The *COMPLETE* HLSL Reference

All the information developers need from shader assembly instruction to a complete HLSL intrinsic function overview covering up to shader model 3.0. Presented in a quick and easy to access format!

Sebastien St-Laurent

Paradoxal Press

Vertex and Pixel Shader Assembly Language

Although the development of assembly level shaders is rare today because of the HLSL language, a proper understanding of the underlaying assembly instruction sets is core to the development and debugging of efficient shaders.

Both the vertex and pixel shader assembly languages are composed of setup and assembly instructions. Throughout the following pages you will find tables containing a summaty of both types of instructions along with a description of their function, parameter information, performance indications and support of various shader versions.

Before starting, I have to make a note in regards to some of the flow control instructions. For example, the *if_comp* instruction defines flow control based on a comparison where the *_comp* component will be replaced by a comparison operation. The table below summarizes all the operations which can be used within flow control:

Flow Control Comparison Modes

Mode	Description
_gt	Greater
_lt	Lesser
_ge	Greater or equal
_le	Lesser or equal
_eq	Equal
_ne	Not equal

In addition , the following tables contain some indication of the support of each instruction on every vertex and pixel shader version using a color coded system. The table below summarizes the coding:

Color	Description
	Requires less than 4 instructions slots.
	Requires between 4 and 7 instruction slots.
	Required 8 or more instruction slots.
	Unsupported.

I must also make a final note in regards to the performance values indicated over the next pages. The quoted values are based on the DirectX specifications and indicate the worst case performance that must be ensured by the rendering hardware. This means that certain hardware implementations may execute certain instructions faster than perscribed by the specifications.

Vertex Shader Assembly

Name	Description	1.1	2.0	2.x	3.0
dcl_usage	Declare input registers.				
def	Define constants.				
defb	Define boolean constants.				
defi	Define integer constants.				
label	Define flow control labels.				
vs	Retrieve the shader version.				

Name	1.1	2.0	2.x	3.0
	Description			
	Additional Parameter Info			
abs dst, src		1	1	1
	Absolute value of the *src*.			
add dst, src0, src1	1	1	1	1
	Add *src0* and *src1* together.			
break			1	1
	Break out of a *loop* or *rep* block.			
break_comp src0, src1			3	3
	Conditionally break out of a *loop* or *rep* block based on a scalar conditional.			
	_comp = comparison mode. src0, src1 = source registers.			
breakp [!]p0.{x|y|z|w}			3	3
	Break out of a *loop* or *rep* block based on a predicate.			
	[!] = optional *NOT* operator. p0 = is a predicate register. {x|y|z|w} = required replicate swizzle.			
call l#		2	2	2
	Call a subroutine defined by a label.			
	l# = label to subroutine.			
callnz_bool l#, [!]b#		2	3	3
	Call a subroutine if register is nonzero.			
	l# = label to call. [!] = optional *NOT* operator. b# = boolean register.			
callnz_pred l#, [!]p0.{x|y|z|w}			3	3
	Call a subroutine based on a predicate.			
	l# = label to call. [!] = optional *NOT* operator. p0 = predicate register. {x|y|z|w} = required replicate swizzle.			
crs dst, src0, src1		2	2	2
	Cross product of *src0* and *src1*.			
dp3 dst, src0, src1	1	1	1	1
	Three-component dot product.			
dp4 dst, src0, src1	1	1	1	1
	Four-component dot product.			

Name	1.1	2.0	2.x	3.0
	Description			
	Additional Parameter Info			
dst dst, src0, src1	1	1	1	1
	Calculate lighting attenuation vector.			
	src0 = (ignored, d×d, d×d, ignored). src1 = (ignored, 1/d, ignored, 1/d). dst = (1, d, d×d, 1/d).			
else		1	1	1
	Begin an *else* block.			
endif		1	1	1
	End an *if/else* block.			
endloop		2	2	2
	End a *loop* block.			
endrep		2	2	2
	End a *rep* block.			
exp dst, src	10	1	1	1
	Full precision 2^X.			
expp dst, src	1	1	1	1
	Partial precision (16-bits) 2^X.			
frc dst, src	1	1	1	1
	Fractional component of input. **vs_1_1 can only write to .y and .xy.**			
if bool		3	3	3
	Begin an *if/else/endif* block.			
	bool = boolean register.			
if_comp src0, src1			3	3
	Begin an *if* block with a comparison.			
	_comp = comparison mode. src0, src1 = source registers.			
if [!]pred.{x|y|z|w}			3	3
	Begin an *if* block with a predicate.			
	[!] = optional *NOT* operator. pred = predicate register. {x|y|z|w} = required replicate swizzle.			
lit dst, src	1	3	3	3
	Partial lighting calculation.			
	dst = destination register. src = (N•L, N•H, unused, exponent).			
log dst, src	10	1	1	1
	Full precision log2(x).			
logp dst, src	1	1	1	1
	Partial precision (16-bits) log2(x).			
loop aL, i#		3	3	3
	Start a *loop* block.			
	aL = loop counting register. i# = constant integer register.			
lrp dst, src0, src1, src2		2	2	2
	Linearly interpolates between *src1* and *src2* using *src0* ([0..1] range).			
m3x2 dst, src0, src1	2	2	2	2
	Product of vector and a 2×3 matrix.			

Name	1.1	2.0	2.x	3.0	Description	Additional Parameter Info
m3x3 dst, src0, src1	3	3	3	3	Product of a vector and a 3×3 matrix.	
m3x4 dst, src0, src1	4	4	4	4	Product of a vector and a 4×3 matrix.	
m4x3 dst, src0, src1	3	3	3	3	Product of a vector and a 3×4 matrix.	
m4x4 dst, src0, src1	4	4	4	4	Product of a vector and a 4×4 matrix.	
mad dst, src0, src1, src2	1	1	1	1	Multiplies *src0* and *src1* then adds *src2*.	
max dst, src0, src1	1	1	1	1	Maximum of *src0* and *src1*.	
min dst, src0, src1	1	1	1	1	Minimum of *src0* and *src1*.	
mov dst, src	1	1	1	1	Move data from one register to another.	
mova dst, src		1	1	1	Move data to an address register.	
mul dst, src0, src1	1	1	1	1	Multiplies *src0* and *src1* together.	
nop	1	1	1	1	No operation.	
nrm dst, src		3	3	3	Normalizes the input (*src/length(src)*).	
pow dst, src0, src1		3	3	3	Returns $src0^{src1}$.	
rcp dst, src	1	1	1	1	Reciprocal of the input (*1/src*).	
rep i#		3	3	3	Starts a *rep* block.	I# = integer register with repeat count in *x* component.
ret		1	1	1	End subroutine or main.	
rsq dst, src	1	1	1	1	Reciprocal square root (*1/sqrt(src)*).	
setp_comp dst, src0, src1			1	1	Set the predicate register.	_comp = comparison mode. dst = destination predicate register. src0, src1 = source registers.
sge dst, src0, src1	1	1	1	1	Greater than or equal compare.	
sgn dst, src0, src1, src2		3	3	3	Returns the sign of the *src0*.	dst = -1 for negative and 1 for positive. src0 = source registers. src1, src2 = temporary registers used for calculation.

Name	1.1	2.0	2.x	3.0	Description	Additional Parameter Info
sincos dst.{x|y|xy}, src0.{x|y|z|w}, src1, src2		8	8	8	Computes both the sine and cosine.	dst = destination register (*.x* contains sin and *.y* contains cos and has to point to a temporary register). src0 = input angle source register. src1 = *D3DSINCOSCONST1*. src2 = *D3DSINCOSCONST2*.
slt dst, src0, src1	1	1	1	1	Less than compare.	
sub dst, sub0, sub1	1	1	1	1	Subtracts *src1* from *src0*.	
texldl dst, src0, src1				2, 5 for Cubemap	Loads a texture sample.	dst = destination register. src0 = texture coordinate. src1 = texture sampler.

Pixel Shader Assembly

Name	Description	1.x	2.0	2.x	3.0
dcl	Declare input registers.				
dcl_samplerType	Declare texture dimensions for a sampler.				
phase	Transition between phase 1 and phase 2 shader code.				
def	Define constants.				
defb	Define boolean constants.				
defi	Define integer constants.				
label	Define labels.				
ps	Retrieve the shader version.				

Name	1.1	1.2	1.3	1.4	2.0	2.x	3.0	Description	Parameter Information
abs dst, src					1	1	1	Return the absolute value of *src*.	
add dst, src0, src1	1	1	1	1	1	1	1	Add *src0* and *src1* together.	
bem dst.rg, src0, src1				2				Apply a fake bumpmaping transform.	dst = destination register (*.rg* only). src0, src1 = source registers.
break						1	1	Break out of a *loop* or *rep* block.	

Name	1.1	1.2	1.3	1.4	2.0	2.x	3.0
	Description						
	Parameter Information						
break_comp src0, src1						3	3
	Conditional break out of a *loop* or *rep*.						
	comp = comparison mode. src0, src1 = source registers.						
breakp [!]p0.{x|y|z|w}						3	3
	Predicate break out of a *loop* or *rep*.						
	[!] = optional *NOT* operator. p0 = predicate register. {x|y|z|w} = required replicate swizzle.						
call l#						2	2
	Call the specified subroutine.						
	l# = label to call						
callnz l#, [!]b#						3	3
	Call a subroutine on a nonzero boolean.						
	l# = label to call. [!] = optional *NOT* operator. b# = boolean register.						
callnz l#, [!]p0.{x|y|z|w}						3	3
	Call a subroutine on a predicate.						
	l# = label to call. [!] = optional *NOT* operator. p0 = predicate register. {x|y|z|w} = required replicate.						
cmp dst, src0, src1		2	2	1	1	1	1
	Compare source vector to 0.						
cnd dst, src0, src1, src2	1	1	1	1			
	Compare *src0* vector to 0.5, returning either *src1* or *src2*.						
crs dst, src0, src1					2	2	2
	Cross product of *src0* and *src1*.						
dp2add dst, src0, src1, src2.{x|y|z|w}					2	2	2
	2D dot product with scalar addition to the result.						
dp3 dst, src0, src1	1	1	1	1	1	1	1
	Three-component vector dot product.						
dp4 dst, src0, src1		2	2	1	1	1	1
	Four-component dot product.						
dsx dst, src						1	1
	Rate of change in the *x* direction.						
dsy dst, src						1	1
	Rate of change in the *y* direction.						
else						1	1
	Begin an *else* block.						
endif						1	1
	End an *if/else* block.						

<table>
<tr><td rowspan="3">Name</td><td>1.1</td><td>1.2</td><td>1.3</td><td>1.4</td><td>2.0</td><td>2.x</td><td>3.0</td></tr>
<tr><td colspan="7">Description</td></tr>
<tr><td colspan="7">Parameter Information</td></tr>
<tr><td rowspan="2">endloop</td><td></td><td></td><td></td><td></td><td></td><td>2</td><td>2</td></tr>
<tr><td colspan="7">End a loop/endloop block.</td></tr>
<tr><td rowspan="2">endrep</td><td></td><td></td><td></td><td></td><td></td><td>2</td><td>2</td></tr>
<tr><td colspan="7">End a repeat block.</td></tr>
<tr><td rowspan="2">exp dst, src</td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Full precision 2^x.</td></tr>
<tr><td rowspan="2">frc dst, src</td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Return the fractional of src.</td></tr>
<tr><td rowspan="3">if bool</td><td></td><td></td><td></td><td></td><td></td><td>3</td><td>3</td></tr>
<tr><td colspan="7">Begin an if /else/endif block.</td></tr>
<tr><td colspan="7">bool = boolean register.</td></tr>
<tr><td rowspan="3">if_comp src0, src1</td><td></td><td></td><td></td><td></td><td></td><td>3</td><td>3</td></tr>
<tr><td colspan="7">Begin an if block, with a comparison.</td></tr>
<tr><td colspan="7">_comp = comparison mode.
src0, src1 = source registers.</td></tr>
<tr><td rowspan="3">if_pred [!]pred.{x|y|z|w}</td><td></td><td></td><td></td><td></td><td></td><td>3</td><td>3</td></tr>
<tr><td colspan="7">Begin an if block with a predicate.</td></tr>
<tr><td colspan="7">[!] = optional NOT operator.
pred = predicate register.
{x|y|z|w} = required replicate swizzle.</td></tr>
<tr><td rowspan="2">log dst, src</td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Full precision log2(x).</td></tr>
<tr><td rowspan="2">lrp dst, src0, src1, src2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td colspan="7">Interpolation of src1 and src2 based on src0 (in the [0..1] range).</td></tr>
<tr><td rowspan="2">m3x2 dst, src0, src1</td><td></td><td></td><td></td><td></td><td>2</td><td>2</td><td>2</td></tr>
<tr><td colspan="7">Product of a vector and a 2×3 matrix.</td></tr>
<tr><td rowspan="2">m3x3 dst, src0, src1</td><td></td><td></td><td></td><td></td><td>3</td><td>3</td><td>3</td></tr>
<tr><td colspan="7">Product of a vector and a 3×3 matrix.</td></tr>
<tr><td rowspan="2">m3x4 dst, src0, src1</td><td></td><td></td><td></td><td></td><td>4</td><td>4</td><td>4</td></tr>
<tr><td colspan="7">Product of a vector and a 4×3 matrix.</td></tr>
<tr><td rowspan="2">m4x3 dst, src0, src1</td><td></td><td></td><td></td><td></td><td>3</td><td>3</td><td>3</td></tr>
<tr><td colspan="7">Product of a vector and a 3×4 matrix.</td></tr>
<tr><td rowspan="2">m4x4 dst, src0, src1</td><td></td><td></td><td></td><td></td><td>4</td><td>4</td><td>4</td></tr>
<tr><td colspan="7">Product of a vector and a 4×4 matrix.</td></tr>
<tr><td rowspan="2">mad dst, src0, src1, src2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Multiply src0 and src1 together adding src2 to the result.</td></tr>
<tr><td rowspan="2">max dst, src0, src1</td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Maximum between src0 and src1.</td></tr>
<tr><td rowspan="2">min dst, src0, src1</td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Minimum between src0 and src1.</td></tr>
<tr><td rowspan="2">mov dst, src</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="7">Move data from one register to another.</td></tr>
</table>

Name	1.1	1.2	1.3	1.4	2.0	2.x	3.0
	Description						
	Parameter Information						
mul dst, src0, src1	1	1	1	1	1	1	1
	Multiplies *src0* and *src1* together.						
nop	1	1	1	1	1	1	1
	No operation.						
nrm dst, src					3	3	3
	Normalize a vector. (*src/length(src)*).						
pow dst, src0, src1					3	3	3
	Return $src0^{src1}$.						
rcp dst, src					1	1	1
	Return the reciprocal of *src*.						
rep i#						3	3
	Start a *rep* block.						
	i# = integer register specifying the loop count in the *x* component.						
ret						1	1
	End a subroutine or main.						
rsq dst, src					1	1	1
	Reciprocal square root, or *1/sqrt(src)*.						
setp_comp dst, src0, src1						1	1
	Set the predicate register.						
	_comp = comparison mode. dst = destination predicate register. src0, src1 = source registers.						
sincos dst.{x|y|xy}, src0.{x|y|z|w}, src1, src2					8	8	8
	Compute both the sine and cosine.						
	dst = destination register (*.x* contains sin and *.y* contains cos and has to point to a temporary register). src0 = input angle. src1 = *D3DSINCOSCONST1*. src2 = *D3DSINCOSCONST2*.						
sub dst, src0, src1	1	1	1	1	1	1	1
	Subtract *src1* from *src0*.						
tex dst	1	1	1				
	Sample a texture.						
texbem dst, src	1	1	1				
	Apply a fake bumpmap transform.						
texbeml dst, src	2	2	2				
	Apply a bumpmap with luminance.						
texcoord dst	1	1	1				
	Treat texture coordinates as color data.						
texcrd dst, src				1			
	Copy texture coordinate as color data.						
texdepth dst				1			
	Calculate depth values.						

Name	1.1	1.2	1.3	1.4	2.0	2.x	3.0
	Description						
	Parameter Information						
texdp3 dst, src		1	1				
	Three-component dot product between texture and the texture coordinates.						
texdp3tex dst, src		1	1				
	Three-component dot product and 1D texture lookup.						
texkill src	1	1	1	1	1	Note 1	2
	Cancel rendering of pixels based on a comparison.						
	src0 = kill pixel if any component < 0.						
texld dst, src				1	1	Note 2	Note 5
	Sample a texture.						
texldb dst, src0, src1					1	Note 3	6
	Sample texture with level of detail bias from *w*-component of the texture coordinates.						
	dst = destination register. src0 = texture coordinates. src1 = sampler.						
texldd dst, src0, src1, src2, src3						3	3
	Sample texture with gradient.						
	dst = destination register. src0 = texture coordinates. src1 = sampler. src2 = x gradient. src3 = y gradient.						
texldl dst, src0, src1							Note 6
	Sample texture with LOD from *w*-component.						
	dst = destination register. src0 = texture coordinates. src1 = sampler.						
Texldp dst, src0, src1					1	Note 4	Note 7
	Sample texture with projective divide by *w*-component.						
	dst = destination register. src0 = texture coordinates. src1 = sampler.						
texm3x2depth dst, src			1				
	Calculate per-pixel depth values.						
texm3x2pad dst, src	1	1	1				
	First row matrix multiply of a two.						
texm3x2tex dst, src	1	1	1				
	Final row matrix multiply of a two.						
texm3x3 dst, src		1	1				
	3×3 matrix multiply.						

Name	1.1	1.2	1.3	1.4	2.0	2.x	3.0
	Description						
	Parameter Information						
texm3x3pad dst, src	1	1	1				
	First or second row multiply of three.						
texm3x3spec dst, src0, src1	1	1	1				
	Final row multiply of three.						
texm3x3tex dst, src	1	1	1				
	Texture look up using a 3×3 matrix multiply.						
texm3x3vspec dst, src	1	1	1				
	Texture look up using a 3×3 matrix multiply, with nonconstant eye-ray vector.						
texreg2ar dst, src	1	1	1				
	Sample a texture using the alpha and red components.						
texreg2gb dst, src	1	1	1				
	Sample a texture using the green and blue components.						
texreg2rgb dst, src		1	1				
	Sample a texture using the red, green and blue components.						

Note 1: If *D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT* is set, slots = 2; otherwise slots = 1.

Note 2: If *D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT* is set and the texture is a cube map, slots = 4; otherwise slot = 1.

Note 3: If *D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT* is set, slots = 6; otherwise slots = 1.

Note 4: If *D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT* is not set, slots = 1; otherwise: if the texture is a cube map, slots = 4; if the texture is not a cube map, slots = 3.

Note 5: If the texture is a cube map, slots = 4; otherwise slots = 1.

Note 6: If the texture is a cube map, slots = 5; otherwise slots = 2.

Note 7: If the texture is a cube map, slots = 4; otherwise slots = 3.

HLSL Intrinsic Functions

To simplify the development of shaders, the HLSL shading language introduces a high-level programming scheme similar to C. This simplifies the developers tasks and take their burden away from manual instruction optimization and register allocation.

The following table contains a complete overview of all the HLSL intrinsic functionality. The table contains information about the function such as a description of its use, parameter overview, performance consideration and an indication of the minimal shader version required to execute the function.

Keep in mind that performance figures given below are based on a simple shader using only the specified instruction in order to avoid optimizations. This should serve as a worst case scenario as real use cases will generally result in better performing shaders.

Name	VS	PS	Performance
	Parameters		
	Description		
ret abs(x)	1.1	1.4	Implemented in one instruction slot.
	x | in | scalar, vector or matrix | float, int **ret | out | same as input x | same as input x**		
	Return the absolute value of *x*.		
ret acos(x)	1.1	2.0	Taylor series taking 17 instruction slots.
	x | in | scalar, vector or matrix | float **ret | out | same as input x | float**		
	Return the arccosine value *x* (valid for [-1, 1]).		
ret all(x)	1.1	1.4	Implemented using 5 instruction slots.
	x | in | scalar, vector or matrix | float, int, bool **ret | out | scalar | bool**		
	Test if all the components of *x* are nonzero.		
ret any(x)	1.1	1.4	Implemented using 3 instruction slots.
	x | in | scalar, vector or matrix | float, int, bool **ret | out | scalar | bool**		
	Tests if any of the components of *x* are nonzero.		
ret asin(x)	1.1	2.0	Taylor series using 18 instruction slots.
	x | in | scalar, vector or matrix | float **ret | out | same as input x | float**		
	Return the arcsine value of *x* (valid for [-π/2, π/2]).		
ret atan(x)	1.1	2.0	Taylor series using 21 instruction slots.
	x | in | scalar, vector or matrix | float **ret | out | same as input x | float**		
	Return the arctan value of *x* (result range [-π/2, π/2]).		
ret atan2(x, y)	1.1	2.0	Taylor series using 25 instruction slots.
	x | in | scalar, vector or matrix | float **y | in | same as input x | float** **ret | out | same as input x | float**		
	Return the arctan of *x/y* (result range [-π/2, π/2]).		

Name	VS	PS	Performance / Parameters / Description
ret ceil(x)	1.1	2.0	Implemented with the *frc* instruction and takes 2 instruction slots but may take more on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Return the integer greater or equal to x.
ret clamp(ret, min, max)	1.1	1.4	Implemented using the *min* and *max* instructions taking 3 instruction slots.
			x | in | scalar, vector or matrix | float, int **min | in | same as input x | same as input x** **max | in | same as input x | same as input x** **ret | out | same as input x | same as input x**
			Returns the input *x* clamped to the range [*min*, *max*].
clip(x)	N/A	1.1	Removes the benefits of early Z rejection.
			x | in | scalar, vector or matrix | float
			This function will discard the current pixel if any of components of *x* are less than zero.
ret cos(x)	1.1	2.0	Taylor series using 15 instruction slots.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Return the cosine of *x*.
ret cosh(x)	1.1	2.0	Numerical approximation using the *exp* instruction. Uses 83 instructions slots on 1.x hardware , 11 slots on 2.0+ hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Return the hyperbolic cosine of *x*.
ret cross(x,y)	1.1	1.4	Emulated on 1.x hardware using 4 slots.
			x | in | vector | float **y | in | vector | float** **ret | out | vector | float**
			Execute the cross product between *x* and *y*.
ret ddx(x)	N/A	2.x	Verify capabilities before using.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Partial derivate in the screen space x coordinate.
ret ddy(x)	N/A	2.x	Verify capabilities before using.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Partial derivate of *x* in the screen space y coordinate.
ret degrees(x)	1.1	2.0	N/A
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Convert *x* from radians to degrees.
ret determinant(x)	1.1	1.4	Implemented using the proper equations taking about 12 instruction slots.
			x | in | matrix | float **ret | out | scalar | float**
			Return the determinant of the square matrix *x*.

Name	VS	PS	Performance	Parameters	Description
ret distance(x, y)	1.1	2.0	Implemented in 5 instructions slots.	**x | in | vector | float** **y | in | vector | float** **ret | out | scalar | float**	Compute the distance between *x* and *y*.
ret dot(x, y)	1.1	2.0	N/A	**x | in | vector | float, int** **y | in | vector | float, int** **ret | out | scalar | float, int**	Compute the dot product between *x* and *y*.
ret exp(x)	1.1	2.0	Excecutes in 5 instructions slots and 41 slots on 1.x hardware.	**x | in | scalar, vector or matrix | float** **ret | out | same as input x | float**	Return base-e exponential of *x*.
ret exp2(x)	1.1	2.0	Executes in 4 instructions slots and 40 slots on 1.x hardware.	**x | in | scalar, vector or matrix | float** **ret | out | same as input x | float**	Return base-2 exponential of *x*.
ret faceforward(n, i, ng)	1.1	1.4	Implemented in 7 instruction slots.	**n | in | vector | float** **i | in | vector | float** **ng | in | vector | float** **ret | out | vector | float**	Test if a face is visible using the following equation: *ret = -n * sign(dot(i, ng))*.
ret floor(x)	1.1	2.0	Implemented using the *frc* function in two instruction slots but may take more on 1.x hardware.	**x | in | scalar, vector or matrix | float** **ret | out | same as input x | float**	Return the largest integer that is lesser than *x*.
ret fmod(x, y)	1.1	2.0	Implemented in 11 instruction slots and 15 slots on 1.x hardware.	**x | in | scalar, vector or matrix | float, int** **y | in | same as input x | same as input x** **ret | out | same as input x | same as input x**	Return the floating-point reminder *f* of *x/y*.
ret frac(x)	1.1	2.0	Implemented in 3 instructions slots using the *frc* instructions and is subject to restrictions on 1.x hardware.	**x | in | scalar, vector or matrix | float** **ret | out | same as input x | float**	Return the fractional part of *x*.
ret frexp(x, out exp)	1.1	2.0	Implemented using a sequence of *exp*, *log* and *rcp* instructions requiring 19 instructions slots and 97 slots on 1.x hardware.	**x | in | scalar, vector or matrix | float** **exp | out | same as input x | float** **ret | out | same as input x | float**	Separate *x* into mantissa and exponent components.

Name	VS	PS	Performance / Parameters / Description
ret fwidth(x)	N/A	2.x	See *ddx* and *ddy* for more information.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Compute the partial derivate of *x*. This function essentially computes *abs(ddx(x)) + abs(ddy(x))*.
ret isfinite(x)	1.1	2.0	Implemented using 3 instruction slots.
			x | in | scalar, vector or matrix | float **ret | out | scalar | bool**
			Return *true* if *x* is finite, *false* otherwise.
ret isinf(x)	1.1	2.0	See *isfinite*.
			x | in | scalar, vector or matrix | float **ret | out | scalar | bool**
			Return *true* if *x* is not finite, *false* otherwise.
ret isnan(x)	1.1	2.0	See *isfinite*.
			x | in | scalar, vector or matrix | float **ret | out | scalar | bool**
			Return *true* if *x* is either *NAN* or *QNAN*.
ret ldexp(x, exp)	1.1	2.0	Implemented using *exp*.
			x | in | scalar, vector or matrix | float **exp | in | scalar, vector or matrix | float** **ret | out | same as input x | float**
			Perform the reverse operation of *frexp* and returns $x * 2^{exp}$.
ret length(x)	1.1	2.0	Implemented in 3 instruction slots.
			x | in | vector | float **ret | out | scalar | float**
			Return the length of the input vector *x*.
ret lerp(x, y, s)	1.1	1.4	Implemented in 4 instruction slots.
			x | in | scalar, vector or matrix | float **y | in | scalar, vector or matrix | float** **s | in | scalar, vector or matrix | float** **ret | out | same as input x | same as input x**
			Linear interpolation between *x* and *y* based on *s*,interpolation factor in the range [0, 1].
ret lit(n dot l, n dot h, m)	1.1	2.0	Implemented using the *lit* instruction and takes 4 instruction slots.
			l | in | scalar | float **h | in | scalar | float** **m | in | scalar | float** **ret | out | vector | float**
			Calculates a lighting vector containing the ambient, diffuse and specular lighting components.
ret log(x)	1.1	2.0	Implemented using the *log* instruction taking 5 slots and 41 slots on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the base-e logarithm of the input *x*.
ret log2(x)	1.1	2.0	Implemented using the *log* instruction taking 4 slots and 40 slots on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the base-2 logarithm of the input *x*.

<table>
<tr><th>Name</th><th>VS</th><th>PS</th><th>Performance</th><th>Parameters</th><th>Description</th></tr>
<tr><td>ret log10(x)</td><td>1.1</td><td>2.0</td><td>Implemented using the log instruction taking 5 slots and 41 slots on 1.x hardware.</td><td>x | in | scalar, vector or matrix | float
ret | out | same as input x | float</td><td>Returns the base-10 logarithm of the input x.</td></tr>
<tr><td>ret max(x, y)</td><td>1.1</td><td>1.4</td><td>N/A</td><td>x | in | scalar, vector or matrix | float, int
y | in | same as input x | same as input x
ret | out | same as input x | same as input x</td><td>Returns the maximum of both x and y.</td></tr>
<tr><td>ret min(x, y)</td><td>1.1</td><td>1.4</td><td>N/A</td><td>x | in | scalar, vector or matrix | float, int
y | in | same as input x | same as input x
ret | out | same as input x | same as input x</td><td>Return the minimum of both x and y.</td></tr>
<tr><td>ret modf(x, out ip)</td><td>1.1</td><td>2.0</td><td>Implemented using the frc using 7 instruction slots and 13 on 1.1 hardware.</td><td>x | in | scalar, vector or matrix | float, int
ip | out | same as input x | same as input x
ret | out | same as input x | same as input x</td><td>Separates x into its fractional and integer part.</td></tr>
<tr><td>ret mul(x, y)</td><td>1.1</td><td>1.1</td><td>Dependant on the input types.</td><td>x | in | scalar | float, int
y | in | scalar | same as input x
ret | out | scalar | same as input x

x | in | scalar | float, int
y | in | vector | float, int
ret | out | vector | float, int

x | in | scalar | float, int
y | in | matrix | float, int
ret | out | matrix | float, int

x | in | vector | float, int
y | in | scalar | float, int
ret | out | vector | float, int

x | in | vector | float, int
y | in | vector | float, int
ret | out | scalar | float, int

x | in | vector | float, int
y | in | matrix | float, int
ret | out | vector | float, int

x | in | matrix | float, int
y | in | scalar | float, int
ret | out | matrix | float, int

x | in | matrix | float, int
y | in | vector | float, int
ret | out | vector | float, int

x | in | matrix | float, int
y | in | matrix | float, int
ret | out | matrix | float, int</td><td>Perform a multiplication between x and y.</td></tr>
<tr><td>ret noise(x)</td><td>N/A</td><td>N/A</td><td>Only operates on texture shaders.</td><td>x | in | vector | float
ret | out | scalar | float</td><td>Generates Perlin noise values based on x.</td></tr>
</table>

Name	VS	PS	Performance / Parameters / Description
ret normalize(x)	1.1	2.0	Implemented in 3 instruction slots.
			x | in | vector | float **ret | out | same as input x | float**
			Returns a normalized version of *x*.
ret pow(x, y)	1.1	2.0	Implemented using *exp* and *log* taking 9 slots and 81 on 1.x hardware.
			x | in | scalar, vector or matrix | float **y | in | same as input x | float** **ret | out | same as input x | float**
			Returns *x* to the power of *y*.
ret radians(x)	1.1	1.1	N/A
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Convert *x* from degrees to radians.
ret reflect(i, n)	1.1	1.1	Implemented in 5 instruction slots.
			i | in | vector | float **n | out | vector | float** **ret | out | vector | float**
			Compute a reflection vector given the incident vector *i* and a reflection surface normal *n*.
ret refract(i, n, ri)	1.1	2.0	Implemented in 13 instruction slots.
			i | in | vector | float **n | in | vector | float** **ri | in | scalar | float** **ret | out | vector | float**
			Computes a refracted vector given an incident vector *i*, a surface normal *n* and a refraction index *ri*.
ret round(x)	1.1	2.0	Implemented using *frc* instruction and is limited on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Rounds the input *x* to its nearest integer.
ret rsqrt(x)	1.1	2.0	N/A
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the reciprocal square root of *x*.
ret saturate(x)	1.1	1.1	See *clamp*.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the absolute of *x* clamped to the range [0, 1].
ret sign(x)	1.1	1.4	Implemented in 3 instruction slots.
			x | in | scalar, vector or matrix | float, int **ret | out | same as input x | int**
			Returns the sign of *x*.
ret sin(x)	1.1	2.0	Taylor series using 15 instruction slots.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the sine of *x*.

Name	VS	PS	Performance / Parameters / Description
ret sincos(x, out s, out c)	1.1	2.0	Taylor series using 24 instruction slots.
			x | in | scalar, vector or matrix | float **s | out | same as input x | float** **c | out | same as input x | float** **ret | out | same as input x | float**
			Returns both the sine and cosine of *x*.
ret sinh(x)	1.1	2.0	Numerical approximation using *exp* using 11 instruction slots or 83 on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the hyperbolic sine of *x*.
ret smoothstep(min, max, x)	1.1	2.0	Implemented using 14 instruciton slots.
			x | in | scalar, vector or matrix | float **min | in | same as input x | float** **max | in | same as input x | float** **ret | out | same as input x | float**
			Returns a smooth Hermite interpolation between 0 and 1 if the input *x* is in the range of [*min*, *max*].
ret sqrt(x)	1.1	2.0	N/A
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the square root of *x*.
ret step(y, x)	1.1	1.4	Implemented using the *sge* instruction.
			x | in | scalar, vector or matrix | float **y | in | same as input x | float** **ret | out | same as input x | float**
			Returns *(x >= y) ? 1 : 0*.
ret tan(x)	1.1	2.0	Taylor series yielding 23 instruction slots and 35 on 1.x hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the tangent of *x*.
ret tanh(x)	1.1	2.0	Numerical approximation with *exp* resulting in 16 slots and 88 on 1.x shader hardware.
			x | in | scalar, vector or matrix | float **ret | out | same as input x | float**
			Returns the hyperbolic tangent of *x*.
ret transpose(x)	1.1	1.1	N/A
			x | in | matrix | float, int, bool **ret | out | matrix | float, int, bool**
			Returns the transpose of the input matrix *x*.

HLSL Texture Functions

There are several flavors of texture access functions serving different purposes. However, each of them exists for several types of textures. The variety of texture function will be presented in the form *texXX*, where *XX* can be one of the following:

- 1D: Texture access to a 1D texture
- 2D: Texture access to a 2D texture.
- 3D: Texture access to a 3D texture.
- CUBE: Texture access to a cubemap texture.

Name	PS	Performance / Parameters / Description
		Parameters
		Description
ret texXX(s, t)	1.1	N/A
		s | in | object | sampler **t | in | vector | float** **ret | out | vector | float**
		Performs a simple texture lookup.
ret texXX(s, t, ddx, ddy)	1.1	N/A
		s | in | object | sampler **t | in | vector | float** **ddx | in | scalar | float** **ddy | in | scalar | float** **ret | out | vector | float**
		Performs a texture lookup using the derivates *ddx* and *ddy* to pick the proper texture mipmap.
ret texXXbias(s, t)	2.0	N/A
		s | in | object | sampler **t | in | scalar, vector | float** **ret | out | vector | float**
		Performs a texture lookup using the *w* component to bias mipnapping.
ret texXXlod(s, t)	3.0	N/A
		s | in | object | sampler **t | in | scalar, vector | float** **ret | out | vector | float**
		Performs a texture lookup using the *w* component to determine the mipmap.
ret texXXgrad(s, t, ddx, ddy)	2.0	Under *ps_3_x*, the texture access must be done outside of flow control.
		s | in | object | sampler **t | in | vector | float** **ddx | in | scalar | float** **ddy | in | scalar | float** **ret | out | vector | float**
		Performs a texture lookup using the derivates *ddx* and *ddy* to pick the proper texture mipmap.
ret texXXproj(s, t)	1.4	N/A
		s | in | object | sampler **t | in | scalar, vector | float** **ret | out | vector | float**
		Performs a projective texture lookup.

Author: Sebastien St-Laurent
Technical Review: Wolfgang Engel
Layout & Design: Sebastien St-Laurent
Proofreader: Nicole St-Laurent

PRINTED IN THE U.S.A.

Use velcro to affix this booklet to the side of your monitor.

Paradoxal Press

Paradoxal Press
9981 Avondale Rd. NE
Redmond, WA 98052
sebastien.st.laurent@gmail.com

$7.99 USA / $9.99 CANADA

ISBN-13 978-0-9766132-0-6
ISBN-10 0-9766132-0-4

50799>

9 780976 613206