) type the possible subregister numbers in Align16 mode are 0 or 4, corresponding to 0 or 1 for this field.
3:0	**Src0.ChanEn – Source 0 Channel Enable** This is the *ChanEn* field for source operand src0. It is ignored if src0 is an immediate operand.

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. It is ignored if src0 is an immediate operand.
20:18	**Src0.Width.** This is the *Width* field for source operand src0. It is ignored if src0 is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand src0. It is ignored if src0 is an immediate operand.
15	**Src0.AddrMode – Source 0 Address Mode.** This is the *AddrMode* for source operand src0. 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. 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. 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

Bits	Description
	src0. It is ignored if src0 is an immediate operand. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.

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. 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. It is ignored if src0 is an immediate operand.
15	**Src0.AddrMode – Source 0 Address Mode.** This is the *AddrMode* for source operand src0. 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. 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. 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. 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. It is ignored if src0 is an immediate operand.

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. It is ignored if src0 is an immediate operand.

Bits	Description
20:18	**Src0.Width.** This is the *Width* field for source operand src0. It is ignored if src0 is an immediate operand.
17:16	**Src0.HorzStride.** This is the *HorzStride* field for source operand src0. It is ignored if src0 is an immediate operand.
15	**Src0.AddrMode – Source 0 Address Mode.** This is the *AddrMode* for source operand src0. 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. 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. 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. It is ignored if src0 is an immediate operand.

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

Instruction DW2 and DW3 3-Source

DW	Bits	Description
DW3	31:30	Reserved: MBZ
	29:22	**Source2 Register Number.** This field contains the register number for source2.
	21:19	**Source2 Subregister Number.** This field contains the subregister number for source2. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.
	18:11	**Source2 Channel Select.** This field contains the swizzle control for source2. See ChanSel in the Common Instruction Fieldssection for a description of the Source Swizzle encodings.
	10:10	**Source2 Replication Control.** This field controls replication for source2. See RepCtrl in the Common Instruction Fieldssection for a description of the Source Replication Control encodings.
	9:9	Reserved: MBZ
	8:1	**Source1 Register Number.** This field contains the register number for source1.
	0	**Source1 Subregister Number.** This field contains the subregister number for source1. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.

DW	Bits	Description
DW2	31:30	**Source1 Subregister Number.** This field contains the subregister number for source1. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.
	29:22	**Source1 Channel Select.** This field contains the swizzle control for source1. See ChanSel in the Common Instruction Fieldssection for a description of the Source Swizzle encodings.
	21:21	**Source1 Replication Control.** This field controls replication for source1. See RepCtrl in the Common Instruction Fieldssection for a description of the Source Replication Control encodings.
	20:20	Reserved: MBZ
	19:12	**Source0 Register Number.** This field contains the register number for source0.
	11:9	**Source0 Subregister Number.** This field contains the subregister number for source0. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.
	8:1	**Source0 Channel Select.** This field contains the swizzle control for source0. See ChanSel in the Common Instruction Fieldssection for a description of the Source Swizzle encodings.
	0:0	**Source0 Replication Control.** This field controls replication for source0. See RepCtrl in the Common Instruction Fieldssection for a description of the Source Replication Control encodings.

5.2.6 Instruction Source 1 Doubleword 3 (DW3)

Instruction Source 1 Doubleword 3 (DW3) contains the second source operand (src1) and is used to hold the 32-bit immediate source (imm32 as src0 or src1). *Instruction Source 1 Doubleword 3 (DW3)* and *Instruction Source 1 Doubleword 3 (DW3)* 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, *Instruction Source 1 Doubleword 3 (DW3)* and *Instruction Source 1 Doubleword 3 (DW3)* define the fields in DW3 for indirectly addressed src1 in Align16 and Align1 modes.

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. 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. 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. It is ignored if src1 is an immediate operand.
12:5	**Src1.RegNum.** This field is the *RegNum* field for src1 operand. It is ignored if src1 is an immediate operand.
4	**Src1.SubRegNum[4].** This field is bit [4] of the *SubRegNum* field for src1. It is ignored if src1 is an immediate operand. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field. For example, using the F (Float) type the possible subregister numbers in Align16 mode are 0 or 4, corresponding to 0 or 1 for this field.
3:0	**Src1.ChanEn – Source 1 Channel Enable.** It is the channel enable field for src1. It is ignored if src1 is an immediate operand.

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. It is ignored if src1 is an immediate operand.
20:18	**Src1.Width.** This is the *Width* field for source operand src1. It is ignored if src1 is an immediate operand.
17:16	**Src1.HorzStride.** This is the *HorzStride* field for source operand src1. 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.

Bits	Description
	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. 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. It is ignored if src1 is an immediate operand. **Note:** The recommended instruction syntax uses GRF subregister numbers in units of element size, which the assembler translates to the appropriate value for this field.

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. 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. It is ignored if src1 is an immediate operand.
15	**Src1.AddrMode – Source 1 Address Mode** This is the *AddrMode* for source operand src1. 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. 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. 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. 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. It is ignored if src1 is an immediate operand.

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. It is ignored if src1 is an immediate operand.
20:18	**Src1.Width** This is the *Width* field for source operand src1. It is ignored if src1 is an immediate operand.
17:16	**Src1.HorzStride** This is the *HorzStride* field for source operand src1. It is ignored if src1 is an immediate operand.
15	**Src1.AddrMode – Source 1 Address Mode** This is the *AddrMode* for source operand src1. 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. 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. 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. It is ignored if src1 is an immediate operand.

5.3 EU Compact Instructions

On receiving an instruction with bit 29 (CmptCtrl) set, HW recognizes it as a 64-bit compact instruction. Hardware then uses the index fields inside the compact instruction to lookup values in the associated compaction tables, then uses the table outputs along with other fields in the compact instruction to reconstruct the 128-bit native-sized instruction.

In flow control instructions, IP offsets, such as the JIP and UIP instruction fields, are measured in 64-bit QWords. Thus a compact 64-bit instruction is 1 unit for IP offset calculations and a native 128-bit instruction is 2 units for IP offset calculations.

The native 128-bit instruction format provides access to all instruction options. Only some instruction options and combinations of instruction options can be represented in the compact instruction formats.

Which native instructions can be represented as compact instructions and the details of the compact instruction formats and the compaction tables used may change with each processor generation.

In the following instruction format tables the Mapping Bits and Mapping Description columns describe the mappings into native instruction fields.

5.3.1 EU Compact Instruction Format

The following table describes the EU compact instruction format. For this processor, instructions with three source operands cannot be compacted.

Compact Instruction Format

Bits	Size	Mapping Bits	Compact Name	Mapping Description
63:56	8	108:101 (Not Imm.) or 103:96 (Imm.)	Src1.RegNum	Src1.RegNum in 108:101 if not immediate. Imm32[7:0] in 103:96 if immediate.
55:48	8	76:69	Src0.RegNum	Src0.RegNum.
47:40	8	60:53	Dst.RegNum	Dst.RegNum.
39:35	5	120:109 (Not Imm.) or 127:104 (Imm.)	Src1Index	Lookup one of 32 12-bit values. If not an immediate operand, maps to bits 120:109, covering the Src1.AddrMode, Src1.ChanSel[7:4], Src1.HorzStride, Src1.SrcMod, Src1.VertStride, and Src1.Width bit fields. If an immediate operand, does not do any lookup. The 5-bit value directly maps to bits 108:104 (Imm32[12:8]) and the upper bit (bit 39 in the compact format, bit 108 in the native format) is replicated to provide bits 127:109 (Imm32[31:13]) in the native format.
34:30	5	88:77	Src0Index	Lookup one of 32 12-bit values. That value is used (from MSB to LSB) for the Src0.AddrMode, Src0.ChanSel[7:4], Src0.HorzStride, Src0.SrcMod, Src0.VertStride, and Src0.Width bit fields. Note that this field spans a DWord boundary within the QWord compacted instruction.
29	1	29	CmptCtrl	Compaction Control. The same in both the compact and native formats: 0: Regular instruction, not compacted. 1: Compacted instruction.
28	1	Not mapped.	Reserved	Not mapped. MBZ.
27:24	4	27:24	CondModifier	CondModifier. The same in both the compact and native formats.
23	1	28	AccWrCtrl	AccWrCtrl.
22:18	5	100:96, 68:64, 52:48	SubRegIndex	Lookup one of 32 15-bit values. That value is used (from MSB to LSB) for various fields for Src1, Src0, and Dst, including ChanEn/ChanSel, SubRegNum, and AddrImm[4] or AddrImm[4:0], depending on AddrMode and AccessMode.
17:13	5	63:61, 46:32	DataTypeIndex	Lookup one of 32 18-bit values. That value is used (from MSB to LSB) for the Dst.AddrMode, Dst.HorzStride, Dst.DstType, Dst.RegFile, Src0.SrcType, Src0.RegFile, Src1.SrcType, and Src1.RegType bit fields.
12:8	5	90:89, 31, 23:8	ControlIndex	Lookup one of 32 19-bit values. That value is used (from MSB to LSB) for the FlagRegNum, FlagSubRegNum, Saturate, ExecSize, PredInv, PredCtrl, ThreadCtrl, QtrCtrl, DepCtrl, MaskCtrl, and AccessMode bit fields.
6:0	7	6:0	Opcode	Opcode. The same in both the compact and native formats.

5.3.1.1 EU Instruction Compaction Tables

The following four tables describe the mappings for the ControlIndex, DataTypeIndex, SubRegIndex, Src0Index, and Src1Index fields in the compact instruction format.

ControlIndex Compact Instruction Field Mappings

ControlIndex	19-Bit Mapping	Mapped Meaning
0	0000000000000000010	Align1 | We | (1) | f0.0
1	0000100000000000000	Align1 | (4) | f0.0
2	0000100000000000001	Align16 | (4) | f0.0
3	0000100000000000010	Align1 | We | (4) | f0.0
4	0000100000000000011	Align16 | We | (4) | f0.0
5	0000100000000000100	Align1 | NoDDClr | (4) | f0.0
6	0000100000000000101	Align16 | NoDDClr | (4) | f0.0
7	0000100000000000111	Align16 | We | NoDDClr | (4) | f0.0
8	0000100000000001000	Align1 | NoDDChk | (4) | f0.0
9	0000100000000001001	Align16 | NoDDChk | (4) | f0.0
10	0000100000000001101	Align16 | NoDDClr, NoDDChk | (4) | f0.0
11	0000110000000000000	Align1 | Q1 | (8) | f0.0
12	0000110000000000001	Align16 | Q1 | (8) | f0.0
13	0000110000000000010	Align1 | We | Q1 | (8) | f0.0
14	0000110000000000011	Align16 | We | Q1 | (8) | f0.0
15	0000110000000000100	Align1 | NoDDClr | Q1 | (8) | f0.0
16	0000110000000000101	Align16 | NoDDClr | Q1 | (8) | f0.0
17	0000110000000000111	Align16 | We | NoDDClr | Q1 | (8) | f0.0
18	0000110000000001001	Align16 | NoDDChk | Q1 | (8) | f0.0
19	0000110000000001101	Align16 | NoDDClr, NoDDChk | Q1 | (8) | f0.0
20	0000110000000010000	Align1 | Q2 | (8) | f0.0
21	0000110000100000000	Align1 | Q1 | +f.xyzw | (8) | f0.0
22	0001000000000000000	Align1 | H1 | (16) | f0.0
23	0001000000000000010	Align1 | We | H1 | (16) | f0.0
24	0001000000000000100	Align1 | NoDDClr | H1 | (16) | f0.0
25	0001000000100000000	Align1 | H1 | +f.xyzw | (16) | f0.0
26	0010110000000000000	Align1 | Q1 | (8) | .sat | f0.0
27	0010110000000010000	Align1 | Q2 | (8) | .sat | f0.0
28	0011000000000000000	Align1 | H1 | (16) | .sat | f0.0
29	0011000000100000000	Align1 | H1 | +f.xyzw | (16) | .sat | f0.0
30	0101000000000000000	Align1 | H1 | (16) | f0.1
31	0101000000100000000	Align1 | H1 | +f.xyzw | (16) | f0.1

DataTypeIndex Compact Instruction Field Mappings

DataTypeIndex	18-Bit Mapping	Mapped Meaning
0	001000000000000001	r:ud | a:ud | a:ud | <1> | dir |
1	001000000000100000	a:ud | r:ud | a:ud | <1> | dir |
2	001000000000100001	r:ud | r:ud | a:ud | <1> | dir |
3	001000000001100001	r:ud | i:ud | a:ud | <1> | dir |
4	001000000010111101	r:f | r:d | a:ud | <1> | dir |
5	001000001011111101	r:f | i:vf | a:ud | <1> | dir |
6	001000001110100001	r:ud | r:f | a:ud | <1> | dir |
7	001000001110100101	r:d | r:f | a:ud | <1> | dir |
8	001000001110111101	r:f | r:f | a:ud | <1> | dir |

DataTypeIndex	18-Bit Mapping	Mapped Meaning
9	001000010000100001	r:ud | r:ud | r:ud | <1> | dir |
10	001000110000100000	a:ud | r:ud | i:ud | <1> | dir |
11	001000110000100001	r:ud | r:ud | i:ud | <1> | dir |
12	001001010010100101	r:d | r:d | r:d | <1> | dir |
13	001001110010100100	a:d | r:d | i:d | <1> | dir |
14	001001110010100101	r:d | r:d | i:d | <1> | dir |
15	001111001110111101	r:f | r:f | a:f | <1> | dir |
16	001111011110011101	r:f | a:f | r:f | <1> | dir |
17	001111011110111100	a:f | r:f | r:f | <1> | dir |
18	001111011110111101	r:f | r:f | r:f | <1> | dir |
19	001111111110111100	a:f | r:f | i:f | <1> | dir |
20	000000001000001100	a:w | a:ub | a:ud | <0> | dir |
21	001000000000111101	r:f | r:ud | a:ud | <1> | dir |
22	001000000010100101	r:d | r:d | a:ud | <1> | dir |
23	001000010000100000	a:ud | r:ud | r:ud | <1> | dir |
24	001001010010100100	a:d | r:d | r:d | <1> | dir |
25	001001110010000100	a:d | a:d | i:d | <1> | dir |
26	001010010100001001	r:uw | a:uw | r:uw | <1> | dir |
27	001101111110111101	r:f | r:f | i:vf | <1> | dir |
28	001111111110111101	r:f | r:f | i:f | <1> | dir |
29	001011110110101100	a:w | r:w | i:w | <1> | dir |
30	001010010100101000	a:uw | r:uw | r:uw | <1> | dir |
31	001010110100101000	a:uw | r:uw | i:uw | <1> | dir |

SubRegIndex Compact Instruction Field Mappings

SubRegIndex	15-Bit Mapping	Mapped Meaning
0	000000000000000	0 | 0 | 0 |
1	000000000000001	0.x | 0.xx | 0.xx
2	000000000010000	8 | 0 | 0 |
3	000000000001111	0.xyzw | 0.xx | 0.xx
4	000000000010000	16 | 0 | 0 |
5	000000010000000	0 | 4 | 0 |
6	000000100000000	0 | 8 | 0 |
7	000000110000000	0 | 12 | 0 |
8	000001000000000	0 | 16 | 0 |
9	000001000010000	16 | 16 | 0 |
10	000001010000000	0 | 20 | 0 |
11	001000000000000	0 | 0 | 4 |
12	001000000000001	0.x | 0.xx | 0.xy
13	001000010000001	0.x | 0.xy | 0.xy
14	001000010000010	0.y | 0.xy | 0.xy
15	001000010000011	0.xy | 0.xy | 0.xy
16	001000010000100	0.z | 0.xy | 0.xy
17	001000010000111	0.xyz | 0.xy | 0.xy
18	001000010001000	0.w | 0.xy | 0.xy
19	001000010001110	0.yzw | 0.xy | 0.xy
20	001000010001111	0.xyzw | 0.xy | 0.xy
21	001001100000000	0 | 12 | 4 |
22	001001111101000	0.w | 0.ww | 0.xy
23	010000000000000	0 | 0 | 8 |
24	010000110000000	0 | 12 | 8 |

SubRegIndex	15-Bit Mapping	Mapped Meaning
25	011000000000000	0 | 0 | 12 |
26	011110010000111	0.xyz | 0.xy | 0.ww
27	100000000000000	0 | 0 | 16 |
28	101000000000000	0 | 0 | 20 |
29	110000000000000	0 | 0 | 24 |
30	111000000000000	0 | 0 | 28 |
31	111000000011100	28 | 0 | 28 |

Src0Index or Src1Index Compact Instruction Field Mappings

Src0Index or Src1Index	12-Bit Mapping	Mapped Meaning
0	000000000000	dir | <0;1,0>
1	000000000010	(-) | dir | <0;1,0>
2	000000010000	dir | <0;>.zx
3	000000010010	(-) | dir | <0;>.zx
4	000000011000	dir | <0;>.wx
5	000000100000	dir | <0;>.xy
6	000000101000	dir | <0;>.yy
7	000001001000	dir | <0;4,1>
8	000001010000	dir | <0;>.zz
9	000001110000	dir | <0;>.zw
10	000001111000	dir | <0;8,4> / dir | <0;>.ww
11	001100000000	dir | <4;>.xx
12	001100000010	(-) | dir | <4;>.xx
13	001100001000	dir | <4;>.yx
14	001100010000	dir | <4;>.zx
15	001100010010	(-) | dir | <4;>.zx
16	001100100000	dir | <4;>.xy
17	001100101000	dir | <4;>.yy
18	001100111000	dir | <4;>.wy
19	001101000000	dir | <4;4,0>
20	001101000010	(-) | dir | <4;4,0>
21	001101001000	dir | <4;>.yz
22	001101010000	dir | <4;>.zz
23	001101100000	dir | <4;>.xw
24	001101101000	dir | <4;>.yw
25	001101110000	dir | <4;>.zw
26	001101110001	(abs) | dir | <4;>.zw
27	001101111000	dir | <4;>.ww
28	010001101000	dir | <8;8,1>
29	010001101001	(abs) | dir | <8;8,1>
30	010001101010	(-) | dir | <8;8,1>
31	010110001000	dir | <16;16,1>

5.4 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.

The 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.

Note: Opcodes appear in the overall Instruction Set Summary Table as well. The following subsections still serve the purpose of describing various instruction groups.

5.4.1 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*.

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 complement (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
11	0x0A	*Reserved*			

Opcode		Instruction	Description	#src	#dst
dec	hex				
12	0x0C	asr	Component-wise arithmetic shift right	2	1
13	0x0D	*Reserved*			
14	0x0E	*Reserved*			
15	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*			
19	0x13	***f32tof16***	Single precision float to half precision float conversion		
20	0x14	***f16to32***	Half precision float to single precision float conversion		
21	0x15	*Reserved*			
22	0x16	*Reserved*			
23	0x17	***bfrev***	Reverse bits	1	1
24	0x18	***bfe***	Bitfield exact	3	1
25	0x19	***bfi1***	Bitfield insert macro instruction 1, generate mask	2	1
26	0x1A	***bfi2***			
27-31	0x1B-0x1F	*Reserved*			

5.4.2 Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	jmpi	Jump indexed	1	0
33	0x21	brd	Branch - Diverging	1	0
34	0x22	if	If	0/2	0
35	0x23	brc	Branch - Converging	1	-
36	0x24	else	Else	1	0
37	0x25	endif	End if	0	0

Opcode		Instruction	Description	#src	#dst
dec	hex				
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	Reserved			
47	0x2F	***Reserved***			

5.4.3 Miscellaneous Instructions

This instruction group has an opcode format of 011xxxxb.

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***			

5.4.4 Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

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
75	0x4B	***fbh***	Find first 1 for UD from msb side, or first 1/0 for D.	1	1
76	0x4C	***fbl***	First first 1 for UD from lsb side	1	1
77	0x4D	***cbit***	Count bits set	1	1
78	0x4E	***addc***	Integer add with carry	2	1 + acc.
79	0x4F	subb	integer subtract with borrow	2	1 + acc.
75-79	0x4B-0x4F	***Reserved***			

5.4.5 Vector Arithmetic Instructions

This instruction group has an opcode format of 101xxxxb.

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	1
91	0x5B	fma(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	0x5D	*reserved*			
94-95	0x5D-0x5F	*reserved*			

5.4.6 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.

Special Instructions

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

5.5 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.

5.5.1 Instruction Groups

<Instruction>::=<UnaryInstruction>

|<BinaryAccInstruction>
|<BinaryInstruction>
|<TriInstruction>
|<JumpInstruction>
|<BranchLoopInstruction>
|<ElseInstruction>
|<BreakInstruction>
|<MaskControlInstruction
|<TriInstruction2>
|<CallInstruction>
|<BranchConvIntruction>
|<BranchDivInstruction>
|<MathInstruction>

|<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**”” | “**pln**”

|“**sad2**” | “**sada2**” | “**dp4**” | “**dph**” | “**dp3**” | “**dp2**”” | “**lrp**”” | “**bfi1**”” | “**addc**”” | “**subb**”

<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**”

<TriInstruction2> ::= <Predicate> <TriInst2> <ExecSize> dst <Src> <Src> <Src><InstOptions>

<TriInst2> ::= <TriOp> <ConditionalModifier><Saturate>

,<TriOp> ::= “bfe”| “bfi2”|”mad”

<BranchConvInstruction> ::= <Predicate> <BranchConvOp> <ExecSize>< RelativeLocation2>

<BranchConvOp> ::= “brc”

<BrancConvInstruction> ::= <Predicate> <BranchDivOp> <ExecSize>< RelativeLocation3>

<BranchDivOp> ::= “brd”

<CallInstruction> ::= <Predicate> <CallOp> <ExecSize>dst< RelativeLocation2>

<CallOp> ::= “call” |”CALLA”

<MathInstruction> ::= <Predicate> <MathInst> <ExecSize>< Dst>< Src>< Src><FC>

<MathInst> ::= <MathOp><Saturate>

<MathOp> ::= “math”

<FC> ::= “**INV**” | “**LOG**” | “**EXP**” | “**SQRT**” |”**RSQ**” | “**POW**” | “**SIN**” | “**COS**” | “**INT DIV**”

<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**”

5.5.2 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>

| <ChannelEnableReg>

|<ThreadControlReg>

|<PerformanceReg>

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

|<DirectMsgReg> | <IndirectMsgReg>

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

|<NullReg>

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

<CurrDst>::=<DirectAlignedMsgReg>

5.5.3 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>

|<ControlReg>

|<StateReg>

|<NotifyReg>

|<IPReg>

|<NullReg>

| <ChannelEnableReg>

|<ThreadControlReg>

|<PerformanceReg>

<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>

5.5.4 Address Registers

<AddrParam>::=<AddrReg> <ImmAddrOffset>

<ImmAddrOffset>::= “”

| “**,**” <ImmAddrNum>

5.5.5 Register Files and Register Numbers

Note: The recommended instruction syntax uses subregister numbers within the GRF in units of actual data element size, corresponding to the data type used. For example for the F (Float) type, the assembler syntax uses subregister numbers 0 to 7, corresponding to subregister byte addresses of 0 to 28 in steps of 4, the element size.

<DirectGenReg>::=<GenRegFile> <GenRegNum> <GenSubRegNum>

<IndirectGenReg>::=<GenRegFile> “**[**“ <AddrParam> “**]**”

<GenRegFile>::=“**r**”

<GenRegNum>:: =“**0**”…“**127**”

<GenSubRegNum>:: = “”

| “.0”…”.3” //incase of DF

| “**.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> ::= “f” <FlagRegNum> <FlagSubRegNum>

<FlagRegNum> :: = “0” | “1”

<FlagReg>::=“**f0**” <FlagSubRegNum>

<FlagSubRegNum>:: =“”

| “.**0**”...“.**1**”

<NotifyReg>::=“**n**” <NotifyRegNum>

<NotifyRegNum>:: =“**0**”...“**2**”

<StateReg>::=“**sr0**” <StateSubRegNum>

<StateSubRegNum>:: =“.**0**”... “.**1**”

<ControlReg>::=“**cr0**” <ControlSubRegNum>

<ControlSubRegNum>:: =“.**0**” ...“.**2**”

<IPReg>::=“**ip**”

<NullReg>::=“**null**”

<ThreadControlReg> ::= “tdr0”<ThreadCntrlSubRegNum>

<ThreadCntrlSubRegNum> ::= “.0”…“.7”

<PerformanceReg> ::= “tm0”

<ChannelEnableReg> ::= “ce0.0”

5.5.6 Relative Location and Stack Control

<RelativeLocation>::= <imm16>

<RelativeLocation2>::= <imm32> | <reg32>

<RelativeLocation3> ::= <imm16> | <reg32>

<LocationStackCtrl>::=<imm32>

5.5.7 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**”

5.5.8 Types

<SrcType> ::= “**:df**” | “**:f**” | “**:ud**” | “**:d**” | “**:uw**” | “**:w**” | “**:ub**” | “**:b**”

<SrcImmType> ::= <SrcType> | “**:v**” | “**:vf**” | “**:uv**”

<DstType> ::= <SrcType>

5.5.9 Write Mask

<WriteMask>::= “”

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

| “.” “**xy**” | “.” “**xz**” | “.” “**xw**” | “.” “**yz**” | “.” “**yw**” | “.” “**zw**”

| “.” “**xyz**” | “.” “**xyw**” | “.” “**xzw**” | “.” “**yzw**”

| “.” “**xyzw**”

5.5.10 Swizzle Control

<Swizzle>::=“”

| “.” <ChanSel>

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

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

5.5.11 Immediate Values

<ImmAddrNum>::=“-**512**”… “**511**”

<Imm64> ::= “0.0”… “±1.0*2-1024…1023” | “0”…”264-1” | “-263”…”263-1”

<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**”

5.5.12 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**”

| “.any32h”| “.all32h”

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**”

5.5.13 Instruction Options

<InstOptions> ::= “”

| “**{**” <InstOption> “**}**”

| “**{**” <InstOption> <InstOptionEx> “**}**”

<InstOptionEx> ::= “”

| “**,**” <InstOption> <InstOptionEx>

<InstOption> ::= <AccessMode>

| <AccWrCtrl>

| <ComprCtrl>

| <DependencyCtrl>

| <MaskCtrl>

| <SendCtrl>

| <ThreadCtrl>

<AccessMode> ::= “**Align1**” | “**Align16**”

<AccWrCtrl> ::= “**AccWrEn**”

<ComprCtrl> ::= “**SecHalf**” | “**Compr**”

<DependencyCtrl> ::= “**NoDDChk**” | “**NoDDClr**”

<MaskCtrl> ::= “**NoMask**”

<SendCtrl> ::= “**EOT**”

<ThreadCtrl> ::= “**Switch**”

+: | “**Atomic**”

Note for Assembler: Compression control “**Compr**” has a direct map to the binary instruction word. It may be omitted if the Assembler can determine whether an instruction is compressable.

5.6 Instruction Set Summary Tables

The columns in the following tables specify instruction mnemonics, hex opcodes, full names, instruction groups, the number of source operands, whether the instruction supports predication, any support for

source modifiers, an indication of supported data types, whether the instruction supports saturation, and any support for conditional modifiers.

See the separate Accumulator Restrictions table for information about how instructions are allowed to use accumulators.

N and Y indicate No (no support for a feature) and Yes (full support for a feature) respectively.

A SrcMod (source modifier) value of Y indicates that a numeric source modifier is allowed, optionally specifying abolute value, negation, or a forced negative value. The value N indicates no source modifier support.

A SrcMod value of ** indicates a numeric source modifier.

In the Src Types and Dst Type columns, Int means any integer type and * means such an extensive list of types that you must refer to the detailed instruction description.

Instruction Set Summary Table A to B (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
add	40	Addition	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
addc	4E	Integer Addition with Carry	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
and	05	Logic And	Move and Logic	2	Y	**	Int	Int	N	Equality only
asr	12	Arithmetic Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
avg	42	Average	Parallel Arithmetic	2	Y	Y	B, UB W, UW D, UD	B, UB W, UW D, UD	Y	Y
bfe	18	Bit Field Extract	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfi1	19	Bit Field Insert 1	Move and Logic	2	Y	N	UD, D	UD, D	N	N
bfi2	1A	Bit Field Insert 2	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfrev	17	Bit Field Reverse	Move and Logic	1	Y	N	UD	UD	N	N
brc	23	Branch Converging	Flow Control	0 or 1	Y	N	D		N	N
brd	21	Branch Diverging	Flow Control	0 or 1	Y	N	D		N	N
break	28	Break	Flow Control	0	Y	N			N	N

Instruction Set Summary Table C to E (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
call	2C	Call	Flow Control	0	Y	N		D, UD	N	N
cbit	4D	Count Bits Set	Move and Logic	1	Y	N	UB, UW, UD	UD	N	N
cmp	10	Compare	Move and Logic	2	Y	Y	*	*	N	Y
cmpn	11	Compare NaN	Move and Logic	2	Y	Y	*	*	N	Y
cont	29	Continue	Flow Control	0	Y	N			N	N
dp2	57	Dot Product 2	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dp3	56	Dot Product 3	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dp4	54	Dot Product 4	Vector Arithmetic	2	Y	Y	F	F	Y	Y
dph	55	Dot Product Homogeneous	Vector Arithmetic	2	Y	Y	F	F	Y	Y
else	24	Else	Flow Control	0	N	N			N	N
endif	25	End If	Flow Control	0	N	N			N	N

Instruction Set Summary Table F to L (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
f16to32	14	Half Precision Float to Single Precision Float	Move and Logic	1	Y	Y	W	F	Y	Y
f32to16	13	Single Precision Float to Half Precision Float	Move and Logic	1	Y	Y	F	W	Y	Y
fbh	4B	Find First Bit from MSB Side	Move and Logic	1	Y	N	D, UD	UD	N	N
fbl	4C	Find First Bit from LSB Side	Move and Logic	1	Y	N	UD	UD	N	N
frc	43	Fraction	Parallel Arithmetic	1	Y	Y	F	F	N	Y
halt	2A	Halt	Flow Control	0	Y	N			N	N
if	22	If	Flow Control	0	Y	N			N	N
illegal	00	Illegal	Special	0	N	N			N	N
jmpi	20	Jump Indexed	Flow Control	1	Y	N	D		N	N
line	59	Line	Vector Arithmetic	2	Y	Y	F	F	Y	Y
lrp	5C	Linear Interpolation	Vector Arithmetic	3	Y	Y	F	F	N	Y
lzd	4A	Leading Zero Detection	Move and Logic	1	Y	Y	D, UD	UD	Y	Y

Instruction Set Summary Table M to P (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
mac	48	Multiply Accumulate	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mach	49	Multiply Accumulate High	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mad	5B	Multiply Add	Parallel Arithmetic	3	Y	Y	*	*	Y	Y
math	38	Extended Math Function	Parallel Arithmetic	2	Y	N	*	*	Y	N
mov	01	Move	Move and Logic	1	Y	Y	*	*	Y	Y
movi	03	Move Indexed	Move and Logic	1	Y	Y	*	*	Y	N
mul	41	Multiply	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
nop	7E	No Operation	Special	0	N	N			N	N
not	04	Logic Not	Move and Logic	1	Y	**	Int	Int	N	Equality only
or	06	Logic Or	Move and Logic	2	Y	**	Int	Int	N	Equality only
pln	5A	Plane	Vector Arithmetic	2	Y	Y	F	F	Y	Y

Instruction Set Summary Table R to X (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
ret	2D	Return	Flow Control	1	Y	N	D, UD		N	N
rndd	45	Round Down	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rnde	46	Round to Nearest or Even	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndu	44	Round Up	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndz	47	Round to Zero	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
sad2	50	Sum of Absolute Difference 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sada2	51	Sum of Absolute Difference Accumulate 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sel	02	Select	Move and Logic	2	Y	Y	*	*	Y	Y
send	31	Send Message	Miscellaneous	1	Y	N	*	*	N	N
sendc	32	Conditional Send Message	Miscellaneous	1	Y	N	*	*	N	N
shl	09	Shift Left	Move and Logic	2	Y	Y	Int	Int	Y	Y
shr	08	Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
subb	4F	Integer Subtraction with Borrow	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
wait	30	Wait	Miscellaneous	1	N	N	UD	UD	N	N
while	27	While	Flow Control	0 or 2	Y	N	*	*	N	Y
while	27	While	Flow Control	0	Y	N			N	N
xor	06	Logic Xor	Move and Logic	2	Y	**	Int	Int	N	Equality only

5.7 Accumulator Restrictions

This section describes restrictions on accumulator access: general restrictions, restrictions for specific instructions, and how those specific restrictions vary for processor generations. See Accumulator Registersfor a description of the accumulator registers.

Accumulator registers can be accessed as explicit source or destination operands, as an implicit source value when specified for a particular instruction (*sada2* for example), and as an implicit destination when the AccWrEn instruction option is used.

These general rules apply to accumulator access:

1. Flow control, *send*, *sendc*, and *wait* instructions cannot use accumulators.
2. Instructions with three source operands cannot use explicit accumulator operands. AccWrEn may be allowed for implicitly updating the accumulator.
3. Instructions that use the accumulator as an implicit source value cannot specify an explicit accumulator source operand.
4. Instructions that specify an implicit accumulator destination (with AccWrEn) cannot specify an explicit accumulator destination operand.
5. An instruction with both an explicit accumulator source operand and an explicit accumulator destination operand must specify the same accumulator register as the source and the destination.

These descriptions are frequently used in this table:

- No restrictions.
- No accumulator access, implicit or explicit.
- Source operands cannot be accumulators.
- Source modifier is not allowed if source is an accumulator.
- Accumulator is an implicit source and thus cannot be an explicit source operand.
- Accumulator cannot be destination, implicit or explicit.
- AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

These minor cases occur occasionally in the table:

- Integer source operands cannot be accumulators.
- No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
- An accumulator can be a source or destination operand but not both.

A few instructions use more than one of the listed restrictions.

Accumulator Restrictions

Instruction	Description
add	No restrictions.
addc	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
and	Source modifier is not allowed if source is an accumulator.
asr	No restrictions.

Instruction	Description
avg	No restrictions.
bfe	No accumulator access, implicit or explicit.
bfi1	No accumulator access, implicit or explicit.
bfi2	No accumulator access, implicit or explicit.
bfrev	No accumulator access, implicit or explicit.
cbit	No accumulator access, implicit or explicit.
cmp	Accumulator cannot be destination, implicit or explicit.
cmpn	Accumulator cannot be destination, implicit or explicit.
dp2	Source operands cannot be accumulators.
dp3	Source operands cannot be accumulators.
dp4	Source operands cannot be accumulators.
dph	Source operands cannot be accumulators.
f16to32	No accumulator access, implicit or explicit.
f32to16	No accumulator access, implicit or explicit.
fbh	No accumulator access, implicit or explicit.
fbl	No accumulator access, implicit or explicit.
frc	No restrictions.
line	Source operands cannot be accumulators.
lrp	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
lzd	Accumulator cannot be destination, implicit or explicit.
mac	Accumulator is an implicit source and thus cannot be an explicit source operand.
mach	Accumulator is an implicit source and thus cannot be an explicit source operand. AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
mad	No explicit accumulator access

Instruction	Description
	because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
math	No accumulator access, implicit or explicit.
mov	An accumulator can be a source or destination operand but not both.
movi	Source operands cannot be accumulators.
mul	Source operands cannot be accumulators.
not	Source modifier is not allowed if source is an accumulator.
or	Source modifier is not allowed if source is an accumulator.
pln	Source operands cannot be accumulators.
rndd	No accumulator access, implicit or explicit.
rnde	No accumulator access, implicit or explicit.
rndu	No accumulator access, implicit or explicit.
rndz	No accumulator access, implicit or explicit.
sad2	Source operands cannot be accumulators.
sada2	Source operands cannot be accumulators.
sel	No restrictions.
shl	Accumulator cannot be destination, implicit or explicit.
shr	No restrictions.
subb	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.
xor	Source modifier is not allowed if source is an accumulator.

6. Instruction Set Reference

This chapter describes the functions of instructions. Each instruction is sorted in alphabetical order according to assembly language mnemonic.

6.1 Conventions

This section describes conventions used in instruction reference pages.

For each instruction that has source or destination types, a table lists the allowed type combinations and may also indicate the processor generations that support certain combinations. A notation like *W indicates that UW and W are both allowed. Multiple types listed together mean that any combination (Cartesian product) of the listed types is allowed.

If a source operand is floating-point, all source operands must have the same floating-point data type.

Accumulator restrictions are described in the Accumulator Restrictions section and also appear in instruction descriptions.

6.1.1 Pseudo Code Format

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

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

Square brackets “[ ]” indicate that a field is optional. Saturation modifiers and instruction options are omitted for simplicity.

6.1.2 General Macros and Definitions

INST_MIN_SIZE is defined as a constant of 8 bytes.

```
#define INST_MIN_SIZE  8  // Instruction minimum size in bytes (for the compact instruction format)
```

The floor function converts a floating point value to an integral floating point value. For a given floating point value, from its closest two integral float values, floor returns the one that is closer to 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)
}
```

The Condition function takes the conditional signals {SN, ZR, OF, IN, NC} of result, generates a Boolean value according to a conditional evaluation controlled by the conditional modifier cmod, and returns the Boolean.

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

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

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

The Jump function jumps the instruction sequence from the current instruction location by InstCount 8-byte units, where each 16-byte native instruction is two units and each 8-byte compact instruction is one unit. If InstCount is positive and greater than zero, is an unconditional jump forward. If InstCount is negative, is an unconditional jump backward. If InstCount is zero, IP stays on the current instruction in an infinite loop.

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

6.2 Evaluate Write Enable

The WrEn should be evaluated as below.

Note: MaskCtrl = NoMask (1) skips the check for PcIP[n] == ExIP before enabling a channel.

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

for ( n = exec_size; n < 32; n++ ) {
  WrEn[n] = 0;
}
```

6.3 Instruction Description

The rest of the chapter contains the description of instructions.

6.4 add – Addition

Opcode	Instruction	Description
64 (0x40)	add dst src0 src1	Add src0 and src1 storing the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
*B, *W, *D	*B, *W, *D
*B, *W, *D	F
F	F
DF	DF

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] ) {
    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 *add – Addition* or *add – Addition*).

Note: Use a source modifier with *add* to implement subtraction.

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/ -denorm/ -finite^	–0/ -denorm^	+0/ -denorm^	+0/ +denorm/ -denorm^	B	+inf	NaN

–0	–inf	B	–0/ -denorm^	–0	+0	+0/ +denorm	B	+inf	NaN
+0	–inf	B	+0/ -denorm^	+0	+0	+0/ +denorm	B	+inf	NaN
+denorm	–inf	B	+0/ +denorm/ -denorm^	+0/ +denorm^	+0/ +denorm^	+0/ +denorm/ +finite^	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:	
^	Non zero results are applicable when denorm is enabled.
*	Result can be { –finite}.
**	Result can be {–finite, –0, +0, +finite}.
***	Result can be { +finite}.

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 A or B is {–inf, +inf, NaN}.

6.5 addc – Integer Addition with Carry

Opcode	Instruction	Description
78 (0x4E)	addc dst src0 src1	Add 32-bit unsigned integers src0 and src1. Store the 32-bit result in dst. Store the carry out (0 or 1) as a 32-bit value in the accumulator.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	N

Src Types	Dst Type
UD	UD

Format:

```
[(pred)] addc[.cmod] (exec_size) dst src0 src1
```

Syntax:

```
[(pred)] addc[.cmod] (exec_size) reg reg reg
 [(pred)] addc[.cmod] (exec_size) reg reg imm32
```

Pseudocode:

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

Description:

The *addc* instruction performs component-wise addition of src0 and src1 and stores the results in dst; it also stores the carry into acc.

If the operation produces a carry out, 0x00000001 is stored in acc, else 0x00000000 is stored in acc.

Restrictions:

AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

6.6 and – Logic 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
Y	N	Y	Y

Generation	Src Types	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
    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.

Register source operands can use source modifiers:

- Any source modifier is numeric, optionally changing a source value s to -s, abs(s), or -abs(s) before the AND operation.

This operation does not produce sign or overflow conditions. Only the **.e/.z** or **.ne/.nz** conditional modifiers should be used.

Restrictions:

Source modifier is not allowed if source is an accumulator.

6.7 asr – Arithmetic Shift Right

Opcode	Instruction	Description
12 (0x0C)	asr dst src0 src1	Arithmetic shift the bits in src0 right by the number of bits indicated in src1. Store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Generation	Src Types	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
     shiftCnt = src1.chan[n] & 0x1F; // Always use low 5 bits for shift count.     if
(src0.chan[n] >= 0) {
       dst.chan[n] = src0.chan[n] >> shiftCnt;
     }
     else {
       int maskLSB = pow(2, shiftCnt) - 1;
       if ( maskLSB & src0.chan[n] == 0 ) {
         dst.chan[n] = sign(src0.chan[n]) * ((abs)src0.chan[n] >> shiftCnt);
```

```
        }
        else {
          dst.chan[n] = sign(src0.chan[n]) * ((abs)src0.chan[n] >> shiftCnt) - 1;
        }
      }
    }
  }
```

Description:

Perform component-wise arithmetic right shift of the bits in src0 by the shift count indicated in src1, storing the results in dst. If src0 has a signed type, insert copies of src0's sign bit in the number of MSBs indicated by the shift count. Otherwise insert 0 bits.

The shift count is taken from the low five bits of src1, regardless of the src1 type and treated as an unsigned integer in the range 0 to 31.

For positive values, this operation is src0 / $2^{shiftCount}$ and for negative values, this operation is src0 / $2^{shiftCount}$ - 1.

Note: If src0 is -1, the result is -1 regardless of the shift count.

Note: For unsigned src0 types, *asr* and *shr* produce the same result.

6.8 avg – Average

Opcode	Instruction	Description
66 (0x42)	avg dst src0 src1	Integer average of two source operands, rounding up. Store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
*B, *W, *D	*B, *W, *D

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] ) {
     dst.chan[n] = (src0.chan[n] + src1.chan[n] + 1) >> 1;  // Use arithmetic shift
right.
   }
 }
```

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.

6.9 bfe – Bit Field Extract

Opcode	Instruction	Description
24 (0x18)	bfe dst src0 src1 src2	Extract a bit field defined by src0 and src1 from src2 and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
UD	UD
D	D

Format:

```
[(pred)] bfe (exec_size) dst src0 src1 src2
```

Syntax:

```
[(pred)] bfe (exec_size) reg reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     UD width = src0.chan[n][4:0];
     UD offset = src1.chan[n][4:0];
     if ( width == 0 ) {
       dst.chan[n] = 0x00000000;
     }
     else if ( (width + offset) < 32 ) {
       dst.chan[n] = src2.chan[n] << (32 - width - offset);
       if (src2 is signed) {
         dst.chan[n] = dst.chan[n] >> (32 - width); // pad sign bit of dst.chan
       }
       else {
         dst.chan[n] = dst.chan[n] >> (32 - width); // pad 0
       }
     }
     else {
       if ( src2 is signed ) {
         dst.chan[n] = src2.chan[n] >> offset; // pad sign bit
       }
        else {
         dst.chan[n] = src2.chan[n] >> offset; // pad 0
       }
     }
   }
```

```
}
```

Description:

Component-wise extract a bit field from src2 using the bit field width from src0 and the bit field offset from src1. Store the extracted bit field value in the low bits of dst and sign extend (if D type) or zero extend (if UD type).

The width and offset values are from the low five bits of src0 and src1 respectively, or src0 & 0x1f and src1 & 0x1f.

If width is zero, the result is zero.

If offset + width > 32 then the extracted bit field is bits offset to 31 of src2, extracting only 32 - offset bits, less than width as the bit field cannot extend past the MSB of the source value. Otherwise extract width bits extending from bit positions offset to offset + width - 1.

Restrictions:

No accumulator access, implicit or explicit.

All three-source instructions have certain restrictions, described in Instruction Machine Formats.

6.9.1 bfi1 – Bit Field Insert 1

Opcode	Instruction	Description
25 (0x19)	bfi1 dst src0 src1	Create a bit mask defined by src0 and src1 and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Types	Dst Type
UD	UD
D	D

Format:

```
[(pred)] bfi1 (exec_size) dst src0 src1
```

Syntax:

```
[(pred)] bfi1 (exec_size) reg reg reg
 [(pred)] bfi1 (exec_size) reg reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     UD width = src0.chan[n][4:0];
     UD offset = src1.chan[n][4:0];
     dst = ((1 << width) - 1) << offset;
   }
 }
```

Description:

The *bfi1* instruction is the first instruction in a two-instruction macro for bfi (Bit Field Insert).

The *bfi1* instruction component-wise generates mask with control from src0 and src1 and stores the results in dst. The mask is used in the bfi2 instruction to generate the final result of bfi.

Create a bit mask corresponding to the bit field width and offset in src0 and src1. Store the bit mask in dst. The mask has all bits in the bit field set to 1 and all other bits as 0.

The width and offset values are from the low five bits of src0 and src1 respectively, or src0 & 0x1f and src1 & 0x1f.

If width is zero, the result is zero.

The bfi macro has four source operands: src0 - bit field width in low five bits, src1 - bit field offset/starting bit position in low five bits, src2 - bit field value to insert, using only the number of least significant bits given by width in src0, and src3 - overall value into which the bit field is inserted, providing all bits other than the inserted bits for the result value.

```
bfi dst src0 src1 src2 src3

// Translates to these two instructions:
bfi1 dst src0 src1
bfi2 dst dst src2 src3
```

Restrictions:

No accumulator access, implicit or explicit.

6.10 bfi2 – Bit Field Insert 2

Opcode	Instruction	Description
26 (0x1A)	bfi2 dst src0 src1 src2	Extract a bit field defined by src0 and src1 from src2 and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
UD	UD
D	D

Format:

```
[(pred)] bfi2 (exec_size) dst src0 src1 src2
```

Syntax:

```
[(pred)] bfi2 (exec_size) reg reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     UD offset = LZD(reverse(src0.chan[n]))-1;
```

```
     // offset is the number of LSB zero bits below the bit mask which has all 1s.
     // width (implied by the logic) is the number of 1 bits in the mask value, which
should be all 1s.
     dst.chan[n] = ((src1.chan[n] << offset) & src0.chan[n]) | (src2.chan[n] & !
src0.chan[n]);
 }
```

Description:

The *bfi2* instruction is the second instruction in a two-instruction macro for bfi (Bit Field Insert).

The *bfi2* instruction component-wise performs the bitfield insert operation on src1 and src2 based on the mask in src0.

Use the mask in src0 to take a bit field value from the low bits of src1 and combine it with the value from src2 (so src2 provides all bits other than those masked out and replaced by the bit field value). Store the result in dst.

The bfi macro has four source operands: src0 - bit field width in low five bits, src1 - bit field offset/starting bit position in low five bits, src2 - bit field value to insert, using only the number of least significant bits given by width in src0, and src3 - overall value into which the bit field is inserted, providing all bits other than the inserted bits for the result value.

```
bfi dst src0 src1 src2 src3

// Translates to these two instructions:
bfi1 dst src0 src1
bfi2 dst dst src2 src3
```

Restrictions:

No accumulator access, implicit or explicit.

All three-source instructions have certain restrictions, described in Instruction Machine Formats.

6.11 bfrev – Reverse Bits

Opcode	Instruction	Description
23 (0x17)	bfrev dst src0	Reverse all bits in src0 and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
UD	UD

Format:

```
[(pred)] bfrev (exec_size) dst src0
```

Syntax:

```
[(pred)] bfrev (exec_size) reg reg
 [(pred)] bfrev (exec_size) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     for ( idx = 0; idx < 32; idx++ ) {
       dst.chan[n][idx] = src0.chan[n][31-idx];
     }
   }
 }
```

Description:

The *bfrev* instruction component-wise reverses all the bits in src0 and stores the results in dst.

Restrictions:

No accumulator access, implicit or explicit.

6.12 brc – Branch Converging

Opcode	Instruction	Description
35 (0x23)	brc JIP UIP	Redirect channel execution to the instruction referenced by UIP.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Types	Dst Type
D	

Format:

```
[(pred)] brc (exec_size) JIP UIP
```

Syntax:

```
[(pred)] brc (exec_size) imm16 imm16
[(pred)] brc (exec_size) reg32
```

Pseudocode:

```
Evaluate(WrEn);
for ( n = 0; n < 32; n++ ) {
  if ( WrEn[n] ) {
    PcIP[n] = IP + UIP;
  }
  else {
    PcIP[n] = IP + 1;
  }
}
if ( all PcIP != IP + 1 ) {  // for all channels
  Jump(IP + JIP);
}
```

Description:

The *brc* instruction redirects the execution forward or backward to the instruction pointed by (current IP + offset). The jump will occur if *all* channels are branched away.

UIP should reference the instruction where all channels are expected to come together. JIP should reference the end of the innermost conditional block.

Immediate Offset

Bits	Description
31:16	**UIP (Update Target Offset)** The relative offset in 64-bit units if a jump is taken for the channel. Format = S15.
15:0	**JIP (Jump Target Offset)** The relative offset in 64-bit units if a jump is taken for the instruction. Format = S15.

Register Offset

Bits	Description
31:16	**UIP (Update Target Offset)** The relative offset in 64-bit units if a jump is taken for the channel. Format = S15.
15:0	**JIP (Jump Target Offset)** The relative offset in 64-bit units if a jump is taken for the instruction. Format = S15.

In the architecture binary, JIP and UIP are at location src1 when immediates and at location src0 when reg32, where reg32 is accessed as a scalar DWord containing both JIP and UIP. The null register must be used (for example, by the assembler) as dst. When offsets are immediate, src0 must be null.

Restrictions:

A *brc* instruction must use the **Switch** instruction option.

Erratum: *brc* instructions cannot be used with offsets in registers, but only with immediate offsets.

Erratum: *brc* instructions cannot be used within if-endif or if-else-endif conditional blocks.

6.13 brd – Branch Diverging

Opcode	Instruction	Description
33 (0x21)	brd JIP	Redirect program execution to the instruction referenced by JIP.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Types	Dst Type
D	

Format:

```
[(pred)] brd (exec_size) JIP
```

Syntax:

```
 [(pred)] brd (exec_size) imm16
  [(pred)] brd (exec_size) reg32
```

Pseudocode:

```
Evaluate(WrEn);
for ( n = 0; n < 32; n++ ) {
  if ( WrEn[n] ) {
    PcIP[n] = IP + JIP;
  }
  else {
    PcIP[n] = IP + 1;
  }
}

if ( any PcIP == ExIP + JIP ) {  // any channel
  Jump(ExIP + JIP);
}
```

Description:

The *brd* instruction redirects the execution forward or backward to the instruction pointed by (current IP + offset). The jump will occur if *any* channels are branched away.

Immediate Offset]

Bits	Description
31:16	Reserved. MBZ.
15:0	**JIP (Jump Target Offset)** The relative offset in 64-bit units if a jump is taken for the instruction. Format = S15.

Register Offset

Bits	Description
31:16	Reserved. MBZ.
15:0	**JIP (Jump Target Offset)** The relative offset in 64-bit units if a jump is taken for the instruction. Format = S15.

In GEN binary, JIP is at location src1 when immediate and at location src0 when reg32, where reg32 is accessed as a scalar DWord. The null register must be used at dst locations.

Restrictions:

A *brd* instruction must use the **Switch** instruction option.

Erratum: *brd* instructions cannot be used with offsets in registers, but only with immediate offsets.

6.14 break – Break

Opcode	Instruction	Description
40 (0x28)	break JIP UIP	Terminate enabled execution channels and conditionally break out from the inner most loop.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

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] ) {
    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 most switch block.

When used in a loop, upon execution, the *break* instruction terminates the loop for all execution channels enabled. If all the enabled channels hit the *break* instruction, jump to the instruction referenced by JIP. JIP should be the offset to the end of the inner most conditional or loop block, UIP should be the offset to the *while* instruction of the loop block.

]: The following table describes the two 16-bit instruction pointer offsets. 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).

Relative Instruction Offsets]

Bits	Description
31:16	**UIP (Update Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the channel.

Bits	Description
	Format = S15.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

If SPF is ON, the UIP must be used to update IP; JIP is not used in this case.

Restrictions:

The execution size must be the same for the *while*, *break*, and *cont* instructions of the same code block.

6.15 call – Call

Opcode	Instruction	Description
44 (0x2C)	call dst JIP	Subroutine call.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Types	Dst Type
	D, UD

Format:

```
[(pred)] call (exec_size) dst JIP
```

Syntax:

```
[(pred)] call (exec_size) reg imm16
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if (WrEn.chan[n] ) {
     PcIP[n] = IP + JIP;
     CallMask[n] = 1;
   }
   else {
     PcIP[n] = IP + 1;
     CallMask[n] = 0;
   }
 }

 if ( PcIP[n] != (IP + 1) ) {  // any channel jumped
   dst.chan[0] = IP + 1;
   dst.chan[1] = CallMask;
   Jump(IP + JIP);
 }
```

Description:

The *call* instruction jumps to a subroutine. It can be predicated or non-predicated. If non-predicated, all enabled channels jump to the subroutine. If predicated, only the channels enabled by PMask jump to the subroutine; the rest of the channels move to the next instruction after the *call* instruction. If none of the channels jump into the subroutine, the *call* instruction is treated as a *nop*.

In case of a jump, the *call* instruction stores the return IP onto the first DWord of the destination register and stores the CallMask in the second DWord of the destination register.

When SPF is on, the predication control must be scalar.

The following table describes JIP, the jump offset, for. JIP must be a signed immediate operand. When a jump occurs, this value is added to IP pre-increment.

Immediate Instruction Offset

Bits	Description
31:16	Reserved: MBZ
15:0	**JIP (Jump Instruction Count).** The jump distance in number of 64-bit units if a jump is taken for the instruction. Format = S15.

Restrictions:

The *call* instruction must have DWord source and destination type, and the destination must be QWord aligned.

] The source0 regioning control must be <2;2,1>.

The execution size must be 2.

Workaround: A *call* instruction before a *send* with EOT should:

- Have no predication.
- Should use {NoMask}.
- Should specify for the destination any general register that is not part of the final send.
- Should specify a JIP value referencing the *send* instruction that has EOT.

Example:

```
call (2) r1 0x2 {NoMask}
send ... {EOT}
```

6.16 cbit – Count Bits Set

Opcode	Instruction	Description
77 (0x4D)	cbit dst src0	Count the number of bits set (1) in src0 and store that count in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
UB, UW, UD	UD

Format:

```
[(pred)] cbit (exec_size) dst src0
```

Syntax:

```
[(pred)] cbit (exec_size) reg reg
 [(pred)] cbit (exec_size) reg imm32
```

Pseudocode:

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

Description:

The *cbit* instruction counts component-wise the total bits set in src0 and stores the resulting counts in dst.

Restrictions:

No accumulator access, implicit or explicit.

6.17 cmp – Compare

Opcode	Instruction	Description
16 (0x10)	cmp.cmod dst src0 src1	Compare src0 and src1 as specified by the conditional modifier and store the resulting flags in dst and in the flag register.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Src Types	Dst Type
*B, *W, *D	*B, *W, *D
*B, *W, *D	F
F	F
DF	DF

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] ) {
    results[n] = src0.chan[n] - src1.chan[n];
    bitMask[n] = Condition(results[n]);
    dst.chan[n] = bitMask[n];  // All bits for dst channel
  }
}
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 all bits of dst channels. If the dst is not null, for the enabled channels, then all bits of the destination channel will contain the flag value for the channel. When the instruction operates on packed word format, one general register may store up to 16 such comparison results. In DWord format, one general register may store up to 8 results.

A conditional modifier must be specified; the conditional modifier field cannot be 0000b. The comparison does not use the NS (NaN source) signals, as described in the Creating Conditional Flags section. Accordingly the conditional modifier should not be **.u** (unordered).

For each enabled channel 0b or 1b is assigned to the appropriate flag bit and 0/all zeros or all ones (e.g, byte 0xFF, word 0xFFFF, DWord 0xFFFFFFFF) is assigned to dst.

When any source type is floating-point, the *cmp* instruction obeys the rules described in the tables in the Floating Point Modes section of the Data Types chapter.

Restrictions:

Accumulator cannot be destination, implicit or explicit. The destination must be a general register or the null register.

A SIMD16 instruction is not allowed for DWord data types. Use two SIMD8 instructions.

If the destination is the null register, the {Switch} instruction option must be used.

6.18 cmpn – Compare NaN

Opcode	Instruction	Description
17 (0x11)	cmpn.cmod dst src0 src1	Compare src0 and src1 with a special NaN comparison as specified by the conditional modifier and store the resulting flags in dst and in the flag register.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Src Types	Dst Type
*B, *W, *D	*B, *W, *D
*B, *W, *D	F
F	F
DF	DF

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 ( WrEn.chan[n] ) {
    results[n] = src0.chan[n] - src1.chan[n];
    bitMask[n] = ConditionNaN(results[n]);
    dst.chan[n][0] = bitMask[n];  // All bits for dst channel
  }
}
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 all bits of dst channels. If the dst is not null, for the enabled channels, then all bits of the destination channel will contain the flag value for the channel. When the instruction operates on packed word format, one general register may store up to 16 such comparison results. In DWord format, one general register may store up to 8 results.

A conditional modifier must be specified; the conditional modifier field cannot be 0000b. More information about the conditional signals used is in the Creating Conditional Flags section.

For each enabled channel 0b or 1b is assigned to the appropriate flag bit and 0/all zeros or all ones (e.g, byte 0xFF, word 0xFFFF, DWord 0xFFFFFFFF) is assigned to dst.

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 the *Min/Max of Floating Point Numbers* section for details).

Restrictions:

Accumulator cannot be destination, implicit or explicit. The destination must be a general register or the null register.

]: A SIMD16 instruction is not allowed for DWord data types. Use two SIMD8 instructions.

]: If the destination is the null register, the {Switch} instruction option must be used.

6.19 cont – Continue

Opcode	Instruction	Description
41 (0x29)	cont JIP UIP	Temporarily disable enabled execution channels for the remainder of the inner most loop and conditionally jump to the last instruction (*while*) of the loop.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

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] ) {
     if ( PMask[n] ) {  // PMask is for all channels enabled for the cont instruction.
       PcIP[n] = IP + UIP;
     }
     else {
       PcIP[n] = IP + 1;
     }
   }
 }
 for ( n = exec_size; n < 32; n++ ) {
   PcIP[n] = IP + 1;
 }
 if ( PcIP != (IP + 1) ) {  // all channels 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. If all enabled channels hit this instruction, jump to the instruction referenced by JIP where execution continues.

UIP should always reference the loop's associated *while* instruction. 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 are the same.

The following table describes the two 16-bit instruction pointer offsets. 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).

Relative Instruction Offsets

Bits	Description
31:16	**UIP (Update Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the channel. Format = S15.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

6.20 dp2 – Dot Product 2

Opcode	Instruction	Description
87 (0x57)	dp2 dst src0 src1	Perform two-wide dot product using the first two elements of each four-tuple from src0 and src1 and store the four-wide replicated results in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
F	F

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] ) dst.chan[n] = fTmp;
   if ( WrEn.chan[n+1] ) dst.chan[n+1] = fTmp;
   if ( WrEn.chan[n+2] ) dst.chan[n+2] = fTmp;
   if ( WrEn.chan[n+3] ) 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 element of src0 (post-source-swizzle if present) are not involved in the computation.

The dot product of two vectors of equal length is the sum of the products of each pair of corresponding elements.

The *dp4* instruction includes all four elements of each vector in the dot product. The *dp3* instruction includes the first three elements of each vector in the dot product.

Restrictions:

Execution size cannot be less than 4.

Horizontal strides must be 1.

Source operands cannot be accumulators.

6.21 dp3 – Dot Product 3

Opcode	Instruction	Description
86 (0x56)	dp3 dst src0 src1	Perform three-wide dot product using the first three elements of each four-tuple from src0 and src1 and store the four-wide replicated results in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
F	F

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] ) dst.chan[n] = fTmp;
   if ( WrEn.chan[n+1] ) dst.chan[n+1] = fTmp;
   if ( WrEn.chan[n+2] ) dst.chan[n+2] = fTmp;
   if ( WrEn.chan[n+3] ) 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.

The dot product of two vectors of equal length is the sum of the products of each pair of corresponding elements.

The *dp4* instruction includes all four elements of each vector in the dot product. The *dp2* instruction includes the first two elements of each vector in the dot product.

Restrictions:

Execution size cannot be less than 4.

Horizontal strides must be 1.

Source operands cannot be accumulators.

6.22 dp4 – Dot Product 4

Opcode	Instruction	Description
84 (0x54)	dp4 dst src0 src1	Perform four-wide dot product of src0 and src1 and store the four-wide replicated results in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
F	F

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] ) dst.chan[n] = fTmp;
   if ( WrEn.chan[n+1] ) dst.chan[n+1] = fTmp;
   if ( WrEn.chan[n+2] ) dst.chan[n+2] = fTmp;
   if ( WrEn.chan[n+3] ) 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.

The dot product of two vectors of equal length is the sum of the products of each pair of corresponding elements.

Restrictions:

Execution size cannot be less than 4.

Horizontal strides must be 1.

Source operands cannot be accumulators.

6.23 dph – Dot Product Homogeneous

Opcode	Instruction	Description
85 (0x55)	dph dst src0 src1	Perform four-wide homogeneous dot product of src0 and src1 and store the four-wide replicated results in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
F	F

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];    // Use 1.0f in place of src0.chan[n+3].
   if ( WrEn.chan[n] ) dst.chan[n] = fTmp;
   if ( WrEn.chan[n+1] ) dst.chan[n+1] = fTmp;
   if ( WrEn.chan[n+2] ) dst.chan[n+2] = fTmp;
   if ( WrEn.chan[n+3] ) 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.

Use the *dp4* instruction to do a four-wide dot product that includes all elements of src0 and src1.

Restrictions:

Execution size cannot be less than 4.

Horizontal strides must be 1.

Source operands cannot be accumulators.

6.24 else – Else

Opcode	Instruction	Description
36 (0x24)	else JIP	An optional statement within an if/else/endif block of code.

Pred	Sat	Cond Mod	Src Mod
N	N	N	N

Format:

```
else (exec_size) JIP
```

Syntax:

```
else (exec_size) imm16
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < 32; n++ ) {
   if ( WrEn.channel[n] ) {
     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 before entering the *if/endif* block remain inactive throughout the entire block.

All enabled channels upon arriving at the *else* instruction are 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 src1 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.

Relative Instruction Offset

Bits	Description
31:16	**Reserved**: MBZ.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

Restrictions:

Predication is not allowed.

The execution size must be the same for the *if*, *else*, and *endif* instructions of the same code block.

6.25 endif – End If

Opcode	Instruction	Description
37 (0x25)	endif JIP	Restore execution to the 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
N	N	N	N

Format:

```
endif JIP
```

Syntax:

```
endif (exec_size) 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 execution to the channels that were active prior to the if/else/endif block.

The *endif* instruction is also used to hop out of 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 JIP. In GEN binary, JIP is at location src1 and must be of type W (signed word integer). JIP must be an immediate operand, it is a signed 16-bit number. This value is added to IP pre-increment.

Relative Instruction Offset

Bits	Description
31:16	Reserved: MBZ.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

Restrictions:

Predication is not allowed.

The execution size must be the same for the *if*, *else*, and *endif* instructions of the same code block.

6.26 f16to32 – Half Precision Float to Single Precision Float

Opcode	Instruction	Description
20 (0x14)	f16to32 dst src0	Convert a 16-bit half precision floating-point value to a 32-bit Float value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
W	F

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] ) {
     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.

Because this instruction does not have a 16-bit floating-point type, the source data type must be Word (W). The destination type must be F (Float).

Restrictions:

The FP Mode (Single Precision Floating Point Mode in cr0) must be IEEE mode.

No accumulator access, implicit or explicit.

Floating-Point Conversion in IEEE mode
Half Precision Float to Single Precision Float]

Half Precision Float	Single Precision Float
-inf	-inf
-finite	-finite
-denorm	-finite
-0	-0
+0	+0
+denorm	+finite
+finite	+finite
+inf	+inf
NaN	NaN

6.27 f32to16 – Single Precision Float to Half Precision Float

Opcode	Instruction	Description
19 (0x13)	f32to16 dst src0	Convert a 32-bit Float value to a 16-bit half precision floating-point value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	W

Format:

```
[(pred)] f32to16[.cmod] (exec_size) dst src0
```

Syntax:

```
[(pred)] f32to16[.cmod] (exec_size) reg reg
 [(pred)] f32to16[.cmod] (exec_size) reg imm32
```

Pseudocode:

```
 Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     dst.chan[n] = convert single precision float to half precision
float(src0.chan[n]);
   }
 }
```

Description:

The *f32to16* instruction converts the single precision float in src0 to half precision float and storing in the lower word of each channel in dst.

Because this instruction does not have a 16-bit floating-point type, the destination data type must be Word (W).

Restrictions:

The destination must be DWord-aligned and specify a horizontal stride (*HorzStride*) of 2. The 16-bit result is stored in the lower word of each destination channel and the upper word is not modified.

The FP Mode (Single Precision Floating Point Mode in cr0) must be IEEE mode.

No accumulator access, implicit or explicit.

Floating-Point Conversion in IEEE mode
Single Precision Float to Half Precision Float]

Single Precision Float	Half Precision Float
-inf	-inf
-finite	-finite/-denorm/-0
-denorm	-0
-0	-0
+0	+0
+denorm	+0
+finite	+finite/+denorm/+0
+inf	+inf
NaN	NaN

6.28 fbh – Find First Bit from MSB Side

Opcode	Instruction	Description
75 (0x4B)	fbh dst src0	Find the first significant bit searching from the high bits in src0 and store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
D, UD	UD

Format:

```
[(pred)] fbh (exec_size) dst src0
```

Syntax:

```
[(pred)] fbh (exec_size) reg reg
 [(pred)] fbh (exec_size) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for ( n = 0; n < exec_size; n++ ) {
  if ( WrEn.chan[n] ) {
    UD cnt = 0;
    if ( src0 is unsigned ) {
      UD udScalar = src0.chan[n];
      while ( (udScalar & (1 << 31)) == 0 && cnt != 32 ) {
```

```
        cnt ++;
        udScalar = udScalar << 1;
      }
      if ( src0.chan[n] == 0x00000000 ) {
        dst.chan[n] = 0xFFFFFFFF;
      }
      else {
        dst.chan[n] = cnt;
      }
    }
    else {  // src0 is signed.
      D dScalar = src0.chan[n];
      bit cval = dScalar[31];
      while ((dScalar & (1 << 31)) == cval && cnt != 32 ) {
        cnt ++;
        dScalar = dScalar << 1;
      }
      if ( (src0.chan[n] == 0xFFFFFFFF) || (src0.chan[n] == 0x00000000) ) {
        dst.chan[n] = 0xFFFFFFFF;
      }
      else {
        dst.chan[n] = cnt;
      }
    }
  }
}
```

Description:

If src0 is unsigned, the *fbh* instruction counts component-wise the leading zeros from src0 and stores the resulting counts in dst.

If src0 is signed and positive, the *fbh* instruction counts component-wise the leading zeros from src0 and stores the resulting counts in dst.

If src0 is signed and negative, the *fbh* instruction counts component-wise the leading ones from src0 and stores the resulting counts in dst.

Special Cases:

If src0 is zero, store 0xFFFFFFFF in dst.

If src0 is signed and is -1 (0xFFFFFFFF), store 0xFFFFFFFF in dst.

Restrictions:

No accumulator access, implicit or explicit.

6.29 fbl – Find First Bit from LSB Side

Opcode	Instruction	Description
76 (0x4C)	fbl dst src0	Find the first 1 bit searching from the low bits in src0 and store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
UD	UD

Format:

```
[(pred)] fbl (exec_size) dst src0
```

Syntax:

```
[(pred)] fbl (exec_size) reg reg
 [(pred)] fbl (exec_size) reg imm32
```

Pseudocode:

```
Evaluate(WrEn);
for ( n = 0; n < exec_size; n++ ) {
  if ( WrEn.chan[n] ) {
    UD cnt = 0;
    UD udScalar = src0.chan[n];
    while ( (udScalar & 1) == 0 && cnt != 32 ) {
      cnt ++;
      udScalar = udScalar >> 1;
    }
    if ( src0.chan[n] == 0x00000000 ) {
      dst.chan[n] = 0xFFFFFFFF;
    }
    else {
      dst.chan[n] = cnt;
    }
  }
}
```

Description:

The *fbl* instruction counts component-wise the number of LSB 0 bits before the first 1 bit in src0, storing that number in dst. If src0 contains no 1 bits, store 0xFFFFFFFF in dst.

Restrictions:

No accumulator access, implicit or explicit.

6.30 frc – Fraction

Opcode	Instruction	Description
67 (0x43)	frc dst src0	Truncate toward minus infinity fraction part of src0 and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Src Type	Dst Type
F	F

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] ) {
    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. The result is in the range [0.0, 1.0] irrespective of the rounding mode.

Floating-point fraction computation follows the rules in the following tables, based on the current floating-point mode.

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 [+0.0, 1.0), not including 1.0.								

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 [+0.0, 1.0), not including 1.0.								
**	Result is undefined if src0 is {–inf, +inf, or NaN}.								

Restrictions:

Erratum: When the Rounding Mode in cr0.0 is Round Down, the result from *frc* must be negated to bring it into the range [0.0, 1.0) not including 1.0.

Workaround: When the Rounding Mode in cr0.0 is *not* Round Down, the result from *frc* must be followed by compare and select instructions to avoid a result of 1.0. Those latter instructions must use the **:ud** type. For example:

```
cmp.ne.f0.0 null r4:ud 0x3f800000:ud
(f0.0)sel r5:f r4:ud 0x3f7fffff:ud
```

6.31 halt – Halt

Opcode	Instruction	Description
42 (0x2A)	halt JIP UIP	Temporarily suspend execution for all enabled execution channels.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Format:

```
[(pred)] halt (exec_size) JIP UIP
```

Syntax:

```
[(pred)] halt (exec_size) imm16 imm16
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < 32; n++ ) {
   if ( WrEn.channel[n] ) {
     PcIP[n] = IP + UIP;
   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. Upon execution, the enabled channels are sent to the instruction at (IP + UIP), if all channels are enabled at HALT, jump to the instruction at (IP + JIP).

If the *halt* instruction is not inside any conditional code block, the values of JIP and UIP should be the same. If the *halt* instruction is inside a conditional code block, the UIP should be the end of the program and the JIP should be the end of the inner most conditional code block.

The following table describes the two 16-bit instruction pointer offsets. 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).

Relative Instruction Offsets

Bits	Description
31:16	**UIP (Update Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the channel. Format = S15.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

If SPF is ON, the UIP must be used to update IP; JIP is not used in this case.

Restrictions:

dst and src0 must be NULL.

Workaround:: When SPF is ON, the JIP must point to a *nop* instruction.

6.32 if – If

Opcode	Instruction	Description
34 (0x22)	if JIP UIP	Indicate the start of an if/else/endif block of code.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Format:

```
[(pred)] if (exec_size) JIP UIP
```

Syntax:

```
[(pred)] if (exec_size) imm16 imm16
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < 32; n++ ) {
   if ( WrEn.channel[n] == 0 ) {
     PcIP[n] = IP + JIP;
   }
   else {
     PcIP[n] = IP + 1;
   }
 }

 if ( PcIP != (IP + 1) ) {  // for all channels
   Jump(IP + JIP);
 }
```

Description:

An *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 the predicate control.

Each *if* instruction must have a matching *endif* instruction and may have up to one matching *else* instruction before the matching *endif*.

If all channels are inactive (for the if/endif or if/else/endif block), a jump is performed to the instruction referenced by JIP. This jump must be to right after the matching *else* instruction when present, or otherwise to the matching *endif* instruction of the conditional block.

The following table describes the two 16-bit instruction pointer offsets. 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).

Relative Instruction Offsets

Bits	Description
31:16	**UIP (Update Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the channel. Format = S15.
15:0	**JIP (Jump Target Offset).** The jump distance in number of eight-byte units if a jump is taken for the instruction. Format = S15.

If SPF is ON, the UIP must be used to update IP; JIP is not used in this case.

Restrictions:

The execution size must be the same for the *if*, *else*, and *endif* instructions of the same code block.

6.33 illegal – Illegal

Opcode	Instruction	Description
0 (0x0)	illegal	Signal an illegal opcode exception, possibly transferring to the System Routine. If the exception is disabled, perform no operation.

Pred	Sat	Cond Mod	Src Mod
N	N	N	N

Format:

```
illegal
```

Syntax:

```
illegal
```

Pseudocode:

```
{
  Set the Illegal Opcode Exception Status bit in cr0.1.
  if ( Illegal Opcode Exception Enable is set in cr0.1 ) {
    Transfer control to the System Routine (return address to AIP, IP = SIP).
  }
}
```

Description:

The Illegal Opcode Exception Enable flag in cr0.1 is normally set so the normal processing of an illegal opcode is to transfer control to the System Routine.

Instruction dispatch treats any unused 8-bit opcode (including bit 7 of the instruction, reserved for future opcode expansion) as if it is the *illegal* opcode.

The *illegal* opcode is zero because that byte value is more likely than most to be read via a wayward instruction pointer.

The *illegal* instruction is an instruction only in the same way that a NULL pointer in software is a pointer. Both are special values indicating invalid instances.

Restrictions:

The *illegal* instruction takes no instruction options.

6.34 jmpi – Jump Indexed

Opcode	Instruction	Description
32 (0x20)	jmpi index	Redirect program execution based on an index relative to the post-incremented instruction pointer.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
D	

Format:

```
[(pred)] jmpi (1) index {NoMask}
```

Syntax:

```
[(pred)] jmpi (1) reg32 {NoMask}
 [(pred)] jmpi (1) imm32 {NoMask}
```

Pseudocode:

```
Evaluate(WrEn);
 if ( WrEn != 0 ) {
   Jump(IP + 1 + index );  // IP + 1 is a pseudocode idiom for the IP of the following
instruction.
 }
```

Description:

The *jmpi* instruction redirects program execution to an index offset relative to the *post-incremented* instruction pointer. The index is a signed integer value, with positive or zero integers for forward jumps, and negative integers for backward jumps.

Note: Unlike other flow control instructions, the offset used by *jmpi* is relative to the incremented instruction pointer rather than the IP value for the instruction itself.

In GEN binary, index is at location src1. The ip register must be put (for example, by the assembler) at the dst and src0 locations.

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 execution continues with the next instruction.

Index

Bits	Description
31:16	Reserved: MBZ
15:0	**index (Jump Index)** This field specifies the jump distance in number of 64-bit units if a jump is taken for the instruction. Format = S15.

Restrictions:

The execution size must be 1.

The {NoMask} instruction option must be specified.

The index data type must be D (Signed DWord Integer).

Programming Notes:

An index of 0 does nothing, continuing execution with the next instruction.

An index of -2 (if the *jmpi* instruction is in native format) or -1 (if the *jmpi* instruction is in compact format) is an infinite loop on the *jmpi* instruction.

6.35 line – Line

Opcode	Instruction	Description
89 (0x59)	line dst src0 src1	Compute a line equation (v = p * u + q) of src0 and src1 and store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
F	F

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++ ) {
   dwP = src0.RegNum.SubRegNum[bits4:2];       // A DWord-aligned scalar.
   dwQ = src0.RegNum.(SubRegNum[bit4] | 0x8);  // Fourth component.
   if ( WrEn.chan[n] ) {
     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 stores 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 supports an execution size (*ExecSize*) of 8 or 16.

The src0 region must be a replicated scalar (with *HorzStride* == *VertStride* == 0).

src0 must specify **.0** or **.4** as the subregister number, corresponding to a subregister byte offset of 0 or 16.

Source operands cannot be accumulators.

6.36 lrp – Linear Interpolation

Opcode	Instruction	Description
92 (0x5c)	lrp dst src0 src1 src2	Compute the linear interpolation equation (w = u*x + v*(1 - x)) of vectors (x, u, v) from src0, src1, and src2, storing the results in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Src Type	Dst Type
F	F

Format:

```
[(pred)] lrp[.cmod] (exec_size) dst src0 src1 src2
```

Syntax:

```
[(pred)] lrp[.cmod] (exec_size) reg reg reg
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     dst.chan[n] = src1.chan[n] * src0.chan[n] + src2.chan[n] * (1.0 – src0.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.

Restrictions:

The vertical stride (*VertStride*) is overloaded to 4 in HW for 3-source instructions.

The overflow conditional modifier (**.o**) is not allowed.

No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.

All three-source instructions have certain restrictions, described in Instruction Machine Formats.

6.37 lzd – Leading Zero Detection

Opcode	Instruction	Description
74 (0x4A)	lzd dst src0	Count the number of leading 0 bits (from MSB) in src0 and store that count in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
D, UD	UD

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] ) {
     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 stores the resulting counts in dst.

If src0 is zero, store 32 in dst.

Restrictions:

If the source is signed, the **abs** source modifier must be used to convert any negative source value to a positive value.

Accumulator cannot be destination, implicit or explicit.

6.38 mac – Multiply Accumulate

Opcode	Instruction	Description
72 (0x48)	mac dst src0 src1	Multiply src0 and src1, add to accumulator, and store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
*B, *W	*B, *W, *D
F	F
DF	DF

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] ) {
     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 implicit source and thus cannot be an explicit source operand.

6.39 mach – Multiply Accumulate High

Opcode	Instruction	Description
73 (0x49)	mach dst src0 src1	Multiply the upper word of src0 (masking off the lower word) times src1, adding the 64-bit product to the 64-bit accumulator value, storing the 64-bit result in the accumulator and storing the high DWord of the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y		Y	Y

Src Types	Dst Type
D	D
UD	UD

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++ ) {
  acc.chan[n][63:0] = (src0.chan[n][31:16] * src1.chan[n][31:0]) << 16
                    + acc.chan[n][63:0];
  if ( WrEn.chan[n] ) {
    dst.chan[n][31:0] = acc.chan[n][63:32];
  }
}
```

Description:

The *mach* instruction performs DWord integer multiply-accumulate operation and outputs the high DWord (bits 63:32).

For each enabled channel, this instruction multiplies the DWord in src1 with the high word of the DWord in src0, left shifts the result by 16 bits, adds it with the corresponding accumulator values, and keeps the whole 64-bit result 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 using the large number of bits available in the accumulator. For example, the following four instructions perform vector multiplication of two 32-bit signed integer sources from r2 and r3 and store the resulting vectors with the high 32 bits in r5 and the low 32 bits in r6.

```
mul  (8) acc0:d r2.0<8;8,1>:d r3.0<8;8,1>:d  //All channels must be enabled

mach (8) rTemp<1>:d r2.0<8;8,1>:d r3.0<8;8,1>:d  //All channels must be enabled

mov  (8) r5.0<1>:d rTemp<8;8,1>:d  // High 32 bits
 mov  (8) r6.0<1>:d acc0:d   // Low 32 bits
```

The *mul* and *mach* instructions must 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 the 64-bit DWord integer results, saturation modifier should not be used. In fact, saturation modifier should not be used for any of these four 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 implicit source and thus cannot be an explicit source operand.

AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

6.40 mad – Multiply Add

Opcode	Instruction	Description
91 (0x5B)	mad dst src0 src1 src2	Add src1 times src2 to src0 and store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	F
DF	DF

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] ) {
     dst.chan[n] = src1.chan[n] * src2.chan[n] + src0.chan[n];
   }
 }
```

Description:

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:

No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.

All three-source instructions have certain restrictions, described in Instruction Machine Formats.

6.41 math – Extended Math Function

Opcode	Instruction	Description
56 (0x38)	math dst src0 src1 <FC>	Component-wise extended math function on src0 (and src1) to dst based on the <FC>.

Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Type
Y	Y			F	F
Y	Y			D	D
Y	Y			UD	UD

Format:

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

Syntax:

```
[(pred)] math (exec_size) reg reg reg imm4
```

Pseudocode:

```
Evaluate(WrEn);
 for (n = 0; n < exec_size; n++) {
       if (WrEn.channel[n] == 1) {
              switch FC[3:0] {
                     case 1h:
                            dst.channel[n] = rcp(src0.channel[n]);
                     case 2h:
                            dst.channel[n] = log(src0.channel[n]);
                     case 3h:
                            dst.channel[n] = exp(src0.channel[n]);
                     case 4h:
                            dst.channel[n] = sqrt(src0.channel[n]);
                     case 5h:
                            dst.channel[n] = rsq(src0.channel[n]);
                     case 6h:
                            dst.channel[n] = sin(src0.channel[n]);
                     case 7h:
                            dst.channel[n] = cos(src0.channel[n]);
                     case 9h: // src0 / src1
                            dst.channel[n] = fdiv(src0.channel[n], src1.channel[n]);
                     case Ah:
                            dst.channel[n] = pow(src0.channel[n], src1/channel[n]);
                     case Bh: // src0 / src1
                            idiv(src0.channel[n], src1.channel[n]);
                     dst.channel[n] = quotient;
                            dst+1.channel[n] = remainder;
                     case Ch:
```

```
                        idiv(src0.channel[n], src1.channel[n]);
                        dst.channel[n] = quotient;
                    case Dh:
                        idiv(src0.channel[n], src1.channel[n]);
                        dst.channel[n] = remainder;
            }
        }
    }
```

Description:

The *math* instruction performs extended math function on the components in src0, or src0 and src1, and write the output to the channels of dst. The type of extended math function are based on the FC[3:0] encoding in the table below.

Function Control[3:0]	Description
[3:0]	Function Description: 0h: Reserved 1h: INV (reciprocal) 2h: LOG 3h: EXP 4h: SQRT 5h: RSQ 6h: SIN 7h: COS 8h: Reserved 9h: FDIV Ah: POW Bh: INT DIV – return quotient and remainder Ch: INT DIV – return quotient only Dh: INT DIV – return remainder Eh: Reserved Fh: Reserved

Restrictions:

No accumulator access, implicit or explicit.

The math instruction must use GRF registers as source(s) and destination.

The *math* instruction does not support indirect addressing modes.

The only supported rounding mode for *math* instruction is Round to Nearest Even.

For INTDIV with both quotient and remainder, the second destination register is implied from the first destination register with “+1” to get the register number.

Mixed DW and UD sources are not allowed for the INT DIV function.

INT DIV function does not support SIMD16.

The FDIV function is not supported in ALT_MODE.

6.41.1 INV - Inverse

Precision:1 ULP

Src->	+inf	+0 / +Denorm	- 0 / -Denorm	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN
Dest – ALT mode		+fmax	-fmax		NaN

6.41.2 LOG – Logarithm

Precision:

DirectX 10 and below

If src0 is [0.5..2], **absolute error** must be no more than 2-21. If src0 is (0..0.5) or (2..+INF], **relative error** must be no more than 2-21

Note: In ALT mode log is computed as Log_2 (abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**-inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**-fmax**	**-fmax**		**+F**	**NaN**

6.41.3 EXP - Exponent

Precision:

DirectX 10 Relative error <= 3 ULP

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**1**	**1**	**0**	**+F**	**NaN**
Dest – ALT mode		**1**	**1**		**+F**	**NaN**

6.41.4 SQRT

Precision:

DirectX 10 (and below) Relative error <= 1 ULP

Notes:In ALT mode SQRT is computed as SQRT(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+inf**	**0**	**-0**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**0**	**0**		**+F**	**NaN**

6.41.5 RSQ

Precision:

DirectX 10 and below Relative error <= 3 ULP

Notes:In ALT mode RSQ is computed as RSQ(abs (src0))

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	**+0**	**+inf**	**-inf**	**NaN**	**NaN**	**NaN**
Dest – ALT mode		**+fmax**	**+fmax**		**+F**	**NaN**

6.41.6 POW

Precision:

POWR, POWN is not supported in hardware.

IEEE Mode:

Src0->								
Src1	abs(F > 1)	abs(F < 1)	abs(+F = = 1)	+inf	+0 / +Denorm	-Denorm / -0	-inf	NaN
+inf	+inf	0	NaN	+inf	0	0	+inf	NaN
+0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F			+inf	0	0	NaN	NaN

ALT Mode:

Src0->							
Src1	+F	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
+inf							
+0 / Denorm	1		1	1		1	NaN
-0 / Denorm	1		1	1		1	NaN
-inf							
-F	+F		+fmax	+fmax		+F	NaN
NaN			NaN	NaN		NaN	NaN
+F	+F		0	0		+F	NaN

6.41.7 SIN

Precision:

DirectX 10 and below Absolute error <= 0.0008 for the range of +/- 100* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+0	-0		-1 to 1	NaN

6.41.8 COS

Precision:

DirectX 10 and below: Absolute error <= 0.0008 for the range of +/- 100* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of COS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

Src->	+inf	+0 / +Denorm	-0 / -Denorm	-inf	-F	NaN
Dest – IEEE mode	NaN	+0	-0	NaN	-1 to 1	NaN
Dest – ALT mode		+1	+1		-1 to 1	NaN

6.41.9 INT DIV

Precision:32-bit integer

For signed inputs, INT DIV behavior is illustrated by the table below:

Inputs:	**Numerator**	**+**	**+**	**-**	**-**
	Denominator	**+**	**-**	**+**	**-**
Outputs:	**Quotient**	**+**	**-**	**-**	**+**
	Remainder	**+**	**+**	**-**	**-**

6.41.10 mov – Move

Opcode	Instruction	Description
1 (0x01)	mov dst src0	Copy the value from src0 to dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
*B, *W, *D	*B, *W, *D
*B, *W, *D	F
F	*B, *W, *D
F	F
*W, *D	DF
F	DF
DF	*W, *D
DF	F
DF	DF

Format:

```
[(pred)] mov[.cmod] (exec_size) dst src0
```

Syntax:

```
[(pred)] mov[.cmod] (exec_size) reg reg
 [(pred)] mov[.cmod] (exec_size) reg imm32
```

Pseudocode:

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

Description:

The *mov* instruction moves the components in src0 into the channels of dst. If src0 and dst are of different types, format conversion is performed. If src0 is a scalar immediate, the immediate value is loaded into enabled channels of dst.

A *mov* with the same source and destination type, no source modifier, and no saturation is a *raw move*. A packed byte destination region (B or UB type with *HorzStride* == 1 and *ExecSize* > 1) can only be written using raw move.

Restrictions:

Raw move is not supported for Float values in ALT mode if any values are infinities or NaNs.

An accumulator can be a source or destination operand but not both.

If the source type and destination type differ, conditional modifiers are not allowed.

Programming Notes:

There is no direct conversion from B/UB to DF or DF to B/UB. Use two instructions and a word or DWord intermediate type.

6.42 6movi – Move Indexed

Opcode	Instruction	Description
3 (0x3)	movi dst src0	Fast indexed move from src0 to dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	N	Y

Generation	Src Type	Dst Type
	B	B
	UB	UB
	W	W
	UW	UW
	D	D
	UD	UD
	F	F

Note: As shown in the table, the source and destination type must be the same.

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] ) {
     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.

The 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 cannot be accumulators. The source operand must be a general register.

The source and destination must have the same type.

The execution size must be <= 8 (1, 2, 4, or 8).

The address register for the source must be aligned to the base (a0.0).

The destination register (directly or indirectly addressed) must be 16-byte aligned.

The destination stride in bytes must equal the source element size in bytes.

The Align16 access mode is not allowed.

All the index registers (address subregisters) used must point to the same GRF register.

The instruction 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) is always used to select the first element of the destination start from offset zero. Each index register is used to select 1 element if type is byte, 2 elements if type is Word, or 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] & 00b), byte1 is selected by (a0.0[4:2] & 01b), byte2 is selected by (a0.0[4:2] & 10b), byte3 is selected by (a0.0[4:2] & 11b), byte4 is selected by (a0.1[4:2] & 00b), byte5 is selected by (a0.1[4:2] & 01b), ..., byte31 is selected by (a0.7[4:2] & 11b).

For all 3 conditions above, a0.n[4:0] = a0.n[4:0] + addr_imm[4:0].

6.43 mul – Multiply

Opcode	Instruction	Description
65 (0x41)	mul dst src0 src1	Multiply src0 and src1 storing the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
W	W
W	D
W, D	D

Src Types	Dst Type
F	F
DF	DF

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] ) {
     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 low 16 bits of each element of src1 are used. The accumulator maintains full 48-bit precision. The macro described in the *mach* instruction should be used to obtain the full precision 64-bit multiplication result.

Multiplication of two floating-point numbers follows the rules in *mul – Multiply* or *mul – Multiply*) based on the applicable floating-point mode.

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}.										

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 A or B is {–inf, +inf, NaN}.										

Restrictions:

Source operands cannot be accumulators.

When operating on integers with at least one of the source being a DWord type (signed or unsigned), the destination cannot be floating-point (implementation note: the data converter only looks at the low 34 bits of the result).

When operating on integers with at least one source having a DWord type (signed or unsigned), the Overflow and Sign flags are undefined. Therefore, conditional modifiers and saturation (**.sat**) cannot be used in this case.

]: When multiplying a DW and a W, the W has to be on src1, and the DW has to be on src0.

6.44 nop – No Operation

Opcode	Instruction	Description
126 (0x7E)	nop	Issue a dummy instruction and perform no operation.

Pred	Sat	Cond Mod	Src Mod
N	N	N	N

Format:

```
nop
```

Syntax:

```
nop
```

Pseudocode:

```
{
  ;  // The null statement, which does nothing.
}
```

Description:

Do nothing. The *nop* instruction takes an instruction dispatch but performs no operation. It can be used for assembly patching in memory, or to insert a delay in the program sequence.

Restrictions:

The *nop* instruction takes no instruction options other than **Breakpoint**.

6.45 not – Logic Not

Opcode	Instruction	Description
4 (0x04)	not dst src0	Bitwise NOT of src0 storing the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Generation	Src Type	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
     dst.chan[n] = ~ src0.chan[n];
   }
 }
```

Description:

The *not* instruction performs logical NOT operation (or one's complement) of src0 and storing the results in dst.

A register source operand can use a source modifier:

- Any source modifier is numeric, optionally changing a source value s to -s, abs(s), or -abs(s) before the NOT operation.

This operation does not produce sign or overflow conditions. Only the **.e**/**.z** or **.ne**/**.nz** conditional modifiers should be used.

Restrictions:

Source modifier is not allowed if source is an accumulator.

6.46 or – Logic Or

Opcode	Instruction	Description
6 (0x06)	or dst src0 src1	Bitwise OR of src0 and src1 storing the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Generation	Src Types	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
     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.

Register source operands can use source modifiers:

- Any source modifier is numeric, optionally changing a source value s to -s, abs(s), or -abs(s) before the OR operation.

This operation does not produce sign or overflow conditions. Only the **.e**/**.z** or **.ne**/**.nz** conditional modifiers should be used.

Restrictions:

Source modifier is not allowed if source is an accumulator.

6.47 pln – Plane

Opcode	Instruction	Description
90 (0x5A)	pln dst src0 src1	Compute a plane equation (w = p*u + q*v + r) of scalar (p, q, r) from src0 and vector src1 (and implied vector src2). Store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	

Src Types	Dst Type
F	F

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++ ) {
   float dwP = src0.RegNum.SubRegNum[bits4:2];          // A DWord-aligned scalar.
   float dwQ = src0.RegNum.(SubRegNum[bit4:2] | 0x1);   // Second component.
   float dwR = src0.RegNum.(SubRegNum[bit4:2] | 0x3);   // Fourth component.
   if ( ExecSize == 8 )
     src2 = src1.(RegNum + 1)  // Next GRF register.
   else  // ExecSize == 16
     src2 = src1.(RegNum + 2)  // The two GRF registers after RegNum and RegNum + 1.
   if ( WrEn.chan[n] ) {
     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 stores the results in dst. src1 is the input vector *u*.

If *ExecSize* is 8 then the second input vector v is implied from src1 as the next adjacent GRF register.

If *ExecSize* is 16 then the second input vector v is implied from src1 as the two GRF registers after the two registers used by input vector *u*.

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 the src0 region. Specifically, *q* is the second component and *r* is the fourth component of the 4-tuple (128-bit aligned) that *p* belongs to.

Restrictions:

This is a specialized instruction that only supports an execution size (*ExecSize*) of 8 or 16.

The src0 region must be a replicated scalar (with *HorzStride* == *VertStride* == 0).

src0 must specify **.0** or **.4** as the subregister number, corresponding to a subregister byte offset of 0 or 16.

Source operands cannot be accumulators.

6.48 ret – Return

Opcode	Instruction	Description
45 (0x2d)	ret null src0	Return from a subroutine.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
D, UD	

Format:

```
[(pred)] ret (exec_size) null src0
```

Syntax:

```
[(pred)] ret (exec_size) null reg
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.chan[n] ) {
     PcIP[n] = src0.chan[0];
     CallMask[n] = 0;
   }
   else {
     PcIP[n] = IP + 1;
   }
 }
 for ( n = exec_size; n < 32; n++ ) {
   PcIP[n] = IP + 1;
 }

 if ( CallMask[n:0] == 0) ) {  // all channels are zero
   Jump(src0.chan[0]);
   CallMask = src0.chan[1];
 }
```

Description:

Return execution to the code sequence that called a subroutine.

The *ret* instruction can be predicated or non-predicated. If non-predicated, all channels jump to the return IP in the first channel of src0 and restore CallMask from the second channel of src0. If predicated, the enabled channels jump to the return IP from the first channel of src0 and the corresponding bits in the CallMask are cleared to zero; if all CallMask bits are zero after the *ret* instruction, then execution jumps to the return IP from the first channel of src0.

When SPF is on, the predication control must be scalar.

Restrictions:

This instruction cannot take accumulator as source.

The src0 regioning control must be <2;2,1>,

6.49 rndd – Round Down

Opcode	Instruction	Description
69 (0x45)	rndd dst src0	Round floating-point src0 down to an integral floating-point value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	F

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] ) {
     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.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Down in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	-finite	^	–0	+0	+0	**	+inf	NaN
Notes:									
^	-0 as denorms are flushed to zero.								
**	Result may be {+finite, +0}.								

Floating-Point Round Down in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	-finite	–0	–0	+0	+0	**	+fmax	

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	-finite	–0	–0	+0	+0	**	+fmax	
Notes:									
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

Restrictions:

No accumulator access, implicit or explicit.

6.50 rnde – Round to Nearest or Even

Opcode	Instruction	Description
70 (0x46)	rnde dst src0	Round floating-point src0 to the nearest or even integral floating-point value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	F

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] ) {
     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 ( floor(src0.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.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round to Nearest or 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}.

Floating-Point Round to Nearest or 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}.
***	Result is undefined if src0 is {–inf, +inf, NaN}.

Restrictions:

No accumulator access, implicit or explicit.

6.51 rndu – Round Up

Opcode	Instruction	Description
68 (0x44)	rndu dst src0	Round floating-point src0 up to an integral floating-point value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	F

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] ) {
     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.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Up in IEEE mode

src0	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
dst	–inf	*	–0	–0	+0	^	+finite	+inf	NaN

Notes:

*	Result may be {–finite, –0}.
^	+0 as denorms are flushed to zero.

Floating-Point Round Up in ALT mode

src0	–fmax	–finite	–denorm	–0	+0	+denorm	+finite	+fmax	***
dst	–fmax	*	–0	–0	+0	+0	+finite	+fmax	
Notes:									
*	Result may be {–finite, –0}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

Restrictions:

No accumulator access, implicit or explicit.

6.52 rndz – Round to Zero

Opcode	Instruction	Description
71 (0x47)	rndz dst src0	Round floating-point src0 toward zero to an integral floating-point value and store in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Type	Dst Type
F	F

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] ) {
     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.

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}.								

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}.								
***	Result is undefined if src0 is {–inf, +inf, NaN}.								

Restrictions:

No accumulator access, implicit or explicit.

6.53 sad2 – Sum of Absolute Difference 2

Opcode	Instruction	Description
80 (0x50)	sad2 dst src0 src1	Perform a two-wide sum-of-absolute-difference operation on a 2-tuple basis of src0 and src1, and store the scalar result to the first channel per 2-tuple in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
B, UB	W, UW

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] ) {
     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.

The results are also stored in the accumulator register. The destination operand and the accumulator maintain 16 bits per channel precision.

The destination register must be aligned to even word (DWord). The even words in the destination region will contain the correct data. The odd words are also written but with undefined values.

Restrictions:

Source operands cannot be accumulators.

The execution size cannot be 1 as the computation requires at least two data channels.

6.54 sada2 – Sum of Absolute Difference Accumulate 2

Opcode	Instruction	Description
81 (0x51)	sada2 dst src0 src1	Perform a two-wide sum-of-absolute-difference operation on a 2-tuple basis of src0 and src1, added to that from the accumulator, and store the scalar result to the first channel per 2-tuple in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
B, UB	W, UW

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.chan[n] - src1.chan[n])
         + abs(src0.chan[n+1] - src1.chan[n+1]);
   if ( WrEn.chan[n] ) {
     dst.chan[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.

The destination operand and the accumulator maintain 16 bits per channel precision. Higher precision (guide bits) stored in the accumulator allows up to 64 rounds of *sada2* instructions to be issued back to back without overflowing the accumulator.

The destination register must be aligned to even word (DWord). The even words in the destination region will contain the correct data. The odd words are also written but with undefined values.

Restrictions:

Source operands cannot be accumulators.

The execution size cannot be 1 as the computation requires at least two data channels.

6.55 sel – Select

Opcode	Instruction	Description
2 (0x02)	(pred) sel dst src0 src1	Copy either src0 or src1 to dst based on predication or conditional modifier.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Src Types	Dst Type
*B, *W, *D	*B, *W, *D
F	F
DF	DF

Format:

```
(pred) sel[.cmod] (exec_size) dst src0 src1
```

Syntax:

```
(pred) sel[.cmod] (exec_size) reg reg reg
 (pred) sel[.cmod] (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.chan[n] ) {
      if ( PMask.channel[n] ) {
        dst.chan[n] = src0.chan[n];
      }
      else {
        dst.chan[n] = src1.chan[n];
      }
    }
  }
}
else {  // with CMod
  Evaluate(CMod);
  for ( n = 0; n < exec_size; n++ ) {
    if ( WrEn.chan[n] ) {
      if ( CMod.chan[n] ) {
        dst.chan[n] = src0.chan[n];
      }
      else {
        dst.chan[n] = src1.chan[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. The predicate clause is mandatory if cmod is omitted/0000b. If both predication and the conditional modifier are omitted, the results are undefined.

If the conditional modifier is specified (not 0000b, a compare is performed and the resulting condition flag is used for the *sel* instruction. Conditional modifiers **.ge** and **.l** follow the *cmpn* rules, and all other conditional modifiers follow the *cmp* rules. Predication is not allowed in this mode.

A *sel* instruction with cmod **.l** is used to emulate a MIN instruction.

A *sel* instruction with cmod **.ge** is used to emulate a MAX instruction.

For a *sel* instruction with a **.l** or **.ge** conditional modifier, if one source is NaN and the other not NaN, the non-NaN source is the result. If both sources are NaNs, the result is NaN. For all other conditional modifiers, if either source is NaN then src1 is selected.

A *sel* instruction without a conditional modifier always copies a denorm source value to a denorm destination value (in the manner of a raw move).

A *sel* instruction with a conditional modifier flushes any selected denorm source value to a zero destination value.

Format conversion is not allowed.

The *sel* instruction uses any conditional modifier internally and does not update the flag register if a conditional modifier is used.

Restrictions:

]: The maximum execution size is 16.

6.56 send Message

Opcode	Instruction	Description
49 (0x31)	send <dest> <src> <ex_desc> <desc>	Send a message stored in GRF starting at <src> 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 Type
Y					[FLT] [INT]

Format:

```
[(pred)] send (exec_size) <dest> <src> <ex_desc> <desc>
```

Syntax:

```
[(pred)] send (exec_size) reg greg imm6 reg32a
 [(pred)] send (exec_size) reg greg imm6 imm32
```

Pseudocode:

```
Evaluate(WrEn);
 <MsgChEnable> = WrEn;
```

```
<SourceReg> = <src>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>, <desc>, <ex_dest>);
```

Description:

The *send* instruction performs data communication between a thread and external function units, including shared functions (Sampler, Data Port Read, Data Port Write, URB, and Message Gateway) and some fixed functions (e.g. Thread Spawner, who also have an unique Shared Function ID). The *send* instruction adds an entry to the EU's message request queue. The request message is stored in a block of contiguous GRF registers. The response message, if present, will be returned to a block of contiguous GRF registers. The return GRF writes may be in any order depending on the external function units. <src> is the lead GRF register for request. <dest> is the lead GRF register for response. The message descriptor field <desc> contains the Message Length (the number of consecutive GRF registers) and the Response Length (the number of consecutive GRF registers). It also contains the header present bit, and the function control signals. The extend mesage descriptor field <ex_desc> contains the target function ID. WrEn is forwarded to the target function in the message sideband.

The *send* instruction is the only way to terminate a thread. When the EOT (End of Thread) bit of <desc> is set, it indicates the end of thread to the EU, the Thread Dispatcher and, in most cases, the parent fixed function.

Message descriptor field <desc> can be a 32-bit immediate, *imm32*, or a 32-bit scalar register, <reg32a>. GEN restricts that the 32-bit scalar register <reg32a> must be the leading dword of the address register. It should be in the form of a0.0<0;1,0>:ud. When <desc> is a register operand, only the lower 29 bits of <reg32a> are used.

<ex_desc> is a 6-bit immediate, imm6. The lower 4bits of the <ex_desc> specifies the SFID for the message. The MSb of the message descriptor, the EOT field, always comes from bit 127 of the instruction word, which is the MSb of imm6. A thread must terminate with a *send* instruction with EOT turned on.

<src> is a 256-bit aligned GRF register. It serves as the leading GRF register of the request.

<dest> serves for two purposes: to provide the leading GRF register location for the response message if present, and to provide parameters to form the channel enable sideband signals.

<dest> signals whether there is a response to the message request. It can be either a null register, a direct-addressed GRF register or a register-indirect GRF register. Otherwise, hardware behavior is undefined.

If <dest> is null, there is no response to the request. Meanwhile, the Response Length field in <desc> must be 0. Certain types of message requests, such as memory write (store) through the Data Port, do not want response data from the function unit. If so, the posted destination operand can be null.

If <dest> is a GRF register, the register number is forwarded to the shared function. In this case, the target function unit must send one or more response message phases back to the requesting thread. The number of response message phases must match the Response Length field in <desc>, which of course cannot be zero. For some cases, it could be an empty return message. An empty return message is defined as a single phase message with all channel enables turned off.

The subregister number, horizontal stride, destination mask and type fields of <dest> are always valid and are used in part to generate the WrEn. This is true even if <dest> is a null register (this is an exception for null as for most cases these fields are ignored by hardware).

The 16-bit channel enables of the message sideband are formed based the WrEn. Interpretation of the channel enable sideband signals is subject to the target external function. In general for a 'send' instruction with return messages, they are used as the destination dword write mask for the GRF registers

starting at <dest>. For a message that has multiple return phases, the same set of channel enable signals applies to all the return phases.

Thread managed memory coherency: A special usage of using non-null <dest> is to support write-commit signaling for memory write service by the Data Port Write unit. If <post_dest> is not null for a memory write request, the Data Port along with the Data Cache or Render Cache will wait until all the posted writes for the request have reached the coherent domain before sending back to the requesting thread an empty message to <dest> register. A memory write reaching the coherent domain, also referred to as reaching the global observable state, means that subsequent read to the same memory location, no matter which thread issues the read, must return the data of the write.

The destination dependency control, {NoDDClr}, can be used in this instruction. This allows software to control the destination dependencies for multiple 'read'-type messages similar to that for multiple instructions using EU execution pipeline. As *send* does not check register dependencies for the post destination, {NoDDChk} should not be used for this instruction.

Message Descriptor Definition

Bit	Description
31	Reserved : MBZ
30	Reserved : MBZ
29	Reserved : MBZ
28:25	**Message Length.** This field specifies the number of 256-bit GRF registers starting from <src> to be sent out on the request message payload. Valid value ranges from 1 to 15. A value of 0 is considered erroneous. Format = U4 Range = [1,15]
24:20	**Response Length.** This field indicates the number of 256-bit registers expected in the message response. The valid value ranges from 0 to 16. A value 0 indicates that the request message does not expect any response. The largest response supported is 16 GRF registers. Format = U5 Range = [0,16]
19	**Header Present.** If set, indicates that the message includes a header. Depending on the target shared function, this field may be restricted to either enabled or disabled. Refer to the specific shared function section for details. Format = Enable
18:0	**Function Control** This field is intended to control the target function unit. Refer to the section on the specific target function unit for details on the contents of this field.

Extended Message Descriptor Definition

Bit	Description
5	**End Of Thread** This field, if set, indicates that this is the final message of the thread and the thread's resources can be reclaimed.
4	reserved

Bit	Description
3:0	**Target Function ID** This field indicates the function unit for which the message is intended. ***Refer to "GPU Overview" document for the mapping of Shared Function IDs***

Sideband Signals Associated with Each Message Sent to the Shared Function provides a summary of the signals associated with each message that is sent to the shared function. It contains fields from the send instruction as well as fields from control register *cr0* and state register *sr0*.

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 *send* with EOT should use register space R112-R127 for <src>. This is to enable loading of a new thread into the same slot while the message with EOT for current thread is pending dispatch.

6.57 sendc – Conditional Send Message

Opcode	Instruction	Description
50 (0x32)	sendc dst src desc exdesc	Wait for dependencies in the TDR Register to clear, then send a message stored in registers starting at src to a shared function identified by exdesc along with control from desc with a general register writeback location at dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Src Type	Dst Type
	[FLT], [INT]

Format:

```
[(pred)] sendc (exec_size) dst src0 exdesc desc
```

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] ) {
  wait;
}
Evaluate(WrEn);
MsgChEnable = WrEn;
SourceReg = src0.RegNum;
MessageEnqueue(MsgChEnable, ResponseReg, SourceReg, desc, exdesc);
```

Description:

The *sendc* instruction has the same behavior as the *send* instruction except the following.

sendc first checks the dependent threads inside the Thread Dependency Register. There are up to 8 dependent threads in the TDR register. The *sendc* instruction executes only when all the dependent threads in the TDR register are retired.

Restrictions:

The *sendc* instruction has the same restrictions as the *send* instruction.

6.58 shl – Shift Left

Opcode	Instruction	Description
9 (0x09)	shl dst src0 src1	Shift the bits in src0 left by the number of bits indicated in src1. Store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Generation	Src Types	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
     shiftCnt = src1.chan[n] & 0x1F; // Always use low 5 bits for shift count.
     dst.chan[n] = src0.chan[n] << shiftCnt;
   }
 }
```

Description:

Perform component-wise logical left shift of the bits in src0 by the shift count indicated in src1, storing the results in dst, inserting zero bits in the number of LSBs indicated by the shift count.

The shift count is taken from the low five bits of src1, regardless of the src1 type and treated as an unsigned integer in the range 0 to 31.

Hardware detects overflow properly and uses it to perform any saturation operation on the result, as long as the shifted result is within 33 bits. Otherwise, the result is undefined.

Note: For word and DWord operands, the accumulators have 33 bits.

Restrictions:

Accumulator cannot be destination, implicit or explicit.

Results of saturation in packed-DWord mode are unpredicable.

6.59 shr – Shift Right

Opcode	Instruction	Description
8 (0x08)	shr dst src0 src1	Shift the bits in src0 right by the number of bits indicated in src1. Store the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	Y	Y	Y

Generation	Src Types	Dst Type
	UB, UW, UD	UB, UW, UD

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] ) {
     shiftCnt = src1.chan[n] & 0x1F; // Always use low 5 bits for shift count.
     dst.chan[n] = src0.chan[n] >> shiftCnt;
   }
 }
```

Description:

Perform component-wise logical right shift with zero insertion of the bits in src0 by the shift count indicated in src1, storing the results in dst. Insert zero bits in the number of MSBs indicated by the shift count.

The shift count is taken from the low five bits of src1, regardless of the src1 type and treated as an unsigned integer in the range 0 to 31.

src0 and dst can have different types and can be signed or unsigned.

Note: For word and DWord operands, the accumulators have 33 bits.

Note: For unsigned src0 types, *shr* and *asr* produce the same result.

6.60 subb – Integer Subtraction with Borrow

Opcode	Instruction	Description
79 (0x4F)	subb dst src0 src1	Subtract unsigned integer src1 from src0. Store the result in dst and store the borrow (0 or 1) as a 32-bit value in acc.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	N

Src Types	Dst Type
UD	UD

Format:

```
[(pred)] subb[.cmod] (exec_size) dst src0 src1
```

Syntax:

```
[(pred)] subb[.cmod] (exec_size) reg reg reg
 [(pred)] subb[.cmod] (exec_size) reg reg imm32
```

Pseudocode:

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

Description:

The *subb* instruction performs component-wise subtraction of src0 and src1 and stores the results in dst, it also stores the borrow into acc.

If the operation produces a borrow (src0 < src1), write 0x00000001 to acc, else write 0x00000000 to acc.

.

Restrictions:

AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

6.61 wait – Wait Notification

Opcode	Instruction	Description
48 (0x30)	wait nreg	Wait for notification on the notification register nreg.

Pred	Sat	Cond Mod	Src Mod
N	N	N	N

Src Type	Dst Type
UD	UD

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, thread execution is suspended 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 nonzero, the thread wakes up to continue execution and at the same time the notification register is decremented by one. If 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 nonzero again.

When multiple notifications are received, software must use *wait* instructions to decrement notification count registers for each notification.

Notification register n0:ud is for thread to thread communication (via the Message Gateway shared function) and n1:ud for host to thread communication (through MMIO registers). See the Message Gateway chapter for thread-thread communication .

Restrictions:

src0 and dst must be n0, n1, or n2.

Execution size must be 1 as the notification registers are scalar.

Predication is not allowed.

Two back-to-back *wait* instructions are not allowed. At minimum, a *nop* instruction must be inserted between two *wait* instructions.

6.62 while – While

Opcode	Instruction	Description
39 (0x27)	while JIP	Mark the end of a do-while block of code.

Pred	Sat	Cond Mod	Src Mod
Y	N	N	N

Format:

```
[(pred)] while (exec_size) JIP
```

Syntax:

```
]: [(pred)] while (exec_size) imm16
```

Pseudocode:

```
Evaluate(WrEn);
 for ( n = 0; n < 32; n++ ) {
   if (WrEn.chan[n] ) {
     PcIP[n] = IP + JIP;
   }
   else {
     PcIP[n] = IP + 1;
   }
 }

 if (cmod == 0) {
   if ( | PMask == 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 on the current channel enables and the predication flags specified in the instruction. If any channel has not terminated, a branch is taken to a destination address specified in the instruction, and the loop continues for those channels. Otherwise, execution continues 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 src1 and must be of type W (signed word integer).

JIP

Bits	Description
31:16	Reserved: MBZ.
15:0	**JIP (Jump Target Offset).** Specifies the jump distance in 64-bit units if a jump is taken for the instruction. Format = S15.

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

Restrictions:

The execution size must be the same for the *while* instruction and any *break* and *cont* instructions of the same code block.

6.63 xor – Logic Xor

Opcode	Instruction	Description
7 (0x07)	xor dst src0 src1	Bitwise XOR of src0 and src1 storing the result in dst.

Pred	Sat	Cond Mod	Src Mod
Y	N	Y	Y

Generation	Src Types	Dst Type
	*B, *W, *D	*B, *W, *D

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] ) {
    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.

Register source operands can use source modifiers:

- Any source modifier is numeric, optionally changing a source value s to -s, abs(s), or -abs(s) before the XOR operation.

This operation does not produce sign or overflow conditions. Only the **.e/.z** or **.ne/.nz** conditional modifiers should be used.

Restrictions:

Source modifier is not allowed if source is an accumulator.

7. EU Programming Guide

7.1 Assembler Pragmas

7.2 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*

7.2.1 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.

7.2.2 Example Pragma Usages

Example: 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.

Declaration

```
// 32x4 Byte Array.declare BlockBase=r0 ElementSize=1 Region=<32;8,1>Type=b
```

Fully-Expressed Instr

```
mov(32)?:br0.16<32;8,1>:b// r0 xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r1
xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r2 xxxxxxxxxxxxxxxxoooooooooxxxxxxxx// r3
xxxxxxxxxxxxxxxxoooooooooxxxxxxxx
```

Short-handed Instr

Mov?:bBlock(0,16)// (0,16): RegNum=0, SubRegNum=16

Example: 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.

Declaration

```
// 8x4 float Array starting at r5.declare Trans Base=r5 ElementSize=4 Region=<0;8,1>
Type=f
```

Fully-Expressed Instr

```
mov(8)?:fr6.0<0;8,1>:f// 2nd 16x1 Row of Trans. Matrix // r5 FFFFFFFF// r6 OOOOOOOO//
r7 FFFFFFFF// r8 FFFFFFFF
```

Short-handed Instr

mov?:fTrans(1) // RegNum = 5+1 = 6

Example: 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.

Declaration

```
//8x4 float data array and 16x1 word address array.declare TransBase=r5 ElementSize=4
Region=<0;8,1> Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,224]<0;8,1>:f

Short-handed Instr

mov?:fTrans[a0.0,2] // [a0.0 + 5*32 + 2*32]

Example: 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 TransBase=r5 ElementSize=4
Region=<0;8,1> Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,224]<1,0>:f

Short-handed Instr

mov?:fTrans[a0.0,2]<1,0> // [a0.0+224] [a0.1+224] … [a0.7+224]

Example: 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 TransBase=r5 ElementSize=4
Region=<0;8,1> Type=f
```

Fully-Expressed Instr

mov(8)?:fr[a0.0,244]<4,1>:f

Short-handed Instr

```
mov?:fTrans[a0.0,2]<4,1>// [a0.0+224] [a0.1+224]
```

7.2.3 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 ------------- // ...
```

7.3 Usage Examples

7.3.1 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. 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

7.3.1.1 Supporting DirectX 10 Pixel Shader Indexing

When a DirectX 10 Pixel Shader program is converted to run on GEN in channel-serial mode at 16 pixels in parallel, the per-pixel index must be translated into 16 indices with per channel offset. The creation of the per-channel offset can be achieved using the vector immediate.

Consider a generic DirectX 10 Pixel Shader instruction in the form of

opr4r[ind]r2

and assume that r0-r1 contain the 16 indices packed every other words, and r2-r3 contains source 1 and r4-r5 contain the destination. This instruction can be converted into the following GEN instructions. The corresponding operations are illustrated in *Supporting DirectX 10 Pixel Shader Indexing*.

mov (16) r11.0<1>:w 0x01234567:v// assigning a ramp vector, repeated once

mul (16)acc0:wr11.0<0;16,1>:w4:w// expand ramp range to 4 bytes per step

mac (16)r10.0<1>:wr0.0<16;8,2>:w32:w// r10 = index*32 + 0|4|…|28|0|4…|28

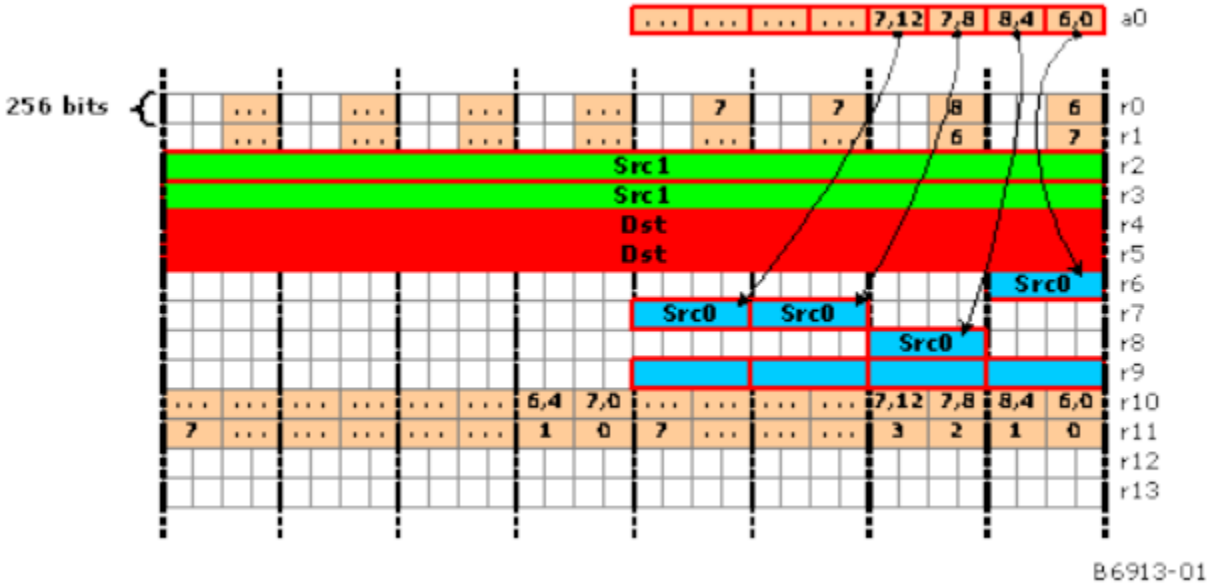
mov (8)a0.0<1>:wr10.0<0;8,1>:w

op (8)r4.0<1>:fr[a0.0]<1,0>:fr2.0<0;8,1>:w// Operate on the first half

mov (8)a0.0<1>:wr10.8<0;8,1>:w// Index values are off by a reg (32b)

op (8)r5.0<1>:fr[a0.0+32]<1,0>:fr3.0<0;8,1>:w// Operate on the second half.

Pixel Shader example using vector immediate.



Without vector immediate support, such translation has to either use a long sequence of scalar instructions which is very inefficient or use a constant load which requires additional constant to be managed in memory.

7.3.1.2 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

SWZr1r0.0-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>:fr0.xzxz0x1F111F11:v// Constant vector of (1 -1 1 1 1 -1 1 1)

mov (1)f0.08b'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:

SWZr1r0.0zx-1// Expected results: r1.x = 0; r1.y = r0.z; r1.z = r0.x; r1.w = -1

GEN:

mov (1)f0.08b'01100110// Set flag and masked out channels x and w

(f0.0)sel (8) r1.0<1>:f r0.yzxy0x000F000F:v// Constant vector of (0 0 0 -1 0 0 0 -1)

7.3.2 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 DirectX 10 Vertex Shader macro of matrix-vector product that is implemented on GEN in the pair of 4-component vector mode. The DirectX 10 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, NoDDCChk}

dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}

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

7.3.3 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.

7.3.4 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 DirectX 10 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 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 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.#
```

7.3.5 Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of DirectX Shader instructions.

7.3.5.1 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 DirectX 10 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

There are several ways to translate the DirectX instruction, depending on the operand/instruction modifiers present in the DirectX instruction. If predicate is not present and the source swizzle is replication, say, src0.y, which is r2-r3, the translation could be as the following instructions

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:fr8:f // All destination color chan's are same

mov (16) r12:fr8:f // MOV is faster than most MathBox func's

mov (16) r14:fr8: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 DirectX 10 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:fr8:f // All destination color chan's are same

(f2[.any4v|.all4v]) mov (16) r12:fr8:f // MOV is faster than most MathBox func's

(f3[.any4v|.all4v]) mov (16) r14:fr8:f // These MOV's are compressed instructions

The same instructions works also for predication with select component modifier. We simply replase f0 to f3 above by the selected flag register, say, f1. The modifier of any4h/all4v would also work.

7.3.5.2 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 in DirectX 10 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
```

7.3.6 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 absoluteadd (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.

IP is aligned to an 8-byte boundary, thus the 3 LSBs 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) IP4*16// jumps to HERE_1

add (1) IP0x35// 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 loopadd (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).

7.3.7 Execution Masking

7.3.7.1 Branching

Example. 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).

7.3.7.2 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 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...
```

7.3.7.3 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. 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
....
```

7.3.7.4 Compound Branches

Compound branches are supported through the ability logically combine flag registers for each intermediate result.

Example 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 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...
```

7.3.7.5 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

Looping 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 *Looping*) 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 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-through

...
```

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 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 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-through

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 *Looping*. This save/restore operations are not required if the loop being entered does not have any occurrence of a continue instruction.

Example 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-through

...

7.3.7.6 Indexed Jump

Example 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...
```

Revision History

Revision Number	Description	Revision Date
1.0	First 2012 OpenSource edition	May 2012

§§