SPIR-V Extended Instructions for GLSL

John Kessenich, Google

Version 1.00, Revision 12

Table of Contents

Chapter 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.

Chapter 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.

Any **Capability** restrictions.

Number	*Operand 1*	*Operand 2*	…

Extended instructions:

Round

Result is the value equal to the nearest whole number to *x*. The fraction 0.5 rounds 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 rounds toward the nearest even whole number. (Both 3.5 and 4.5 for *x* round to 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*

<table>
<tr><td colspan="2">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.</td></tr>
<tr><td>3</td><td><id>
x</td></tr>
</table>

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

Result is +0.0 if x is ±0.0, x if x > 0.0, and -x if x < 0.0.

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.</td></tr>
<tr><td>4</td><td><id>
x</td></tr>
</table>

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

Result is x if x ¬ 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.

This instruction can be decorated with NoSignedWrap.</td></tr>
<tr><td>5</td><td><id>
x</td></tr>
</table>

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

Result is 1.0 if x > 0, -1.0 if x < 0, +0.0 if x = +0.0, and ±0.0 if x = -0.0. If x = ±NaN, the result can be any of ±1.0 or ±0.0, regardless of whether shader_float_controls is in use.

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.</td></tr>
<tr><td>6</td><td><id>
x</td></tr>
</table>

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* * ¬ / 180. The operand *degrees* must be a scalar or vector whose component type is 16-bit or 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 / ¬.

The operand *radians* must be a scalar or vector whose component type is 16-bit or 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 16-bit or 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 16-bit or 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 16-bit or 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 [-¬ / 2, ¬ / 2]. The resulting value is undefined if **abs** *x* > 1.

The operand *x* must be a scalar or vector whose component type is 16-bit or 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, ¬]. The resulting value is undefined if **abs** *x* > 1.

The operand *x* must be a scalar or vector whose component type is 16-bit or 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 [-¬ / 2, ¬ / 2].

The operand *y_over_x* must be a scalar or vector whose component type is 16-bit or 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 16-bit or 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 16-bit or 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 16-bit or 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 16-bit or 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**. The resulting value is undefined if *x* < 1. The operand *x* must be a scalar or vector whose component type is 16-bit or 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. The resulting value is undefined if **abs** *x* ¬ 1.

The operand *x* must be a scalar or vector whose component type is 16-bit or 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 [-¬, ¬]. The resulting value is undefined if *x* and *y* are both 0.

The operand *x* and *y* must be a scalar or vector whose component type is 16-bit or 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. The resulting value is undefined if *x* < 0. Result is undefined if *x* = 0 and *y* ¬ 0.

The operand *x* and *y* must be a scalar or vector whose component type is 16-bit or 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 16-bit or 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 $x = e^y$. The resulting value is undefined if *x* ¬ 0.

The operand *x* must be a scalar or vector whose component type is 16-bit or 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 16-bit or 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 $x = 2^y$. The resulting value is undefined if *x* ¬ 0.

The operand *x* must be a scalar or vector whose component type is 16-bit or 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*. The resulting value 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 reciprocal of **sqrt** *x*. The resulting value 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.

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 resulting values 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 **Modf** is deprecated, use **ModfStruct** instead. 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 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 and have a floating-point component type. Results are computed per component.		
35	*<id>* *x*	*<id>* *i*

ModfStruct Result is a structure containing both the fractional part of *x* and the whole number part of *x*. *Result Type* must be an **OpTypeStruct** with two members. Member 0 holds the fractional part. Member 1 holds the whole number part. Both members get the same sign as *x*. These two members and *x* must all be the same type. Results are computed per component. The operand *x* must be a scalar or vector whose component type is floating-point.

36	*<id>* *x*

FMin

Result is *y* if *y* < *x*, either *x* or *y* if both *x* and *y* are zeros, otherwise *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*

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.

38	*<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.

39	*<id>* *x*	*<id>* *y*

FMax

Result is *y* if *x* < *y*, either *x* or *y* if both *x* and *y* are zeros, otherwise *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.

40	*<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.		
41	*<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.		
42	*<id>* *x*	*<id>* *y*

FClamp Result is min(max(*x*, *minVal*), *maxVal*). The resulting value 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.			
43	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

UClamp Result is min(max(*x*, *minVal*), *maxVal*), where *x*, *minVal* and *maxVal* are interpreted as unsigned integers. The resulting value 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.			
44	*<id>* *x*	*<id>* *minVal*	*<id>* *maxVal*

SClamp Result is min(max(*x*, *minVal*), *maxVal*), where *x*, *minVal* and *maxVal* are interpreted as signed integers. The resulting value 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.

45	*<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.

46	*<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.

48	*<id>* *edge*	*<id>* *x*

SmoothStep

Result is 0.0 if *x* ¬ *edge0* and 1.0 if *x* ¬ *edge1* and performs smooth Hermite interpolation between 0 and 1 if *edge0* < *x* < *edge1*. This is equivalent to:

t * *t* * (3 - 2 * *t*), where *t* = clamp ((*x* - *edge0*) / (*edge1* - *edge0*), 0, 1)

The resulting value is undefined if *edge0* ¬ *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.

49	*<id>* *edge0*	*<id>* *edge1*	*<id>* *x*

Fma Computes *a* * *b* + *c*. In uses where this operation is decorated with **NoContraction**: - **fma** is considered a single operation, whereas the expression *a* * *b* + *c* is considered two operations. - The precision of **fma** can differ from the precision of the expression *a* * *b* + *c*. - **fma** is computed with the same precision as any other **fma** decorated with **NoContraction**, giving invariant results for the same input values of *a*, *b*, and *c*. Otherwise, in the absence of a **NoContraction** decoration, 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.			
50	*<id>* *a*	*<id>* *b*	*<id>* *c*

Frexp **Frexp** is deprecated, use **FrexpStruct** instead. Splits *x* into a floating-point significand in the range (-1.0, 0.5] or [0.5, 1.0) and an integral exponent of 2, such that: $x = significand * 2^{exponent}$ The *significand* is the instruction result. An *x* of *-0.0* results in a significand *-0.0*, while an *x* of *0.0* results in *0.0*. For a floating-point value that is an infinity or is not a number, the significand is undefined. The operand *x* must be a scalar or vector whose component type is floating-point. The exponent is returned through the pointer-parameter *exp*. 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. If *x* is a zero, the exponent is 0.0. If *x* is an infinity or a NaN, the exponent is undefined. *Result Type* must be the same type as the type of *x*. Results are computed per component.		
51	*<id>* *x*	*<id>* *exp*

FrexpStruct

Result is a structure containing *x* split into a floating-point significand in the range (-1.0, 0.5] or [0.5, 1.0) and an integral exponent of 2, such that:

$x = significand * 2^{exponent}$

If *x* is a zero, the exponent is 0.0. If *x* is an infinity or a NaN, the exponent is undefined. If *x* is *0.0*, the significand is *0.0*. If *x* is *-0.0*, the significand is *-0.0*

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 the exponent. These two members and *x* must have the same number of components.

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

52	*<id>* *x*

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 resulting value is undefined. If *exp* is greater than +128 (single precision) or +1024 (double precision), the resulting value is undefined. If *exp* is less than -126 (single precision) or -1022 (double precision), 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-denormalized 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.

53	*<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 is written to the least significant bits of the output; the last component is 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.	
54	*<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 is written to the least significant bits of the output; the last component is 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.	
55	*<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 is written to the least significant bits of the output; the last component is 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.	
56	*<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 is written to the least significant bits of the output; the last component is 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.	
57	*<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.

58	*<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. Use of this instruction requires declaration of the **Float64** capability.	
59	*<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 is extracted from the least significant bits of the input; the last component is 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.	
60	*<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 is extracted from the least significant bits of the input; the last component is 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.

61	*<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. Subnormal numbers are either preserved or flushed to zero, consistently within an implemenation.

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.

62	*<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 is extracted from the least significant bits of the input; the last component is 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.

63	*<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 is extracted from the least significant bits of the input; the last component is 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.	
64	*<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. Use of this instruction requires declaration of the **Float64** capability.	
65	*<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*.	
66	*<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.

67	*<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.

68	*<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.

69	*<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.

70	*<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*

This computation assumes *N* is already normalized.

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.

71	*<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*

This computation assumes the input parameters for the incident vector *I* and the surface normal *N* are already normalized.

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 floating-point scalar.

Result Type, the type of *I*, the type of *N*, and the type of *eta* must all have the same component type.

72	*<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 has all bits set (e.g., -1 if interpreted as signed). *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.	
73	*<id>* *Value*

FindSMsb Signed-integer most-significant bit, with *Value* interpreted as a signed integer. For positive numbers, the result is the bit number of the most significant 1-bit. For negative numbers, the result is the bit number of the most significant 0-bit. For a *Value* of 0 or -1, the result has all bits set (e.g., -1 if interpreted as signed). *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.	
74	*<id>* *Value*

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 has all bits set (e.g., -1 if interpreted as signed). *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.	
75	*<id>* *Value*

InterpolateAtCentroid

Result is the value of the input *interpolant* sampled at a location inside both the fragment 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 that *interpolant* points to must be the same type.

Use of this instruction requires declaration of the **InterpolationFunction** capability.

76	*<id>* *interpolant*

InterpolateAtSample

Result is the value of the input *interpolant* variable at the location of sample number *sample*. 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 that *interpolant* points to must be the same type.

Use of this instruction requires declaration of the **InterpolationFunction** capability.

77	*<id>* *interpolant*	*<id>* *sample*

InterpolateAtOffset

Result is the value of the input *interpolant* variable sampled at an offset from the center of the fragment 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 fragment. 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 that *interpolant* points to must be the same type.

Use of this instruction requires declaration of the **InterpolationFunction** capability.

78	*<id>* *interpolant*	*<id>* *offset*

NMin

Result is *y* if *y* < *x*, either *x* or *y* if both *x* and *y* are zeros, otherwise *x*. If one operand is a NaN, the other operand is the result. If both operands are NaN, the result 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.

79	*<id>* *x*	*<id>* *y*

NMax

Result is *y* if *x* < *y*, either *x* or *y* if both *x* and *y* are zeros, otherwise *x*. If one operand is a NaN, the other operand is the result. If both operands are NaN, the result 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.

80	*<id>* *x*	*<id>* *y*

<table>
<tr><td colspan="4">**NClamp**

Result is min(max(*x*, *minVal*), *maxVal*). The resulting value 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.</td></tr>
<tr><td>81</td><td>*<id>*
x</td><td>*<id>*
minVal</td><td>*<id>*
maxVal</td></tr>
</table>

Chapter 3. Appendix A: Changes

3.1. 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

3.2. 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).

3.3. 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.
- Bug 14548 Document the **Capability** needed for **Double2x32** and **InterpolateAt** instructions.

3.4. Changes from Version 1.00, Revision 1

- Bug 14548 Document the **Capability** needed for **UnpackDouble2x32**.

3.5. Changes from Version 1.00, Revision 2

- Change **precise** to **NoContraction**

3.6. Changes from Version 1.00, Revision 3

- Allow both 16-bit and 32-bit floating-point types in most places where before only 32-bit floating-point types were allowed. This does not effect whether 16-bit floating point types are allowed, which is selected independently. Since 16-bit types were historically disallowed, this is a backward compatible change.
- Fix Khronos internal issue #109: be more clear for NMin/NMax: If both operands are NaN, the result is a NaN.

3.7. Changes from Version 1.00, Revision 4

- Be clear about **UnpackHalf2x16** denorm rules.

3.8. Changes from Version 1.00, Revision 5

Fixed:

- Khronos SPIR-V Issue #211: As with **FindSMsb** and **FindUMsb**, **FindILsb** needs 32-bit components.

3.9. Changes from Version 1.00, Revision 6

Fixed:

- Khronos SPIR-V Issue #337: The component types of the operands for **Refract** must all be the same.
- Khronos SPIR-V Issue #331: Correct the types in **ModfStruct**.

3.10. Changes from Version 1.00, Revision 7

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

3.11. Changes from Version 1.00, Revision 8

Fixed:

- Khronos SPIR-V Issue #466: **FAbs** of *-0.0* is *+0.0*, **FSign** of *-0.0* can be either ±*0.0*. **FMin**, **FMax**, **NMin**, and **NMax** are allowed to return either operand when both are zeros.
- Khronos SPIR-V Issue #458: For **Frexp**, be more clear about negative values, and also about which returned value is being discussed.

3.12. Changes from Version 1.00, Revision 9

- Corrected the output range of **Atan**.

3.13. Changes from Version 1.00, Revision 10

- State what **FSign** of ±*NaN* is.

3.14. Changes from Version 1.00, Revision 11

- Khronos SPIR-V Issue #555: Deprecate **Modf**, use **ModfStruct** instead. Deprecate **Frexp**, use **FrexpStruct** instead.
- Khronos SPIR-V Issue #284: Say all bits are set, instead of saying -1, for some results of **FindILsb**, **FindSMsb**, and **FindUMsb**.
- Khronos SPIR-V MR #181: Use "fragment" instead of "pixel" in **InterpolateAtCentroid**, **InterpolateAtSample**, and **InterpolateAtOffset**.