OpenCL Extended Instruction Set Specification

Boaz Ouriel, Intel

Version 1.00, Revision 5

December 5, 2019

Contents

Contributors and Acknowledgments

- Yaxun Liu, AMD
- Brian Sumner, AMD
- Marty Johnson, AMD
- Mandana Baregheh, AMD
- Andrew Richards, Codeplay
- Ben Ashbaugh, Intel
- Alexey Bader, Intel
- Guy Benyei, Intel
- Raun Krisch, Intel
- Boaz Ouriel, Intel
- Yuan Lin, NVIDIA
- Lee Howes, Qualcomm
- Chihong Zang, Qualcomm
- Ben Gaster, Qualcomm
- Jack Liu, Qualcomm
- Ronan Keryell, Xilinx

1 Introduction

This is the specification of **OpenCL.std** extended instruction set.

The library is imported into a SPIR-V module in the following manner:

<ext-inst-id> OpExtInstImport "OpenCL.std"

The library can only be imported when **Memory Model** is set to **OpenCL**

2 Binary Form

This section contains the semantics and exact form of execution of OpenCL extended instructions using the **OpExtInst** instruction.

In this section we use the following naming conventions:

- *void* denote an **OpTypeVoid**.
- *half*, *float* and *double* denote an **OpTypeFloat** with a width of 16, 32 and 64 bits respectively.
- *i8*, *i16*, *i32* and *i64* denote an **OpTypeInt** with a width of 8, 16, 32 and 64 bits respectively.
- *bool* denotes an **OpTypeBool**.

- *size_t* denotes an i32 when the **Addressing Model** is **Physical32** and i64 when the **Addressing Model** is **Physical64**.
- *vector(n)* denotes an **OpTypeVector** where *n* indicates the component count.
  - *vector*(n_1, n_2, …, n_i) abbreviates *vector*(n_1), *vector*(n_2), … or *vector*(n_i).
- *integer* denotes *i8*, *i16*, *i32* or *i64*.
- *floating-point* denotes *half*, *float*, *double*.
- *pointer*(*storage*) denotes an **OpTypePointer** which points to *storage* **Storage Class**.
  - *pointer*(*constant*) denotes an OpTypePointer with **UniformConstant Storage Class**.
  - *pointer*(*generic*) denotes an OpTypePointer with **Generic Storage Class**.
  - *pointer*(*global*) denotes an OpTypePointer with **CrossWorkgroup Storage Class**.
  - *pointer*(*local*) denotes an OpTypePointer with **Workgroup Storage Class**.
  - *pointer*(*private*) denotes an OpTypePointer with **Function Storage Class**.
  - *pointer*(s_1, s_2, …, s_i) abbreviates *pointer*(s_1), *pointer*(s_2), … or *pointer*(s_i).
- *image* defines all types of image memory objects (See image encoding section).
- *sampler* a SPIR-V sampler object (See sampler encoding section).

2.1 Math extended instructions

This section describes the list of external math instructions. The external math instructions are categorized into the following:

- A list of instructions that have scalar or vector argument versions, and,
- A list of instructions that only take scalar float arguments.

The vector versions of the math instructions operate component-wise. The description is per-component.

The math instructions are not affected by the prevailing rounding mode in the calling environment, and always return the same value as they would if called with the round to nearest even rounding mode.

acos

Compute the arc cosine of *x*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	0	*<id>* *x*

acosh

Compute the inverse hyperbolic cosine of *x* .

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	1	*<id>* *x*

acospi

Compute **acos**(*x*) / π.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	2	*<id>* *x*

asin

Compute the arc sine of *x*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	3	*<id>* *x*

asinh

Compute the inverse hyperbolic sine of *x*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	4	*<id>* *x*

asinpi

Compute **asin**(*x*) / π.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	5	*<id>* *x*

atan

Compute the arc tangent of *x*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	6	*<id>* *x*

atan2

Compute the arc tangent of *y* / *x*.

Result is an angle in radians.

Result Type, *y* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	7	*<id>* *y*	*<id>* *x*

atanh

Compute the hyperbolic arc tangent of *x*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	8	*<id>* *x*

atanpi

Compute **atan**(*x*) / *π*.

Result is an angle in radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	9	*<id>* *x*

atan2pi

Compute **atan**(*y*, *x*) / *π*.

Result is an angle in radians.

Result Type, *y* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	10	*<id>* *y*	*<id>* *x*

cbrt

Compute the cube root of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	11	*<id>* *x*

ceil

Round *x* to integral value using the round to positive infinity rounding mode.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	12	*<id>* *x*

copysign

Returns *x* with its sign changed to match the sign of *y*.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	13	*<id>* *x*	*<id>* *y*

cos

Compute the cosine of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	14	*<id>* *x*

cosh

Compute the hyperbolic cosine of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	15	*<id>* *x*

cospi

Compute **cos**(*x*) / π radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	16	*<id>* *x*

erfc

Complementary error function of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	17	*<id>* *x*

erf

Error function of *x* encountered in integrating the normal distribution.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	18	*<id>* *x*

exp

Compute the base-e exponential of *x*. (i.e. e^x)

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	19	*<id>* *x*

exp2

Computes 2 raised to the power of *x*. (i.e. 2^x)

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	20	*<id>* *x*

exp10

Computes 10 raised to the power of *x*. (i.e. 10^x)

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	21	*<id>* *x*

expm1

Computes $e^x - 1.0$.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	22	*<id>* *x*

fabs

Compute the absolute value of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	23	*<id>* *x*

fdim

x - *y* if *x* > *y*, *+0* if *x* is less than or equal to *y*.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	24	*<id>* *x*	*<id>* *y*

floor

Round *x* to the integral value using the round to negative infinity rounding mode.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	25	*<id>* *x*

fma

Compute the correctly rounded floating-point representation of the sum of *c* with the infinitely precise product of *a* and *b*. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.

Result Type, *a*, *b* and *c* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	26	*<id>* *a*	*<id>* *b*	*<id>* *c*

fmax Returns *y* if *x* < *y*, otherwise it returns *x*. If one argument is a NaN, **fmax** returns the other argument. If both arguments are NaNs, **fmax** returns a NaN. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type. **Note: fmax** behaves as defined by C99 and may not match the IEEE 754-2008 definition for **maxNum** with regard to signaling NaNs. Specifically, signaling NaNs may behave as quiet NaNs							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	27	*<id>* *x*	*<id>* *y*

fmin Returns *y* if *y* < *x*, otherwise it returns *x*. If one argument is a NaN, **fmin** returns the other argument. If both arguments are NaNs, **fmin** returns a NaN. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type. **Note: fmin** behaves as defined by C99 and may not match the IEEE 754-2008 definition for **minNum** with regard to signaling NaNs. Specifically, signaling NaNs may behave as quiet NaNs							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	28	*<id>* *x*	*<id>* *y*

fmod Modulus. Returns *x* - *y* * **trunc**(*x*/*y*). *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	29	*<id>* *x*	*<id>* *y*

fract

Returns **fmin**(*x* - **floor**(*x*), *0x1.fffffep-1f*). **floor**(*x*) is returned in *ptr*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

ptr must be a *pointer*(*global*, *local*, *private*, *generic*) to *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type, or must be a pointer to the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	30	*<id>* *x*	*<id>* *ptr*

frexp

Extract the mantissa and exponent from *x*. The *Result Type* holds the mantissa, and *exp* points to the exponent. For each component the mantissa returned is a *floating-point* with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

exp must be a *pointer*(*global*, *local*, *private*, *generic*) to *i32* or *vector(2,3,4,8,16)* of *i32* values.

Result Type and *x* operands must be of the same type. *exp* operand must point to an *i32* with the same component count as *Result Type* and *x* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	31	*<id>* *x*	*<id>* *exp*

hypot

Compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	32	*<id>* *x*	*<id>* *y*

ilogb

Return the exponent of *x* as an *i32* value.

Result Type must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

x must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

Result Type and *x* operands must have the same component count.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	33	*<id>* *x*

ldexp

Multiply *x* by 2 to the power *k*.

k must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

Result Type and *x* operands must be of the same type. *k* operand must have the same component count as *Result Type* and *x* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	34	*<id>* *x*	*<id>* *k*

lgamma

Log gamma function of *x*. Returns the natural logarithm of the absolute value of the gamma function.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	35	*<id>* *x*

lgamma_r

Log gamma function of *x*. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the *signp* operand

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

signp must be a *pointer*(*global*, *local*, *private*, *generic*) to *i32* or *vector(2,3,4,8,16)* of *i32* values.

Result Type and *x* operands must be of the same type. *signp* operand must point to an *i32* with the same component count as *Result Type* and *x* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	36	*<id>* *x*	*<id>* *signp*

log

Compute the natural logarithm of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	37	*<id>* *x*

log2

Compute the base 2 logarithm of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	38	*<id>* *x*

log10

Compute the base 10 logarithm of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	39	*<id>* *x*

log1p

Compute $\mathbf{log_e}(1.0 + x)$.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	40	*<id>* *x*

logb

Compute the exponent of *x*, which is the integral part of $\log_r |x|$.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	41	*<id>* *x*

mad

Computes $a * b + c$. **mad** may compute $a * b + c$ with reduced accuracy in the embedded profile - see the OpenCL SPIR-V Environment specification for details. On some hardware the **mad** instruction may provide better performance than the expanded computation of $a * b + c$.

Result Type, *a*, *b* and *c* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: For some usages, e.g. **mad**(*a*, *b*, -*a* * *b*), the definition of **mad** is loose enough that almost any result is allowed from **mad** for some values of *a* and *b*.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	42	*<id>* *a*	*<id>* *b*	*<id>* *c*

maxmag

Returns *x* if $|x| > |y|$, *y* if $|y| > |x|$, otherwise **fmax**(*x*, *y*).

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	43	*<id>* *x*	*<id>* *y*

minmag

Returns x if $| x | < | y |$, y if $| y | < | x |$, otherwise **fmin**(x, y).

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	44	*<id>* *x*	*<id>* *y*

modf

Decompose a *floating-point* number. The **modf** instruction breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

iptr must be a *pointer*(*global*, *local*, *private*, *generic*) to *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type, or must be a pointer to the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	45	*<id>* *x*	*<id>* *iptr*

nan

Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.

Result Type must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

nancode must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

Result Type and *nancode* operands must have the same component count. The primitve data type size of *nancode* and *Result Type* must be equal.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set <*id*>	46	*<id>* *nancode*

nextafter Computes the next representable *floating-point* value following *x* in the direction of *y*. Thus, if *y* is less than *x*, **nextafter** returns the largest representable floating-point number less than *x*. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	47	*<id>* *x*	*<id>* *y*

pow Compute *x* to the power *y*. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	48	*<id>* *x*	*<id>* *y*

pown Compute *x* to the power *y*, where *y* is an *i32* integer. *y* must be *i32* or *vector(2,3,4,8,16)* of *i32* values. *Result Type* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. *Result Type* and *x* operands must be of the same type. *y* operand must have the same component count as *Result Type* and *x* operands.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	49	*<id>* *x*	*<id>* *y*

powr Compute *x* to the power *y*, where *x* is >= 0. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	50	*<id>* *x*	*<id>* *y*

remainder

Compute the value r such that r = *x* - n**y*, where n is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, n shall be the even one. If r is zero, it is given the same sign as *x*.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	51	*<id>* *x*	*<id>* *y*

remquo

The **remquo** instruction computes the value r such that r = *x* - k**y*, where k is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, k shall be the even one. If r is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** instruction. **remquo** also calculates at least the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

quo must be a *pointer*(*global*, *local*, *private*, *generic*) to *i32* or *vector(2,3,4,8,16)* of *i32* values.

Result Type, *x* and *y* operands must be of the same type. *quo* operand must point to an *i32* with the same component count as *Result Type*, *x* and *y* operands.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	52	*<id>* *x*	*<id>* *y*	*<id>* *quo*

rint

Round *x* to integral value (using round to nearest even rounding mode) in floating-point format.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	53	*<id>* *x*

rootn

Compute *x* to the power 1/*y*.

y must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

Result Type and *x* operands must be of the same type. *y* operand must have the same component count as *Result Type* and *x* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	54	*<id>* *x*	*<id>* *y*

round

Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	55	*<id>* *x*

rsqrt

Compute inverse square root of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	56	*<id>* *x*

sin

Compute sine of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	57	*<id>* *x*

sincos

Compute sine and cosine of *x* radians. The computed sine is the return value and computed cosine is returned in *cosval*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

cosval must be a *pointer*(*global*, *local*, *private*, *generic*) to *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type, or must be a pointer to the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	58	*<id>* *x*	*<id>* *cosval*

sinh

Compute hyperbolic sine of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	59	*<id>* *x*

sinpi

Compute *sin* (π x) radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	60	*<id>* *x*

sqrt

Compute square root of *x*.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	61	*<id>* *x*

tan

Compute tangent of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	62	*<id>* *x*

tanh

Compute hyperbolic tangent of *x* radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	63	*<id>* *x*

tanpi

Compute *tan* (π x) radians.

Result Type and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	64	*<id>* *x*

tgamma Compute the gamma function of *x*. *Result Type* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	65	*<id>* *x*

trunc Round *x* to integral value using the round to zero rounding mode. *Result Type* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	66	*<id>* *x*

half_cos Compute cosine of *x* radians, where *x* must be in the range $-2^{16} \ldots +2^{16}$.ha *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	67	*<id>* *x*

half_divide Compute *x* / *y*. *Result Type*, *x* and *y* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	68	*<id>* *x*	*<id>* *y*

half_exp Compute the base-e exponential of *x*. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	69	*<id>* *x*

half_exp2 Compute the base 2 exponential of *x*. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	70	*<id>* *x*

half_exp10 Compute the base 10 exponential of *x*. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	71	*<id>* *x*

half_log Compute the natural logarithm of *x*. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	72	*<id>* *x*

half_log2

Compute the base 2 logarithm of *x*.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	73	*<id>* *x*

half_log10

Compute the base 10 logarithm of *x*.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	74	*<id>* *x*

half_powr

Compute *x* to the power *y*, where *x* is >= 0.

Result Type, *x* and *y* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	75	*<id>* *x*	*<id>* *y*

half_recip

Compute the reciprocal of *x*.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	76	*<id>* *x*

half_rsqrt

Compute the inverse square root of *x*.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	77	*<id>* *x*

half_sin

Compute the sine of *x* radians, where *x* must be in the range $-2^{16} \ldots +2^{16}$.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	78	*<id>* *x*

half_sqrt

Compute the the square root of *x*.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	79	*<id>* *x*

half_tan

Compute tangent value of *x* radians, where *x* must be in the range $-2^{16} \ldots +2^{16}$.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	80	*<id>* *x*

native_cos

Compute cosine of *x* radians over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	81	*<id>* *x*

native_divide

Compute *x* / *y* over an implementation-defined range. The maximum error is implementation-defined.

Result Type, *x* and *y* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	82	*<id>* *x*	*<id>* *y*

native_exp

Compute the base-e exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	83	*<id>* *x*

native_exp2

Compute the base- 2 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined..

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	84	*<id>* *x*

native_exp10

Compute the base- 10 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined..

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	85	*<id>* *x*

native_log

Compute natural logarithm of *x* over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	86	*<id>* *x*

native_log2

Compute a base 2 logarithm of *x* over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	87	*<id>* *x*

native_log10

Compute a base 10 logarithm of *x* over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	88	*<id>* *x*

native_powr

Compute *x* to the power *y*, where *x* is >= 0.

Result Type, *x* and *y* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	89	*<id>* *x*	*<id>* *y*

native_recip

Compute reciprocal of *x* over an implementation-defined range. The range of x and y are implementation-defined. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	90	*<id>* *x*

native_rsqrt

Compute inverse square root of *x* over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	91	*<id>* *x*

native_sin

Compute sine of *x* radians over an implementation-defined range. The maximum error is implementation-defined.

Result Type and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	92	*<id>* *x*

native_sqrt Compute the square root of *x* over an implementation-defined range. The maximum error is implementation-defined. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type. **Note:** This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	93	*<id>* *x*

native_tan Compute tangent value of *x* radians over an implementation-defined range. The maximum error is implementation-defined. *Result Type* and *x* must be *float* or *vector(2,3,4,8,16)* of *float* values. All of the operands, including the *Result Type* operand, must be of the same type. **Note:** This instruction may map to one or more native device instructions and will typically have better performance compared to the corresponding non-native instruction. Support for denormal values is implementation-defined for native instructions.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	94	*<id>* *x*

2.2 Integer instructions

This section describes the list of integer instructions that take scalar or vector arguments. The vector versions of the integer instructions operate component-wise. The description is per-component.

s_abs

Returns $|x|$, where *x* is treated as signed integer.

Result Type and *x* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

This instruction can be decorated with **NoSignedWrap**.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	141	*<id>* *x*

s_abs_diff

Returns $| x - y |$ without modulo overflow, where *x* and *y* are treated as signed integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	142	*<id>* *x*	*<id>* *y*

s_add_sat

Returns the saturated value of $x + y$, where *x* and *y* are treated as signed integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	143	*<id>* *x*	*<id>* *y*

u_add_sat

Returns the saturated value of *x* + *y*, where *x* and *y* are treated as unsigned integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	144	*<id>* *x*	*<id>* *y*

s_hadd

Returns the value of (*x* + *y*) >> 1, where *x* and *y* are treated as signed integers. The intermediate sum does not modulo overflow.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	145	*<id>* *x*	*<id>* *y*

u_hadd

Returns the value of (*x* + *y*) >> 1, where *x* and *y* are treated as unsigned integers. The intermediate sum does not modulo overflow.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	146	*<id>* *x*	*<id>* *y*

s_rhadd

Returns the value of (*x* + *y* + 1) >> 1, where *x* and *y* are treated as signed integers. The intermediate sum does not modulo overflow.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	147	*<id>* *x*	*<id>* *y*

u_rhadd							
Returns the value of (*x* + *y* + 1) >> 1, where *x* and *y* are treated as unsigned integers. The intermediate sum does not modulo overflow. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	148	*<id>* *x*	*<id>* *y*

s_clamp								
Returns *s_min*(*s_max*(*x*,*minval*),*maxval*), where *x*, *minval*, and *maxval* are treated as signed integers. Results are undefined if *minval* > *maxval*. *Result Type*, *x*, *minval* and *maxval* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.								
8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	149	*<id>* *x*	*<id>* *minval*	*<id>* *maxval*

u_clamp								
Returns *u_min*(*u_max*(*x*,*minval*),*maxval*), where *x*, *minval*, and *maxval* are treated as unsigned integers. Results are undefined if *minval* > *maxval*. *Result Type*, *x*, *minval* and *maxval* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.								
8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	150	*<id>* *x*	*<id>* *minval*	*<id>* *maxval*

clz						
Returns the number of leading 0 bits in *x*, starting at the most significant bit position. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector. *Result Type* and *x* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	151	*<id>* *x*

ctz

Returns the count of trailing 0 bits in *x*. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector.

Result Type and *x* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	152	*<id>* *x*

s_mad_hi

Returns *mul_hi*(*a*, *b*) + *c*, where *a*,*b* and *c* are treated as signed integers.

Result Type, *a*, *b* and *c* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	153	*<id>* *a*	*<id>* *b*	*<id>* *c*

u_mad_sat

Returns *x* * *y* + *z* and sautrates the result where *x*, *y* and *z* are treated as unsigned integers.

Result Type, *x*, *y* and *z* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	154	*<id>* *x*	*<id>* *y*	*<id>* *z*

s_mad_sat

Returns *x* * *y* + *z* and sautrates the result where *x*, *y* and *z* are treated as signed integers.

Result Type, *x*, *y* and *z* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	155	*<id>* *x*	*<id>* *y*	*<id>* *z*

s_max

Returns *y* if *x* < *y*, otherwise it returns *x*, where *x* and *y* are treated as signed integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	156	*<id>* *x*	*<id>* *y*

u_max

Returns *y* if *x* < *y*, otherwise it returns *x*, where *x* and *y* are treated as unsigned integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	157	*<id>* *x*	*<id>* *y*

s_min

Returns *y* if *y* < *x*, otherwise it returns *x*, where *x* and *y* are treated as signed integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	158	*<id>* *x*	*<id>* *y*

u_min

Returns *y* if *y* < *x*, otherwise it returns *x*, where *x* and *y* are treated as unsigned integers.

Result Type, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	159	*<id>* *x*	*<id>* *y*

s_mul_hi Computes *x* * *y* and returns the high half of the product of *x* and *y*, where *x* and *y* are treated as signed integers. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	160	*<id>* *x*	*<id>* *y*

rotate For each element in *v*, the bits are shifted left by the number of bits given by the corresponding element in *i*. Bits shifted off the left side of the element are shifted back in from the right. *Result Type*, *v* and *i* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	161	*<id>* *v*	*<id>* *i*

s_sub_sat Returns the saturated value of *x* - *y*, where *x* and *y* are treated as signed integers. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	162	*<id>* *x*	*<id>* *y*

u_sub_sat Returns the saturated value of *x* - *y*, where *x* and *y* are treated as unsigned integers. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	163	*<id>* *x*	*<id>* *y*

u_upsample

When *hi* and *lo* component type is i8:

Result = ((upcast. . . to i16)*hi* << 8) | *lo*

When *hi* and *lo* component type is i16:

Result = ((upcast. . . to i32)*hi* << 8) | *lo*

When *hi* and *lo* component i32:

Result = ((upcast. . . to i64)*hi* << 8) | *lo*

hi and *lo* are treated as unsigned integers.

hi and *lo* must be *i8*, *i16* or *i32* or *vector(2,3,4,8,16)* of *i8*, *i16* or *i32* values.

Result Type must be *i16*, *i32* or *i64* or *vector(2,3,4,8,16)* of *i16*, *i32* or *i64* values.

hi and *lo* operands must be of the same type. When *hi* and *lo* component type is i8, the *Result Type* component type must be i16. When *hi* and *lo* component type is i16, the *Result Type* component type must be i32. When *hi* and *lo* component type is i32, the *Result Type* component type must be i64. *Result Type* must have the same component count as *hi* and *lo* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	164	*<id>* *hi*	*<id>* *lo*

s_upsample

When *hi* and *lo* component type is i8:

Result = ((upcast. . . to i16)*hi* << 8) | *lo*

When *hi* and *lo* component type is i16:

Result = ((upcast. . . to i32)*hi* << 8) | *lo*

When *hi* and *lo* component i32:

Result = ((upcast. . . to i64)*hi* << 8) | *lo*

hi is treated as a signed integer and *lo* is treated as an unsigned integer.

hi and *lo* must be *i8*, *i16* or *i32* or *vector(2,3,4,8,16)* of *i8*, *i16* or *i32* values.

Result Type must be *i16*, *i32* or *i64* or *vector(2,3,4,8,16)* of *i16*, *i32* or *i64* values.

hi and *lo* operands must be of the same type. When *hi* and *lo* component type is i8, the *Result Type* component type must be i16. When *hi* and *lo* component type is i16, the *Result Type* component type must be i32. When *hi* and *lo* component type is i32, the *Result Type* component type must be i64. *Result Type* must have the same component count as *hi* and *lo* operands.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	165	*<id>* *hi*	*<id>* *lo*

popcount

Returns the number of non-zero bits in *x*.

Result Type and *x* must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	166	*<id>* *x*

s_mad24

Multiply two 24-bit integer values *x* and *y* and add the 32-bit integer result to the 32-bit integer *z*. Refer to definition of s_mul24 to see how the 24-bit integer multiplication is performed.

Result Type, *x*, *y* and *z* must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	167	*<id>* *x*	*<id>* *y*	*<id>* *z*

u_mad24

Multiply two 24-bit integer values *x* and *y* and add the 32-bit integer result to the 32-bit integer *z*. Refer to definition of u_mul24 to see how the 24-bit integer multiplication is performed.

Result Type, *x*, *y* and *z* must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	168	*<id>* *x*	*<id>* *y*	*<id>* *z*

s_mul24

Multiply two 24-bit integer values *x* and *y*, where *x* and *y* are treated as signed integers. *x* and *y* are 32-bit integers but only the low-order 24 bits are used to perform the multiplication. s_mul24 should only be used when values in *x* and *y* are in the range $[-2^{23}, 2^{23}-1]$. If *x* and *y* are not in this range, the multiplication result is implementation-defined.

Result Type, *x* and *y* must be *i32* or *vector(2,3,4,8,16)* of *i32* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	169	*<id>* *x*	*<id>* *y*

u_mul24							
Multiply two 24-bit integer values *x* and *y*, where *x* and *y* are treated as unsigned integers. *x* and *y* are 32-bit integers but only the low-order 24 bits are used to perform the multiplication. u_mul24 should only be used when values in *x* and *y* are in the range [0, 2^{24}-1]. If *x* and *y* are not in this range, the multiplication result is implementation-defined. *Result Type*, *x* and *y* must be *i32* or *vector(2,3,4,8,16)* of *i32* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	170	*<id>* *x*	*<id>* *y*

u_abs						
Returns |*x*|, where *x* is treated as unsigned integer. *Result Type* and *x* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	201	*<id>* *x*

u_abs_diff							
Returns | *x* - *y* | without modulo overflow, where *x* and *y* are treated as unsigned integers. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	202	*<id>* *x*	*<id>* *y*

u_mul_hi							
Computes *x* * *y* and returns the high half of the product of *x* and *y*, where *x* and *y* are treated as unsigned integers. *Result Type*, *x* and *y* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	203	*<id>* *x*	*<id>* *y*

u_mad_hi Returns *mul_hi*(*a*, *b*) + *c*, where *a*,*b* and *c* are treated as unsigned integers. *Result Type*, *a*, *b* and *c* must be *integer* or *vector(2,3,4,8,16)* of *integer* values. All of the operands, including the *Result Type* operand, must be of the same type.								
8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	204	*<id>* *a*	*<id>* *b*	*<id>* *c*

2.3 Common instructions

This section describes the list of common instructions that take scalar or vector arguments. The vector versions of the integer instructions operate component-wise. The description is per-component. The common instructions are implemented using the round to nearest even rounding mode.

fclamp

Returns *fmin*(*fmax*(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.

Result Type, *x*, *minval* and *maxval* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	95	*<id>* *x*	*<id>* *minval*	*<id>* *maxval*

degrees

Converts *radians* to degrees, i.e. (180 / π) * *radians*.

Result Type and *radians* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	96	*<id>* *radians*

fmax_common

Returns *y* if *x* < *y*, otherwise it returns *x*. If x or y are infinite or NaN, the return values are undefined.

Result Type, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	97	*<id>* *x*	*<id>* *y*

fmin_common Returns *y* if *y* < *x*, otherwise it returns *x*. If x or y are infinite or NaN, the return values are undefined. *Result Type*, *x* and *y* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	98	*<id>* *x*	*<id>* *y*

mix Returns the linear blend of *x* & *y* implemented as: $x + (y - x) * a$ *Result Type*, *x*, *y* and *a* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type. **Note:** This instruction can be implemented using contractions such as **mad** or **fma**.								
8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	99	*<id>* *x*	*<id>* *y*	*<id>* *a*

radians Converts *degrees* to radians, i.e. (π / 180) * *degrees*. *Result Type* and *degrees* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	100	*<id>* *degrees*

step Returns 0.0 if *x* < *edge*, otherwise it returns 1.0. *Result Type*, *edge* and *x* must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	101	*<id>* *edge*	*<id>* *x*

smoothstep

Returns 0.0 if $x \Leftarrow edge_0$ and 1.0 if $x >= edge_1$ and performs smooth Hermite interpolation between 0 and 1, when $edge_0 < x < edge_1$.

This is equivalent to :

t = *fclamp*((x - $edge_0$) / ($edge_1$ - $edge_0$), 0, 1);

return t * t * (3 - 2 * t);

Results are undefined if $edge_0 >= edge_1$ or if x, $edge_0$ or $edge_1$ is a NaN.

Result Type, $edge_0$, $edge_1$ and x must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

Note: This instruction can be implemented using contractions such as **mad** or **fma**.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	102	*<id>* $edge_0$	*<id>* $edge_1$	*<id>* x

sign

Returns 1.0 if $x > 0$, -0.0 if x = -0.0, +0.0 if x = +0.0, or -1.0 if $x < 0$. Returns 0.0 if x is a NaN.

Result Type and x must be *floating-point* or *vector(2,3,4,8,16)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	103	*<id>* x

2.4 Geometric instructions

This section describes the list of geometric instructions. In this section *x*,*y*,*z* and *w* denote the first, second, third and fourth component respecitively, of vectors with 3 and four components.The geometric instructions are implemented using the round to nearest even rounding mode.

Note: The geometric instructions can be implemented using contractions such as mad or fma

cross

Returns the cross product of p_0.xyz and p_1.xyz.

When the vector component count is 4, the w component returned will be 0.0.

Result Type, p_0 and p_1 must be *vector(3,4)* of *floating-point* values.

All of the operands, including the *Result Type* operand, must be of the same type.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	104	*<id>* p_0	*<id>* p_1

distance

Returns the distance between p_0 and p_1. This is calculated as *length*(p_0 - p_1).

Result Type must be *floating-point*.

p_0 and p_1 must be *floating-point* or *vector(2,3,4)* of *floating-point* values.

p_0 and p_1 operands must have the same type. *Result Type*, p_0 and p_1 operands must have the same component type

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	105	*<id>* p_0	*<id>* p_1

length

Return the length of vector *p*, i.e. $sqrt(p.x^2 + p.y^2 + \ldots)$

Result Type must be *floating-point*.

p must be *floating-point* or *vector(2,3,4)* of *floating-point* values.

Result Type and *p* operands must have the same component type

6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	106	*<id>* *p*

normalize						
Returns a vector in the same direction as *p* but with a length of 1. *Result Type* and *p* must be *floating-point* or *vector(2,3,4)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	107	*<id>* *p*

fast_distance							
Returns *fast_length*(p_0 - p_1). *Result Type* must be *floating-point*. p_0 and p_1 must be *floating-point* or *vector(2,3,4)* of *floating-point* values. p_0 and p_1 operands must have the same type. *Result Type*, p_0 and p_1 operands must have the same component type							
7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	108	*<id>* p_0	*<id>* p_1

fast_length						
Return the length of vector *p* computed as: *half_sqrt*($p.x^2 + p.y^2 + \ldots$) *Result Type* must be *floating-point*. *p* must be *vector(2,3,4)* of *floating-point* values. *Result Type* and *p* operands must have the same component type						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	109	*<id>* *p*

fast_normalize Returns a vector in the same direction as *p* but with a length of 1 computed as: *p* * *half_rsqrt*($p.x^2 + p.y^2 \ldots$) The result shall be within 8192 ulps error from the infinitely precise result of: if (*all*(*p* == 0.0f)) { result = *p*; } else { result = *p* / *sqrt*($p.x^2 + p.y^2 + \ldots$); } with the following exceptions : 1) If the sum of squares is greater than FLT_MAX then the value of the floating-point values in the result vector are undefined. 2) If the sum of squares is less than FLT_MIN then the implementation may return back *p*. 3) If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than *sqrt*(FLT_MIN) may be flushed to zero before proceeding with the calculation. *Result Type* and *p* must be *floating-point* or *vector(2,3,4)* of *floating-point* values. All of the operands, including the *Result Type* operand, must be of the same type.						
6	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	110	*<id>* *p*

2.5 Relational instructions

This section describes the list of relational instructions that take scalar or vector arguments. The vector versions of the integer instructions operate component-wise. The description is per-component.

bitselect

Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.

Result Type, *a*, *b* and *c* must be *floating-point* or *integer* or *vector(2,3,4,8,16)* of *floating-point* or *integer* values.

All of the operands, including the *Result Type* operand, must be of the same type.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	186	*<id>* *a*	*<id>* *b*	*<id>* *c*

select

For each component of a vector type, the result is *a* if the most significant bit of *c* is zero, otherwise it is *b*.

For a scalar type, the result is *a* if *c* is zero, otherwise it is *b*.

c must be *integer* or *vector(2,3,4,8,16)* of *integer* values.

Result Type, *a* and *b* must be *floating-point* or *integer* or *vector(2,3,4,8,16)* of *floating-point* or *integer* values.

Result Type, *a* and *b* must have the same type. *c* operand must have the same component count and component bit width as the rest of the operands.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	187	*<id>* *a*	*<id>* *b*	*<id>* *c*

2.6 Vector Data Load and Store instructions

This section describes the list of instructions that allow reading and writing of vector types from a pointer to memory.

vloadn

Reads *n* components from the address computed as (*p* + (*offset* * *n*)) and creates a vector result value from the *n* components.

The computed address must be 8-bit aligned if *p* points to an i8 value; 16-bit aligned if *p* points to an i16 or half value; 32-bit aligned if *p* points to an i32 or float value; 64-bit aligned if *p* points to an i64 or double value.

offset must be *size_t*.

p must be a *pointer*(*global*, *local*, *private*, *constant*, *generic*) to *floating-point*, *integer*.

Result Type must be *vector(2,3,4,8,16)* of *floating-point* or *integer* values.

Result Type component count must be equal to *n* and its component type must be equal to the type pointed by *p*.

n must be 2, 3, 4, 8 or 16.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	171	*<id>* *offset*	*<id>* *p*	Literal *n*

vstoren

Writes *n* components from the *data* vector value to the address computed as (*p* + (*offset* * *n*)), where *n* is equal to the component count of the vector *data*.

The computed address must be 8-bit aligned if *p* points to an i8 value; 16-bit aligned if *p* points to an i16 or half value; 32-bit aligned if *p* points to an i32 or float value; 64-bit aligned if *p* points to an i64 or double value.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *floating-point*, *integer*.

data must be *vector(2,3,4,8,16)* of *floating-point* or *integer* values.

data component type must be equal to the type pointed by *p*.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	172	*<id>* *data*	*<id>* *offset*	*<id>* *p*

vload_half

Reads a half value from the address computed as (*p* + (*offset*)) and converts it to a float result value.

The computed address must be 16-bit aligned.

Result Type must be *float*.

offset must be *size_t*.

p must be a *pointer*(*global*, *local*, *private*, *constant*, *generic*) to *half*.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	173	*<id>* *offset*	*<id>* *p*

vload_halfn

Reads *n* half components from the address (*p* + (*offset* * *n*)), converts to *n* float components, and creates a float vector result value from the *n* float components.

The computed address must be 16-bit aligned.

offset must be *size_t*.

p must be a *pointer*(*global*, *local*, *private*, *constant*, *generic*) to *half*.

Result Type must be *vector(2,3,4,8,16)* of *float* values.

Result Type component count must be equal to *n*.

n must be 2, 3, 4, 8 or 16.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	174	*<id>* *offset*	*<id>* *p*	Literal *n*

vstore_half

Converts the *data* float or double value to a half value using the default rounding mode and writes the half value to the address computed as (*p* + *offset*).

The computed address must be 16-bit aligned.

data must be *float* or *double*.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

8	12	*<id> Result Type*	*Result <id>*	extended instructions set *<id>*	175	*<id> data*	*<id> offset*	*<id> p*

vstore_half_r

Converts the *data* float or double value to a half value using the specified rounding mode *mode* and writes the half value to the address computed as (*p* + *offset*).

The computed address must be 16-bit aligned.

data must be *float* or *double*.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

9	12	*<id> Result Type*	*Result <id>*	extended instructions set *<id>*	176	*<id> data*	*<id> offset*	*<id> p*	FP Rounding Mode *mode*

vstore_halfn

Converts the *data* vector of float or vector of double values to a vector of half values using the default rounding mode and writes the half values to memory.

Let *n* be the component count of the vector *data*.

The *n* components from the converted vector of half values are written to the address computed as (*p* + (*offset* * *n*)).

The computed address must be 16-bit aligned.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

data must be *vector(2,3,4,8,16)* of *float* or *double* values.

8	12	*<id> Result Type*	*Result <id>*	extended instructions set *<id>*	177	*<id> data*	*<id> offset*	*<id> p*

vstore_halfn_r

Converts the *data* vector of float or vector of double values to a vector of half values using the specified rounding mode *mode* and writes the half values to memory.

Let *n* be the component count of the vector *data*.

The *n* components from the converted vector of half values are written to the address computed as (*p* + (*offset* * *n*)).

The computed address must be 16-bit aligned.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

data must be *vector(2,3,4,8,16)* of *float* or *double* values.

9	12	*<id> Result Type*	*Result <id>*	extended instructions set *<id>*	178	*<id> data*	*<id> offset*	*<id> p*	FP Rounding Mode *mode*

vloada_halfn

Reads a vector of *n* half values from aligned memory and converts it to a float vector result value.

For *n* equal to 2, 4, 8, and 16, the vector of *n* half values is read from the address computed as (*p* + (*offset* * *n*)). The computed address must be aligned to (*sizeof(half)* * *n*) bytes.

For *n* equal to 3, the vector of *n* half values are read from the address computed as (*p* + (*offset* * 4)). The computed address must be aligned to (*sizeof(half)* * 4) bytes.

offset must be *size_t*.

p must be a *pointer*(*global*, *local*, *private*, *constant*, *generic*) to *half*.

Result Type must be *vector(2,3,4,8,16)* of *float* values.

Result Type component count must be equal to *n*.

n must be 2, 3, 4, 8 or 16.

8	12	*<id> Result Type*	*Result <id>*	extended instructions set *<id>*	179	*<id> offset*	*<id> p*	Literal *n*

vstorea_halfn

Converts the *data* vector of float or vector of double values to a vector of half values using the default rounding mode, and then writes the converted vector of half values to aligned memory.

Let *n* be the component count of the vector *data*.

For *n* equal to 2, 4, 8, and 16, the converted vector of half values is written to the address computed as (*p* + (*offset* * *n*)). The computed address must be aligned to (*sizeof(half)* * *n*) bytes.

For *n* equal to 3, the converted vector of half values is written to the address computed as (*p* + (*offset* * 4)). The computed address must be aligned to (*sizeof(half)* * 4) bytes.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

data must be *vector(2,3,4,8,16)* of *float* or *double* values.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	180	*<id>* *data*	*<id>* *offset*	*<id>* *p*

vstorea_halfn_r

Converts the *data* vector of float or vector of double values to a vector of half values using the specified rounding mode *mode*, and then write the converted vector of half values to aligned memory.

Let *n* be the component count of the vector *data*.

For *n* equal to 2, 4, 8, and 16, the converted vector of half values is written to the address computed as (*p* + (*offset* * *n*)). The computed address must be aligned to (*sizeof(half)* * *n*) bytes.

For *n* equal to 3, the converted vector of half values is written to the address computed as (*p* + (*offset* * 4)). The computed address must be aligned to (*sizeof(half)* * 4) bytes.

offset must be *size_t*.

Result Type must be *void*.

p must be a *pointer*(*global*, *local*, *private*, *generic*) to *half*.

data must be *vector(2,3,4,8,16)* of *float* or *double* values.

9	12	*<id>* *Result Type*	*Result <id>*	extended instruc-tions set *<id>*	181	*<id>* *data*	*<id>* *offset*	*<id>* *p*	FP Rounding Mode *mode*

2.7 Miscellaneous Vector instructions

This section describes additional vector instructions.

<table>
<tr><td colspan="8">shuffle

Construct a permutation of components from x vector value, returning a vector value with the same component type as x and component component count that is the same as shuffle mask.

For this instruction, only the ilogb(2 m -1) least significant bits of each mask element are considered, where m is equal to the component count of x.

shuffle mask operand specifies, for each component in the result vector, which component of x it gets.

The size of each component in shuffle mask must match the size of each component in Result Type.

Result Type must have the same component type as x and component count as shuffle mask.

shuffle mask must be vector(2,4,8,16) of integer values.

Result Type and x must be vector(2,4,8,16) of floating-point or integer values.</td></tr>
<tr><td>7</td><td>12</td><td><id>
Result Type</td><td>Result <id></td><td>extended instructions set <id></td><td>182</td><td><id>
x</td><td><id>
shuffle mask</td></tr>
</table>

shuffle2

Construct a permutation of components from *x* and *y* vector values, returning a vector value with the same component type as *x* and *y* and component count that is the same as *shuffle mask*.

For this instruction, only the *ilogb*(2 *m* - 1) + 1 least significant bits of each mask component are considered, where *m* is equal to the component count of *x* and *y*.

shuffle mask operand specifies, for each component in the result vector, which component of *x* or *y* it gets. Where component count begins with *x* and then proceeds to *y*.

x and *y* must be of the same type.

The size of each component in *shuffle mask* must match the size of each component in *Result Type*.

Result Type must have the same component type as *x* and component count as *shuffle mask*.

shuffle mask must be *vector(2,4,8,16)* of *integer* values.

Result Type, *x* and *y* must be *vector(2,4,8,16)* of *floating-point* or *integer* values.

8	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	183	*<id>* *x*	*<id>* *y*	*<id>* *shuffle mask*

2.8 Misc instructions

This section describes additional miscellaneous instructions.

printf

The *printf* extended instruction writes output to an implementation-defined stream such as stdout under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The printf instruction returns when the end of the format string is encountered

printf returns 0 if it was executed successfully and -1 otherwise

Result Type must be *i32*.

format must be a *pointer*(*constant*) to *i8*.

6 + variable	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	184	*<id>* *format*	*<id>, <id>, . . .* *additional arguments*

prefetch

Prefetch *num_elements* * size in bytes of the type pointed by *p*, into the global cache. The prefetch instruction is applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

num_elements must be *size_t*.

Result Type must be *void*.

ptr must be a *pointer*(*global*) to *floating-point*, *integer* or *vector(2,3,4,8,16)* of *floating-point*, *integer* values.

7	12	*<id>* *Result Type*	*Result <id>*	extended instructions set *<id>*	185	*<id>* *ptr*	*<id>* *num_elements*

A Changes and TBD

- Fork the revision stream, changes section, TBD, etc. from the core specification, so this specification has its own, starting numbering at revision 1. This document now lives independently.

A.1 Changes from Version 0.99, Revision 1

- Move to use the updated image/texturing/sampling, instead of extended instructions. Also, see changes in core specification related to this.
  - 14241 Implement OpenCL Extended Instructions for images/samplers with core OpImageSample instructions
- Fixed internal bugs

- 13455 Merged the OpenCL 1.2, 2.0, and 2.1 extended-instruction set into a single OpenCL extended-instruction set.

- Fixed public bugs

A.2 Changes from Version 0.99, Revision 2

- 14679 moved precision information to the OpenCL environment spec
- 14636 clarified trig functions to accept and return radians

A.3 Changes from Version 0.99, Revision 3

- Fixed internal bugs:
  - 14862 removed remaining image instructions as core versions are sufficient
  - 14636 Fixed type-o's in several trig functions accepting radian inputs and/or producing radian results
  - Flattened opcode numbers

A.4 Changes from Version 1.0, Revision 1

- Fixed internal bugs:
  - Issue 8 - order of parameters for prefetch was reversed; pointer operand should be first.
  - Issue 103 - typo: *singp* should be *signp*
- Fixed public bugs
  - 1469 - incorrect specification of **pow** and **pown**

A.5 Changes from Version 1.0, Revision 2

- Fixed internal bugs:
  - Issue 261 - clarified that **s_mad24** and **u_mad24** only support 32-bit integers
  - Issue 262 - added scalars to the types supported by **length**
  - Issue 266 - fixed **shuffle** and **shuffle2** description
  - Issue 267 - fixed description of **ldexp** operands

A.6 Changes from Version 1.0, Revision 3

- Moved image and sampler encoding to the OpenCL environment specification
- Editorial fixes and improvements
- Fixed internal bugs:
  - Issue 271 - storage class inconsistency between **vloadn/vstoren** and **vload_half/vstore_half**
  - Issue 312 - bad wording for **vstorea_halfn**

A.7 Changes from Version 1.0, Revision 4

Support SPV_KHR_no_integer_wrap_decoration, in the **s_abs** instruction.

A.8 Changes from Version 1.0, Revision 5

- Fixed internal bugs:
  - Issue 497 - fixed description for **s_upsample**