OpenCL Extended Instruction Set Specification

Boaz Ouriel, Intel

Version 1.00, Revision 5
April 24, 2020
Contents

1 Introduction

2 Binary Form
   2.1 Math extended instructions
   2.2 Integer instructions
   2.3 Common instructions
   2.4 Geometric instructions
   2.5 Relational instructions
   2.6 Vector Data Load and Store instructions
   2.7 Miscellaneous Vector instructions
   2.8 Misc instructions

A Changes and TBD
   A.1 Changes from Version 0.99, Revision 1
   A.2 Changes from Version 0.99, Revision 2
   A.3 Changes from Version 0.99, Revision 3
   A.4 Changes from Version 1.0, Revision 1
   A.5 Changes from Version 1.0, Revision 2
   A.6 Changes from Version 1.0, Revision 3
   A.7 Changes from Version 1.0, Revision 4
   A.8 Changes from Version 1.0, Revision 5
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, \ldots, n_i)` abbreviates `vector(n_1), vector(n_2), \ldots \text{ 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, \ldots, s_i)` abbreviates `pointer(s_1), pointer(s_2), \ldots \text{ 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.

<table>
<thead>
<tr>
<th>acos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute the arc cosine of $x$.</td>
</tr>
<tr>
<td>Result is an angle in radians.</td>
</tr>
<tr>
<td>Result Type and $x$ must be floating-point or vector(2,3,4,8,16) of floating-point values.</td>
</tr>
<tr>
<td>All of the operands, including the Result Type operand, must be of the same type.</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>acosh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute the inverse hyperbolic cosine of $x$.</td>
</tr>
<tr>
<td>Result is an angle in radians.</td>
</tr>
<tr>
<td>Result Type and $x$ must be floating-point or vector(2,3,4,8,16) of floating-point values.</td>
</tr>
<tr>
<td>All of the operands, including the Result Type operand, must be of the same type.</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
**acospi**

Compute $\text{acos}(x) / \pi$.

*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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th></th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>Result Type</td>
<td>Result &lt;id&gt;</td>
<td>extended instructions set &lt;id&gt;</td>
<td>2</td>
<td>$x$</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th></th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>Result Type</td>
<td>Result &lt;id&gt;</td>
<td>extended instructions set &lt;id&gt;</td>
<td>3</td>
<td>$x$</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th></th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>Result Type</td>
<td>Result &lt;id&gt;</td>
<td>extended instructions set &lt;id&gt;</td>
<td>4</td>
<td>$x$</td>
</tr>
</tbody>
</table>
### asinpi

Compute $\text{asin}(x) / \pi$.

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>5</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>6</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>7</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
</table>

| 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 \( \text{atan}(x) / \pi \).

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 \( \text{atan}(y, x) / \pi \).

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>&lt;id&gt;</td>
<td>&lt;id&gt;</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>&lt;id&gt;</td>
<td>&lt;id&gt;</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
<td>&lt;id&gt;</td>
<td>&lt;id&gt;</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

|   |   | <id>         | <id>         |   | y |
|---|---|--------------|--------------|---|

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Result Type</th>
<th>extended instructions set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>&lt;id&gt;</td>
<td>extended instructions set</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>15</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

cospi

Compute $\cos(x) / \pi$ 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>16</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>
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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>17</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>
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 | <id> | Result Type | 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>19</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>20</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>21</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>22</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>23</th>
<th>x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>24</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>25</th>
<th>x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>26</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>27</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>28</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>

**fmod**

Modulus. Returns $x - y \times \text{trunc}(x/y)$.

*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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>29</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>
**fract**

Returns \( \text{fmin}( x - \text{floor}(x), 0x1.fffffep-1f ) \). \( \text{floor}(x) \) 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>30</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; ptr</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>31</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; exp</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>32</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>
i log  b

Return the exponent of \( x \) as an \( i32 \) value.

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

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

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

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>33</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

l d e x p

Multiply \( x \) by 2 to the power \( k \).

\( k \) must be \( i32 \) or \( \text{vector}(2,3,4,8,16) \) of \( i32 \) values.

*Result Type* and \( x \) must be *floating-point* or \( \text{vector}(2,3,4,8,16) \) 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>34</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; k</th>
</tr>
</thead>
</table>

l g m a r r a

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 \( \text{vector}(2,3,4,8,16) \) of *floating-point* values.

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

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>35</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
*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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>36</th>
<th>(&lt;id&gt;)</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( \text{signp} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>37</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>38</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>39</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**log1p**

Compute \( \log_e(1.0 + x) \).

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>40</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

**logb**

Compute the exponent of \( x \), which is the integral part of \( \log_e |x| \).

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>41</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

**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 \( b \).

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>42</th>
<th>&lt;id&gt;</th>
<th>a</th>
<th>&lt;id&gt;</th>
<th>b</th>
<th>&lt;id&gt;</th>
<th>c</th>
</tr>
</thead>
</table>

**maxmag**

Returns \( x \) if \( |x| > |y| \), \( y \) if \( |y| > |x| \), otherwise \( \text{fmax}(x, y) \).

*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 | extended instructions set <id> | 43 | <id> | x | <id> | y |
|---|---|---|---|---|---|---|---|---|---|---|---|
**minmag**

Returns $x$ if $|x| < |y|$, $y$ if $|y| < |x|$, otherwise $f\text{min}(x, y)$.

*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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>$&lt;id&gt;$</th>
<th>Result Type</th>
<th>extended instructions set $&lt;id&gt;$</th>
<th>44</th>
<th>$&lt;id&gt;$</th>
<th>$x$</th>
<th>$&lt;id&gt;$</th>
<th>$y$</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>$&lt;id&gt;$</th>
<th>Result Type</th>
<th>extended instructions set $&lt;id&gt;$</th>
<th>45</th>
<th>$&lt;id&gt;$</th>
<th>$x$</th>
<th>$&lt;id&gt;$</th>
<th>$iptr$</th>
</tr>
</thead>
</table>

**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 primitive data type size of $nancode$ and *Result Type* must be equal.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>$&lt;id&gt;$</th>
<th>Result Type</th>
<th>extended instructions set $&lt;id&gt;$</th>
<th>46</th>
<th>$&lt;id&gt;$</th>
<th>$nancode$</th>
</tr>
</thead>
</table>
### 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 | Result Type | Result Type | extended instructions set <id> | 47 | <id> | <id> \\
   |     | \( x \)     | \( y \)     |                        |    |      |      \\
```

### 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 | Result Type | Result Type | extended instructions set <id> | 48 | <id> | <id> \\
   |     | \( x \)     | \( y \)     |                        |    |      |      \\
```

### 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 | Result Type | Result Type | extended instructions set <id> | 49 | <id> | <id> \\
   |     | \( x \)     | \( y \)     |                        |    |      |      \\
```

### 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 | Result Type | Result Type | extended instructions set <id> | 50 | <id> | <id> \\
   |     | \( x \)     | \( y \)     |                        |    |      |      \\
```
remainder

Compute the value r such that $r = x - \lfloor x/y \rfloor$, 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>51</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

remquo

The **remquo** instruction computes the value $r$ such that $r = x - \lfloor k y \rfloor$, 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>52</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td>quo</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>53</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>(\text{Result Type})</th>
<th>(\text{Result }&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>54</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
<th>(&lt;id&gt;)</th>
<th>(y)</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>(\text{Result Type})</th>
<th>(\text{Result }&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>55</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>(\text{Result Type})</th>
<th>(\text{Result }&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>56</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

**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 | \(<id>\) | \(\text{Result Type}\) | \(\text{Result }<id>\) | 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>58</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>59</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sinpi

Compute \( sin(\pi x) \) 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>60</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>61</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>62</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>63</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**tanpi**

Compute \( \tan (\pi \times x) \) 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>64</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>65</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>66</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

### half_cos

Compute cosine 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>67</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>69</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>70</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>71</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>72</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
**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.

```
<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>73</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
```

**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.

```
<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>74</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
```

**half_powr**

Compute \( x \) to the power \( y \), where \( x \) is \( \geq 0 \).

*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.

```
<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>75</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
```

**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.

```
<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>76</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>77</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>78</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>79</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>80</th>
<th>&lt;id&gt;</th>
<th>x</th>
</tr>
</thead>
</table>
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>84</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>85</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

### 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 | \(<id>\) | 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th><em>Result Type</em></th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>87</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th><em>Result Type</em></th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>88</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
</tr>
</thead>
</table>

**native_powr**

Compute \( x \) to the power \( y \), where \( x \) is \( \geq 0 \).

*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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th><em>Result Type</em></th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>89</th>
<th>(&lt;id&gt;)</th>
<th>(x)</th>
<th>(&lt;id&gt;)</th>
<th>(y)</th>
</tr>
</thead>
</table>

31
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>90</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( x )</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>91</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( x )</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>92</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( x )</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>93</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td>extended instructions set &lt;id&gt;</td>
<td></td>
<td>$x$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>94</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result &lt;id&gt;</td>
<td>extended instructions set &lt;id&gt;</td>
<td></td>
<td>$x$</td>
</tr>
</tbody>
</table>
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.

\textbf{s_abs}

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

\textit{Result Type} and $x$ must be integer or vector(2,3,4,8,16) of integer values.

All of the operands, including the \textit{Result Type} operand, must be of the same type.

This instruction can be decorated with \textbf{NoSignedWrap}.

\begin{tabular}{|ccc|c|}
\hline
6 & 12 & \textit{Result Type} & \textit{extended instructions set <id>} & 141 \\
\hline
\end{tabular}

\textbf{s_abs_diff}

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

\textit{Result Type}, $x$ and $y$ must be integer or vector(2,3,4,8,16) of integer values.

All of the operands, including the \textit{Result Type} operand, must be of the same type.

\begin{tabular}{|ccc|c|}
\hline
7 & 12 & \textit{Result Type} & \textit{extended instructions set <id>} & 142 \\
\hline
\end{tabular}

\textbf{s_add_sat}

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

\textit{Result Type}, $x$ and $y$ must be integer or vector(2,3,4,8,16) of integer values.

All of the operands, including the \textit{Result Type} operand, must be of the same type.

\begin{tabular}{|ccc|c|}
\hline
7 & 12 & \textit{Result Type} & \textit{extended instructions set <id>} & 143 \\
\hline
\end{tabular}
**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>144</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

**s_hadd**

Returns the value of $(x + y) \gg 1$, 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>145</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

**u_hadd**

Returns the value of $(x + y) \gg 1$, 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>146</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

**s_rhadd**

Returns the value of $(x + y + 1) \gg 1$, 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 \((x + y + 1) \gg 1\), 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>148</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

**s_clamp**

Returns \(s_{\text{min}}(s_{\text{max}}(x,\text{minval}),\text{maxval})\), where \(x\), \(\text{minval}\), and \(\text{maxval}\) are treated as signed integers. Results are undefined if \(\text{minval} > \text{maxval}\).

*Result Type*, \(x\), \(\text{minval}\) and \(\text{maxval}\) 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>149</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; minval</th>
<th>&lt;id&gt; maxval</th>
</tr>
</thead>
</table>

**u_clamp**

Returns \(u_{\text{min}}(u_{\text{max}}(x,\text{minval}),\text{maxval})\), where \(x\), \(\text{minval}\), and \(\text{maxval}\) are treated as unsigned integers. Results are undefined if \(\text{minval} > \text{maxval}\).

*Result Type*, \(x\), \(\text{minval}\) and \(\text{maxval}\) 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>150</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; minval</th>
<th>&lt;id&gt; maxval</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>151</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>152</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set &lt;id&gt;</td>
<td>152</td>
<td>&lt;id&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**s_mad_hi**

Returns \(\text{mul_hi}(a, b) + c\), 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>153</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set &lt;id&gt;</td>
<td>153</td>
<td>&lt;id&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**u_mad_sat**

Returns \(x \times y + z\) 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>154</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set &lt;id&gt;</td>
<td>154</td>
<td>&lt;id&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**s_mad_sat**

Returns \(x \times y + z\) 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>155</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td>extended instructions set &lt;id&gt;</td>
<td>155</td>
<td>&lt;id&gt;</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>156</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td>$y$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>157</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td>$y$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>158</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td>$y$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>159</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td>$y$</td>
</tr>
</tbody>
</table>
### s_mul_hi

Computes $x \times y$ 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>160</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>161</th>
<th>&lt;id&gt; v</th>
<th>&lt;id&gt; i</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>162</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

### 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:

\[ \text{Result} = ((\text{upcast...to i16}) \text{hi} << 8) | \text{lo} \]

When hi and lo component type is i16:

\[ \text{Result} = ((\text{upcast...to i32}) \text{hi} << 8) | \text{lo} \]

When hi and lo component type is i32:

\[ \text{Result} = ((\text{upcast...to i64}) \text{hi} << 8) | \text{lo} \]

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.

<table>
<thead>
<tr>
<th>?</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>164</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
</table>
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 type is 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 | 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 | 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>167</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
<th>&lt;id&gt; z</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>168</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
<th>&lt;id&gt; z</th>
</tr>
</thead>
</table>

### 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>169</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>
\textbf{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. \texttt{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.

\textit{Result Type}, \(x\) and \(y\) must be \(i32\) or \texttt{vector(2,3,4,8,16)} of \(i32\) values.

All of the operands, including the \textit{Result Type} operand, must be of the same type.

\begin{tabular}{|l|l|l|l|l|}
\hline
\texttt{u abs} & \textit{Result Type} & \texttt{Result <id>} & extended instructions set \(<id>\) & \texttt{170} \\
\hline
\texttt{u abs diff} & \textit{Result Type} & \texttt{Result <id>} & extended instructions set \(<id>\) & \texttt{201} \\
\hline
\texttt{u mul hi} & \textit{Result Type} & \texttt{Result <id>} & extended instructions set \(<id>\) & \texttt{203} \\
\hline
\end{tabular}
u_mad_hi

Returns $\text{mul_hi}(a, b) + c$, 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>204</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Result Type</td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>95</th>
<th>&lt;id&gt;</th>
<th>x</th>
<th>&lt;id&gt;</th>
<th>minval</th>
<th>&lt;id&gt;</th>
<th>maxval</th>
</tr>
</thead>
</table>

**degrees**

Converts radians to degrees, i.e. $(180 / \pi) \times$ 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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>96</th>
<th>&lt;id&gt;</th>
<th>radians</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>97</th>
<th>&lt;id&gt;</th>
<th>x</th>
<th>&lt;id&gt;</th>
<th>y</th>
</tr>
</thead>
</table>


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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>98</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
</tr>
</thead>
</table>

mix

Returns the linear blend of \( x \) & \( y \) implemented as:

\[ x + (y - x) \times a \]

*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**.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>99</th>
<th>&lt;id&gt; x</th>
<th>&lt;id&gt; y</th>
<th>&lt;id&gt; a</th>
</tr>
</thead>
</table>

radians

Converts degrees to radians, i.e. \((\pi / 180) \times \text{degrees}\).

*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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>100</th>
<th>&lt;id&gt; degrees</th>
</tr>
</thead>
</table>

step

Returns 0.0 if \( x < \text{edge} \), otherwise it returns 1.0.

*Result Type*, \( \text{edge} \) 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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>101</th>
<th>&lt;id&gt; edge</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
**smoothstep**

Returns 0.0 if \( x \leq edge_0 \) and 1.0 if \( x \geq edge_1 \) 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 \( edge_0 \geq edge_1 \) 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*.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>102</th>
<th>&lt;id&gt; edge_0</th>
<th>&lt;id&gt; edge_1</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt; extended instructions set &lt;id&gt;</th>
<th>103</th>
<th>&lt;id&gt; x</th>
</tr>
</thead>
</table>
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 respectively, 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 | $<id>$ | extended instructions set $<id>$ | 104 | $<id>$ | $p_0$ | $p_1$ |
|---|----|--------|-------------|-------|---------------------------------|-----|-------|-------|

### 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 | $<id>$ | extended instructions set $<id>$ | 105 | $<id>$ | $p_0$ | $p_1$ |
|---|----|--------|-------------|-------|---------------------------------|-----|-------|-------|

### 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.*

<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>$&lt;id&gt;$</th>
<th>Result Type</th>
<th>$&lt;id&gt;$</th>
<th>extended instructions set $&lt;id&gt;$</th>
<th>106</th>
<th>$&lt;id&gt;$</th>
<th>$p$</th>
</tr>
</thead>
</table>
**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.

```
<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>107</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```  

**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.

```
<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>108</th>
<th>&lt;id&gt;</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```  

**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.

```
<table>
<thead>
<tr>
<th>6</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>109</th>
<th>&lt;id&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
fast_normalize

Returns a vector in the same direction as \( p \) but with a length of 1 computed as:

\[
p * \text{half_rsqrt}(p.x^2 + p.y^2 \ldots)
\]

The result shall be within 8192 ulps error from the infinitely precise result of:

\[
\text{if (all}(p == 0.0f)) \text{ { result = } p; } \\
\text{else } \text{ { result = } p / \text{sqrt}(p.x^2 + p.y^2 + \ldots); } \\
\]

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 "denoms are flushed to zero" mode, individual operand elements with magnitude less than \( \text{sqrt}(\text{FLT_MIN}) \) may be flushed to zero before proceeding with the calculation.

Result Type and \( p \) must be floating-point or \( \text{vector}(2,3,4) \) of floating-point values.

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

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>186</th>
<th>(&lt;id&gt;)</th>
<th>(&lt;id&gt;)</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>(&lt;id&gt;)</td>
<td>Result Type</td>
<td>(&lt;id&gt;)</td>
<td>extended instructions set (&lt;id&gt;)</td>
<td>186</td>
<td>(&lt;id&gt;)</td>
<td>(&lt;id&gt;)</td>
<td>(&lt;id&gt;)</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>(&lt;id&gt;)</th>
<th>Result Type</th>
<th>(&lt;id&gt;)</th>
<th>extended instructions set (&lt;id&gt;)</th>
<th>187</th>
<th>(&lt;id&gt;)</th>
<th>(&lt;id&gt;)</th>
<th>(&lt;id&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>(&lt;id&gt;)</td>
<td>Result Type</td>
<td>(&lt;id&gt;)</td>
<td>extended instructions set (&lt;id&gt;)</td>
<td>187</td>
<td>(&lt;id&gt;)</td>
<td>(&lt;id&gt;)</td>
<td>(&lt;id&gt;)</td>
</tr>
</tbody>
</table>
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 \((p + (\text{offset} \times n))\) 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.

\( \text{offset} \) must be \text{size_t}.

\( p \) must be a \text{pointer(global, local, private, constant, generic)} to floating-point, integer.

**Result Type** must be \text{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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>171</th>
<th>&lt;id&gt; offset</th>
<th>&lt;id&gt; p</th>
<th>Literal n</th>
</tr>
</thead>
</table>

**vstoren**

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

\( \text{offset} \) must be \text{size_t}.

**Result Type** must be \text{void}.

\( p \) must be a \text{pointer(global, local, private, generic)} to floating-point, integer.

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

\text{data} component type must be equal to the type pointed by \( p \).

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>extended instructions set &lt;id&gt;</th>
<th>172</th>
<th>&lt;id&gt; data</th>
<th>&lt;id&gt; offset</th>
<th>&lt;id&gt; p</th>
</tr>
</thead>
</table>

52
**vload_half**

Reads a half value from the address computed as \((p + (\text{offset}))\) 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*.

**vload_halfn**

Reads *n* half components from the address \((p + (\text{offset} * n))\), 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.

**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 \((p + \text{offset})\).

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*.
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.

vstore_halffn

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.
**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.

---

**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.
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 $(p + (\text{offset} \times n))$. The computed address must be aligned to $(\text{sizeof(half)} \times n)$ bytes.

For $n$ equal to 3, the converted vector of half values is written to the address computed as $(p + (\text{offset} \times 4))$. The computed address must be aligned to $(\text{sizeof(half)} \times 4)$ 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result Type</th>
<th>extended instructions set</th>
<th>180</th>
<th>data</th>
<th>offset</th>
<th>p</th>
</tr>
</thead>
</table>

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 $(p + (\text{offset} \times n))$. The computed address must be aligned to $(\text{sizeof(half)} \times n)$ bytes.

For $n$ equal to 3, the converted vector of half values is written to the address computed as $(p + (\text{offset} \times 4))$. The computed address must be aligned to $(\text{sizeof(half)} \times 4)$ 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 | Result Type | Result Type | extended instructions set | 181 | data | offset | p | FP Rounding Mode mode |
2.7 Miscellaneous Vector instructions

This section describes additional vector instructions.

**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 \( \text{ilogb}(2 \cdot m - 1) \) 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.
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 \( \text{ilogb}(2^m - 1) + 1 \) least significant bits of each mask component are considered, where \( m \) is equal to the component count of \( x \) and \( y \).

\( \text{shuffle mask} \) 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 \( \text{shuffle mask} \) must match the size of each component in Result Type.

Result Type must have the same component type as \( x \) and component count as \( \text{shuffle mask} \).

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

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

<table>
<thead>
<tr>
<th>6 + variable</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>184</th>
<th>&lt;id&gt;</th>
<th>format</th>
<th>&lt;id&gt;, &lt;id&gt;, ...</th>
<th>additional arguments</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>extended instructions set &lt;id&gt;</th>
<th>185</th>
<th>&lt;id&gt;</th>
<th>ptr</th>
<th>&lt;id&gt;</th>
<th>num_elements</th>
</tr>
</thead>
</table>

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 vload/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 \texttt{s_upsample}