SPIR-V Extended Instructions for GLSL

John Kessenich, LunarG

Version 1.00, Revision 3 in progress

February 4, 2016

Contents

1 Introduction

This specifies the GLSL.std.450 extended instruction set. It provides instructions for the GLSL built-in functions that do not directly map to native SPIR-V instructions.

Import this extended instruction set using an **OpExtInstImport** "GLSL.std.450" instruction.

2 Binary Form

Documentation form for each extended instruction:

Extended Instruction Name Instruction description. *Result Type* will describe the *Result Type* for the **OpExtInst** instruction. *Number* is the extended instruction number to use in the **OpExtInst** instruction. *Operand 1*, *Operand 2*,... are the operands listed for the **OpExtInst** instruction.			
Number	*Operand 1*	*Operand 2*	...

Extended instructions:

Round Result is the value equal to the nearest whole number to *x*. The fraction 0.5 will round in a direction chosen by the implementation, presumably the direction that is fastest. This includes the possibility that **Round** *x* is the same value as **RoundEven** *x* for all values of *x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
1	*<id>* *x*

RoundEven Result is the value equal to the nearest whole number to *x*. A fractional part of 0.5 will round toward the nearest even whole number. (Both 3.5 and 4.5 for *x* will be 4.0.) The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
2	*<id>* *x*

Trunc Result is the value equal to the nearest whole number to *x* whose absolute value is not larger than the absolute value of *x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
3	*<id>* *x*

FAbs Result is *x* if $x \geq 0$; otherwise result is -*x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
4	*<id>* *x*

SAbs Result is *x* if $x \geq 0$; otherwise result is -*x*, where *x* is interpreted as a signed integer. *Result Type* and the type of *x* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.	
5	*<id>* *x*

FSign Result is 1.0 if $x > 0$, 0.0 if $x = 0$, or -1.0 if $x < 0$. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
6	*<id>* *x*

SSign Result is 1 if $x > 0$, 0 if $x = 0$, or -1 if $x < 0$, where *x* is interpreted as a signed integer. *Result Type* and the type of *x* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.	
7	*<id>* *x*

Floor Result is the value equal to the nearest whole number that is less than or equal to *x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
8	*<id>* *x*

Ceil Result is the value equal to the nearest whole number that is greater than or equal to *x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
9	*<id>* *x*

Fract Result is *x* - **floor** *x*. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
10	*<id>* *x*

Radians Converts *degrees* to radians, i.e., *degrees* * π / 180. The operand *degrees* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *degrees* must be the same type. Results are computed per component.	
11	*<id>* *degrees*

Degrees Converts *radians* to degrees, i.e., *radians* * 180 / π. The operand *radians* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *radians* must be the same type. Results are computed per component.	
12	*<id>* *radians*

Sin The standard trigonometric sine of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
13	*<id>* *x*

Cos The standard trigonometric cosine of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
14	*<id>* *x*

Tan The standard trigonometric tangent of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
15	*<id>* *x*

Asin Arc sine. Result is an angle, in radians, whose sine is *x*. The range of result values is $[-\pi / 2, \pi / 2]$. Result is undefined if **abs** $x > 1$. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
16	*<id>* *x*

Acos Arc cosine. Result is an angle, in radians, whose cosine is *x*. The range of result values is $[0, \pi]$. Result is undefined if **abs** $x > 1$. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.

17	*<id>* *x*

Atan Arc tangent. Result is an angle, in radians, whose tangent is *y_over_x*. The range of result values is [-π, π]. The operand *y_over_x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *y_over_x* must be the same type. Results are computed per component.	
18	*<id>* *y_over_x*

Sinh Hyperbolic sine of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
19	*<id>* *x*

Cosh Hyperbolic cosine of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
20	*<id>* *x*

Tanh Hyperbolic tangent of *x* radians. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
21	*<id>* *x*

Asinh Arc hyperbolic sine; result is the inverse of **sinh**. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.

22	<*id*> *x*

Acosh Arc hyperbolic cosine; Result is the non-negative inverse of **cosh**. Result is undefined if $x < 1$. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
23	<*id*> *x*

Atanh Arc hyperbolic tangent; result is the inverse of tanh. Result is undefined if **abs** $x \geq 1$. The operand *x* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of *x* must be the same type. Results are computed per component.	
24	<*id*> *x*

Atan2 Arc tangent. Result is an angle, in radians, whose tangent is y / x. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of result values is $[-\pi, \pi]$. Result is undefined if *x* and *y* are both 0. The operand *x* and *y* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
25	<*id*> *y*	<*id*> *x*

Pow Result is *x* raised to the *y* power; x^y. Result is undefined if $x < 0$. Result is undefined if $x = 0$ and $y \leq 0$. The operand *x* and *y* must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
26	<*id*> *x*	<*id*> *y*

Exp Result is the natural exponentiation of x; e^x. The operand x must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
27	<*id*> *x*

Log Result is the natural logarithm of x, i.e., the value y which satisfies the equation $x = e^y$. Result is undefined if $x \leq 0$. The operand x must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
28	<*id*> *x*

Exp2 Result is 2 raised to the x power; 2^x. The operand x must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
29	<*id*> *x*

Log2 Result is the base-2 logarithm of x, i.e., the value y which satisfies the equation $x = 2^y$. Result is undefined if $x \leq 0$. The operand x must be a scalar or vector whose component type is 32-bit floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
30	<*id*> *x*

Sqrt Result is the square root of x. Result is undefined if $x < 0$. The operand x must be a scalar or vector whose component type is floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
31	<*id*> *x*

InverseSqrt Result is the reciprical of **sqrt** x. Result is undefined if $x \leq 0$. The operand x must be a scalar or vector whose component type is floating-point. *Result Type* and the type of x must be the same type. Results are computed per component.	
32	*<id>* *x*

Determinant Result is the determinant of x. The operand x must be a square matrix. *Result Type* must be the same type as the component type in the columns of x.	
33	*<id>* *x*

MatrixInverse Result is a matrix that is the inverse of x. The values in the result are undefined if x is singular or poorly conditioned (nearly singular). The operand x must be a square matrix. *Result Type* and the type of x must be the same type.	
34	*<id>* *x*

Modf Result is the fractional part of x and stores through i the whole number part (as a whole-number floating-point value). Both the result and the output parameter will have the same sign as x. The operand x must be a scalar or vector whose component type is floating-point. The operand i must have a pointer type. *Result Type*, the type of x, and the type i points to must all be the same type Results are computed per component.		
35	*<id>* *x*	*<id>* *i*

ModfStruct Same semantics as in **Modf**, except that the entire result is in the instruction's result; there is not a pointer operand to write through. *Result Type* must be an **OpTypeStruct** with two members. Member 0 holds the fractional part. Member 1 holds the whole number part. These two members, and the *Result Type* must all have the same type. This structure type must be explicitly declared by the module.	
36	*<id>* *x*

FMin Result is y if $y < x$; otherwise result is x. Which operand is the result is undefined if one of the operands is a NaN. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
37	*<id>* *x*	*<id>* *y*

NMin Result is y if $y < x$; otherwise result is x. If one operand is a NaN, the other operand is the result. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
38	*<id>* *x*	*<id>* *y*

UMin Result is y if $y < x$; otherwise result is x, where x and y are interpreted as unsigned integers. *Result Type* and the type of x and y must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.		
39	*<id>* *x*	*<id>* *y*

SMin Result is y if $y < x$; otherwise result is x, where x and y are interpreted as signed integers. *Result Type* and the type of x and y must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.		
40	*<id>* *x*	*<id>* *y*

FMax Result is *y* if $x < y$; otherwise result is *x*. Which operand is the result is undefined if one of the operands is a NaN. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
41	*<id>* *x*	*<id>* *y*

NMax Result is *y* if $x < y$; otherwise result is *x*. If one operand is a NaN, the other operand is the result. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
42	*<id>* *x*	*<id>* *y*

UMax Result is *y* if $x < y$; otherwise result is *x*, where *x* and *y* are interpreted as unsigned integers. *Result Type* and the type of *x* and *y* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.		
43	*<id>* *x*	*<id>* *y*

SMax Result is *y* if $x < y$; otherwise result is *x*, where *x* and *y* are interpreted as signed integers. *Result Type* and the type of *x* and *y* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.		
44	*<id>* *x*	*<id>* *y*

FClamp Result is min(max(*x*, *minVal*), *maxVal*). Result is undefined if $minVal > maxVal$.The semantics used by min() and max() are those of FMin and FMax. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.			
45	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

NClamp Result is min(max(*x*, *minVal*), *maxVal*). Result is undefined if *minVal* > *maxVal*.The semantics used by min() and max() are those of NMin and NMax. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.			
46	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

UClamp Result is min(max(*x*, *minVal*), *maxVal*), where *x*, *minVal* and *maxVal* are interpreted as unsigned integers. Result is undefined if *minVal* > *maxVal*. *Result Type* and the type of the operands must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.			
47	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

SClamp Result is min(max(*x*, *minVal*), *maxVal*), where *x*, *minVal* and *maxVal* are interpreted as signed integers. Result is undefined if *minVal* > *maxVal*. *Result Type* and the type of the operands must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.			
48	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

FMix Result is the linear blend of *x* and *y*, i.e., *x* * (1 - *a*) + *y* * *a*. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.			
49	*<id>* *x*	*<id>* *y*	*<id>* *a*

Step Result is 0.0 if $x < edge$; otherwise result is 1.0. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.		
51	*<id>* *edge*	*<id>* *x*

SmoothStep Result is 0.0 if $x \leq edge0$ and 1.0 if $x \geq edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is equivalent to: $t * t * (3 - 2 * t)$, where $t = \text{clamp}((x - edge0) / (edge1 - edge0), 0, 1)$ Result is undefined if $edge0 \geq edge1$. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.			
52	*<id>* *edge0*	*<id>* *edge1*	*<id>* *x*

Fma Computes $a * b + c$. In uses where the result is eventually consumed by a variable decorated as **precise**: - **fma** is considered a single operation, whereas the expression $a * b + c$ consumed by a variable declared **precise** is considered two operations. - The precision of **fma** can differ from the precision of the expression $a * b + c$. - **fma** will be computed with the same precision as any other **fma** consumed by a precise variable, giving invariant results for the same input values of *a*, *b*, and *c*. Otherwise, in the absence of precise consumption, there are no special constraints on the number of operations or difference in precision between **fma** and the expression $a * b +c$. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type. Results are computed per component.			
53	*<id>* *a*	*<id>* *b*	*<id>* *c*

<table>
<tr><td colspan="3">Frexp

Splits x into a floating-point significand in the range [0.5, 1.0) and an integral exponent of two, such that:

x = significand * $2^{exponent}$

The significand is the result and the exponent is returned through the pointer-parameter exp. For a floating-point value of zero, the significand and exponent are both zero. For a floating-point value that is an infinity or is not a number, the result is undefined.

If an implementation supports negative 0, Frexp -0 should result in -0; otherwise it will result in 0.

The operand x must be a scalar or vector whose component type is floating-point.

The exp operand must be a pointer to a scalar or vector with integer component type, with 32-bit component width. The number of components in x and what exp points to must be the same.

Result Type must be the same type as the type of x. Results are computed per component.</td></tr>
<tr><td>54</td><td><<i>id</i>>
<i>x</i></td><td><<i>id</i>>
<i>exp</i></td></tr>
</table>

<table>
<tr><td colspan="2">FrexpStruct

Same semantics as in Frexp, except that the entire result is in the instruction's result; there is not a pointer operand to write through.

Result Type must be an OpTypeStruct with two members. Member 0 must have the same type as the type of x. Member 0 holds the significand. Member 1 must be a scalar or vector with integer component type, with 32-bit component width. Member 1 holds exponent. These two members must have the same number of components. This structure type must be explicitly declared by the module.</td></tr>
<tr><td>55</td><td><<i>id</i>>
<i>x</i></td></tr>
</table>

<table>
<tr><td>Ldexp

Builds a floating-point number from x and the corresponding integral exponent of two in exp:

significand * $2^{exponent}$

If this product is too large to be represented in the floating-point type, the result is undefined. If exp is greater than +128 (single-precision) or +1024 (double-precision), the result undefined. If exp is less than -126 (single-precision) or -1022 (doubleprecision), the result may be flushed to zero. Additionally, splitting the value into a significand and exponent using frexp and then reconstructing a floating-point value using ldexp should yield the original input for zero and all finite non-denormized values.

The operand x must be a scalar or vector whose component type is floating-point.

The exp operand must be a scalar or vector with integer component type. The number of components in x and exp must be the same.

Result Type must be the same type as the type of x. Results are computed per component.</td></tr>
</table>

56	<*id*> *x*	<*id*> *exp*

PackSnorm4x8

First, converts each component of the normalized floating-point value *v* into 8-bit integer values. These are then packed into the result.

The conversion for component *c* of *v* to fixed point is done as follows:

round(clamp(*c*, -1, +1) * 127.0)

The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits.

The *v* operand must be a vector of 4 components whose type is a 32-bit floating-point.

Result Type must be a 32-bit integer type.

57	<*id*> *v*

PackUnorm4x8

First, converts each component of the normalized floating-point value *v* into 8-bit integer values. These are then packed into the result.

The conversion for component *c* of *v* to fixed point is done as follows:

round(clamp(*c*, 0, +1) * 255.0)

The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits.

The *v* operand must be a vector of 4 components whose type is a 32-bit floating-point.

Result Type must be a 32-bit integer type.

58	<*id*> *v*

PackSnorm2x16 First, converts each component of the normalized floating-point value *v* into 16-bit integer values. These are then packed into the result. The conversion for component *c* of *v* to fixed point is done as follows: round(clamp(*c*, -1, +1) * 32767.0) The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits. The *v* operand must be a vector of 2 components whose type is a 32-bit floating-point. *Result Type* must be a 32-bit integer type.	
59	*<id>* *v*

PackUnorm2x16 First, converts each component of the normalized floating-point value *v* into 16-bit integer values. These are then packed into the result. The conversion for component *c* of *v* to fixed point is done as follows: round(clamp(*c*, 0, +1) * 65535.0) The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits. The *v* operand must be a vector of 2 components whose type is a 32-bit floating-point. *Result Type* must be a 32-bit integer type.	
60	*<id>* *v*

PackHalf2x16 Result is the unsigned integer obtained by converting the components of a two-component floating-point vector to the 16-bit **OpTypeFloat**, and then packing these two 16-bit integers into a 32-bit unsigned integer. The first vector component specifies the 16 least-significant bits of the result; the second component specifies the 16 most-significant bits. The *v* operand must be a vector of 2 components whose type is a 32-bit floating-point. *Result Type* must be a 32-bit integer type.	
61	*<id>* *v*

PackDouble2x32 Result is the double-precision value obtained by packing the components of *v* into a 64-bit value. If an IEEE 754 Inf or NaN is created, it will not signal, and the resulting floating-point value is unspecified. Otherwise, the bit-level representation of *v* is preserved. The first vector component specifies the 32 least significant bits; the second component specifies the 32 most significant bits. The *v* operand must be a vector of 2 components whose type is a 32-bit integer. *Result Type* must be a 64-bit floating-point scalar.	
62	*<id>* *v*

UnpackSnorm2x16 First, unpacks a single 32-bit unsigned integer *p* into a pair of 16-bit signed integers. Then, each component is converted to a normalized floating-point value to generate the result. The conversion for unpacked fixed-point value *f* to floating point is done as follows: clamp(*f* / 32767.0, -1, +1) The first component of the result will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits. The *p* operand must be a scalar with 32-bit integer type. *Result Type* must be a vector of 2 components whose type is 32-bit floating point.	
63	*<id>* *p*

UnpackUnorm2x16 First, unpacks a single 32-bit unsigned integer *p* into a pair of 16-bit unsigned integers. Then, each component is converted to a normalized floating-point value to generate the result. The conversion for unpacked fixed-point value *f* to floating point is done as follows: *f* / 65535.0 The first component of the result will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits. The *p* operand must be a scalar with 32-bit integer type. *Result Type* must be a vector of 2 components whose type is 32-bit floating point.	
64	*<id>* *p*

UnpackHalf2x16 Result is the two-component floating-point vector with components obtained by unpacking a 32-bit unsigned integer into a pair of 16-bit values, interpreting those values as 16-bit floating-point numbers according to the OpenGL Specification, and converting them to 32-bit floating-point values. The first component of the vector is obtained from the 16 least-significant bits of *v*; the second component is obtained from the 16 most-significant bits of *v*. The *v* operand must be a scalar with 32-bit integer type. *Result Type* must be a vector of 2 components whose type is 32-bit floating point.	
65	*<id>* *v*

UnpackSnorm4x8 First, unpacks a single 32-bit unsigned integer *p* into four 8-bit signed integers. Then, each component is converted to a normalized floating-point value to generate the result. The conversion for unpacked fixed-point value *f* to floating point is done as follows: clamp(*f* / 127.0, -1, +1) The first component of the result will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits. The *p* operand must be a scalar with 32-bit integer type. *Result Type* must be a vector of 4 components whose type is 32-bit floating point.	
66	*<id>* *p*

UnpackUnorm4x8 First, unpacks a single 32-bit unsigned integer *p* into four 8-bit unsigned integers. Then, each component is converted to a normalized floating-point value to generate the result. The conversion for unpacked fixed-point value *f* to floating point is done as follows: *f* / 255.0 The first component of the result will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits. The *p* operand must be a scalar with 32-bit integer type. *Result Type* must be a vector of 4 components whose type is 32-bit floating point.	
67	*<id>* *p*

UnpackDouble2x32 Result is the two-component unsigned integer vector representation of *v*. The bit-level representation of *v* is preserved. The first component of the vector contains the 32 least significant bits of the double; the second component consists of the 32 most significant bits. The *v* operand must be a scalar whose type is 64-bit floating point. *Result Type* must be a vector of 2 components whose type is a 32-bit integer.	
68	*<id>* *v*

Length Result is the length of vector *x*, i.e., sqrt(*x* [0] ^{2} + *x* [1] ^{2} + . . .). The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* must be a scalar of the same type as the component type of *x*.	
69	*<id>* *x*

Distance Result is the distance between *p0* and *p1*, i.e., length(*p0* - *p1*). The operands must all be a scalar or vector whose component type is floating-point. *Result Type* must be a scalar of the same type as the component type of the operands.		
70	*<id>* *p0*	*<id>* *p1*

Cross Result is the cross product of *x* and *y*, i.e., the resulting components are, in order: *x*[1] * *y*[2] - *y*[1] * *x*[2] *x*[2] * *y*[0] - *y*[2] * *x*[0] *x*[0] * *y*[1] - *y*[0] * *x*[1] All the operands must be vectors of 3 components of a floating-point type. *Result Type* and the type of all operands must be the same type.		
71	*<id>* *x*	*<id>* *y*

Normalize Result is the vector in the same direction as *x* but with a length of 1. The operand *x* must be a scalar or vector whose component type is floating-point. *Result Type* and the type of *x* must be the same type.	
72	*<id>* *x*

FaceForward If the dot product of *Nref* and *I* is negative, the result is *N*, otherwise it is *-N*. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type.			
73	*<id>* *N*	*<id>* *I*	*<id>* *Nref*

Reflect For the incident vector *I* and surface orientation *N*, the result is the reflection direction: *I* - 2 * dot(*N*, *I*) * *N* *N* must already be normalized in order to achieve the desired result. The operands must all be a scalar or vector whose component type is floating-point. *Result Type* and the type of all operands must be the same type.		
74	*<id>* *I*	*<id>* *N*

Refract

For the incident vector *I* and surface normal *N*, and the ratio of indices of refraction *eta*, the result is the refraction vector. The result is computed by

k = 1.0 - *eta* * *eta* * (1.0 - dot(*N*, *I*) * dot(*N*, *I*))

if k < 0.0 the result is 0.0

otherwise, the result is *eta* * *I* - (*eta* * dot(*N*, *I*) + sqrt(k)) * *N*

The input parameters for the incident vector *I* and the surface normal *N* must already be normalized to get the desired results.

The type of *I* and *N* must be a scalar or vector with a floating-point component type.

The type of *eta* must be a 32-bit floating-point scalar.

Result Type, the type of *I*, and the type of *N* must all be the same type.

75	*<id>* *I*	*<id>* *N*	*<id>* *eta*

FindILsb

Integer least-significant bit.

Results in the bit number of the least-significant 1-bit in the binary representation of *Value*. If *Value* is 0, the result is -1.

Result Type and the type of *Value* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.

76	*<id>* *Value*

FindSMsb

Signed-integer most-significant bit, with *Value* interpreted as a signed integer.

For positive numbers, the result will be the bit number of the most significant 1-bit. For negative numbers, the result will be the bit number of the most significant 0-bit. For a *Value* of 0 or -1, the result is -1.

Result Type and the type of *Value* must both be integer scalar or integer vector types. *Result Type* and operand types must have the same number of components with the same component width. Results are computed per component.

This instruction is currently limited to 32-bit width components.

77	*<id>* *Value*

<table>
<tr><td colspan="2">FindUMsb

Unsigned-integer most-significant bit.

Results in the bit number of the most-significant 1-bit in the binary representation of Value. If Value is 0, the result is -1.

Result Type and the type of Value must both be integer scalar or integer vector types. Result Type and operand types must have the same number of components with the same component width. Results are computed per component.

This instruction is currently limited to 32-bit width components.</td></tr>
<tr><td>78</td><td><id>
Value</td></tr>
</table>

<table>
<tr><td colspan="2">InterpolateAtCentroid

Result is the value of the input interpolant sampled at a location inside both the pixel and the primitive being processed. The value obtained would be the same value assigned to the input variable if it were decorated as Centroid.

The operand interpolant must be a pointer to the Input Storage Class.

The operand interpolant must be a pointer to a scalar or vector whose component type is 32-bit floating-point.

This instruction is only valid in the Fragment execution model.

Result Type and the type of interpolant must be the same type.</td></tr>
<tr><td>79</td><td><id>
interpolant</td></tr>
</table>

<table>
<tr><td colspan="3">InterpolateAtSample

Result is the value of the input interpolant variable at the location of sample number sample. If multisample buffers are not available, the input variable will be evaluated at the center of the pixel. If sample sample does not exist, the position used to interpolate the input variable is undefined.

The operand interpolant must be a pointer to the Input Storage Class.

The operand interpolant must be a pointer to a scalar or vector whose component type is 32-bit floating-point.

This instruction is only valid in the Fragment execution model.

The sample operand must be a scalar 32-bit integer.

Result Type and the type of interpolant must be the same type.</td></tr>
<tr><td>80</td><td><id>
interpolant</td><td><id>
sample</td></tr>
</table>

<table>
<tr><td colspan="3">InterpolateAtOffset

Result is the value of the input interpolant variable sampled at an offset from the center of the pixel specified by offset. The two floating-point components of offset, give the offset in pixels in the x and y directions, respectively. An offset of (0, 0) identifies the center of the pixel. The range and granularity of offsets supported are implementation-dependent.

The operand interpolant must be a pointer to the Input Storage Class.

The operand interpolant must be a pointer to a scalar or vector whose component type is 32-bit floating-point.

This instruction is only valid in the Fragment execution model.

The offset operand must be a vector of 2 components of 32-bit floating-point type.

Result Type and the type of interpolant must be the same type.</td></tr>
<tr><td>81</td><td><id>
interpolant</td><td><id>
offset</td></tr>
</table>

A Changes

A.1 Changes from Version 0.99, Revision 1

- Fork the revision stream, changes section, etc. from the core specification, so this specification has its own, starting numbering at revision 1. This document now lives independently.
- Added integer versions of abs, sign, min, max, and clamp.
- Removed floatBitsToInt, floatBitsToUint, intBitsToFloat, and uintBitsToFloat; these can be handled with **OpBitcast**.
- Removed fTransform, not needed.
- Fixed internal bugs
  - 13721: Add **OpTypeStruct**-result versions of **Modf** and **Frexp**: **ModfStruct** and **FrexpStruct**.
- Fixed public bugs
  - 1322: GLSL.std.450 frexp wasn't saying the *exp* argument was a pointer to the result

A.2 Changes from Version 0.99, Revision 2

- Moved AddCarry, SubBorrow, and MulExtended type of instructions to the core specification.
- Added integer variant of **Mix**, creating **FMix** and **IMix** (14480).
- Modified spellings to be more regular (14614).

A.3 Changes from Version 0.99, Revision 3

- Add "**N**" version of **Min**, **Max**, and **Clamp**, creating a version that favors non-NaN operands over NaN operands.
- Bug 15452 Remove **IMix**.
- Bug 15300 Be more consistent that the **InterpolateAt** instructions take a pointer.