Reference Guide

AMD Accelerated Parallel Processing TECHNOLOGY

AMD Intermediate Language (IL)

October 2011

v. 2.4

doc_rev 2.4

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453
www.amd.com

For AMD Accelerated Parallel Processing:

URL:	**developer.amd.com/appsdk**
Developing:	**developer.amd.com/**
Support:	**developer.amd.com/appsdksupport**
Forum:	**developer.amd.com/openclforum**

Contents

Tables

Preface

About This Document

This document describes the instruction set for the AMD IL compiler.

The document serves two purposes:

1. It specifies the language constructs and behavior, including the organization of each type of instruction in both text syntax and binary format.
2. It provides a reference of instruction operation that compiler writers can use to maximize performance of the processor.

Audience

This document is intended for programmers writing application and system software, including operating systems, compilers, loaders, linkers, device drivers, and system utilities. It assumes an understanding of the AMD GPU processor microarchitecture and of programming practices for either graphics or general-purpose computing.

Contact Information

To submit questions or comments about this document, contact our technical documentation staff at: streamcomputing@amd.com.

For questions concerning AMD Accelerated Parallel Processing products, please email: streamcomputing@amd.com.

For questions about developing with AMD Accelerated Parallel Processing, please submit a helpdesk request at AMD_Software_Developer_Help_Request.

You can learn more about AMD Accelerated Parallel Processing at: http://www.amd.com/stream.

We also have a growing community of AMD Accelerated Parallel Processing users! Come visit us at the AMD Accelerated Parallel Processing Developer Forum (http://www.amd.com/streamdevforum) to find out what applications other users are trying on their AMD Accelerated Parallel Processing products!

Organization

This document begins with an overview summarizing the similarities and differences between AMD Intermediate Language (IL) and general-purpose computer languages. It describes text and binary formats of the IL program instructions. Then, it describes the types of instructions in detail, presenting a high-level description of the instruction fields, and restrictions that must be observed. It also describes the instruction syntax for text representation. Further, it presents the specification of each type of instruction. A glossary of terms and acronyms ends the document.

Endian Order

The R600, R700, and Evergreen GPU architectures address memory and registers using little-endian byte-ordering and bit-ordering. Multi-byte values are stored with their least-significant (low-order) byte (LSB) at the lowest byte address; they are illustrated with their LSB at the right side. Byte values are stored with their least-significant (low-order) bit (lsb) at the lowest bit address; they are illustrated with their lsb at the right side.

Conventions

The following conventions are used in this document.

`mono-spaced font`	A filename, file path, or code.
*	Any number of alphanumeric characters in the name of a microcode format, microcode parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
1011b	A binary value, in this example a 4-bit value.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.

Terminology

In graphics applications, programs written for a GPU often are called “shaders.” In compute applications, similar programs usally are called “kernels.” This document defines a low-level language that can be used to write both shaders and kernels.

Related Documents

- AMD, *R600-Family Instruction Set Architecture*, Sunnyvale, CA, 2008. This document includes the RV670 GPU instruction details.
- ISO/IEC 9899:TC2 - *International Standard - Programming Languages - C*
- Kernighan Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1978.
- IEEE, *754-1985 IEEE Standard for Binary Floating-Point Arithmetic*, 2003.
- I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *“Brook for GPUs: stream computing on graphics hardware,”* ACM Trans. Graph., vol. 23, no. 3, pp. 777–786, 2004.
- Buck, Ian; Foley, Tim; Horn, Daniel; Sugerman, Jeremy; Hanrahan, Pat; Houston, Mike; Fatahalian, Kayvon. “BrookGPU” http://graphics.stanford.edu/projects/brookgpu/
- Buck, Ian. “Brook Spec v0.2”. October 31, 2003. http://merrimac.stanford.edu/brook/brookspec-05-20-03.pdf
- *OpenGL Programming Guide*, at http://www.glprogramming.com/red/
- *Microsoft DirectX Reference Website*, at http://msdn.microsoft.com/en-us/library/bb219740(VS.85).aspx
- *Microsoft Programming Guide for HLSL*, http://msdn2.microsoft.com/en-us/library/bb509635.aspx
- *GPGPU*: http://www.gpgpu.org, and Stanford BrookGPU discussion forum http://www.gpgpu.org/forums/

Chapter 1 Overview

This document defines the format and behavior of the IL. The Intermediate Language (IL) is an abstract representation for hardware vertex, pixel, and geometry shaders, as well as compute kernels that can be taken as input by other modules implementing the IL. An IL compiler uses an IL shader or kernel in conjunction with driver state information to translate these shaders into hardware instructions or a software emulation layer.

1.1 Open Design

The IL adopts an open design, where most instructions can appear in any kind of shader.

Note that the IL does not enforce usage restrictions of the input language, but only verifies that the IL, as specified, is properly programmed. For example, in DX, instruction modifiers such as `_x2` general-purpose not be applied to texture address instructions. The IL does not enforce this restriction, since both the IL and current architectures support such operations.

Also, this manual does not describe how to optimally code the IL. By design, the IL has an extensive set of operations, but some of these are not supported by existing hardware. The programmer is advised to adhere in most cases to the syntax restrictions and performance guidelines of the input language.

1.2 DirectX as a Design Basis

Because of changes between DX9 and DX10, many instructions occur in multiple forms:

- unconditional
- conditional (comparing two float values)
- logical (comparing a single integer value with zero)
- Boolean (comparing a Boolean register with true)

To support DX10, many operations required several similar IL opcodes. For example, there are three forms of break: unconditional, break on a Boolean, and break on a logical value.

Other operations were required both in vector and scalar forms. For example, there are two `rsq` instructions: `rsq` corresponds to the IL 1 scalar opcode, `rsq_vec` corresponds to the DX10 vector form that computes the reciprocal square root on each component.

In DX9, 0* any value was defined to be 0. DX10 changed this to more closely match IEEE arithmetic which defines 0*Nan = Nan.

All float operations containing a multiply now take a flag to specify Nan behavior.

1.3 Threading Model

A hierarchical threading model is used. A *kernel* (a shader program running on a GPU) can be launched with a number of work-groups. Work-items within this group can communicate through local shared memory. There is no supported communication between work-groups. Work-items in a work-group run in units called wavefronts. All work-items in a wavefront run in SIMD fashion (in lock steps). All wavefronts within a work-group can be synchronized using a synchronization barrier operation.

1.4 Access Model for Local Shared Memory

Each processor has an amount of local memory that can be shared across the work-items in a work-group.[1] IL provides two models of memory access to local data share (LDS).

The first memory access model, called owner-computes, is supported by the HD4000-family of devices. In owner-computes, each work-item in a work-group owns an area of LDS memory. The size of the area is declared in the shader. Each work-item in a group can write only to the area of memory it owns; however, a work-item can read any chunk of memory that is owned by either itself or other work-items. An LDS shared memory read is specified by (`owner_thread_ID`, offset): read the memory area owned by that thread_ID with an offset within the area.

Different from the access model for work-items within a wavefront, the access mode for different wavefronts (within a work-group) is specified by the sharing mode, which is either relative or absolute. If it is relative, new and consecutive space is allocated for each wavefront; if it is absolute, all wavefronts are mapped to the same set of memory starting at address 0. In this mode, wavefronts can overwrite each other's data.

The second memory access model is a general read write: each work-item can read or write any address in the LDS. This model is supported on HD5XXX series graphics cards.

Both models allow work-items to read or write memory (video or system), but do not provide synchronization to memory.

Supported inter-work-item communication includes:

1. Note that generally there is a correspondence between work-item (the OpenCL nomenclature) and the previously used term "thread." This also is the case for work-group and "thread group."

- SR – Globally shared registers.
- Sharing between all wavefronts in a SIMD.
- Column sharing on the SIMD.
- Persistent registers.
- LDS – local data share - read/write. These are read/write registers that support sharing between all work-items in a group.
- GDS – global data share. These read/write registers support sharing between all work-items in all groups. Requires synchronization.
- Data sharing between all work-items in a group.
- Required synchronization.
- Memory - read/write.
- Constant buffers
- Texture cache

The following indexing values are available in the compute shader:

- vTid – ID of work-item within a group
- aTid – ID of work-item within a kernel

Chapter 2 Binary Stream Format

The following chapter defines the format in which kernels written using the IL are passed to the compiler.

2.1 IL Stream

Clients pass kernels as a stream of 32-bit tokens organized as variable-length instruction packets. These tokens include information about the client language, shader type, and instruction packets that describe the operation of the kernels. Table 2.1 indicates the ordering of packets.

Table 2.1 IL Stream Instruction Packet Ordering

Instruction Packet	Description
1	IL_Lang token. See Section 2.2.1, on page 2-2.
2	IL_Version token. See Section 2.2.2, on page 2-2.
3	IL_Opcode token describing the operation of the first instruction in the stream and the beginning of the first IL instruction packet.
…	*… More IL instruction packets*. See Chapter 3, “Text Instruction Syntax”.
n (number of 32-bit tokens in the stream)	*IL instruction packet for an* END *instruction*. See page 7-21.

Instruction Packets start out with a special token: ILOpcode. They contain all the information needed to perform the single instruction specified in this token. This information can include data about source and destination operands, destination or target labels, and additional data needed to perform the instruction.

There are assorted IL statements that can be used to declare resources, samplers, or registers. Any declaration of an object must appear before all uses of the object. There is no requirement to group all declarations at the start of the program.

Most IL statements and types can be used in any kind of shader. However, as noted below, some statements and types are restricted to specific kinds of shaders.

2.2 IL Token Descriptions

This section describes the generic tokens used in the IL stream. There are additional tokens for use in single specific instruction packets. Those tokens are described under the instruction packet description for the instruction in which they can be used.

The first two tokens in an IL binary stream are the IL_Lang and IL_Version tokens.

2.2.1 Language Token

This token indicates the type of client generating the IL. This token must be at the beginning of every IL stream passed to the compiler.

Table 2.2 IL_Lang: Source Language Information

Field Name	Bits	Description
client_type	7:0	Specifies the client API. Can be any value of the enumerated type ILLanguage-Type. This value is not used, but can allow IL compilers to make API specific workarounds and optimizations.
reserved	31:8	Must be zero.

2.2.2 Version Token

This token specifies the version of IL used in this IL stream. It also specifies the type of kernel the IL stream represents (pixel or vertex).

Table 2.3 IL_Version: Source Version Information

Field Name	Bits	Description
minor_version	7:0	The minor version.
major_version	15:8	The major version.
shader_type	23:16	Specifies the type of shader described by this binary token stream. See Section 6.25, "ILShader," page 6-19.
multipass	24	0 = Outputs are not ignored. 1 = This shader is for multipass use only (output are ignored).
realtime	25	0 = This shader is not for real-time use. 1 = This shader is for real-time use. This can be set only when the shader type is IL_PIXEL_SHADER.
reserved	31:26	Must be zero.

IL Text combines ther IL_Lang and IL_Version tokens into a single version instruction with the following syntax.

```
il_ps_major_minor
il_vs_major_minor
il_cs_major_minor
il_gs_major_minor
```

2.2.3 Opcode Token

This token specifies the current operation and information required to perform the operation.

Table 2.4 IL_Opcode: Instruction Opcode Details

Field Name	Bits	Description
code	15:0	The operation for the current instruction. This value can be any of the enumerated type ILOpCode (see Section 6.20, “ILOpcode,” page 6-9).
control	29:16	Opcode specific control. Possible values for this depend on the value of *code*. Specifies further instruction behavior. This field must be zero for all instructions not using it.
sec_modifier_present	30	Specifies whether an opcode-specific token describing further instruction behavior follows the primary modifier token. 0 An opcode specific token does not follow. 1 An opcode specific token follows. This field must be zero for all instructions not using it.
pri_modifier_present	31	Specifies whether an opcode-specific token describing further instruction behavior follows this token. 0 An opcode specific token does not follow. 1 An opcode specific token follows. This field must be zero for all instructions not using it.

2.2.4 Destination Token

This token specifies the register to which the hardware passes the result of the current instruction and other information pertaining to this result. This token can only be issued after an IL_Opcode token has been issued as part of an instruction packet. By default, all components of the register specified are written unless the modifier_present field is 1 and an IL_Dst_Mod token follows.

DX10 allows an additional kind of indexing: some temporary objects can be indexed by a register; thus, the IL now allows an additional modifier (register relative modifier). Two kinds of destinations can be indexed: IL_TEMP_ARRAY and IL_OUTPUT.

The immediate_present field is used for indexed data types: itemp, cb, etc. It can be used if the data type uses absolute, reg-relative, loop, or addr relative addressing. See Section 2.2.6, “Source Token,” page 2-5, for examples.

Table 2.5 IL_Dst: Destination Operand Information

Field Name	Bits	Description
register_num	15:0	Register number to which the result of the current instruction is written. See Chapter 5, “Register Types,” for acceptable values.
register_type	21:16	This value can be any of the enumerated type ILRegType (see Section 6.23, “ILRegType,” page 6-18).
modifier_present	22	Specifies whether an IL_Dst_Mod token follows this token: 0 An IL_Dst_Mod token does not follow. 1 An IL_Dst_Mod token follows.

Table 2.5 IL_Dst: Destination Operand Information (Cont.)

Field Name	Bits	Description
relative_address	24:23	Specifies whether register_num represents an offset from the address register specified the following IL_Rel_Addr token: 0 Absolute addressing is used. 1 Relative addressing is used (an IL_Rel_Addr token follows). See Section 2.2.8, on page 2-10. 2 Register relative addressing is used (tokens describing the index follows) Dimension specifies the number of source tokens that need to follow.
dimension	25	Number of additional following dimensions (0 == 1 dimension). so v[3][5] is src_token 1: il_regtype_vertex dim = 1 num = 3 src_token 2: il_regtype_vertex dim = 0 num = 5
immediate_present	26	0 There is no immediate value. 1 A 32-bit value containing the immediate value follows this token, modifier tokens, and src tokens used in register relative addressing.
reserved	27:30	Must be zero.
extended	31	0 No extended register addressing. 1 The register_number is a 32-bit value. The low 16-bits is represented by the register_num field, and following this token is a 32-bit word containing the high 16-bits and 16-bits reserved.

2.2.5 Destination Modifier Token

This token specifies modifications to the destination operand. This token can be issued only immediately after an IL_Dst token and only if that IL_Dst token has the `modifier_present` field set to 1.

Component-wise write masks on destination registers have the following IL Text syntax.

```
reg.{x|_|0|1}{y|_|0|1}{z|_|0|1}{w|_|0|1}
reg.{x|_|0|1}{g|_|0|1}{b|_|0|1}{a|_|0|1}
```

where:

the letter specifies the component to be written,
the underscore specifies components not to be written,
masks 0 and 1 always write into the destination register, even if the instruction normallly does not write that component.

Instruction modifiers are applied after the instruction executes, but before writing the result to the destination register.

Table 2.6 Instruction Modifiers

Modifier	Description	Example
_x2	Shift scale left by 2 modifier.	`add_x2 r0, r1, r2`
_x4	Shift scale left by 4 modifier.	`add_x4 r0, r1, r2`
_x8	Shift scale left by 8 modifier.	`add_x8 r0, r1, r2`
_d2	Shift scale right by 2 modifier.	`add_d2 r0, r1, r2`

Table 2.6 Instruction Modifiers (Cont.)

Modifier	Description	Example
_d4		
_d8		

_sat	Saturate or clamp result to [0,1]	`add_sat r0, r1, r2`
		`add_x2_sat r0, r1, r2`

The modifiers precedence is:

1. shift_scale
2. clamp

An example of all these operations performed on the x-component of a destination operand is:

```
dstCmpMod(clamp(shift_scale(dst.x)))
```

Table 2.7 IL_Dst_Mod: Destination Modification Information

Field Name	Bits	Description
component_x_r	1:0	This value can be any of the enumerated type ILModDstComp[1].
component_y_g	3:2	This value can be any of the enumerated type ILModDstComp[1].
component_z_b	5:4	This value can be any of the enumerated type ILModDstComp[1].
component_w_a	7:6	This value can be any of the enumerated type ILModDstComp[1].
clamp	8	Specifies whether to clamp the value to 0.0 and 1.0: 0 Do not clamp. 1 Clamp. Clamp(NaN) returns 0.
shift_scale	12:9	This value can be any of the enumerated type ILShiftScale[2].
reserved	31:13	Must be zero.

1. See Section 6.18, "ILModDstComponent," page 6-9.
2. See Section 6.26, "ILShiftScale," page 6-19.

Destination scale modifiers can be applied to either float or double operations, but cannot be applied to integer or unsigned operations.

The clamp modifier can be used only with float operands.

2.2.6 Source Token

This token specifies the register that the instruction uses as a source operand. This token can only be issued after an IL_Opcode token as part of an instruction packet. If an IL_Src_Mod token does not follow, then:

- the first component is set to the x component,
- the second component is set to the y component,
- the third component is set to the z component, and
- the fourth component is set to the w component.

Starting with IL 2.0, some source tokens are allowed to use register relative indexing. Only one level of indexing is allowed. The type of register that has indexed sources must be one of the following: IL_REGTYPE_ITEMP, IL_REGTYPE_CONST_BUFF, or IL_REGTYPE_INPUT. Since the index must be a scalar value, a modifier field must be used to replicate a single component into four slots.

Table 2.8 lists and briefly describes the IL_src source operands. IL version 2.0 also allows a source token to refer to a literal defined in a dcl_literal statement. DX10 statements such as:

```
add r1, r2, float4 (1.0f, 2.0f, 3.0f, 4.0f)
```

can be translated into:

```
dcl_literal_float4, l1, 1.0f, 2.0f, 3.0f, 4.0f
```

```
add r1, r2, l1
```

Table 2.8 IL_Src: Source Operand Information

Field Name	Bits	Description
register_num	15:0	Register number from which to retrieve the operand. See Chapter 5, "Register Types," for acceptable values of this field.
register_type	21:16	This value can be any of the enumerated type ILRegType. See Chapter 5, "Register Types," for a description of these types.
modifier_present	22	Specifies whether an IL_Src_Mod token follows this token: 0 An IL_Src_Mod token does not follow. 1 An IL_Src_Mod token follows.
relative_address	24:23	Specifies whether the register_num represents an offset from the address register specified in the following IL_Rel_Addr token or an absolute address: 0 Absolute addressing is used. 1 Relative addressing is used (an IL_Rel_Addr token follows). See Section 2.2.8, on page 2-10. 2 Register relative addressing is used (source tokens follows). Dimension specifies the number of source tokens that need to follow.
dimension	25	Number of additional following dimensions (0 == 1 dimension). so v[3][5] is src_token 1: il_regtype_vertex dim = 1 num = 3 src_token 2: il_regtype_vertex dim = 0 num = 5
immediate_present	26	0 There is no immediate value. 1 A 32-bit value containing the immediate value follows this token, modifier tokens and src tokens used in register relative addressing.
reserved	27:30	Must be zero.
extended	31	0 There is no extended register addressing. 1 The register_number is a 32-bit value. The low 16-bits is represented by the register_num field and following this token is a 32-bit word containing the high 16-bits and 16-bits reserved.

2.2.7 Source Modifier Token

This token specifies modifications to the source operand. It can be issued only immediately following an IL_Src token, and only if the preceding IL_Src token has the modifier_present field set to 1.

When this token is used in conjunction with an IL_Rel_Addr token, modifiers are applied to the register referenced by the relative address, not to the relative-address register itself. Also, this token always precedes the IL_Rel_Addr token if both are used.

Notes about this token:

- If this token is not present, the x, y, z, w corresponds to the first, second, third, and fourth components, respectively.
- For floating point arithmetic instructions, the negate modifier simply flips the sign of the number(s) in the source operand, including on INF values. Applying negate on NaN preserves NaN, although the particular NaN bit pattern that results is not defined.
- For double instructions the negate modifier only modifies the upper half of the double source. It is ignored on the lower half of a double. In the same way, `abs` only modifies the upper half of a double source (changing the sign) without any effect on the lower half.
- For integer instructions, the negate modifier takes the 2's complement of the number(s) in the source operand.
- For floating point arithmetic instructions: the abs modifier simply forces the sign of the number(s) on the source operand positive, including on INF values. Applying abs on NaN preserves NaN, although the NaN bit pattern that results is not defined.
- The 1 swizzle inserts a floating point 1.0f, even if the opcode is an integer operation. This can lead to unexpected results. For example, when evaluating the second source of iadd r1, r1, r1.1_neg(xyzw), implementations take a floating point 1.0f and treat it as an integer. To negate an integer, use the INEGATE function.

- The modifiers use the following precedence:
  1. `swizzle` - rearranges and/or replicates components
  2. `_invert` - inverts components 1 - x
  3. `_bias` - biases components x - 0.5
  4. `_x2` - multiplies components by 2.0
     `_bx2` - signed scaling: combines _bias and _x2 modifiers
  5. `_sign` - signs components: components < 0 become -1; components = 0 become 0; components > 1 become 1
  6. `_divComp(type)` - performs division based on divcomp value; type y, z, w unknown
  7. `_abs` - takes the absolute value of components
  8. `neg(comp)` - provides per-component negate
  9. `clamp` - clamps the value

An example of all of these operations performed on the x-component of a source operand is:

```
clamp(negate (abs (divComp(sign(x2(bias(invert(swizzle(src’s)))))))))
```

The modifiers *bias*, *x2*, *divComp*, and *clamp* cannot be used when the opcode of the instruction specifies an integer or logical operation.

Other examples are:

```
mov r0, r1.zyx1
mov r0, r1.xxyy
add r0, r1_invert, r2
add r0, r1, r2_bias
add r0, r1, r2_x2
add r0, r1, r2_bx2
mov r0, r1_sign
texld\_stage(0) r0, vT0_divcomp(y)
mov r0, r1_abs
mov r0, r1_neg(xw)
```

Table 2.9 IL_Src_Mod: Source Operand Modification Information

Field Name	Bits	Description
swizzle_x_r	2:0	This value can be any of the enumerated type ILComponentSelect[1]. If this token is not present, the IL behaves as if this field is set to IL_COMPSEL_X_R.
negate_x_r	3	Specifies whether to negate the first component: 0 Do not negate. 1 Negate.
swizzle_y_g	6:4	This value can be any of the enumerated type ILComponentSelect[1]. If this token is not present, the IL behaves as if this field is set to IL_COMPSEL_Y_G.
negate_y_g	7	Specifies whether to negate the second component: 0 Do not negate. 1 Negate.

Table 2.9 IL_Src_Mod: Source Operand Modification Information (Cont.)

Field Name	Bits	Description
swizzle_z_b	10:8	This value can be any of the enumerated type ILComponentSelect[1]. If this token is not present, the IL behaves as if this field is set to IL_COMPSEL_Z_B.
negate_z_b	11	Specifies whether to negate the third component: 0 Do not negate. 1 Negate.
swizzle_w_a	14:12	This value can be any of the enumerated type ILComponentSelect[1]. If this token is not present, the IL behaves as if this field is set to IL_COMPSEL_W_A.
negate_w_a	15	Specifies whether to negate the fourth component: 0 Do not negate. 1 Negate.
invert	16	Specifies whether to invert each value of the source register prior to the operation. (s = 1.0 – s). The 'invert' modifier has higher precedence then bias, x2, negate sign, divComp ops, or abs: 0 Do not invert. 1 Invert.
bias	17	Specifies whether to bias each value of the source register prior to the operation. (s = s – 0.5). This takes precedence over x2, negate, sign, divComp ops, and abs: 0 Do not bias. 1 Bias.
x2	18	Specifies whether to multiply each value by 2.0 prior to the operation (s = 2.0 * s). This takes precedence over negate, sign, divComp ops, and abs: 0 Do not perform x2. 1 Perform x2.
sign	19	Specifies whether to sign each value of the source register prior to the operation (s = (s < 0) ? –1 : ((s == 0) ? 0 : 1)). This takes precedence before divComp ops, and abs: 0 Do not sign. 1 Sign.
abs	20	Specifies whether to take the absolute value of each value of the source register prior to the operation. (s = (s < 0) ? –s : s): 0 Do not take the absolute value. 1 Take the absolute value.
divComp	23:21	Specifies component divide modifier. This can be any value of the enumerated type ILDivComp. This takes precedence over abs. See Section 6.6, “ILDivComp,” page 6-3. This field indicates one of the following: • No component divide necessary. • The first component is divided by the second component. • The first and second component is divided by the third component. • The first, second, and third component is divided by the fourth component. Any one of these four are performed based on this field.
clamp	24	Specifies whether or not to clamp the value to [0.0, 1.0]. 0 Do not clamp. 1 Clamp. clamp(NaN) returns 0.
reserved	31:25	

1. See Section 6.4, “ILComponentSelect,” page 6-2.

Table 2.10 shows the relationship between the modifiers and types.

Table 2.10 Modifiers and Types Relationship

Modifier	float	double	int	uint
swizzle	ok	ok	ok	ok
_invert	ok	na	na	na
_bias	ok	na	na	na
_x2	ok	na	na	na
_bx2	ok	na	na	na
_sign	ok	na	na	na
_divcomp(type)	ok	na	na	na
_abs	ok	na	na	na
neg	ok	ok (per component)	ok	ok

2.2.8 Source Token Examples

Some source token encodings can be complicated, as shown in the following examples.

Example 1: *Source Operand:* `x5[6].y`

- IL_Src
  - register_num = 5
  - register_type = IL_REGTYPE_ITEMP (see Section 7.5, “Declaration and Initialization Instructions,” page 7-32)
  - modifier_present = 1
  - relative_address = IL_ADDR_ABSOLUTE
  - dimension = 0
  - immediate_present = 1
  - extended = 0
- IL_Src_Mod
  - swizzle_x_r = y
  - swizzle_y_g = y
  - swizzle_z_b = y
  - swizzle_w_a = y
- IL_Literal
  - val = 6

Example 2: *Source Operand:* `x5[r2.x+6].y`

Here are the IL tokens for this:

- IL_Src x5
  - register_num = 5
  - register_type = IL_REGTYPE_ITEMP
  - modifier_present = 1
  - relative_address = IL_ADDR_REG_RELATIVE
  - dimension = 0
  - immediate_present = 1
  - extended = 0
- IL_Src_Mod
  - swizzle_x_r = y
  - swizzle_y_g = y
  - swizzle_z_b = y
  - swizzle_w_a = y
- IL_Src
  - register_num = 2
  - register_type = IL_REGTYPE_TEMP
  - modifier_present = 1
  - relative_address = IL_ADDR_ABSOLUTE
  - dimension = 0
  - immediate_present = 0
  - extended = 0
- IL_Src_Mod .x for r2
  - swizzle_x_r = x
  - swizzle_y_g = x
  - swizzle_z_b = x
  - swizzle_w_a = x
- IL_Literal
  - val = 6

Example 3: *Source Operand:* `v[1][2]` (all fields set to zero, unless otherwise stated)

- IL_Src
  - register_num = 1
  - register_type = IL_REGTYPE_VERTEX

  - dimension = 1
- IL_Src
  - register_num = 2
  - register_type = IL_REGTYPE_VERTEX

Example 4: *Source Operand:* `v[1][2].xyxx` (all fields set to zero unless otherwise stated)

- IL_Src
  - register_num = 1
  - register_type = IL_REGTYPE_VERTEX
  - dimension = 1
  - modifier_present = 1
- IL_Src_Mod
  - swizzle_x_r = x
  - swizzle_y_g = y
  - swizzle_z_b = x
  - swizzle_w_a = x
- IL_Src
  - register_num = 2
  - register_type = IL_REGTYPE_VERTEX

Example 5: *Source Operand:* `cb[r6.w+2][r2.x+4].y`

There are nine IL tokens.

- IL_Src cb
  - register_num = 0
  - register_type = IL_REGTYPE_CONSTANT BUFFER
  - modifier_present = 1
  - relative_address = IL_ADDR_REG_RELATIVE
  - dimension = 1
  - immediate_present = 1
  - extended = 0
- IL_Src_Mod .y
  - swizzle_x_r = y
  - swizzle_y_g = y
  - swizzle_z_b = y
  - swizzle_w_a = y
- IL_Src r6

- register_num = 6
- register_type = IL_REGTYPE_TEMP
- modifier_present = 1
- relative_address = IL_ADDR_ABSOLUTE
- dimension = 0
- immediate_present = 0
- xxtended = 0

- IL_Src_Mod .w for r6
  - swizzle_x_r = w
  - swizzle_y_g = w
  - swizzle_z_b = w
  - swizzle_w_a = w
- IL_Literal
  - val = 2
- IL_Src cb (only to set dimension and immediate present)
  - register_num = 0
  - register_type = 0
  - modifier_present = 0
  - relative_address = IL_ADDR_REG_RELATIVE
  - dimension = 0
  - immediate_present = 1
  - extended = 0
- IL_Src r2
  - register_num = 2
  - register_type = IL_REG_TYPE_TEMP
  - modifier_present = 1
  - relative_address = IL_ADDR_ABSOLUTE
  - dimension = 0
  - immediate_present = 1
  - extended = 0
- IL_Src_Mod .x for r2
  - swizzle_x_r = x
  - swizzle_y_g = x
  - swizzle_z_b = x
  - swizzle_w_a = x

- IL_Literal
  - val = 4

Chapter 3 Text Instruction Syntax

IL Text syntax is designed to closely match the IL specification, so that there is an almost complete one-to-one mapping.

Below is a simple vertex and pixel shader pair written in IL Text syntax that renders green stripes.

```
il_vs
dclv_elem(0) v0                    ; Declare position
dclv_elem(1) v1                    ; Declare color
dclv_elem(2) v2                    ; Declare texture coordinates

mmul_matrix(4x4) oPos, v0, c[0]    ; Transform position to clip space

mov oPriColor0, v1                 ; Export vertex color
mov oT0, v2                        ; Export texture coordinates

end

il_ps
dclpi_x(1)_y(1)_z(1)_w(1) vPriColor0 ; Declare primary color import
dclpi_x(1)_y(1)_z(*)_w(*) vT0      ; Declare vs import texture
                                     coordinates

def c0, 0.5, 1, 0, 0
def c1, 0.0, 1.0, 0.0, 1.0         ; Green color

mod r0.x, vT0.x, c0.y              ; x = mod( s, 1.0 )

ifc_relop(lt) r0.x, c0.x           ; if ( x < 0.5 )
   mov oC0, vPriColor0             ; Output surface color
else ; else
   mov oC0.rgb1, c1                ; Output green color
endif

end
```

3.1 Version

The first two tokens in an IL binary stream are IL_Lang and IL_Version. IL Text syntax combines these into a single version instruction. The IL translator sets the language type to IL_LANG_VERSION and disables the language defaults. The IL_Version token has the following syntax:

il_ps_*major_minor_mp_rt*
il_vs_*major_minor_mp_rt*

If the major and minor version are not specified, the IL translator inserts them based on its own version. Typically, the version information is omitted, unless a specific optimization made in the compiler is wanted. If a shader represents a multipass shader, append `_mp` to the statement. If this shader represents a real-time shader, append `_rt` to the statement.

3.2 Registers

Registers that are prefixed with the letter "v" are read-only (import) buffers, registers prefixed with the letter "o" are write-only (export) buffers. This section only lists the registers. See Chapter 5 for more information on register types.

Common Registers:

```
b#, c#, i#, a#, aL, and r#
```

Vertex Shader:

```
v#, oPos#, oPriColor#, oSecColor#, oT#, oInterp#, oFog,
oSprite, vBaryCood, vPrimIndex, vQuadIndex
```

Pixel Shader:

```
vPriColor#, vSecColor#, vT#, vInterp#, vFog, vSprite, vFace,
vWinCoord, oC#, oDepth
```

3.3 Control Specifiers

Control specifiers affect the behavior of the instruction. Binary control specifiers, which typically indicate there are two possible actions, have the following syntax:

```
<instr>[_ctrl]
```

If the control specifier `_ctrlspec` is left off the instruction, the default action is performed.

Control specifiers that have more than two modes of operation have the following syntax:

```
instruction_ctrl(value)]
```

For these specifiers, the parenthesis is mandatory, and the value specifies the mode of operation. Sections 3.4, 3.5, 3.6, and 3.7 describe the control specifiers in the IL spec. See Chapter 7, "Instructions," to determine which specifiers go with which instructions.

3.4 Destination Modifiers

Instruction modifiers are applied after the instruction runs but before writing the result to the destination register.

Table 3.1 Destination Modifiers

Modifier	Description	Example
_x2 _x4 _x8 _d2 _d4 _d8	Shift scale modifiers.	`add_x2 r0, r1, r2`

_sat	Saturate or clamp result to [0,1].	`add_sat r0, r1, r2`
		`add_x2_sat r0, r1, r2`

3.5 Write Mask

Component-wise write masks on the destination have the following syntax in IL Text:

```
reg.{x|_|0|1}{y|_|0|1}{z|_|0|1}{w|_|0|1}
reg.{r|_|0|1}{g|_|0|1}{b|_|0|1}{a|_|0|1}
```

A letter specifies a vector component to be written, while an underscore specifies that an component is not written. An component can also be forced to zero or one.

Examples:

```
mov r0.x, r1
mov r0.y r1
mov r0.z r1
mov r0.x_zw, r1
mov r0.y_w, r1
mov r0.__w, r1
mov r0._1_w, r1
mov r0.01z, r1
mov r0.01z1, r1
mov r0.11, r1
mov r0._11, r1
mov r0.__11, r1
mov r0.xyz1, r1
mov r0_1_0, r1
mov r0.11_0, r1
mov r0.y00, r1
mov r0.yz1, r1
mov r0.y_1, r1
```

3.6 Source Modifiers

Table 3.2 Source Modifiers

Modifier	Description	Example
swizzle	Rearrange and/or replicate components.	mov r0, r1.zyx1 mov r0, r1.xxyy

Table 3.2 Source Modifiers (Cont.)

Modifier	Description	Example
`_abs`	Takes the absolute value of components.	`mov r0, r1_abs`
`_bias`	Components are biased (x – 0.5).	`add r0, r1, r2_bias`
`_bx2`	Signed scaling. Combined bias and x2 modifiers.	`add r0, r1, r2_bx2`
`_divcomp(`*type*`)`	Performs division based on *type*. *type*: y, z, w, unknown	`texId_stage(0) r0, vT0_divcomp(y)`
`_invert`	Invert components (1.0 - x).	`add r0, r1_invert, r2`
`_neg(`*comp*`)`	Provides per component negate.	`mov r0 r1_neg(xw)`
`_sign`	Signs Components: Components less then 0 become -1. Components equal to 0 become 0. Components greater then 0 become 1.	`mov r0, r1_sign`
`_x2`	Multiply components by 2.0.	`add r0, r1, r2_x2`

3.7 Comments

Only single-line comments are supported using a semicolon as the delimiter. Other commenting styles, such as /* */ matching pairs, also are supported. A comment can start from any position on the line.

Example:

```
; The following instruction moves the contents of r1 into r0
mov r0, r1 ; mov instruction
```

3.8 Checkerboard Shader Example

The following example is a RenderMan shader for implementing a red checkerboard pattern on a surface.

```
surface
redchecker ( sfreq = 2.0,
             tfreq = 2.0;
             color redcolor = color (1.0, 0.0, 0.0) )
{
    float smod = mod (s* sfreq, 1),
          tmod = mod (t* tfreq, 1);

    if (smod < 0.5) {
        if (tmod < 0.5)
            Ci = Cs;           // surface color
        else
            Ci = redcolor;     // red color
    }
    else {
        if (tmod < 0.5)
            Ci = redcolor;     // red color
        else
            Ci = Cs;           // surface color
    }
}
```

The corresponding IL text example of a vertex and pixel shader pair that implements the same red checkerboard pattern is shown below. Note that this example purposely does not use the IL `mod` instruction in order to show the syntax for the `call` instruction.

```
il_vs
dclv_elem(0) v0                   ; Declare position
dclv_elem(1) v1                   ; Declare color
dclv_elem(2) v2                   ; Declare texture coordinates

mmul_matrix(4x4) oPos, v0, c[0]   ; Xform position to clip space

mov oPriColor0, v1
mov oT0, v2

end

il_ps
dclpi_x(1)_y(1)_z(1)_w(1) vPriColor0 ; Declare vs import color
dclpi_x(1)_y(1)_z(*)_w(*) vT0        ; Declare vs import texture
coordinates

def c0, 0, 1, 0.5, 0
def c1, 2.0, 2.0, 0.0, 0.0           ; tfreq, sfreq
def c2, 1.0, 0.0, 0.0, 0.0           ; red checker color

; Perform mod( s * sfreq, 1 )
mul r10.x, vT0.x, c1.x               ; a = s * sfreq
mov r10.y, c0.y                      ; b = 1
call 1
mov r1.x, r11.x                      ; smod = mod( a, b )

; Perform mod( t * tfreq, 1 )
mul r10.x, vT0.y, c1.y            ; a = t * tfreq
mov r10.y, c0.y                   ; b = 1
call 1
mov r2.x, r11.x                   ; tmod = mod( a, b )

ifc_relop(lt) r1.x, c0.z          ; if ( smod < 0.5 ) {
    ifc_relop(lt) r2.x, c0.z      ;  if ( tmod < 0.5 )
        mov r0, vPriColor0        ;         Ci = Cs
    else                          ;  else
        mov r0, c2                ;         Ci = redcolor
    endif                         ; }
else                              ; else {
    ifc_relop(lt) r2.x, c0.z      ;  if ( tmod < 0.5 )
        mov r0, vPriColor0        ;         Ci = Cs
    else                          ;  else
        mov r0, c2                ;         Ci = redcolor
    endif                         ;
endif                             ; }

mov oC0, r0.rgb1                  ; ensure alpha is 1
```

```
endmain                 ; seperator between shader and subroutines.


; float c = mod( float a, float b )
;    r10.x is argument a
;    r10.y is argument b
;    r11.x is return value c
; Reference: Ebert et al. Texturing and Modeling, pg. 28
; Translated from DX shader written by D. Mooney

func 1
   rcp r12.y, r10.y             ; ( 1 / b )
   mul r12.x, r10.x, r12.y      ; ( a / b )
   frc r12.z, r12.x
   sub r12.x, r12.x, r12.z      ; n = (int)(a/b)
   mul r12.x, r12.x, r10.y      ; n*b
   sub r10.x, r10.x, r12.x      ; a -= (n*b)

   ifc_relop(lt) r10.x, c0.x    ; if ( a < 0 )
      add r10.x, r10.x, r10.y   ;      a += b
   endif

   mov r11.x, r10.x             ; return a;
ret

end
```

Chapter 4 Shader Operations

This section describes the shader operations.

4.1 Shader Requirements

The following is a list of requirements for a shader to be acceptable as an IL input:

- Pixel shaders must write to a PCOLOR or DEPTH register (i.e., must export a color or depth value) unless the shader is multipass.
- Vertex shaders must write to the POS register or write to an VOUTPUT defined as having usage IL_IMPORTUSAGE_POS (i.e., must export a position) unless the shader is multipass.
- The shader must begin with a Language token immediately followed by a Version token.
- There can be only one END instruction in a shader.
- The END instruction must be the last instruction in the shader (therefore it cannot be within a *flow-control-block*).
- The ENDMAIN instruction must be used before any subroutines are defined. (ENDMAIN is only required if functions are defined).
- All instruction after and ENDMAIN instruction except for the END instruction must be in a *subroutine*.
- Loop relative addressing can be used only on PINPUT, VOUTPUT, VERTEX, INTERP, or TEXCOORD registers.
- Base relative addressing can only be used on CONST_FLOAT registers.
- A DCLARRAY instruction must be issued to use a range of INTERP or TEXCOORD registers with loop relative addressing.
- An INITV or DCLV instruction must be issued on a VERTEX register before it is used as a source any other instruction.
- A DCLPI instruction must be issued on each INTERP, TEXCOORD, PRICOLOR, SECCOLOR, FOG, and WINCOORD register used before it is used in a regular pixel shader.
- A shader can use more than one unique constant in an instruction and can also use more than one different constant in a single instruction. For example, `r0 = c0 + c0` is legal; `r0 = c0 + c1` is also legal.

- A pixel shader can use an INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG register even if it was not written to in the vertex shader. The value of these register depends on the DCLPI instruction.
- Must be able to nest CALL and CALLNZ up to four levels.
- Must be able to nest LOOP-ENDLOOP up to four levels.
- Must be able to nest IFNZ-ELSE-ENDIF and IFC-ELSE-ENDIF up to 24 levels.
- A DCLPP instruction must be issued on each PINPUT register used before it is used in a real-time pixel shader.
- A real-time pixel shader follows all of the same rules of a normal pixel shader with the following exceptions:
  - The DCLPI and DCLPIN instructions cannot be used. Instead, a DCLPP instruction must be used.
  - WINCOORD, SPRITECOORD, and FACE registers cannot be used.
  - A DCLPT instruction must be issued for each texture stage before TEXLD, TEXLDD, TEXLDB, TEXLDMS, TEXWEIGHT, PROJECT, or LOD is used on the stage.
  - At least one DCLPIN instruction must be issued on each PINPUT register before it is used.
  - At least one DCLVOUT instruction must be issued on each VOUTPUT register before it is used.

4.2 Link Restrictions

- A vertex shader using an VOUTPUT register cannot link with a pixel shader using a INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG registers. It can only link with a pixel shader using PINPUT registers.
- A pixel shader using an PINPUT register cannot link with a vertex shader using POS, SPRITE, INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG registers. It can only link with a vertex shader using VOUTPUT registers.
- If multiple usage-usageIndex pairs are packed in a single VOUTPUT register in a vertex shader, they must also be packed in a single PINPUT register in the linked pixel shader.
- If multiple usage-usageIndex pairs are packed in a single PINPUT register in a pixel shader, they must also be packed in a single VOUTPUT register in the linked vertex shader.

4.3 Multi-Pass Shaders

Multipass shaders are shaders where outputs are ignored. A shader is multi-pass if the *multipass* bit in the IL_Version token is set to 1. In a multipass vertex shader, writes to the INTERP, TEXCOORD, PRICOLOR, SECCOLOR, FOG, SPRITE, POS, and VOUTPUT registers are ignored. As well, writes to the POS register or an VOUTPUT declared as having usage IL_IMPORTUSAGE_POS is not required (i.e. exporting a position is not required and any exports to position are ignored). In a pixel shader, writes to the PCOLOR and DEPTH registers are ignored. They are also not required.

4.4 Real-Time Shaders

A pixel shader is a real-time shader if the *realtime* bit in the IL_Version token is set to 1. A real-time pixel shader follows all of the same rules of a normal pixel shader with the following exceptions:

- The DCLPI and DCLPIN instructions cannot be used. Instead, DCLPP must be used.
- WINCOORD, SPRITECOORD, and FACE registers cannot be used.
- INTERP, TEXCOORD, PRICOLOR, SECCOLOR, and FOG registers also cannot be used.

A vertex shader cannot be real-time. Therefore if the *shader_type* field of the IL_Version token is set to IL_SHADER_VERTEX the *realtime* bit must be set to 0.

Chapter 5 Register Types

This chapter describes the AMD IL valid register types.

Tables 5.1 through 5.4 show the mappings of DX register types to IL types.

Table 5.1 Registers Mapping: DX10 to IL

DX10	IL Regtype	GS Declaration	VS Declaration	PS Declaration
Temp r#	temp	-- by use --	-- by use --	-- by use --
Indexed temp x#	itemp	dcl_indexableTemp	dcl_indexableTemp	dcl_indexableTemp
Input register v#	Input	dcl_input	dcl_input	dcl_input
Output register o#	Output	dcl_output	dcl_output	NA
Input resource t#		dcl_resource	dcl_resource	dcl_resource
Sampler s#		-- by use ---	-- by use ---	-- by use --
Constant Buffer	cb	dcl_cb	dcl_cb	dcl_cb
Literal f4	literal	literal	literal	literal
Literal I4	literal4	literal	literal	literal
Literal f1	literalf1	literal	literal	literal
Literal i1	literal1	literal	literal	literal
vPrim	PRIMITIVE_INDEX	dcl_input_sv	NA	NA
VertexID (semantic) in		dcl_input	dcl_input_sv	NA
PrimitiveID (semantic) in		dcl_input_sv	dcl_input_sv	dcl_input
InstanceID (semantic) in		dcl_input	dcl_input_sv	dcl_input
IsFrontFace semantic) in		NA	NA	dcl_input_sv
ClipDistance		dcl_output_sv	dcl_input_sv or dcl_output_sv	dcl_input_sv
CullDistance		dcl_output_sv	dcl_input_sv or dcl_output_sv	dcl_input_sv
Position		dcl_output_sv	dcl_input_sv or dcl_output_sv	dcl_input_sv
RenderTargetArrayIndex		NA	dcl_output_sv	dcl_input_sv
ViewPortArrayIndex		NA	dcl_output_sv	dcl_input_sv

Table 5.1 Registers Mapping: DX10 to IL (Cont.)

DX10	IL Regtype	GS Declaration	VS Declaration	PS Declaration
vCoverageMask		NA	dcl_output_sv	by use
oDepthLE		NA	NA	by use
oDepthGE		NA	NA	by use

Table 5.2 Mapping of DX10 Declaration Information to IL

DX10	IL Regtype_
Input primitive	dcl_input_primitive
Maxoutputvertexcount	dcl_max_output
Topology	dcl_out_topology

Table 5.3 Mapping of DX9 (3.0) Registers to IL Types

DX9 (3.0)	IL Type	Vertex Initialization	Pixel Initialization
V# in vertex shader	vextex	Dclv	Na
Temps	temp	-- by use --	-- by use --
Constants	float	dcldef	dcldef
Address register a0	addr	-- by use --	
Boolean constants	const_bool	Defb	Defb
Integer constants	const_int	def	def
Loop counter Al	Loop relative addr	-- by use --	-- by use --
Predicate register	Temp	Na	Na
Sampler	- stage number -	Na	dclpt
O# in vertex shader	voutout	dclvout	
Face	Face	Na	-- by use --
vPos	Wincord	Na	-- by use --
V# in pixel shader	pinput	Na	dclpin
Oc#	pcolor	Na	-- by use --
oDepth	Depth	Na	-- by use --

Table 5.4 Mapping of DX9 (2.0 and Lower) Registers to IL Types

DX9 (2.0 and Lower)	IL Type	Vertex Initialization	Pixel Initialization
V# in vertex shader	vertex	Dclv or initv or -- by use --	NA
Temps	temp	-- by use --	-- by use --
Constants	float	dcldef	dcldef
Address register a0	addr	-- by use --	NA
Boolean constants	const_bool	Defb	Defb
Integer constants	const_int	def	Def
Loop counter Al	Loop relative addr	-- by use --	NA
oPos	Pos	-- by use --	NA
oFog	Fog	-- by use --	NA
oPts	Sprite	-- by use --	NA
oD0/oD1 color	Pricolor Seccolor	-- by use --	NA
oT#	Texcoord	-- by use --	NA
V0/v1 Color register	Pricolor Seccolor	NA	Dclpi
Predicate register	Temp	NA	Na
Sampler	- stage number -	NA	dclpt
T#	Texcoord	NA	dclpi
Oc#	pcolor	NA	-- by use --
oDepth	Depth	NA	-- by use --
Window coord	wincoord	NA	Dclpi

Table 5.5 Special HOS-Related Fields

Indexing Mode	Application
index	--by use--
object_index	--by use--
Barycentric coord	--by use--
Primitive index	--by use--
Quad index	--by use--

Table 5.6 Mapping of DX10 and DX11 Compute Shader (CS) Registers to IL Types

DX10 and DX11 CS Type	IL Type	Initialization
vThreadId	Absolute_Thread_Id	by use, do not need Dcls
vThreadIdInGroup	Thread_Id_In_Group	by use, do not need Dcls
vThreadIdInGroupFlattened	Thread_Id_In_Group_Flattened	by use, do not need Dcls
vThreadGroupId	Thread_Group_Id	by use, do not need Dcls

Table 5.7 Registers and Their Restrictions

Name	IL Regtype_	Syntax	Com-po-nents	Read	Write	Relative Address			Notes
						Loop	A0	Index	
Addr	ADDR	A0	4	Yes	Unsup-ported	No	No	No	
Boolean	CONST_BOOL	b#	1	No	Defb only	No	No	No	Can be used only as the source for static control flow.
Float	CONST_FLOAT	c#	4	Yes	Def only	No	Yes	No	
Index	INDEX	vINDEX	4	Yes					
Input	INPUT	v#	4	Yes	No	No	No	No	To support DX10.
Int	CONST_INT	I#	4	Loop only	Def only	No	No	No	
Output	OUTPUT	o#	4	No	Yes	No	No	No	To support DX10.
Prim id	VPRIM	vPrim	1	Yes	No	No	No	No	Can be used in all shader types except vertex.
Timer	INPUT	Tmr	2	Yes	No	No	No	No	To support DX11.
Vertex	VERTEX	V#	4	Yes	Dclv or initv	No	No	No	Cannot be used in pixel shader.

Table 5.8 IL Register Types Overview

Register	Read/Write	Syntax	Components
IL_REGTYPE_ABSOLUTE_THREAD_ID	r	vAbsTid	3
IL_REGTYPE_ABSOLUTE_THREAD_ID_FLAT	r	vAbsTidFlat (vaTid is deprecated)	1
IL_REGTYPE_ADDR	w	a0	1
IL_REGTYPE_BARYCENTRIC_COORD	r	vBaryCoord	4
IL_REGTYPE_CLIP	w	oClip#	1
IL_REGTYPE_CONST_BOOL	r	b#	1
IL_REGTYPE_CONST_BUFF	r	cb#[n]	4
IL_REGTYPE_CONST_FLOAT	r	c#	4
IL_REGTYPE_CONST_INT	r	i#	3
IL_REGTYPE_DEPTH	w	oDepth	1
IL_REGTYPE_DEPTH_GE	w	oDepthGE	1
IL_REGTYPE_DEPTH_LE	w	oDepthLE	1
IL_REGTYPE_DOMAINLOCATION	r	vDomain	4
IL_REGTYPE_EDGEFLAG	w	oEdgeFlag]	1
IL_REGTYPE_FACE	r	vFace	1
IL_REGTYPE_FOG	r/w	oFog	1
IL_REGTYPE_GENERIC_MEM	r/w	mem	4
IL_REGTYPE_GLOBAL	r/w	g[addr]	4
IL_REGTYPE_IMMED_CONST_BUFF	r	icb[n]	4
IL_REGTYPE_INDEX	r	vIndex#	4
IL_REGTYPE_INPUT	r	v#[n]	4
IL_REGTYPE_LINE_STIPPLE	r	vLineStipple	2
IL_REGTYPE_INPUT_ARG	r/w	in#	4
IL_REGTYPE_INPUT_COVERAGE_MASK	w	vCoverageMask	1
IL_REGTYPE_INPUTCP	r	vicp[vertex#][attr#]	4
IL_REGTYPE_INTERP	w r	oInterp# vInterp#	4
IL_REGTYPE_ITEMP	r/w	x#[n]	4
IL_REGTYPE_LITERAL	r	l#	4
IL_REGTYPE_OBJECT_INDEX	r	vObjIndex	1
IL_REGTYPE_OCP_ID	w	voutputcontrolpointid	4
IL_REGTYPE_OMASK	w	oMask	1
IL_REGTYPE_OUTPUT	w	o#[n]	4
IL_REGTYPE_OUTPUT_ARG	r/w	out#	4
IL_REGTYPE_OUTPUTCP	w	vocp	1
IL_REGTYPE_PATCHCONST	r	vpc[id#]	4
IL_REGTYPE_PCOLOR	w	oC#	4
IL_REGTYPE_PERSIST	r/w	p[addr]	4
IL_REGTYPE_PINPUT	r/w	vPixIn#	4

Table 5.8 IL Register Types Overview (Cont.)

Register	Read/Write	Syntax	Components
IL_REGTYPE_POS	w	oPos	4
IL_REGTYPE_PRICOLOR	r w	oPriColor# vPriColor#	4
IL_REGTYPE_PRIMCOORD	r	vPrimCoord	2
IL_REGTYPE_PRIMITIVE_INDEX	r	vPrimIndex	4
IL_REGTYPE_PRIMTYPE	r	vPrimType	2
IL_REGTYPE_PS_OUT_FOG	w	oPsFog#	1
IL_REGTYPE_QUAD_INDEX	r	vQuadIndex	4
IL_REGTYPE_SECCOLOR	r w	vSecColor# oSecColor#	4
IL_REGTYPE_SHADER_INSTANCE_ID	r	vInstanceID	1
IL_REGTYPE_SHARED_TEMP	r/w	sr	4
IL_REGTYPE_SPRITE	w	oSprite	1
IL_REGTYPE_SPRITECOORD	r	vSpriteCoord	2
IL_REGTYPE_STENCIL	w	oSTENCIL	1
IL_REGTYPE_TEMP	r/w	r#	4
IL_REGTYPE_TEXCOORD	w r	oT# vT#	4
IL_REGTYPE_THIS	r	this	4
IL_REGTYPE_THREAD_GROUP_ID	r	vThreadGrpID	3
IL_REGTYPE_THREAD_GROUP_ID_FLAT	r	vThreadGrpIDFlat	1
IL_REGTYPE_THREAD_ID_IN_GROUP	r	vTidInGrp	3
IL_REGTYPE_THREAD_ID_IN_GROUP_FLAT	r	vTidInGrpFlat (vTid is deprecated)	1
IL_REGTYPE_TIMER	r	Tmr	2
IL_REGTYPE_VERTEX	r	v#	4
IL_REGTYPE_VOUTPUT	w	oVtxOut	4
IL_REGTYPE_VPRIM	r/w	vPrim	4
IL_REGTYPE_WINCOORD	r	vWinCoord	4

5.1 ABSOLUTE_THREAD_ID

Enum: IL_REGTYPE_ABSOLUTE_THREAD_ID

Text Syntax: `vAbsTid`

Components per Register: 3

Description:

This read-only input register contains an absolute work-item ID. The ID is three-dimensional. It is used only in compute shaders. The xyz components of the register can be used as an index or in integer operations. The w component is not valid and must not be used. This register is used only in compute shaders.

Valid in R7XX GPUs and later.

Example:

```
mov r2.x,  vAbsTid.xyzx
mov g[vAbsTid.x], r2
```

5.2 ABSOLUTE_THREAD_ID_FLATTENED

Enum: IL_REGTYPE_ABSOLUTE_THREAD_ID_FLAT

Text Syntax: `vAbsTidFlat` (also as `vaTid` for back-compatibility)

Components per Register: 1

Description:

This read-only input register contains the flattened absolute work-item ID.

Assuming the work-group size is (Dx, Dy, Dz), the flattened value is computed as

```
    vAbsTidFlat.x = vThreadGrpIdFlat.x*Dx*Dy*Dz + vTidInGrpFlat.x
```

This register can be used as an index or in integer operations. Only the x component has a meaningful value. The y, z, and w components replicate the value of the x component. This is used only in a compute shader.

Valid in R7XX GPUs and later.

Example:

```
    mov g[vAbsTidFlat.x], r2
```

5.3 BARYCENTRIC_COORD

Enum: IL_REGTYPE_BARYCENTRIC_COORD

Text Syntax: `vBaryCoord`

Components per Register: 4

Description:

This register type is valid only for HOS rendering. For non-HOS (higher-order surface) rendering, this register type is invalid and its contents are undefined. For HOS rendering, this is a vertex shader import for the barycentric coordinates of the current, tessellated vertex.

This read-only register cannot be used with relative addressing.

It is an error to use this register in a pixel shader.

5.4 CONST_BUFF

Enum: IL_REGTYPE_CONST_BUFF

Text Syntax: `cb#[`*n*`]`

Components per Register: 4

Description:

Read-only register with a maximum of 4096 elements.

5.5 DEPTH

Enum: IL_REGTYPE_DEPTH

Text Syntax: `oDepth`

Components per Register: 1

Description:

Pixel shader export for depth data. This is a scalar register where the depth values is contained in the first component. This write-only register cannot be the source of an instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing. The second, third, and fourth components of this register are unused and undefined. DEPTH, DEPTHLE, and DEPTHGE are mutually exclusive; use at most one of these in a shader.

5.6 DEPTH_GE

Enum: IL_REGTYPE_DEPTH_GE

Text Syntax: `oDepthGE`

Components per Register: 1

Description:

Pixel shader export for depth data, guaranteed to be greater than or equal to rasterizer depth value. This is a scalar register where the depth value is contained in the first component. This write-only register cannot be the source of any instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing. The second, third, and fourth components of this register are unused and undefined. DEPTH, DEPTHLE and DEPTHGE are mutually exclusive, at most one of them can be used by a shader.

If rasterizer depth is not declared in the shader, its interpolation mode is set to sample if shader runs at sample rate (the shader declares a sample index or sample attributes); otherwise, centroid interpolation mode is used.

Valid in Evergreen GPUs and later.

5.7 DEPTH_LE

Enum: IL_REGTYPE_DEPTH_LE

Text Syntax: `oDepthLE`

Components per Register: 1

Description:

Pixel shader export for depth data, guaranteed to be less than, or equal to, rasterizer depth value. This scalar register's depth value is in the first component. This write-only register cannot be the source of any instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing. The second, third, and fourth components of this register are unused and undefined. DEPTH, DEPTHLE and DEPTHGE are mutually exclusive; use at most one of these in a shader.

If rasterizer depth is not declared in the shader, its interpolation mode is set to `sample` if the shader runs at the sample rate (if the shader declares `sample index` or `sample attributes`); otherwise, the centroid interpolation mode is used.

Valid in Evergreen GPUs and later.

5.8 DOMAINLOCATION

Enum: IL_REGTYPE_DOMAINLOCATION

Text Syntax: `vDomain`

Components per Register: 4

Description:

This read-only register is used in the domain shader as input only.

Example:

```
mov r1, vDomain
```

5.9 EDGEFLAG

Enum: IL_REGTYPE_EDGEFLAG

Text Syntax: `oEdgeFlag`

Components per Register: 1

Description:

This is the vertex shader export for an edge flag. The first channel contains the edge flag. This write-only register cannot be the source of any instruction. This register cannot be used with relative addressing. It is an error to use this register if VOUTPUT register is used. It is an error to use this register in a pixel shader. The y, z, and w components of this register are undefined.

5.10 FACE

Enum: IL_REGTYPE_FACE

Text Syntax: `vFace`

Components per Register: 1

Description:

Pixel shader import for primitive facing. The x component is negative if the pixel is the back-face of the primitive. The x component is positive if the pixel is the front-face of the primitive. Point and Line primitives are always front-facing. Points and lines rendered as a result of polygons using point or line fill mode inherit the facing of the polygon. This i read-only register cannot be the destination of an instruction. It is an error to use this register in a vertex shader. It is an error to use this register in a real-time pixel shader. This register cannot be used with relative addressing. The second, third, and fourth components of this register are undefined.

5.11 FOG

Enum: IL_REGTYPE_FOG

Text Syntax (VS): `oFog` (write-only)

Text Syntax (PS): `oFog` (read-only)

Components per Register: 1

Description:

Vertex shader export and pixel shader import for interpolated fog data. This is a scalar register where the value is contained in the first component. In a vertex shader, the second, third, and fourth components must be masked (cannot be written to). In a pixel shader the second, third, and fourth components are undefined. Perspective correct interpolation is performed on the values of this registers when passed from the vertex shader to the pixel shader. Can only use a total of 16 of INTERP, TEXCOORD, PRICOLOR, SECCOLOR, and FOG registers in a single shader.

A DCLPI instruction must be issued on this shader type before is it used in any other instruction. This register cannot be used with relative addressing.

It is an error to use a register of this type in a vertex shader if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_FOG.

It is an error to use a register of this type in a pixel shader if a PINPUT register is used. You can achieve similar functionality by using the DCLPIN instruction on a PINPUT register and declaring its usage as IL_IMPORTUSAGE_FOG.

In a vertex shader, this is a write-only register. It cannot be the source of an instruction.

In a pixel shader, this is a read-only register. It cannot be the destination of any instruction.

5.12 GENERIC_MEMORY

Enum: IL_REGTYPE_GENERIC_MEM

Text Syntax: `mem`

Components per Register: 4

Description:

This is register provides a mask or swizzle. It is used by instructions such as `write_lds_`, which does not have a `dst` register, but requires a `dst` mask.

Valid in Evergreen GPUs and later.

Example:

```
write_lds mem.x_z_, r0.xyzw
```

5.13 GLOBAL

Enum: IL_REGTYPE_GLOBAL

Text Syntax: `g[address]`
Note that the address must be a single-component integer.

Components per Register: 4

Description:

This is a read/write register that can be used to address global memory in the text form of the shader. Global variables are indicated by a `g[address]`.

Example:

```
add g[2].x, r4, g[4]
```

reads address 4, adds r4 to the contents of that address, and scatters the result to address 2. Each address corresponds to a 128-bit, four dword location. Most address locations are indexed.

Global g registers can be indexed using the temp (r) registers. For example,
`add g [r5.x].x r4, g[r6.y]`

5.14 IMMED_CONST_BUFFER

Enum: IL_REGTYPE_IMMED_CONST_BUFF

Text Syntax: `icb[n]`

Components per Register: 4

Description:

Read-only register. Used to access the immediate constant buffer. Used in the same way as cb[n] for accessing a constant buffer.

5.15 INDEX

Enum: IL_REGTYPE_INDEX

Text Syntax: `vIndex`

Components per Register: 4

Description:

Vertex shader import for the index from the index buffer of the current vertex processed. Not guaranteed to be incremental. For non-HOS (high-order surface) rendering, the first component is the index of the current vertex processed. For HOS rendering (when HOS is enabled) the first, second, third, and fourth components represent the indices for the superprim vertices for the current tessellated vertex being processed. The superprim indices are output by the tessellation engine based on the relevant HOS state. This read-only register cannot be used with relative addressing. It is an error to use this register in a pixel shader.

5.16 INPUT

Enum: IL_REGTYPE_INPUT

Text Syntax: `v#[n]`

Components per Register: 4

Description:

This read-only register is the formal parameter of a macro. The register can be used only within a macro definition. When a macro is called, the actual parameters are copied into the macro input registers. When the macro returns, the registers are restored.

5.17 INPUT_ARG

Enum: IL_REGTYPE_INPUT_ARG

Text Syntax: `in#`

Components per Register: 4

Description:

This read-write register is the formal parameter of a macro. It can be used only within a macro definition. When a macro is called, the actual parameters are copied into the macro input registers. When the macro returns, the registers are restored.

5.18 INPUT_COVERAGE_MASK

Enum: IL_REGTYPE_INPUT_COVERAGE_MASK

Text Syntax: `vCoverageMask`

Components per Register: 1

Description:

Pixel shader input coverage mask.

This is a scalar register where coverage mask is contained in the 1st component. It is an error to use this register outside of pixel shader. This register cannot be used with relative addressing. The 2nd, 3rd, and 4th components of this register are unused and undefined. Input coverage mask is a bitfield, where bit i from the LSB indicates (with 1) if the current primitive covers sample i in the current pixel on the RenderTarget. Regardless of whether the Pixel Shader is configured to be invoked at pixel frequency or sample frequency, the first n bits in input coverage mask from the LSB are used to indicate primitive coverage, given an n sample per pixel RenderTarget and/or Depth/Stencil buffer is bound at the Output Merger. The rest of the bits are 0. The input coverage bitfield is not affected by depth/stencil tests, but it is ANDed with the SampleMask Rasterizer state. If no samples are covered, such as on helper pixels executed of the bounds of a primitive to fill out 2x2 pixel stamps, input coverage mask is 0.

Supported on Evergreen GPUs and later.

5.19 INPUTCP

Enum: IL_REGTYPE_INPUTCP

Text Syntax: `vicp[vertex#][attr#]`

Components per Register: 4

Description:

Used in the hull shader and domain shader as input only. The `vertex#` is between 0 and 31.

Valid in Evergreen GPUs and later.

Example:

```
mov r1, vicp[5][2]
```

5.20 INTERP

Enum: IL_REGTYPE_INTERP

Text Syntax (VS): `oInterp#` (write-only)

Text Syntax (PS): `vInterp#` (read-only)

Components per Register: 4

Description:

General-purpose vertex shader export and pixel shader import for interpolated data. Perspective correct interpolation is performed on the values of these registers when passed from the vertex shader to the pixel shader. A DCLPI instruction must be issued on this shader type before is it used in any other instruction.

It is an error to use a register of this type in a vertex shader if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on an VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_GENERIC.

It is an error to use a register of this type in a pixel shader if a PINPUT register is used. You can achieve similar functionality by using the DCLPIN instruction on a PINPUT register and declaring its usage as IL_IMPORTUSAGE_GENERIC.

This register can be used only with loop relative addressing. In a vertex shader, this write-only register cannot be the source of an instruction. In a pixel shader, this read-only register cannot be the destination of an instruction.

5.21 ITEMP

Enum: IL_REGTYPE_INDEXED_TEMP

Text Syntax: `x#[n]`

Components per Register: 4

Description:

Read-write register. See `DCL_INDEXED_TEMP_ARRAY` (page 7-34) to see how to declare indexed temps. x#[n] can be used in any ALU instruction.

5.22 LINE_STIPPLE

Enum: IL_REGTYPE_LINE_STIPPLE

Text Syntax: `vLineStipple`

Components per Register: 2

Description:

Pixel shader input for the Line Stipple Texture Coord. SPI calculates the 32-bit line stipple texture coordinate and stores it in the position buffer.

```
X = 32b tex coord
Y = prim type (POINT = 0, LINE = 1, TRI = 2)
```

Supported on Evergreen and later GPUs.

5.23 LITERAL

Enum: IL_REGTYPE_LITERAL

Text Syntax: `l#`   where `#` is the literal register number.

Components per Register: 4

Description:

This four-components, constant, typeless, read-only register can be used in place of a GPR. The format of this register can be either integer, floating point, or four-byte hex value. A given literal can only be defined once in a shader.

5.24 OBJECT_INDEX

Enum: IL_REGTYPE_OBJECT_INDEX

Text Syntax: `vObjIndex`

Components per Register: 1

Description:

Vertex shader import for the ordered index for the current vertex processed (ordered vertex shader instance). Pixel shader import for the ordered index for the current pixel processed (ordered pixel shader instance). This value starts at 0 and is incremented for each successive pixel/vertex. The first component of this register contains the current vertex processed. This read-only register cannot be used with relative addressing. It is an error to use the second, third, and fourth components register.

5.25 OCP_ID

Enum: IL_REGTYPE_OCP_ID

Text Syntax: `vOutputControlPointID0` (only 0 is allowed)

Components per Register: 4

Description:

Output Control Point ID used in the hull shader as input only. Read-only register.

5.26 OMASK

Enum: IL_REGTYPE_OMASK

Text Syntax: `oMask`

Components per Register: 1

Description:

When the pixel shader runs at sample-frequency, the coverage mask is ANDed with a mask that selects the sample currently being processed. As a result, sample N is always masked by bit N of oMask. This allows a shader to run at either sample-frequency or pixel-frequency with identical oMask behavior. The same rule applies to Alpha and to Coverage when the shader runs at sample-frequency.

This is a scalar register where the mask value is contained in the first component. Values assigned to oMask are treated as integer. This write-only register cannot be the source of an instruction and cannot be used with relative addressing. It is an error to use this register in a vertex shader. The second, third, and fourth components of this register are unused and undefined.

5.27 OUTPUT

Enum: IL_REGTYPE_OUTPUT

Text Syntax: o#[*n*]

Components per Register: 4

Description:

This is an output from the shader. See DCL_OUTPUT (page 7-45) for more detail.

5.28 OUTPUT_ARG

Enum: IL_REGTYPE_OUTPUT_ARG

Text Syntax: `out#`

Components per Register: 4

Description:

This write-only register is the formal output parameter of a macro. It can be used only within a macro definition. When a macro returns the values in the macro output registers, they are copied to the actual arguments.

5.29 OUTPUT_CONTROL_POINT

Enum: IL_REGTYPE_OUTPUTCP

Text Syntax: `vocp[#][#]`

Components per Register: 1

Description:

Output Control Point used in the hull shader as input only. Read-only register.

5.30 PATCH_CONST

Enum: IL_REGTYPE_PATCHCONST

Text Syntax: `vpc[`*`id#`*`]`

Components per Register: 4

Description:

This is used in the domain shader as input only. The `id#` is between 0 and 31.

Example:

```
mov r1, vpc[5]
```

5.31 PCOLOR

Enum: IL_REGTYPE_PCOLOR

Text Syntax: `oC#`

Components per Register: 4

Description:

Pixel shader export for color data. The register number corresponds to the color buffer to which data is output. This write-only register cannot be the source of an instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing.

5.32 PERSIST

Enum: IL_REGTYPE_PERSIST

Text Syntax: `p[`*`address`*`]`

Components per Register: 4

Description:

This read/write register addresses persistent memory.

5.33 PINPUT

Enum: IL_REGTYPE_PINPUT

Text Syntax: `vPixIn#`

Components per Register: 4

Number Per Shader: 16

Pixel shader input data. It is an error to use this register in a vertex shader. This read and write register can be used with loop-relative addressing as a source only. A DCLPIN or DCLPP instruction must be issued on a register of this type before it is used. It is an error to use this register if an INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG register is used.

5.34 POS

Enum: IL_REGTYPE_POS

Text Syntax: `oPos`

Components per Register: 4

Description:

Vertex shader export for position data. The position of the vertex in clip space. This write-only register cannot be the source of an instruction. It is an error to use this register if a VOUTPUT register is used. Similar functionality is provided by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_POS. This register cannot be used with relative addressing. It is an error to use this register in a pixel shader.

5.35 PRICOLOR

Enum: IL_REGTYPE_PRICOLOR

Text Syntax (VS): `oPriColor#` (write-only)

Text Syntax (PS): `vPriColor#` (read-only)

Components per Register: 4

Description:

Vertex shader export and pixel shader import for interpolated primary color data. By convention, register number 0 represents the front-facing color, while register number 1 represents the back-facing color. When flat shading is used, no interpolation is performed on the values of these registers when passed from the vertex shader to the pixel shader. Instead, only the value of the provoking vertex is passed to the pixel shader. When smooth shading is used, perspective correct interpolation is performed on the values. A DCLPI instruction must be issued on this shader type before is it used in any other instruction. This register cannot be used with relative addressing.

It is an error to use a register of this type in a vertex shader if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_COLOR or IL_IMPORTUSAGE_BACKCOLOR.

It is an error to use a register of this type in a pixel shader if a PINPUT register is used. You can achieve similar functionality by using the DCLPIN instruction on a PINPUT register and declaring its usage as IL_IMPORTUSAGE_COLOR or IL_IMPORTUSAGE_BACKCOLOR.

In a vertex shader, these are write-only registers. In a pixel shader, these are read-only registers.

5.36 PRIMCOORD

Enum: IL_REGTYPE_PRIMCOORD

Text Syntax: `vPrimCoord`

Components per Register: 2

Description:

This is a graphics-only feature. Pixel shader input for point-aa or line-aa texture coordinates. SPRITECOORD cannot be used in any shader that uses PRIMCOORD. The first and second components contain the pixel's S and T coordinate for the point/line primitive rendered. The values of this register are undefined if the primitive rendered is not a point or a line. The third and fourth components of this register are undefined. This read-only register cannot be the destination of an instruction. This register cannot be used with relative addressing. It is an error to use this register in a vertex or geometry shader.

5.37 PRIMITIVE_INDEX

Enum: IL_REGTYPE_PRIMITIVE_INDEX

Text Syntax: `vPrimIndex`

Components per Register: 4

Description:

This read-only register type is valid only for HOS rendering. For normal, non-HOS rendering, this register type is invalid, and its contents are undefined. For HOS rendering, this is the incremental index of the current tessellation primitive generated by the tessellation engine for certain types of HOS rendering. This register cannot be used with relative addressing. It is an error to use this register in a pixel shader.

5.38 PRIMTYPE

Enum: IL_REGTYPE_PRIMTYPE

Text Syntax: `vPrimType`

Components per Register: 2

Description:

This is a graphics only feature. It is a pixel shader input for a primitive type. The first component has sign bit = 1, this is a point. Values in other bits are undefined. The second component has sign bit = 1, this is a line. Values in other bits are undefined. The third and fourth components of this register are undefined. This read-only register cannot be the destination of an instruction. This register cannot be used with relative addressing. It is an error to use this register in a vertex or geometry shader.

5.39 PSOUTFOG

Enum: IL_REGTYPE_PS_OUT_FOG

Text Syntax: `oPsFog#`

Components per Register: 1

Description:

Pixel shader output for fog factor. The first component contains a fog factor. The second, third, and fourth components of this register are ignored. This write-only register cannot be used with relative addressing. It is an error to use this register in a vertex or geometry shader.

5.40 QUAD_INDEX

Enum: IL_REGTYPE_QUAD_INDEX

Text Syntax: `vQuadIndex`

Components per Register: 4

Description:

This register type is valid only for HOS rendering. For normal, non-HOS rendering, this register type is invalid and its contents are undefined. For HOS rendering, the first component is a quad index generated by the tessellation engine for certain types of HOS rendering. This read-only register cannot be used with relative addressing. It is an error to use this register in a pixel shader.

5.41 SECCOLOR

Enum: IL_REGTYPE_SECCOLOR

Text Syntax (VS): `oSecColor#` {write-only)

Text Syntax (PS): `vSecColor#` (read-only)

Components per Register: 4

Description:

Vertex shader export and pixel shader import for interpolated secondary color data. By convention, register number 0 represents the front-facing color, while register number 1 represents the back-facing color. When flat shading is used (AS_SHADE_MODE is set to FLAT), no interpolation is performed on the values of these registers when passed from the vertex shader to the pixel shader. Instead, only the value of the provoking vertex is passed to the pixel shader.

When smooth shading is used (AS_SHADE_MODE is set to SMOOTH), perspective correct interpolation is performed on the values.

This register cannot be used with relative addressing. A DCLPI instruction must be issued on this shader type before is it used in any other instruction.

It is an error to use a register of this type in a vertex shader if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_COLOR or IL_IMPORTUSAGE_BACKCOLOR.

It is an error to use a register of this type in a pixel shader if a PINPUT register is used. You can achieve similar functionality by using the DCLPIN instruction on a PINPUT register and declaring its usage as IL_IMPORTUSAGE_COLOR or IL_IMPORTUSAGE_BACKCOLOR.

In a vertex shader, these are write-only registers. In a pixel shader, these are read-only registers.

5.42 SHADER_INSTANCE_ID

Enum: IL_REGTYPE_SHADER_INSTANCE_ID

Text Syntax: `vInstanceID`

Components per Register: 1

Description:

This read-only register is used by the geometry shader or hull shader as a system-generated input.

Example:

```
mov r1, vInstanceID
```

5.43 SHARED_TEMP

Enum: IL_REGTYPE_SHARED_TEMP

Text Syntax: `sr#` (where # is any shared register number)

Components per Register: 4

Description:

This read/write register is shared by all wavefronts running on a SIMD. Only absolute address mode is allowed; for example: `sr2`. It is used only in compute shaders. Operations on shared registers are guaranteed atomic only when the read and write occur in the same instruction.

Supported on R7XX and Evergreen GPUs.

Example:

```
iadd sr2.xy_w, sr2.xyzw, r0.xxxx
```

5.44 SPRITE

Enum: IL_REGTYPE_SPRITE

Text Syntax: `oSprite`

Components per Register: 1

Description:

Vertex shader export for point size. The first component contains the point size. This write-only register cannot be the source of an instruction. This register cannot be used with relative addressing.

It is an error to use this register if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_POINTSIZE.

It is an error to use this register in a pixel shader. The second, third and fourth components of this register are undefined.

5.45 SPRITECOORD

Enum: IL_REGTYPE_SPRITECOORD

Text Syntax: `vSpriteCoord`

Components per Register: 2

Description:

Pixel shader input for sprite texture coordinate. The first and second components contain the pixel's S and T coordinate for the point primitive rendered. The values of this register are undefined if the primitive rendered is not a point. The third and fourth components of this register are undefined. This read-only register cannot be the destination of an instruction. This register cannot be used with relative addressing. It is an error to use this register in a vertex shader.

5.46 STENCIL

Enum: IL_REGTYPE_STENCIL

Text Syntax: `oSTENCIL`

Components per Register: 1

Description:

This is a scalar register where the stencil value is contained in the first component. The second, third, and fourth components of this register are unused and undefined. Values assigned to Stencil are treated as integer. This write-only register cannot be the source of an instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing.

5.47 TEMP

Enum: IL_REGTYPE_TEMP

Text Syntax: `r`

Components per Register: 4

Description:

This is a simple, non-indexable, read-write temporary register.

5.48 TEXCOORD

Enum: IL_REGTYPE_TEXCOORD

Text Syntax (VS): `oT#` (write-only)

Text Syntax (PS): `vT#` (read-only)

Components per Register: 4

Description:

Vertex shader export and pixel shader import interpolated for texture coordinate data. Perspective correct interpolation is performed on the values of these registers when passed from the vertex shader to the pixel shader. A DCLPI instruction must be issued on this shader type before is it used in any other instruction.

It is an error to use a register of this type in a vertex shader if a VOUTPUT register is used. You can achieve similar functionality by using the DCLVOUT instruction on a VOUTPUT register and declaring its usage as IL_IMPORTUSAGE_GENERIC.

In a vertex shader, this write-only register cannot be the source of an instruction.

In a vertex shader, this register can be used with loop-relative addressing as a destination only.

It is an error to use a register of this type in a pixel shader if a PINPUT register is used. You can achieve similar functionality by using the DCLPIN instruction on a PINPUT register and declaring its usage as IL_IMPORTUSAGE_GENERIC.

In a pixel shader, this is a read-only register.

In a pixel shader, this register can be used with loop-relative addressing as a source only.

5.49 THIS

Enum: IL_REGTYPE_THIS

Text Syntax: `this`

Components per Register: 4

Description:

This read-only register can be used only within the body of a virtual function/interface. The four values contain instance data for the function:

- x: selects the constant buffer.
- y: specifies the offset into the selected constant buffer where data for this instance begins.
- z: contains the instance sample ID.
- w: contains the instance texcoord.

This is always indexed, so `this[r0.x]` refers to the instance with number r0.x.

Valid in Evergreen GPUs and later.

Example:

```
mov r1.y, this[1].x
  mov r1, this[r3.w + 2].z
```

5.50 THREAD_GROUP_ID

Enum: IL_REGTYPE_THREAD_GROUP_ID

Text Syntax: `vThreadGrpId`

Components per Register: 3

Description:

This read-only input register contains the work-group ID, which is three-dimensional. The x, y, and z components of this register can be used as an index or in integer operations. The w component is not valid; do not use it.

This register is used only in a compute shader.

Valid in R7XX GPUs and later.

Example:

```
mov r2,  vThreadGrpId.xyzy
mov g[vThreadGrpId.x], r2
```

5.51 THREAD_GROUP_ID_FLATTENED

Enum: IL_REGTYPE_THREAD_GROUP_ID_FLAT

Text Syntax: `vThreadGrpIdFlat` (also as `vTGroupid` for back-compatibility)

Components per Register: 1

Description:

This read-only input register contains the flattened work-group ID. It assumes the number of work-groups to be dispatched is (Dx, Dy, Dz). The flattened value is computed as:

```
vThreadGrpIdFlat.x = vThreadGrpId.z*Dx*Dy  +  vThreadGrpId.y*Dx
+ vThreadGrpId.x
```

This register can be used as index or in integer operations. Only the x component has a meaningful value. The y, z, and w components replicate the value of the x component.

This register is used only in a compute shader.

Valid in R7XX GPUs and later.

Example:

```
mov g[vThreadGrpIdFlat.x], r2
```

5.52 THREAD_ID_IN_GROUP

Enum: IL_REGTYPE_THREAD_ID_IN_GROUP

Text Syntax: `vTidInGrp`

Components per Register: 3

Description:

Read-only register.

This read-only input register contains the work-item ID within a work-group. The ID is three-dimensional. The xyz components of this register can be used as an index or in an integer operation. The w component is not valid and must not be used. This register is used only in a compute shader.

Example:

```
mov r1, vTidInGrp.xyzz
mov g[vTidInGrp.x], r1
```

5.53 THREAD_ID_IN_GROUP_FLATTENED

Enum: IL_REGTYPE_THREAD_ID_IN_GROUP_FLAT

Text Syntax: `vTidInGrpFlat` (also as `vTid` for back-compatibility)

Components per Register: 1

Description:

This read-only input register contains the flattened work-item ID within a work-group. It assumes the work-group size is (Dx, Dy, Dz). The flattened value is computed as:

```
vTidInGrpFlat.x = vTidInGrp.z*Dx*Dy   +   vTidInGrp.y*Dx +
vTidInGrp.x
```

This register can be used as an index or in integer operations. Only the x component has a meaningful value. The y, z, and w components replicate the value of the x component.

It is used only in a compute shader.

Valid in R7XX GPUs and later.

Example:

```
mov g[vTidInGrpFlat.x], r2
```

5.54 TIMER

Enum: IL_REGTYPE_TIMER

Text Syntax: `Tmr`

Components per Register: 2

Description:

The current value of the cycle timer. This time as an absoulute cycle count and is incremented even when the shader instance is not active. The result is a 64-bit unsigned integer value returned as `Timer.xy`.

This read-only register is available to any type of shader, not just compute shaders. The third and fourth commponents of this register are undefined.

The timer can only appear as the source of a move instruction. No source modifiers are allowed on this input.

The counter is an implementation-dependent measure of cycles in the GPU engine. A single reading of the cycle counter is meaningless. But any shader invocation can poll the counter value any number of times. Computing a delta from cycle counter readings within a shader invocation is meaningful. Computing a delta from cycle counter readings across separate shader invocations is not meaningful on all hardware. Since execution of a shader can be interrupted by wavefront switching, delta measurements are arbitrarily larger than the number of cycles spent executing instructions in a given work-item.

There is no supported way to find out the frequency of the counter. There is no way to correlate this shader internal counter with external timers such as asynchronous time queries. If the GPU speed changes (for power saving), there is no way to know this happened, or its effect on cycle measurements.

The compiler treats reads of the cycle counter as memory barriers. In addition, instructions cannot be moved across a counter read, and counter reads cannot be merged.

The x value of the timer is the low 32 bits LSB of the counter, the y value is the upper bits. The counter wraps on overflow.

There is only one timer, so `tmr4` is an error.

Valid for Evergreen GPUs and later.

5.55 VERTEX

Enum: IL_REGTYPE_VERTEX

Text Syntax: `v#`

Components per Register: 4

Description:

Read-only register. Input to a vertex shader that typically is generated in a previous phase and passed to the current phase. It is most frequently passed as values in the .xy channels, although all channels are available.

5.56 VOUTPUT

Enum: IL_REGTYPE_VOUTPUT

Text Syntax: `oVtxOut#`

Components per Register: 4

Number Per Shader: 18

Description:

Vertex shader data. This write-only register cannot be the source of an instruction. It is an error to use this register in a pixel shader. This register can be used with loop-relative addressing as a destination only. A `DCLVOUT` instruction must be issued on a register of this type before it is used. It is an error to use this register if a POS, SPRITE, INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG register is used.

5.57 VPRIM

Enum: IL_REGTYPE_VPRIM

Text Syntax: `vPrim`

Components per Register: 4

Description:

This is a scalar that can be applied to each interior primitive in a geometry shader.

5.58 vWINCOORD

Enum: IL_REGTYPE_WINCOORD

Text Syntax: `vWinCoord.xy`

Components per Register: 2

Description:

Pixel shader import for screen position data. The first and second components are the X and Y position of the pixel in the domain of execution. The third component is the Z coordinate of the pixel in window space. The fourth component is W.

A DCLPI instruction must be issued on this shader type before is it used in any other instruction. The DCLPI instruction specifies whether the X and Y coordinate is relative to the lower-left or upper-left corner of the window and whether it represents the center or upper-left corner of the pixel.

This read-only register cannot be the destination of an instruction. It is an error to use this register in a vertex shader. This register cannot be used with relative addressing.

Chapter 6
Enumerated Types

This chapter lists and briefly describes the AMD IL enumerated types.

6.1 ILAddressing

See `IL_Dst` (page 3) and `IL_Src` (page 5) for more information.

Table 6.1 ILAddressing Enumeration Types

Enumeration	Description
IL_ADDR_ABSOLUTE (no = 0)	Absolute addressing is used.
IL_ADDR_REG_RELATIVE	Register-relative addressing is used.
IL_ADDR_RELATIVE	Relative addressing is used.

6.2 ILAnisoFilterMode

See the `TEXLD` (page 103), `TEXLDB` (page 106), and `TEXLDD` (page 110) instructions for more information.

Table 6.2 ILAnisoFilterMode Enumeration Types

Enumeration	Text Syntax	Description
IL_ANISOFILTER_DISABLED	_aniso(*disabled*)	Never use anisotropic filtering.
IL_ANISOFILTER_MAX_1_TO_1	_aniso(*1*)	Use a maximum of 1 sample for anisotropic filtering.
IL_ANISOFILTER_MAX_16_TO_1	_aniso(*16*)	Use a maximum of 16 samples for anisotropic filtering.
IL_ANISOFILTER_MAX_2_TO_1	_aniso(*2*)	Use a maximum of 2 samples for anisotropic filtering.
IL_ANISOFILTER_MAX_4_TO_1	_aniso(*4*)	Use a maximum of 4 samples for anisotropic filtering.
IL_ANISOFILTER_MAX_8_TO_1	_aniso(*8*)	Use a maximum of 8 samples for anisotropic filtering.
IL_ANISOFILTER_UNKNOWN	_aniso(*unknown*)	Use the anisotropic filtering mode specified by state.

6.3 ILCmpValue

See the `CMP` instruction (page 155) for usage.

Table 6.3 ILCmpValue Enumeration Types

Enumeration	Text Syntax	Description
IL_CMPVAL_0_0	_cmpval(*0.0*)	Compare vs. 0.0.
IL_CMPVAL_0_5	_cmpval(*0.5*)	Compare vs. 0.5.
IL_CMPVAL_1_0	_cmpval(*1.0*)	Compare vs. 1.0.
IL_CMPVAL_NEG_0_5	_cmpval(*-0.5*)	Compare vs. -0.5.
IL_CMPVAL_NEG_1_0	_cmpval(*-1.0*)	Compare vs. -1.0.

6.4 ILComponentSelect

See Section 2.2.7, “Source Modifier Token,” page 2-7 for usage. IL Text details for component selection can be found in Chapter 3, “Text Instruction Syntax.”

Table 6.4 ILComponentSelect Enumeration Types

Enumeration	Description
IL_COMPSEL_0	Force this component to 0.0.
IL_COMPSEL_1	Force this component to 1.0.
IL_COMPSEL_W_A	Select the fourth component (w/alpha) for the component.
IL_COMPSEL_X_R	Select the first component (x/red) for the component.
IL_COMPSEL_Y_G	Select the second component (y/green) for the component.
IL_COMPSEL_Z_B	Select the third component (z/blue) for the component.

6.5 ILDefaultVal

See the DCLDEF instruction (page 53) for usage.

Table 6.5 ILDefaultVal Enumeration Types

Enumeration	Text Syntax	Description
IL_DEFVAL _0	_*<comp>*(0) where *<comp>* is *x, y, z, or w*	Indicates that the default value for this component of the register type is 0.0.
IL_DEFVAL_1	_*<comp>*(1) where *<comp>* is *x, y, z, or w*	Indicates that the default value for this component of the register type is 1.0.
IL_DEFVAL_NONE	_*<comp>*(*) where *<comp>* is *x, y, z, or w* Example: `dcldef_z(*)_w(*)`	Indicates there is no default value for this component of the register type.

6.6 ILDivComp

See Section 3.6, “Source Modifiers,” page 3-3 for usage.

Table 6.6 ILDivComp Enumeration Types

Enumeration	Text Syntax	Description
IL_DIVCOMP_NONE	_divcomp(*none*)	None
IL_DIVCOMP_UNKNOWN	_divcomp(*unknown*)	The component used to divide is not known at shader create time. The component is determined by a state setting at shader run time. This value can only be used in a TEXLD, TEXLDD, TEXLDB, TEXLDMS, TEXWEIGHT, PROJECT, or LOD instruction.
IL_DIVCOMP_W	_divcomp(*w*)	Divide the first, second, and third components by the fourth component. If the fourth component is 0.0, the first, second, and third component become ±FLT_MAX.
IL_DIVCOMP_Y	_divcomp(*y*)	Divide the first component by the second component. If the second component is 0.0, the first component becomes ±FLT_MAX.
IL_DIVCOMP_Z	_divcomp(*z*)	Divide the first and second components by the third component. If the third component is 0.0, the first and second component become ±FLT_MAX.

6.7 ILElementFormat

Table 6.7 ILElementFormat Enumeration Types

Enumeration	Description
IL_ELEMENTFORMAT_FLOAT	Element uses a single-precision, floating point format.
IL_ELEMENTFORMAT_MIXED	Element uses more than one format.
IL_ELEMENTFORMAT_SINT	Element uses a signed, integer format.
IL_ELEMENTFORMAT_SNORM	Element uses a signed, normalized format.
IL_ELEMENTFORMAT_SRGB	Element uses an integer format in the sRGB color space.
IL_ELEMENTFORMAT_UINT	Element uses an unsigned, integer format.
IL_ELEMENTFORMAT_UNKNOWN	Element uses an unknown or user-defined format.
IL_ELEMENTFORMAT_UNORM	Element uses an unsigned, normalized format.

6.8 ILFirstBitType

Table 6.8 IL_FIRSTBIT Enumeration Types

Enumeration	Description
IL_FIRSTBIT_TYPE_LOW_UINT	Find the first bit set in an integer from the most significant (upper-most) bit.
IL_FIRSTBIT_TYPE_HIGH_UINT	Find the first bit set in an integer from the least significant (lower-most) bit.
IL_FIRSTBIT_TYPE_HIGH_INT	Find the first bit set in a positive integer from the most significant bit, or find the first bit clear in a negative integer from the most significant bit.

6.9 ILImportComponent

See the DLCPI (page 54), DCLPIN (page 56), and DCLVOUT (page 62) instructions for usage.

Table 6.9 ILImportComponent Enumeration Types

Enumeration	Text Syntax	Description
IL_IMPORTSEL_DEFAULT0	_<*comp*>(0) where <*comp*> is *x, y, z, or w*	This component is enabled and can be used in a vertex shader. If this register or this component of this register is not written to, the component defaults to 0.0 when used as a source or when the shader terminates. In a pixel shader, if this register or this component of this register is not exported in the vertex shader, the component is set to 0.0. If used in with the DCLVOUT instruction in the vertex shader, this component will default to 0.0 if it is not written to. The component is considered to be exported in the vertex shader in this case; thus, any component in a pixel shader mapped to this component will be set to 0.0 regardless of its default value set by the DCLPIN instruction.
IL_IMPORTSEL_DEFAULT1	_<*comp*>(1) where <*comp*> is *x, y, z, or w*	This component is enabled and can be used. In a vertex shader, if this register or this component of this register is not written to, the component defaults to 1.0 when used as a source or when the shader terminates. In a pixel shader, if this register or this component of this register is not exported in the vertex shader, set the component to 1.0. If used with the DCLVOUT instruction in the vertex shader, this component defaults to 1.0 if it's not written to. The component is considered to be exported in the vertex shader in this case; thus, any component in a pixel shader mapped to this component is set to 1.0, regardless of its default value set by the DCLPIN instruction.
IL_IMPORTSEL_UNDEFINED	_<*comp*>(*) where <*comp*> is *x, y, z, or w*	This component is enabled and can be used. If this register or this component of the register is not exported in the vertex shader, the value of the component is undefined (the value of the component does not matter).
IL_IMPORTSEL_UNUSED	_<*comp*>(-) where <*comp*> is *x, y, z, or w* Example: `dclpi_z(-)_w(-)`	This component is disabled and cannot be used in the shader. It is an error to reference the component in the shader.

6.10 ILImportUsage

See the `DCLVOUT` (page 62) and `DCLPIN` (page 56) instructions for more information.

Table 6.10 ILImportUsage Enumeration Types

Enumeration	Text Syntax	Description
IL_IMPORTUSAGE_BACKCOLOR	_usage(*backcolor*)	When used to declare a vertex shader output, this usage indicates the register contains a color value to be interpolated across the primitive and passed to the pixel shader once the shader terminates. The processed value can be read from the PINPUT register declared with this usage and the matching *usageIndex.* When used to declare a pixel shader input, this usage indicates the register contains a color value that has been interpolated across the primitive. The value originates from the VOUTPUT register in the vertex shader declared with this *usage* and matching *usageIndex*. Hardware can use lower-precision interpolators for colors.
IL_IMPORTUSAGE_COLOR	_usage(*color*)	When used to declare a vertex shader output, this usage indicates the register contains a color value to be interpolated across the primitive and passed to the pixel shader once the shader terminates. The processed value can be read from the PINPUT register declared with this usage and the matching *usageIndex.* When used to declare a pixel shader input, this usage indicates the register contains a color value that has been interpolated across the primitive. The value originates from the VOUTPUT register in the vertex shader declared with this *usage* and matching *usageIndex*. Hardware can use lower-precision interpolators for colors.
IL_IMPORTUSAGE_DENSITY_TESSFACTOR	_usage(*density_tessfactor*)	Can be used only in a hull shader to declare an output is a density tessfactor.
IL_IMPORTUSAGE_DETAIL_TESSFACTOR	_usage(*detail_tessfactor*)	Can be used only in a hull shader to declare an output is a detail tessfactor.
IL_IMPORTUSAGE_EDGE_TESSFACTOR	_usage(*edge_tessfactor*)	Can be used only in a hull shader to declare an output is an edge tessfactor.

Table 6.10 ILImportUsage Enumeration Types (Cont.)

Enumeration	**Text Syntax**	**Description**
IL_IMPORTUSAGE_FOG	_usage(*fog*)	When used to declare a vertex shader output, this usage indicates that the x component of the register has the vertices' fog value at shader termination. The value is interpolated across the primitive and passed to the pixel shader. The processed value can be read in the pixel shader from the PINPUT register declared with this *usage*. When used to declare a pixel shader input, indicates that the x component of the register contains the fog value at shader execution. The value of the VOUTPUT register in the vertex shader declared with the DCLVOUT instruction as having this *usage* is interpolated across the primitive and passed to the PINPUT register declared with this *usage*. For any register declared with this *usage*, the *usageIndex* must be set to 0.
IL_IMPORTUSAGE_GENERIC	_usage(*generic*)	When used to declare a vertex shader output, this usage indicates the register contains a generic value to be interpolated across the primitive and passed to the pixel shader once the shader terminates. The processed value can be read from the PINPUT register declared with this usage and the matching *usageIndex.* When used to declare a pixel shader input, this usage indicates the register contains a generic value that has been interpolated across the primitive. The value originates from the VOUTPUT register in the vertex shader declared with this *usage* and matching *usageIndex*.
IL_IMPORTUSAGE_INSIDE_TESSFACTOR	_usage(*inside_tessfactor*)	Can be used only in a hull shader to declare an output is an inside tessfactor.
IL_IMPORTUSAGE_POINTSIZE	_usage(*pointsize*)	When used to declare a vertex shader output, this usage indicates the x component of the register contains the vertices' point size when the shader terminates. *usageIndex* must be zero when this *usage* is set. Only VOUTPUT register 1 can have this *usage*. This *usage* cannot be used in a pixel shader input.
IL_IMPORTUSAGE_POS	_usage(*pos*)	When used to declare a vertex shader output, this usage indicates the register contains the vertices' position when the shader terminates. *usageIndex* must be zero when this *usage* is set. Only VOUTPUT register 0 can have this *usage*. This *usage* cannot be used in a pixel shader input.
IL_IMPORTUSAGE_WINCOORD	_usage(*wincoord*)	Can only be used in a pixel shader. The x, y, z, w values correspond to the screen position.

6.11 ILInterpMode

Table 6.11 ILInterpolation Enumeration Types

Enumeration	Text Syntax
IL_INTERPMODE_CONSTANT	_interp(*constant*)
IL_INTERPMODE_LINEAR	_interp(*linear*)
IL_INTERPMODE_LINEAR_CENTROID	_interp(*centroid*)
IL_INTERPMODE_LINEAR_NOPERSPECTIVE	_interp(*noperspective*)
IL_INTERPMODE_LINEAR_NOPERSPECTIVE_CENTROID	_interp(*noper_centroid*)
IL_INTERPMODE_LINEAR_NOPERSPECTIVE_SAMPLE	_interp(*noper_sample*)
IL_INTERPMODE_LINEAR_SAMPLE	_interp(*sample*)
IL_INTERPMODE_NOTUSED = 0	_interp(*notused*)

6.12 ILLanguageType

Table 6.12 ILLanguageType Enumeration Types

Enumeration	Description
IL_LANG_DX10_GS	DX 4.x geometry shader.
IL_LANG_DX10_PS	DX 4.x pixel shader.
IL_LANG_DX10_VS	DX 4.x vertex shader.
IL_LANG_DX11_CS	DX 4.x compute shader.
IL_LANG_DX11_DS	DX 4.x domain shader.
IL_LANG_DX11_GS	DX 5.x geometry shader.
IL_LANG_DX11_HS	DX 4.x hull shader.
IL_LANG_DX11_PS	DX 5.x pixel shader.
IL_LANG_DX11_VS	DX 5.x vertex shader.
IL_LANG_DX8_PS	DX 1.x pixel shader (DX 8).
IL_LANG_DX8_VS	DX 1.x vertex shader (DX 8).
IL_LANG_DX9_PS	DX 2.x and 3.x pixel shader.
IL_LANG_DX9_VS	DX 2.x and 3.x vertex shader.
IL_LANG_GENERIC	Allows for language-specific overrides.
IL_LANG_OPENGL	Valid for any version of OpenGL.

6.13 ILLdsSharingMode

Table 6.13 IL LDS Sharing Mode

Enumeration	Description
IL_LDS_SHARING_MODE_RELATIVE	For `wavefront_rel`.
IL_LDS_SHARING_MODE_ABSOLUTE	For `wavefront_abs`.

6.14 ILLoadStoreDataSize

Table 6.14 IL LOAD_STORE_DATA_SIZE

Enumeration	Description
IL_LOAD_STORE_DATA_SIZE_DWORD	Dword, 32 bits.
IL_LOAD_STORE_DATA_SIZE_SHORT	Short, 16 bits.
IL_LOAD_STORE_DATA_SIZE_BYTE	Byte, 8 bits.

6.15 ILLogicOp

Table 6.15 ILLogicOp Enumeration Types

Enumeration	Text Syntax	Description
IL_LOG_EQ	_log(*eq*)	Equal (32-bit integer compare).
IL_LOG_NE	_log(*ne*)	Not equal (32-bit integer compare).

6.16 ILMatrix

Table 6.16 ILMatrix Enumeration Types

Enumeration	IL Text
IL_MATRIX_3X2	_matrix(*3x2*)
IL_MATRIX_3X3	_matrix(*3x3*)
IL_MATRIX_3X4	_matrix(*3x4*)
IL_MATRIX_4X3	_matrix(*4x3*)
IL_MATRIX_4X4	_matrix(*4x4*)

6.17 ILMipFilterMode

See the TEXLD (page 103), TEXLDB (page 106), and TEXLDD (page 110) instructions for more information.

Table 6.17 ILMipFilterMode Enumeration Types

Enumeration	Text Syntax	Description
IL_MIPFILTER_BASE	_mip(*base*)	Always sample the base mipmap only.
IL_MIPFILTER_LINEAR	_mip(*linear*)	Always use linear filtering across mipmaps.
IL_MIPFILTER_POINT	_mip(*point*)	Always sample the nearest mipmap only.
IL_MIPFILTER_UNKNOWN	_mip(*unknown*)	Use the filtering mode specified by state.

6.18 ILModDstComponent

See Section 3.4, “Destination Modifiers,” page 3-2 for usage. IL Text details for write mask can be found in Section 3.5, “Write Mask,” page 3-3.

Table 6.18 ILModDstComp Enumeration Types

Enumeration	Description
IL_MODCOMP_0	Write 0.0 to this component.
IL_MODCOMP_1	Write 1.0 to this component.
IL_MODCOMP_NOWRITE	Do not write to this component. The contents of this component of the destination register are not changed.
IL_MODCOMP_WRITE	Write the result of the instruction to this component.

6.19 ILNoiseType

See the `NOISE` instruction (page 191) for more details.

Table 6.19 ILNoiseType Enumeration Types

Enumeration	Text Syntax	Description
IL_NOISETYPE_PERLIN1D	_type(*perlin1D*)	Compute 1D Perlin noise function.
IL_NOISETYPE_PERLIN2D	_type(*perlin2D*)	Compute 2D Perlin noise function.
IL_NOISETYPE_PERLIN3D	_type(*perlin3D*)	Compute 3D Perlin noise function.
IL_NOISETYPE_PERLIN4D	_type(*perlin4D*)	Compute 4D Perlin noise function.

6.20 ILOpcode

Table 6.20 ILOpcode Enumeration Types

Enumeration	Description
IL_OP_ABS	Computes the absolute value of each element of a vector.
IL_DCL_CONST_BUFFER	Declares a constant buffer.
IL_DCL_INDEXED_TEMP_ARRAY	Declares an indexed array.
IL_DCL_INPUT	Declares an input register.
IL_DCL_INPUT_PRIMITIVE	Used to declare the input primitive type.
IL_DCL_LITERAL	Declares a literal value.
IL_DCL_MAX_OUTPUT_VERTEX_COUNT	Used to declare the maximum number of vertices that will be emitted by a shader
IL_DCL_ODEPTH	Used to declare that the pixel shader intends to write to its scalar output oDepth register.
IL_DCL_OUTPUT	Declares an output register.
IL_DCL_OUTPUT_TOPOLOGY	Used to declare the output topology of a primitive.
IL_DCL_PERSIST	Used to declare the amount of persistent storage used by a shader.
IL_DCL_RESOURCE	Declares an input buffer.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_DCL_VPRIM	Used to declare an input register.
IL_OP_ACOS	Used to compute the inverse cosine of a component.
IL_OP_ADD	Computes the single-precision, floating point addition of the values in each element of a vector to the values in the corresponding element in another vector.
IL_OP_AND	Performs a logical-AND on two vectors.
IL_OP_ASIN	Used to compute the inverse sine of a component.
IL_OP_ATAN	Used to compute the inverse tangent of a component.
IL_OP_BREAK	Unconditionally terminates a LOOP or SWITCH block.
IL_OP_BREAK_LOGICALNZ	Conditionally terminates a LOOP or SWITCH block if the parameter is not zero.
IL_OP_BREAK_LOGICALZ	Conditionally terminates a LOOP or SWITCH block if the parameter is zero.
IL_OP_BREAKC	Conditionally terminates a LOOP or SWITCH block.
IL_OP_CALL	Unconditionally calls a FUNC block.
IL_OP_CALL_LOGICALNZ	Calls a FUNC block if the parameter is not zero.
IL_OP_CALL_LOGICALZ	Calls a FUNC block if the parameter is zero.
IL_OP_CALLNZ	Used to call a FUNC block if the contents of the Constant Boolean register are not zero.
IL_OP_CASE	Compares <case-value> to <switch-value>, and conditionally executes the CASE instruction block if they are equal.
IL_OP_CLAMP	Clamps a value to two others.
IL_OP_CLG	Computes the ceiling of the value in each element of a vector.
IL_OP_CMOV	Performs a single-precision, floating point comparison of the value in each element of a vector to 0.0f, and conditionally moves the value in each element of another vector to the corresponding element of a third vector if the comparison evaluates FALSE.
IL_OP_CMOV_LOGICAL	Performs an integer comparison of the value in each element of a vector to zero, and moves the value in each element of a second vector to the corresponding element of a third vector if the comparison evaluates TRUE; otherwise, moves the corresponding element of a fourth vector to the corresponding element of the third vector.
IL_OP_CMP	Performs a logical comparison of the value in each element of a comparison value, and conditionally moves the value in each element of another vector to the corresponding element of a third vector if the comparison evaluates TRUE; otherwise, the value in the corresponding element of a fourth vector is moved.
IL_OP_COLORCLAMP	Clamps (or does not clamp) an output color to specified values.
IL_OP_COMMENT	Allows a comment to be passed into IL.
IL_OP_CONTINUE	Unconditionally continues execution at the next LOOP or WHILELOOP instruction.
IL_OP_CONTINUE_LOGICALNZ	Conditionally continues execution at the next LOOP or WHILELOOP instruction if the parameter is not zero.
IL_OP_CONTINUE_LOGICALZ	Conditionally continues execution at the next LOOP or WHILELOOP instruction if the parameter is zero.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_CONTINUEC	Conditionally continues execution at the next LOOP or WHILELOOP instruction.
IL_OP_COS	Computes the cosine of a component.
IL_OP_COS_VEC	Computes the cosine of each element of a vector.
IL_OP_CRS	Used to compute the cross product of two 3-vectors.
IL_OP_CUT	Used to close a primitive topology and open a new one.
IL_OP_D_2_F	Converts a double-precision float value to a single-precision float value.
IL_OP_D_ADD	Performs a double-precision, floating point addition of the values in each element of a vector to the values in the corresponding element in another vector.
IL_OP_D_EQ	Performs a double-precision float equality comparison.
IL_OP_D_FRAC	Returns a double-precision, floating point fraction (mantissa).
IL_OP_D_FREXP	Splits a double-precision, floating point value into fraction (mantissa) and exponent values.
IL_OP_D_GE	Performs a double-precision float greater than or equal comparison.
IL_OP_D_LDEXP	Combines fraction (mantissa) and exponent values into a double-precision, floating point value.
IL_OP_D_LT	Performs a double-precision float less than comparison.
IL_OP_D_MAD	Performs a double-precision, floating point multiplication of two values, then performs a double-precision addition on the result with a third value.
IL_OP_D_MUL	Performs a double-precision, floating point multiplication of two values.
IL_OP_D_NE	Performs a double-precision float inequality comparison.
IL_OP_DCLARRAY	Used to declare a range of registers to be included in an array.
IL_OP_DCLDEF	Not used.
IL_OP_DCLPI	Used to declare the interpolator properties.
IL_OP_DCLPIN	Used to declare an input register.
IL_OP_DCLPP	Not used.
IL_OP_DCLPT	Used to declare a texture's properties.
IL_OP_DCLV	Used to declare the mapping for a vertex shader's inputs.
IL_OP_DCLVOUT	Used to declare the output register for a vertex shader.
IL_OP_DEF	Used to declare the constant integer or float value for a register.
IL_OP_DEFAULT	Starts the default instruction block within a SWITCH instruction block.
IL_OP_DEFB	Used to declare the constant Boolean value for a register.
IL_OP_DET	Used to calculate the determinant of a 4x4 matrix.
IL_OP_DISCARD_LOGICALNZ	Logically compares a parameter to zero, and conditionally stops execution and discards the results of a kernel invocation if the parameter is not zero. When used in the compute kernel, this instruction produces undefined results.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_DISCARD_LOGICALZ	Logically compares a parameter to zero, and conditionally stops execution and discards the results of a kernel invocation if the parameter is zero. When used in the compute kernel, this instruction produces undefined results.
IL_OP_DIST	Used to compute the vector distance between two 3-vectors.
IL_OP_DIV	Performs a single-precision, floating point division of the values in each element of a vector by the values in the corresponding element in another vector.
IL_OP_DP2	Computes the dot product of two 2-vectors.
IL_OP_DP2ADD	Used to compute the dot product of two 2-vectors and scalar adds a 32-bit value to the result.
IL_OP_DP3	Computes the dot product of two 3-vectors.
IL_OP_DP4	Computes the dot product of two 4-vectors.
IL_OP_DST	Computes a distance-related value.
IL_OP_DSX	Used to compute the single-precision, floating point rate of change of a vector in the x-direction.
IL_OP_DSY	Used to compute the single-precision, floating point rate of change of a vector in the y-direction.
IL_OP_DXSINCOS	Computes sine and cosine values for the legacy DX9 sincos instruction.
IL_OP_ELSE	Starts the ELSE clause within an IF instruction block.
IL_OP_EMIT	Used to generate a primitive.
IL_OP_EMIT_THEN_CUT	Used to generate and close a primitive then start a new primitive.
IL_OP_END	Indicates the end of an IL stream.
IL_OP_ENDFUNC	Indicates the end of a FUNC instruction block.
IL_OP_ENDIF	Indicates the end of an IF or ELSE instruction block.
IL_OP_ENDLOOP	Indicates the end of a LOOP or WHILELOOP instruction block.
IL_OP_ENDMAIN	Indicates the end of a main program in a kernel.
IL_OP_ENDSWITCH	Indicates the end of a SWITCH instruction block.
IL_OP_EQ	Performs a float equality comparison.
IL_OP_EXN	Used to compute the single-precision, floating point value of *e* raised to a power.
IL_OP_EXP	Computes the single-precision, floating point value of 2 raised to a power.
IL_OP_EXP_VEC	Computes the single-precision, floating point value of 2 raised to a power for each element in a vector.
IL_OP_EXPP	Used to compute the partial-precision, floating point value of 2 raised to a power.
IL_OP_F_2_D	Converts a single-precision float value to a double-precision float value.
IL_OP_FACEFORWARD	Performs the following calculation: *d*(dst = *src2**sign(dot(*src0*, *src1*))
IL_OP_FLR	Computes the floor of the value in each element of a vector.
IL_OP_FRC	Returns a single-precision, floating point fraction (mantissa).

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_FTOI	Converts a single-precision float value to an integer value.
IL_OP_FTOU	Converts a single-precision float value to an unsigned integer value.
IL_OP_FUNC	Indicates the start of a FUNC instruction block.
IL_OP_FWIDTH	Computes the sum of the absolute derivative in x and y.
IL_OP_GE	Performs a float greater than or equal comparison.
IL_OP_I_ADD	Computes the integer addition of the values in each element of a vector to the values in the corresponding element in another vector.
IL_OP_I_EQ	Performs an integer equality comparison.
IL_OP_I_GE	Performs an integer greater than or equal comparison.
IL_OP_I_LT	Performs an integer less than comparison.
IL_OP_I_MAD	Performs an integer multiplication of the values in each element of a vector by the values in the corresponding elements of another vector; it then adds the value in the corresponding elements of a third vector to the results.
IL_OP_I_MAX	Performs an integer comparison of the value in each element of a vector to the value in the corresponding element of another vector, and returns the larger of the two values in the corresponding element of a third vector.
IL_OP_I_MIN	Performs an integer comparison of the value in each element of a vector to the value in the corresponding element of another vector, and returns the smaller of the two values in the corresponding element of a third vector.
IL_OP_I_MUL	Performs an integer multiplication of the values in each element of a vector by the values in the corresponding element in another vector, and returns the lower 32-bits of the result.
IL_OP_I_MUL_HIGH	Performs an integer multiplication of the values in each element of a vector by the values in the corresponding element in another vector, and returns the upper 32-bits of the result.
IL_OP_I_NE	Performs an integer inequality comparison.
IL_OP_I_NEGATE	Computes the two's complement negation of the values in each element in a vector.
IL_OP_I_NOT	Performs a bit-wise one's complement on a vector.
IL_OP_I_OR	Performs a logical-OR on two vectors.
IL_OP_I_SHL	Shifts integer values in each element of a vector the specified number of bits to the left.
IL_OP_I_SHR	Shifts the integer values in each element of a vector a the specified number of bits to the right through sign extension.
IL_OP_I_XOR	Performs a logical-XOR on two vectors.
IL_OP_IF_LOGICALNZ	Logically compares a parameter to zero, and executes the IF instruction block if the comparison evaluates FALSE.
IL_OP_IF_LOGICALZ	Logically compares a parameter to zero, and executes the IF instruction block if the comparison evaluates TRUE.
IL_OP_IFC	Logically compares two parameters, and conditionally executes the IF instruction block if the comparison evaluates TRUE.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_IFNZ	Used to conditionally execute the IF instruction block if the parameter is not zero.
IL_OP_INITV	Initializes a vertex shader input to the value of `src`.
IL_OP_ITOF	Converts an integer value to a single-precision float value.
IL_OP_KILL	Used to perform a float comparison on a parameter and conditionally stop execution and discard the results of a pixel shader invocation if any element in the parameter is less that 0.0f.
IL_OP_LEN	Used to compute the length of a 3-vector.
IL_OP_LIT	Calculates lighting coefficients for ambient, diffuse, and specular light contributions.
IL_OP_LN	Computes the single-precision, floating point natural logarithm of a component.
IL_OP_LOAD	Used to fetch data from a specified Buffer or Texture without filtering.
IL_OP_LOD	Uses the provided texture coordinate to determine the computed mipmap level(s) sampled from at *src0*. The LOD value returned includes any clamping and biasing defined by the texture currently bound to the specified texture unit.
IL_OP_LOG	Computes the single-precision, floating point binary logarithm of a component.
IL_OP_LOG_VEC	Computes the single-precision, floating point binary logarithm of the values in each element of a vector.
IL_OP_LOGP	Used to compute the partial-precision, floating point binary logarithm of a component.
IL_OP_LOOP	Indicates the start of a LOOP instruction block.
IL_OP_LRP	Computes the linear interpolation between two vectors.
IL_OP_LT	Performs a float less-than comparison.
IL_OP_MAD	Performs a single-precision, floating point multiplication of the values in each element of a vector by the values in the corresponding elements of another vector; it then adds the value in the corresponding elements of a third vector to the results.
IL_OP_MAX	Performs a single-precision, floating point comparison of the value in each element of a vector to the value in the corresponding element of another vector, and returns the larger of the two values in the corresponding element of a third vector.
IL_OP_MEMEXPORT	Used to export the value in *src1* to a memory stream.
IL_OP_MEMIMPORT	Used to import a value in memory to a TEMP register.
IL_OP_MIN	Performs a single-precision, floating point comparison of the value in each element of a vector to the value in the corresponding element of another vector, and returns the smaller of the two values in the corresponding element of a third vector.
IL_OP_MMUL	Performs a matrix multiply of the generic matrices *src0* and *src1*.
IL_OP_MOD	Computes the single-precision, floating point division of the values in each element of a vector by the values in the corresponding elements of another vector, and returns the remainder of each division.
IL_OP_MOV	Unconditionally moves the value in each element of a vector to the corresponding element of another vector.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_MUL	Performs a single-precision, floating point multiplication of the values in each element of a vector by the values in the corresponding element in another vector, and returns the lower 32-bits of the result.
IL_OP_NE	Performs a float inequality comparison.
IL_OP_NOISE	Uses *src0* as the seed of a pseudo-random number generator to compute a noise value.
IL_OP_NOP	Used to pad IL streams so that they are the same length in memory without unnecessary computations.
IL_OP_NRM	Computes the single-precision, floating point, normalized value of the value in each element of a vector.
IL_OP_PIREDUCE	Reduces all four components of the vector in *src0* to the range $[-\pi, \pi]$.
IL_OP_POW	Computes the value in a component raised to the power of the value in another component.
IL_OP_PROJECT	Moves a value from the source to the destination register.
IL_OP_RCP	Computes the single-precision, floating point reciprocal of the value in a component.
IL_OP_REFLECT	Computes the reflection direction of a vector.
IL_OP_RESINFO	Used to get information about a resource.
IL_OP_RET	Used to unconditionally transfer control from the end of a FUNC instruction block back to the caller.
IL_OP_RET_DYN	Unconditionally transfers control from anywhere in a FUNC instruction block back to the caller.
IL_OP_RET_LOGICALNZ	Compares a parameter to zero, and conditionally transfers control from anywhere in a FUNC instruction block back to the caller if the parameter is not zero.
IL_OP_RET_LOGICALZ	Compares a parameter to zero, and conditionally transfers control from anywhere in a FUNC instruction block back to the caller if the parameter is zero.
IL_OP_RND	Rounds the single-precision, floating point value in each element of a vector to the nearest integer.
IL_OP_ROUND_NEAR	Rounds the single-precision, floating point value in each element of a vector to the nearest even integer.
IL_OP_ROUND_NEG_INF	Rounds the single-precision, floating point value in each element of a vector towards $-\infty$.
IL_OP_ROUND_PLUS_INF	Rounds the single-precision, floating point value in each element of a vector towards ∞.
IL_OP_ROUND_ZERO	Rounds the single-precision, floating point value in each element of a vector towards zero.
IL_OP_RSQ	Computes the single-precision, floating point square root of the reciprocal of the value in a component.
IL_OP_RSQ_VEC	Computes the single-precision, floating point square root of the reciprocal of the value in each element of a vector.
IL_OP_SAMPLE	Performs a single-precision, floating point sample of a resource.
IL_OP_SAMPLE_B	Used to perform a single-precision, floating point sample of a resource with filtering and bias.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_SAMPLE_C	Used to perform a single-precision, floating point sample of a resource with filtering and comparison.
IL_OP_SAMPLE_C_LZ	Used to perform a single-precision, floating point sample of a resource with filtering and comparison.
IL_OP_SAMPLE_G	Used to perform a single-precision, floating point sample of a resource with filtering and gradient.
IL_OP_SAMPLE_L	Performs a single-precision, floating point sample of a resource with filtering.
IL_OP_SET	Performs a single-precision, floating point comparison of the value in each element of a vector with the value in each element of another vector, and places a 1.0f in the corresponding element of a third vector if the comparison evaluates TRUE; otherwise, it places a 0.0f in the corresponding element of the third vector.
IL_OP_SGN	Computes the single-precision, floating point sign of the value in each element of a vector.
IL_OP_SIN	Computes the sine of the value in a component.
IL_OP_SIN_VEC	Computes the sine of the value in each element of a vector.
IL_OP_SINCOS	Used to compute the sine and cosine of a component.
IL_OP_SQRT	Computes the single-precision, floating point square root of the value in a component.
IL_OP_SQRT_VEC	Computes the single-precision, floating point square root of the value in each element of a vector.
IL_OP_SUB	Computes the single-precision, floating point subtraction of the values in each element of a vector from the values in the corresponding element in another vector.
IL_OP_SWITCH	Indicates the start of a SWITCH instruction block, and provides a <switch-value> to later CASE instructions.
IL_OP_TAN	Used to compute the tangent of a component.
IL_OP_TEXLD	Samples a texture at specified coordinates.
IL_OP_TEXLDB	Samples a texture specified by `stage` at coordinates specified by the *src0* register and biased by the value in the fourth component of *src1.*
IL_OP_TEXLDD	Gradient Texture load. The source src0 is used as the texture coordinate to sample from the specified texture stage.
IL_OP_TEXLDMS	Samples a multi-sample texture specified by stage at coordinates specified by the src0.
IL_OP_TEXWEIGHT	Retrieves the weights used by a bilinear filtered fetch based upon the texture coordinate provided in *src0.*
IL_OP_TRANSPOSE	Transposes the rows and columns of the 4x4 matrix.
IL_OP_TRC	Performs a single-precision, floating point truncation of the value in each element of a vector.
IL_OP_U_DIV	Performs an unsigned integer division of the values in each element of a vector by the values in the corresponding element in another vector.
IL_OP_U_GE	Performs an unsigned integer greater than or equal comparison.
IL_OP_U_LT	Performs an unsigned integer less than comparison.

Table 6.20 ILOpcode Enumeration Types (Cont.)

Enumeration	Description
IL_OP_U_MAD	Performs an unsigned integer multiplication of the values in each element of a vector by the values in the corresponding elements of another vector; it then adds the value in the corresponding elements of a third vector to the results.
IL_OP_U_MAX	Performs an unsigned integer comparison of the value in each element of a vector to the value in the corresponding element of another vector; it then returns the larger of the two values in the corresponding element of a third vector.
IL_OP_U_MIN	Performs an unsigned integer comparison of the value in each element of a vector to the value in the corresponding element of another vector; it then returns the smaller of the two values in the corresponding element of a third vector.
IL_OP_U_MOD	Computes the unsigned integer division of the values in each element of a vector by the values in the corresponding elements of another vector; it then returns the remainder of each division.
IL_OP_U_MUL	Performs an unsigned integer multiplication of the values in each element of a vector by the values in the corresponding element in another vector; it then returns the lower 32-bits of the result.
IL_OP_U_MUL_HIGH	Performs an unsigned integer multiplication of the values in each element of a vector by the values in the corresponding element in another vector; it then returns the upper 32-bits of the result.
IL_OP_U_SHR	Shifts without sign extension unsigned integer values in each element of a vector the specified number of bits to the right.
IL_OP_UTOF	Converts an unsigned integer value to a single-precision float value.
IL_OP_WHILE	Indicates the start of a WHILELOOP instruction block.

6.21 ILOutputTopology

Primitive types that can be output from a geometry shader

Table 6.21 IL_OUTPUT_TOPOLOGY Enumeration Types (Output from a Geometry Shader)

Enumeration	Description
IL_OUTPUT_TOPOLOGY_LINESTRIP	Primitive type representing a line.
IL_OUTPUT_TOPOLOGY_POINTLIST	Primitive type representing a point.
IL_OUTPUT_TOPOLOGY_TRIANGLE_STRIP	Primitive type representing a triangle.

6.22 ILPixTexUsage

There are a maximum of eight values. See `DCLPT` instruction (page 59) for more information.

Table 6.22 ILPixTexUsage Enumeration Types

Enumeration	Text Syntax (dclpt)	Text Syntax (sample_ext)
IL_USAGE_PIXTEX_1D	_type(*1d*)	1d
IL_USAGE_PIXTEX_1DARRAY	_type(*1darray*)	1darray
IL_USAGE_PIXTEX_2D	_type(*2d*)	2d
IL_USAGE_PIXTEX_2DARRAY	_type(*2darray*)	2darray
IL_USAGE_PIXTEX_2DARRAYMSAA	_type(*2darraymsaa*)	2darraymsaa
IL_USAGE_PIXTEX_2DMS_ARRAY	_type(*2dms_array*)	2dms_array
IL_USAGE_PIXTEX_2DMSAA	_type(*2dmsaa*)	2dmsaa
IL_USAGE_PIXTEX_3D	_type(*3d*)	3d
IL_USAGE_PIXTEX_4COMP	_type(*4c*)	4c
IL_USAGE_PIXTEX_BUFFER	_type(*buffer*)	buffer
IL_USAGE_PIXTEX_CUBEMAP	_type(*cubemap*)	cubemap
IL_USAGE_PIXTEX_CUBEMAPARRAY	_type(*cubemaparray*)	cubemaparray
IL_USAGE_PIXTEX_UNKNOWN	_type(*unknown*)	unknown

6.23 ILRegType

See Chapter 5, “Register Types,” for information on the IL register types.

6.24 ILRelOp

See `IFC`, `CONTINUEC`, `BREAKC`, `CMP`, and `SET` for usage.

Table 6.23 ILRelOp Enumeration Types

Enumeration	Text Syntax	Description
IL_RELOP_EQ	_relop(*eq*)	Equal.
IL_RELOP_GE	_relop(*ge*)	Greater than or equal.
IL_RELOP_GT	_relop(*gt*)	Greater than.
IL_RELOP_LE	_relop(*le*)	Less than or equal.
IL_RELOP_LT	_relop(*lt*)	Less than.
IL_RELOP_NE	_relop(*ne*)	Not equal.

6.25 ILShader

Table 6.24 ILShader Enumeration Types

Enumeration	Description
IL_SHADER_COMPUTE	This code describes a compute shader. Valid for R7XX GPUs and later.
IL_SHADER_DOMAIN	This code describes a domain shader. Valid for Evergreen GPUs and later.
IL_SHADER_GEOMETRY	This code describes a geometry shader. Valid for R6XX GPUs and later.
IL_SHADER_HULL	This code describes a hull shader. Valid for Evergreen GPUs and later.
IL_SHADER_PIXEL	This code describes a kernel that uses a work-item creation pattern optimized for graphics (pixel kernel).
IL_SHADER_VERTEX	This code describes a vertex shader.

6.26 ILShiftScale

See Section 3.4, “Destination Modifiers,” page 3-2 for usage.

Table 6.25 ILShiftScale Enumeration Types

Enumeration	Text Syntax	Description
IL_SHIFT_D2	_d2	Shift value right by 1 bit (divide by 2).
IL_SHIFT_D4	_d4	Shift value right by 2 bits (divide by 4).
IL_SHIFT_D8	_d8	Shift value right by 3 bits (divide by 8).
IL_SHIFT_NONE		Do not shift.
IL_SHIFT_X2	_x2	Shift value left by 1 bit (multiply by 2).
IL_SHIFT_X4	_x4	Shift value left by 2 bits (multiply by 4).
IL_SHIFT_X8	_x8	Shift value left by 3 bits (multiply by 8).

6.27 ILTexCoordMode

See DCLPT instruction for more information.

Table 6.26 ILTexCoordMode Enumeration Types

Enumeration	Text Syntax	Description
IL_TEXCOORDMODE_NORMALIZED	_coordmode(*normalized*)	The texture coordinates given in the texture load instructions are non-parametric. (the coordinate range [0.0-1.0] spans the entire texture)
IL_TEXCOORDMODE_UNKNOWN	_coordmode(*unknown*)	At shader create time, it is not known if the texture coordinates given in the texture load instructions are normalized. Instead, this is determined at shader run time based on a state value.
IL_TEXCOORDMODE_UNNORMALIZED	_coordmode(*unnormalized*)	The texture coordinates given in the texture load instructions are parametric. (the coordinate range [0.0, dimension of the texture] spans the entire dimension of the texture)

6.28 ILTexFilterMode

See the TEXLD (page 103), TEXLDB (page 106), and TEXLDD (page 110) instructions for more information.

Table 6.27 ILTexFilterMode Enumeration Types

Enumeration	Text Syntax	Description
IL_TEXFILTER_ANISO	_<filter>(*aniso*) where <filter> is min, mag, volmag, or volmin	Always use anisotropic filtering if aniso filtering is enabled in state or based on the aniso parameter in the instruction.
IL_TEXFILTER_LINEAR	_<filter>(*linear*) where <filter> is min, mag, volmag, or volmin	Always use linear filtering.
IL_TEXFILTER_POINT	_<filter>(*point*) where <filter> is min, mag, volmag, or volmin	Always use point filtering.
IL_TEXFILTER_UNKNOWN	_<filter>(*unknown*) where <filter> is min, mag, volmag, or volmin	Use the filtering mode specified by state.

6.29 ILTexShadowMode

See the TEXLD (page 103), TEXLDB (page 106), and TEXLDD (page 110) instructions for more information.

Table 6.28 ILTexShadowMode Enumeration Types

Enumeration	Text Value	Description
IL_TEXSHADOWMODE_NEVER	_shadowmode(*never*)	Never do a shadow map comparison.
IL_TEXSHADOWMODE_UNKNOWN	_shadowmode(*unknown*)	Do a compare vs. the third component of the texture coordinate.
IL_TEXSHADOWMODE_Z	_shadowmode(*z*)	Always do a compare vs. the third component of the texture coordinate.

6.30 ILTopologyType

Primitive types that can be input to a geometry shader.

ILLogicOp

Table 6.29 IL_TOPOLOGY Enumeration Types (Input to a Geometry Shader)

Enumeration	Description
IL_TOPOLOGY_LINE	Input primitive type is a line; two vertices.
IL_TOPOLOGY_LINE_ADJ	Input primitive type is a line with two adjacent vertices; total of four vertices.
IL_TOPOLOGY_POINT	Input primitive type is a point; one vertex.
IL_TOPOLOGY_TRIANGLE	Input primitive type is a triangle; three vertices.
IL_TOPOLOGY_TRIANGLE_ADJ	Input primitive type is a triangle with adjacent vertices; total of six vertices.
IL_TOPOLOGY_PATCH1 . . . IL_TOPOLOGY_PATCH32	Input primitive type is a patch with one control point. . . . Input primitve type is a patch with 32 control points.

Can be used in IF_LOGICAL*, CONTINUE_LOGICAL*, and BREAK_LOGICAL*.

6.31 ILTsDomain

Tessellation domain declared in the hull shader.

Table 6.30 ILTsDomain Enumeration Types

Enumeration	Description
IL_TS_DOMAIN_ISOLINE	Indicates the tessellation domain uses an isoline, consisting of the line density tessellation factor and the line detail tessellation factor.
IL_TS_DOMAIN_QUAD	Indicates a quadrilateral; tessellation factor for the left, top, right, and bottom edges of the patch.
IL_TS_DOMAIN_TRI	Indicates a triangle; tessellation factors for the u:0, v:0, and w:0 edges of the patch.

6.32 ILTsOutputPrimitive

Tessellation domain declared in the hull shader.

Table 6.31 ILTsOutputPrimitive Enumeration Types

Enumeration	Description
IL_TS_OUTPUT_LINE	Indicates a line output primitive.
IL_TS_OUTPUT_POINT	Indicates a point output primitive.
IL_TS_OUTPUT_TRIANGLE_CW	Indicates a triangle output primitive for which the points that make up the triangle are returned in clockwise order.
IL_TS_OUTPUT_TRIANGLE_CCW	Indicates a triangle output primitive for which the points that make up the triangle are returned in counter-clockwise order.

6.33 ILTsPartition

Tessellation partitioning declared in the hull shader.

Table 6.32 ILTsPartition Enumeration Types

Enumeration	Description
IL_TS_PARTITION_FRACTIONAL_EVEN	Indicates that the tessellation factors are to be interpreted as a fractional even.
IL_TS_PARTITION_FRACTIONAL_ODD	Indicates that the tessellation factors are to be interpreted as a fractional odd.
IL_TS_PARTITION_INTEGER	Indicates that the tessellation factors are to be interpreted asan integer.
IL_TS_PARTITION_POW2	Indicates that the tessellation factors are to be interpreted as a power-of-2.

6.34 ILZeroOp

See the `RSQ` (page 201), `RCP` (page 196), `LOG` (page 179), `LOGP` (page 181), `LN` (page 178), `NRM` (page 192), and `DIV` (page 161) instructions for more information.

Table 6.33 ILZeroOp Enumeration Types

Enumeration	Text Syntax	Description
IL_ZEROOP_0	_zeroop(*zero*)	Return 0.0 when the instruction operates on 0.0. Cannot be used with LOG, LOGP, or LN instructions.
IL_ZEROOP_FLTMAX	_zeroop(*fltmax*)	Return the max floating point value when the instruction operates on 0.0.
IL_ZEROOP_INF_ELSE_MAX	_zeroop(*inf_else_max*)	Implementation-dependant: Return IEEE infinity if the implementation can support it; otherwise, use FLT_MAX when the instruction operates on 0.0.
IL_ZEROOP_INFINITY	_zeroop(*infinity*)	Return IEEE Infinity when the instruction operates on 0.0.

Chapter 7
Instructions

An IL stream consists mainly of IL instruction packets. Each packet begins with an IL_Opcode token. The type and number of tokens that follow depends upon the value of the *code* field in the initial IL_Opcode token, as well as the `modifier_present` field in each of the following IL_Dst tokens and IL_Src tokens. This chapter describes each instruction packet, including its operation and usage within the IL stream. Some instructions are defined by pseudo code. In this chapter, the symbol `V[i]` refers to source `i` post swizzle.

7.1 Formats

Most instructions use a standardized format. Rather than repeat the format in each instruction, the common format is given here. Tokens have the following order:

1. Opcode token.
2. Destination information (zero or one set of tokens: *dst* token, any relative addressing, any modifiers).
3. Source information (zero or more sets of tokens), *src* token, any relative addressing information, any modifiers.

Formats include: zero or one destination, followed by any number of sources. See the specific opcode to determine the number of destinations/sources allowed.

7.2 Instruction Notes

7.2.1 Notes on Comparison Instructions

Comparison instructions support four comparison types: float, double, integer, and unsigned integer, with different instructions for each type. Instructions return either TRUE (comparison condition met) or FALSE (comparison condition not met). Integer comparison instructions return 0xFFFFFFFF (TRUE) and 0x00000000 (FALSE). Float comparison instructions can be configured to return 1.0f (TRUE) and 0.0f (FALSE).

There are several sets of IL comparison operations, which differ in the kind of result returned. The first set uses -1 (0xFFFFFFF) for TRUE and 0 (0x00000000) for FALSE. This works quite well with the many control flow operations that check for any bit set.

A less obvious example might be:

```
if ( r1.x == r2.y) {
    r3.x = 5.0;
} else {
    r3.x = 0.0;
}
```

Assuming that l5.x contains 0x40A00000 (the IEEE OAT representation for 5.0), this can be written as

```
ieq r3.x, r1.x, r2.y
iand r3.x, r3.x, l5.x     }}
```

Integer instructions use signed arithmetic when comparing operands; unsigned integer instructions use unsigned arithmetic. Float instructions use signed arithmetic when comparing operands; however, denorms are flushed before all float instructions (the original source registers remain untouched). Thus, +0 and -0 are equivalent for float comparisons. Also, float instructions return FALSE when either operand holds NaN. See the IEEE 754 documentation for more information on floating point rules.

The Result of a double compare is either 0, or 0xFFFFFFFF is broadcast to all destination channels. The IL compare always looks at the first two components of the sources. So it can compute one 32-bit result. The result is broadcast to all channels of the destination.

The corresponding DX instruction can compare two values and produce two results.

The behavior of an int64/uint64 compare is similar to that of the double compare: the compare always looks at `_rst` two components of the source and broadcasts the result to all destination channels.

7.2.2 Notes on Flow Control Instructions

Flow control instructions determine the order in which instructions are executed by the hardware. Control flow within IL is structured.

A *flow-control-block* is code within:

- A subroutine: code between FUNC and RET, or between FUNC and ENDFUNC.
- Between IFNZ and ENDIF.
- Between IFC and ENDIF.
- Between LOG_IF and ENDIF.
- Between ELSE and ENDIF.
- Between IFNZ and ELSE.
- Between IFC and ELSE.
- Between LOOP and ENDLOOP.

- Between WHILELOOP and ENDLOOP.

There are two forms of loop: LOOP/ENDLOOP used for DX9-style counted loops, and WHILELOOP/ENDLOOP used for DX10-style while loops.

The following are the restrictions on when particular control flow instructions are allowed.

- LOOPs and WHILELOOPs must terminate in the same *flow-control-block* in which they begin.
- END, ENDMAIN, and ENDFUNC cannot be placed within a *flow-control-block.*

7.2.3 Notes on Input/Output Instructions

Instructions in this section are expected to be used only by graphic clients. Consult the DX10 documentation or other graphics programming reference for details. As of the ATI Radeon™ HD 5XXX series, it is possible to index the texture and sampler used in the following instruction.

The control field contains four items (plus one for padding).

Bit controls_resource	: 8
Bit controls_sampler	: 4
Bit indexed_args	: 1
Bit aoffset_present	: 1
Bit Reserved	: 2

Many instructions use the control field to indicate the resource and sampler used.

If the `indexed_args` bit is set to 1, there are two additional source arguments, corresponding to resource index and sampler index.

These arguments can be either a register or a literal. The resource-index argument is added to the `controls_resource` field to form the final resource index. The sampler-index argument is added to the `control_sampler` field to form the final sampler index. If the instruction takes only a resource, the `controls_sampler` field is ignored, and the sampler-index argument must be literal 0.

If the `pri_modifier_present` bit is set to 1, the Dword following the op token is the primary modifier.

IL PrimarySample Mod for sample instructions from 4.0 and later shader models.

IL_PrimarySample_Mod	1:0	gather4_comp_sel
	3:2	tex_coord_type
	4	is_uav
	31:5	reserved

`gather4_comp_sel` is valid for Evergreen and later.

- Holds the value of the enumerated type ILComponentSelect, only IL_COMPSEL_X_R, IL_COMPSEL_Y_G, IL_COMPSEL_Z_B, IL_COMPSEL_W_A are valid.
- If present, specifies the component to fetch for a multi-component texture resource. If absent, x channel is fetched.
- Applicable to `fetch4`, `fetch4c`, `fetch4po` and `fetch4poc`.
- Example:

  ```
  fetch4 resource(n) sampler(m)[ compselect(comp)] dst, src0
  ```

`tex_coord_type` is valid for R6XX and later.

- Holds the value of the enumerated type ILTexCoordMode.
- Applicable to `sample`, `sample_b`, `fetch4`, `fetch4c`, `fetch4po`, `fetch4poc`, `sample_g`, `sample_l`, `sample_c_lz`, `sample_c`, `sample_c_g`, `sample_c_l`, `sample_c_b`.
- Example:

  ```
  sample resource(n) sampler(m)[ coordtype(ILTexCoordMode)] dst,
  src0
  ```

`is_uav` is valid for Evergreen and later.

- 1 if it is a UAV; otherwise, 0.
- Applicable to `resinfo` and `bufinfo`.
- Cannot be enabled when index args field is enabled.
- Example:
  ```
  bufinfo resource(n)[ uav] dst
  ```

If `sec_modifier_present` bit is set to 1, the next Dword is the secondary modifier.

If the `indexed_args` bit is set to 1, the next Dword is the resource format token `ILPixTexUsage`.

If the `aoffset_present` bit is set to 1, the next Dword is the address offset.

For example:

```
sample_ext_resource(n)_sampler(m)[_resourcetype(pixtexusage)][_addroffimmi
(u,v,w)] dst, src0,src1, scr2
```

where:

`controls_resource` = n

`controls_sampler` = m

`indexed_args` = 1

final resource index = src1 + n

final sampler index = src2 + m

One extra Dword is required after the opcode token for resource format in Enum ILPixTexUsage.

7.2.4 Notes on Conversion Instructions

Even though IL is an untyped language, conversion instructions are required to convert between different data formats so that source data for an instruction appears in the proper format. Without format conversion, instructions could provide incorrect results.

7.2.5 Notes on Double Precision Instructions

Double precision values are represented by a pair of registers. Outputs are either the pair yx or to the pair wz, where the msb is stored in y/w. For example:

`Idata 3.0 => (0x4008000000000000)` in register `r` looks like:

```
r.w =  0x40080000  ;high Dword
r.z =   0x00000000 ;low  Dword
```

Or by:

```
r.y =  l0x40080000 ;high Dword
r.x =  l0x00000000 ;low  Dword
```

All source double inputs must be in xy (after swizzle operations). For example:

```
d_add r1.xy, r2.xy, r2.xy
```

Or

```
d_add r1.zw, r2.xy, r2.xy
```

Each computes twice the value in r2.xy, and places the result in either xy or zw.

The user can set the output mask to either xy or zw. The msb is in y/w, so users can test the sign of the result with single precision operations.

All inputs are in the first two components xy of each source.

These instructions are supported on RX70 GPUs (R670, R770, Cypress, etc.).

7.2.6 Notes on Arithmetic Instructions

7.2.6.1 IAND, IOR, IXOR, INOT

It is often useful to treat a vector element as if it were a vector of 32 individual bits. The IL language provides a set of logical operations which operate simultaneously on each bit of an element. Each of these operations reads the components of the sources, applies the operation to each separate bit, and writes the 32-bit result into the corresponding component of dst.

Logical NOT computes the 1's complement of each 32-bit value in *src0*.

Logical operations do not support input or output modifiers.

7.2.6.2 Simple Arithmetic Instructions

The IL provides a set of simple arithmetic operations. Each of these operations reads the components of the sources(*src0*, *src1*, etc.), applies the operation, and writes the 32-bit result into the corresponding component of *dst*. For all of these operations, the control field must be zero. Integer and unsigned integer operations are available on all GPUs, starting with R6XX series. Double operations are available on all GPUs that support double starting, with the R7XX series.

7.2.7 Notes on Shift Instructions

The IL language provides the usual set of arithmetic and logical shift operations. Shifts are supported on r600 and later chips. All instructions in this section require that the control `_eld` be zero. In each case, the amount to shift ignores all but the lower 5 bits of *src1*; so, shifts of 33 and 1 are treated identically.

Corresponding to the integer shift operations, the AMD IL provides 64-bit logical and arithmetic shift operations. These instructions read from the xy components of the `_rst` source and the x component of the second source; they can write to either *dst.xy* or *dst.zw*. In all these instructions, the amount to shift ignores all but the lower 6 bits of *src1* (the shift amount is in the range of 0-63).

7.2.8 Notes on Simple 64-Bit Integer Instructions

Corresponding to the simple arithmetic instructions, IL provides a set of 64-bit integer instructions. These instructions require that the control field be zero. Each of these operations reads from the xy components of each source and can write to either *dst.xy* or *dst.zw*.

7.2.9 Notes on Bit Operations

A common operaton on elements is packing and unpacking of bit strings. The IL provides these operations to access bit and byte strings within elements.

7.2.10 Note on LDS Memory Operations

Each processor has some local memory that can be shared across the work-items in a work-group. IL provides two models of memory access to the local data share (LDS). The first, called owner-computes, is supported on R7XX GPUs. In owner-computes, each work-item in a work-group owns a part of LDS memory. The size of the part is declared in the shader. Each work-item in a work-group can only write to the part of memory it owns; however, a work-item can read any part of memory owned by itself or by other work-items. Thus, an LDS shared memory read is specified by the owning work-item ID and offset, which indicates reading the part of memory owned by the work-item ID with an offset within that part.

7.2.10.1 LDS Access

The access for work-items within a wavefront differs from the access for different wavefronts (within a work-group); it is specified by the so-called Sharing-Mode (relative versus absolute: if the sharing mode is relative, new and consecutive space is allocated for each wavefront; if it is absolute, all wavefronts are mapped to the same set of memory starting at address 0). In this mode, wavefronts can overwrite each other's data.

Owner-computes is a legacy mode; programmers are expected to move to random access when possible.

The second compute model is a general read/write. Each work-item can read or write any address in the LDS. This model is supported on Evergreen GPUs and later.

Both models allow work-items to read or write memory (video or system), but do not provide synchronization to memory.

IL provides two ways to allocate LDS memory.

- Owner-compute, which allocates addreseses in the LDS for each work-item.
- Random Access, which allocates the LDS independent of work-items.

Each style has read and write operations. It is not valid to mix and match. For example, using owner-compute allocate and random acess read is not expected to work correctly.

7.2.10.2 LDS Programming Model

The Evergreen series of GPUs adds much functionality to the LDS and removes most of the R7XX series of GPU's LDS instructions. The programming model consists of the following.

1. There is one LDS per SIMD. Dx11 calls this group (or 'g') memory.
2. Work-items are organized in communicating units called *work-groups*.
3. All addresses to LDS are relative to the work-group; thus, no work-item can read data from different work-group.
4. Most LDS operations read or write Dwords from LDS memory; however, the address is in bytes.
5. When a work-group starts, the LDS memory is not in a known state; the application code must initialize it.
6. LDS references can be used only in compute shaders.
7. Both pixel and compute shaders can reference memory; DX11 calls this UAV memory.

The current `dcl_num_threads_per_group` declares the number of work-items in a group. This statement is required in a kernel that uses LDS.

7.2.10.3 LDS Operations

Starting with the Evergreen series of GPUs, IL supports a large number of binary atomic LDS operations. Each operation reads *v* as a scalar (32-bit) old value at a given LDS location, then combines *v* and *src1.x* using a specified operation, and stores the result back into LDS.

There are addtional atomic read and op versions that return the orignal value of *v*.

If multiple work-items execute the same atomic operation, then IL does not guarantee any specific ordering. Even repeated executions of the same sequence of instructions need not produce repeatable answers.

The operation is a 32-bit integer ADD.

If LDS is declared `typeless`, *src0.x* specifies a byte address relative to the work-group. The address must be aligned to a Dword (the lower two bits of the address must be zero).

```
a = src0.x/4
lds[a] += src1.x
```

If the LDS is declared as a `struct`, *src0.x* specifies the index into the array, *src0.y* specifies the offset into the `struct`. The offset is in bytes.

```
a = (src0.x *lds_stride + src0.y)/4
lds[a] += src1.x
```

7.2.11 Notes on GDS Memory Operations

Starting with the Evergreen family of GPUs, each processor has global memory that can be shared across all SIMDs. This provides a general read/write model, where each work-item can read or write any address in the GDS.

The programming model for GDS memory operations is:

- There is one GDS across all SIMDs.
- Most GDS operations read or write Dwords from or to GDS memory; however, the address is in bytes.
- When an application starts, the GDS memory is not in a known state, so application code must initialize it.
- Both pixel and compute shaders can reference GDS memory.

Starting with the Evergreen family of GPUs, IL supports a large number of binary atomic GDS operations. Each operation reads *v*, which is a scalar (32-bit) value at a given GDS location, combines *v* and `src1.x` using a specified operation, and stores the result back into GDS.

There are additional atomic read and op versions that return the original value of *v*.

If multiple work-items execute the same atomic operation, IL does not guarantee a specific ordering. Even repeated executions of the same sequence of instructions need not produce repeatable answers.

Starting with the Evergreen family of GPUs, IL supports a large number of binary atomic GDS operations. Each operation reads *v*, a 32-bit scalar value at a given GDS location, combines *v* and `src1.x` using a specified operation, and returns *v*.

If multiple work-items execute the same atomic operation, then IL does not guarantee any specific ordering. Even repeated executions of the same sequence of instructions need not produce repeatable answers.

7.2.12 Notes on UAV Memory Operations

IL supports a general read/write memory called unordered access view (UAV). UAV memory can be declared in three ways.

1. A UAV can be raw: data is 32-bit typeless, and addressing is linear using a single integer.
2. A UAV can be specified to have a data format composed of items with *dimension type*. For example, the data format could be 2D with dimension type `float`.
3. A UAV can be structured. As an array of structures with addressing using two integers, the first is an array index, the second an offset into the structure.

Note: The R7XX series of GPUs allows only one UAV. This UAV cannot be typed.

Similar to the atomic operations on the LDS, IL provides a set of atomic operations on UAV memory. Each operation reads *v* as a scalar (32-bit) old value at a given memory location, then combines *v* and *src1.x* using a specified operation, and stores the result back into memory.

An atomic single-component integer add to uav: `uav[src0] += src1.x`.

There are addtional atomic read and op versions that return the orignal value of *v*.

If multiple work-items execute the same atomic operation, then IL does not guarantee any specific ordering. Even repeated executions of the same sequence of instructions do not necessarily produce repeatable answers.

Not all UAVs can be used with atomic operations. To use an atomic operation, the UAV with ID n must have been declared as `raw`, `structured`, or `typed`. If the UAV was declared to be `typed`, it must have been declared as `R32 UINT` or `SINT`.

The register *src0* provides the address. If `raw`, *src0.x* provides the address in bytes; if `structured`, *src0.xy* provides the address of `struct index` and the offset in bytes.

```
a = src0.x/4
uav[a] += src1.x
```

If typed, the number of components used for the address depends on the dimension of the UAV. For examples, for Texture1D Arrays, *src0.x* provides the buffer address, and *src0.y* provides the index/offset of the array. For typed UAV access, the address is in elements (Dwords).

```
a = src0.x
uav[a] += src1.x
```

If the UAV is declared as a `struct`, *src0.x* specifies the index into the array, *src0.y* specifies the offset into the `struct`. The offset is in bytes.

```
a = (src0.x*lds_stride + src0.y)/4
uav[a] += src1.x
```

The register *src1.x* provides a 32-bit Dword.

The 32-bit UAV memory specified by the address in *src0* is updated atomically by `iadd(uav[src0], src1.x)`. Nothing is returned.

Instructions with out-of-range addresses write nothing to the UAV surface, with the exception that for structured UAVs, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the kernel invocation is inactive, nothing is written to the UAV surface.

The arena modifier differentiates between the regular UAV and arena UAV IDs. For example: `uav_add_id(1) r0.x, r1.x`, and `uav_add_id(1)_arena r0.w, r1.x` . When an atomic operation is performed on an arena UAV, the data size must be a Dword. Arena UAVs are supported only on Evergreen and Northern Island GPUs; the arena can be only UAV 8.

7.2.13 Notes on Multi-Media Instructions

These instructions support fast multi-media operations. They are available on Evergreen GPUs and later. No input or output modifiers are supported for these instructions.

The four unpack operations can be used extract a single byte from each register component and then convert that byte to float.

7.2.14 Notes on Evergreen GPU Series Memory Controls

The Evergreen GPU series adds much functionality to the local data share (LDS); also, it removes most of the LDS instructions for the R7XX GPU series.

The programming model is described in the following.

1. There is one LDS per SIMD. Dx11 calls this ‘g’ memory.
2. Work-items are organized in communicating units called work-groups.
3. All addresses to LDS are relative to the work-group; no work-item can read data from different work-group.

4. Most LDS operations read or write Dwords from LDS memory; however, the address is in bytes.
5. When a work-group starts, the LDS memory is not in a known state; thus, the application code must to initialize it.
6. LDS references can be used only in compute shaders.
7. Both pixel and compute shaders can reference memory. DX11 calls this UAV memory.

The current `dcl_num_threads_per_group` is used to declare the number of work-items in a work-group, This statement is required in a shader that uses LDS.

7.3 Prefix Instruction

A prefix instruction specifies additional controls for the next instruction. This instruction takes no inputs or outputs. Its control field specifies attributes that must be applied to the immediately following IL instruction.

OpCode Specifier *IL_Prefix_OpCode*

code	15:0
precise x	16:16
precise y	17:17
precise z	18:18
precise w	19:19
reserved	31:20

code	Must be set to IL_OP_PREFIX
precise x/y/z/w	Per-component precise control. If set to 1, the operation on the given component in the following instruction must remain precise (not refactorable). This control overrides `REFACTORING_ALLOWED` declared in the global flag.

Valid for Evergreen GPUs and later.

Usage

If components of a MAD instruction are tagged as PRECISE, the hardware must execute a MAD *orexactequivalent*, and cannot split it into a multiply followed by an add. Conversely, a multiply followed by an add, where either or both are flagged as PRECISE, cannot be merged into a fused MAD.

This affects any operation, not just arithmetic.Take the following sequence of instructions as an example.

1. Write the value of the variable `foo` to memory address x in a UAV.
2. ... (execute any sequence of instructions)
3. Read from memory address x in the UAV.

If `REFACTORING_ALLOWED` is present, the above sequence of instructions can be optimized so that the value normally be read from address X is replaced with the variable `foo` instead. This optimization does not occur if a memory fence operation is requested between the write and the read. If `REFACTORING_ALLOWED` is not declared for the shader, or if it is present but the read x is marked as PRECISE, the compiler/drivers must leave the read as is. This can reveal a behavior difference between the optimized version and the PRECISE version. For example, if memory address x is out of bounds of the UAV, the write does not happen, and the read out-of-bounds has some other well-defined behavior; thus, the read does not produce `foo`.

Text Syntax

In text, the prefix is specified on its following instruction. It takes two formats:

- `_prec`: when precise x/y/z/w are all set.
- `_precmask`: where mask contains that channels enabled in precise x/y/z/w.

Examples

Note that prec modifier is always added immediately following the opcode.

`add_prec r0.xyz , r1, r2` — all output channels must remain precise.

`add_prec(y) r0.xyz , r1, r2` — result in y must remain precise.

`uav_load_prec_id(1) r1, r2` — all output channels of the load must remain precise.

7.4 Flow Control Instructions

Unconditional BREAK out of Loop or Switch Constructs

Instructions **BREAK**

Syntax `BREAK`

Description BREAK is used only in a loop or switch statement. It terminates the smallest enclosing loop or switch. Control passes to the statement following the terminated statement, if any.

Valid for all GPUs.

Format 0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_BREAK
reserved	31:16	Must be zero.

Related BREAKC, BREAK_LOGICALZ, BREAK_LOGICALNZ.

Conditional BREAK Using Floating Point

Instructions **BREAKC**

Syntax `breakc_relop(`*op*`)` *src0*, *src1*

Description BREAKC compares two registers using a float comparison on the x component of the sources. It is used only in a loop or switch statement. It terminates the smallest enclosing loop or switch. Control passes to the statement following the terminated statement, if any.

Valid for all GPUs.

Operation:

```
if (v0.x relop v1.x) {
    Break;
}
```

Format 2-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_BREAKC
`control`	18:16	relop(*op*). See Table 6.23 on page 6-18.
reserved	31:19	Must be zero.

Related BREAK, BREAK_LOGICALZ, BREAK_LOGICALNZ.

Conditional BREAK Using Integers

Instructions **`BREAK_LOGICALNZ, BREAK_LOGICALZ`**

Syntax

Opcode	Syntax	Description
`IL_OP_BREAK_LOGICALNZ`	`break_logicalnz` *`src0`*	If *src0*.x ≠ 0 break.
`IL_OP_BREAK_LOGICALZ`	`break_logicalz` *`src0`*	If *src0*.x == 0 break.

Description This form of Break compares a register component using integer compare to zero. The break is done if all bits of *src0* are zero. *src0* must have a swizzle that selects a single component. The break statement occurs only in a loop-*statement* or a switch statement. It causes termination of the smallest enclosing loop or switch statement; control passes to the statement following the terminated statement, if any.

Valid for all GPUs.

Format 1-input.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
reserved	31:16	Must be zero.

Related BREAK, BREAKC.

Unconditional CALL

Instructions **`CALL`**

Syntax `call` *`<integer label>`*

Description Call a subroutine. The subroutine is identified by a label id, corresponding to the ID in a function statement. Subroutines can nest up to 32 levels deep. Direct or indirect recursion is permitted. If there are already 32 entries on the return address stack, and a "call" is issued, the call is skipped. The opcode token is followed immediately by a Dword containing the label ID of the subroutine.

Valid for all GPUs.

.Ordinal	Token
1	IL_Opcode token with code set to IL_OP_CALL (control field ignored).
3	Unsigned integer representing the subroutine label.

Format 0-input.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_CALL
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Must be zero.		
3	Unsigned integer representing label of the subroutine.		

Related CALL_LOGICALNZ, CALL_LOGICALLZ, CALL_LOGICALNZ

CALL if Boolean register is not zero

Instructions	**`CALLNZ`**
Syntax	`callnz` *`src0, <integer label>`*

Description

CALLNZ compares *src0*.x to zero using an integer comparison. If any bits of *src0*.x are not zero, CALLNZ pushes the address of the next instruction in the shader onto the return address stack and transfers control to the FUNC block identified by <*integer label*>. CALLs can be nested up to 32 levels deep. If all bits of *src0*.x are zero, or the return address stack already contains 32 addresses, the CALL is skipped and execution continues at the next instruction in the shader. Recursion is permitted either directly, using another CALL instruction, or indirectly using relative addressing. The opcode token is followed immediately by a Dword containing the label ID of the subroutine.

src0 must be a CONST_BOOL register. See Chapter 5 for information on the register types.

CALLNZ cannot be used with relative addressing or a source modifier. Both the `modifier_present` and `relative_address` fields of the IL_Src token must be zero.

Valid for all GPUs.

Operation:

```
if (v0.x != 0) {
Call label
}
```

Format 1-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_CALLNZ
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

2 IL_Src token (*src0)* with `register_type` set to IL_REGTYPE_CONST_BOOL. The `modifer_present` and `relative_address` field must be set to 0.

Field Name	Bits	Description
`register_num`	15:0	Any available register.
`register_type`	21:16	CONST_BOOL
Remaining fields	31:22	Must be zero.

3 Unsigned integer representing the label of the subroutine.

Related CALL, CALL_LOGICALZ, CALL_LOGICALNZ.

Conditional CALL Based on Integer

Instructions **`CALL_LOGICALNZ`**

Syntax `call_logicalnz,` *`src0`*

Description Conditional call subroutine at the label. The 32-bit register component (*src0*) is tested. If any bits are not zero, the instruction performs the call. It checks only the x component (after swizzling) of *src0*. If *src0*.x ≠ 0, call <integer label>. All four forms call a subroutine. The subroutine is identified by a label id, corresponding to the id in a function statement. Subroutines can nest up to 32 levels deep. Recursion is permitted, directly or indirectly. If there are already 32 entries on the return address stack, and a “call” is issued, the call is skipped. The opcode token is followed immediately by a Dword containing the label ID of the subroutine.

Valid for all GPUs.

Operation:

```
if (v0.x ne 0) {
      Call subroutine;
}
```

Format 1-input.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_CALL_LOGICALNZ
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Src token and IL_Src_Mod token (if required) representing any valid IL Source. See Section 2.2.6, “Source Token,” page 2-5.		
3	Unsigned integer representing the label of the subroutine.		

Related CALL, CALLNZ.

Conditional CALL Based on Boolean

Instructions **CALL_LOGICALZ**

Syntax `call_logicalnz src0 <integer label>`

Description Performs conditional call based on a Boolean constant. Scr0 must be a CONST_BOOL register. If *src0*.x == 0, call <*integer label*>. This instruction does not allow the use of a source modifier. The `modifier_present` field of the IL_Src token must be set to 0. This instruction does not allow relative addressing. The relative_address field of the IL_Src token must be set to 0.

The subroutine is identified by a label id, corresponding to the ID in a function statement. Subroutines can nest up to 32 levels deep. Recursion is permitted, directly or indirectly. If there are already 32 entries on the return address stack and a “call” is issued, the call is skipped. The opcode token is followed immediately by a Dword containing the label ID of the subroutine.

Valid for all GPUs.

.Ordinal	Token
1	IL_Opcode token with code set to IL_OP_CALL_LOGICALZ (control field ignored).
2	(*src0*) representing any valid IL Source (unspecified number of tokens).
3	Unsigned integer following all source tokens representing the subroutine label.

Operation:

```
if (v0.x eq 0) {
       Call subroutine;

}
```

Format 1-input.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_CALL_LOGICALZ
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Src token and IL_Src_Mod token (if required) representing any valid IL Source. See Section 2.2.6, “Source Token,” page 2-5.		
3	Unsigned integer representing the label of the subroutine.		

Related CALL, CALLNZ

CASE Statement

Instructions	**`CASE`**
Syntax	`case case-value`
Description	This case statement is used in a switch. This instruction is followed by a 32 integer that identifies the case. Compares are done using integer arithmetic. Falling through cases is valid, as in C. This can be used to implement a DX10 case instruction. See the SWITCH instruction for operation details. Valid for R6XX GPUs and later.
Format	1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CASE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related DEFAULT, ENDSWITCH, SWITCH.

Unconditional CONTINUE

Instructions **`CONTINUE`**

Syntax `continue`

Description CONTINUE causes shader execution to continue at the previous LOOP or WHILELOOP instruction. This is used only in a LOOP - ENDLOOP instruction block.

Valid for R6XX GPUs and later.

Operation:

```
If this is a counted loop {
  LoopIterationCount = LoopIterationCount - 1;
  LoopCounter = LoopCounter + LoopStep;
  LoopCounter = (LoopCounter > 0) ? LoopCounter : 0;
  if (LoopIterationCount > 0)
    Continue execution at the StartLoopOffset;
} else {
       Continue execution at the StartLoopOffset;
}
```

Format 0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CONTINUE
reserved	31:16	Must be zero.

Related CONTINUEC, CONTINUE_LOGICALZ, CONTINUE_LOGICALNZ.

CONTINUE Using Floating Point

Instructions	**`CONTINUEC`**
Syntax	`continuec_relop(`*`op`*`)` *`src0, src1`*

Description

Conditionally continues execution at previous LOOP or WHILELOOP instruction if the condition is true. Can only be used in a LOOP - ENDLOOP instruction block.

Valid for R6XX GPUs and later.

Operation:

```
if (v0.x relop v1.x) {
       Continue;
}
```

Format 2-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CONTINUEC
`control`	18:16	relop(*op*). See Table 6.23 on page 6-18.
reserved	31:19	Must be zero.

Related CONTINUE, CONTINUE_LOGICALZ, CONTINUE_LOGICALNZ.

Conditional CONTINUE Using Integers

Instructions **`CONTINUE_LOGICALNZ, CONTINUE_LOGICALZ`**

Syntax

Opcode	Syntax	Description
`IL_OP_CONTINUE_LOGICALNZ`	`continue_logicalnz` *`src0`*	If *src0*.x ≠ 0 call <integer label>.
`IL_OP_CONTINUE_LOGICALZ`	`continue_logicalnz` *`src0`*	If *src0*.x == 0 call <integer label>.

Description

Conditionally continues execution at the beginning of the current loop:
CONTINUE_LOGICAL_Z continues the loop if all bits of *src0*.x are zero.
CONTINUE_LOGICAL_NE continues the loop if any bit of *src0*.x is not zero.

Can only be within a LOOP - ENDLOOP switch block.

Valid for R6XX GPUs and later.

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
reserved	31:16	Must be zero.

Related CONTINUE, CONTINUEC.

DEFAULT Statement

Instructions	**`DEFAULT`**
Syntax	`default`

DEFAULT Statement

Description DEFAULT starts an instruction block within a SWITCH instruction block (see page 30). Unlike a CASE label, a DEFAULT label does not provide a value for comparison.

This is like the default in C. Falling through or into a DEFAULT section is valid. There can be only one DEFAULT statement in each SWITCH block.

Valid for R6XX GPUs and later.

Format 0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DEFAULT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related CASE, ENDSWITCH, SWITCH.

ELSE

Instructions **`ELSE`**

Syntax `else`

Description This instruction is the start of the ELSE clause of an IFNZ-ELSE-ENDIF or IFC-ELSE-ENDIF or LOG_IF-ELSE-ENDIF block. ELSE must be after an IFC, IFNZ, or LOG_IF instruction in the stream.

Valid for all GPUs.

Format 0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ELSE
reserved	31:16	Must be zero.

Related IFNZ, IFC, ENDIF, IF_LOGICALNZ, IF_LOGICALZ, ENDFUNC, END.

End of Stream

Instructions	**`END`**
Syntax	`end`
Description	END indicates the end of an IL stream and must be the last statement in the stream. All shader programming, including subroutines, must be placed before this instruction. Valid for all GPUs.
Format	0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_END
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related ENDFUNC, DEFAULT, ENDIF, IF_LOGICALZ, IF_LOGICALNZ, IFC, IFNZ, ENDMAIN.

End of FUNC (subroutine)

Instructions	**`ENDFUNC`**
Syntax	`endfunc`
Description	ENDFUNC indicates the end of a shader subroutine. Only FUNC blocks and the END statement can follow an ENDFUNC statement. This instruction is required only if the shader contains subroutines. Valid for all GPUs.
Format	0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ENDFUNC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related FUNC, END, ENDIF, IF_LOGICALZ, IF_LOGICALNZ, IFC, IFNZ, ENDMAIN.

End of IF block

Instructions	**ENDIF**
Syntax	endif
Description	Indicates the end of an IFNZ - ENDIF, IFC - ENDIF, IFNZ - ELSE - ENDIF, or IFC - ELSE - ENDIF block. The ENDIF statement must follow an ELSE, IFC, IFNZ or LOG_IF instruction. Valid for all GPUs.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ENDIF
reserved	31:16	Must be zero.

Related ELSE, IFC, IFNZ, IF_LOGICAL*, END, ENDFUNC, IFC, IFNZ.

End of a LOOP or WHILELOOP block

Instructions **ENDLOOP**

Syntax endloop

Description Indicates the end of a LOOP - ENDLOOP or WHILELOOP - ENDLOOP instruction block. The instruction must follow a LOOP/WHILELOOP instruction. This instruction cannot be with a FUNC - RET, IFNZ - ENDIF, IFC - ENDIF, IFNZ - ELSE, IFC - ELSE, or ELSE - ENDIF block unless its corresponding LOOP or WHILELOOP is also in that block.

Valid for all GPUs.

Operation:
(For counted loops)

```
LoopIterationCount = LoopIterationCount - 1;
LoopCounter = LoopCounter + LoopStep;
LoopCounter = (LoopCounter > 0) ? LoopCounter : 0;
if (LoopIterationCount > 0)
       Continue execution at the StartLoopOffset;
```

For While loops

```
   Transfer control back to the test of the while loop.
```

Format 0-input.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ENDLOOP
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related LOOP, WHILELOOP.

End of Main Program

Instructions	**ENDMAIN**
Syntax	`endmain`
Description	ENDMAIN indicates the end of the shader source for the main program. Only FUNC instruction blocks and the END statement (see page 21) can follow an ENDMAIN statement. An ENDMAIN statement is only required if the shader contains subroutines. Valid for all GPUs. Operation: End of shader execution, and beginning of subroutine definitions.
Format	0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ENDMAIN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related END, ENDFUNC.

End of Hull Shader CP/Fork/Join Phase

Instructions	**ENDPHASE**
Syntax	`endphase`
Description	Marks the end of execution of a hull shader control-point, fork, or join phase. Valid for Evergreen GPUs and later.
Format	0-input.

Opcode

Field Name	Control Field
`IL_OP_ENDPHASE`	Must be zero.

Related HS_CP_PHASE, HS_FORK_PHASE, HS_JOIN_PHASE.

End SWITCH Instruction Block

Instructions **`ENDSWITCH`**

Syntax `endswitch`

Description ENDSWITCH indicates the end of a SWITCH instruction block.

Valid for R6XX GPUs and later.

Format 0-input.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ENDSWITCH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related CASE, DEFAULT, SWITCH.

Start of a FUNC Instruction Block

Instructions **`FUNC`**

Syntax `func <`*`integer label`*`>`

Description Indicates the start of a subroutine. Each FUNC statement must have a unique label, <*integer-label*>. A RET instruction must be before the END instruction. This instruction can be used only after an ENDMAIN instruction. The code between FUNC and RET is executed if a CALL or CALLNZ with the same label value is executed.

Valid for all GPUs.

Operation:

Defines the lexical start of a subroutine.

Format 0-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_FUNC
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Unsigned integer representing the label of the subroutine.		

Related ENDFUNC, END.

Start Control-Point Phase of Hull Shader

Instructions **`HS_CP_PHASE`**

Syntax `hs_cp_phase`

Description Marks the beginning of a hull shader control-point phase. Appears only in the main body of the hull shader.

Valid for Evergreen GPUs and later.

Format 0-input.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_HS_CP_PHASE
	`control`	31:16	Must be zero.

Related IL_OP_HS_FORK_PHASE, IL_OP_HS_JOIN_PHASE, ENDPHASE.

Start Fork Phase of Hull Shader

Instructions **`HS_FORK_PHASE`**

Syntax `hs_fork_phase n`

Description Marks the beginning of a hull shader fork phase. Appears only in the main body of the hull shader.

Valid for Evergreen GPUs and later.

Format 0-input.

Opcode

Field Name	Control Field
`IL_OP_HS_FORK_PHASE`	Number of instances.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_HS_FORK_PHASE
	`control`	31:16	Number of instances.

Related IL_OP_HS_CP_PHASE, IL_OP_HS_JOIN_PHASE, ENDPHASE.

Start Join Phase of the Hull Shader

Instructions **HS_JOIN_PHASE**

Syntax `hs_join_phase n`

Description Marks the beginning of a hull shader join phase. Appears only in the main body of the hull shader.

Valid for Evergreen GPUs and later.

Format 0-input.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_HS_JOIN_PHASE
	control	31:16	Number of instances.

Related IL_OP_HS_CP_PHASE, IL_OP_HS_FORK_PHASE, ENDPHASE.

Conditional IF Using Integers

Instructions **IF_LOGICALNZ, IF_LOGICALZ**

Syntax

Opcode	Syntax	Description
IL_OP_IF_LOGICALNZ	if_logicalnz *src0.x*	If *src0*.x ≠ 0 execute instruction block.
IL_OP_IF_LOGICALZ	if_logicalz *src0.x*	If *src0*.x == 0 execute instruction block.

Description These are integer versions of the IF statement. They skip a block of code based on the value of *src0*.x. The LOG_IF block must end with and ELSE or ENDIF instruction. The source selector must replicate the component to be tested into all four components. The test uses integer tests, so values like Nan or -0 are not equal to zero.

Valid for R6XX GPUs and later.

Operation for IF_LOGICALNZ:

```
if (v0.x has any bit non-zero) {
    Execute following instructions;
} else {
    Jump to the instruction following the next ELSE or ENDIF instruction;
}
```

Operation for IF_LOGICALZ:

```
if (v0.x has all bits zero) {
    Execute following instructions;
} else {
    Jump to the instruction following the next ELSE or ENDIF instruction;
}
```

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
code	15:0	See *Syntax*, above.
reserved	31:16	Must be zero.

Related IFC, IFNZ, IF_LOGICAL*, ELSE, ENDIF.

Conditional IF Using Floating Point

Instructions	**IFC**
Syntax	`ifc_relop(op) src0, src1`

Description

This is the start of an IFC - ENDIF or IFC - ELSE - ENDIF block. It skips a block of code based on the value of *src0* compared to *src1*. The IFC block must end with and ELSE or ENDIF instruction.

Valid for all GPUs.

Operation:

```
if (v0.x relop v1.x ) {
    Execute following instructions;
} else {
    Jump to the instruction following the next ELSE or ENDIF instruction;
}
```

Format 2-input.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_IFC
control	18:16	relop(*op*). See Table 6.23 on page 6-18.
reserved	31:19	Must be zero.

Related IF_LOGICALNZ, IF_LOGICALZ, IFNZ, ELSE, END, ENDIF.

Execute instruction block if Boolean register is not zero

Instructions	**IFNZ**
Syntax	`ifnz src0`

Description

This starts an IFNZ - ENDIF or IFNZ - ELSE - ENDIF block. IFNZ cannot be used with relative addressing or a source modifier. Both the `relative_address` and `modifier_present` fields of the IL_Src token must be zero. An IFNZ block must end with an ELSE or ENDIF instruction.

Valid for all GPUs.

Operation:

```
if (v0.x) {
    Execute following instructions;
} else {
    Jump to the instruction following the next ELSE or ENDIF instruction;
}
```

Format 1-input.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_IFNZ
reserved	31:16	Must be zero.

Related IFC, IF_LOGICALNZ, IF_LOGICALZ, ELSE, END, ENDIF.

Start of a Counted LOOP Block

Instructions **LOOP**

Syntax `loop [repeat] src0`

Description LOOP indicates the start of a LOOP instruction block. *src0* must be a register of type IL_REGTYPE_CONST_INT. *src0*.x specifies the iteration count used for the loop. If repeat is set to 0, *src0*.y specifies the initial value for the current *loop-counter* used for relative addressing. If repeat is set to 1, *src0.y* and *src0*.z are not used, and the current auto-increment loop counter is not incremented during this loop. Otherwise, the three components src0.xyz are interpreted as unsigned eight-bit integers. The special register aL is initialized to src0.y when the loop starts, and incremented by src0.z each loop iteration. The register aL then can be used to index V or O registers.

The loop initial value and loop iteration count cannot be negative.

This instruction starts a LOOP - ENDLOOP block.

If this instruction is within a FUNC - RET, IFNX - ENDIF, IFX - ENDIF, IFNZ - ELSE, IFC - ELSE, or ELSE - ENDIF block, its corresponding ENDLOOP must also be within that block.

This instruction is provided for iteration. It only increments the current auto-incremented loop counter if repeat is set to 0.

ENDLOOP must follow the last instruction of a loop block. The ENDLOOP instruction offset must be greater than the LOOP instruction offset.

If repeat is set to 0, the loop initial value and the loop iteration count cannot be negative.

LOOP cannot be used with relative addressing or a source modifier. Both the `relative_address` and `modifier_present` fields of the IL_Src token must be zero.

Valid for all GPUs.

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LOOP
`rep`	16	*repeat* flag 0 *src0*.y holds the initial value for the current loop-counter used for relative addressing. *src0*.z holds the loop step. *src0*.y and *src0*.z cannot be negative. 1 *src0*.y and *src0*.z are not used and the current auto-increment loop-counter is not incremented during this loop. *src0*.y and *src0*.z can be negative.
`control_reserved`	28:17	Must be zero.
reserved	29	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related ENDLOOP, WHILELOOP.

RETURN from a FUNC Block (end of subroutine)

Instructions **`RET`**

Syntax `ret`

Description Returns from a subroutine, and indicates the end of the subroutine. It marks the end of a subroutine and can be present only at the end (it cannot be within a flow-control block). This instruction can implement DX9 returns.

Valid for all GPUs.

Operation:

Continue execution after the call statement which executed this subroutine.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_RET
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RET_DYN, RET_LOGICALNZ, RET_LOGICALZ.

RETURN from a FUNC Block (not end of subroutine)

Instructions **`RET_DYN`**

Syntax `ret_dyn`

Description Returns from a subroutine to the instruction after the call. It can appear anywhere in a subroutine, any number of times.

Valid for R600 GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_RET_DYN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RET, RET_LOGICALNZ, RET_LOGICALZ.

Conditional RETURN from FUNC Block Using Integer

Instructions **RET_LOGICALNZ, RET_LOGICALZ**

Syntax

Opcode	Syntax	Description
IL_OP_IF_LOGICALNZ	if_logicalnz *src0.x*	If *src0*.x ≠ 0 execute instruction block.
IL_OP_IF_LOGICALZ	if_logicalz *src0.x*	If *src0*.x == 0 execute instruction block.

Description These instructions conditionally return tot he instruction after the call. *src0*.x is tested after swizzle. The instructions can appear anywhere in a subroutine, any number of times.

The 32-bit value supplied by *src0* is tested at the bit level:

For RET_LOGICALNZ, if any bit is non-zero, the statement returns.
For RET_LOGICALZ, if all bits are zero, the statement returns.

Valid for R6XX GPUs and later.

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
code	15:0	See *Syntax*, above.
control	29:16	logic_op
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related RET, RET_DYN.

Start of a SWITCH Block

Instructions **SWITCH**

Syntax switch *src0*

Description A switch/endswitch construct behaves exactly as a switch construct in the C language. The *src0* must be a 32-bit register component or immediate quantity. Compares are done using integer arithmetic. Falling through cases are valid, as in C. This instruction can be used to implement DX10 case instruction. Switch statements can be nested without limits.

Valid for R6XX GPUs and later.

Operation:

Same as a C switch statement.

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_SWITCH
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related CASE, DEFAULT, ENDSWITCH.

Start of a WHILELOOP Block

Instructions	**`WHILELOOP`**
Syntax	`whileloop`
Description	WHILELOOP indicates the start of a WHILELOOP instruction block. A WHILELOOP block can iterate indefinitely, exiting only when a BREAK instruction is executed. It can be used as the translation of a DX10 loop statement. There is no limit to the amount of loop nesting. Although a WHILELOOP can iterate indefinitely, overall execution of the shader can be terminated after some number of instructions are executed. Valid for R600 GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_WHILE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related ENDLOOP, LOOP.

7.5 Declaration and Initialization Instructions

Declare a Constant Buffer

Instructions	**`DCL_CB`**
Syntax	`dcl_cb` *`cbm[n]`*
Description	Declares that the shader can use the constant buffer, *cbm*, or size *n* registers. This instruction must occur before any use of the constant buffer. The source of this instruction must be of type IL_REGTYPE_CONSTANT_BUFF. The cb index m can be in the range of 0-14. The index n can be up to 4096 (4k). DCL_CB cannot be used with relative addressing or a source modifier. Both the `modifier_present` and `relative_address` fields of the IL_Src token must be zero. Buffer Modifier: If the instruction is being used to specify an immediate constant buffer, set the `pri_modifier_present` bit to 1. The subsequent 32 bits are an unsigned integer specifying the number of elements in the constant buffer, followed by one 32-bit floating point value specifying each member of the immediate constant buffer. Valid for R600 GPUs and later.
Format	1-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_CONST_BUFFER
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	0: Instruction declares a constant buffer. 1: Does not declare a constant buffer. Instruction declares an Immediate constant buffer.
2	`pri_modifier_present` == 0		IL_Src token (*src0*).
	`pri_modifier_present` == 1		Number of elements, n, in an immediate constant buffer.
3	`pri_modifier_present` == 0		Not used.
	`pri_modifier_present` == 1		First (index 0) 32-bit element of the immediate constant buffer.
4 to n+2	`pri_modifier_present` == 0		Not used.
	`pri_modifier_present` == 1		Second (index 1) through nth (index n-1) 32-bit elements of the immediate constant buffer.

Related None.

Declare Global Flags

Instructions	**`DCL_GLOBAL_FLAGS flag1	flag2	...`**
Syntax	`dcl_input_primitive prim_type(op)`

Description

Declares shader global flags.

Example:

```
dcl_global_flags refactoringAllowed |
    forceEarlyDepthStencil |
    enableRawStructuredBuffers
```

The `refactoring_allowed` parameter is valid for R6XX GPUs and later.

The `force_early_depth_stencil` parameter is valid for Evergreen GPUs and later.

The `enable_raw_structured_buffers` parameter is valid for Evergreen GPUs and later.

The `enable_double_precision_float_ops` parameter is valid for Evergreen GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`refactoring_allowed`	0	Valid for R6XX GPUs and later.
`force_early_depth_stencil`	1	Valid for Evergreen GPUs and later.
`enable_raw_structured_buffers`	2	Valid for Evergreen GPUs and later.[1]
`enable_double_precision_float_ops`	3	Valid for Evergreen GPUs and later.[1]
reserved	31:4	

Related None.

1. Currently not used. Given an IL shader that uses raw or structured buffers, the shader compiler compiles it if the underlying hardware supports it; otherwise, it fails.

Declare an Indexed Array

Instructions	**`DCL_INDEXED_TEMP_ARRAY`**
Syntax	`dcl_indexed_temp_array` *`src0`*`[`*`n`*`]`
Description	Declares a temporary array. Each indexed temp array to be used in the Shader must be declared. *src0* must be of type IL_REGTYPE_ITEMP. The modifier_present and relative_address fields must be zero. `n` is the maximum size in 128-bit units. DX10 limits the total storage for temps (indexed plus non-indexed) to be ≤ 4096 registers (each a four-component vector). Valid for R600 GPUs and later.
Format	1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_INDEXED_TEMP_ARRAY
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare an Input Register

Instructions **`DCL_INPUT`**

Syntax `dcl_input[_usage(`*`usage`*`)][_interp(`*`mode`*`)]` *`dst`*`[.mask]`

Description Declares an input register for the shader. *`dst`* must be an IL_REGTYPE_INPUT register type (see Chapter 5, “Register Types”) and can have a mask. It is legal to declare a subset of the component mask from what is output from the previous shader in the pipeline; however, mutually exclusive masks are not allowed (for example: VS outputting o3.xy means PS declaring v3.z input is invalid, but v3.x, v3.y, or v3.xy is legal). Interpolation mode, _interp(mode), is only applicable to pixel shaders; it is an error to use this in GS or VS.

DCL_INPUT is used for shader model 4.0 and above. which supports perspective, linear, and interpolation. Do not use DCL_INPUT and DCLPI in the same shader.

Valid for R600 GPUs and later.

Possible values for _interp(mode) are:

IL_INTERPMODE_NOTUSED	IL_INTERPMODE_LINEAR_CENTROID
IL_INTERPMODE_CONSTANT	IL_INTERPMODE_LINEAR__NOPERSPECTIVE
IL_INTERPMODE_LINEAR	IL_INTERPMODE_LINEAR_SAMPLE

IL_INTERPMODE_LINEAR_NOPERSPECTIVE_CENYTROID
IL_INTERPMODE_LINEAR_NOPERSPECTIVE_SAMPLE

Converting from DirectX 10:

In the DX statment `dcl_input_generic vicp[1][15]`, the first dimension [1] is the size. There is one element in the array, while the second dimension [15] is the attribute number - attribute 15. The dcl statement declares vicp attribute 15 as an array of one element. When referencing the register `mov o15, vicp[0][15]` means to move the first element of attribute 15.

If the declaration is `dcl_input_generic vicp[3][15]`, then one can use `vicp[0][15]` or `vicp[1][15]` or `vicp[2][15]` as the source.

This is the rule used by geometry shader input arrays, for example: `dcl input generic v[3][2]`.

Valid for R600 GPUs and later.

Format

Field Name	Bits	Description
`code`	15:0	IL_DCL_INPUT
`control`	20:16	*usage*. Unless using system value (SV), must be IL_IMPORTUSAGE_GENERIC (see Table 6.10 on page 6-5). The usage index is specified by the register number.
`interp mode`	23:21	Interpolation mode.
`center`	24	This field is always ignores. Center vs centroid must be specified through _interp(mode).

Declare an Input Register (Cont.)

	`bias`	25	This is ignored, unless the *dst* type is wincoord (i.e., IL_IMPORTUSAGE_POS). For the window coordinate register, this bit specifies if the compiler generates code that adds a bias supplied in constants that are named SC_CONS_SRC_VIEWPORT_BIAS_{X,Y}. Without a bias, the origin is located at the upper-left corner of the screen (DX Style). A bias can be used to move the origin. For example, to move the origin to the lower-left corner of a window (OGL style), DX must set this to 0, OGL must set this to 1. The compiler expects the bias values in the constant file entry. All AMD hardware supports either constant files or constant buffers, not both. Therefore, it is an errror to mix the use of bias and constant buffers. Clients that need bias and constant buffers must adjust the source shader.
	`invert`	26	This is ignored, unless the *dst* type is wincoord. If the register_type field of the following IL_DstToken is IL_REGTYPPE_WINCOORD, and this bit is set, then the compiler inverts (multiplies by -1) the y coordinate of the position. By default, the y axis goes down (DX style), but this bit can be used to change the direction to up (OGL style).
	`centered`	27	This is ignored, unless the *dst* type is wincoord. For the window coordinate register (i.e., IL_IMPORTUSAGE_POS), this bit controls (value 0) defines the origin as the upper-left corner of the RenderTarget. Thus, pixel centers are offset by (0.5f,0.5f) from integer locations on the RenderTarget. This choice of origin makes rendering screen-aligned textures trivial, as the pixel coordinate system is aligned with the texel coordinate system. This choice matches OGL and DX10. The second value (1) defines the origin as the center of the upper-left pixel in the RenderTarget. In other words, the origin is (0.5,0.5) away from the upper left corner of the RenderTarget. Thus, pixel centers are at integer locations. This choice of origin matches DX9. 0 – center of the pixel is 0.5,0.5 1 – center of the pixel is 0,0
	`reserved`	29:28	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
Related	None.		

Declare a Primitive

Instructions **`DCL_INPUTPRIMITIVE`**

Syntax `dcl_input_primitive prim_type(op)`

Description Declare the type of primitive that can be accepted by a geometry shader. Must appear in a geometry shader. Only valid in a geometry shader. The `prim_type` must be an element of the enumeration IL_TOPOLOGY.

Valid for R600 GPUs and later.

Format 0-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_INPUTPRIMITIVE
	`control`	29:16	`prim_type`.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Src token (*src0*) where the `register_type` field is set to IL_REGTYPE_LITERAL.		
3	x-bits, 32-bit untyped literal.		
4	y-bits, 32-bit untyped literal.		
5	z-bits, 32-bit untyped literal.		
6	w-bits, 32-bit untyped literal.		

Related None.

Declare the LDS Sharing Mode

Instructions **`DCL_LDS_SHARING_MODE`**

Syntax `dcl_lds_sharing_mode _wavefrontRel` or `_wavefrontAbs`

Description Local data share (LDS) memory has two sharing mode: wavefront relative or absolute. Relative means each wavefront has its private LDS memory. Absolute means all wavefronts share the same piece of LDS memory. Only used in a compute kernel.

Valid only for R7XX GPUs.

Format 0-input, 0-output, 0 additional token.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ DCL_LDS_SHARING_MODE
`control`	29:16	Mode: 0 _wavefrontRel 1 _wavefrontAbs
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Examples

```
dcl_lds_sharing_mode _wavefrontRel
dcl_lds_sharing_mode _wavefrontAbs
```

Related None.

Declare LDS Size Used in a Shader

Instructions **`DCL_LDS_SIZE_PER_THREAD`**

Syntax `dcl_lds_size_per_thread n`

Description Declares the space or size of LDS memory (in Dwords) to be used in a compute shader. The value must be:

- in Dwords
- no greater than 64, and
- a factor of 4.

Only valid for R7XX GPUs.

Format 0-input, 0-output, no additional token.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_LDS_SIZE_PER_THREAD
`control`	29:16	`n` = size in Dwords.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Example `dcl_lds_size_per_thread 8`

Related INIT_SHARED_REGISTERS.

Declare a Literal

Instructions	**`DCL_LITERAL`**
Syntax	`dcl_literal` *`src0`*`, <x-bits>, <y-bits>, <z-bits>, <w-bits>`

Description

DCL_LITERAL declares the literal to be used in the following instruction. The instruction is followed by four words containing the actual bits of the literal, in order x, y, z, w.

The lexically nearest preceding value is used. The 32-bit component literals (x-bits, y-bits, z-bits, and w-bits) are untyped, so that integer and float literals can be initialized with this instruction. *`src0`* must be a IL_REGTYPE_LITERAL or IL_REGTYPE_MLITERAL register type (see Section 5.23, “LITERAL,” page 5-15). No modifier bits are allowed. A given literal can be defined only once in a shader. This instruction cannot be placed in an unreachable code block such as after an unconditional break or return instruction. No modifiers are allowed.

Valid for R600 GPUs and later.

Format 1-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_LITERAL
	`control`	31:16	Must be zero.
2	IL_Src token (*src0*) where the `register_type` field is set to IL_REGTYPE_LITERAL.		
3	x-bits, 32-bit untyped literal.		
4	y-bits, 32-bit untyped literal.		
5	z-bits, 32-bit untyped literal.		
6	w-bits, 32-bit untyped literal.		

Related None.

Declare the Maximum Number of Vertices

Instructions	**`DCL_MAX_OUTPUT_VERTEX_COUNT`**
Syntax	`dcl_max_output_vertex_count` *n*

Description

DCL_MAX_OUTPUT_VERTEX_COUNT declares the maximum number of vertices that a single invocation of a geometry shader can emit. A geometry shader can emit a maximum of 1024 32-bit values. Thus, n can be no larger than floor (1024/number of 32-bit values per vertex). For example, if a geometry shader emits one 4-component vector (four 32-bit values) per vertex then n must be 256 or less.

Some implementations may be able to make optimizations by knowing the maximum number of vertices a single geometry shader invocation emits (for a single input primitive). The upper bound on the number of vertices that a geometry shader can produce depends on how large each vertex is. The sum of the number of components in each declared geometry shader output register defines how many 32-bit values are present in a single vertex. For example, if a geometry shader declares that it outputs a single four-component position, plus a three-component color per vertex, then the maximum number of vertices that can be declared for output by a single invocation is floor(1024 / 7). Or, if a Geometry Shader declares that it outputs 32 four-component vectors, the maximum number of vertices that can be declared for output by a single invocation is floor(1024 / 128).

DCL_MAX_OUTPUT_VERTEX_COUNT sets an upper limit on the number of vertices that can be emitted and, a geometry shader invocation can terminate after emitting fewer vertices than the maximum number allowed. An invocation terminates when DCL_MAX_OUTPUT_VERTEX_COUNT is reached. There is no requirement on the minimum number of vertices a geometry shader invocation must emit. The amount of vertices generated by a geometry shader invocation is simply the total number of `emit*` instructions executed in an invocation.

Valid for R600 GPUs and later.

Format

0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_MAX_OUTPUT_VERTEX_COUNT
`control`	29:16	*n*, maximum number of vertices.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Declare Upper Limit of Tessellation Factor from Hull Shader

Instructions	**`DCL_MAX_TESSFACTOR`**
Syntax	`dcl_max_tessfactor max_tessfactor`
Description	Declares the maximum tessellation factor for a hull shader. Must appear in a hull shader. It is valid only in a hull shader. The additional token, `max_tessfactor`, is a float value between 1.0 and 64.0. Valid for Evergreen GPUs and later.
Format	0-input, 0-output, one additional token format.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_MAX_TESSFACTOR
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare Maximum Number of Work-Items Per Work-Group

Instructions	**`DCL_MAX_THREAD_PER_GROUP`**
Syntax	`dcl_max_thread_per_group n`
Description	Declares the maximum number of work-items per work-group. This can be used when the program does not have a known work-group dimension at compile time. Therefore, this dcl and DCL_NUM_THREAD_PER_GROUP cannot present in the same program. This instruction can only be used in a compute shader. Valid for Evergreen GPUs and later.
Format	0-input, 0-output, one additional token format.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_MAX_THREAD_PER_GROUP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Example

```
dcl_max_thread_per_group 256
```

Related DCL_NUM_THREAD_PER_GROUP, DCL_NUM_ICP, DCL_NUM_INSTANCES, DCL_NUM_OCP.

Statically Declare Number of Input Control Points per Patch in Hull Shader

Instructions **`DCL_NUM_ICP`**

Syntax `dcl_num_icp n`

Description Statically declares the number of input control points per patch. Only used in a hull shader.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_ DCL_NUM_ICP
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n		

Example `Dcl_num_icp 7`

Related DCL_NUM_THREAD_PER_GROUP, DCL_MAX_THREAD_PER_GROUP, DCL_NUM_INSTANCES, DCL_NUM_OCP.

Statically Declare Number of Instances

Instructions **`DCL_NUM_INSTANCES`**

Syntax `dcl_num_instance n`

Description Statically declares the number of instances. Only used in a geometry shader.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_ DCL_NUM_INSTANCE
	`control`	29:16	Number of instances.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example `Dcl_num_instances 7`

Related DCL_NUM_THREAD_PER_GROUP, DCL_NUM_ICP, DCL_MAX_THREAD_PER_GROUP, DCL_NUM_OCP.

Statically Declare Number of Output Control Points per Patch in Hull Shader

Instructions **DCL_NUM_OCP**

Syntax `dcl_num_ocp n`

Description Statically declares the number of output control points per patch. Only used in a hull shader.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_NUM_OCP
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example `Dcl_num_ocp 7`

Related DCL_NUM_THREAD_PER_GROUP, DCL_NUM_ICP, DCL_NUM_INSTANCES, DCL_MAX_THREAD_PER_GROUP.

Declare the Work-Group Size

Instructions **DCL_NUM_THREAD_PER_GROUP**

Syntax `dcl_num_thread_per_group n1, n2, n3`

Description Specifies the umber of work-items per work-group. The sizes in three dimensions are n1, n2, and n3. For the HD4000-family of devices, the product of the three sizes can be at most 1024. Only used in a compute kernel. The value of n1 must be specified; those of n2 and n3 are optional. If n2 and n3 are omitted, a default value of 1 is assigned. Used only in a compute shader.

Valid for R7XX GPUs and later.

Format 0-input, 0-output, 1 to 3 additional tokens: up to three unsigned integers representing the literal value for n (n1, n2, n3).

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_NUM_THREAD_PER_GROUP
`control`	29:16	Number of dimensions (1 to 3).
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Examples `dcl_num_thread_per_group 5, 10, 1`

`dcl_num_thread_per_group 5` (which is the same as `dcl_num_thread_per_group` 5, 1, 1)

Related DCL_MAX_THREAD_PER_GROUP, DCL_NUM_ICP, DCL_NUM_INSTANCES, DCL_NUM_OCP.

Declare that the Pixel Shader intends to write to its scalar output oDepth register

Instructions **`DCL_ODEPTH`**

Syntax `dcl_odepth`

Description Declare that the pixel shader intends to write to its scalar output oDepth register. DX10 has some rules for what happens if oDepth is declared, but the shader does not write it.

Valid for R600 GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_ODEPTH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare an Output Register

Instructions **DCL_OUTPUT**

Syntax `dcl_output[_usage(type)] dst`

Description Declares the usage of a shader output register. *dst* must be of type IL_REGTYPE_OUTPUT and can have a modifier that specifies a component mask (see Section 3.5, "Write Mask," page 3-3). The component mask can be any subset of [xyzw]. DCL_OUTPUT can declare a superset of the component mask declared for input by the next shader stage, indicating the current shader writes more registers than the next shader stage reads; however, mutually exclusive masks are not allowed. For example, a vertex shader that writes o3.xy means the pixel shader reading only v3.z is invalid, but reading v3.x or v3.y or v3.xy would be valid. An output variable can be declared more than once, so that different components can be given different output types. Also, it is possible to have multiple clip distances. DX10 specifies that any component which is of type "none" must appear in xyzw order after components with non-none types. The component mask must be appropriate to the particular type. For example, the current set of system-generated values are all scalars, so the mask must have only one component.

A system-generated value cannot be output from a stage that is before the place in the pipeline where the hardware normally generates the value. For example, a geometry shader cannot output `IsFrontFace`, and VS cannot output `PrimitiveID`.

Valid for R600 GPUs and later.

Output type is:

Type	IL Enumeration (No EXPORTUSAGE in IL Headers)
Generic	IL_IMPORTUSAGE_GENERIC
Position	IL_IMPORTUSAGE_POS
ClipDistance	IL_IMPORTUSAGE_CLIPDISTANCE
CullDistance	IL_IMPORTUSAGE_CULLDISTANCE
PrimitiveID	IL_IMPORTUSAGE_PRIMITIVEID
VertexID	IL_IMPORTUSAGE_VERTEXID
InstanceID	IL_IMPORTUSAGE_INSTANCEID
RenderTargetArrayIndex	IL_IMPORTUSAGE_RENDERTARGET_ARRAY_INDEX
ViewportArrayIndex	IL_IMPORTUSAGE_VIEWPORT_ARRAY_INDEX

Format 0-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_OUTPUT
`control`	29:16	Any value of the enumerated type ILImportUsage. See Table 6.10 on page 6-5.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related DCL_OUTPUT_TOPOLOGY.

Declare Primitive Topology of Geometry Shader Output

Instructions	**`DCL_OUTPUT_TOPOLOGY`**
Syntax	`dcl_output_topology` *n*
Description	The geometry shader can only emit a single primitive topology from a given shader; the choices are: pointlist, linestrip or trianglestrip. Geometry shaders must contain this declaration. Note that for strip topologies, a single invocation of the geometry shader can emit multiple strips by using the cut instruction. This instruction is required in a geometry shader. Valid for R600 GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_DCL_OUTPUT_TOPOLOGY
`control`	29:16	Any value of the enumerated type ILTopologyType. See Table 6.29 on page 6-21.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related DCL_OUTPUT.

Declare Amount of Persistent Store Used by Shader

Instructions	**`DCL_PERSISTENT`**
Syntax	`Dcl_persistent src0`
Description	Declares the amount of persistent space used in the shader. `scr0` must be of type IL_REGTYPE_PERSIST with subscript indicating one more than the maximum allowed index. `dcl_peristent 4` allocates four slots: 0, 1, 2, and 3. Valid for R670 GPUs only.
Format	0-input, 0-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_DCL_PERSISTENT
`control`	31:16	Must be zero.

Related None.

Declare an Input Buffer

Instructions **DCL_RESOURCE**

Syntax `dcl_resource_id(`*`n`*`)_type(`*`pixtexusage`*`[,unnorm])_fmtx(`*`fmt`*`)_fmty(`*`fmt`*`)_fmtz(`*`fmt`*`)_fmtw(`*`fmt`*`)`

Description Declares an input buffer, specifies its dimension, and assigns it to a resource number. It identifies the resource type as a Buffer, Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D or TextureCube. Resources of type buffer can be used in an ld instruction. Resources of type texture * can as used in both ld and sample* instructions.

Use the ILPixTexUsage enumeration for this field. The instruction can also indicate if a texture resource has been normalized. Return types identify the data type fetched from the input buffer on a per-component basis.

Declares an input buffer, assigns it a number, and specifies its dimension.

Valid for R600 and later; valid for shader model 4 (SM4) and later.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_RESOURCE
	`id`	23:16	Resource id number, 0 to 255.
	`type`	27:24	Must be set to any value of the enumerated type ILPixTexUsage (see Section 6.22, "ILPixTexUsage," page 6-18. Possible types: Buffer, Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture2DMSarray, Texture3D, or TextureCube. Use the ILPixTexUsage enumeration for this field. Resources of type buffer can be used in an ld instruction. Resources of type texture * can be used in both ld and sample* instructions.
	`reserved`	30:24	Must be zero.
	`unnormalize`	31	0: Texture is normalized. 1: Texture is not normalized. The optional `_unnorm` option in the textural source can be used to specify this bit. The _unnorm option in the textural source can be used to specify this bit.
2	The return type identifies the data type fetched from the input buffer. Return types are four groups, each of three bits that must be set to any value of the enumerated type ILElementFormat. See Section 6.7, "ILElementFormat," page 6-3. Return-types are specified on a per-component basis, DX10 specification has no need to repeat identical return types; however, IL requires the type to be repeated all four times.		
	reserved	19:0	Must be zero.
	`fmtx`	22:20	x-component format.
	`fmty`	25:23	y-component format.
	`fmtz`	28:26	z-component format.
	`fmtw`	31:29	w-component format.

Example `dcl_resource_id(1)_type(1d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)`

Related None.

Declare Shared Registers

Instructions	**`DCL_SHARED_TEMP`**
Syntax	`dcl_shared_temp src#`
Description	Declares the number of shared GPRs used by this kernel (shared for each SIMD). *src0* must be of type IL_REGTYPE_SHARED_TEMP, with the number (#) indicating one more than the maximum used index. For example, `dcl_shared_temp sr4` indicates that src0, src1, src2, and src3 are used in the shader. The number declared must be smaller than the maximum available in hardware (for example: 32 for the HD4XXX-family of devices). Operations on shared registers are guaranteed atomic only when the read and write occur in the same instruction. Valid for R700 GPUs and later.
Format	1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_SHARED_TEMP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare Stream Number

Instructions	**`DCL_STREAM`**
Syntax	`dcl_stream` *n*

Description

Declares the geometry shader stream number. Valid only for geometry shaders. This is used with `DCL_OUTPUT` to specify the output registers for a given stream ID. Each `DCL_STREAM` must appear before all output declarations for the stream.

Valid for Evergreen GPUs and later.

Example:

```
dcl_stream 0
dcl_output o[0].xyzw
dcl_stream 1
dcl_output o[0].xyzw
dcl_output o[1].xyzw
dcl_stream 2
dcl_output o[0].xyz
dcl_output o[1].xyz
dcl_output o[2].xyz
dcl_stream 3
dcl_output o[0].xy
dcl_output o[0].zw
dcl_output o[1].xy
dcl_output o[1].zw
```

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_STREAM
`control`	29:16	Unsigned integer, representing the stream ID.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare Total Number of Work-Groups

Instructions	**`DCL_TOTAL_NUM_THREAD_GROUP`**
Syntax	`dcl_total_num_thread_group n1, n2, n3`
Description	Declares the total number of work-groups in a dispatch call. The numbers are in three dimensions: n1, n2, and n3. This is used only when the numbers are statically defined in the shader (n1 must be specified, while n2 and n3 are optional). If n2 and/or n3 are omitted, a default value of 1 is assigned. This is used only in a compute shader. Valid for R7XX GPUs and later.
Format	0-input, 0-output, 1 to 3 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_TOTAL_NUM_THREAD_GROUP
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional tokens: up to three unsigned integers representing the literal value for n (n1, n2, n3).		

Example

- `Dcl_total_num_thread_group 5, 2, 1`
- `Dcl_total_num_thread_group 5` (which is the same as `Dcl_total_num_thread_group 5, 1, 1`)

Related None.

Declare Type of Tessellation Domain for Tessellator

Instructions	**`DCL_TS_DOMAIN`**
Syntax	`dcl_ts_domain ts_domain_[isoline|tri|quad]`
Description	Declares the type of tessellation domain for tessellation. Must appear in a hull shader. Is valid only in a hull shader. `Ts_domain_type` must be an element of the ILTsDomain enumeration. Valid for Evergreen GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_TS_DOMAIN
`control`	31:16	Must be zero.

Related DCL_TS_OUTPUT_PRIMITIVE, DCL_TS_PARTITION.

Declare Type of Tessellation Output Primitive

Instructions	**`DCL_TS_OUTPUT_PRIMITIVE`**
Syntax	`dcl_ts_output_primitive ts_output_primitive_type`
Description	Declares the type of output primitive by tessellator. Must appear in a hull shader. It is valid only in a hull shader. `Ts_output_primitive_type` must be an element of the ILTsOutputPrimitive enumeration. Valid for RXX GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_TS_OUTPUT_PRIMITVE
`control`	31:16	Must be zero.

Related DCL_TS_DOMAIN, DCL_TS_PARTITION.

Declare Type of Tessellator Partitioning

Instructions	**`DCL_TS_PARTITION`**
Syntax	`dcl_ts_partition ts_partition_type`
Description	Declares the type of tessellation used by the tessellator. Must appear in a hull shader. It is valid only in a hull shader. `Ts_partition_type` must be an element of the ILTsPartition enumeration. Valid for Evergreen GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_DCL_TS_PARTITION
`control`	31:16	Must be zero.

Related DCL_TS_DOMAIN, DCL_TS_OUTPUT_PRIMITIVE.

Declare a Primitive ID

Instructions **`DCL_VPRIM`**

Syntax `dcl_vprim`

Description Declares that the geometry shader intends to use its scalar input register vPrim. For the geometry shader, input primitive data only comes in the form of a scalar (vPrim, no mask). Also, there is no Primitive Data for adjacent primitives available in a geometry shader invocation.

Valid for R600 GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCL_VPRIM
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare Registers as Part of an Array

Instructions **`DCLARRAY`**

Syntax `dclarray` *`src0, src1`*

Description Declares a range of registers as part of an array indexed access (to be accessed through base/loop relative addressing). Only the INTERP and TEXCOORD registers can be used. *src0* and *src1* must be of the same register type. Base and loop relative addressing cannot be used on TEMP, INTERP, and TEXCOORD registers not declared within the range of *src1* and *src2*. If registers outside the range of registers in the array declared with this instruction are accessed, the compiler cannot guarantee the stability of the shader.

Base and loop relative addressing perform indexing based on the register number, not on the register's position in the array. (Declaring an array of registers 2 to 6, and indexing when the loop counter is 3, accesses register number 3, not register number 5.)

There can be numerous DCLARRAYs for each type, so long as they do not overlap in range.

DCLARRAY cannot be used with relative addressing or a source modifier. Both the `modifier_present` and `relative_address` fields of the IL_Src tokens must be zero.

Valid for all GPUs.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DCLARRAY
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Declare Register Defaults

Instructions **DCLDEF**

Syntax `dcldef_x(`*`xdefault`*`)_y(`*`ydefault`*`)_z(`*`cozdefaultmp`*`)_w(`*`wdefault`*`)` *`dst`*

Description Declares the default value for register *`dst`* and cannot be used more than once per register type in a shader. *dst* must be a TEMP or ADDR register type (see Section 6.23, on page 6-18). Each component in the register is set individually to an ILDefaultVal enumerated type. See Table 6.5 on page 6-2 for ILDefaultVal values. The xdefault, ydefault, zdefault, and wdefault describe the default value for each component of the register.

DCLDEF cannot be used with relative addressing or a destination modifier. Both the `modifier_present` and `relative_address` fields of the IL_Dst token must be zero.

Valid for all GPUs.

Format 0-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCLDEF
	`xdefault`	17:16	x-component default value.
	`ydefault`	19:18	y-component default value.
	`zdefault`	21:20	z-component default value.
	`wdefault`	23:22	w-component default value.
	reserved	31:24	Must be zero.

The component default types can be any value of the enumerated type ILDefaultVal. See Table 6.5 on page 6-2.

2 IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_TEMP or IL_REGTYPE_ADDR. The `modifer_present` and `relative_address` fields must be set to 0.

Example `dcldef_z(*)_w(*) dst` indicates that there is no default value for the z and w components of the dst register.

Related None.

Declare Interpolator Properties

Instructions **DCLPI**

Syntax `dclpi_x(`*`comp`*`)_y(`*`comp`*`)_z(`*`comp`*`)_w(`*`comp`*`)_center_bias_invert_centered]` *`dst`*

Description Declares properties of pixel shader named interpolator inputs and the window coordinate register.

A shader cannot use this instruction on the same register more than once. There can be at most one DCLPI per register.

`ximport`, `yimport`, `zimport`, and `wimport` each describe what components of the register are used by the pixel shader. See the enumerated type for more information.

In shaders where PINPUT registers are used, this instruction can only be used to declare a WINCOORD register.

You cannot use a destination modifier with this instruction. The `modifier_present` field of the IL_Dst token must be set to 0.

You cannot use relative addressing with this instruction. The `relative_address` field of the IL_Dst token must be set to 0.

To use both bias and centered, set just bias and adjust the bias values: when invert is not set, add 0.5 to both bias.x and bias.y; when invert is set, add 0.5 to bias.x and -0.5 to bias.y. This optimization reduces instructions generated to support these control fields.

A DCLPI token can be used to declare interpolated outputs in a vertex shader and interpolated inputs in a pixel shader.

Use DCLPI to support shader models prior to 4.0, which support only perspective gradients.

Do not use DCL_INPUT in shaders that use DCLPI.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCLPI
	`ximport`	17:16	Any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
	`yimport`	19:18	Any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
	`zimport`	21:20	Any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
	`wimport`	23:22	Any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
	`center`	24	For interpolator registers, this field specifies if center or centroid sampling is used. If the `register_type` field of the following IL_Dst token is IL_REGTYPE_WINCOORD, this bit determines whether to use the center (1), or the nearest sample (0), called centroid, within the pixel as the interpolation value when multisampling, since the center might be outside the polygon.

Declare Interpolator Properties (Cont.)

	`bias`	25	This field has no effect if the `register_type` field of the following IL_Dst token is not IL_REGTYPE_WINCOORD. For the window coordinate register, specifies if the compiler generates instructions to add a bias supplied in constants named SC_CONS_SRC_VIEWPORT_BIAS_{X,Y}. Without a bias, the origin is located at the upper-left corner of the screen (DX Style). A bias can be used to move the origin. For example, to move the origin to the lower-left corner of a window (OGL style): • DX must set this to 0 • OGL must set this to 1.
	`invert`	26	This field does nothing if the register_type field of the following IL_Dst token is not IL_REGTYPE_WINCOORD. If the register_type field of the following IL_DstToken is IL_REGTYPPE_WINCOORD and this bit is set, the compiler inverts (multiplies by -1) the y coordinate of the position. By default, the y axis goes down (DX style), but this bit can be used to change the direction to up (OGL style).
	`centered`	27	This field does nothing if the register_type field of the following IL_Dst token is not IL_REGTYPE_WINCOORD. If the register_type field of the following IL_Dst token is IL_REGTYPE_WINCOORD, this bit specifies the following: 0 defines the origin as the upper-left corner of the RenderTarget. Thus, pixel centers are offset by (0.5f,0.5f) from integer locations on the RenderTarget. This origin makes rendering screen-aligned textures trivial, as the pixel coordinate system is aligned with the texel coordinate system. This choice matches OGL and DX10. 1 defines the origin as the center of the upper-left pixel in the RenderTarget: the origin is (0.5,0.5) away from the upper-left corner of the RenderTarget. Thus, pixel centers are at integer locations. This choice of origin matches DX9. 0 center of the pixel is (0.5,0.5) 1 center of pixel is (0,0)
	reserved	31:28	Must be zero.
2	IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_INTERP, IL_REGTYPE_FOG, IL_REGTYPE_PRIMCOORD, IL_REGTYPE_TEXCOORD, IL_REGTYPE_PRICOLOR, or IL_REGTYPE_SECCOLOR, or IL_REGTYPE_WINCOORD. The `modifer_present` and `relative_address` fields must be set to 0.		

Related None.

Declare Pixel Shader Input Register

Instructions **DCLPIN**

Syntax `dclpin_usage(op)_usageIndex(n)_x(comp)_y(comp)_z(comp)_w_(comp)[_centroid] dst`

Description Declares a mapping of a vertex shader output to a pixel shader input. This instruction or a DCLPP instruction must be issued on each PINPUT register before the register is used in a shader. There can be at most one DCLPIN instruction per *usage-usageIndex* pair.

An *enabled* component is a component set to IL_IMPORTSEL_UNDEFINED, IL_IMPORTSEL_DEFAULT0, or IL_IMPORTSEL_DEFAULT1.

A *disabled* component is a component set to IL_IMPORTSEL_UNUSED.

Packed Registers: a PINPUT register can be declared multiple times with this instruction; thus, a single register can have multiple unique *usage-usageIndex* pairs. However, the same component of a register cannot be *enabled* in both declarations. Thus, if in one declaration a component is *enabled*, the component must be *disabled* in the other declaration(s). For example, if vIN3 is declared as having *usage(interp)_usageIndex(1)_x (*),* and vIN3 is also declared as having *usage(interp)_usageIndex(2),* then that declaration must set x*(-).*

If an IL_PrimaryDCLPIN_Mod token is not preset, the shader behaves as if ximport, yimport, zimport, and wimport are set to IL_IMPORTSEL_UNDEFINED and centroid is set to 0.

Only one register can be declared to have the usage IL_IMPORTUSAGE_FOG. In this case, usageIndex must be zero.

It is an error to use this instruction in a vertex shader or a real-time pixel shader.

Note that a shader using a PINPUT register cannot use the INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG registers.

A source modifier cannot be used with this instruction. The `modifier_present` field of the IL_Dst token must be set to 0.

You cannot use relative addressing with this instruction. The `relative_address` field of the IL_Dst token must be set to 0.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCLPIN The lower five bits of the control field must be set to an acceptable value of the enumerated type ILImportUsage(usage). The next eight bits of the control field must be a unique number for usage that specifies a mapping of usage type to a vertex shader output (`usageIndex`).
	`usage`	20:16	Any value of the enumerated type ILImportUsage. See Table 6.10 on page 6-5.
	`usageIndex`	28:21	Unique number for *usage* which specifies a mapping of *usage* type to a vertex shader output (*usageIndex*), 0 to 255.
	reserved	30:29	Must be zero.
	`pri_instruction_modifier`	31	If `pri_modifier_present` is set to 1, an IL_PrimaryDCLPIN_Mod token immediately follows this token.

Declare Pixel Shader Input Register (Cont.)

2 Primary pixel shader input register declaration modifier. IL_PrimaryDCLPIN_Mod token described below: Note that the IL_PrimaryDCLPIN_Mod is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token.

Field Name	Bits	Description
`ximport`	1:0	Specifies if the x component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`yimport`	3:2	Specifies if the y component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`zimport`	5:4	Specifies if the z component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`wimport`	7:6	Specifies if the w component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`centroid`	8	Specifies if this value represents a value at the pixel centroid or the center 0 Value at the center. 1 Value at the centroid.
`constant`	9	Constant interpolation.
`no_perspective`	10	Do not perform perspective divide during interpolation.
reserved	31:11	Must be zero.

3 IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_PINPUT. The `modifier_present` and `relative_address` fields must be set to 0.

Related None.

Map Interpolator Register to Realtime Interpolator Parameter

Instructions **DCLPP**

Syntax `dclpp_param(`*`n`*`)` *`dst`*

Description Maps a PINPUT register to a realtime interpolator parameter. This instruction can be used only in a real-time pixel shader. It is an error to use this instruction in a normal pixel shader or vertex shader. This instruction must be issued on each PINPUT register before the register is used in a real-time pixel shader. There can be at most one DCLPP per register.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCLPP
	`param`	23:16	Specifies the realtime state register to which the register is mapped (`param`).
	reserved	31:24	Must be zero.
2	IL_Dst toke (*dst*), where *register_type* is set to IL_REGTYPE_PINPUT. The *modifier_present* and *relative_address* fields must be set to 0.		

Related None.

Declare Texture Properties

Instructions	**`DCLPT`**
Syntax	`dclpt_stage(`*`n`*`)_type(`*`op`*`)_coordmode(`*`mode`*`)`

Description

Declares properties of a texture stage. This instruction is used for shader model 3; it cannot be used for DX10.

The texture stage must be declared before a TEXLD, TEXLDD, TEXLDB, TEXLDMS, TEXWEIGHT, PROJECT, or LOD instruction is issued on the stage.

If *type* is IL_USAGE_PIXTEX_UNKNOWN, this instruction indicates that the texture type of the texture on the stage/unit indicated by *stage* is not known at shader-create time

If `coordmode` is set to IL_TEXCOORDMODE_NORMALIZED, then the texture coordinates are normalized (scaled from 0 to 1.0).

If `coordmode` is set to IL_TEXCOORDMODE_UNNORMALIZED, the texture coordinate is not normalized. In this case, the x texture coordinate ranges from 0.0 to the width of the texture, the y texture coordinate ranges from 0.0 to the height of the texture, and the z texture coordinate ranges from 0.0 to the depth of the texture.

If `coordmode` is set to IL_TEXCOORDMODE_UNKNOWN, the value of AS_TEX_DENORM_N (*stage*) determines if the texture coordinate used in any subsequent TEXLD, TEXLDD, TEXLDB, TEXLDMS, TEXWEIGHT, PROJECT, or LOD instructions are normalized at shader run time.

If cleartype mode is set, the compiler multiplies the result of any fetch on this texture by 1/(kernel height * width). This is used for DX9 only.

There must be one DCLPT per stage.

Valid for all GPUs.

Format

Field Name	Bits	Description
`code`	15:0	IL_OP_DCLPT
`controls_stage`	23:16	Stage or unit number.
`controls_type`	26:24	Any value of the enumerated type ILPixTexUsage. See Table 6.22 on page 6-18.
`controls_coordmode`	28:27	Any value of the enumerated type ILTexCoordMode. See Table 6.26 on page 6-20.
`cleartype_mode`	29	1 = yes, 0 = no.
reserved	31:30	Must be zero.

Related None.

Declare Mapping for Vertex Shader Inputs

Instructions **DCLV**

Syntax dclv_elem(*n*)_x(*comp*)_y(*comp*)_z(*comp*)_w(*comp*) *dst*

Description Declares a mapping between vertex buffer elements (logical streams) set through state and vertex shader inputs. It is an error to use this instruction in a pixel shader.

The INITV and DCLV instruction are mutually exclusive for a given VERTEX register. There can be at most one DCLV per register. Use only registers of type TEMP and VERTEX. Each VERTEX register must either be declared at least once with a DCLV, or initialized with an INITV before it is used as a source in any instruction.

The value of `elem` corresponds to vertex buffer element *n.*

An *enabled* component is a component set to IL_IMPORTSEL_UNDEFINED, IL_IMPORTSEL_DEFAULT0, or IL_IMPORTSEL_DEFAULT1.

A *disabled* component is a component set to IL_IMPORTSEL_UNUSED.

Packed Registers: A VERTEX register can be initialized multiple times with this instruction; thus, a single register can be mapped to multiple unique vertex buffer elements. However, the same component of a register cannot be used in both declarations: if in one declaration a component is *enabled*, the component must be *disabled* in the other declaration(s) of the register.

The i^{th}-enabled component receives the i^{th} piece of data of the vertex buffer element specified by `elem` if it exists. If the data does not exist (the dimension of the vertex buffer element specified in state is less than i), the i^{th} component receives the default value specified in this instruction.

Do not use a destination modifier with this instruction. The `modifier_present` field of the IL_Dst token must be set to 0.

Do not use relative addressing with this instruction. The `relative_address` field of the IL_Dst token must be set to 0.

Operation:

```
VECTOR v;
for (i=0; i < 4; i++)
{
    if(i < ElementDimension(elem)) # AS_VS_DECL_TYPE_DIMENSION_N(elem)
    v[i]=FetchData(elem,currentIndex,i);
else
{
    v[i]=Default(i);
}}
WriteResult(v, dst);
```

Declare Mapping for Vertex Shader Inputs (Cont.)

Format	Token	Field Name	Bits	Description
	1	`code`	15:0	IL_OP_DCLV
		`elem`	21:16	Vertex buffer element, 0 to 63. These bits specify the vertex buffer element from which vertex data is loaded/fetched (`elem`).
		`reserved`	30:22	Must be zero.
		`pri_modifier_present`	31	If `pri_modifier_present` is set to 1, an IL_PrimaryDCLV_Mod token immediately follows this token.
	2	Primary vertex shader input register declaration modifier. IL_PrimaryDCLV_Mod token described below. The IL_PrimaryDCLV_Mod is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token.		
		`ximport`	1:0	Specifies if the x component is enabled for the vertex buffer element specified by `elem` for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
		`yimport`	3:2	Specifies if the y component is enabled for the vertex buffer element specified by `elem` for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
		`zimport`	5:4	Specifies if the z component is enabled for the vertex buffer element specified by `elem` for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
		`wimport`	7:6	Specifies if the w component is enabled for the vertex buffer element specified by `elem` for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
		reserved	31:8	Must be zero.
	3	IL_Dst token with `register_type` set to IL_REGTYPE_TEMP or IL_REGTYPE_VERTEX (*dst*). The `modifier_present` and `relative_address` field must be set to 0.		

Related None.

Declare Vertex Shader Output Register

Instructions **DCLVOUT**

Syntax `dclvout_usage(`*op*`)_usageIndex(`*n*`)_x(`*comp*`)_y(`*comp*`)_z(`*comp*`)_w(`*comp*`)` *dst*

Description Declares the usage of a vertex shader output register, as well as a mapping of a vertex shader output to a pixel shader input.

This instruction must be issued on each VOUTPUT register before the register is used in a shader. A shader using a VOUTPUT register cannot use the POS, SPRITE, INTERP, TEXCOORD, PRICOLOR, SECCOLOR, or FOG registers.

There can be at most one DCLOUT instruction per `usage-usageIndex` pair.

An *enabled* component is a component set to IL_IMPORTSEL_UNDEFINED, IL_IMPORTSEL_DEFAULT0, or IL_IMPORTSEL_DEFAULT1.

A *disabled* component is a component set to IL_IMPORTSEL_UNUSED.

Packed Registers: A VOUTPUT register can be declared multiple times with this instruction; thus, a single register can have multiple unique `usage-usageIndex` pairs. However, the same component of a register cannot be enabled in both declarations: if in one declaration a component is enabled, the component must be disabled in the other declaration(s). For example, if oOUT3 is declared as having `usage(interp)_usageIndex(1)_x (*)`, and oOUT3 is also declared as having `usage(interp)_usageIndex(2)`, then the second declaration must set x(-).

If an IL_PrimaryDCLVOUT_Mod token is not preset, the shader behaves as if `xexport`, `yexport`, `zexport`, and `wexport` are set to IL_IMPORTSEL_UNDEFINED.

Only one register can be declared to have the `usage` IL_IMPORTUSAGE_POS. If loop relative addressing is used on vertex shader outputs, that register can only be the VOUTPUT register number 0.

`usageIndex` must be if IL_IMPORTUSAGE_POINTSIZE is used.

Only one register can be declared to have the `usage` IL_IMPORTUSAGE_POINTSIZE. If loop relative addressing is used on vertex shader outputs, that register can only be the VOUTPUT register number 1. `usageIndex` must be zero if IL_IMPORTUSAGE_POINTSIZE is used.

Only one register can be declared to have the `usage` IL_IMPORTUSAGE_FOG. `usageIndex` must be zero in this case if IL_IMPORTUSAGE_FOG is used.

It is an error to use this instruction in a pixel shader.

You cannot use a source modifier with this instruction. The `modifier_present` field of the IL_Dst token must be set to 0.

You cannot use relative addressing with this instruction. The `relative_address` field of the IL_Dst token must be set to 0.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCLVOUT
	`usage`	20:16	Any value of the enumerated type ILImportUsage. See Table 6.10 on page 6-5. The first five bits of the must be set to an acceptable value of the enumerated type ILImportUsage (`usage`). The next eight bits must be a unique number for usage which specifies a mapping of usage type to a pixel shader input (`usageIndex`).
	`usageIndex`	28:21	Unique number for *usage* which specifies a mapping of *usage* type to a pixel shader input (*usageIndex*), 0 to 255.
	reserved	30:29	Must be zero.

Declare Vertex Shader Output Register (Cont.)

`pri_modifier _present`	31	If `pri_modifier_present` is set to 1, an IL_PrimaryDCLOUT_Mod token immediately follows this token.

Primary vertex shader output register declaration modifier. IL_PrimaryDCLVOUT_Mod token[1] described below.

Field Name	Bits	Description
`xexport`	1:0	Specifies if the x component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`yexport`	3:2	Specifies if the y component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`zexport`	5:4	Specifies if the z component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
`wexport`	7:6	Specifies if the w component is enabled for the usage-usageIndex for this declaration. Also specifies the default value if the component is enabled. Can be any value of the enumerated type ILImportComponent. See Table 6.9 on page 6-4.
reserved	31:8	Must be zero.

3 IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_VOUTPUT. The `modifier_present` and `relative_address` fields must be set to 0.

Related None.

1. The IL_PrimaryDCLOUT_Mod is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token.

Constant Integer or Float Register Definition

Instructions **DEF**

Syntax `def dst, <number>, <number>, <number>, <number>`

Description DEF declares the constant value for register *dst* and cannot be used more than once on the same register. It indicates that a floating point or integer constant contains a given value when the shader is executed. This instruction is followed by four words that contain the bit values of the constant. *dst* must be a CONST_INT or CONST_FLOAT register type (See Chapter 5, "Register Types"). Each component of a register can be set individually. CONST_INT registers do not use the w-component.

Clients must ensure that the constant values set through state match the values indicated in this instruction.

DEF cannot be used with relative addressing or a destination modifier. Both the `modifier_present` and `relative_address` fields of the IL_Dst token must be zero.

Valid for all GPUs.

Operation:

```
VECTOR v;
v[0] = x;
v[1] = y;
v[2] = z;
v[3] = w;
WriteResult(v, dst);
```

Format 0-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DEF
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_CONST_INT or IL_REGTYPE_CONST_FLOAT. The `modifer_present` and `relative_address` fields must be set to zero.		
3	x-component, 32-bit integer or float		
4	y-component, 32-bit integer or float		
5	z-component, 32-bit integer or float		
6	w-component, 32-bit float only (not used for a CONST_INT register)		

Related DEFB.

Constant Boolean Register Definition

Instructions **DEFB**

Syntax `defb dst, <unsigned integer>`

Description Indicates that a boolean constant contains a given value when the shader is executed. Clients must ensure that the constant values set through state match the values indicated in this instruction. A shader cannot use this instruction on the same register more than once.

If the value of the given unsigned integer is 0, the boolean is set to false. If the value the given unsigned integer is non-zero, set the boolean to true.

DEFB cannot be used with relative addressing or a destination modifier. Both the modifier_present and relative_address fields of the IL_Dst token must be zero.

Valid for R600 GPUs and later.

Operation:

```
CONST_BOOL b;
b = TRUE;
if(val == 0) {
    b=FALSE;
}
WriteResult(b, dst);
```

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DEFB
	`control`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Dst token (*dst*) where the `register_type` field is set to IL_REGTYPE_CONST_BOOL. The `modifer_present` and `relative_address` fields must be set to zero.		
3	Boolean value, 32-bit unsigned integer • 0: FALSE • not 0: TRUE		

Related DEF.

Initialize Shared Registers (SR) Used by Sync Barriers

Instructions	**`INIT_SHARED_REGISTERS`**
Syntax	`init_shared_registers`

Description

For HD4000 series devices, a pre-shader is needed to initialize shared registers (SR) used to implement synchronization barriers.

This special instruction must be the only instruction in a compute shader. It takes inputs from `cb0[0].xyz` and initializes the SR as required.

The implementation assumes the following input is available:

`cb0[0].x` – starting SR index (inclusive), an integer value.

`cb0[0].y` – ending SR index (exclusive), an integer value.

`cb0[0].z` – values to be written to SR (a float value representing the number of wavefronts in a work-group).

- For all i in the range of [cb0[0].x, cb0[0].y), this instruction will initialize the SR[i] as

`SR[i]. x` = 0xFFFFFFFF

`SR[i]. y` = cb0[0].z

`SR[i]. z` = 0xFFFFFFFF

`SR[i]. w` = 0xFFFFFFFF

For example, if `cb0[0].x` = 3 and `cb0[0].y` = 9, then sr3 to sr8 are initialized as shown above.

This is used only in a compute shader.

Valid for R7XX GPUs and later.

Example

- il_cs_2
- `init_shared_registers`
- `end`

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_INIT_SR
`control`	31:16	Must be zero.

Related None.

Initialize Vertex

Instructions	**`INITV`**
Syntax	`initv dst, src`
Description	Initializes a vertex shader input to the value of `src`. This instruction provides another way to initialize the vertex shader input registers. See `DCLV` (page 60) for normal operation. This instruction typically is used for higher-order surface shaders. Do not use this instruction in a pixel shader. The `INITV` and `DCLV` instruction are mutually exclusive for a given `VERTEX` register. There can be at most one `INITV` per `VERTEX` register. The `INITV` instruction for a given `VERTEX` register must occur before the given register is used as a source register. This instruction cannot be used within a flow-control-block. Thus, loop relative addressing cannot be used with this instruction. `VERTEX` registers cannot be used as a source until a `DCLV` or `INITV` instruction is used on it. Default values for `VERTEX` registers must be explicitly done in the shader prior to this instruction. The register used to initialize the `VERTEX` register must contain any default values required by the client. Valid for all GPUs.
Format	0-input, o-output, 1 additional token.

Opcode

Token	Description
1	IL_Opcode token with code set to IL_OP_INITV.
2	IL_Dst token (dst) where register_type is set to IL_REGTYPE_VERTEX. The relative_address field must be set to 0.
3	IL_Dst_Mod token[1].
4	IL_Src token (src).
5	IL_Src_Mod token[2].
6	IL_Rel_Addr token[3] where loop_relative is set to 0.

Operation

```
VECTOR v = EvalSource(src);
WriteResult(v, dst);
```

Related None.

1. `IL_Dst_Mod` token only present if `modifier_present` field is 1 in previous `IL_Dst` token.
2. `IL_Src_Mod` token only present if `modifier_present` field is 1 in previous `IL_Src` token.
3. `IL_Rel_Addr` token only present if `relative_address` field is 1 in the preceding `IL_Src` or `IL_Dst` token.

7.6 Input/Output Instructions

Query Information from a (Non-Constant) Buffer

Instructions	**BUFINFO**

Syntax	`buifinfo_resource(n) [_uav] dst`
	`bufinfo_ext_resource(n) [_uav] dst, src0, src1`

Description	The `dst` receives the integer size in elements of the buffer; all four components of `dst` get the same value. Low-order eight bits contain the `resource id (n)`. Valid for Evergreen GPUs and later.
Format	0-input, 1-output. 2-input, 1-output.
Opcode	

Field Name	Control Field
`il_op_buf_info`	IL_OP_BUFINFO. Contains the resource id (n).
`il_op_buf_info_ext`	Same as `il_op_buf_info`.

Low-order eight bits contain the resource id (n).

Related None.

Finish Current Topology

Instructions	**CUT**
Syntax	`cut`
Description	Completes the current primitive topology (if any vertices have been emitted), and any previous primitive topology, then starts a new topology of the type declared by the geometry shader. No vertices are leftover since a geometry shader can only emit pointlist, linestrip and trianglestrip topologies. When CUT is executed, any previously emitted topology by the Geometry Shader invocation is completed. If not enough vertices were emitted for the previous primitive topology, the extra vertices are discarded. After the previous topology (if any) is completed, CUT causes a new topology to begin, using the topology declared as the geometry shader's output. CUT can be used only in a geometry shader.This instruction can appear any number of times in a geometry shader and is executed implicitly at the end of an invocation of the geometry shader. Valid for R600 GPUs and later.
Format	0-input, 0-output.
Opcode	

Field Name	Bits	Description
`code`	15:0	IL_OP_CUT
`control`	31:16	Must be zero.

Related CUT_STREAM.

Finish Current Topology in Geometry Shader

Instructions	**`CUT_STREAM`**
Syntax	`cut_stream`
Description	Finishes the current topology in the geometry shader. This is similar to `CUT`, except it operates only on the specified stream. Valid for Evergreen GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CUT_STREAM
`control`	31:16	Stream index.

Related CUT.

Discard Results Based on Integer

Instructions **`DISCARD_LOGICALNZ, DISCARD_LOGICALZ`**

Syntax

Opcode	Syntax	Description
`IL_OP_DISCARD_LOGICALNZ`	`discard_logicalnz` *`src0.select_component`*	Discard results if all bits in *src0*.{x|y|z|w} ≠ 0.
`IL_OP_DISCARD_LOGICALZ`	`discard_logicalz` *`src0.select_component`*	Discard results if all bits in *src0*.{x|y|z|w} == 0.

Description

Conditionally flags results of pixel shader to be discarded when the end of the program is reached. This instruction flags the current pixel as terminated, while continuing execution, so that other pixels executing in parallel can obtain gradients, if necessary. Although execution continues, all pixel shader output writes before or after the discard_* instruction are discarded.

The discard_* instruction can be present inside any flow control construct.

Multiple discard instructions can be present in a pixel shader, and if any is executed, the pixel is terminated.

Can be used only in a pixel shader.

Valid for R600 GPUs and later.

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See Opcode part of *Syntax*, above.
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related KILL.

Emit a Vertex

Instructions	**`EMIT`**
Syntax	`emit`
Description	Causes all declared o# registers to be read out of a geometry shader to generate a vertex. Multiple EMIT instructions are used to generate a primitive. Any number of EMIT instructions can appear in a geometry shader, including within flow control. This instruction can be used only in a geometry shader. Valid for R600 GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EMIT
`control`	31:16	Must be zero.

Related EMIT_STREAM, EMIT_THEN_CUT, EMIT_THEN_CUT_STREAM,

Emit a Vertex for a Given Stream

Instructions	**`EMIT_STREAM`**
Syntax	`emit_stream`
Description	Same as `EMIT`, except this operates on a given stream. Valid for Evergreen GPUs and later.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EMIT_STREAM
`control`	31:16	Stream index.

Related EMIT, EMIT_THEN_CUT, EMIT_THEN_CUT_STREAM

Emit Followed by Cut

Instructions **`EMIT_THEN_CUT`**

Syntax `emitthencut`

Description Operation is the same as an EMIT instruction immediately followed by a CUT instruction. It has the same restrictions as the union of restrictions for the EMIT and CUT commands.

Valid for R600 GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EMIT_THEN_CUT
`control`	31:16	Must be zero.

Related EMIT, EMIT_STREAM, EMIT_THEN_CUT_STREAM.

Emit Then Cut for a Given Stream

Instructions **`EMIT_THEN_CUT_STREAM`**

Syntax `emitcut_stream`

Description Same as `EMIMT_THEN_CUT`, except this operates on a given stream.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EMIT_THEN_CUT_STREAM
`control`	31:16	Stream index..

Related EMIT, EMIT_STREAM, EMIT_THEN_CUT.

Pull-Model Evaluation at Pixel Centroid

Instructions **`EVAL_CENTROID`**

Syntax `eval_centroid dst, src0`

Description `Src0` must be of type `IL_REGTYPE_INPUT`. After the instruction executes, `dst` contains `src0` interpolated with the centroid location.

No source modifiers are allowed.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EVAL_CENTROID
`control`	31:16	Must be zero.

Related EVAL_SAMPLE_INDEX, EVAL_SNAPPED.

Pull-Model Evaluation with Sample Index

Instructions **`EVAL_SAMPLE_INDEX`**

Syntax `eval_sample_index dst, src0, src1`

Description `Src0` must be of type `IL_REGTYPE_INPUT`. `Src1` must be of type `IL_REGTYPE_LITERAL`(`MLITERAL`). `src1.yzw` must be 0. `src1.x` contains an integer value.

After the instruction executes, `dst` contains `src0` interpolated at the sample location given in `src1.x`.

No source modifiers are allowed.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EVAL_SAMPLE_INDEX
`control`	31:16	Must be zero.

Related EVAL_CENTROID, EVAL_SNAPPED.

Pull-Model Evaluation with Pixel Offset

Instructions **`EVAL_SNAPPED`**

Syntax `eval_snapped dst, src0, src1`

Description `Src0` must be of type `IL_REGTYPE_INPUT`.

`Src1` must be `IL_REGTYPE_TEMP`; `src1` must be two-component, containing floating point x, y offsets from the pixel center.

After the instruction executes, `dst` contains `src0` interpolated at the pixel offset given in `src1.xy`.

No source modifiers are allowed.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EVAL_SNAPPED
`control`	31:16	Must be zero.

Example

DX11:
```
eval_snapped r0.xyzw, v[0].xyzw, l(0xf, 0x7, 0, 0); // -0.0625,
0.4375
```

IL
```
r1.xy =  l(-1.0, 7.0, 0, 0) / l(16.0, 16.0, 16.0, 16.0);
eval_snapped r0.xyzw, v[0].xyzw, r1.xy;
```

DX11:
```
eval_snapped r0.xyzw, v[0].xyzw, l(0xf, 0x7, 0, 0); // -0.0625,
0.4375
```

IL
```
r1.xy =  l(-1.0, 7.0, 0, 0)  / l(16.0, 16.0, 16.0, 16.0);
eval_snapped r0.xyzw, v[0].xyzw, r1.xy;
```

Related EVAL_CENTROID, EVAL_SAMPLE_INDEX

Fence for Synchronization Work-Items, and/or LDS and/or GDS and/or Global Memory or UAV

Instructions **FENCE**

Syntax `fence[_threads] [_lds] [gds] [_memory] [_sr]`

Description It must include at least one option. It is an error if none is specified. All selected options must complete before the work-item continues.

`_threads` - synchronize work-items in a work-group so that all work-items must reach this point before any work-item can go further. The instruction cannot be used inside of any control flow. It can be used only in a compute shader (except for bit 18; see below).

`_lds` - shared memory fence. It ensures that:

- no LDS read/write instructions can be re-ordered or moved across this fence instruction.
- all LDS write instructions are complete (the data has been written to LDS memory, not in internal buffers) and visible to other work-items.

`_memory` - global/scatter memory fence. It ensures that:

- no memory import/export instructions can be re-ordered or moved across this fence instruction.
- all memory export instructions are complete (the data has been written to physical memory, not in the cache) and is visible to other work-items.

`_sr` - shared register write/read fence. No shared register writes/reads can be re-ordered or moved across this fence instruction. Also, all prior writes to shared registers done by this work-item become visible to all other work-items.

`_mem_write_only` - same as `_memory`, except that the memory import (load) instructions can move across this fence instruction. If this happens, the load instruction must be disambiguated as not an alias of store instructions before the fence.

`_mem_read_only` - same as `_memory` except that the memory export (store) instructions can move across this fence instruction. If this happens, the store instruction must be disambiguated as not an alias of load instructions before the fence.)

`_gds`: shared memory fence. Ensures that (1) no GDS read/write instructions can be reordered or moved across this fence instruction; and (2) all GDS write instructions are complete in the sense that the data has been written to GDS memory (not to internal buffers). Note that this option does not cause synchronization between members of a work-group. This requires the addition of the work-items option.

All prior writes to shared registers done by this work-item become visible to all other work-items.

Pixel kernels can only use fence instructions for global memory. Compute kernels can use all of options. In pixel kernels, use of discard instructions implies a fence_memory.

Use of discard with a `fence_threads` instruction is undefined in IL, although specific implementations can select an interpolation.

DX11 has `sync_uglobal` and `sync_ulocal`. They both are m apped to `fence_memory` in IL.

Valid for R7XX GPUs and later.

Formats 0-input, 0-output, no additional token.

Examples

```
lds_write_vec  mem._y__,  r0.yyyy
fence_threads_lds
lds_read_vec r1,  r0.xy
```

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_FENCE
_threads	16	1 = this is a fence for work-item synchronization.

Fence for Synchronization Work-Items, and/or LDS and/or GDS and/or Global Memory or UAV (Cont.)

	`_lds`	17	1 = this is a fence for the LDS.
	`_memory`	18	1 = this is a fence for global memory or UAV.
	`_sr`	19	1 = this is a fence for *sr*.
	`_mem_write_only`	20	1 = this is a global memory or UAV write-only fence
	`_mem_read_only`	21	1 = this is a global memory or UAV read-only fence.
	`_gds`	22	1 = this is a fence for gds.
	`Controls`	29:20	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Examples

```
lds_write_vec mem._y__, r0.yyyy
fence_threads_lds
lds_read_vec r1, r0.xy
```

Related None.

Fetch Data From Four Texels Into One Register

Instructions **FETCH4**

Syntax

```
fetch4_resource(n)_sampler(m)[_coordtype(ILTexCoordMode)]_compselect(comp)[_ad
droffimmi(u,v,w)] dst, src0
```

```
fetch4 _ext_
resource(n)_sampler(m)[_resourcetype(pixtexusage)][_coordtype(ILTexCoordMode)]
[_compselect(comp)][_addroffimmi(u,v,w)] dst, src0,src1, scr2
```

Description

Gathers the four texels to be used in a bi-linear filtering operation and packs them into a single register. Only works with 2D, 2D array, cubemaps, and cubemap arrays. For 2D textures, only the addressing modes of the sampler and the top level of any mip pyramid are used. Set W to zero.

This behaves like the `SAMPLE` instruction, but a filtered sample is not generated. The four samples that contribute to filtering are placed into xyzw in counter-clockwise order, starting with the sample to the lower-left of the queried location. This is the same as point sampling with the (u,v) texture coordinate delta at the following locations: (-,+),(+,+),(+,-),(-,-), where the magnitude of the deltas are half a texel.

If component select is present, it specifies the component to fetch for a multi-component texture resource. If it is absent, x channel is fetched.

This instruction has the same restrictions as the `SAMPLE` instruction.

If the `pri_modifier_present` bit is set, component select appears at the next Dword. Its value is of enumerated type ILComponentSelect. Only `IL_COMPSEL_X_R`, `IL_COMPSEL_Y_G`, `IL_COMPSEL_Z_B`, and `IL_COMPSEL_W_A` are valid.

If the sign bit of the control field is set, offsets appear at the following Dword (bit29).

The first syntax example is valid for R5XX GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.

Format

1-input, 1-output.

3-input, 1-output.

Opcode

Field Name	Control Field			
`code`	15:0	IL_OP_FETCH4		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related FETCH4_PO_C, FETCH4C, FETCH4po.

FETCH4 and Texel Comparison

Instructions **FETCH4_PO_C**

Syntax

```
fetch4poc_resource(n)_sampler(m)_compselect(comp) dst, src0, src1, src2

fetch4poc _ext_
resource(n)_sampler(m)[_coordtype(ILTexCoordMode)][_resourcetype(pixtexusage)
]_compselect(comp)] dst, src0,src1, scr2, src3, src4
```

Description

See `SAMPLE_C` for how `src1.x` is compared against each fetched texel. Unlike `SAMPLE_C`, `FETCH4_PO_C` returns each comparison result, rather than filtering them. For texturecube corners, where there are three real texels and a fourth is synthesized, the synthesis must occur after the comparison step. The returned comparison result for the synthesized texel can be 0, 0.25, 0.75, or 1.

`Src1.x` is the value to compare against; `src2.xy` are offsets as in `FETCH4_PO`.

If component select is present, it specifies the component to fetch for a multi-component texture resource. If it is absent, the x channel is fetched.

This instruction has the same restrictions as `SAMPLE`.

`FETCH4C` only works with the set of formats that work with sample c.

If the `pri_modifier_present` bit is set, component select appears at the next Dword. Its value is of enumerated type `ILComponentSelect`. Only `IL_COMPSEL_X_R`, `IL_COMPSEL_Y_G`, `IL_COMPSEL_Z_B`, and `IL_COMPSEL_W_A` are valid.

Valid for Evergreen GPUs and later. The second syntax example supports indexing.

Format

3-input, 1-output.

5-input, 1-output.

Opcode

Field Name	Control Field			
code	15:0	IL_OP_FETCH4_PO_C		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = aoffimmi does not exist.
				1 = aoffimmi exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related FETCH4, FETCH4C, FETCH4po.

FETCH4 and Compare Texels

Instructions **FETCH4C**

Syntax

```
fetch4c_resource(n)_sampler(m)[_coordtype(ILTexCoordMode)][_compselect(comp)]
[_addroffimmi(u,v,w)] dst, src0, src1

fetch4c _ext_
resource(n)_sampler(m)[_coordtype(ILTexCoordMode)][_resourcetype(pixtexusage)
][_compselect(comp)][_addroffimmi(u,v,w)] dst, src0,src1, scr2, src3
```

Description See existing SAMPLE_C for how src1.x is compared against each fetched texel. Unlike SAMPLE_C, FETCH4_C returns each comparison result, rather than filtering them. For texture cube corners, where there are three real texels and a fourth is synthesized, the synthesis must occur after the comparison step. The returned comparison result for the synthesized texel can be 0, 0.25, 0.75, or 1.

If component select is present, it specifies the component to fetch for a multi-component texture resource. If it is absent, the x channel is fetched.

This instruction has the same restrictions as SAMPLE.

FETCH4C only works with the set of formats that work with sample c.

If the pri_modifier_present bit is set, component select appears at the next Dword. Its value is of enumerated type ILComponentSelect. Only IL_COMPSEL_X_R, IL_COMPSEL_Y_G, IL_COMPSEL_Z_B, and IL_COMPSEL_W_A are valid.

If the sign bit of the control field is one, offsets appear at the following Dword (bit29).

Valid for Evergreen GPUs and later. The second syntax example supports indexing.

Format 2-input, 1-output.

4-input, 1-output.

Opcode

Field Name	Control Field			
code	15:0	IL_OP_FETCH4_C		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = aoffimmi does not exist. 1 = aoffimmi exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related FETCH4, FETCH4_PO_C, FETCH4Cpo.

Extends FETCH4 for Offsets to be Programmable

Instructions	**FETCH4po**

Syntax	`fetch4po_resource(n)_sampler(m)[_compselect(comp)] [_coordtype(ILTexCoordMode)] dst, src0, src1`
	
	`fetch4po _ext_ resource(n)_sampler(m)[_resourcetype(pixtexusage)][_compselect(comp)] dst, src0,src1, scr2, src3`

Description	Extends the `FETCH4` offset range to be larger and programmable. The `po` suffix stands for programmable offset. Instead of supporting an immediate offset [-8...7], the offset comes as a parameter to the instruction, and also has larger range of [-32...31]. The first two components of the four-vector `src1` parameter supply 32-bit integer offsets. The other components of this parameter are ignored. The six least-significant bits of each offset value are taken as a signed value, yielding a range of [-32...31]. `GATHER4PO` only works with 2D textures. Valid modes in the sampler are the addressing modes. Only the most detailed mip in the resource view is used. Note: if the address falls on a texel center, it does not mean the other texels can be zeroed out. If component select is present, it specifies the component to fetch for a multi-component texture resource. If it is absent, the x channel is fetched. If the `pri_modifier_present` bit is set, component select appears at the next Dword. Its value is of enumerated type `ILComponentSelect`. Only `IL_COMPSEL_X_R`, `IL_COMPSEL_Y_G`, `IL_COMPSEL_Z_B`, and `IL_COMPSEL_W_A` are valid. This instruction has the same restrictions as the `SAMPLE`. Valid for Evergreen GPUs and later. The second syntax example supports indexing.
Format	2-input, 1-output. 4-input, 1-output.

Opcode

Field Name	Control Field			
code	15:0	IL_OP_FETCH4_PO		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = `aoffimmi` does not exist. 1 = `aoffimmi` exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related FETCH4, FETCH4_PO_C, FETCH4C.

Pixel Kill

Instructions	**KILL**
Syntax	`kill[_stage(n)_sample] src0`

Description

Component-wise test to terminate current pixel shader execution and discard all results.

If sample is set to 1, KILL does not use the actual value of *src0* to test. Instead, the shader performs a KILL based upon a texture sample at the coordinate specified by *src0* on the texture stage/unit specified by stage.

It is an error to use this instruction in a vertex or geometry shader.

To kill based on a subset of the four components, set the swizzle for the component not used for the test to IL_COMPSEL_1.

Valid for all GPUs.

Operation:

```
VECTOR v;
If(sample == 1)
{
    v = Sample(src0, stage);
}
else
{
    v = EvalSource(src0);
}
if((v[0] < 0.0) || (v[1] < 0.0) || (v[2] < 0.0) || (v[3] < 0.0))
{
    Discard outputs;
    Terminate pixel shader;

}
```

Format 1-input, 0-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_KILL
control	29:16	see below
reserved	31:25	Must be zero.

control (29:16):

Field Name	Bits	Description
stage	23:16	Texture stage or unit number, 0 to 255, if sample is 1; otherwise, must be zero.
sample	24	0: *src0* is the test value. 1: Test value is sampled from texture stage or unit *stage*. *src0* is the sample coordinate, and the resulting sample is used as the test value.

`IL_Src token (src0)`

`IL_Src_Mod` token: only present if modifier_present field is 1 in previous IL_Src token.

`IL_Src token (src0`: only present if relative_address field is 1 in the preceding IL_Src or IL_Dst token.

Related DISCARD_LOGICALNZ, DISCARD_LOGICALZ.

Read Local Data Share (LDS) Memory into a Vector

Instructions **`LDS_READ_VEC`**

Syntax `lds_read_vec (_neighborExch)(_sharingMode) dst, src0.xy`

Description Reads the LDS memory into the *dst* register (can read a vector of up to four components). This uses the ownership accessing model. Each work-item owns a part of the LDS memory. The LDS location is specified using the owner work-item ID and offset, such as “Read data written/owned by work-item Tid at Offset.”

Register *src0* specifies the location with *src0*.x = Tid, *src0*.y = offset.

Dst can be any writable register.

The sharing modes _sharingMode(rel) and _sharingMode(abs) are relative and absolute, respectively. The flag is optional. If it is not specified, the instruction takes the default sharing mode specified by `DCL_LDS_SHARING_MODE`.

The flag neighborExch is optional. If it is specified, the output of lds memory is exchanged with its neighbor work-items, so that the first work-item receives all values from x-components, the second work-item receives all values from y-components, etc. This flag is useful for applications like FFT matrix transpose.

It is used only in a compute shader.

Valid only for R7XX GPUs.

Formats 1-input, 1-output, 0 additional token.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_VEC
`control`	17:16	_sharingMode: 0: IL_LDS_SHARING_MODE_RELATIVE 1: IL_LDS_SHARING_MODE_ABSOLUTE 2-3: reserved
	18	_neighborExch:0: off 1: on and enabled
	29:19	Not used. Must be 0
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Examples

```
lds_read_vec r1, r0.xy
lds_read_vec_neighborExch r1, r0.xy
lds_read_vec_sharingMode(rel) r1, r0.xy
lds_read_vec_neighborExch_sharingMode(abs) r1, r0.xy
```

Related LDS_WRITE_VEC

Write a Vector to Local Data Share (LDS) Memory

Instructions	**`LDS_WRITE_VEC`**
Syntax	`lds_write_vec _lOffset(N) (_sharingMode)` *`dst`*`,` *`src0`*

Description

Writes data (a vector of up to four components) in *src0* into LDS memory. This uses the ownership accessing model: each work-item owns a part of the LDS memory. Each work-item can write only to the part it owns. The location is specified by offset.

src0 can be any register.

Dst must be of type `IL_REGTYPE_GENERIC_MEM`. This is used to provide the write mask.

`_lOffset()` is optional. If not specified, the offset is 0. If it is specified, such as `_lOffset(n)`, *n* must be a value of multiples of four in the range of [0,60], and must be smaller than the declared `lds_size_per_thread`.

`_sharingMode(rel)` or `_sharingMode(abs)` are for relative or absolute sharing mode, respectively. The flag is optional. If it is not specified, the instruction takes the default sharing mode specified by `DCL_LDS_SHARING_MODE`.

This instruction is used only in a compute kernel.

Valid for R7XX GPUs only.

Formats

1-input, 1-output, no additional token.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_WRITE_VEC
`control`	17:16	_sharingMode: 0: IL_LDS_SHARING_MODE_RELATIVE 1: IL_LDS_SHARING_MODE_ABSOLUTE 2-3: reserved
	23:18	_ lOffset: offset within its reserved space, must be one of [0,4, 8, … 60]
	29:24	Not used. Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Examples

```
lds_write_vec mem._y__, r0.yyyy
lds_write_vec_lOffset(4) mem.xy__, r0.xyzw
lds_write_vec_sharingMode(rel) mem.__zw, r0.xyzw
lds_write_vec_lOffset(4)_sharingMode(abs) mem.x_z_w, r0.xyzw
```

Related

LDS_READ_VEC.

LOAD from Buffer

Instructions **LOAD**

Syntax `load_resource(`*n*`)[_aoffimmi(`*u,v,w*`)]` *dst, src0*

`load_ext_resource(`*n*`)[_resourcetype(pixtexusage)][_addroffimmi(`*u,v,w*`)]` *dst, src0*

Description Simplified alternative to the "sample" instruction. Using the provided signed integer address, LOAD fetches data from the specified buffer or texture without any filtering (for example, point sampling). The source data can come from any resource type except TextureCube.

src0 must specify a single component used as the address.

DX10 allows an output swizzle on the ld instruction. IL requires an additional move if the swizzle is used.

Unlike SAMPLE, LOAD can fetch data from buffers. The buffer with the data being fetched is identified by the resource id stored in the control field.

src0 provides the set of texture coordinates needed to sample the texture in the form of signed integers.

If the *src*Address is out of the range [0, (#texels in dimension -1)], the load returns 0.

src0.a (post-swizzle) always provides a signed integer mipmap level. If the value is out of range [0, (num miplevels in resource-1)], the load returns 0. If the resource has no mipmaps, *src0*.a is ignored.

src0.gb (post-swizzle) are ignored for buffers and Texture1D (non-Array). *src*Address.b (post-swizzle) is ignored for Texture1D Arrays and Texture2Ds.

For Texture1D Arrays, *src0*.g (post-swizzle) provides the array index as a signed integer. If the value is out of range of the available array (indices [0, (Array size-1)]), the load returns 0.

For Texture2D Arrays, *src0*.b (post-swizzle) provides the array index; otherwise, it has the same semantics as Texture1D, described above.

Aoffset and indexed are allowed.

The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later.

<u>Address Offset</u>

- If the optional `_aoffimmi` suffix is included (bit 29 of the Opcode token is set), a 32-bit value containing the packed offsets immediately follows the Opcode token (and modifier, if present).
- The `optional` field indicates that the texture coordinates for the load are to be offset by the provided immediate texel space integer constant values. The literal values must be a set of unnormalized (texel space) values represented as signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5]. *src1* can be used with all resources, including Texture1D/2D Arrays and Texture3D; however, it cannot be used with a resource of type TextureCube.
- Individual chips might not support the full range. For example, R6XX and later GPUs support only s3.1 with a range of [-8.0..7.5]. If the GPU does not support the full range, the upper bits are ignored. The Southern Island family of devices support the range [{32..31] and ignore the fractional bit.
- The offsets are added to the texture coordinates, in texel space, relative to the miplevel being accessed by the load.
- Address offsets are not applied along the array axis of Texture1D/2D Arrays.
- `_aoffimmi` v,w components are ignored for Buffers, and Texture1Ds.
- `_aoffimmi` w component is ignored for Texture2Ds.

LOAD from Buffer (Cont.)

Return Type Control

The data format returned by LOAD to the *dst* register is determined in the same way as described for the sample instruction; it is based on the format bound to the Resource parameter (n).

As with the SAMPLE instruction, returned values for load are four-component vectors (with format-specific defaults for components not present in the format).

DX10 allows a swizzle on the resource that is used to determine how to swizzle the four-component result back from the texture load, after which `.mask` on *dst* is used to determine which components in *dst* is updated. The equivalent effect can be achieved in IL by adding an extra move after LOAD.

The resource cannot be a TextureCube or a Constant Buffer.

Out-of-bounds access on any axis always results in 0 as the result of the memory fetch.

Formats 3-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_LOAD		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related LOAD_FPTR.

Load Fragment Pointer of an MSAA Resource

Instructions **LOAD_FPTR**

Syntax

```
load_fptr_resource(n)[_addroffimmi(u,v,w)] dst, src0
load_fptr_ext_resource(n)[_resourcetype(pixtexusage)][addrommi(u,v,w)] dst,
src0, src1
```

Description Similar to LOAD. It differs from `LOAD` in the following ways:

1. Resource must be 2DMSAA or 2D array MSAA.
2. Src0.a is ignored.
3. Return data is a bit field of fragment pointers for each sample, from the lsb. Each four bits correspond to a sample.

Let N be number of sample of the given resource.

```
If N > 1
then
For each i < N,
  result[4*i+3 : 4*i] = fragment pointer for sample i, valid values are 0 to N - 1
For each i >= N,
  result[4*i+3 : 4*i] = 0
else
  result is undefined
```

Valid for Evergreen GPUs and later.

Formats 1-input, 1-output, no additional token.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_LOAD_FPTR		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = `aoffimmi` does not exist. 1 = `aoffimmi` exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related LOAD.

Get Texture Mipmap Level of Detail

Instructions **LOD**

Syntax `lod_stage(n) dst, src0`

Description Uses the provided texture coordinate to determine the computed mipmap level(s) sampled from at *src0*. The LOD value returned includes any clamping and biasing defined by the texture currently bound to the specified texture unit. This instruction cannot be used on a stage that has not been declared with a DCLPT instruction.

You can project using the `divComp` source modifier on *src0*. If the `divComp` source modifier is used, and the component to divide by is negative, the result of this instruction is undefined (if IL_DIVCOMP_Y is used, and the second component of *src0* is negative, the results of this texture load is undefined).

Operation:

src0 is the texture coordinate with which the mipmap LOD is determined. The control field of the IL_Opcode token specifies the texture stage to determine the LOD. Associated with the sampler specified by control are 1) a texture, 2) the texture's dimension, and 3) all filter, wrap, bias, and clamp states. The LOD value is replicated on each component of *dst*.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_LOD The first eight bits specify the texture stage/unit from which to determine the texture's level of detail (stage).
	`control`	29:16	**Field Name / Bits / Description:** stage — 23:16 — Texture stage or unit number. *reserved* — 29:24 — Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	IL_Dst token (*dst*)		
3	IL_Dst_Mod token is present only if the modifier_present field is 1 in previous IL_Dst token.		
4	IL_Rel_Addr token is present only if the relative_address field is 1 in the preceding IL_Src or IL_Dst token.		
5	IL_Src token (*src0*).		
6	IL_Src_Mod token is present only if the modifier_present field is 1 in previous IL_Src token.		
7	IL_Rel_Addr token is present only if the relative_address field is 1 in the preceding IL_Src or IL_Dst token.		

Related None.

Export Data to Memory Stream

Instructions **`MEMEXPORT`**

Syntax `memexport_exportStream(`*`n`*`)_elemOffset(`*`i`*`)` *`src0`*`,` *`src1`*

Description Exports a value in *src1* to a memory stream. *exportStream* corresponds to the number of the export stream buffer to which data is written. *elemOffset* specifies the offset in terms of elements (not bytes or Dwords) to which data is written. *src0*.x contains the index to which data is written. *src1* contains the data to write to the export stream.

Note that this instruction is available only on graphics cards that support memory export.

Valid for R670 GPUs and later.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_MEMEXPORT
	`exportStream`	21:16	Export stream number, 0 to 16
	reserved	22	Must be zero.
	`elemOffset`	28:23	Offset from the start of the export stream in elements.
	reserved	31:29	Must be zero.
2	IL_Src token (*src0*).		
3	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.		
4	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src token.		
5	IL_Src token (*src1*).		
6	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.		
7	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src token.		

Related MEMIMPORT.

Import Memory from Stream

Instructions **MEMIMPORT**

Syntax `memimport_elem(n) dst, src0`

Description Imports data from buffers set externally. This instruction provides an alternate means of fetching vertex element data using an arbitrary index. Refer the DCLV instruction for normal operation. This instruction typically is used only for higher-order surface shaders (see the INITV instruction).

In a vertex shader:

- the value of *elem* corresponds to vertex buffer element *n*.
- if the vertex buffer corresponding to *elem* is disabled in state, apply defaults when this instruction is executed.

In a pixel shader

- the value of *elem* corresponds to pixel buffer element *n*.
- if the pixel buffer corresponding to *elem* is disabled in state, apply defaults when this instruction is executed.

The first component of *src0* (*src0*.x) contains the index for the element from which to fetch. The value is floored before it is used.

dst contains the data to be fetched from memory.

dst can only be a TEMP register.

Valid for R670 GPUs and later.

Operation:

```
VECTOR v;
v = Fetch(importElement, FLOOR(src0.x));
WriteResult(v, dst);
```

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_MEMIMPORT The lower six bits are set to a number between 0 and 16 inclusive, specifying the import element from which memory is loaded/fetched (`elem`).
	`elem`	21:16	Element number, 0 to 16
	reserved	31:22	Must be zero.
2	IL_Dst token (*dst*) where `register_type` is set to IL_REGTYPE_TEMP.		
3	IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.		
4	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		
5	IL_Src token (*src0*).		
6	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.		
7	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		

Related MEMEXPORT.

Query Information from a Resource

Instructions **RESINFO**

Syntax

```
resinfo_resource(n)[_uint] dst, src0
resinfo_ext_resource(n)[_resourcetype(pixtexusage)][_uav][_uint] dst, src0
```

Description

src0.x is the 2's complement integer scalar.

DX10 supports both destination swizzles and repFloat modifiers on this instruction. Clients must insert additional IL instructions to achieve the same effect.

dst receives <*width*, *height*, *depth*, *total mip count*>.

If *src0*.x is out of the range of the available number of miplevels in the resource, [0, (*numMipLevels* – 1)], then resinfo returns [0, 0, 0, *total mip count*].

The returned width, height, and depth values are for the mip-level selected by *src0*.x; they are in number of texels, independent of texel data size.

Returned values are all floating point, unless the control field is negative (_uint modifier is used), in which case the returned values are integers.

dst.w always receives total-mip-count (if .w is included in write mask). Thus, the total-mip-count is independent of *src0*.x.

DX10 allows a destination swizzle. IL, however, requires an extra move to allow the returned values to be swizzled arbitrarily before they are written to the destination.

If the resource is not a Texture3D or Texture1D/2D with Array > 1, then depth is zero.

For a Texture1D/2D with Array > 1 or Texture3D, depth always represents the number of array slices.

If the resource is a Buffer or Texture1D, the height is zero, unless it is a Texture1D Array, in which case height is the number of array slices.

If the resource is a TextureCube, then width and height represent individual cube face dimensions, and depth is zero.

If *src0*.x is out of the range of the available number of miplevels in the resource, then resinfo returns [0, 0, 0, *total mip count*].

The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later.

Formats

1-input, 1-output.

3-input, 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_RESINFO		
control	29:16	**Field Name**	**Bits**	**Description**
		resource_id	23:16	Resource ID.
		return type	24	0x0: Return type is Float.
				0x1: Return type is Integer.
		reserved	29:25	Must be zero.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related None.

Sample Data from Resource with Filter

Instructions **SAMPLE**

Syntax

`sample_resource(n)_sampler(m)[_coordtype(ILTEXCOORDMODE)][_aoffimmi(u,v,w)] dst, src0`

`sample_ext_resource(n)_sampler(m)_resourcetype(pixetexusage)[_coordtype(ILTEXCOORDMODE)][_addroffimmi(u,v,w)] dst, src0, src1, src2`

Description

Samples data from the specified Element/texture using the filtering mode identified by the given sampler. The source data can come from any resource type other than buffers. *src0* provides the set of texture coordinates needed sampling as floating point values, referencing normalized space in the texture. Address wrapping modes (wrap, mirror, clamp, border, etc.) are applied to texture coordinates outside [0...1] range, taken from the sampler state, and applied after any address offset is applied to texture coordinates (see Address Offset section, below).

For Texture2D Arrays, *src0*.b (post-swizzle) selects the Array Slice from which to fetch and uses the same semantics described for Texture1D Arrays.

The first syntax example above is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second syntax example supports indexing.

Address Offset

The optional `_aoffimmi` suffix is specified by the `addroff_present` bit in IL_SampleOpCode. If set, a 32 bit value containing the packed offsets immediately follows the opcode token and modifier, if present.

The optional *address offset by immediate integer* indicates that the texture coordinates for the sample are offset by the provided immediate texel space integer constant values. The literal values must be a set of unnormalized (texel space) values represented as Signed fixed point with 1 bit of fraction (7.1) that allows ranges from [–64,63.5]. This modifier is defined for all resources, including Texture1D/2D Arrays and Texture3D; but it is undefined for TextureCube.

The offsets are added to the texture coordinates, in texel space, relative to each mip level being accessed. Thus, even though texture coordinates are provided as normalized float values, the offset applies a texel-space integer offset. Address offsets are not applied along the array axis of Texture1D/2D Arrays.

The v and w components are ignored for buffers and Texture1Ds.

The w component is ignored for Texture2Ds.

Address wrapping modes (wrap, mirror, clamp, border, etc.) from the sampler state are applied after any address offset is applied to texture coordinates.

Return Type Control

The data format returned by sample to the destination register is determined by the resource format (`WGFFMT*`). For example, if the resource format is `WGFFMT_A8B8G8R8_UNORM_SRGB`, the sampling operation converts sampled texels from gamma 2.0 to 1.0, applies filtering, and the result are written to the destination register as floating point values in the range [0,1].

Returned values are 4-vectors (with format-specific defaults for components not present in the format).

DX10 supports an output swizzle in this instruction. To achieve the same effect in IL, use an extra move.

Sample Data from Resource with Filter (Cont.)

LOD Calculation

See the `deriv_rtx` and `deriv_rty` instructions on how derivatives are calculated while determining LOD for filtering. The sample instruction implicitly computes derivatives on the texture coordinates using the same definition that the `deriv*` Shader instructions use. This does not apply to `sample_l` or `sample_g` instructions; for these, LOD or gradients are provided directly by the application.

Miscellaneous Details

src0.w (post-swizzle) must be zero. *src0* provides the set of texture coordinates to perform the sample as floating point values referencing normalized space within the texture. Address wrapping modes (wrap, mirror, clamp, border, etc.) are applied for texture coordinates outside the [0, 1] range, taken from the sampler state, and applied after any address offset (see subsection, above) is applied to texture coordinates.

The information required for the hardware to perform sampling is split into two orthogonal parts. The first part (texture register), provides source data type information, including if the texture contains SRGB data; also, it references the memory being sampled. The second part (sampler register), defines the filtering mode to apply to the sample operation.

For Texture1D Arrays, *src0*.y (post-swizzle) selects which Array Slice to fetch from. This is treated as a scaled float value, as opposed to the normalized space for standard texture coordinates; and a round-to-nearest even is applied on the value, followed by a clamp to the available BufferArray range.

Formats 1-input, 1-output or 3-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLE		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id.
		`sampler`	27:24	sampler_id.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Resource with Filter and Bias

Instructions

`SAMPLE_B`

Syntax

`sample_b_resource(`*`n`*`)_sampler(`*`m`*`)[_coordtype(ILTEXCOORDMODE)][_aoffimmi(u,v,w)]` *`dst, src0 , src1`*

`sample_b_ext_resource(n)_sampler(m)[_resourcetype(pixtexusage)][_coordtype(ILTEXCOORDMODE)][_addroffimmi(u,v,w)] dst, src0, src1, src2, src3`

Description

Samples data from the specified element/texture using the filtering mode identified by the given sampler. The source data can come from any resource type except for buffers. An additional bias is applied to the level of detail computed as part of the instruction execution.

SAMPLE_B is like the SAMPLE instruction with the addition of applying the specified srcLOBBias (in *src1*) value to the level of detail value prior to selecting the mip map.

src1 must be either a literal value or a replicated single component.

SAMPLE_B has the same restrictions as the SAMPLE instruction.

If the sign nbit of the control field is 1, offsets appear at the next word.

The first syntax example above is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second one also supports indexing.

Formats

2-input, 1-output or 4-input, 1-output.

Opcode

Field Name	Bits	Description			
`code`	15:0	IL_OP_SAMPLE_B			
`control`	29:16	**Field Name**	**Bits**	**Description**	
		`resource`	23:16	resource_id, 0 to 255.	
		`sampler`	27:24	sampler_id, 0 to 15.	
		`arguments`	28	indexed_args.	
		`aoffimmi`	29	0 = `aoffimmi` does not exist.	
				1 = `aoffimmi` exists.	
`sec_modifier_present`	30	Must be zero.			
`pri_modifier_present`	31	Must be zero.			

Related

SAMPLE, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Resource with Filter and Comparison

Instructions **`SAMPLE_C`**

Syntax

`sample_c_resource(`*m*`)_sampler(`*n*`)sample_c_resource(`*m*`)_sampler(`*n*`)[_coordtype(ILTexCoordMode)][_addroffimmi(u,v,w)]` *`dst, src0, src1`*

`sample_c_ext_resource(`*m*`)_sampler(`*n*`)[_resourcetype(pixtexusage)][_coordtype(ILTexCoordMode)][_addroffimmi(u,v,w)]` *`dst, src0, src1, src2, src3`*

Description

Performs a comparison filter. *src0* is the index. *src1* contains the reference value. The primary purpose for SAMPLE_C is to provide a building-block for percentage-closer depth filtering. The C in SAMPLE_C stands for comparison. The source data can come from any resource type, other than buffer.

The operands to SAMPLE_C are identical to those of SAMPLE, except for an additional float32 source operand, *src1*.

If SAMPLE_C is used with a resource that is not a texture1D/2D/Cube, or an unsupported format, then this instruction produces undefined results. It also produces undefined results if used with texture arrays.

When the SAMPLE_C instruction is executed, the hardware uses the current sampler's comparison function to compare *src1*.x against the corresponding texture value at each filter “tap” location (texel) that the currently configured texture filter covers, based on the provided coordinates (*src0*).

The comparison occurs after the source texel's x-component is converted to float32, prior to filtering texels.

For texels that fall off the resource, the x-component value is determined by applying the address modes (and BorderColorR, if in Border mode) from the sampler.

Each comparison that passes returns 1.0f as the x-component value for the texel; each comparison that fails returns 0.0f as the x value for the texture. Then, filtering occurs as specified by the sampler states, operating only in the x-component, and returning a single scalar filter result to the shader (replicated to all masked *dst* components).

The use of SAMPLE_C is orthogonal to all other general-purpose filtering controls (SAMPLE_C works with the other general-purpose filter modes). What SAMPLE_C changes the behavior of the general-purpose filters so that the values being filtered all become 1.0f or 0.0f (comparison results).

See the SAMPLE instruction description for operation of this instruction other than those specified here.

The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing. Source src0 is the index; source src1.x has the reference value.

Formats 2-input, 1-output or 4-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLE_C		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.

Sample Data from Resource with Filter and Comparison

	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related SAMPLE, SAMPLE_B, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data From Resource (Not Buffer) With Filter and Comparison

Instructions **`SAMPLE_C_B`**

Syntax

```
sample_c_b_resource(n)_sampler(m)[_coordtype(ILTexCoordMode)][_addroffimmi(u,
v,w)] dst, src0 , src1, src2
```

```
sample_c_b_ext_
resource(n)_sampler(m)[_resourcetype(pixtexusage)][_coordtype(ILTexCoordMode)
][_addroffimmi(u,v,w)] dst, src0, src1, scr2, src3, src4
```

Description Samples data from the specified element/texture using the filtering mode identified by the given sampler. The source data can come from any non-array resource type, other than buffers.

This instruction behaves exactly as the `sample_c_z` instruction, except that `src2.x` contains a LOD bias value. This instruction produces undefined results if used with texture arrays.

This instruction has the same restrictions as `SAMPLE_C_L`.

The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.

Format 3-input, 1-output.

5-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLE_C_B		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data From Element/Texture With Gradient Using Filter

Instructions **SAMPLE_C_G**

Syntax sample_c_g*_resource(n)_sampler(m)*[_coordtype(ILTexCoordMode)] *[_addroffimmi(u,v,w)] dst, src0, src1, src2, src3*

sample_c_g*_ext_resource(n)_sampler(m) [_resourcetype(pixtexusage)]*[_coordtype(ILTexCoordMode)] *[_addroffimmi(u,v,w)] dst, src0, src1, src2, src3, src4, src5*

Description Sample data from the specified element/texture using the filtering mode identified by the given sampler. The source data can come from any resource type other than buffers.

Behaves like the SAMPLE_C, except that gradients for the source address in the x direction and the y direction are provided through *src2* and *src3*, respectively.

The x, y, and z components of *src2*/*src3* (post-swizzle) provide du/dx, dv/dx, and dw/dx. The w component (post-swizzle) is ignored.

This instruction has the same restrictions as the SAMPLE_C and SAMPLE_G instructions.

The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing. For R7xx and later GPUs, this instruction works on all resource types, other than buffers. For R6xx GPUs, this instruction works on non-array resource types, other than buffers. This instruction produces undefined results if it is used on an unsupported format.

Format 4-input 1-output or 6-input 1-ouput.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_SAMPLE_C_G		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = aoffimmi does not exist.
				1 = aoffimmi exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Element/Texture with LOD

Instructions	**SAMPLE_C_L**
Syntax	sample_c_l*_resource(n)_sampler(m)*[_coordtype(ILTexCoordMode)] *[_addroffimmi(u,v,w)] dst, src0 , src1, src2* sample_c_l*_ext_resource(n)_sampler(m) [_resourcetype(pixtexusage)]*[_coordtype(ILTexCoordMode)] *[_addroffimmi(u,v,w)] dst, src0 , src1, src2, src3, src4*
Description	Sample data from the specified element/texture using the filtering mode identified by the given sampler. The source data can come from any non-array resource type other than buffers. This instruction produces undefined results if used with texture arrays. Behaves like the SAMPLE_C instruction, except that LOD is provided directly through *src2*.x. This instruction has the same restrictions as the SAMPLE_C and SAMPLE_L instructions. The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.
Format	3-input 1-output or 5-input 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_SAMPLE_C_L		
control	29:16	**Field Name**	**Bits**	**Description**
		resource	23:16	resource_id, 0 to 255.
		sampler	27:24	sampler_id, 0 to 15.
		arguments	28	indexed_args.
		aoffimmi	29	0 = aoffimmi does not exist. 1 = aoffimmi exists.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Resource with Filter and Comparison Level Zero

Instructions	**`SAMPLE_C_LZ`**
Syntax	`sample_c_lz_resource(`*n*`)_sampler(`*m*`)[_coordtyped(ILTexCoordMode][_addroffimmi(u,v,w)]` *dst, src0, src1* `sample_c_lz_ext_resource(`*n*`)_sampler(`*m*`)[_resourcetype(pixtexusage)][_coordtyped(ILTexCoordMode][_addroffimmi(u,v,w)]` *dst, src0, src1, src2, src3*
Description	Performs a comparison filter. SAMPLE_C mainly provides a building-block for Percentage-Closer Depth filtering. The 'c' in SAMPLE_C stands for comparison. *src0* is the index. *src1*.x contains the reference value. Same as SAMPLE_C, except LOD is 0, and derivatives are ignored (as if they are 0). The LZ stands for level-zero. Because derivatives are ignored, this instruction is available in vertex and geometry shaders. It can also be used inside of control flow. The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.
Formats	2-input, 1-output or 4-input, 1-output.

Opcode

Field Name	**Bits**	**Description**		
`code`	15:0	IL_OP_SAMPLE_C_LZ		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_G, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Resource with Filter and Gradient

Instructions **SAMPLE_G**

Syntax

```
sample_g_resource(n)_sampler(m)[_coordtyped(ILTexCoordMode)][_addroffimmi(u,v
,w)] dst, src0, src1, src2
```

```
sample_g_ext_resource(n)_sampler(m)[_resourcetype(pixtexusage)][_coordtyped(IL
TexCoordMode][_addroffimmi(u,v,w)] dst, src0, src1, src2, src3, src4
```

Description

Samples data from the specified texture using the filtering mode identified by the given sampler. The source data can come from any Resource Type, other than Buffers.

SAMPLE_G behaves exactly as the “sample” instruction, except that gradients for the source address in the x direction and the y direction are provided by extra parameters, *src1* and *src2*, respectively.

The x, y, and z components of *src1*/*src2* (post-swizzle) provide du/dx, dv/dx and dw/dx. The w component (post-swizzle) is ignored.

The w component (post-swizzle) is ignored.

This instruction has the same restrictions as the SAMPLE instruction.

The first syntax example (not indexed) is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.

Formats 3-input, 1-output or 5-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLE_G		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_L, SAMPLEINFO, SAMPLEPOS.

Sample Data from Specified Memory Using Given Sampler. Source Data can come from any Resource Type

Instructions	**`SAMPLE_L`**

Syntax	`sample_l_resource(`*`n`*`)_sampler(`*`m`*`)[_coordtyped(ILTexCoordMode] `*`dst, src0, src1`*
	`sample_l_ext_resource(`*`n`*`)_sampler(`*`m`*`)[_resourcetype(pixtexusage)][_coordtyped(ILTexCoordMode][_addroffimmi(u,v,w)] `*`dst, src0, src1, src2, src3`*

Description	This is identical to the SAMPLE instruction (7-90), except that the level of detail (LOD) is provided directly by the application as a scalar value, representing no anisotropy. This instruction also is available in all programmable shader stages, not only the pixel shader (as with SAMPLE). It samples the texture using *src1*.x to choose the LOD. If the LOD value is negative, the 0'th (biggest map) is chosen with MAGFILTER applied. Since *src1*.x is a floating point value, the fractional value is used to interpolate (if MIPFILTER is linear) between two mip levels. This instruction ignores address gradients (filtering is isotropic). See the description of the SAMPLE instruction for operational details of this instruction other than the LOD calculation. Note that when used in the pixel kernel, `sample_l` implies the choice of LOD is per-pixel, with no effect from neighboring pixels. The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.
Formats	2-input, 1-output or 4-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLE_L		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource`	23:16	resource_id, 0 to 255.
		`sampler`	27:24	sampler_id, 0 to 15.
		`arguments`	28	indexed_args.
		`aoffimmi`	29	0 = `aoffimmi` does not exist.
				1 = `aoffimmi` exists.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLEINFO, SAMPLEPOS.

Query Information from a Resource

Instructions	**SAMPLEINFO**
Syntax	`sampleinfo_resource(`*n*`)[_`*uint*`]` *dst*, *src0* `sampleinfo_ext_resource(`*n*`)[_resourcetype(pixtexusage)][_`*uint*`]` *dst*, *src0*, *src1*
Description	*dst* receives a number in components x. The returned value is floating point, unless the control field is negative (_uint modifier is used), in which case the returned value is integers. If the resource is not a multi-sample resource and not a render target, the result is 0. The first syntax example above is valid for R670 GPUs and later. The second syntax example above is valid for Evergreen GPUs and later. This second form supports indexing.
Formats	0-input, 1-output. 2-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_SAMPLEINFO		
`control`	29:16	**Field Name**	**Bits**	**Description**
		`resource_id`	23:16	Resource ID
		`return type`	24	0x0: Return type is Float.
				0x1: Return type is Integer.
		reserved	29:25	Must be zero.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEPOS.

Query Position Information from a Resource

Instructions **SAMPLEPOS**

Syntax

`samplepos_resource(n)[_uint] dst, src0`

`samplepos_ext_resource(n)[_resourcetype(pixtexusage)][_uint] dst src0 src1 src2`

Description The returned value is a float4 (x,y,0,0) indicating where the sample is located. If the resource is not a multi-sample resource and not a render target, the result is 0.

The first syntax example above is valid for R670 GPUs and later.

The second syntax example above is valid for Evergreen GPUs and later. This second form supports indexing.

Formats 1-input, 1-output.

3-input, 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_SAMPLEPOS		
control	29:16	**Field Name**	**Bits**	**Description**
		resource_id	23:16	Resource ID
		return type	24	0x0: Return type is Float.
				0x1: Return type is Integer.
		reserved	29:25	Must be zero.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related SAMPLE, SAMPLE_B, SAMPLE_C, SAMPLE_C_B, SAMPLE_C_G, SAMPLE_C_L, SAMPLE_C_LZ, SAMPLE_G, SAMPLE_L, SAMPLEINFO.

Export Data to Memory

Instructions	**`SCATTER`**
Syntax	`scatter_quad src0`
Description	Exports a single value to memory. This instruction views memory as a 3D array, with coordinates x, y, and index. `Src0.xy` contains the first two coordinates of the address. `Src0.z` contains the 3rd coordinate of the address (called the index). `Src0.w` contains the data to be written. Valid for R520 GPUs only.
Format	1 input, 0 output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EMIT
`control`	31:16	If this field is zero, all four pixels in the quad use the same value for the index.

Related None.

Texture LOAD

Instructions **TEXLD**

Syntax `texld_stage(`*n*`)_centroid_shadowmode(`*op*`)_mag(`*op*`)_min(`*op*`)_volmag(`*op*`)_volmin(`*op*`)_mip(`*op*`)_aniso(`*op*`)_lodbias(`*f*`)_xoffset(`*x*`)_yoffset(`*y*`)_zoffset(`*z*`)` *dst*, *src0*

Description Samples a texture specified by *stage* at the coordinates specified by the *src0* register.

This instruction cannot be used on a *stage* that has not been declared with a DCLPT instruction. Also, it cannot be used on a `stage` set to IL_USAGE_PIXTEX_2DMSAA by a DCLPT instruction. The value of `stage` corresponds to the stage/unit.

The coordinates can be projected using the divComp source modifier on *src0.*

- If the divComp source modifier is used, and the component to divide by is negative, the result of this instruction is undefined. That is, if IL_DIVCOMP_Y is used, and the second component of *src0* is negative, the results of this instruction are undefined.

If `centroid` is 1, sampling is done based on the pixel centroid, not center.

The `lodbias` value specifies a constant value to bias the mipmap from which to load for this instruction. This value is added to the bias value set in the state (the value set through `AS_TEX_LODBIAS_N(`*stage*`)`), and the bias value in the fourth component of *src2*.

The following determines the mipmap level(s) from which to sample:

- The *computed LOD* is the mipmap level-of-detail determined based on the ratio of texels in the base texture to the pixel.
- The *instruction LOD* is the value specified by the *lodbias* parameter in the IL_PrimaryTEXLD_Mod token).
- The *minLOD* is the state-based floating point minimum mipmap LOD value.
- The *maxLOD* is the state-based floating point maximum mipmap LOD value.
- The *minLevel* is the smallest mipmap level specified by state to use.
- The *maxLevel* is the largest mipmap level specified by state to use.
- The mipmap level(s) to sample from are determined by:
  - Adding the *state based LOD* to the *computed LOD.*
  - If LOD clamping is enabled in state, clamping the resulting value to *minLOD* and *maxLOD.*
  - Adding the *instruction LOD*.
  - Clamping the resulting value to *minLevel* and *maxLevel.*

The following pseudocode demonstrates the algorithm used to determine the mipmap level(s) to sample from:

- (initial bias LOD) = (state based bias) + (computed LOD)
- (clamped LOD) = (minLOD) ≤ (initial bias LOD) ≤ (maxLOD)
- (secondary bias LOD) = (clamped LOD) + (instruction LOD)
- (final LOD) = (minLevel) ≤ (secondary bias LOD) ≤ (minLevel)

The `mag`, `min`, `volmag`, `volmin`, `mip`, and `aniso` parameters specify whether (and how) to override filter settings. If the IL_PrimaryTEXLD_Mod token is not present, the filters set through external state are used.

Texture LOAD (Cont.)

The values of `xoffset`, `yoffset`, and `zoffset` are added to the unnormalized values of the first, second, and third components of *src0,* respectively, within the sample mipmap. These values are applied whether or not normalized texture coordinates are used. Clamping policy is obeyed as usual when sampling outside the texture's dimensions using these offset parameters. If the IL_SecondaryTEXLD_Mod token is not present, *xoffset, yoffset,* and *zoffset* default to 0.0.

The `shadowMode` parameter specifies if this instruction performs a shadow map load (compare the texture value to the z-component of *src0*). `shadowMode` indicates one of the following:

- A shadow load never occurs.
- A shadow load always occurs.

If a shadow load occurs with this instruction, the `mag`, `min`, `volmag`, `volmin`, `aniso`, `xoffset`, `yoffset`, and `zoffset` parameters are ignored.

Valid for all GPUs.

Operation:

The *src0* provides the texture coordinates for the texture sample. The first eight bits of the control field of the IL_Opcode token specify the texture stage from which to sample. Associated with the stage specified in the control field are: 1) a texture image, 2) the texture's dimension, 3) and all format, filter, wrap, bias, and clamp settings specified in state and through the DCLPT instruction.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_TEXLD
	`stage`	23:16	Stage or unit number, from which to sample.
	`centroid`	24	0: Sample on pixel center. 1: Sample on pixel centroid.
	reserved	25	Must be zero.
	`shadowmode`	27:26	Any value of the enumerated type ILTexShadowMode. See Table 6.28 on page 6-21.
	reserved	29:28	Must be zero.
	`sec_modifier_present`	30	0: No secondary modifier token is present. 1: IL_SecondaryTEXLD_Mod token immediately follows this token or an IL_PRIMARYTEXLD_Mod token, if bit 31 is set.
	`pri_modifier_present`	31	0: No primary modifier token is present. 1: IL_PrimaryTEXLD_Mod token immediately follows this token.
2	Primary Texture Load Instruction Modifier token (is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token).		
	`mag`	2:0	Specifies how to filter texture values in the S and T directions when a single texel maps to multiple pixels. Can be any value of the enumerated type ILTexFilterMode. See Table 6.27 on page 6-20.
	`Min`	5:3	Specifies how to filter texture values in the S and T directions when multiple texels map to a single pixel. Can be any value of the enumerated type ILTexFilterMode.
	`volmag`	8:6	Specifies how to filter texture values in the R direction when the pixel maps to an area less than one texel. Can be any value of the enumerated type ILTexFilterMode.

Texture LOAD (Cont.)

	`volmin`	11:9	Specifies how to filter texture values in the R direction when the pixel maps to an area greater than one texel. Can be any value of the enumerated type ILTexFilterMode.
	`mip`	14:12	Specifies how to filter values of multiple mipmaps when multiple texels of the base level maps a single pixel. Can be any value of the enumerated type ILMipFilterMode.
	`aniso`	17:15	When anisotropic filtering is enabled in the min or mag filter, specifies the maximum number of samples to use for anisotropic filtering. When anisotropic filtering is used for the min and mag filter, this value is the same. Can be any value of the enumerated type ILAnisoFilterMode. See Table 6.2 on page 6-1.
	`lodbias`	24:18	Specifies a constant mipmap level of detail bias applied to the texture load. Signed fixed point with 4 bits of fraction (3.4), which allows a bias range from [–4, 3.9375]. This value is added to the computed texture LOD value.
	reserved	31:25	Must be zero.
3	Secondary Texture Load Instruction Modifier token (is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token).		
	`Xoffset`	7:0	Unnormalized (texel space) values added to the X texture coordinate before sampling. Signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5].
	`Yoffset`	15:8	Unnormalized (texel space) values added to the Y texture coordinate before sampling. Signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5].
	`Zoffset`	23:16	Unnormalized (texel space) values added to the Z texture coordinate before sampling. Signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5].
	reserved	31:24	Must be zero.
4	IL_Dst token (*dst*)		
5	IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.		
6	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		
7	IL_Src token (*src0*)		
8	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.		
9	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		

Related TEXLDB, TEXLDD, TEXLDMS, TEXWEIGHT.

Biased Texture LOAD

Instructions **TEXLDB**

Syntax

```
texldb_stage(n)[_centroid][_absolute]_shadowmode(op)_mag(op)_min(op)_volmag(op)_
volmin(op)_mip(op)_aniso(op)_lodbias(f)_qualitybias_xoffset(x)_yoffset(y)_zoffse
t(z) dst, src0, src1
```

Description Samples a texture specified by `stage` at coordinates specified by the *src0* register and biased by the value in the fourth component of *src1.*

This instruction cannot be used on a `stage` that has not been declared with a DCLPT instruction. Also, it cannot be used on a *stage* set to IL_USAGE_PIXTEX_2DMSAA by a DCLPT instruction.

The value of `stage` corresponds to the stage/unit defined externally.

The fourth component of *src2* (*src2*.w) is a factor in determining the mipmap level from which to sample.

The coordinates can be projected using the divComp source modifier on *src0*.

- If the divComp source modifier is used, and the component to divide by is negative, the result of this instruction is undefined. That is, if IL_DIVCOMP_Y is used, and the second (y) component of *src0* is negative, the result of this instruction is undefined.
- divComp can be set to IL_DIVCOMP_UNKNOWN in this instruction; in this case the component used to divide is specified externally.

If *centroid* is 1, sampling is done based on the pixel centroid, not center.

The *lodbias* value specifies a constant value to bias the mipmap from which to load for this instruction. This value is added to the bias value set in the in state and the bias value in the fourth component (w) of *src2*.

The following is used to determine the mipmap level(s) to sample from:

- The *computed LOD* is the mipmap level of detail determined based on the ratio of texels in the base texture to the pixel.
- The *instruction LOD* is the value specified by the `lodbias` parameter in the IL_PrimaryTEXLDB_Mod token)
- The `state based bias` is the texture bias value specified externally.
- The *minLOD* is the state based floating point minimum mipmap LOD value.
- The *maxLOD* is the state based floating point maximum mipmap LOD value.
- The *minLevel* is the smallest mipmap level specified by state to use.
- The *maxLevel* is the largest mipmap level specified by state to use.
- When *absolute* is 0, the computed LOD is a factor in determining the mipmap level(s) to sample from. The mipmap level(s) to sample from are determined by:
  - Adding the fourth component (w) of *src2*, the *state based LOD*, and the *computed LOD*
  - If LOD clamping is enabled in state, clamping the resulting value to *minLOD* and *maxLOD.*
  - Adding the *instruction LOD*.
  - Clamping the resulting value to *minLevel* and *maxLevel.*

The following pseudo code demonstrates the algorithm used to determine the mipmap level(s) to sample from:

```
(initial bias LOD) = (fourth component of src1) + (state based bias) + (computed
LOD)
(clamped LOD) = (minLOD) ≤ (initial bias LOD) ≤ (maxLOD)
(secondary bias LOD) = (clamped LOD) + (instruction LOD)
(final LOD) = (minLevel) ≤ (secondary bias LOD) ≤ (minLevel)
```

Biased Texture LOAD (Cont.)

- When *absolute* is 1, the computed LOD is not a factor in determining the mipmap level(s) from which to sample. These level(s) are determined by:
  - Adding the fourth component of *src2* and the *state based LOD*.
  - If LOD clamping is enabled in state, clamping the resulting value to *minLOD* and *maxLOD.*
  - Adding the *instruction LOD*.
  - Clamping the resulting value to *minLevel* and *maxLevel.*

The following pseudo code demonstrates the algorithm used to determine the mipmap level(s) to sample from:

```
(initial bias LOD) = (fourth component of src2) + (state based bias)
(clamped LOD) = (minLOD) ≤ (initial bias LOD) ≤ (maxLOD)
(secondary bias LOD) = (clamped LOD) + (instruction LOD)
(final LOD) = (minLevel) ≤ (secondary bias LOD) ≤ (minLevel)
```

The `mag`, `min`, `volmag`, `volmin`, `mip`, and `aniso` parameters specify whether (and how) to override external filter settings. If the IL_PrimaryTEXLD_Mod token is not present, the filters are set externally.

The values of `xoffset`, `yoffset`, and `zoffset` are added to the unnormalized values of the first, second, and third components of *src0*, respectively, within the sample mipmap. These values are applied whether or not normalized texture coordinates are used. Clamping policy is obeyed when sampling outside the texture's dimensions using these offset parameters. If the IL_SecondaryTEXLD_Mod token is not present, `xoffset`, `yoffset`, and `zoffset` default to 0.0.

The `shadowmode` parameter specifies if this instruction performs a shadow map load, (compare the texture value to the z-component of *src0)*. shadowMode indicates if a shadow load never occurs or always occurs.

See shadow texture load appendix for texture load algorithm.

If a shadow load occurs with this instruction, the *mag*, *min, volmag, volmin, aniso, xoffset, yoffset,* and *zoffset* parameters are ignored.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_TEXLDB
	`stage`	23:16	Stage or unit number.
	`centroid`	24	0: Sample on pixel center. 1: Sample on pixel centroid.
	`absolute`	25	0: the fourth component of *src2* is a relative mipmap. 1: the fourth component of *src2* is an absolute mipmap.
	`shadowmode`	27:26	Any value of the enumerated type ILTexShadowMode. See Table 6.28 on page 6-21.
	reserved	29:28	Must be zero.
	`sec_modifier_present`	30	0: No secondary modifier token is present. 1: IL_SecondaryTEXLD_Mod token immediately follows the IL_OP_TEXLDB token or an IL_PrimaryTEXLDB_Mod token if bit 31 is set.
	`pri_modifier_present`	31	0: No primary modifier token is present. 1: IL_PrimaryTEXLD_Mod token immediately follows the IL_OP_TEXLDB token.

Biased Texture LOAD (Cont.)

2 Primary Texture Load Instruction Modifier token (is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token).

`Mag`	2:0	Specifies how to filter texture values in the S and T directions when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
`Min`	5:3	Specifies how to filter texture values in the S and T directions when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
`volmag`	8:6	Specifies how to filter texture values in the R direction when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
`Volmin`	11:9	Specifies how to filter texture values in the R direction when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
`Mip`	14:12	Specifies how to filter values of multiple mipmaps when the pixel maps to an area greater than one texel of the base map. Can be any value of the enumerated type ILMipFilterMode.
`Aniso`	17:15	When anisotropic filtering is enabled in the min or mag filter, specifies the maximum number of samples to use for anisotropic filtering. When anisotropic filtering is used for the min and mag filter, this value is the same. Can be any value of the enumerated type ILAnisoFilterMode.
`lodbias`	24:18	Specifies a constant mipmap level of detail bias applied to the texture load. Signed fixed point with 4 bits of fraction (3.4), which allows a bias range from [–4, 3.9375].
`qualitybias`	25	Specifies if the bias is per pixels or per quad: 0 = Per quad. 1 = Per pixel.
reserved	31:26	Must be zero.

3 Secondary Texture Load Instruction Modifier token (is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token).

`Xoffset`	7:0	Unnormalized (texel space) values added to the X texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
`Yoffset`	15:8	Unnormalized (texel space) values added to the Y texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
`Zoffset`	23:16	Unnormalized (texel space) values added to the Z texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
reserved	31:24	Must be zero.

Biased Texture LOAD (Cont.)

4	IL_Dst token (*dst*)
5	IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.
6	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.
7	IL_Src token (*src1*)
8	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.
9	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.
10	IL_Src token (*src*2)
11	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.
12	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

Related TEXLD, TEXLDD, TEXLDMS, TEXWEIGHT.

Gradient Texture LOAD

Instructions **TEXLDD**

Syntax `texldd_stage(`*n*`)_centroid_shadowmode(`*op*`)_mag(`*op*`)_min(`*op*`)_volmag(`*op*`)_volmin(`*op*`)_mip(`*op*`)_aniso(`*op*`)_lodbias(`*f*`)_xoffset(`*x*`)_yoffset(`*y*`)_zoffset(`*z*`)` *dst*, *src0*, *src1*, *src2*

Description *src0* contains the texture coordinate to sample from the specified texture stage.

This instruction cannot be used on a *stage* set to IL_USAGE_PIXTEX_2DMSAA by a DCLPT instruction. Also, it cannot be used on a *stage* that has not been declared with a DCLPT instruction.

The value of *stage* corresponds to the stage/unit defined in state.

The coordinates can be projected using the divComp source modifier on *src0*.

- If the divComp source modifier is used, and the component to divide by is negative, the result of this instruction is undefined. That is, if IL_DIVCOMP_Y is used, and the second component of *src0* is negative, the result of this instruction is undefined.
- divComp can be set to IL_DIVCOMP_UNKNOWN in this instruction; in this case, the component used to divide is specified externally.

If *centroid* is set to 1, sampling is done based on the pixel centroid, not center.

The *lodbias* value specifies a constant value to bias the mipmap from which to load for this instruction. This value is added to the bias value set in the in state and the bias value in the fourth component of *src2*.

The following is used to determine the mipmap level(s) from which to sample.

- The *computed LOD* is the mipmap level of detail determined based on the ratio of texels in the base texture to the pixel.
- The *instruction LOD* is the value specified by the *lodbias* parameter in the IL_PrimaryTEXLD_Mod token)
- The *state based bias* is the texture bias value specified externally.
- The *minLOD* is the state based floating point minimum mipmap LOD value.
- The *maxLOD* is the state based floating point maximum mipmap LOD value.
- The *minLevel* is the smallest mipmap level specified by state to use.
- The *maxLevel* is the largest mipmap level specified by state to use.
- The mipmap level(s) to sample from are determined by:
  – Adding the *state based LOD* to the *computed LOD.*
  – If LOD clamping is enabled in state, clamping the resulting value to *minLOD* and *maxLOD.*
  – Adding the *instruction LOD*.
  – Clamping the resulting value to *minLevel* and *maxLevel.*

The following pseudo code demonstrates the algorithm used to determine the mipmap level(s) to sample from:

- (initial bias LOD) = (state based bias) + (computed LOD)
- (clamped LOD) = (minLOD) ≤ (initial bias LOD) ≤ (maxLOD)
- (secondary bias LOD) = (clamped LOD) + (instruction LOD)
- (final LOD) = (minLevel) ≤ (secondary bias LOD) ≤ (minLevel)

The *mag*, *min, volmag, volmin, mip,* and *aniso* parameters specify whether (and how) to override filter settings. If the IL_PrimaryTEXLD_Mod token is not present, the external filters settings are used.

Gradient Texture LOAD (Cont.)

The values of *xoffset, yoffset,* and *zoffset* are added to the unnormalized values of the first, second, and third components of *src0*, respectively, within the sample mipmap. These values are applied whether or not normalized texture coordinates are used. Clamping policy is obeyed when sampling outside the texture's dimensions using these offset parameters. If the IL_SecondaryTEXLD_Mod token is not present, *xoffset, yoffset,* and *zoffset* default to 0.0.

The *shadowmode* parameter specifies if this instruction performs a shadow map load (compare the texture value to the z-component of *src0*). shadowMode indicates if a shadow load never occurs or always occurs.

If a shadow load occurs with this instruction, the *mag*, *min, volmag, volmin, aniso, xoffset, yoffset,* and *zoffset* parameters are ignored.

Valid for all GPUs.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_TEXLDD
	`stage`	23:16	Stage or unit number, 0 to 255.
	`centroid`	24	0: Sample on pixel center.
			1: Sample on pixel centroid.
	reserved	25	Must be zero.
	`shadowmode`	27:26	Any value of the enumerated type ILTexShadowMode. See Table 6.28 on page 6-21.
	reserved	29:28	Must be zero.
	`sec_modifier_present`	30	0: No secondary modifier token is present.
			1: IL_SecondaryTEXLD_Mod token is present.
	`pri_modifier_present`	31	0: No primary modifier token is present.
			1: IL_PrimaryTEXLD_Mod token is present.
2	Primary Texture Load Instruction Modifier token (is present only if the `pri_modifier_present` field is 1 in the previous IL_Opcode token).		
	`Mag`	2:0	Specifies how to filter texture values in the S and T directions when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
	`Min`	5:3	Specifies how to filter texture values in the S and T directions when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
	`volmag`	8:6	Specifies how to filter texture values in the R direction when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
	`Volmin`	11:9	Specifies how to filter texture values in the R direction when the pixel maps to an area than one texel. Can be any value of the enumerated type ILTexFilterMode.
	`Mip`	14:12	Specifies how to filter values of multiple mipmaps when the pixel maps to an area greater than one texel of the base map. Can be any value of the enumerated type ILMipFilterMode.

Gradient Texture LOAD (Cont.)

`Aniso`	17:15	When anisotropic filtering is enabled in the min or mag filter, specifies the maximum number of samples to use for anisotropic filtering. When anisotropic filtering is used for the min and mag filter, this value is the same. This can be any value of the enumerated type ILAnisoFilterMode.
`lodbias`	24:18	Specifies a constant mipmap level of detail bias applied to the texture load. Signed fixed point with 4 bits of fraction (3.4), which allows a bias range from [–4, 3.9375]. This value is added to the computed texture LOD value.
reserved	31:25	Must be zero.

3 Secondary Texture Load Instruction Modifier token (is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token.)

`Xoffset`	7:0	Unnormalized (texel space) values added to the X texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
`Yoffset`	15:8	Unnormalized (texel space) values added to the Y texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
`Zoffset`	23:16	Unnormalized (texel space) values added to the Z texture coordinate before sampling. Signed fixed point with 1 bit of fraction (7.1) which allows ranges from [–64, 63.5].
reserved	31:24	Must be zero.

4 IL_Dst token (*dst*)

5 IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.

6 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

7 IL_Src token (*src1*)

8 IL_Src_Mod token is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token.

9 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

10 IL_Src token (*src*2)

11 IL_Src_Mod token is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token.

12 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

13 IL_Src token (*src3*)

14 IL_Src_Mod token is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token.

15 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

Related TEXLD, TEXLDB, TEXLDMS, TEXWEIGHT.

Multisample Texture LOAD

Instructions **TEXLDMS**

Syntax `texld_stage(`*n*`)_mag(`*op*`)_min(`*op*`)_aniso(`*op*`)_sample(`*n*`)_xoffset(`*x*`)_yoffset(`*y*`)` *dst, src0*

Description Samples a multi-sample texture specified by *stage* at coordinates specified by the *src0.*

This instruction cannot be used on a *stage* that has not been declared with a DCLPT instruction.

This instruction can only be executed on a stage where the *usage* of the DCLPT instruction is set to IL_USAGE_PIXTEX_2DMSAA.

The value of *stage* corresponds to the stage/unit defined in state.

The first two components of *src0* (*src0*.xy) are used as the texture coordinate to sample from the specified texture stage.

The sum of the third component of *src0*, *src0*.z, and the value specified in the *sample* parameter specifies the single sample to retrieve from the multi-sample buffer when the state based multi-sample filter is set to point filtering.

- The sum of the third component of *src0*, *src0*.z, and the value specified in the *sample* parameter specifies the single sample to retrieve when AS_MSAA_TEX_FILTER_FUNCTION_N(*stage*) is set to POINT.

The coordinates are treated as unmodified regardless of how thye are declared.

- If the divComp source modifier is used, and the component to divide by is negative, the results of this instruction are undefined. For example, if IL_DIVCOMP_Y is used and the second component of *src0* is negative, the result of this instruction is undefined.
- divComp can be set to IL_DIVCOMP_UNKNOWN in this instruction in which case the component used to divide is specified externally.

The values of *xoffset* and *yoffset* are added to the unnormalized values of the first, second, and third components of *src0* respectively within the sample mipmap. Clamping policy is obeyed when sampling outside the texture's dimensions using these offset parameters. If the IL_SecondaryTEXLDMS_Mod token is not present, *xoffset* and *yoffset* default to 0.0

Operation:

src0 provides the texture coordinates for the texture sample. The first eight bits of the control field of the IL_Opcode token specify the texture stage from which to sample.

Valid for R6XX GPUs and later.

Format

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_TEXLDMS
	`stage`	23:16	Stage or unit number.
	reserved	29:24	Must be zero.
	`sec_modifier_present`	30	0: No secondary modifier token is present. 1: IL_SecondaryTEXLD_Mod token immediately follows the IL_OP_TEXLDMS token or an IL_PrimaryTEXLDMS_Mod token if bit 31 is set.
	`pri_modifier_present`	31	0: No primary modifier token is present. 1: IL_PrimaryTEXLD_Mod token immediately follows the IL_OP_TEXLDMS token.

Multisample Texture LOAD (Cont.)

All fields in the primary modifier are ignored.

3 Secondary Multi-sample Texture Load Instruction Modifier token (is present only if the `sec_modifier_present` field is 1 in the previous IL_Opcode token).

`Xoffset`	7:0	Unnormalized (texel space) values added to the X texture coordinate before sampling. Signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5].
`Xoffset`	15:8	Unnormalized (texel space) values added to the Y texture coordinate before sampling. Signed fixed point with one bit of fraction (7.1) that allows ranges from [–64, 63.5].
reserved	2816	Must be zero.
reserved	31:29	Must be zero.

4 IL_Dst token (*dst*)

5 IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.

6 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

7 IL_Src token (*src0*)

8 IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.

9 IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.

Related TEXLD, TEXLDB, TEXLDD, TEXWEIGHT.

Get Texel Weight

Instructions	**TEXWEIGHT**
Syntax	`texweight_stage(`*n*`)` *dst, src0*

Description

Retrieves the weights used by a bilinear filtered fetch based upon the texture coordinate provided in *src0.*

The *dst*.x represents the horizontal LERP factor; the *dst*.y represents the vertical LERP factor.

The *dst*.z and *dst*.w component are not written to by this instruction; however, if the *component_z_b* or *component_w_a* field of the IL_Dst_Mod token is set to IL_MODCOMP_0 or IL_MODCOMP_1, the *dst*.z and *dst*.w are written (*dst*.z and *dst*.w can be set to 0.0 or 1.0 if IL_MODCOMP_0 or IL_MODCOMP_1 is used on the *component_z_b* or *component_w_a* field of the IL_Dst_Mod token).

The coordinates can be projected using the divComp source modifier on *src0.*

- If the divComp source modifier is used, and the component to divide by is negative, the result of this instruction is undefined. That is, if IL_DIVCOMP_Y is used, and the second component of *src0* is negative, the results of this texture load is undefined.
- divComp can be set to IL_DIVCOMP_UNKNOWN in this instruction; in this case, the component used to divide is specified in state.

The value of *stage* corresponds to the stage/unit.

This instruction can be used on only a *stage* that has been declared with a DCLPT instruction.

Valid for all GPUs.

Operation:

src0 provides the texture coordinates for the texture weight. The first eight bits of the control field of the IL_Opcode token specify the texture stage from which to retrieve the weight. Associated with the stage specified in the control field are 1) a texture, 2) the texture's dimension, 3) and all format, filter, wrap, bias, and clamp states defined externally and through the DCLPT instruction.

ALU instructions perform arithmetic and relational operations. All take in at least one source operand and output to 1 destination operand.

Formats

1-input, 1-output.

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_TEXWEIGHT
	`stage`	23:16	Stage or unit number, 0 to 255.
	reserved	31:24	Must be zero.
2	IL_Dst token (*dst*)		
3	IL_Dst_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Dst token.		
4	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		
5	IL_Src token (*src0*)		
6	IL_Src_Mod token is present only if the `modifier_present` field is 1 in the previous IL_Src token.		
7	IL_Rel_Addr token is present only if the `relative_address` field is 1 in the preceding IL_Src or IL_Dst token.		

Related TEXLD, TEXLDB, TEXLDD, TEXLDMS.

7.7 Integer Arithmetic Instructions

64-Bit Bitwise Integer Addition

Instructions **`I64_ADD`**

Syntax **`i64add`** ***`dst, src0, src1`***

Description Component-wise add of 64-bit operands *src0* and *src1*. No carry or borrow is done beyond the 32-bit values of each component; thus, this instruction is not sensitive to the sign of the operand.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I64_ADD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

64-Bit Integer Compare

Instructions **I64EQ, I64GE, I64LT, I64NE**

Syntax

Function	Opcode	Syntax	Description
==	`IL_OP_I64_EQ`	`i64eq` *dst*, *src0*, *src1*	Compares if integer vector ins scr0 is equal to the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
≥	`IL_OP_I64_GE`	`i64ge` *dst*, *src0*, *src1*	Compares if integer vector in *src0* is greater or equal to the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
<	`IL_OP_I64_LT`	`i64lt` *dst*, *src0*, *src1*	Compares if integer vector in src0 is less than the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
≠	`IL_OP_I64_NE`	`i64ne` *dst*, *src0*, *src1*	Compares two integer vectors, one in *src0*, the other in *src1*, to check if they are not equal. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.

Description Component-wise compares two vectors using an integer comparison. If *src0*.{xy|zw} and the corresponding component in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE; otherwise, it is set to FALSE. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the control field, `pri_modifier_present`, and `sec_modifier_present` fields must be zero.

All these instructions are valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
`control`	31:16	Must be zero.

Related None.

64-Bit Integer Maximum, Integer Minimum

Instructions **`I64MAX, I64MIN`**

Syntax

Function	Opcode	Syntax	Description
I64MAX	`IL_OP_I64_MAX`	`i64max` *`dst, src0, src1`*	*dst* = *src0* > *src1* ? *src0*:*src1*
64IMIN	`IL_OP_I64_MIN`	`i64min` *`dst, src0, src1`*	*dst* = *src0* < *src1* ? *src0*:*src1*

Description Component-wise integer maximum (I64_MAX) and minimum (I64_MIN).

Compares each component of *src0* with the corresponding component of *src1*. If the comparison evaluates true, *src0* is returned in the corresponding component of *dst*; otherwise, *src1* is returned.

For the MAX and MIN instructions, if *src0*.{x|y|z|w} or *src1*.{x|y|z|w} is NaN, then the other component, *src1*.{x|y|z|w} or *src0*.{x|y|z|w}, respectively, is returned. Denorms are flushed before the comparison is performed. If a flushed denorm is the maximum value (for the MAX instruction) or minimum value (for the MIN instruction), then the flushed denorm is returned. The MAX instruction uses a greater-than-or-equal-to comparison. Thus, if min(*src0*, *src1*) = *src0*, then max(*src0*, *src1*) = *src1*, including cases with +0 and -0, such as when denorms are flushed to sign preserve zero.

If the _ieee flag is included, then MIN and MAX follow IEEE-754r rules for minnum and maxnum (except denorms are flushed before the comparison is made). Also, MIN returns -0, and MAX returns +0, for comparisons between -0 and +0.

Both instructions are valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I64_MAX, IL_OP_I64_MIN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

64-Bit Two's Complement Negate

Instructions	**`I64NEGATE`**
Syntax	`i64negate` *`dst`*`,` *`src0`*
Description	Computes the two's complement negation of each component in *src0.* The 64-bit results are placed in the corresponding components of *dst*. Valid for Evergreen GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I64_NEGATE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

64-Bit Integer Shift Left, Integer Shift Right

Instructions **I64SHL, I64SHR**

Function	Opcode	Syntax	Description
I64SHL	`IL_OP_I64_SHL`	`i64shl` *dst, src0, src1*	*src1* must be a scalar value; thus, all four swizzles on *src1* must be the same. Component-wise shift of each 64-bit value in *src0* left by an unsigned integer bit count provided by the lower five bits (0-63 range) in *src1*, inserting 0. Shifts of 65 and 1 are treated identically. The 64-bit per component results are placed in *dst*. dst = s(src0) x $2^{src1\ \&\ 0x3F}$ The count is a scalar value applied to all components. Produces the same value as USHR if src0 is non-negative, or if the lower five bits of the shift (src1) are zero. Valid for Evergreen GPUs and later.
I64SHR	`IL_OP_I64_SHR`	`i64shr` *dst, src0, src1*	Component-wise arithmetic shift of each 64-bit value in *src0* right by an unsigned integer bit count provided by the lower five bits (0-63 range) in *src1*, replicating the value of bit 63. Shifts of 65 and 1 are treated identically. The 64-bit per component result is placed in *dst*. dst = $\lfloor s(src0)/2^{src1\ \&\ 0x3F} \rfloor$ The count is a scalar value applied to all components. Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Bitwise Integer Addition

Instructions **`IADD`**

Syntax **`iadd`** ***`dst, src0, src1`***

Description Component-wise add of 32-bit operands *src0* and *src1*. The 32-bit result is placed in *dst*. No carry or borrow is done beyond the 32-bit values of each component; thus, this instruction is not sensitive to the sign of the operand.

Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_ADD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer AND

Instructions **`IAND`**

Syntax `iand` *`dst`*, *`src0`*, *`src1`*

Description Component-wise logical AND of each pair of 32-bit values from *src0* and *src1*. The 32-bit result is placed in *dst*.

Valid for R600 GPUs and later.

Format 2-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_AND
`control`	31:16	Must be zero.

Related None.

Integer Subtract-Borrow

Instructions **IBORROW**

Syntax `iborrow dst, src0, src1`

Description Forms the borrow bit after subtracting two integer vectors.

First `scr1` is subtracted from `src0`. Then, `dst` is set to 1 if a borrow is produced; otherwise, a 0.

Operates per component.

The neg modifier can be used on any of the inputs to this instruction.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_BORROW
`control`	31:16	Must be zero.

Related None.

Integer Add-Carry

Instructions **ICARRY**

Syntax `icarry dst, src0, src1`

Description Forms the carry bit after adding two integer vectors.

`If (src0 + src1 > 0xFFFFFFFF { dst = 1;} else {dst = 0}`

Operates per component.

The neg modifier can be used on any of the inputs to this instruction.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_CARRY
`control`	31:16	Must be zero.

Related None.

Integer Compare

Instructions **IEQ, IGE, ILT, INE**

Syntax

Function	Opcode	Syntax	Description
==	IL_OP_I_EQ	ieq *dst*, *src0*, *src1*	Compares if integer vector ins scr0 is equal to the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
≥	IL_OP_I_GE	ige *dst*, *src0*, *src1*	Compares if integer vector in *src0* is greater or equal to the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
<	IL_OP_I_LT	ilt *dst*, *src0*, *src1*	Compares if integer vector in src0 is less than the one in *src1*. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.
≠	IL_OP_I_NE	ine *dst*, *src0*, *src1*	Compares two integer vectors, one in *src0*, the other in *src1*, to check if they are not equal. If TRUE, 0xFFFFFFFF is returned for that component; otherwise, 0x00000000 is returned.

Description Component-wise compares two vectors using an integer comparison. If *src0*.{x|y|z|w} and the corresponding component in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE; otherwise, it is set to FALSE. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the control field, `pri_modifier_present`, and `sec_modifier_present` fields must be zero.

All these instructions are valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	See *Syntax*, above.
control	31:16	Must be zero.

Related None.

Signed Integer Multiply and Add

Instructions	**`IMAD`**
Syntax	`imad` *`dst, src0, src1, src2`*
Description	Component-wise integer multiply of 32-bit signed operands *src0* and *src1*, keeping the low 32 bits (per component) of the result, followed by an integer add of *src2* to produce the low 32-bit (per component) result. and placing it in *dst*. Valid for R600 GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MAD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

24-Bit Signed Integer Multiply and Add

Instructions	**`IMAD24`**
Syntax	`imad24` *`dst, src0, src1, src2`*
Description	Component-wise integer multiply of 24-bit signed operands *src0* and *src1*, keeping the low 32 bits (per component) of the result, followed by an integer add of *src2* to produce the low 32-bit (per component) result. and placing it in *dst*. Operands *scr0* and *src1* are treated as 24-bit signed integers; bits [31:24] are ignored. The result represents the low-order 32-bits of the multiply-add. *dst* = *src0*[23:0] * *src1*[23:0] + *src2*[31:0] Valid for Northern Islands GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MAD24
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer Maximum, Integer Minimum

Instructions **`IMAX, IMIN`**

Syntax

Function	Opcode	Syntax	Description
IMAX	`IL_OP_I_MAX`	`imax` *`dst`*`,` *`src0`*`,` *`src1`*	*dst* = *src0* > *src1* ? *src0*:*src1*
IMIN	`IL_OP_I_MIN`	`imin` *`dst`*`,` *`src0`*`,` *`src1`*	*dst* = *src0* < *src1* ? *src0*:*src1*

Description

Component-wise integer maximum (I_MAX) and minimum (I_MIN).

Compares each component of *src0* with the corresponding component of *src1*. If the comparison evaluates true, *src0* is returned in the corresponding component of *dst*; otherwise, *src1* is returned.

For the MAX and MIN instructions, if *src0*.{x|y|z|w} or *src1*.{x|y|z|w} is NaN, then the other component, *src1*.{x|y|z|w} or *src0*.{x|y|z|w}, respectively, is returned. Denorms are flushed before the comparison is performed. If a flushed denorm is the maximum value (for the MAX instruction) or minimum value (for the MIN instruction), then the flushed denorm is returned. The MAX instruction uses a greater-than-or-equal-to comparison. Thus, if min(*src0*, *src1*) = *src0*, then max(*src0*, *src1*) = *src1*, including cases with +0 and -0, such as when denorms are flushed to sign preserve zero.

If the _ieee flag is included, then MIN and MAX follow IEEE-754r rules for minnum and maxnum (except denorms are flushed before the comparison is made). Also, MIN returns -0, and MAX returns +0, for comparisons between -0 and +0.

Both instructions are valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MAX, IL_OP_I_MIN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Signed Integer Multiplication (Low 32 Bits)

Instructions **`IMUL`**

Syntax `imul` *`dst, src0, src1`*

Description Component-wise multiply of 32-bit operands *src0* and *src1*. The lower 32 bits of the 64-bit result, which is placed in *dst*.

Valid for R600 GPUs and later.s

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MUL
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related IMUL_HIGH.

Signed Integer Multiplication (High 32 Bits)

Instructions **`IMUL_HIGH`**

Syntax `imul_high` *`dst, src0, src1`*

Description Component-wise multiply of 32-bit signed operands *src0* and *src1*. The upper 32 bits of the 64-bit result is placed in the corresponding component of *dst*.

Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MUL_HIGH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related IMUL.

24-Bit Signed Integer Multiply

Instructions **`IMUL24`**

Syntax `imul24` *`dst, src0, src1`*

Description Component-wise integer multiply of 24-bit signed operands *src0* and *src1*, keeping the low 32 bits (per component) of the result and placing it in *dst*. Operands *scr0* and *src1* are treated as 24-bit signed integers; bits [31:24] are ignored. The result represents the low-order 32-bits of the multiply.

dst = *src0*[23:0] * *src1*[23:0]

Valid for Northern Islands GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MAD24
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

High-Order 24-Bit Signed Integer Multiply

Instructions **`IMUL24_HIGH`**

Syntax `imul24_high` *`dst, src0, src1`*

Description Component-wise integer multiply of 24-bit signed operands *src0* and *src1*, keeping the low 32 bits (per component) of the result and placing it in *dst*. Operands *scr0* and *src1* are treated as 24-bit signed integers; bits [31:24] are ignored. The result represents the high-order 16-bits of the 48-bit multiply.

dst = *src0*[23:0] * *src1*[23:0]

Valid for Northern Islands GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_MAD24_HIGH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Two's Complement Negate

Instructions	**`INEGATE`**
Syntax	`inegate` *`dst, src0`*
Description	Computes the two's complement negation of each component in *src0.* The 32-bit results are placed in the corresponding components of *dst*. Valid for R600 GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_OP_I_NEGATE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer Negate

Instructions	**`INOT`**
Syntax	`inot` *`dst, src0`*
Description	Performs a bit-wise one's complement on each component of *src0*. The 32-bit results are placed in the corresponding components of *dst*. Valid for R600 GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_OP_I_NOT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer inclusive-OR, Integer Exclusive-OR

Instructions **IOR, IXOR**

Syntax

Function	Opcode	Syntax	Description
IOR	IL_OP_I_OR	ior *dst*, src0, *src1*	Integer inclusive-OR, bit-wise OR of each component in *src0* with the corresponding component in *src1*
IXOR	IL_OP_I_XOR	ixor *dst*, *src0*, *src1*	Integer exclusive-OR, bit-wise XOR of each component in *src0* with the corresponding component in *src1*

Description Performs a bit-wise logical operation of each component of *src0* with the corresponding component of *src1*. The 32-bit results are placed in the corresponding components of *dst*.

These instructions are valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	See *Syntax*, above.
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Integer Shift Left, Integer Shift Right

Instructions **ISHL, ISHR**

Syntax

Function	Opcode	Syntax	Description
ISHL	`IL_OP_I_SHL`	`ishl` *`dst`*`,` *`src0`*`,` *`src1`*	*src1* must be a scalar value; thus, all four swizzles on *src1* must be the same. Component-wise shift of each 32-bit value in *src0* left by an unsigned integer bit count provided by the lower five bits (0-31 range) in *src1*, inserting 0. Shifts of 33 and 1 are treated identically. The 32-bit per component results are placed in *dst*. dst = s(src0) x $2^{\text{src1 \& 0x1F}}$ The count is a scalar value applied to all components. Produces the same value as USHR if src0 is non-negative, or if the lower five bits of the shift (src1) are zero. Valid for R600 GPUs and later.
ISHR	`IL_OP_I_SHR`	`ishr` *`dst`*`,` *`src0`*`,` *`src1`*	Component-wise arithmetic shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the lower five bits (0-31 range) in *src1*, replicating the value of bit 31. Shifts of 33 and 1 are treated identically. The 32-bit per component result is placed in *dst*. dst = $\lfloor \text{s(src0)}/2^{\text{src1 \& 0x1F}} \rfloor$ The count is a scalar value applied to all components. Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

7.8 Unsigned Integer Operations

Unsigned Integer Maximum, Unsigned Integer Minimum

Instructions **`U64MAX, U64MIN`**

Syntax

Function	Opcode	Syntax	Description
U64MAX	`IL_OP_U64_MAX`	`u64max` *`dst, src0, src1`*	unsigned integer maximum, *dst* = *src0* > *src1* ? *src0*:*src1*
U64MIN	`IL_OP_U64_MIN`	`u64min` *`dst, src0, src1`*	unsigned integer minimum, *dst* = *src0* < *src1* ? *src0*:*src1*

Description Compares each component of *src0* with the corresponding component of *src1*. If the comparison is TRUE, *src0* is returned in the corresponding component of *dst*; otherwise, *src1* is returned.

For the MAX and MIN instructions, if *src0*.{xy|zw} or *src1*.{xy|zw} is NaN, then the other component, *src1*.{xy|zw} or *src0*.{xy|zw}, respectively, is returned. Denorms are flushed before the comparison is performed. If a flushed denorm is the maximum value (for the MAX instruction) or minimum value (for the MIN instruction), then the flushed denorm is returned. The MAX instruction uses a greater-than-or-equal-to comparison. Thus, if min(*src0*, *src1*) = *src0*, then max(*src0*, *src1*) = *src1*, including cases with +0 and -0 such as when denorms are flushed to sign preserve zero.

If the `_ieee` flag is included, MIN and MAX follow IEEE-754r rules for minnum and maxnum (except denorms are flushed before the comparison is made). In addition, MIN returns -0 and MAX returns +0 for comparisons between -0 and +0.

Both instructions are valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Floating Point Division, Unsigned Integer Division

Instructions	**`UDIV`**
Syntax	`udiv` *`dst, src0, src1`*
Description	Component-wise unsigned division of the 32-bit operand in *src0* by the corresponding component in *src1*. The 32-bit quotient is placed in the corresponding component of *dst*. Valid for R600 GPUs and later.
Format	2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_DIV
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

64-Bit Unsigned Integer Compare

Instructions **`U64GE, U64LT`**

Syntax

Function	Opcode	Syntax
≥	`IL_OP_U64_GE`	`u64ge` *`dst, src0, src1`*
<	`IL_OP_U64_LT`	`u64lt` *`dst, src0, src1`*

Description Component-wise compares two vectors using an unsigned integer comparison. For UGE: (*src0* ≥ *src1*); for ULT: (*src0* < *src1*). If *src0*.{xy|zw} and the corresponding component in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, it is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the control field, `pri_modifier_present`, and `sec_modifier_present` fields must be zero.

These instructions are valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_GE, IL_OP_U_LT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

64-Bit Integer Shift Left, Integer Shift Right

Instructions **U64SHR**

Syntax `u64shr` *dst, src0, src1*

Description Component-wise shift of each 64-bit value in *src0* right by an unsigned integer bit count provided by the lower five bits (0-63 range) in *src1*, inserting 0. The 64-bit per component result is placed in *dst*. Shifts of 65 and 1 are treated identically. The count is a scalar value applied to all components.

$dst = \lfloor u(src0)/2^{src1 \;\&\; 0x3F} \rfloor$

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U64_SHR
`control`	63:16	Must be zero.

Related None.

Unsigned Integer Compare

Instructions **UGE, ULT**

Syntax

Function	Opcode	Syntax
≥	`IL_OP_U_GE`	`uge` *dst, src0, src1*
<	`IL_OP_U_LT`	`ult` *dst, src0, src1*

Description Component-wise compares two vectors using an unsigned integer comparison. For UGE: (*src0* ≥ *src1*); for ULT: (*src0* < *src1*). If *src0*.{x|y|z|w} and the corresponding component in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, it is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the control field, `pri_modifier_present`, and `sec_modifier_present` fields must be zero.

These instructions are valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_GE, IL_OP_U_LT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Unsigned Integer Multiply and Add

Instructions **`UMAD`**

Syntax `umad` *`dst, src0, src1, src2`*

Description Component-wise unsigned multiply of the 32-bit operand *src0* by the 32-bit operand *src1* is added to the corresponding component in *src2*. The result of the multiply-add operation is placed in the corresponding component of *dst*.

Valid for R600 GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MAD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Unsigned 24-Bit Integer Multiply and Add

Instructions **`UMAD24`**

Syntax `umad24` *`dst, src0, src1, src2`*

Description Component-wise 24-bit unsigned integer muladd. The operands *src0* and *src1* are treated as 24-bit, unsigned integers; bits [31:24] are ignored. The operand *src2* is treated as a 32-bit signed or unsigned integer. The result represents the low-order 32-bits of the multiply-add operation.

dst = *src0*[23:0] * *src1*[23:0] + *src2*[31:0]

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MAD24
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Unsigned Integer Maximum, Unsigned Integer Minimum

Instructions **UMAX, UMIN**

Syntax

Function	Opcode	Syntax	Description
UMAX	`IL_OP_U_MAX`	umax *dst*, *src0*, *src1*	unsigned integer maximum, *dst* = *src0* > *src1* ? *src0*:*src1*
UMIN	`IL_OP_U_MIN`	umin *dst*, *src0*, *src1*	unsigned integer minimum, *dst* = *src0* < *src1* ? *src0*:*src1*

Description Compares each component of *src0* with the corresponding component of *src1*. If the comparison is TRUE, *src0* is returned in the corresponding component of *dst*; otherwise, *src1* is returned.

For the MAX and MIN instructions, if *src0*.{x|y|z|w} or *src1*.{x|y|z|w} is NaN, then the other component, *src1*.{x|y|z|w} or *src0*.{x|y|z|w}, respectively, is returned. Denorms are flushed before the comparison is performed. If a flushed denorm is the maximum value (for the MAX instruction) or minimum value (for the MIN instruction), then the flushed denorm is returned. The MAX instruction uses a greater-than-or-equal-to comparison. Thus, if min(*src0*, *src1*) = *src0*, then max(*src0*, *src1*) = *src1*, including cases with +0 and -0 such as when denorms are flushed to sign preserve zero.

If the _ieee flag is included, MIN and MAX follow IEEE-754r rules for minnum and maxnum (except denorms are flushed before the comparison is made). In addition, MIN returns -0 and MAX returns +0 for comparisons between -0 and +0.

Both instructions are valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	See *Syntax*, above.
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Unsigned Integer Modulo

Instructions	**`UMOD`**
Syntax	`umod` *`dst, src0, src1`*
Description	Component-wise unsigned division of the 32-bit operand *src0* by the 32-bit operand *src1*. The 32-bit remainder is placed in the corresponding component of *dst*. Valid for R600 GPUs and later.
Format	2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MOD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Unsigned Integer Multiplication

Instructions	**`UMUL`**
Syntax	`umul` *`dst, src0, src1`*
Description	Component-wise multiply of 32-bit unsigned operands *src0* and *src1*. The lower 32 bits of the 64-bit result (per component) is placed in the corresponding component of *dst*. Valid for R600 GPUs and later.
Format	2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MUL
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related UMUL_HIGH.

Unsigned Integer Multiplication

Instructions **`UMUL_HIGH`**

Syntax `umul_high` *`dst, src0, src1`*

Description Component-wise multiply of 32-bit unsigned operands *src0* and *src1*. The upper 32 bits of the 64-bit result (per component) is placed in the corresponding component of *dst*.

Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MUL_HIGH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related UMUL.

24-Bit Unsigned Integer Multiply

Instructions **`UMUL24`**

Syntax `umul24` *`dst, src0, src1`*

Description Component-wise 24-bit unsigned integer multiply. The operands *src0* and *src1* are treated as 24-bit, unsigned integers; bits [31:24] are ignored. The result represents the low-order 32-bits of the 48-bit multiply operation.

dst = *src0*[23:0] * *src1*[23:0]

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MUL24
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

24-Bit, High-Order Unsigned Integer Multiply

Instructions	**`UMUL24_high`**
Syntax	`umul24_high` *`dst, src0, src1`*
Description	Component-wise 24-bit unsigned integer multiply. The operands *src0* and *src1* are treated as 24-bit, unsigned integers; bits [31:24] are ignored. The result represents the high-order 16-bits of the 48-bit multiply operation. *dst* = *src0*[23:0] * *src1*[23:0] Valid for Evergreen GPUs and later.
Format	2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_MUL24_HIGH
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer Shift Left, Integer Shift Right

Instructions	**`USHR`**
Syntax	`ushr` *`dst, src0, src1`*
Description	Component-wise shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the lower five bits (0-31 range) in *src1*, inserting 0. The 32-bit per component result is placed in *dst*. Shifts of 33 and 1 are treated identically. The count is a scalar value applied to all components. dst = $\lfloor u(src0)/2^{src1\ \&\ 0x1F} \rfloor$ Valid for R600 GPUs and later.
Format	2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U_SHR
`control`	31:16	Must be zero.

Related None.

7.9 Bit Operations

Bit Extraction (Signed Shifts)

Instructions	**`IBIT_EXTRACT`**
Syntax	`ibit_extract` *`dst, src0, src1, src2`*

Description

Enables arbitrary sized bit-string extracts. The first two sources specify a bit-width and a bit-start position.

`width = src0 ^ 0x1F` selects a bit field width (0-31).

`offset = src1 ^ 0x1F` selects a bit field offset (0-31).

The field starts at bit-position offset and extends the width to the left. The field is placed right justified into *dst* and is sign-extended.

The following pseudo code provides the details.

```
if (width == 0) {
    dst = 0;
} else {
    if (width + offset < 32) {
    dst = (sr2 << (32- (width + offset)));
    dst = dst >> (32-width); // signed shifts
    } else {
        dst = src2 >> offset; // signed shifts
    }
  }
}
```

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_BIT_EXTRACT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Integer Count Bits

Instructions **ICOUNTBITS**

Syntax `icbits dst, src0`

Description Component wise count of the number of bits set to 1 in `src0`.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ICBITS
`control`	31:16	Must be zero.

Related None.

Integer Find First Bit

Instructions **IFIRSTBIT**

Syntax `ffb(option) dst, src0`

Description Find the first bit set in a number, either from the LSB or the MSB. A third variant that interprets the number as signed and behaves differently based on the sign (see following valid options).

Valid options are:
- lo – search from low bit to high (control = 0)
- hi – search from high bit to low (control = 1)
- shi - returns the first 0 from the MSB if the number is negative, else the first 1 from the MSB (control = 2)

Returns, component-wise, the integer position of the first bit set in the 32-bit input, starting from the LSB for `ffb_lo` or MSB for `ffb_hi`. For example:
`ffb_lo` on `0x00000001` gives the result 0.

All variants of the instruction return 0xFFFFFFFF (-1) in the `dst` register if no match was found.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_I_FIRSTBIT
`control`	31:16	Specifies the option.

Related None.

Bit Extraction (Unsigned Shifts)

Instructions **UBIT_EXTRACT**

Syntax `ubit_extract dst, src0, src1, src2`

Description Enables arbitrary sized bit-string extracts. The first two sources specify a bit-width and a bit-start position.

`width = src0 ^ 0x1F` selects a bit field width (0-31).

`offset = src1 ^ 0x1F` selects a bit field offset (0-31).

The field starts at bit-position offset and extends the width to the left. The field is placed right justified into *dst* and is zero-extended.

The following pseudo code provides the details.

```
if (width == 0) {
    dst = 0;
} else {
    if (width + offset < 32) {
    dst = (src2 << (32- (width + offset)));
    dst = dst >> (32-width);     // unsigned shifts
    } else {
        dst = src2 >> offset;    // unsigned shifts
    }
}
```

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_U_BIT_EXTRACT
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related UBIT_INSERT, UBIT_REVERSE.

Bit Insert

Instructions **UBIT_INSERT**

Syntax `ubit_insert` *dst, src0, src1, src2, src3*

Description Replaces a range of bits. Component-wise, the five lsb of *src0* specify a bitfield width (0-31); the five lsb of *src1* specify the bit offset from bit 0; *src2* specifies the replacement bits; and *src3* specifies Dword for which the bits are to be replaced.

```
dst = src4
If ( width != 0 ) {
   bitmask = (((1<<width)-1) << offset) & 0xFFFFFFFF
   dst = ((src2<<offset) & bitmask) | (src3 & ~bitmask)
}
```

Valid for Evergreen GPUs and later.

Format 4-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_U_BIT_INSERT
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related UBIT, UBIT_EXTRACT, UBIT_REVERSE.

Reverse Bits in a Register

Instructions **UBIT_REVERSE**

Syntax `ubit_reverse` *dst, src0*

Description Reverses the bits in a register.

For example, `0x12345678` results in `0x1E6A2C4`.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_U_BIT_REVERSE
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related UBIT, UBIT_EXTRACT, UBIT_INSERT.

7.10 Conversion Instructions

Double to Float Conversion

Instructions **D2F**

Syntax `d2f dst, src0`

Description Converts a double to a float.

dst.xyzw = (float)src0.xy

The single double in src0.xy is converted to a float. The result then is broadcast to all the unmasked destination channels. This is different from the DX specification, which converts src.xy to the first unmasked component of the destination and src.zw to the second component.

```
dx: d2f r1.xy, r2
il: d2f r1.x, r2.xy
    dwf r1.z, r2.zw
```

Abs and negate modifiers can be used on the source. All four output components are set to the float converted value. Nan/inf convert to nan/inf; signs are preserved. If the result does not fit as a float, inf is returned. Uses round to nearest even.

d	-NaN	-inf	-F	-1.0	-denom	-0	+0.0	+denom	+1.0	+F	+inf	+NaN
f	-NaN	-inf	-F	-1.0	-0.0	-0.0	+0.0	+0.0	+1.0	+F	+inf	+NaN

Valid for all GPUs that support double precision.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_2_F
`control`	31:16	Must be zero.

Related F2D.

Float to Double Conversion

Instructions	**F2D**
Syntax	`f2d` *dst*, *src0*

Description

The source is interpreted as a float in component x (after swizzles). Abs and negate modifiers can be used on the source. Output yx or output wz is set to the double converted value. Nan/inf convert to nan/inf; signs are preserved.

Rounding is performed towards zero on each component of *src0*, following the ANSI C convention for casts from float to double. Applications that require different rounding semantics can invoke the ROUND_* instructions before using F2D.

f	-NaN	-F	-1.0	-denorm	-0.0	+0.0	+denorm	+1.0	+F	+inf	NaN
d	-NaN	-F	-1.0	-0.0	-0.0	+0.0	+0.0	+1.0	+F	+inf	Nan*

Valid for all GPUs that support double precision.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_F_2_D
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Converts Float to 16-Bit Float

Instructions	**F_2_F16**
Syntax	`f2f16 dst, src0`

Description

Each channel of `src0` is converted to a float16; the result is placed into the 16 lsb of `dst`.

This instruction packs values before writing them to memory.

There are no float16 arithmetic operations.

Valid for Evergreen GPUs and later with double-precision floating-point.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_F_2_F16
`control`	31:16	Must be zero.

Related F16_2_F.

Converts 16-bit Float to Float

Instructions	**`F16_2_F`**
Syntax	`f162f dst, src0`

Description

Each channel of `src0` is converted form lsb float16 to a float and written into `dst`.

This instruction unpacks values after reading them from memory.

Conversions are exact, with no rounding required.

There are no float16 arithmetic operations.

Becuase there are many represenions of a NaN, converting a NaN from one floating point size to another and then back to the origial is not guaranteed to return the same bits for a NaN input.

Valid for Evergreen GPUs and later with double-precision floating-point.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_F16_2_F
`control`	31:16	Must be zero.

Related F_2_F16.

Float to Signed Integer Conversion

Instructions	**`FTOI`**
Syntax	`ftoi` *`dst`*, *`src0`*

Description

Converts each component of *src0* in the range [-2147483648.999f, 2147483647.999f] to a signed integer, and puts the result in the corresponding component of *dst*. Using values outside the specified range produces undefined results.

After the instruction executes, *`dst`* contains a 32-bit signed integer 4-tuple. The conversion is performed per component.

Rounding is performed towards zero on each component of *src0*, following the C convention for casts from float to int. Applications that require different rounding semantics can invoke the ROUND_* instructions before using FTOI.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_FTOI
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related FTOU, ITOF, UTOF.

Float to Unsigned Integer Conversion

Instructions **`FTOU`**

Syntax `ftou` *`dst`*, *`src0`*

Description Converts each component of *src0* in the range [0.0f, 4294967296.999f] to an unsigned integer and puts the result in the corresponding component of *dst*. Using values outside the specified range produces undefined results.

After the instruction executes, *dst* contains a 32-bit unsigned integer 4-tuple. The conversion is performed per component.

Rounding is performed towards zero on each component of *src0*, following the C convention for casts from float to unsigned int. Applications that require different rounding semantics can invoke the ROUND_* instructions before using FTOU.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_FTOU
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related FTOI, FTOU, ITOF, UTOF.

Signed Integer to Float Conversion

Instructions **`ITOF`**

Syntax `itof` *`dst`*, *`src0`*

Description Converts each component of *src0* to a float, and puts the result in the corresponding component of *dst*. Each component of *src0* is assumed to contain a signed 32-bit integer 4-tuple. After conversion, *dst* contains a floating-point 4-tuple.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ITOF
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related UTOF, FTOU, FTOI.

Unsigned Integer to Float Conversion

Instructions **`UTOF`**

Syntax `utof` *`dst`*, *`src0`*

Description Converts each component of *src0* to a float and puts the result in the corresponding component of *dst*. Each component of *src0* is assumed to contain an unsigned 32-bit integer 4-tuple.

After the instruction executes, *dst* contains a floating-point 4-tuple.

If an integer is not represented exactly, the nearest representable value is used. Rounding is performed towards zero on each component of *src0*, following the C convention for casts from unsigned int to float. Applications that require different rounding semantics can invoke the ROUND_* instructions before using UTOF.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_UTOF
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related ITOF, FTOI, FTOU.

7.11 Float Instructions

Absolute Value

Instructions	**ABS**
Syntax	`abs dst, src0`

Description Computes the absolute value of each component in a vector (*src0*).

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
for (=0; i < 4; i++)
     v[i] = abs(v1[i]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ABS
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Inverse Cosine (arccos)

Instructions	**ACOS**
Syntax	`acos dst, src0`

Description Computes the inverse cosine in radians of *src0*.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the w component. *src0*.w must be within the range [-1.0, 1.0]; otherwise, the result is undefined. The maximum absolute error is 0.002.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v
v[0] = v[1] = v[2] = v[3] = acos(v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ACOS
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Floating Point Addition

Instructions **ADD**

Syntax `add dst, src0, src1`

Description Adds two float vectors: *src0*.{x|y|z|w} + *src1*.{x|y|z|w}.

Valid for R600 GPUs and later.

Operation:

```
VECTOR v1 = EvalSource(src1);
VECTOR v2 = EvalSource(src2);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v1[i] + v2[i];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ADD
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Logical-AND

Instructions **AND**

Syntax `and dst, src0, src1`

Description Performs a component-wise logical AND of each pair of 32-bit values from *src1* and *src2*. The 32-bit result is placed in *dst*.

Valid for R6XX GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_AND
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Inverse Sine (arcsin)

Instructions **ASIN**

Syntax `asin dst, src0`

Description Computes the inverse sine in radians of the w component of *src0*. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the w component. *src0*.w must be within the range [-1.0, 1.0]; otherwise, the result is undefined. The maximum absolute error is 0.002.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v
v[0] = v[1] = v[2] = v[3] = asin(v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ASIN
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Inverse Tangent (arctan)

Instructions	**`ATAN`**
Syntax	`atan dst, src0`

Description

Computes the inverse tangent in radians of *src0*.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the w component. *src0*.w must be within

the range [$-\frac{\pi}{2}$, $\frac{\pi}{2}$]; otherwise, the result of this instruction is undefined.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v
v[0] = v[1] = v[2] = v[3] = atan(v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ATAN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Clamp to Given Range

Instructions **CLAMP**

Syntax `clamp` *`dst`*, *`src0`*, *`src1`*, *`src2`*

Description Clamps *src0* to *src1* and *src2*.

Valid for all GPUs.

Example:

dst = min(max(src0,src1), src2)

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v3 = EvalSource(src2);
VECTOR v;
for (i=0; i < 4; i++)
{
    v[i] = v1[i];
    if(v[i] < v2[i])
    {
        v[i] = v2[i];
    }
    if(v[i] > v3[i])
    {
        v[i] = v3[i];
    }
}
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CLAMP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Component-Wise Conditional Move

Instructions **CMOV**

Syntax `cmov dst, src0, src1`

Description Conditionally moves a value from *src1* to *dst*. If *src0*.{x|y|z|w} is not 0.0f, the value of the corresponding component in *dst* becomes the value of the corresponding component of *src1*; otherwise, the corresponding component in *dst* remains unchanged.

The compare is float, so both 0.0 and -0:0 do the move.

IL_DSTMOD_1 and IL_DSTMOD_0 in the IL_Dst_Mod token are treated as IL_DSTMOD_NOWRITE for components where *src1* is 0.0f.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
{
    v[i] = v2[i];
    if(v1[i] == 0.0)
        v[i].MASK = NOWRITE;
}
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_CMOV
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related CMOV_LOGICAL.

Component-wise Conditional Logical Move

Instructions	**`CMOV_LOGICAL`**
Syntax	`cmov_logical` *`dst`*`,` *`src0`*`,` *`src1`*`,` *`src2`*
Description	For each component in *src0* (post-swizzle): if the component has any bit set, the component in *src1* (post-swizzle) is copied to the corresponding component in *dst*; otherwise, the corresponding component in *src2* is copied to the corresponding component in *dst*. dst[i] = (src0[i]! = 0)?src1[i] : src2[i] The compare is integer, so only a value of 0x0 causez *src2* to be used. Valid for R600 GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_CMOV_LOGICAL
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related CMOV.

Compare with Value

Instructions **CMP**

Syntax `cmp_relop(op)_cmpval(cmpval) dst, src0, src1, src2`

Description For every component of *src0* that passes (is TRUE for) its relational comparison with *cmpval*, that component is set to the corresponding component of *src1*; otherwise, it is set to the corresponding component of *src2*.

Valid on all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src1);
VECTOR v2 = EvalSource(src2);
VECTOR v3 = EvalSource(src3);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = (v1[i] relop cmpval) ? v2[i] : v3[i];
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_CMP		
control	29:16	**Field Name**	**Bits**	**Description**
		relop	18:16	Any value of the enumerated type ILRelOp(*relop*). See Table 6.23 on page 6-18.
		reserved	20:19	Must be zero.
		cmpval	23:21	Any value of the enumerated type ILCmpValue(*cmpval*). See Table 6.3 on page 6-2.
		reserved	29:24	Must be zero.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related None.

Clamp Color Output Values

Instructions **COLORCLAMP**

Syntax `colorclamp` *`dst`*`,` *`src0`*

Description Clamps (or does not clamp) an output color to values specified by the frame buffer color clamp state.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
int buffer = RegisterNum(src0);
VECTOR v;
for (i=0; i < 4; i++)
{
    if(AS_CB_CLAMP_MODE_N(buffer) == 0_1)
{
    v[i] = (v1[i] <= 0.0) ? 0.0 : v1[i];
    v[i] = (v[i] > 1.0) ? 1.0 : v[i];
}
else if (AS_CB_CLAMP_MODE_N(buffer) == NEG_1_1)
{
    v[i] = (v1[i] < -1.0) ? -1.0 : v1[i];
    v[i] = (v[i] > 1.0) ? 1.0 : v[i];
}
else if (AS_CB_CLAMP_MODE_N(buffer) == NONE)
{
    v[i]=v1[i];
}
}
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_COLORCLAMP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Cosine (cos)

Instructions	`COS`
Syntax	`cos dst, src0`

Description

Computes the cosine of src0.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component.

The fourth component of *src0* must be in the range [-π, π]; otherwise, the result is of this instruction is undefined.

The max absolute error is 0.002.

Valid for all GPUs.

Example:

dst = cos(src0)

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_COS
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related COS_VEC.

Component-Wise Cosine

Instructions	`COS_VEC`
Syntax	`cos_vec dst, src0`

Description

Computes the cosine of each component of *src0* in radians. The maximum absolute error is 0.0008 in the range [-100*π, 100*π].

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_COS_VEC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related COS.

Cross Product

Instructions **CRS**

Syntax `crs dst, src0, src1`

Description Computes a cross product. This instruction does not write to the *dst*.w component; however, the if the component_w_a field of the IL_Dst_Mod token is set to IL_MODCOMP_0 or IL_MODCOMP_1, *dst*.w is written. That is, *dst*.w can be set to 0.0 or 1.0 if IL_MODCOMP_0 or IL_MODCOMP_1 is used on the component_w_a field of the IL_Dst_Mod token.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
v[0] = v1[1] * v2[2] - v1[2] * v2[1];
v[1] = v1[2] * v2[0] - v1[0] * v2[2];
v[2] = v1[0] * v2[1] - v1[1] * v2[0];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_CRS
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Determinant of a 4x4 Matrix

Instructions **DET**

Syntax `det dst, src0`

Description Calculates the determinant of a 4x4 matrix. Relative addressing and source modifiers of *src0* are applied to each row of the matrix beginning with *src0*. The 32-bit scalar result is placed in all four components of *dst.*

Valid for all GPUs.

Operation:

```
VECTOR v0 = EvalSource(src0);
VECTOR v1 = EvalSource(src0+1); # Register number of src0 + 1
VECTOR v2 = EvalSource(src0+2); # Register number of src0 + 2
VECTOR v3 = EvalSource(src0+3); # Register number of src0 + 3
VECTOR v;
float f;

f = v0[0]*(v1[1]*(v2[2]*v3[3]-v2[3]*v3[2])
   -v1[2]*(v2[1]*v3[3]-v3[1]*v2[3])
   +v1[3]*(v2[1]*v3[2]-v3[1]*v2[2]))
   -v0[1]*(v1[0]*(v2[2]*v3[3]-v2[3]*v3[2])
   -v1[2]*(v2[0]*v3[3]-v3[0]*v2[3])
   +v1[3]*(v2[0]*v3[2]-v3[0]*v2[2]))
   +v0[2]*(v1[0]*(v2[1]*v3[3]-v2[3]*v3[1])
   -v1[1]*(v2[0]*v3[3]-v3[0]*v2[3]
   +v1[3]*(v2[0]*v3[1]-v3[0]*v2[1]))
   -v0[3]*(v1[0]*(v2[1]*v3[2]-v2[2]*v3[1])
   -v1[1]*(v2[0]*v3[2]-v3[0]*v2[2])+v1[2]*(v2[0]*v3[1]-v3[0]*v2[1]));
v[0] = v[1] = v[2] = v[3] = f;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DET
`control`	29:16	Must be the enumerated type ILMatrix(*IL_MATRIX_4X4*). See Table 6.16 on page 6-8.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Vector Distance

Instructions	**DIST**
Syntax	`dist` *`dst`*`,` *`src0`*`,` *`src1`*

Description Computes the vector distance from *src0*.[xyz] to *src1*.[xyz]. The 32-bit scalar result is placed in all four components of *dst.*

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0;
VECTOR v2 = EvalSource(src1;
VECTOR v;
V[0] = v[1] = v[2] = v[3] =
    sqrt((v1[0]-v2[0])*(v1[0]-v2[0]) +
         (v1[1]-v2[1])*(v1[1]-v2[1]) +
         (v1[2]-v2[2])*[v1[2]-v2[2]) +
         (v1[3]-v2[3])*(v1[3]-v2[3]));
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DIST
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Floating Point Division

Instructions **DIV**

Syntax `div_zeroop(op) dst, src0, src1`

Description Floating point division *src0*.{x|y|z|w} / *src1*.{x|y|z|w}. The 32-bit quotient is placed in the corresponding component of *dst*. DX10 requires the DIV instruction to use zeroop = IL_ZEROOP_INFINITY. If no zeroop value is provided, zeroop(fltmax) is used.

dst - 1.0/src0

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
float f;
for (i=0; i < 4; i++)
{
    if(v2[i] == 0.0)
    {
        if(zeroop == IL_ZEROOP_0)
            f = 0.0;
        else if(zeroop == IL_ZEROOs_FLT_MAX)
            f = FLT_MAX;
    else if(zeroop == IL_ZEROOP_INFINITY)
            f = INFINITY;
        else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
            f = INFINITY or FLT_MAX;  # Depends on IL Implementation

        if(v1[i] < 0.0)
            f = -f;

        v[i] = f;
   }
    else
        v[i] = v1[i] / v2[i];
}
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_DIV
control	29:16	DIV: Any value of the enumerated type ILZeroOp(*zeroop*). See Table 6.33 on page 6-22.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Two-Component Dot Product

Instructions	**DP2**
Syntax	dp2[_ieee] *dst, src0, src1*

Description Two-component dot product of vector operands *src0*.[xy] and *src1*.[xy]. The 32-bit scalar result is placed in all four components of *dst*. If the control field is 0, then 0*n = 0, even if n = NaN.

dst.xyzw = src0.x ⊗ src1.x + src0.y ⊗ src1.y

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
v[0] = v[1] = v[2] = v[3] = v1[0]*v2[0] + v1[1]*v2[1];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_DP2
control	29:16	0 DX9 DP2-style multiple. 1 IEEE-style multiply.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related DP2ADD, DP3, DP4.

Two-Component Dot Product and Scalar Add

Instructions **DP2ADD**

Syntax dp2add *dst*, *src0*, *src1*, *src2*

Description Computes the two-component dot product of *src0*.[xy] and *src1*.[xy] and adds scalar *src2*.z to the result. The 32-bit scalar result is placed in all four components of *dst*. This instruction corresponds to the DX10 dp2 operation. If the control field is 0, then 0*n equals zero, even if n = NaN.

dst.xyzw = src0.x ⊗ src1.x + src0.y ⊗ src1.y + src2.z

Not IEEE exact.

Valid for R600 GPUs and later.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v3 = EvalSource(src2);
VECTOR v;
v[0] = v[1] = v[2] = v[3] =
    v1[0]*v2[0] + v1[1]*v2[1] + v3[2];
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_DP2ADD
control	29:16	0 DX9 DP2-style multiple. 1 IEEE-style multiply.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related DP2, DP3, DP4.

Three-Component Dot Product

Instructions **DP3**

Syntax `dp3[_ieee] dst, src0, src1`

Description Three-component dot product vector operands *src0*.[xyz] and *src1*.[xyz]. The 32-bit scalar result is placed in all four components of *dst*. If the control field is 0, then 0*n = 0, even if n = NaN.

dst.xyzw = src0.x ⊗ src1.x + src0.y ⊗ src1.y + src2.z ⊗ src1.z

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
v[0] = v[1] = v[2] = v[3] =
    v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_DP3
control	29:16	0 DX9 DP2-style multiple. 1 IEEE-style multiply.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related DP2, DP2ADD, DP4.

Four-Component Dot Product

Instructions **DP4**

Syntax `dp4[_ieee]` *`dst, src0, src1`*

Description Four-component dot product vector operands *src0*.[xyzw] and *src1*.[xyzw]. The 32-bit scalar result is placed in all four components of *dst*. If the control field is 0, then 0*n = 0, even if n = NaN.

dst.xyzw = src0.x $\otimes$ src1.x + src0.y $\otimes$ src1.y + src0.z $\otimes$ src1.z + src0.w $\otimes$ src1.w

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
v[0] = v[1] = v[2] = v[3] =
    v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2] + v1[3]*v2[3];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DP4
`control`	29:16	0 DX9 DP2-style multiple. 1 IEEE-style multiply.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related DP2, DP2ADD, DP3.

Vector Distance

Instructions **DST**

Syntax dst *dst*, *src0*, *src1*

Description Computes a distance-related value from operands *src0* and *src1*. The result vector is placed in *dst*. If the control field is 0, then 0*n = 0, even if n = NaN.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
V[0] = 1.0;
V[1] = v1[1] * v2[1];
V[2] = v1[2];
V[3] = v2[3];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_DST
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Instantaneous Derivative in X

Instructions **DSX**

Syntax `dsx[_fine]` *`dst, src0`*

Description Instantaneous derivative in the x-direction. Computes the rate of change of each float32 component of *src0* (post-swizzle) in the RenderTarget x direction. The results are undefined if used in a vertex or geometry shader.

When `_fine` is not specified, the data in the current pixel shader invocation may or may not participate in the calculation of the requested derivative, since the derivative is calculated only once per 2x2 quad.

GPUs before the Evergreen series ignore the `_fine` setting.

Valid for all GPUs.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_DSX		
`control`	29:16	**Field Name**	**Bits**	**Description**
		reserved	22:16	Must be zero.
		`_fine`	23	0: Gradients can be computed once per quad. 1: Gradients are computed for each pixel.
		reserved	29:24	Must be zero.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related DSY, DXSINCOS.

Instantaneous Derivative in Y

Instructions	**`DSY`**
Syntax	`dsy[_fine]` *`dst`*`,` *`src0`*

Description

Instantaneous derivative in the y-direction. Computes the rate of change of each float32 component of *src0* (post-swizzle) in the RenderTarget y direction. The results are undefined if used in a vertex or geometry shader. If bit 8 is zero, only one x,y derivative pair is computed for each 2x2 stamp of pixels.

When `_fine` is not specified, the data in the current pixel shader invocation may or may not participate in the calculation of the requested derivative, since the derivative is calculated only once per 2x2 quad.

GPUs before the Evergreen series ignore the `_fine` setting.

Valid for all GPUs.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description		
`code`	15:0	IL_OP_DSY		
`control`	29:16	**Field Name**	**Bits**	**Description**
		reserved	22:16	Must be zero.
		`fine`	23	0: Gradients can be computed once per quad. 1: Gradients are computed for each pixel.
		reserved	29:24	Must be zero.
`sec_modifier_present`	30	Must be zero.		
`pri_modifier_present`	31	Must be zero.		

Related DSX, DXSINCOS.

Compute Sine and Cosine

Instructions **DXSINCOS**

Syntax `dxsincos` *dst, src0, src1, src2*

Description

Computes sine and cosine values of *src0.w* for the legacy DX9 sincos instruction. The results are returned in *dst*.x and *dst*.y.

This instruction supports the DX legacy instruction *sincos dst*, *src0*, *src1*, *src2*. *src1* and *src2* must be constant float registers and are used in a Taylor series expansion. See the *DX SDK* to understand how the two constants are used in a Taylor series expansion. Devices with native sincos support ignore *src1* and *src2*.

By default this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. The fourth component of *src* must be within the range [-π, π]; otherwise, the result of this instruction is undefined.

This instruction does not write to the *dst*.z and *dst*.w components; however, if the `component_z_b` or the `component_w_a` field of the IL_Dst_Mod token is set to IL_MODCOMP_0 or IL_MODCOMP_1, the *dst*.z and *dst*.w are written. That is, *dst*.z and *dst*.w can be set to 0.0 or 1.0 if IL_MODCOMP_0 or IL_MODCOMP_1 is used on the `component_z_b` or `component_w_a` field of the IL_Dst_Mod token.

The maximum absolute error is 0.002.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
v[0] = cos(v1[3]);  // approximated by using Taylor series
v[1] = sin(v1[3]);  // approximated by using Taylor series
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_DXSINCOS
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related DSX, DSY.

Equality Compare Floats

Instructions	**EQ**
Syntax	eq *dst*, *src0*, *src1*

Description

Performs the float comparison *src0* == *src1* for each component, and writes the result to *dst*. If the comparison is true, 0xFFFFFFFF is returned for that component; otherwise, 0x0000000 is returned. This instruction follows DX10 Floating Point Rules. Denorms are flushed before comparison (original source registers untouched). +0 equals -0. Comparison with NaN returns false.

Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_EQ
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Full-Precision e to the Power of x

Instructions	**EXN**
Syntax	exn *dst*, *src0*

Description

Full-precision base e power *src0*: e^{src}.

By default, this instruction operates on *src0.w*, but can operate on any component by swizzling it into the fourth component.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
v[0] = v[1] = v[2] = v[3] = exp2(v1[3] * log2(e));
WriteResult(v, dst);
```

Format 1-input 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_EXN
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related EXP, EXP_VEC, EXPP.

2 Raised to a Power

Instructions **`EXP`**

Syntax `exp` *`dst, src0`*

Description Full precision base 2 power *src0*: 2^{src}.

Computes two to the power of *src0.w*. The floating point result is placed in all four components of `dst`. By default this instruction operates on *src0.w*, but can operate on any component by swizzling it into the fourth component.

Valid for all GPUs.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EXP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related EXP_VEC, EXPP, EXN.

Component-Wise Full Precision 2 to the Power of X

Instructions **`EXP_VEC`**

Syntax `exp_vec` *`dst, src0`*

Description EXP computes a scalar version, EXP_VEC computes the same result per component.

Computes 2 raised to the power of each component of *src0*. The result of the operation is accurate to at least 21 bits. The 32-bit floating point results are placed in the corresponding components of *dst*.

Valid for R600 GPUs and later.

Example:

dst = exp(src0)

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_EXP_VEC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related EXN, EXP, EXPP.

Component-Wise Partial Precision 2 to the Power of X

Instructions **EXPP**

Syntax `expp dst, src0`

Description Computes partial precision of 2 raised to the power of each component of *src0*. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. The result of the operation is accurate to at least 10 bits. The 32-bit floating point results are placed in the corresponding components of *dst*.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f = floor(v1[3]);
v[0] = exp2(f); At least 10 bits of precision.
v[1] = v1[3] - f;
v[2] = exp2(v1[3]);
v[3] = 1.0;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_EXPP
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related EXN, EXP, EXP_VEC.

Face Forward

Instructions **FACEFORWARD**

Syntax `faceforward dst, src0, src1, src2`

Description Performs the following calculation:

d(dst = *src2**sign(dot(*src0*, *src1*))

Valid for all GPUs.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_FACEFORWARD
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Floor

Instructions **FLR**

Syntax `flr` *`dst`*`,` *`src0`*

Description Performs a component-wise floor operation on the operand to generate a result vector. Calculates the floor of each component of *src0*, and places the 32-bit result in the corresponding component of *dst*.The floor of a component is defined as the largest integer less-than-or -equal to the value in the component. For example, the floor of 2.3 is 2.0 and the floor of -3.6 is -4.0. The operation is identical to ROUND_NEG_INF.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = floor(v1[i]);
WriteResult(v, dst);
```

Example:

`(float)(⌊src0⌋) floor(2.3) = 2.0, floor(-3.6) = -4.0`

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_FLR
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Fused Multiply Add

Instructions **FMA**

Syntax `fma` *`dst`*`,` *`src0`*`,` *`src1`*`,` *`src2`*

Description Computes `(src0 * src1) + src2` in infinite precision; rounds the result to single precision and stores the result in `dst`. This is not equivalent to a mul followed by an add since there are two roundings: (one after the mul, and one after the add).

Valid for Evergreen GPUs and later with double precision capability.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_FMA
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Fraction

Instructions	**`FRC`**
Syntax	`frc dst, src0`

Description

Extracts the fractional portion of each component of *src0*. The results are returned in the corresponding component of *dst*. The fractional portion of a component is defined as the result after subtracting the floor of the component from the component (see FLR). The result is always in the range [0.0, 1.0).

For negative values, the fractional portion is not the number to the right of the decimal point. For example, the fractional portion of -1.7 is 0.3, not 0.7. In this case, it is produced by subtracting the floor of -1.7 - (-2.0) from 1.7.

Valid for all GPUs.

Example:

src0 - floor(src0); frc(1.7) = 0.3

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v1[i] - (float)floor(v1[i]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_FRC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Filter Width

Instructions **FWIDTH**

Syntax `fwidth dst, src0`

Description Computes the sum of the absolute derivative in x and y using local differencing for each component of *src0*. The result is returned in the corresponding component of *dst*. If used in a vertex shader, the results are undefined.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
for (i=0; i < 4; i++)
    v[i] = abs(dPdx(v1[i])) + abs(dPdy(v1[i]));
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_FWIDTH
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Greater or Equal Floats

Instructions **GE**

Syntax `ge dst, src0, src1`

Description Compares two float vectors *src0* and *src1* for each component, and writes the result to *dst*. If the comparison is true, 0xFFFFFFFF is returned for that component; otherwise, 0x0000000 is returned. This instruction follows DX10 Floating Point Rules. Denorms are flushed before comparison (original source registers untouched). +0 equals -0. Comparison with NaN returns false.

Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for R600 GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_GE
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related

Move Data

Instructions **INV_MOV**

Syntax `invariant_move dst, src0`

Summary Move data between registers.

Description Moves value from *src0* to *dst*. As a result of using this instruction, the compiler ensures that any other shader that computes this source using the same instructions gets the identical answer. Generally, use of INV_MOVE prevents many compiler optimizations and lowers performance.

Valid for R600 GPUs and later.

Operation:

```
VECTOR v = EvalSource(src0);
WriteResult(v, dst);
```

Format 1-input 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_INVARIANT_MOV
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Vector Length

Instructions **LEN**

Syntax `len dst, src0`

Description Computes the length of a vector. Computes the vector length of the three component vector in *src0*.[xyz]. The scalar 32-bit floating point result is placed in all four components of *dst*.

Valid for all GPUs.

Example:

$\sqrt{\texttt{dst4(src0,src1)}}$

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
V[0] = v[1] = v[2] = v[3] =
    sqrt(v1[0]*v1[0]+ v1[1]*v1[1] + v1[2]*v1[2] + v1[3]*v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_LEN
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.

Vector Length

	Field Name	Bits	Description
	`pri_modifier_present`	31	Must be zero.

Related None.

Lighting Coefficient

Instructions **`LIT`**

Syntax `lit` *`dst`*`,` *`src0`*

Description Calculates lighting coefficients for ambient, diffuse, and specular light contributions.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float epsilon = 1.192092896e-07F;
float g = v1[3];
if(g < -(128.0-epsilon))
    g = -(128.0-epsilon);
else if (g > (128.0-epsilon))
    g = 128.0-epsilon;
v[0] = 1.0;
v[1] = (v1[0] > 0.0) ? v1[0] : 0.0;
v[2] = ((v1[0] > 0.0) && (V1[1] > 0)) ? EXP2(g * LOG2(v1[1])) : 0.0;
v[3] = 1.0;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LIT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Natural Logarithm

Instructions	**LN**
Syntax	`ln_zeroop(`*`op`*`)` *`dst`*`,` *`src0`*

Description

Computes the natural logarithm of *src0*.w. The result of the operation is accurate to at least bits. The 32-bit floating point result is placed in all four components of *dst*. By default this instruction operates on *src0.w*, but can operate on any component by swizzling it into the fourth component. zeroop can be any value of the enumerated type ILZeroOp(zeroop) except IL_ZEROOP_0. If no zeroop value is provided, zeroop(fltmax) is used.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f;
if (v1[3] == 0.0)
{
    if(zeroop == IL_ZEROOP_FLT_MAX)
        f = -FLT_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
        f = -INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
        f = -INFINITY or -FLT_MAX;  # Depends on IL Implementation
}
else if (v1[3] < 0.0)
{
    f = undefined;
}
else
{
    f = (float)(log10(v1[3])/log10(e));
}
v[0] = v[1] = v[2] = v[3] = f;
WriteResult(v, dst);
```

Format

1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LN
`control`	29:16	Any value of the enumerated type ILZeroOp(*zeroop*). See Table 6.33 on page 6-22.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Base-2 Logarithm

Instructions **LOG**

Syntax `log_zeroop(`*`op`*`) `*`dst, src0`*

Description Computes the base-2 logarithm of *src0*.w. The result of the operation is accurate to at least 21 bits. The 32-bit floating point result is placed in all four components of *dst*. By default this instruction operates on *src0.w*, but can operate on any component by swizzling it into the fourth component. zeroop can be any value of the enumerated type ILZeroOp(zeroop) except IL_ZEROOP_0. If no zeroop value is provided, zeroop(fltmax) is used.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f;
if (v1[3] == 0)
{
    if(zeroop == IL_ZEROOP_FLT_MAX)
        f = -FLT_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
        f = -INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
        f = -INFINITY or -FLT_MAX;  # Depends on IL Implementation
}
else if (v1[3] < 0.0)
{
    f = undefined;
}
else
{
    f = (float)(log10(v1[3])/log10(2));
}
v[0] = v[1] = v[2] = v[3] = f;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LOG
`control`	29:16	Any value of the enumerated type ILZeroOp(*zeroop*). See Table 6.33 on page 6-22.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Component-wise Base-2 Logarithm

Instructions **`LOG_VEC`**

Syntax `log_vec` *`dst, src0`*

Description Computes the base-2 logarithm of each component of *src0*. The result of the operation is accurate to at least 21 bits. This is the vector version of LOG with zeroop set to il_zeroop_infinity.

Valid for R600 GPUs and later.

Example:

dst = $\log_2$(src0)

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LOG_VEC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Base-2 Logarithm (partial precision)

Instructions **LOGP**

Syntax `logp_zeroop(op) dst, src0`

Description Computes the base-2 logarithm of *src0*.w using partial precision. The result of this computation is accurate to at least 10-bits. The 32-bit floating point result is placed in *dst*.z. By default, this instruction operates on *src0.w*, but can operate on any component by swizzling it into the fourth component.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f;
if (v1[3] == 0)
{
    if(zeroop == IL_ZEROOP_FLT_MAX)
        f = -FLT_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
        f = -INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
        f = -INFINITY or -FLT_MAX;  # Depends on IL Implementation
}
else if (v1[3] < 0.0)
{
    f = undefined;
}
else
{
    f = log2(v1[3]);
}
v[0] = floor(f);
v[1] = v1[3] / exp2(floor(f));
v[2] = f
v[3] = 1.0;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LOGP
`control`	29:16	Any value of the enumerated type ILZeroOp(`zeroop`) except IL_ZEROOP_0. See Table 6.33 on page 6-22.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Linear Interpolation

Instructions **LRP**

Syntax `lrp dst, src0, src1, src2`

Description Computes the linear interpolation between two vectors.

Valid for all GPUs.

Example:

src1 ⊗ src0 + src2 ⊗ (1.0 - src0)

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v3 = EvalSource(src2);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v2[i] * v1[i] + v3[i] * (1 – v1[i]);
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_LRP
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Less Than

Instructions	**`LT`**
Syntax	`lt` *`dst`*, *`src0`*, *`src1`*

Description

Compares two float vectors, *src0* and *src1*, for each component, and writes the result to *dst*. If the comparison is true, 0xFFFFFFFF is returned for that component; otherwise, 0x0000000 is returned. This instruction follows DX10 Floating Point Rules. Denorms are flushed before comparison (original source registers untouched). +0 equals -0. Comparison with NaN returns false.

Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for R6XX GPUs and later.

Format

2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Floating Point Multiply and Add

Instructions **MAD**

Syntax `mad[_ieee] dst, src0, src1, src2`

Description Multiplies a component in *src0* by the corresponding component in *src1*. In the ATI Radeon™ HD 3XXX and ATI Radeon™ HD 4XXX series graphics cards, after the ADD of *src0* and *src1*, one round occurs using round to next even (RNE). The multiplier result mantissa is resolved to 26 bits plus sticky and overflow. No normalization occurs. The lower 32 bits of the intermediate ADD result then is added to *src2*. Exponent overflow and underflow checks are done after rounding. Denorms are flushed to zeros on input and output.

The result of the multiply-add operation is placed in the corresponding component of *dst*.

dst = src0 $\otimes$ src1 +src2

Valid for R600 GPUs and later.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v3 = EvalSource(src2);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v1[i] * v2[i] + v3[i];
WriteResult(v, dst);
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MAD
control	29:16	MAD: 0 0*n = o, even if n is NaN. 1 IEEE-style multiply.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Floating Point Maximum

Instructions **MAX**

Syntax `max[_ieee] dst, src0, src1`

MAX	IL_OP_MAX	max[_ieee] *dst*, *src0*, *src1*	floating point maximum, *src0*.{x|y|z|w} ≥ *src1*.{x|y|z|w} ? *src0*.{x|y|z|w} : *src1*.{x|y|z|w}

Description

Computes maximum value per component. Both max(+0, -0) and max(-0, +0) return +0. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, any NaN representation is returned.

Denorms are flushed to sign preserved 0s before comparison; if it is the maximum, the flushed denorm is written to *dst*.

If the _ieee flag is included then MIN and MAX follow IEEE-754r rules for minnum and maxnum except denorms are flushed before the comparison is made. In addition, MIN returns -0 and MAX returns +0 for comparisons between -0 and +0.

Valid for R600 GPUs and later that support the IEEE flag.

dst = max(src0, src1)

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = (v1[i] > v2[i]) ? v1[i] : v2[i];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MAX
control	29:16	MAX: 0 No special NaN rules. 1 IEEE-style NaN rules.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Floating Point Minimum

Instructions **MIN**

Syntax `min[_ieee] dst, src0, src1`

Description Computes minimum value per component. Both min(+0, -0) and min(-0, +0) return -0. NaN has special handling: If one source operand is NaN, then the other source operand is returned (choice made per-component). If both are NaN, any NaN representation is returned. Denorms are flushed to sign preserved 0s before comparison; if it is the minimum, the flushed denorm is written to *dst*.

If the _ieee flag is included then MIN and MAX follow IEEE-754r rules for minnum and maxnum except denorms are flushed before the comparison is made. In addition, MIN returns -0 and MAX returns +0 for comparisons between -0 and +0.

dst - min(src0, src1)

Valid for R600 GPUs and later that support the IEEE flag.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = (v1[i] < v2[i]) ? v1[i] : v2[i];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MIN
control	29:16	MIN: 0 No special NaN rules. 1 IEEE-style NaN rules.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Matrix Multiply by Vector

Instructions **MMUL**

Syntax `mmul_matrix(`*op*`)` *`dst, src0, src1`*

Description Performs a matrix multiply of the generic matrices *src0* and *src1*. Data must be arranged in one of the formats indicated by the enumerated type ILMatrix(*matrix*). *src1* must be of type IL_REGTYPE_CONST_FLOAT or IL_REGTYPE_TEMP. Relative addressing and source modifiers of *src1* are applied to each row of the matrix beginning with *src1.*

Certain components are not returned by this instruction, depending on the value of *matrix.* Each component is returned only if the component mask field (*component_z_b* or *component_w_a*) of the IL_Dst_Mod token for that component is set to IL_MODCOMP_0 or IL_MODCOMP_1, in which case 0.0 or 1.0 is returned.

Valid for all GPUs.

Operation:

```
VECTOR v2 = EvalSource(src1+1); // Register number of src1 + 1
VECTOR v3 = EvalSource(src1+2); // Register number of src1 + 2
VECTOR v4 = EvalSource(src1+3); // Register number of src1 + 3
VECTOR v;
switch(matrix){
    case IL_MATRIX_4X4:
        dst.x = dot4(src0, src1)
        dst.y = dot4(src0, v2)
        dst.z = dot4(src0, v3)
        dst.w = dot4(src0, v4)
        break;
    case IL_MATRIX_4X3:
        dst.x = dot4(src0, src1)
        dst.y = dot4(src0, v2)
        dst.z = dot4(src0, v3)
        break;
    case IL_MATRIX_3X4:
        dst.x = dot3(src0, src1)
        dst.y = dot3(src0, v2)
        dst.z = dot3(src0, v3)
        dst.w = dot3(src0, v4)
        break;
    case IL_MATRIX_3X3:
        dst.x = dot3(src0, src1)
        dst.y = dot3(src0, v2)
        dst.z = dot3(src0, v3)
        break;
    case matrix == IL_MATRIX_3X2:
        dst.x = dot3(src0, src1)
        dst.y = dot3(src0, v2)
        break;
    }
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_MMUL
`control`	29:16	Any value of the enumerated type ILMatrix(*matrix*). See Table 6.16 on page 6-8.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Floating Point Modulo

Instructions **MOD**

Syntax `mod dst, src0, src1`

Description Performs the modulo operation. The sign remains intact (for example, `mod(-1.7,1.0)=-0.7`).

Valid for all GPUs.

Example:

src1 = src0 x $\lfloor$src1/srco$\rfloor$

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
{
    if(v2[i] == 0.0)
        v[i] = v1[i];
    else
        v[i] = v1[i] - v2[i] * trunc(v1[i]/v2[i]);
}
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MOD
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Move Data Between Registers

Instructions	**MOV**
Syntax	`mov` *dst*, *src0*

Description Places all four components of *src0* into the corresponding components of *dst*.

Valid for all GPUs.

Operation:

```
VECTOR v = EvalSource(src0);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MOV
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Floating Point Multiplication

Instructions	**MUL**
Syntax	`mul[_ieee]` *dst*, *src0*, *src1*

Description Multiplies two vectors.

dst - src0 ⊗ src1

Valid for all GPUs. For R600 GPUs, IEEE flag was added.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v1[i] * v2[i];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_MUL
control	29:16	MUL: 0 DX9 DP2-style multiple. 1 IEEE-style multiply.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Not Equal

Instructions	**`NE`**
Syntax	`nt` *`dst`*, *`src0`*, *`src1`*

Description

Compares two float vectors, *src0* and *src1*, for each component, and writes the result to *dst*. If the comparison is true, 0xFFFFFFFF is returned for that component; otherwise, 0x0000000 is returned. This instruction follows DX10 Floating Point Rules. Denorms are flushed before comparison (original source registers untouched). +0 equals -0. Comparison with NaN returns false.

Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for R600 GPUs and later.

Format

2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_NE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Compute Noise Value

Instructions **NOISE**

Syntax `noise_type(op) dst, src0`

Description Uses *src0* as the seed of a pseudo-random number generator to compute a noise value and places it in *dst*.

Valid for all GPUs.

Operation:

The values in *dst* has the following characteristics:

- They are in the range [-1.0, 1.0]
- Over many iterations, their average value is 0.0.
- The value is repeatable. That is, *dst* is always the same value given the same value of *src0*.
- The value is statistically invariant under rotation (no matter how the domain is rotated, it has the same statistical character).
- The value is statistically invariant under translation (no matter how the domain is translated, it has the same statistical character).
- The value is typically different under translation.
- Over many iterations, the values have a narrow band pass limit in frequency (the values have no visible features larger or smaller than a certain narrow-size range).
- Over many iterations, the values are C_1 continuous everywhere (the first derivative is continuous).

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_NOISE		
control	29:16	**Field Name**	**Bits**	**Description**
		noise	19:16	Any value of the enumerated type ILNoiseType(type). See Table 6.19 on page 6-9.
		reserved	29:20	Must be zero.
sec_modifier_present	30	Must be zero.		
pri_modifier_present	31	Must be zero.		

Related None.

Three or Four Component Normalization

Instructions **NRM**

Syntax `nrm[_nrm4|nrm3]_zeroop(op) dst, src0`

Description Normalize a 4D vector using three or four components. Each component of the vector in *src0* is divided by the square root of the sum of squares of the components in *src0*. This operation has the result of limiting each component to the range [-1.0, 1.0]. The result is placed in *dst*. If nrm4 is one, all four components of *src0* are used in normalization. If nrm4 is zero, only x, y, and z are used in the normalization calculation. If neither nrm3 or nrm4 is specified, nrm3 is used. If no zeroop value is provided, zeroop(fltmax) is used.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
Float  f;

If( nrm4 == 0)
{
    f = v1[0]*v1[0] + v1[1]*v[1] + v1[2]*v1[2];
else
{
    f = v1[0]*v1[0] + v1[1]*v[1] + v1[2]*v1[2] + v1[3]*v1[3];
}
if (f == 0.0)
{
    if(zeroop == IL_ZEROOP_0)
        f = 0.0;
    else if(zeroop == IL_ ZEROOP_FLT_MAX)
        f = FLT_MAX;
   else if(zeroop == IL_ ZEROOP_INFINITY)
        f = INFINITY;
    else if(zeroop == IL_ ZEROOP_INF_ELSE_MAX)
        f = INFINITY or FLT_MAX;  # Depends on IL Implementation
}
else
{
    f = 1.0/sqrt(f);
}
for (i=0; i < 4; i++)
    v[i] = v1[i] * f;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description		
code	15:0	IL_OP_NRM		
control	18:16	**Field Name**	**Bits**	**Description**
		reserved	17:16	Any value of the enumerated type ILZeroOp(*zeroop*). See Table 6.33 on page 6-22 when the first three components of *src0* are 0.0.
		nrm4	18	0: Normalize *src0*.xyz 1: Normalize *src0*.xyzw
reserved	31:19			

Related None.

Reduce Vector to [–π, π]

Instructions	**`PIREDUCE`**
Syntax	`pireduce` *`dst`*`,` *`src0`*

Description

All four components of the vector in *src0* are reduced to the range [-π, π].

Valid for all GPUs.

Example:

dst = (fract((src0/2π)) + 0.5) x 2 x π) - π P

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = (frac((v1[i]/(2*Pi))+0.5)* 2 * PI) – PI;
WriteResult(v, dst);
```

Format

1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_PIREDUCE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Project Vector

Instructions	**`POW`**
Syntax	`project_stage(n) dst, src_divComp (unknown)`

Description

Moves a value from `src` to `dst`.

An `IL_Src_Mod` token is required in this instruction. The `modifier_present` field of the `IL_Src` token must be set to 1.

This instruction cannot be used on a stage that has not been declared with a DCLPT instruction.

Inputs are *src0.w* and *src1.w*. Input is in radians. Results are broadcast to all four channels of *dst*.

The divComp source modifier must be set to `IL_DIVCOMP_UNKNOWN`, so the component used to divide is specified by `AS_TEX_PROJECTED_N(stage)`.

If the component to divide by is negative, the result of this instruction is undefined.

Valid for all GPUs.

Operation:

```
VECTOR v = EvalSource(src);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_POW
`control`	31:16	Specifies the texture/stage unit.

Related None.

X to the Power of Y

Instructions **POWER**

Syntax pow *dst*, *src0*, *src1*

Description Computes *src0*.w to the power of *src1*.w ($src0.w^{src1.w}$). By default, this instruction operates on *src0*.w and *src1*.w, but can operate on any component of either operand by swizzling it into the fourth component.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
If(v1[3] < 0.0)
{
    v[0] = v[1] = v[2] = v[3] = undefined;
}
else
{
    v[0] = v[1] = v[2] = v[3] = exp2(v2[3] * log2(v1[3]));
}
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_POW
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Reciprocal

Instructions	**RCP**
Syntax	`rcp_zeroop(op) dst, src0`

Description Computes the reciprocal of *src0*.w. By default, this instruction operates on *src0*.w and *src1*.w, but can operate on any component of either operand by swizzling it into the fourth component. If no zeroop value is provided, zeroop(fltmax) is used.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f = v1[3];
if (f == 0.0)
{
    if(zeroop == IL_ZEROOP_0)
        f = 0.0;
    else if(zeroop == IL_ZEROOP_FLT_MAX)
        f = FLT_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
        f = INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
        f = INFINITY or FLT_MAX;  # Depends on IL Implementation
}
else if (f != 1.0)
    f = 1/f;
v[0] = v[1] = v[2] = v[3] = f;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_RCP
control	17:16	The value of the enumerated type ILZeroOp(*zeroop*), which controls how this instruction behaves when the value of *src0* is 0.0. See Table 6.33 on page 6-22.
	29:18	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Compute Reflection Vector

Instructions	**REFLECT**
Syntax	`reflect_normalize` *dst*, *src0*, *src1*

Description

Computes the reflection direction of the vector in *src0* using the source vector in *src1*, and placing the direction vector in *dst*. `_normalize` specifies if *src1* is normalized before computing the reflection vector.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
float f = 2 * (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2] + v1[3]*v2[3]);
if(normalize)
{
    float fnorm = (v2[0]*v2[0] + v2[1]*v2[1] + v2[2]*v2[2] + v2[3]*v2[3]);
for (i=0; i < 4; i++)
   v[i] = fnorm * v1[i] - f * v2[i];
}
else
{
for (i=0; i < 4; i++)
   v[i] = v1[i] - f * v2[i];
}
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_REFLECT
control	16	0: Do not normalize *src1*. 1: Normalize *src1* before computing the reflection vector.
	29:17	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Round

Instructions **RND**

Syntax `rnd dst, src0`

Description Rounds the float value in each component of a floating point vertex (*src0*) to the nearest integer.

Valid for R600 GPUs and later.

Example:

$\lfloor src0 + 0.5 \rfloor$

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = floor(v1[i] + 0.5);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_RND
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related ROUND_NEAREST, ROUND_NEG_INF, ROUND_PLUS_INF, ROUND_ZERO.

Float Round to Nearest Even Float Integral

Instructions **ROUND_NEAREST**

Syntax `round_nearest dst, src0`

Description Rounds the float value in each component of *src0* to the nearest even integral floating-point.

Valid for all GPUs.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_ROUND_NEAR
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related RND, ROUND_NEG_INF, ROUND_PLUS_INF, ROUND_ZERO.

Float Round to –∞

Instructions	**`ROUND_NEG_INF`**
Syntax	`round_neginf` *`dst, src0`*
Description	Rounds the float values in each component of *src0* towards -∞. This is sometimes called a floor() instruction. Valid for R600 GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ROUND_NEG_INF
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RND, ROUND_NEAREST, ROUND_PLUS_INF, ROUND_ZERO.

Float Round to +∞

Instructions	**`ROUND_PLUS_INF`**
Syntax	`round_plusinf` *`dst, src0`*
Description	Rounds the float values in each component of *src0* towards ∞. This is sometimes called a ceil() instruction. Valid for R600 GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ROUND_PLUS_INF
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RND, ROUND_NEAREST, ROUND_NEG_INF, ROUND_ZERO.

Float Round to Zero

Instructions **`ROUND_ZERO`**

Syntax `round_z` *`dst, src0`*

Description Rounds the float values in each component of *src0* towards zero.

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ROUND_ZERO
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RND, ROUND_NEAREST, ROUND_NEG_INF, ROUND_PLUS_INF.

Reciprocal Square Root

Instructions **RSQ**

Syntax `rsq_zeroop(op) dst, src0`

Description Computes the reciprocal of the square root (positive only) of *src0*.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component of *src0*. If no zeroop value is provided, zeroop(fltmax) is used.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
float f = v1[3];
if (f == 0.0)
{
    if(zeroop == IL_ZEROOP_0)
        f = 0.0;
    else if(zeroop == IL_ZEROOP_FLT_MAX)
        f = FLT_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
        f = INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
        f = INFINITY or FLT_MAX;  # Depends on IL Implementation
}
else if (f < 0.0)
{
    f = undefined;
}
else
{
    f = 1.0/(float)sqrt(f);
}
v[0] = v[1] = v[2] = v[3] = f;
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_RSQ
`control`	29:16	Any value of the enumerated type ILZeroOp(*zeroop*). See Table 6.33 on page 6-22.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RSQ_VEC.

Component-wise Reciprocal Square Root

Instructions **`RSQ_VEC`**

Syntax `rsq_vec` *`dst`*`,` *`src0`*

Description Computes the reciprocal of the square root of each component of *src0*.

Valid for R600 GPUs and later.

Example:

dst = $1.0\sqrt{\text{src0}}$

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_RSQ_VEC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related RSQ.

Set on Comparison

Instructions **`SET`**

Syntax `set_relop(`*`op`*`)` *`dst`*`,` *`src0`*`,` *`src1`*

Description Compares each component of *src0* with the corresponding component of *src1*. The type of comparison performed is dictated by relop(*op*). If the comparison *src0*.{x|y|z|w} relop(*op*) *src1*.{x|y|z|w} evaluates TRUE, the result is 1.0; otherwise, the result is 0.0.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = (v1[i] relop v2[i]) ? 1.0 : 0.0;
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SET
`control`	29:16	Any value of the enumerated type ILRelOp(*relop*). See Table 6.23 on page 6-18.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Compute Sign

Instructions	SGN
Syntax	sgn *dst*, *src0*

Description

Computes the sign of each component of *src0*.

Valid for all GPUs.

Operation:

```
VECTOR v = EvalSource(src0);
for (i=0; i < 4; i++)
{
    if (v[i] < 0.0)
        v[i] = -1.0;
    else if (v[i] == 0.0)
        v[i] = 0.0;
    else
        v[i] = 1.0;
}
WriteResult(v, dst);
```

Format

1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_SGN
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related

None.

Sine (sin)

Instructions	**`SIN`**
Syntax	`sin` *`dst, src0`*

Description

Computes the sine of *src0*.w. *src0*.w is in radians for trigonometric functions. *src0*.w must be within the range [-π,π] for each function; otherwise, the results are undefined. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. The maximum absolute error is 0.002.

Valid for R600 GPUs and later.

Example:

dst = sin(src0)

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SIN
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related COS, COS_VEC, SIN_VEC, SINCOS.

Compute Sine and Cosine

Instructions **SINCOS**

Syntax `sincos dst, src0`

Description Computes sine and cosine values of *src0*.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. The maximum absolute error is 0.002. The 32-bit floating point results are returned in *dst*.x (COS) and *dst*.y (SIN). *src0*.w must be in the range [-π, π]; otherwise, the results are undefined. *dst*.z and *dst*.w are not written by this instruction; however, the if component_z_b or component_w_a field of the IL_Dst_Mod token is set to IL_MODCOMP_0 or IL_MODCOMP_1, the *dst.z* and *dst.w* are written. That is, *dst.z* and *dst.w* can be set to 0.0 or 1.0 if IL_MODCOMP_0 or IL_MODCOMP_1 is used on the component_z_b or component_w_a field of the IL_Dst_Mod token.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v;
v[0] = cos(v1[3]);
v[1] = sin(v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_SINCOS
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related COS, COS_VEC, SIN, SIN_VEC.

Component-Wise Sine

Instructions **SIN_VEC**

Syntax `sin_vec dst, src0`

Description Computes the sine of each component of *src0* in radians. The maximum absolute error is 0.0008 in the range [-100*π, 100*π].

Valid for R600 GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_SIN_VEC
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related COS, SIN, SINCOS, COS_VEC.

Square Root

Instructions	**`SQRT`**
Syntax	`sqrt` *`dst, src0`*
Description	Computes the square root of *src0*.w. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. If *src0*.w is less than zero, the result is undefined. The result is approximate. Valid for all GPUs.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SQRT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related SQRT_VEC.

Component-Wise Square Root

Instructions	**`SQRT_VEC`**
Syntax	`sqrt_vec` *`dst, src0`*
Description	Computes the square root of each component of *src0*. If a component of *src0* is less than zero, the result for that component is undefined. The result is approximate. Valid for R600 GPUs and later. Example: dst = $\sqrt{\text{src0}}$
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SQRT_VEC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related SQRT.

Floating Point Subtraction

Instructions **SUB**

Syntax `sub dst, src0, src1`

Description Subtracts each component of *src0* from the corresponding component of *src1*. No carry or borrow beyond the 32-bit values of each component is performed.

Valid for all GPUs.

Example:

dst = src0 - src1

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v2 = EvalSource(src1);
VECTOR v;
for (i=0; i < 4; i++)
    v[i] = v1[i] - v2[i];
WriteResult(v, dst);
```

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_SUB
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Tangent (tan)

Instructions **TAN**

Syntax `tan dst, src0`

Description Computes the tangent of *src0*.w. *src0*.w is in radians. *src0*.w must be within the range [-π, π] for each function; otherwise, the results are undefined. By default, this instruction operates on *src0*.w, but can operate on any component by swizzling it into the fourth component. The maximum absolute error is 0.002.

Valid for all GPUs.

Operation:

```
VECTOR v1 = EvalSource(src0);
VECTOR v
v[0] = v[1] = v[2] = v[3] = tan(v1[3]);
WriteResult(v, dst);
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_TAN
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Transpose a 4x4 Matrix

Instructions **TRANSPOSE**

Syntax `transpose dst, src0`

Description Transposes the rows and columns of the 4x4 matrix indicated by *src*.0.

Valid for all GPUs.

Operation:

```
VECTOR vsrc0 = EvalSource(src0);
VECTOR vsrc1 = EvalSource(src0+1); # Register number of src0 + 1
VECTOR vsrc2 = EvalSource(src0+2); # Register number of src0 + 2
VECTOR vsrc3 = EvalSource(src0+3); # Register number of src0 + 3
VECTOR vdst0;
VECTOR vdst1;
VECTOR vdst2;
VECTOR vdst3;
vdst0[0] = vsrc0[0];
vdst0[1] = vsrc1[0];
vdst0[2] = vsrc2[0];
vdst0[3] = vsrc3[0];
vdst1[0] = vsrc0[1];
vdst1[1] = vsrc1[0];
vdst1[2] = vsrc2[1];
vdst1[3] = vsrc3[1];
vdst2[0] = vsrc0[2];
vdst2[1] = vsrc1[2];
vdst2[2] = vsrc2[2];
vdst2[3] = vsrc3[2];
vdst3[0] = vsrc0[3];
vdst3[1] = vsrc1[3];
vdst3[2] = vsrc2[3];
vdst3[3] = vsrc3[3];
WriteResult(vdst0, dst);
WriteResult(vdst1, dst+1); # Register number of dst + 1
WriteResult(vdst2, dst+2); # Register number of dst + 2
WriteResult(vdst3, dst+3); # Register number of dst + 3
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
code	15:0	IL_OP_TRANSPOSE
control	29:16	Must be the enumerated type ILMatrix(*IL_MATRIX_4X4*). See Table 6.16 on page 6-8.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

7.12 Double-Precision Instructions

Add Two Doubles

Instructions **DADD**

Syntax dadd *dst*, *src0*, *src1*

Description Abs and negate modifiers can be used on the source. Output is either wz or yx. Output is the correct IEEE result.

	src1						
	-inf	**-F**	**-0**	**+0**	**+F**	**+inf**	**NaN**
scr0							
-inf	-inf	-inf	-inf	-inf	-inf	NaN	NaN
-F	-inf	-F	*src0*	*src0*	+-F or +-0	+inf	NaN
-0	-int	*src1*	-0	+0	*src1*	+inf	NaN
+0	-inf	*src1*	+0	+0	*src1*	+inf	NaN
+F	-inf	+-F or +-0	*src0*	*src0*	-F	+inf	NaN
+inf	NaN	+inf	+inf	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Valid for all GPUs that support double floating-point operations.

Format 2-input 1-output.

Opcode

Field Name	**Bits**	**Description**
code	15:0	IL_OP_D_ADD
control	29:16	Must be zero.
sec_modifier_present	30	Must be zero.
pri_modifier_present	31	Must be zero.

Related None.

Divide Two Doubles

Instructions **`D_DIV`**

Syntax `ddiv` *`dst`*`,` *`src0`*`,` *`src1`*

Description Abs and negate modifiers can be used on the source. Output is either wz or yx. Output is the correct IEEE result.

dst = src0.xy / src1.xy

Valid for all GPUs that support double floating-points.

Format 2-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_DIV
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Double Equal Compare

Instructions **`D_EQ`**

Syntax `deq` *`dst`*, *`src0`*, *`src1`*

Description Component-wise compares two vectors using a float comparison that follows floating point rules. If *src0*.{xy|zw} and the corresponding component pair in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, the corresponding component of *dst* is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

A compare with NaN returns FALSE.

Valid for R670 GPUs and later that support double floating-points.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_EQ
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Return Fractional Part of a Double

Instructions **D_FRAC**

Syntax `dfrac dst, src0`

Description Returns the fractional part of the source in range [0.0 – 1.0). Abs and negate modifiers can be used. Output.yx or output wz is set to the double result.

x	-inf	-F	-1.0	-denorm	-0	+0	+denorm	+1.0	+F	+inf	NaN
result	NAN	[+0.0,+1.0)*	+0	+0	d+0	+0	+0	+0	[+0.0,+1.0)*	NAN	NaN

Valid for all GPUs that support double floating-points.

Format 1-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_FRAC
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Split Double into Fraction and Exponent

Instructions **D_FREXP**

Syntax `dfrexp dst, src0`

Description Output x is zero. Output y is the exponent as an integer. Output wz is the mantissa in range (-1.0,-0.5][0.5,1.0). Abs or negate modifiers can be used on the source. If the input is +-0, all four results are zero. The sign is preserved for the wz output. If the input is a NaN, y is set to -1, and wz is set to a NaN = {s, 0x7ff, 1'b1, mant[50:0]} // QNaN. If the input is an +-iff, y is set to -1, and wz is set t0 a NaN = 0xfff8000000000000.

double	-inf/+inf	-0/+0	-denorm/+denorm	NaN
output wz	NAN	{sign,0}	{sign,0}	d
output y	0xFFFFFFFF	0	0	0xFFFFFFFF

Valid for all GPUs that support double floating-points.

Format 1-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_FREXP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Double Greater-Than or Equal Compare

Instructions **`D_GE`**

Syntax dge *dst*, *src0*, *src1*

Description Component-wise compares two vectors using a float comparison that follows floating point rules. If *src0*.{xy|zw} and the corresponding component pair in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, the corresponding component of *dst* is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for R670 GPUs and later. Boundary cases are:

	src1								
src0	**-inf**	**-F**	**-denorm**	**-0**	**+0**	**+denorm**	**+F**	**+inf**	**NaN**
-inf	T	F	F	F	F	F	F	F	F
-F	T	T/F	F	F	F	F	F	F	F
-denorm	T	T	T	T	T	T	F	F	F
-0	T	T	T	T	T	T	F	F	F
+0	T	T	T	T	T	T	F	F	F
+denorm	T	T	T	T	T	T	F	F	F
+F	T	T	T	T	T	T	T/F	F	F
+inf	T	T	T	T	T	T	T	T	F
NaN	F	F	F	F	F	F	F	F	F

Format 2-input, 1-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_OP_D_GE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Pack an EXP and Mantissa into a Double

Instructions	**`D_LDEXP`**
Syntax	`dldexp` *`dst`*, *`src0`*, *`src1`*
Description	Puts an exp and a mantissa into a double. The computation is result = source1 * 2 source. Abs and negate modifiers can be used. Output.yx or output.wz are set to the double result. NaN in either input produces a NaN. Overflow produces inf. Underflow produces 0. Src0 is a double (-1.0d < src0.xy < 1.0d). Src1.x is an integer from -1024 to +1024. Since the second source src1 is an integer exponent, only the neg modifier is allowed. The value of *src1.x* (post swizzle) is used. Valid for all GPUs that support double floating-points.
Format	2-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_LDEXP
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Double Less-Than Compare

Instructions	**`D_LT`**
Syntax	`dlt` *`dst`*, *`src0`*, *`src1`*

Description

Component-wise compares two vectors using a float comparison that follows floating point rules. If *src0*.{xy\|zw} and the corresponding component pair in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, the corresponding component of *dst* is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

Valid for all GPUs from the R6XX series that support double floating-point.

Boundary cases are:

	src1								
src0	**-inf**	**-F**	**-denorm**	**-0**	**+0**	**+denorm**	**+F**	**+inf**	**NaN**
-inf	F	T	T	T	T	T	T	T	F
-F	F	T/F	T	T	T	T	T	T	F
-denorm	F	F	T/F	T	T	T	T	T	F
-0	F	F	F	F	F	T	T	T	F
+0	F	F	F	F	F	T	T	T	F
+denorm	F	F	F	F	F	T/F	T	T	F
+F	F	F	F	F	F	F	T/F	T	F
+inf	F	F	F	F	F	F	F	F	F
NaN	F	F	F	F	F	F	F	F	F

Format

2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_LT
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related

None.

Component-Wise Double-Precision Maximum

Instructions	**`D_MAX`**
Syntax	`dmax dst, src0, src1`
Description	Abs and negate modifiers can be used on the source. Output is either xy or zw and is IEEE correct. `dst = src0.xy >= src1.xy ? src0.xy : src1.xy` `>=` is used so that if (min(x,y) = x ,then max (x,y) = y. The output is either xy or zw. Nan has special handling: if one source operand is Nan, the other is returned. This corresponds to IEEE 754 rev. 2008. Valid for Evergreen GPUs and later that support double-precision floating-points.
Format	2-input, 1-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_OP_D_MAX
`control`	31:16	Must be zero.

Related None.

Component-Wise Double-Precision Maximum

Instructions	**`D_MIN`**
Syntax	`dmin dst, src0, src1`
Description	Abs and negate modifiers can be used on the source. Output is either wz or yx and IEEE correct. `dst = src0.xy < src1.xy ? src0.xy : src1.xy` `>=` is used so that if (min(x,y) = x, then max (x,y) = y. The output is either xy or zw. Nan has special handling: if one source operand is Nan, the other is returned. This corresponds to IEEE 754 rev. 2008. Valid for Evergreen GPUs and later that support double-precision floating-points.
Format	2-input, 1-output.

Opcode

Field Name	**Bits**	**Description**
`code`	15:0	IL_OP_D_MIN
`control`	31:16	Must be zero.

Related None.

Move Double-Precision Value

Instructions	**`D_MOV`**
Syntax	`dmov dst, src0`

Description

Moves a double precision value from one register to another.

The result goes into either `dst.xy` or `dst.zw`.

src0 is a single 32-bit integer input (0 or non zero). No modifiers are allowed on *src0*. The input is the first (post swizzle) channel of *src0*.

Valid for Evergreen GPUs and later that support double floating-point.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_MOV
`control`	31:16	Must be zero.

Related D_MOVC.

Double-Precision Conditional Move

Instructions	**`D_MOVC`**
Syntax	`dmovc dst, src0,src1,src2`

Description

Conditionally moves a double-precision value from one register to another.

Result goes into either `dst.xy` or `dst.zw`.

Valid for Evergreen GPUs and later that support double floating-point.

Operation:

```
if (src0.x == 0) { /* This is integer comparison */
  dst = src1;
} else {
  dst = src2;
}
```

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_MOVC
`control`	31:16	Must be zero.

Related D_MOV.

Multiply Two Doubles

Instructions **D_MUL**

Syntax `dmul` *dst*, *src0*, *src1*

Description Abs and negate modifiers can be used on the source. Output is either wz or yx. Output is the correct IEEE result.

src0 \ ***src1***	**-inf**	**-F**	**-1.0**	**-0**	**+0**	**+1.0**	**+F**	**+inf**	**NaN**
-inf	+inf	+inf	+inf	NaN	NaN	inf	-inf	-inf	NaN
-F	+inf	+F	-*src0*	+0	-0	*src0*	-F	-inf	NaN
-1.0	+int	-*src1*	+1.0	+0	-0	-1.0	-*src1*	-inf	NaN
-0	NaN	+0	+0	+0	-0	-0	-0	NaN	NaN
+0	NaN	-0	-0	-0	+0	+0	+0	NaN	NaN
+1.0	-inf	*src1*	-1.0	-0	+0	+1.0	*src1*	+inf	NaN
+F	-inf	-F	-*src0*	-0	+0	*src0*	+F	+inf	NaN
+inf	-inf	-inf	-inf	NaN	NaN	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Valid for all GPUs that support double floating-points.

Format 2-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_MUL
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Double Multiply and Add

Instructions **D_MULADD**

Syntax `dmad dst, src0, src1, src2`

Description Computes: *src0* * *src1* + *src2*. Abs and negate can be used on the sources. The output in either yz or wz is the result. The compiler determines if this is identical to separate MULs and ADD based on the chipset. On HD38XX systems, the operation truncates internally, so the result is not identical to a MUL followed by an ADD.

From Evergreen GPUs and later, hardware maps this instruction to FMA64.

dst = src0.xy x src1.xy + src2.xy

Valid for all GPUs that support double floating-points.

Format 3-input 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_MULADD
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Double Not Equal Compare

Instructions **D_NE**

Syntax `dne dst, src0, src1`

Description Component-wise compares two vectors using a float comparison that follows floating point rules. If *src0*.{xy|zw} and the corresponding component pair in *src1* satisfy the comparison condition, the corresponding component of *dst* is set to TRUE and returns 0xFFFFFFFF; otherwise, the corresponding component of *dst* is set to FALSE and returns 0x00000000. Primary and secondary opcode modifiers are not permitted, and no additional control options are supported. Thus, the `pri_modifier_present` and `sec_modifier_present` fields must be zero.

A compare with NaN returns TRUE.

Valid for R670 GPUs and later that support double floating-points.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_NE
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related None.

Reciprocal

Instructions	**D_RCP**
Syntax	`drcp_zeroop(op) dst, src`

Description

The first two bits of control are set to a value of the enumerated type ILZeroOp(zeroop), which controls how this instruction behaves when the value of `src` is 0.0.

It computes reciprocal value of the 4th channel of src. By default, this instruction operates on `src.w`, but can operate on any component by swizzling it into the fourth channel.

If no zeroop value is provided, zeroop(fltmax) is used.

Valid for Evergreen GPUs and later that support double floating-point.

Operation:

```
VECTOR v1 = EvalSource(src);
VECTOR v;
Double d = v1[2,1];
if (d == 0.0)
{
    if(zeroop == IL_ZEROOP_0)
            d = 0.0;
    else if(zeroop == IL_ZEROOP_FLT_MAX)
            d = DBL_MAX;
   else if(zeroop == IL_ZEROOP_INFINITY)
            d = INFINITY;
    else if(zeroop == IL_ZEROOP_INF_ELSE_MAX)
            d = INFINITY or dbl_MAX; # Depends on IL Implementation
}
else if (d != 1.0)
    d = 1/d;
v[1,0] = d;
WriteResult(v, dst);
```

Format

1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_RCP
`control`	31:16	Must be zero.

Related

None.

Reciprocal Square Root

Instructions **`D_RSQ`**

Syntax `drsq_zeroop (op) dst, src`

Description Computes the reciprocal square root (positive only). By default, this instruction operates on `src.w`, but can operate on any component by swizzling it into the fourth channel.

If no zeroop value is provided, zeroop(fltmax) is used.

Valid for Evergreen GPUs and later that support double floating-point.

Operation:

```
if (d > 0.0} {
    v = 1.0d/dsqrt(d);
}
else if (d == 0.0) {
    v = apply ZeroOp rule to d
}
WriteResult(v, dst);
return;
}
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_RSQ
`control`	31:16	Value of type ILZeroOp(zeroop), which controls how this instruction behaves when the value of `src` is 0.0.

Related None.

Square Root

Instructions **`D_SQRT`**

Syntax `dsqrt dst, src0`

Description Computes the double square root of `src0.xy`. The result goes into either `dst.xy` or `dst.zw`..

dst = $\sqrt{\text{src0.xy}}$

dst = |src0.xy|

The result is approximate. This opcode is not part of DX11.

Valid for Evergreen GPUs and later that support double floating-points.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_D_SQRT
`control`	31:16	Must be zero.

Related None.

7.13 Multi-Media Instructions

Align Bit Data for Video

Instructions	**`BitAlign`**
Syntax	`bitalign dst, scr0, src1, src2`

Description	Aligns bit data for video. This is a special instruction for multi-media video.
	`dst = ((src0,src1) >> src2[4:0]) & 0xFFFFFFFF`
	This corresponds to:
	`.src0 << (32 - src2[4:0]) | src1 >> src[4:])`
	The 32-bit result is replicated to all four vector output slots.
	Valid for Evergreen GPUs and later.

Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_BIT_ALIGN
`control`	31:16	Must be zero.

Example If R1.y has 0x1234567, and R1.x has 0x89ABCDEF, then:

```
dcl_literal l10, 1, 11, 24, 0
Bitalign r0.x, r1.y, r1.x, l10.x results in r0.x having 0xA4D5E6F7
Bitalign r0.x, r1.y, r1.x, l10.y results in r0.x having 0xACF13579
Bitalign r0.x, r1.y, r1.x, l10.z results in r0.x having 0x23456789
```

Related None.

Align Byte Data for Video

Instructions	**`ByteAlign`**
Syntax	`bytealign dst, scr0, src1, src2`
Description	Aligns byte data for video. This is a special instruction for multi-media video. `dst = ((src0 << (32 - 8 * src2.x)) | (src1 >> 8 * src2.x)).` `src2.x` must be 0, 1, 2, or 3. The 32-bit result is replicated to all four vector output slots. Valid for Evergreen GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_BYTE_ALIGN
`control`	31:16	Must be zero.

Related None.

Pack Four Floats Into a Special Packed 4 Unsigned INT Format

Instructions **`F_2_u4`**

Syntax `f_2_u4 dst, scr0`

Description Packs four floats into a special packed four unsigned integer format. This is a special instruction for multi-media video. The 32-bit result is replicated to all four vector output slots.

Inputs outside range 255.999f give undefined results.

Valid for Evergreen GPUs and later.

Operation:

```
*
dst.xyzw = (flt_to_uint(src0.w &0xff) << 24))
          +(flt_to_uint(src0.z) & 0xFF) << 16))
          +(flt_to_uint(src0.y) & 0xFF) << 8))
          +(flt_to_uint(src0.x) & 0xFF)))
```

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_F_2_u4
`control`	31:16	Must be zero.

Related None.

Sum of Absolute Differences

Instructions **`SAD`**

Syntax `sad dst, scr0, src1, src2`

Description `Sad8(src, src1)` forms the sum of absolute differences, treating *src0*, *src1* as a vector of eight-bit unsigned integers. This is a special instruction for multi-media video. The result = `sad(src0,src1)+ src2` (done for each component).

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SAD
`control`	31:16	Must be zero.

Related SAD_HI, SAD4.

Sum of Absolute Differences

Instructions	**`SAD_HI`**
Syntax	`sadhi dst, scr0, src1, src2`
Description	`Sad8(src, src1)` forms the sum of abs differences treating src0, src1 as a vector of eight-bit unsigned integers, using the high bits. This is a special instruction for multi-media video. The result = `sad(src0,src1)<< 16 + src2` (done for each component). Valid for Evergreen GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SAD_HI
`control`	31:16	Must be zero.

Related SAD, SAD4.

Sum of Absolute Differences

Instructions	**`SAD4`**
Syntax	`sad4, scr0, src1, src2`
Description	`Sad8(src, src1)` forms the sum of absolute differences, treating `src0` and `src1` as a vector of eight-bit unsigned integers. This is a special instruction for multi-media video. `dst.xyzw = sad8(src0.x,src1.x) + sad8(src0.y,src1.y) + sad8(src0.z,src1.z) + sad8(src0.w, src1.w) + r2.x .` The 32-bit result is replicated to all four vector output slots. Valid for Evergreen GPUs and later.
Format	3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_SAD4
`control`	31:16	Must be zero.

Related SAD, SAD_HI.

LERP for Multi-Media Format

Instructions **U4LERP**

Syntax `u4lerp dst, scr0, src1, src2`

Description Performs linear interpolation on four unsigned char values. This is a special instruction for multi-media video. The result =

```
dst = ((src0[31:24] + src1[31:24] + src2[24]) >> 1) << 24 +
      ((src0[23:16] + src1[23:16] + src2[16]) >>1) << 16 +
      ((src0[15:8] + src1[15:8] + src2[8]) >> 1) << 8 +
      ((src0[7:0] + src1[7:0] + src2[0]) >> 1);
```

The 32-bit result is replicated to all four vector output slots.

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_U4LERP
`control`	31:16	Must be zero.

Related None.

Unpack First byte of a packed unsigned int into a float

Instructions **Unpack0**

Syntax `unpack0 dst, scr0`

Description Unpacks the first byte of a packed unsigned integer into a float. This is a special instruction for multi-media video. The result = `uint2flt (src0 & 0xFF)`.

The 32-bit result is replicated to all four vector output slots.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_UNPACK0
`control`	31:16	Must be zero.

Related Unpack1, Unpack2, Unpack3.

Unpack Second Byte of a Packed Unsigned Int Into a Float

Instructions	**`Unpack1`**
Syntax	`unpack1 dst, scr0`
Description	Unpacks the third byte of a packed unsigned integer into a float. This is a special instruction for multi-media video. The result = `uint2flt (src0 >> 8 & 0xFF)`. The 32-bit result is replicated to all four vector output slots. Valid for Evergreen GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_UNPACK1
`control`	31:16	Must be zero.

Related Unpack0, Unpack2, Unpack3.

Unpack Third Byte of a Packed Unsigned Int Into a Float

Instructions	**`Unpack2`**
Syntax	`unpack2 dst, scr0`
Description	Unpacks the third byte of a packed unsigned integer into a float. This is a special instruction for multi-media video. The result = `uint2flt (src0 >> 16 & 0xFF)`. The 32-bit result is replicated to all four vector output slots. Valid for Evergreen GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_UNPACK2
`control`	31:16	Must be zero.

Related Unpack0, Unpack1, Unpack3.

Unpack Fourth Bytes of a Packed Unsigned Int Into a Float

Instructions	**`Unpack3`**
Syntax	`unpack3 dst, scr0`
Description	Unpacks the fourth byte of a packed unsigned integer into a float. This is a special instruction for multi-media video. The result = `uint2flt (src0 >> 24 & 0xFF)`. The 32-bit result is replicated to all four vector output slots. Valid for Evergreen GPUs and later.
Format	1-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_UNPACK3
`control`	31:16	Must be zero.

Related Unpack0, Unpack1, Unpack2.

7.14 Miscellaneous Special Instructions

Embed Comment in Stream

Instructions	**`COMMENT`**
Syntax	`<comment string>`
Description	Allows a comment to be passed into the IL. Must be four-byte aligned character including the null-terminator. Valid for all GPUs.
Format	

Ordinal	Token		
1	IL_Opcode token with code set to IL_OP_COMMENT		
2	**Field Name**	**Bits**	**Description**
	length	15:0	The length of the comment in DwordS. This is a 1-based integer representing the number of tokens following this token. (1 indicates that one Dword follows this token.)
	Reserved	31:16	Reserved; must be zero.

Related None.

Null Operation

Instructions	**`NOP`**
Syntax	`nop`
Description	No operation performed. Valid for all GPUs.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_NOP
`control`	31:16	Must be zero.

Related None.

7.15 Evergreen GPU Series Memory Controls

Assign a New Slot in Append Buffer and Return the Address of the Slot

Instructions	**`APPEND_BUF_ALLOC`**
Syntax	`append_buf_alloc_id(n) dst.x`

Description

The append buffer is essentially a UAV buffer (must be raw or structured) with the Append flag set by the driver at binding time. The Append buffer lets a shader write data to memory in a compacted and unordered way.

`Append_buf_alloc_id(n)` assigns a new empty slot in the append buffer with ID(n) for each active work-item, and returns the position/index of the assigned slot to `dst`. (Internally, this instruction increments a hidden counter associated with an append buffer, and returns the original value of the counter.)

The returned index in `dst` can be used by subsequent instructions to compute the address for UAV instructions. For example, the returned value can be multiplied by 4 to get the byte address for `UAV_RAW_STORE`.

`dst` must have a mask of `.x`, `.y`, `.z`, or `.w`.

A single append buffer can not be used for both `Append_buf_alloc` and `Append_buf_consume`.

This instruction is only for pixel or compute shaders.

Valid for Evergreen GPUs and later.

Format 0-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_APPEND_BUF_ALLOC
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `append_buf_alloc_id(1) r2.x`

Related None.

Consume an existing data slot in append buffer and return the address of the slot

Instructions **`APPEND_BUF_CONSUME`**

Syntax `append_buf_consume_id(n) dst.x`

Description Append buffer is essentially a UAV buffer (must be raw or structured) with the Append flag set by the driver at binding time. The append buffer lets a shader write data to memory in a compacted and unordered way.

`Append_buf_consume_id(n)` consumes an existing data slot in the append buffer with ID(n) for each active work-item, and returns the position/index of the data slot to `dst`. (Internally, this instruction decrements a hidden counter associated with an append buffer, and returns the original value of the counter.)

The returned index in `dst` can be used by subsequent instructions to compute the address for UAV instructions. For example, the returned value can be multiplied by 4 to get the byte address for `UAV_RAW_LOAD`.

`dst` must have a mask of `.x`, `.y`, `.z`, or `.w`.

A single append buffer can not be used for both `Append_buf_alloc` and `Append_buf_consume`.

This instruction is only for pixel or compute shaders.

Valid for Evergreen GPUs and later.

Format 0-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_APPEND_BUF_CONSUME
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `append_buf_consume_id(1) r3.x`

Related None.

Declare Arena UAV

Instructions	**`DCL_ARENA_UAV`**
Syntax	`dcl_arena_uav_id(`*n*`)`
Description	Declare an arena UAV with an ID(n). An arena UAV is a special kind of UAV buffer that supports multiple access modes (Dword/short/byte). It can be accessed by loads and stores in units of Dword, short, and byte. Note: This instruction must be declared for memory objects that have an alignment of 256 bytes or greater. Valid only for Evergreen and Northern Islands GPUs.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_ARENA_UAV
	`id`	25:16	Resource ID.
	`reserved`	31:26	Must be zero.

Example 1	`dcl_arena_uav_id(5)`
Related	UAV_ARENA_LOAD, UAV_ARENA_STORE, UAV_LOAD, UAV_RAW_LOAD, UAV_STRUCT_LOAD, UAV_STORE, UAV_RAW_STORE, UAV_STRUCT_STORE

Declare a Typeless LDS

Instructions	**`DCL_LDS`**
Syntax	`dcl_lds_id(id) n`
Description	Declares an LDS with ID and a size. The size, n, is in bytes and must be four-byte aligned. The ID is used in the subsequent LDS instructions in the shader. This instruction is used only in a compute shader. Valid for R7XX GPUs and later. The ATI Radeon™ HD 4000 series supports only a single ID, which must be zero. The ATI Radeon™ HD 5000 series The ATI Radeon™ HD 4000 series allows multiple IDs, but the sum of all LDS allocations must be <= 32k bytes.
Format	0-input, 0-output, 1 additional token: opcode is `il_dcl_lds`; control is the ID.
Opcode	

Token	Field Name	Bits	Description
1	`code`	15:0	IL_ DCL_LDS
	`ID`	29:16	Must be zero.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example	`Dcl_lds_id(0) 1024`
Related	None.

Declare a Raw SRV Buffer

Instructions	**`DCL_RAW_SRV`**
Syntax	`dcl_raw_srv_id(n)`
Description	Declares a raw SRV (shader resource view) buffer, and assigns it a number, `n`. The SRV is read-only input buffer that can be used by all shader types. The contents of the buffer have no type. Operations performed on the memory implicitly assume the associated type. Valid for R7XX GPUs and later.
Format	0-input, 0-output, 0 additional token.
Opcode	

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_RAW_SRV
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example	`dcl_raw_srv_id(1)`
Related	None.

Declare Typeless UAV

Instructions	**`DCL_RAW_UAV`**
Syntax	`dcl_raw_uav_id(n)`
Description	Declares a raw UAV and assigns it a unique ID. The associated memory must be a buffer. Operations performed on the memory implicitly assume the associated type. Restriction for the HD4000 series: only one UAV is allowed. Valid for R700 GPUs and later.
Format	0-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_RAW_UAV
	`ID`	29:16	Resource ID.
	reserved	31:30	Must be zero.

Example `dcl_raw_uav_id(0)`

Related None.

Declare a Structured LDS

Instructions	**`DCL_STRUCT_LDS`**
Syntax	`dcl_struct_lds_id(id) n1, n2`
Description	Declares a structured LDS memory, which is viewed as an array of struct. The size of each struct is n1; the length of array is n2. The instruction is followed by two Dwords, the first is a stride size, the second is a struct-count. Stride-size n1 is in bytes, but must be four-byte aligned. The ID is used in the subsequent LDS instructions in the shaders. This instruction is used only in a compute shader. Valid for R7XX GPUs and later. The ATI Radeon™ HD 4000 series allows multiple IDs, but the sum of all LDS allocations must be <= 32k bytes. Restriction for the ATI Radeon™ HD 4000 series only: n2 must be equal to the total number of work-items in the work-group.
Format	0-input, 0-output, 2 additional tokens: opcode is `il_dcl_struct_lds`; control is the id.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_STRUCT_LDS
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `Dcl_struct_lds_id(0) 1024,2`

Related None.

Declare Structured SRV Buffer with a Stride

Instructions	**`DCL_STRUCT_SRV`**
Syntax	`dcl_struct_srv_id(n) k`
Description	Declares a structured SRV (shader resource view) buffer with ID and a stride. The stride, `k`, is in bytes and must be >0 and a multiple of 4. The SRV is a read-only input buffer that can be used by all shader types. The contents of the buffer have no type. Valid for R7XX GPUs and later.
Format	0-input, 0-output, 1additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_STRUCT_SRV
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the stride in bytes.		

Example	`dcl_struct_srv_id(1) 32`
Related	None.

Declare Structured UAV With a Stride

Instructions	**`DCL_STRUCT_UAV`**
Syntax	`dcl_struct_uav_id(n) k`
Description	Declares a structured UAV with ID and a stride. The stride, `k`, is in bytes. It must be >0 and a multiple of 4. Similar to raw UAV, the contents of the structured UAV has no type. Some implementations may run more efficiently if strides are used. Restriction for the HD4000 series: only one UAV is allowed. Valid for R700 GPUs and later. R700 GPUs can only have one UAV. Evergreen GPUs can use 0 to 7 UAV in the AMD IL.
Format	0-input, 0-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_STRUCT_UAV
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the stride in bytes.		

Example	`dcl_struct_uav_id(1) 32`
Related	None.

Declare Typed UAV

Instructions **`DCL_UAV`**

Syntax `dcl_uav_id(n)_type(pixtexusage)_fmtx(fmt)`

Description Declares a typed UAV, assigns it to a number, and specifies its dimension.

Identifies the dimension/type of the UAV, as a buffer, texture1D, texture1Darray, texture2D, texture2Darray, or texture3D. Use the ILPixTexUsage enumeration for this field.

To specify the data format; use the ILElementFormat enumeration for this field. Supported data formats are: UNIT, SINT, FLOAT, UNORM, SNORM, SRGH, and MIXED.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_UAV
	`ID`	19:16	Resource ID.
	`fmtx`	23:20	Data format. Use the ILPixTexUsage enumeration (Section 6.22, on page 6-18).
	`type`	29:24	Dimension type. Use the ILElementFormat enumeration (Section 6.7, on page 6-3).
	reserved	31:30	Must be zero.

Example `dcl_uav_id(1)_type(1d)_fmtx(float)`

Related None.

7.16 LDS Instructions

See page 7-6 for common functionality of the LDS instuctions.

Add to LDS With Signed Integer Add

Instructions **`LDS_ADD`**

Syntax `lds_add_id(id) src0, src1`

Description Address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] += src1.x`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] += src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_ADD
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

AND of Src1 and LDS

Instructions **`LDS_AND`**

Syntax `lds_and_id(id) src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = and(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = and(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_AND
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Conditional Store Into (LDS) memory

Instructions **`LDS_CMP`**

Syntax `lds_cmp_id(id) src0, src1, src2`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `if (lds[src0x] == src1.x) {lds[src0.x] = src2.x;}`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `If (lds[(src0.x*lds_stride + src0.y)/4] == src1.x) {lds[(src0.x*lds_stride + src0.y)/4] = src2.x}`

Valid for Evergreen GPUs and later.

Format 3-input, 0-output; opcode is `il_op_lds_cmp`; control is the resource ID.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_CMP
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Integer Decriment of LDS

Instructions	**`LDS_DEC`**
Syntax	`lds_dec_id(id) src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = and(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = and(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_DEC
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Integer Increment of LDS

Instructions	**`LDS_INC`**
Syntax	`lds_inc_id(id) src0, src1`

Description

Reads a scalar (32-bit) value at a given LDS address, conditionally increment the value by 1, and write the new value back. The address is in bytes, but must be a multiple of 4. The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

```
lds[byte_addr] = lds[byte_addr] >= src1.x ? 0 : lds[byte_addr] + 1
```

where `byte_addr = src0.x` for typeless LDS, and `src0.x*lds_byte_stride+src0.y` for structured LDS.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_INC
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read From LDS

Instructions	**`LDS_LOAD`**
Syntax	`lds_load_id(id) dst, src0`

Description

Returns a scalar 32-bit value at a given LDS address.

The address is in bytes, but the two least significant bits must be zero. The load result is a Dword. If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

If LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

`dst.x = lds[byte_addr]`

where `byte_addr = src0.x` for typeless LDS, and `src0.x*lds_byte_stride+src0.y` for structured LDS.

Valid for R700 GPUs and later.

Format 1-input, 1-output;

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_LOAD
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read Byte From LDS

Instructions	**`LDS_LOAD_BYTE`**
Syntax	`lds_load_byte_id(id) dst, src0`

Description

The address is in bytes. The load result is a byte, sign-extended to a Dword. If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `dst.x = lds[src0.x]` (using integer add)

If LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x * lds_stride + src0.y)]`

Valid for Evergreen GPUs and later.

Format 1-input, 1-output;

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_LOAD_BYTE
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read Short From LDS

Instructions **`LDS_LOAD_SHORT`**

Syntax `lds_load_short_id(id) dst, src0`

Description The address is in bytes. The load result is a short, sign-extended to a Dword. If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `dst.x = lds[src0.x]` (using integer add)

If LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x * lds_stride + src0.y)/2]`

Valid for Evergreen GPUs and later.

Format 1-input, 1-output;

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_LOAD_SHORT
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read Unsigned Byte From LDS

Instructions **`LDS_LOAD_UBYTE`**

Syntax `lds_load_ubyte_id(id) dst, src0`

Description The address is in bytes. The load result is an unsigned byte, extended to a Dword. If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `dst.x = lds[src0.x]` (using integer add)

If LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x * lds_stride + src0.y)]`

Valid for Evergreen GPUs and later.

Format 1-input, 1-output;

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_LOAD_UBYTE
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read Unsigned Short From LDS

Instructions	**`LDS_LOAD_USHORT`**
Syntax	`lds_load_ushort_id(id) dst, src0`
Description	The address is in bytes. The load result is an unsigned short, sign-extended to a Dword. If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group. • `dst.x = lds[src0.x]` (using integer add) If LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes. • `Dst.x = lds[(src0.x * lds_stride + src0.y)/2]` Valid for Evergreen GPUs and later.
Format	1-input, 1-output;
Opcode	

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_LOAD_USHORT
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related	None.

Read Four Dword Vector from LDS

Instructions **`LDS_LOAD_VEC`**

Syntax `lds_load_vec_id(n) dst.mask, src0, src1`

Description Reads a vector of four words from LDS memory, and writes one to four Dwords into `dst`. For the HD5000 series, LDS with ID n can be declared as either raw or structured. For the HD4000 series, LDS with ID n must have been declared as structured.

For a structured LDS, `src0.x` (post swizzle) specifies the index of the structure. `Src1.x` (post swizzle) specifies the offset within the structure. Src1 is a separate argument for the offset because it is often a literal. The offset is in bytes and must be four-bytes aligned.

For a raw LDS, `src0.x` (post swizzle) specifies the address in bytes. It must be four-bytes aligned. `Src1` is not used.

Four consecutive Dwords are read from LDS memory. One to four Dwords are written to `dst`, depending on the `dst` mask.

Instructions with out of range addresses return 0, unless the offset is out of bound, in which case the return value is undefined.

Valid for R700 GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_LDS_LOAD_VEC
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example `Lds_load_vec_id(1) r2.x_zw, r0.x, r1.y`

Related None.

Max of Src1 and LDS

Instructions	**`LDS_MAX`**
Syntax	`lds_max_id(id) src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = max(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = max(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_MAX
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Min of Src1 and LDS, Signed Integer Compare

Instructions	**`LDS_MIN`**
Syntax	`lds_min_id(id) src0, src1`

Description

Address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = min(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = min( lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_MIN
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Atomic Integer Mask OR of Src1 and LDS

Instructions	**`LDS_MSKOR`**
Syntax	`lds_mskor_id(id) src0, src1, src2`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless:

```
a = src0.x/4
lds[a] = (lds[a]a & ( ~src1.x )) | src2.x
```

If the LDS is declared as struct:

```
a = src0.x * lds_stride + src0.y)/y
lds[a] = (lds[a]a & ( ~src1.x )) | src2.x
```

Valid for Evergreen GPUs and later.

Format 3-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_MSKOR
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Integer OR of Src1 and LDS

Instructions	**`LDS_OR`**
Syntax	`lds_or_id(id) src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = or(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = or(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_OR
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Add Src1

Instructions **`LDS_READ_ADD`**

Syntax `lds_read_add_resource(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] += src1.x`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] += src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_ADD
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then ADD With Src1

Instructions **`LDS_READ_AND`**

Syntax `lds_read_and_resource(id) dst, src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = and(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = and(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_AND
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Compare Src1 With Read LDS memory, If Equal Replace With Src2

Instructions **`LDS_READ_CMP_XCHG`**

Syntax `lds_read_cmp_xchg_resource(id) dst, src0, src1, src2`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `if (lds[src0x] == src1.x) {lds[src0.x] = src2.x;}`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `If (lds[(src0.x*lds_stride + src0.y)/4] == src1.x) { lds[(src0.x*lds_stride + src0.y)/4] = src2.x }`

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_CMP_XCHG
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Take Max

Instructions	**`LDS_READ_MAX`**
Syntax	`lds_read_max_resource(id) dst, src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, src0.x (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = max(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = max(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_MAX
`conrtrol`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Take Mins

Instructions **LDS_READ_MIN**

Syntax `lds_read_min_resource(id) dst, src0, src`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, src0.x (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = min(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = min(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_MIN
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then OR With Src1

Instructions	**`LDS_READ_OR`**
Syntax	`lds_read_or_resource(id) dst, src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = or(lds[src0.x], src1.x)`

If the LDS is declared as struct, src0.x (after swizzle) specifies the index into the array, src0.y (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = or(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_OR
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Reverse Subtract

Instructions **`LDS_READ_RSUB`**

Syntax `lds_read_rsub_resource(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = src1.x – lds[src0.x]`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = src1.x - lds[(src0.x*lds_stride + src0.y)/4]`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_RSUB
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Subtract Src1

Instructions	**`LDS_READ_SUB`**
Syntax	`lds_read_sub_resource(id) dst, src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] -= src1.x`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] -= src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_SUB
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Take Unsigned Max

Instructions **`LDS_READ_UMAX`**

Syntax `lds_read_umax_resouce(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, src0.x (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = umax(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = umax(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_UMAX
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then Take Unsigned Min

Instructions **`LDS_READ_UMIN`**

Syntax `lds_read_umin_resouce(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, src0.x (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = umin(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = umin(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_UMIN
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS and Exchange With Src1

Instructions **`LDS_READ_XCHG`**

Syntax `lds_read_xchg_resource(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The address must be aligned to a Dword (the lower two bits of the address must be zero). The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `dst.x = lds[src0.x]`
- `lds[src0.x]  = src1.x`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_XCHG
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Read LDS, Then XOR With Src1

Instructions **`LDS_READ_XOR`**

Syntax `lds_read_xor_resource(id) dst, src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `Dst.x = lds[src0.x]`
- `lds[src0.x] = xor(lds[src0.x], src1.x)`

If the LDS is declared as struct, src0.x (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `Dst.x = lds[(src0.x*lds_stride + src0.y)/4]`
- `lds[(src0.x*lds_stride + src0.y)/4] = xor(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_READ_XOR
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Reverse Subtract Signed Integer from LDS

Instructions **`LDS_RSUB`**

Syntax `lds_rsub_id(id) src0.component, src1.component`

Description Address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = src1.x – lds[src0.x]`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = src1.x - lds[(src0.x*lds_stride + src0.y)/4]`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_RSUB
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Write LDS

Instructions **`LDS_STORE`**

Syntax `lds_store_id(id) src0, src1`

Description Address is in bytes, but the two least significant bits must be zero. The store data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = src1.x` (using integer add)

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- lds[(src0.x*lds_stride + src0.y)/4] = src1.x

Valid for R700 GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_STORE
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Write LDS Byte

Instructions **`LDS_STORE_BYTE`**

Syntax `lds_store_byte_id(id) src0, src1`

Description Address and store data are in bytes.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = src1.x` (using integer add)

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x * lds_stride + src0.y)] = src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_STORE_BYTE
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Write LDS Short

Instructions	**`LDS_STORE_SHORT`**
Syntax	`lds_store_short_id(id) src0, src1`

Description

Address is in bytes, and store data is a short.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = src1.x` (using integer add)

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x * lds_stride + src0.y)/2] = src1.x`

Valid for Evergreen GPUs and later.

Format

2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_STORE_SHORT
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related

None.

Write Up to Four Dwords to LDS

Instructions **`LDS_STORE_VEC`**

Syntax

```
lds_store_vec_id(n) dst.mask, src0, src1, src2
```

Description Writes up to four Dwords to LDS memory. For the Evergreen GPUs, an LDS with ID n can be declared as either raw or structured. For the R7XX GPUs, an LDS with ID n must have been declared as structured.

The `dst` must be of type `IL_REGTYPE_GENERIC_MEM`, it is used only for the mask.

For a structured LDS, `src0.x` (post swizzle) specifies the index of the structure. `Src1.x` (post swizzle) specifies the offset within the structure. `Src1` is a separate argument for the offset because it is often a literal. The offset is in bytes and must be four-bytes aligned. Restriction for the HD4000 series only: `src0.x` (post swizzle) must be equal to `Thread_Id_In_Group_Flattened.x`. `Src1.x` (post swizzle) must be a literal value known at compile time.

For a raw LDS, `src0.x` (post swizzle) specifies the address in bytes. It must be four-bytes aligned. `Src1` is not used.

`Src2.xyzw` (post swizzle) specifies the source data.

Depending on the `dst` mask, one to four Dwords are written to LDS memory. The `dst.mask` can be only one of: `.x`, `.xy`, `.xyz`, `.xyzw`.

Instructions with out of range addresses write nothing to LDS memory, unless the offset is out of bound, in which case an undefined value is written to the LDS.

Valid for R700 GPUs and later.

Format 3-input, 1-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_LDS_STORE_VEC
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example

```
Lds_store_vec_id(1) mem.xyz_, r0.x, r1.y, r2.xyzw
```

Related None.

Subtract Signed Integer from LDS

Instructions **`LDS_SUB`**

Syntax `lds_sub_id(id) src0.component, src1.component`

Description Address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x]  -= src1.x`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4]  -= src1.x`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_SUB
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Unsigned Max of Src1 and LDS

Instructions **`LDS_UMAX`**

Syntax `lds_umax_id(id) src0, src1`

Description The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x]  = umax(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4]  = umax(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_UMAX
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Unsigned Min of Src1 and LDS

Instructions	**`LDS_UMIN`**
Syntax	`lds_umin_id(id) src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = umin(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = umin(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_UMIN
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Unsigned Integer XOR of Src1 and LDS

Instructions	**`LDS_XOR`**
Syntax	`lds_xor_id(id) src0, src1`

Description

The address is in bytes, but the two least significant bits must be zero. The data is a Dword.

If the LDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `lds[src0.x] = xor(lds[src0.x], src1.x)`

If the LDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `lds[(src0.x*lds_stride + src0.y)/4] = xor(lds[(src0.x*lds_stride + src0.y)/4], src1.x)`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_LDS_XOR
`control`	19:16	Resource ID.
reserved	31:20	Must be zero.

Related None.

Random Access Read From a Raw SRV (Returns up to Four Dwords)

Instructions	**`SRV_RAW_LOAD`**
Syntax	`srv_raw_load_id(n) dst, src0.x`

Description

Read from a raw SRV. The SRV with ID n must have been declared as a raw SRV buffer.

`src0.x` (post swizzle) specifies the address of the raw SRV buffer. The address is in bytes and must be four-bytes aligned.

Four consecutive 32-bit components are read from SRV(n), starting at address `src0.x` (post-swizzle).

One to four Dwords are written to `dst`, depending on the `dst` mask.

DX11 allows an output swizzle on the instruction. IL requires an additional move if the swizzle is used.

An instruction with an out-of-range address returns 0.

The SRV is a read-only input buffer that can be used by all shader types.

Valid for R7XX GPUs and later.

Format 1-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_SRV_RAW_LOAD
	`control`	23:16	Resource ID.
	reserved	27:24	Must be zero.
		28	flag for indexed resource ID
	reserved	31:29	Must be zero.

Examples

```
srv_raw_load_id(1) r1.x_z_, r0.x
srv_raw_load_ext_id(1) r1.x_z, r0.x, r2.y
```

Related None.

Random Access Read From a Structured SRV Buffer (Returns up to Four Dwords)

Instructions	**`SRV_STRUCT_LOAD`**
Syntax	`srv_struct_load_id(n) dst, src0.xy`

Description

Read from a structured SRV. SRV with ID n must have been declared as a structured SRV buffer.

`src0.xy` (post swizzle) specifies the index of the structure and the offset within the structure, respectively. The offset is in bytes and must be four-bytes aligned.

Four consecutive 32-bit components are read from SRV(n) at the address specified by `src0.xy` (post-swizzle).

One to four Dwords are written to `dst`, depending on the `dst` mask.

Output swizzle is not allowed.

An instruction with an out-of-range address returns 0, unless the offset is out-of-bounds, in which case the return value is undefined.

The SRV is a read-only input buffer that can be used by all shader types.

There is a limit on the number of SRV buffers exposed in a shader (currently 128).

Indexed resource ID is supported on Evergreen GPUs or later. This is specified by the `ext` keyword. With this option on, an extra input is needed as indexed input (see example below).

Valid for R7XX GPUs and later.

Format

1-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_SRV_STRUCT_LOAD
	`control`	23:16	Resource ID.
	reserved	27:24	Must be zero.
	`flag`	28	Flag for indexed resource ID.
	reserved	31:29	Must be zero.

Example

```
srv_struct_load_id(1) r1.x_zw, r0.xy
srv_struct_load_ext_id(3) r1.x_zw, r0.xy, r2.x
```

Related

None.

Atomic Add to UAV (Integer add)

Instructions	**`UAV_ADD`**
Syntax	`uav_add_id(n) src0, src1.x`

Description

Atomic single component integer add to UAV: `uav[src0] += src1.x`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as R32 UINT or SINT.

`src0` provides the address, which must be a multiple of fours. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the array index.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in src0 is updated atomically by `iadd(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

Valid for Evergreen GPUs and later.

Format

2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_ADD
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example

`uav_add_id(1) r0.xy, r1.z`

Related

None.

Atomic Bitwise AND to UAV

Instructions	**`UAV_AND`**
Syntax	`uav_and_id(n) src0, src1.x`
Description	Atomic single component bitwise AND to UAV: `uav[src0] &= src1.x` A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT. `src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes. If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array. `src1.x` (post-swizzle) provides the 32-bit Dword to be computed. The 32-bit UAV memory specified by the address in `src0` is updated atomically by `AND(uav[src0], src1.x)`. Nothing is returned. An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. If the shader invocation is inactive, nothing is written to the UAV surface. Valid for Evergreen GPUs and later.
Format	2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_AND
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example	`uav_and_id(1) r0.xy, r1.z`
Related	None.

Read From Arena UAV

Instructions **UAV_ARENA_LOAD**

Syntax `uav_arena_load_id(n)[_cached]_size(...) src0.x, src1.x`

Description Reads one Dword/short/byte to an arena UAV. The data size is specified by the keyword `size(...)`. For example: `_size(short)`. A UAV with `id(n)` must have been declared as an arena UAV.

Src0.x (post-swizzle) specifies the address in bytes.

The *dst* register must have a mask of `.x` or `.y` or `.z` or `.w`.

Depending on the data size specified, one Dword/short/byte is read from the UAV and written to the *dst* register.

The load address alignment must be aligned to the size of the data it reads.

Valid only for Evergreen and Northern Islands GPUs.

Example

```
uav_arena_load_id(1)_size(Dword) r0.x, r1.x
uav_arena_load_id(1)_size(short) r0.w, r1.x
uav_arena_load_id(1)_size(byte) r0.z, r1.y
uav_arena_load_id(1)_size(dword)_cached r0.x, r1.y
```

Format 1-input, 1-output, no additional tokens.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_UAV_ARENA_LOAD
	control	25:16	Resource ID.
	data_size	27:26	The value for data size must be of type IL_LOAD_STORE_DATA_SIZE.
	cached	29:28	Specifies whether load is forced through the cache, which must be of type UAV_READ.
	Reserved	30	Reserved; must be zero.
	atomic	31	Reserved.

Related UAV_ARENA_STORE, UAV_LOAD, UAV_RAW_LOAD, UAV_STRUCT_LOAD, UAV_STORE, UAV_RAW_STORE, UAV_STRUCT_STORE

Write to Arena UAV

Instructions	**`UAV_ARENA_STORE`**
Syntax	`uav_arena_store id(n) size(...) dst.x, src0.x`

Description

Read one Dword/short/byte from an arena UAV. The data size is specified by the keyword `size(...)`. For example: `_size(short)`. A UAV with `id(n)` must have been declared as an arena UAV.

Src0.x (post-swizzle) specifies the address in bytes.

Src1.x (post-swizzle) provides 32-bit data, which is converted to the specified data size (Dword/short/byte) before being written to the arena UAV.

Instructions with out-of-range addresses write nothing to the UAV surface.

Valid only for Evergreen and Northern Islands GPUs.

Example

```
uav_arena_store_id(1)_size(Dword) r0.x, r1.y
uav_arena_store_id(1)_size(short) r0.y, r1.w
uav_arena_store_id(1)_size(byte) r0.x, r1.x
```

Format 1-input, 1-output, no additional tokens.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_ARENA_STORE
	`control`	25:16	Resource ID.
	`data_size`	27:26	The value for data size must be of type IL_LOAD_STORE_DATA_SIZE.
	`cached`	29:28	Reserved.
	`Reserved`	30	Reserved; must be zero.
	`atomic`	31	Reserved.

Related UAV_ARENA_LOAD, UAV_LOAD, UAV_RAW_LOAD, UAV_STRUCT_LOAD, UAV_STORE, UAV_RAW_STORE, UAV_STRUCT_STORE

Atomic Bitwise Compare and Write to UAV

Instructions **`UAV_CMP`**

Syntax `uav_cmp_id(n) src0, src1.x, src2.x`

Description Atomic single component bitwise compare and write to UAV,
`if(uav[src0]==src2.x) uav[src0]=src1.x`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provide 32-bit Dword to be written to UAV.

`src2.x` (post-swizzle) provide 32-bit Dword to be compared (bitwise compare with `uav[src0])`.

The 32-bit UAV memory specified by the address in `src0` is overwritten by `src1.x` if the compared values are identical. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 3-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_CMP
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_cmp_id(1) r0.xy, r1.x, r2.x`

Related None.

Random Access Read from a Typed UAV (Returns up to Four Dwords)

Instructions	**`UAV_LOAD`**
Syntax	`uav_load_id(n) dst, src0`

Description

Reads one element (up to four components) from a typed UAV, converts it, and returns it to `dst`.

A UAV with ID n must have been declared as typed.

`src0`, with possible swizzle, specifies the address (in elements). The number of components used for the address depends on the UAV dimension.

For example, for texture1D arrays, `src0.x` (post-swizzle) provides the buffer address; `src0.y` (post-swizzle) provides the index/offset of the array. If the value is out of the range of available arrays (indices [0... (Array size-1)]), the load returns 0.

`src0.yzw` (post-swizzle) is ignored for buffers and texture1D (non-array). `src0.zw` (post-swizzle) is ignored for texture1D arrays and texture 2D.

Based on the format of UAV(n), up to four components are read and converted to 32-bit per component. The converted values then are written to `dst`, according to the `dst` mask.

DX11 allows an output swizzle on the instruction. IL requires an additional move if the swizzle is used. Also, DX11 has a limitation that the data format must be one of: R32_UINT, SINT, or FLOAT. IL does not have this limitation.

Valid for Evergreen GPUs and later.

Format

1-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_LOAD
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example

```
uav_load_id(1) r1.x_z_, r0.zw
```

Related

None.

Atomic Signed Integer Max to UAV

Instructions **`UAV_MAX`**

Syntax `uav_max_id(n) src0, src1.x`

Description Atomic single component signed integer max to UAV:
`uav[src0] = max(uav[src0], src1.x)`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `min(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_MAX
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_max_id(1) r0.xy, r1.z`

Related None.

Atomic Signed Integer Min to UAV

Instructions	**`UAV_MIN`**
Syntax	`uav_min_id(n) src0, src1.x`

Description

Atomic single component signed integer min to UAV: `uav[src0] = min(uav[src0], src1.x)`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `min(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format

2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_MIN
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example

`uav_min_id(1) r0.xy, r1.z`

Related

None.

Atomic Bitwise OR to UAV

Instructions **`UAV_OR`**

Syntax `uav_or_id(n) src0, src1.x`

Description Atomic single component bitwise OR to UAV: `uav[src0] |= src1.x`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `OR(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_OR
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_or_id(1) r0.xy, r1.z`

Related None.

Random Access Read from a Typeless UAV (Returns up to Four Dwords)

Instructions **UAV_RAW_LOAD**

Syntax `uav_raw_load_id(n) dst, src0.x`

Description Reads from a typeless UAV. A UAV with ID n must have been declared as raw. `src0.x` (post swizzle) specifies the address of the raw UAV buffer. The address is in bytes and must be four-bytes aligned.

Four consecutive 32-bit components are read from UAV(n), starting at address `src0.x` (post-swizzle).

One to four Dwords are written to `dst`, depending on the `dst` mask.

DX11 allows an output swizzle on the instruction. IL requires an additional move if the swizzle is used.

An instruction with an out-of-range address returns 0.

Restriction for the HD4000 (R700 GPUs) series: only one UAV is allowed.

Valid for R700 GPUs and later.

Format 1-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_RAW_LOAD
	`control`	25:16	Resource ID.
	`cached`	27:26	Specifies whether load is forced through the cache, which must be of type UAV_READ.
	`Reserved`	30:28	Must be zero.
	`reserved_arena_atomic`	31	Reserved.

Example
```
uav_raw_load_id(1) r1.x_z_, r0.x
uav_raw_load_id(1)_cached r1.x_z_, r0.x
```

Related None.

Random Access Write to a Typeless UAV (Writes up to Four Dwords)

Instructions	**`UAV_RAW_STORE`**
Syntax	`uav_raw_store_id(n) dst, src0.x, src1`

Description

Writes up to four Dwords to a typeless UAV. A UAV with ID n must have been declared as raw.

The `dst` must be of type IL_REGTYPE_GENERIC_MEM; it is used only as a mask.

`src0.x` (post-swizzle) provides the address for the raw UAV(n), which is in bytes and four-bytes aligned.

`src1.xyzw` (post-swizzle) provides data (four 32-bit Dwords) for writing to UAV(n).

Depending on the `dst` mask, one to four Dwords are written to UAV(n), starting at the address specified by `src0.x`.

An instruction with an out-of-range address writes nothing to the UAV surface.

Dx11 has a limitation that the `dst` mask must be one of the following: x, xy, xyz, xyzw (other masks are invalid). IL does not have this limitation (gap is allowed).

Restriction for the HD4000 series: only one UAV is allowed.

Valid for R700 GPUs and later. For R7XX GPUs, only a single UAV is allowed.

Format

2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_RAW_STORE
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example

```
uav_raw_store_id(1) mem.x_z_, r0.x, r1.xyyy
```

Related

None.

Atomic Integer Add to UAV, Return Old Value in UAV

Instructions **`UAV_READ_ADD`**

Syntax `uav_read_add_id(n) dst.x, src0, src1.x`

Description Atomic single component integer add to UAV: `uav[src0] += src1.x`; then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by address `src0` is updated atomically by `iadd(uav[src0], src1.x)`.

The 32-bit value in the UAV before the computation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_ADD
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_add_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Bitwise AND to UAV, Return Old Value in UAV

Instructions **`UAV_READ_AND`**

Syntax `uav_read_and_id(n) dst.x, src0, src1.x`

Description Atomic single component bitwise AND to UAV: `uav[src0] &= src1.x`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `AND(uav[src0], src1.x)`.

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_AND
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_and_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Bitwise Compare and Write to UAV, Return Old Value in UAV

Instructions	**`UAV_READ_CMP_XCHG`**
Syntax	`uav_read_cmp_xchg_id(n) dst.x, src0, src1.x, src2.x`

Description

Atomic single component bitwise compare and write to UAV:
`if (uav[src0] == src2.x) uav[src0]=src1.x.`

Whether the compared values are identical or not, the old value is always returned in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides a 32-bit Dword to be written to the UAV.

`src2.x` (post-swizzle) provides a 32-bit Dword to be compared (bitwise compare with `uav[src0]`).

The 32-bit UAV memory specified by the address in `src0` is overwritten by `src1.x` if the compared values are identical.

A 32-bit value in the UAV before the operation is always returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 3-input, 1-output, 0 additional token

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_CMP_XCHG
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_cmp_xchg_id(1) r3.x, r0.xy, r1.x, r2.x`

Related None.

Atomic Signed Integer Max to UAV, Return Old Value in UAV

Instructions	**`UAV_READ_MAX`**
Syntax	`uav_read_max_id(n) dst.x, src0, src1.x`

Description

Atomic single component signed integer max to UAV:
`uav[src0] = max(uav[src0], src1.x)`, then returns old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `max(uav[src0], src1.x)`.

A A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_MAX
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_max_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Signed Integer Min to UAV, Return Old Value in UAV

Instructions **`UAV_READ_MIN`**

Syntax `uav_read_min_id(n) dst.x, src0, src1.x`

Description Atomic single component signed integer min to UAV:
`uav[src0] = min(uav[src0], src1.x)`, then returns old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `min(uav[src0], src1.x)`.

A A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_MIN
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_min_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Bitwise OR to UAV, Return Old Value in UAV

Instructions	**`UAV_READ_OR`**
Syntax	`uav_read_or_id(n) dst.x, src0, src1.x`

Description

Atomic single component bitwise OR to UAV: `uav[src0] |= src1.x`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `OR(uav[src0], src1.x)`.

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_OR
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_or_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Integer Reverse Subtract to UAV, Return Old Value in UAV

Instructions **`UAV_READ_RSUB`**

Syntax `uav_read_rsub_id(n) dst.x, src0, src1.x`

Description Atomic single component integer reverse subtract to UAV:
`uav[src0] = (src1.x - uav[src0])`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

A 32-bit UAV memory specified by address `src0` is updated atomically by
`(src1.x - uav[src0])`.

A 32-bit value in the UAV before the computation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_RSUB
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_rsub_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Integer Subtract to UAV, Return Old Value in UAV

Instructions **`UAV_READ_SUB`**

Syntax `uav_read_sub_id(n) dst.x, src0, src1.x`

Description Atomic single component integer subtract to UAV: `uav[src0] -= src1.x`; then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

A 32-bit UAV memory specified by address `src0` is updated atomically by `(uav[src0] - src1.x)`.

A 32-bit value in the UAV before the computation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_SUB
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_sub_id(1) r2.x, r0.xy, r1.z`

Related None.

UAV Atomic Read Unsigned Decrement

Instructions **`UAV_READ_UDEC`**

Syntax `uav_read_udec_id(n) dst, src0, src1`

Description Atomic single component unsigned integer read from UAV; then,
`uav[src0] = ( uav[src0] == 0 || uav[src0] > src1 ) ? src1 : uav[src0] - 1 .`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by address `src0` is updated atomically by
`( uav[src0] == 0 || uav[src0] > src1 ) ? src1 : uav[src0] - 1`.

The 32-bit value in the UAV before the computation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_UDEC
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Single Component Unsigned Integer Read from UAV

Instructions **`UAV_READ_UINC`**

Syntax `uav_read_uinc_id(n) dst, src0, src1`

Description Atomic single component unsigned integer read from UAV; then,
`uav[src0] = ( uav[src0] >= src1 ) ? 0 : uav[src0] + 1.`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in src0 is updated atomically by
`uav[src0] = ( uav[src0] >= src1 ) ? 0 : uav[src0] + 1.`

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_UINC
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Unsigned Integer Max to UAV, Return Old Value in UAV

Instructions **`UAV_READ_UMAX`**

Syntax `uav_read_umax_id(n) dst.x, src0, src1.x`

Description Atomic single component unsigned integer max to UAV:
`uav[src0] = umax(uav[src0], src1.x)`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `umax(uav[src0], src1.x)`.

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_UMAX
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_umax_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Unsigned Integer Min to UAV, Return Old Value in UAV

Instructions **UAV_READ_UMIN**

Syntax `uav_read_umin_id(n) dst.x, src0, src1.x`

Description Atomic single component unsigned integer min to UAV:
`uav[src0] = umin(uav[src0], src1.x)`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `umin(uav[src0], src1.x)`.

A A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_UMIN
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_umin_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Write/Exchange Value to UAV, Return Old Value in UAV

Instructions **`UAV_READ_XCHG`**

Syntax `uav_read_xchg_id(n) dst.x, src0, src1.x`

Description Atomic single component write of src1 to UAV: `uav[src0] = src1.x`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by the value of `src1.x`.

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_XCHG
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_xchg_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Bitwise XOR to UAV, Return Old Value in UAV

Instructions **`UAV_READ_XOR`**

Syntax `uav_read_xor_id(n) dst.x, src0, src1.x`

Description Atomic single component bitwise XOR to UAV: `uav[src0] = XOR(uav[src0], src1.x)`, then returns the old value in `uav[src0]`.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in src0 is updated atomically by `XOR(uav[src0], src1.x)`.

A 32-bit value in the UAV before the operation is returned to `dst`, which must have a mask of `.x`, `.y`, `.z`, or `.w`.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV. An undefined value is returned to `dst`.

If the shader invocation is inactive, nothing is written to the UAV surface, and an undefined value is returned to `dst`.

Valid for Evergreen GPUs and later.

Format 2-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_READ_XOR
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_read_xor_id(1) r2.x, r0.xy, r1.z`

Related None.

Atomic Integer Reverse-Subtract to UAV

Instructions **`UAV_RSUB`**

Syntax `uav_rsub_id(n) src0, src1.x`

Description Atomic single component integer reverse-subtract to UAV:
`uav[src0] = (src1.x – uav[src0])`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by (`src1.x - uav[src0]`). Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_RSUB
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_rsub_id(1) r0.xy, r1.z`

Related None.

Random Access Write to a Typed UAV (Writes up to Four Components)

Instructions	**`UAV_STORE`**
Syntax	`uav_store_id(n) src0, src1`
Description	Writes one element (up to four components) to a typed UAV. A UAV with ID n must have been declared as typed. `src0`, with possible swizzles, specifies the address (in elements). The number of components used for the address depends on the UAV dimenion. For example, for texture1D arrays, `src0.x` (post-swizzle) provides the buffer address; `src0.y` (post-swizzle) provides the index/offset of the array. `src0.yzw` (post-swizzle) is ignored for buffers and texture1D (non-array). `src0.zw` (post-swizzle) is ignored for texture1D arrays and texture 2D. `src1.xyzw` (post-swizzle) provides four 32-bit Dwords of data. Each component of the data is converted according to the format of UAV(n). One to four components of the converted data are written to the UAV, based on the format of the surface. An instruction with an out-of-range address writes nothing to the UAV surface. Valid for Evergreen GPUs and later.
Format	2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_STORE
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example	`uav_store_id(1) r0.zw, r1.xyyy`
Related	None.

Random Access Read from a Structured UAV (Returns up to Four Dwords)

Instructions **`UAV_STRUCT_LOAD`**

Syntax `uav_struct_load_id(n) dst, src0.xy`

Description Reads from a structured UAV. A UAV with ID n must have been declared as structured.

`src0.xy` (post swizzle) specifies the index of the structure and the offset within the structure, respectively. The offset is in bytes and must be four-bytes aligned.

Four consecutive 32-bit components are read from UAV(n) at the address specified by `src0.xy` (post-swizzle).

1-4 Dwords will be written to dst register depending on the dst mask

DX11 allows an output swizzle on the instruction. IL requires an additional move if the swizzle is used.

An instruction with an out-of-range address returns 0 except that, if the offset is out of bound, the return value is undefined.

Restriction for the HD4000 series: only one UAV is allowed.

Valid for R700 GPUs and later. For R7XX, only a single UAV is allowed.

Format 1-input, 1-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_STRUCT_LOAD
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_struct_load_id(1) r1.x_zw, r0.xy`

Related None.

Random Access Write to a Structured UAV (Writes up to Four Dwords)

Instructions	**`UAV_STRUCT_STORE`**
Syntax	`uav_struct_store_id(n) dst, src0.xy, src1`

Description

Writes up to four Dwords to a structured UAV. A UAV with ID n must have been declared as structured.

The `dst` must be of type IL_REGTYPE_GENERIC_MEM; it is used only for as a mask.

Post-swizzle `src0.xy` specifies the index of the structure and the offset within the structure, respectively. The offset is in bytes and four-bytes aligned.

`src1.xyzw` (post-swizzle) provides data (four 32-bit Dwords) for writing to UAV(n).

Depending on the `dst` mask, one to four Dwords are written to UAV(n), starting at the address specified by `src0.xy`.

An instruction with an out-of-range address writes nothing to the UAV surface, unless the offset is out of bounds, in which case an undefined value is written to the UAV.

Dx11 has a limitation that the `dst` mask must be one of the following: x, xy, xyz, xyzw (other masks are invalid). IL does not have this limitation (gap is allowed).

Restriction for the HD4000 series: only one UAV is allowed.

Valid for R7XX GPUs and later.

Format

2-input, 1-output, 0 additional token

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_STRUCT_STORE
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example

```
uav_struct_store_id(1) mem.x_z_, r0.xy, r1.zwxy
```

Related

None.

Atomic Integer Subtract to UAV

Instructions **`UAV_SUB`**

Syntax `uav_sub_id(n) src0, src1.x`

Description Atomic single component integer subtract to UAV: `uav[src0] -= src1.x`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of the struct index and the offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address; `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `isub(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_SUB
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_sub_id(1) r0.xy, r1.z`

Related None.

UAV Atomic Unsigned Decrement

Instructions **UAV_UDEC**

Syntax `uav_udec_id(n) src0, src1.x`

Description Computes `dst = (dst == 0 | (dst > src1))?src1:dst - 1`, where dst is the UAV addressed by src0.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by:

```
uav[src0] = (uav[src0] == 0 || uav[src0] > src1
0 ? src1 : dst - 1.
```

Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_UDEC
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

UAV Atomic Unsigned Increment

Instructions **`UAV_UINC`**

Syntax `uav_uinc_id(n) src0, src1.x`

Description Computes `dst = (dst >= src1)?0:dst + 1`, where dst is the UAV addressed by src0.

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `uav[src0] = (uav[src0] >= src1) ? 0 : uav[src0] + 1`.

Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_UINC
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Unsigned Integer Max to UAV

Instructions **`UAV_UMAX`**

Syntax `uav_umax_id(n) src0, src1.x`

Description Atomic single component unsigned integer max to UAV:
`uav[src0] = umax(uav[src0], src1.x)`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `umax(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_UMAX
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_umax_id(1) r0.xy, r1.z`

Related None.

Atomic Unsigned Integer Min to UAV

Instructions **`UAV_UMIN`**

Syntax `uav_umin_id(n) src0, src1.x`

Description Atomic single component unsigned integer min to UAV:
`uav[src0] = umin(uav[src0], src1.x)`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT.

`src0` provides the address. If raw, `src0.x` (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `umin(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_UMIN
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_umin_id(1) r0.xy, r1.z`

Related None.

Atomic Bitwise XOR to UAV

Instructions **`UAV_XOR`**

Syntax `uav_xor_id(n) src0, src1.x`

Description Atomic single component bitwise XOR to UAV: `uav[src0] = xor(src1.x, uav[src0])`

A UAV with ID n must have been declared as raw, structured, or typed. If typed, it must be declared as 32-bit UINT or SINT.

`src0` provides the address. If raw, src0.x (post-swizzle) provides the address in bytes; if structured, `src0.xy` provides the address of struct index and offset in bytes.

If typed, the number of components used for the address depends on the UAV dimension. For example, for texture1D Arrays, `src0.x` (post-swizzle) provides the buffer address, and `src0.y` (post-swizzle) provides the index/offset of the array.

`src1.x` (post-swizzle) provides the 32-bit Dword to be computed.

The 32-bit UAV memory specified by the address in `src0` is updated atomically by `XOR(uav[src0], src1.x)`. Nothing is returned.

An instruction with an out-of-range address writes nothing to the UAV surface; however, for a structured UAV, if the offset is out-of-bounds, an undefined value is written to the UAV.

If the shader invocation is inactive, nothing is written to the UAV surface.

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, 0 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_UAV_XOR
	`control`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example `uav_xor_id(1) r0.xy, r1.z`

Related None.

7.17 GDS Instructions

Declare a Typeless GDS

Instructions **`DCL_GDS`**

Syntax `dcl_gds_id(id) n`

Description The opcode is followed by a Dword containing the allocation size, n, which is in bytes and must be four-byte aligned. The `id` is used in subsequent GDS instructions. GDS can be used only in a pixel or compute shader. Evergreen GPUs allow multiple id's; however, the sum of all GDS allocations must be <= 32 kB.

Valid for Evergreen GPUs and later.

Format 0-input, 0-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_GDS
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example

```
dcl_gds_id(0) 1024 ; 1024 GDS words total
```

Related None.

Declare a Structured GDS

Instructions **`DCL_STRUCT_GDS`**

Syntax

```
dcl_struct_gds_id(id) n1, n2
```

Description Structured GDS is an array of struct; the size of each struct is n1, the length of the array is n2. The stride-size, n1, is in bytes, but must be four-byte aligned. The resource id is used in subsequent GDS instructions. GDS can be used only in a pixel or compute shader. Evergreen GPUs allow multiple id's; however, the sum of all GDS allocations must be <= 32 kB.

The opcode is followed by two Dwords: the first is a stride size, the second is a struct-count.

Valid for Evergreen GPUs and later.

Format 3-input, 0-output, 1 additional token.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_DCL_STRUCT_GDS
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Additional token: unsigned integer representing the literal value for n.		

Example

```
dcl_struct_gds_id(0) 1024,2
```

Related None.

Atomic Signed Integer Add in GDS Memory

Instructions **GDS_ADD**

Syntax `gds_add_id(id) src0, src1`

Description This is a 32-bit add. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] += src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] += src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_ADD
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic AND in GDS

Instructions	**`GDS_AND`**
Syntax	`gds_and_id(id) src0, src1`

Description

This is a 32-bit bitwise AND operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] &= src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] &= src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_AND
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Conditional Store in GDS Memory

Instructions **GDS_CMP_STORE**

Syntax

```
gds_cmp_store_id(id) src0, src1, src2
```

Description Address is in bytes; the two LSBs must be zero. The data is in Dwords.

If GDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

```
if (gds[src0x] == src1.x) {
gds[src0.x] = src2.x;
}
```

If GDS is declared as a struct, src0.x (after swizzle) specifies the index into the array, `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

```
if (gds[(src0.x * gds_stride + src0.y)/4] == src1.x) {
  gds[(src0.x * gds_stride + src0.y)/4] = src2.x
}
```

Valid for Evergreen GPUs and later.

Format 3-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_CMP_STORE
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic Integer Decrement in GDS Memory

Instructions **GDS_DEC**

Syntax

```
gds_dec_id(id) src0, src1
```

Description

This is a 32-bit integer decrement operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] = ( gds[a] == 0 || gds[a] > src1.x ) ? src1.x : gds[a] - 1
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] = ( gds[a] == 0 || gds[a] > src1.x ) ? src1.x : gds[a] - 1
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_DEC
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic Integer Increment in GDS Memory

Instructions **GDS_INC**

Syntax

```
gds_inc_id(id) src0, src1
```

Description This is a 32-bit integer increment operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] = ( gds[a] >= src1.x ) ? 0 : gds[a] + 1
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] = ( gds[a] >= src1.x ) ? 0 : gds[a] + 1
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_INC
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Read from GDS Memory

Instructions **GDS_LOAD**

Syntax

```
gds_load_id(id) dst, src0
```

Description Address is in bytes, but the two LSBs must be zero. The load result is in Dwords. If GDS is declared typeless, src0.x (after swizzle) specifies the index into the array, src0.y (after swizzle) specifies the offset into the struct. The offset is in bytes.

Valid for Evergreen GPUs and later.

Format 1-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_LOAD
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Example

```
dst.x = gds[(src0.x * gds stride + src0.y)/4]
```

Related None.

Atomic Integer Maximum in GDS Memory

Instructions **GDS_MAX**

Syntax

```
gds_max_id(id) src0, src1
```

Description This is a 32-bit integer operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] = max( gds[a], src1.x )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] = max( gds[a], src1.x )
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_MAX
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic Integer Minimumof in GDS Memory

Instructions **GDS_MIN**

Syntax

```
gds_min_id(id) src0, src1
```

Description This is a 32-bit integer operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] = min( gds[a], src1.x )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] = min( gds[a], src1.x )
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_MIN
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic Mask OR in GDS Memory

Instructions	**GDS_MSKOR**
Syntax	`gds_mskor_id(id) src0, src1, src2`

Description

This is a 32-bit bitwise operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] = ( gds[a] & ~src1.x ) | src2.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] = ( gds[a] & ~src1.x ) | src2.x
```

Valid for Evergreen GPUs and later.

Format 3-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_MSKOR
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Integer OR in GDS Memory

Instructions	**`GDS_OR`**
Syntax	`gds_or_id(id) src0, src1`

Description

This is a 32-bit bitwise OR operation. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
gds[a] |= src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
gds[a] |= src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_OR
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Signed Integer ADD

Instructions	**`GDS_READ_ADD`**
Syntax	`gds_read_add_id(id) dst, src0, src1`

Description

Atomic read global data share (GDS) memory and integer ADD `src1.x`.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] += src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x* gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] += src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_ADD
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Bitwise AND

Instructions **GDS_READ_AND**

Syntax

```
gds_read_and_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and bitwise AND of `src1.x`.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = gds[a] & src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x* gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = gds[a] & src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_AND
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Compare src1 With Lead GDS Memory and, if Equal, Exchange With src2

Instructions **`GDS_READ_CMP_XCHG`**

Syntax `gds_read_cmp_xchg_id(id) dst, src0, src1, src2`

Description Read global data share (GDS) memory, compare with `src1`, and exchange with `src2`.

Compare and exchange is the IL version of a classic lock. Using the same addressing rules as the other GDS atomics (see preceding opcode) it performs the following operation.

```
a = ... as above
dst.x = gds[a]
if (dst.x == src1.x) {
  gds[a] = src2.x;
}
```

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_CMP_XCHG
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Decrement

Instructions **GDS_READ_DEC**

Syntax

```
gds_read_dec_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and integer decrement.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = ( gds[a] > src1.x || gds[a] == 0 ? src1.x : gds[a] - 1 )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x* gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = ( gds[a] > src1.x || gds[a] == 0 ? src1.x : gds[a] - 1 )
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_DEC
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Increment of Src1 in GDS Memory

Instructions **GDS_READ_INC**

Syntax

```
gds_read_inc_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and integer increment.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = ( gds[a] >= src1.x ? 0 : gds[a] + 1 )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = ( gds[a] >= src1.x ? 0 : gds[a] + 1 )
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_INC
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Maximum

Instructions **`GDS_READ_MAX`**

Syntax

```
gds_read_max_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and integer maximum.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = max(gds[a], src1.x)
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = max(gds[a], src1.x)
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_MAX
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Minimum

Instructions **GDS_READ_MIN**

Syntax

```
gds_read_min_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and integer minimum.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = min(gds[a], src1.x)
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = min(gds[a], src1.x)
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_MIN
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Bitwise Mask OR

Instructions **GDS_READ_MSKOR**

Syntax

```
gds_read_mskor_id(id) dst, src0, src1, src2
```

Description Atomic read global data share (GDS) memory and bitwise mask OR.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = ( gds[a] & ~src1.x ) | src2.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = ( gds[a] & ~src1.x ) | src2.x
```

Valid for Evergreen GPUs and later.

Format 3-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_MSKOR
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic GDS Memory Read and Bitwise OR

Instructions	**GDS_READ_OR**
Syntax	`gds_read_or_id(id) dst, src0, src1`

Description

Atomic read global data share (GDS) memory and bitwise OR.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = gds[a] | src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = gds[a] | src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_OR
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Reverse Subtract

Instructions **GDS_READ_RSUB**

Syntax

```
gds_read_rsub_id(id) dst, src0, src1
```

Description Atomic read global data share (GDS) memory and integer reverse subtract.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = src1.x - gds[a]
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = src1.x - gds[a]
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_RSUB
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer Subtract

Instructions **GDS_READ_SUB**

Syntax `gds_read_sub_id(id) dst, src0, src1`

Description Atomic read global data share (GDS) memory and integer subtract.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = gds[a] - src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = gds[a] - src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_READ_SUB
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic GDS Memory Read and Unsigned Integer Maximum

Instructions **GDS_READ_UMAX**

Syntax

```
gds_read_umax_id(id) dst, src0, src1
```

Description Returns the unsigned integer maximum from a global data share (GDS) memory read.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = max (gds[a], src1.x)
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = max (gds[a], src1.x)
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_UMAX
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Unsigned Integer Minimum

Instructions	**`GDS_READ_UMIN`**
Syntax	`gds_read_umin_id(id) dst, src0, src1`

Description

Returns the unsigned integer minimum from a global data share (GDS) memory read.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = min (gds[a], src1.x)
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = min (gds[a], src1.x)
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_UMIN
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic GDS Memory Read and Integer XOR

Instructions **GDS_READ_XOR**

Syntax `gds_read_xor_id(id) dst, src0, src1`

Description Returns the XOR result of an atomic read of `src0` and `src1` in the global data share (GDS) memory.

If GDS is declared typeless, `src0.x` specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
a = src0.x/4
dst.x = gds[a]
gds[a] = gds[a] xor src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array; `src0.y` specifies the offset, in bytes, into the struct.

```
a = (src0.x * gds_stride + src0.y)/4
dst.x = gds[a]
gds[a] = gds[a] xor src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_READ_XOR
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Reverse Subtract Signed Integer from GDS

Instructions **`GDS_RSUB`**

Syntax `gds_rsub_id(id) src0, src1`

Description This is a 32-bit reverse subtraction. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

If the GDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group.

- `gds[src0.x] = src1.x – gds[src0.x]`

If the GDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset into the struct. The offset is in bytes.

- `gds[(src0.x*gds_stride + src0.y)/4] = src1.x - gds[(src0.x*lds_stride + src0.y)/4]`

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_RSUB
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Write to GDS Memory

Instructions	**`GDS_STORE`**
Syntax	`gds_store_id(id) dst, src0, src1`
Description	Address is in bytes, but the two LSBs must be zero. The store data is in Dwords. If GDS is declared typeless, `src0.x` (after swizzle) specifies a byte address relative to the work-group. `gds[src0.x] = src1.x` (using integer add) If GDS is declared as struct, `src0.x` (after swizzle) specifies the index into the array; `src0.y` (after swizzle) specifies the offset (in bytes) into the struct. `gds[(src0.x * gds stride + src0.y)/4] = src1.x` Valid for Evergreen GPUs and later.
Format	2-input, 1-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_STORE
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Example	`dst.x = gds[(src0.x * gds stride + src0.y)/4]`
Related	None.

Atomic Subtract Signed Integer from GDS

Instructions **GDS_SUB**

Syntax `gds_sub_id(id) src0, src1`

Description This is a 32-bit subtraction. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
gds[src0.x/4] = gds[src0.x/4] - src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
gds[(src0.x * gds_stride + src0.y)/4] = gds[(src0.x * gds_stride + src0.y)/4] - src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_SUB
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Unsigned Maximum of Src1 and GDS Memory

Instructions	**GDS_UMAX**
Syntax	`gds_umax_id(id) src0, src1`

Description

This is a 32-bit subtraction. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
gds[src0.x/4] = UMAX( gds[src0.x/4], src1.x )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
gds[(src0.x * gds_stride + src0.y)/4] = UMAX( gds[(src0.x * gds_stride + src0.y)/4],
src1.x )
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_UMAX
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

Atomic Unsigned Minimum if Src1 and GDS Memory

Instructions **GDS_UMIN**

Syntax

```
gds_umin_id(id) src0, src1
```

Description This is a 32-bit integer unsigned minimum. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
gds[src0.x/4] = UMIN( gds[src0.x/4], src1.x )
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
gds[(src0.x * gds_stride + src0.y)/4] = UMIN( gds[(src0.x * gds_stride +
src0.y)/4], src1.x )
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	code	15:0	IL_OP_GDS_UMIN
	ID	29:16	Resource ID.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.

Related None.

Atomic Integer XOR of Src1 and GDS Memory

Instructions **GDS_XOR**

Syntax

```
gds_xor_id(id) src0, src1
```

Description This is a 32-bit exclusive OR. If GDS is declared typeless, src0.x specifies a byte address relative to the work-group. The address must be aligned to a Dword: the lower two bits of the address must be zero.

```
gds[src0.x/4] = gds[src0.x/4] ^ src1.x
```

If GDS is declared as a struct, then `src0.x` specifies the index into the array, `src0.y` specifies the offset, in bytes, into the struct.

```
gds[(src0.x * gds_stride + src0.y)/4] = gds[(src0.x * gds_stride + src0.y)/4]
^ src1.x
```

Valid for Evergreen GPUs and later.

Format 2-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_GDS_XOR
	`ID`	29:16	Resource ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Related None.

7.18 Virtual Function / Interface Support

Declare But Not Define a Virtual Function Body / Interface Routine

Instructions	**`DCL_FUNCTION_BODY`**
Syntax	`dcl_function_body <`*`label`*`>`
Description	Declares a function body. The statement establishes the ID numbers of function bodies. The definition of the function must follow. Use `FUNC` to define the function. The Opcode token is followed by one additional token that contains an unsigned integer representing the label of the function body. Valid for Evergreen GPUs and later.
Format	0-input format.

Opcode

Ordinal	**Token**
1	IL_Opcode token with code set to IL_OP_DCL_FUNCTION_BODY (control field ignored).
2	Unsigned integer representing label of the function body.

Related DCL_INTERFACE_PTR, DCL_FUNCTION_TABLE, FCALL.

Declare But Not Define a Virtual Function Table / Interface Routine

Instructions	**`DCL_FUNCTION_TABLE`**
Syntax	`dcl_function_table_id(id)  = {`*`list of function body labels`*`}`
	Defines the virtual functions/interfaces that can be reached by a set of calls on the same object type. The statement establishes the id for an object type. This is analogous to a C++ virtual function table. The Opcode token is followed by a word containing an unsigned integer specifying the number of table entries (function body IDs), followed by a list of words that contain the function body ID numbers. Valid for Evergreen GPUs and later.
Format	0-input format, followed by n function body labels.

Opcode

Token	**Field Name**	**Bits**	**Description**
1	`code`	15:0	IL_OP_ DCL_FUNCTION_TABLE
	`control`	29:16	Table ID.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	Unsigned integer specifying the number of table entries (function body IDs).		
3	List of Dwords containing the function body id numbers.		

Related DCL_FUNCTION_BODY, DCL_FUNCTION_TABLE, FCALL, DCL_INTERFACE_PTR.

Declare a Set of Function Pointers

Instructions **`DCL_INTERFACE_PTR`**

Syntax `dcl_interface_ptr_dynamic(dynamic)_id(id)[array_size][length] = {list of function table IDs}`

Description The statement defines an array of interface pointers that can be used with an f call. It declares a set of function pointers:

`fp<id>, fp<id+1>, fp<id+2>, …, fp<id + array_size - 1>`

If the instruction is marked dynamic, the instance used in the `f` call can be selected by a register.

The statement defines an array of interface pointers that can be used with an FCALL.

Valid for Evergreen GPUs and later.

Format 0-input format, followed by a number of special words.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_DCL_INTERFACE_PTR
	`control`	28:16	Pointer ID.
		29	Dynamic: 0 Instance id is a literal. 1 Instance ID is NOT a literal (dynamic indexing is used).
	`sec_modifier_present`	30	Must be set zero.
	`pri_modifier_present`	31	Must be set zero.

2 `array_size`: number of interface pointers declared = number of available instances that can be used.

3 `length`: number of times this interface is called = the size of the associated function tables.

4 Unsigned integer specifying the number of table entries (function table IDs).

5 List of Dwords containing the function table ID numbers.

Related DCL_FUNCTION_BODY, DCL_FUNCTION_TABLE, FCALL.

Execute a Call Through an Interface

Instructions **`FCALL`**

Syntax `fcall_id(id) src0, <integer label>`

Description This is a subroutine call with late binding. The `id` identifies the interface pointer used for the call. `src0.x` is either a literal or a gpr; it identifies the instance being used. The second argument is an integer that indexes into the function tables.

The call is set up as follows:

- At compilation, the second argument is used to select code from each function table in the interface.
- At execution, `src0.x` is added to the `id` to produce an instance `id`.
- The instance `id` indexes into a table provided by the driver; this, in turn, selects the function table.
- The code selected from this function table is then executed.

The Opcode is followed by an additional word (label).

Valid for Evergreen GPUs and later.

Format 2-input, 0-output, no additional tokens.

Opcode IL_Opcode

Token	**Field Name**	**Bits**	**Description**
1	`code`	15:0	IL_OP_ IL_OP_FCALL
	`control`	29:16	Pointer ID.
	`sec_modifier_present`	30	Must be set zero.
	`pri_modifier_present`	31	Must be set zero.

Execute a Call Through an Interface (Cont.)

Example The following code snippets are taken from a translation to IL of the complex example in the DX11 speclet on subroutines:

```
dcl_function_body 0
dcl_function_body 1
dcl_function_body 2
dcl_function_body 3
dcl_function_body 4
dcl_function_body 5
dcl_function_body 6
dcl_function_body 7
dcl_function_body 8
dcl_function_body 9
dcl_function_body 10
dcl_function_body 11
dcl_function_table_id(0) = {0, 3}
dcl_function_table_id(1) = {1, 6}
dcl_function_table_id(2) = {2, 9}
dcl_function_table_id(3) = {4, 7, 10}
dcl_function_table_id(4) = {5, 8, 11}
dcl_interface_ptr_id(0)_dynamic(0)[1][2] = {0, 1, 2}
dcl_interface_ptr_id(1)_dynamic(1)[9][3] = {3, 4}
…
dcl_literal 10, 0x00000000, 0x00000000, 0x00000000, 0x00000000
fcall_id(0) 10, 0
...
fcall_id(0) 10, 1
...
fcall_id(1) r0, 0
...
fcall_id(1) r1, 1
...
fcall_id(1) r1, 2
...
func 0
...
func 1
...
func 2
...
func 3
...
func 4
...
func 5
...
func 6
...
func 7
...
func 8
...
func 9
...
func 10
...
func 11
...
```

Related DCL_FUNCTION_BODY, DCL_FUNCTION_TABLE, DCL_INTERFACE_PTR.

7.19 Macro Processor Support

The AMD IL supports a simple macro processor language. This allows libraries and simplifies writing complex IL sequences. IL macros are supported both in token format and in text format.

Macro definitions must be at the top of any program that defines macros. All definitions must precede the IL-language token, so that clients can concatenate a macro file with ps, vs, cs, gs, and other IL files.

Macros introduce two new IL_REG_TYPES: IN and OUT.

Formal input arguments are: `in0`, `in1`, `in2`, etc.

Formal output arguments are: `out0`, `out1`, etc.

Define a Macro

Instructions **MACRODEF**

Syntax `mdef(k)_out(m)_in(n)[_outline]`

Description The macro identifier, *k*, must be a unique numerical ID. Following tokens, up to the EndMacro token, are called macro body. The `mdef` instruction cannot appear within the body of a macro. The special IL register types IN and OUT can be used within the macro body. All temporary registers and all literals are scoped to the macro.

The IN registers are zero-based and map up to n-1, as specified in the macro definition. The OUT registers are zero-based and map up to m-1, as specified in the macro definition.

If the outline option is not specified, all macros are inlined into the calling function.

Valid for all GPUs.

Format 0-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_ MACRODEF
	`control`	29:16	Macro *k*.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.
2	`in`	7:0	Number of inputs.
	`out`	15:8	Number of outputs.
	`outline`	16	Outline option use.
	`reserved`	31:17	Reserved.

Example The following macro computes the overflow of adding `in0` and `in1` as unsigned 32-bit integers. It returns 1 in case of an overflow, 0 if the result fits in 32 bits. The result can be used as a carry flag for a multi-word add.

```
mdef(0)_out(1)_in(2)
  INOT r1, in0
  ULT r2, r1, in1
  Dcl_literal l1, 1, 1, 1, 1
  IAND out0, r2, l1
mend
```

To outline this macro as a function:

```
mdef(0)_out(1)_in(2)_outline
  INOT r1, in0
  ULT r2, r1, in1
  Dcl_literal l1, 1,1,1,1
  IAND out0, r2, l1
```

Related MACROEND, MCALL.

End a Macro

Instructions	**`MACROEND`**
Syntax	`mend`
Description	Closes the lexical scope of a macro started by `MACRODEF`. It can be followed by another macro definition or an IL language token. It can appear only at the end of a macro body. Valid for all GPUs.
Format	0-input, 0-output.

Opcode

Field Name	Bits	Description
`code`	15:0	IL_OP_ MACROEND
`control`	29:16	Must be zero.
`sec_modifier_present`	30	Must be zero.
`pri_modifier_present`	31	Must be zero.

Related MACRODEF, MCALL.

Call a Macro

Instructions	**MCALL**
Syntax	`mcall(`*k*`)  (`*list of outputs*`),  (`*list of inputs*`)`

Description

Calls the macro identified by *k*, which is the unique integer identifier of the macro, and expands the macro following the macro expansion rules noted below. Replaces the call with the text from the expanded macro.

Note that in the syntax the parentheses are required, even if there are no inputs or outputs.

Each actual parameter can be either a TEMP or LITERAL.

Macro calls can be nested.

Each output must be a destination token; each input must be a source token.

Valid for all GPUs.

Macro expansion consists of the following steps.

1. The macro expander inserts a prolog consisting of a set of moves from actual input parameters to formal input arguments.
2. A copy of the macro body is inserted.
3. An epilog consisting of moves from formal output arguments to actual output parameters is added.
4. The macro registers are renamed. Renaming consists of:
   a. All temporary registers in the macro body are replaced by new, unique temporary registers.
   b. All literal registers in the macro body are replaced by new, unique literal registers.
   c. All input and output argument registers are replaced by new temporary registers.

For example: Given `mcall(5), r1, r2, l1`, macro expansion generates the following.

Macro	Expansion	Rename
mdef(5)_out(1)_in(2)	; prolog mov in0, r2 mov in1, l1	mov rm0, r2 mov rm1, l1
mov r1, in1 iadd out0, in0, r1	mov r1, in1 iadd out0, in0, r1	mov rm2, rm1 iadd rm3, rm0, rm2
mend	; epilog mov r1, out0	mov r1, rm3

Conditionals in macros can be expressed with normal `S` statements.

Format

0-input, 0-output.

Opcode

Token	Field Name	Bits	Description
1	`code`	15:0	IL_OP_ MCALL
	`control`	29:16	Macro *k*.
	`sec_modifier_present`	30	Must be zero.
	`pri_modifier_present`	31	Must be zero.

Call a Macro (Cont.)

2	count	15:0	Number of outputs.
		31:16	Number of inputs.

Example The following example shows a macro to generate greater than or equal sequence (for different types).

```
mdef(2)_out(1),in(3)
// 1 for unsigned
if in3.x eq 1
   uge out0, in0, in1
else
// 2 for signed
 if in3.x eq 2
   ge out0, in0, in1
   else
// 3 for double
     if in3.x eq 3
       dge out0, in0, in1
     endif
   endif
endif
mend


def l1, 1,2,3,4
mcall(2), (r1),(r2,r3,l1.x) //  unsigned greater than

mcall(2), (r1),(r2,r3,l1.z) //  double greater than
```

Related MACROEND, MACRODEF.

Appendix A Shadow Texture Loads

The following algorithm is used when a shadow texture fetch is required by the value of the shadowmode field in the `TEXLD`, `TEXLDB`, and `TEXLDD` instructions and by the shadow enable value set through `STATE`. While the `POINT` shadow filter must be the fasted, the `WEIGHTED_QUAD` produce a smoother edge in the shadow. The `BEST` shadow filter is provided for implementations that cannot accelerate the `WEIGHTED_QUAD` filter.

```
VECTOR vsrc = EvalSource(src);
VECTOR vdst;
VECTOR temp;
VECTOR weights;
float failVal = Eval(AS_TEX_SHADOW_COMPARE_FAIL_VALUE_N(stage));
float topTexel;
float bottomTexel;
float texel;

if(AS_TEX_SHADOW_FILTER_N(stage) == WEIGHTED_QUAD)
{
    temp[0]=tfetch(stage, vsrc[0] - xoffset(0.5), vsrc[1] - yoffset(0.5), 0.0, 1.0);
    temp[1]=tfetch(stage, vsrc[0] + xoffset(0.5), vsrc[1] - yoffset(0.5), 0.0, 1.0);
    temp[2]=tfetch(stage, vsrc[0] - xoffset(0.5), vsrc[1] + yoffset(0.5), 0.0, 1.0);
    temp[3]=tfetch(stage, vsrc[0] + xoffset(0.5), vsrc[1] + yoffset(0.5), 0.0, 1.0);

    if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == GEQUAL)
    {
            for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] >= temp[i]) ? 1.0 : 0.0;
}
else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == LEQUAL)
{
    for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] <= temp[i]) ? 1.0 : 0.0;
}
else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == GREATER)
{
    for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] > temp[i]) ? 1.0 : 0.0;
}
else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == LESS)
{
    for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] < temp[i]) ? 1.0 : 0.0;
}
else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == EQUAL)
{
    for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] == temp[i]) ? 1.0 : 0.0;
}
else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == NOTEQUAL)
{
    for (i=0; i < 4; i++)
            temp[i] = (vsrc[2] != temp[i]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == ALWAYS)
    {
            for (i=0; i < 4; i++)
                    temp[i] = 1.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == NEVER)
    {
            for (i=0; i < 4; i++)
                    temp[i] = 0.0;
    }

    # max temp.xyzw temp.xyzw failval
            for (i=0; i < 4; i++)
            temp[i] = (temp[i] > failVal) ? temp[i] : failVal;

    # perform a weighted bilinear filter with lerps
    weights         = GetWeights(vsrc);
    topTexel        = temp[0] * weights[0] + temp[1] * (1.0 - weights[0]);
    bottomTexel     = temp[2] * weights[0] + temp[3] * (1.0 - weights[0]);
    texel           = bottomTexel * weights[1] + topTexel * (1.0 - weights[1]);
}
else if(AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == POINT)
{
    temp[0] = tfetch(stage, vsrc[0], vsrc[1], vsrc[2], vsrc[3]);
    if (AS_TEX_SHADOW_COMPARE_FUNC_N == GEQUAL)
    {
            temp[0] = (vsrc[2] >= temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == LEQUAL)
    {
            temp[0] = (vsrc[2] <= temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == GREATER)
    {
            temp[0] = (vsrc[2] > temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == LESS)
    {
            temp[0] = (vsrc[2] < temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == EQUAL)
    {
            temp[0] = (vsrc[2] == temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == NOTEQUAL)
    {
            temp[0] = (vsrc[2] == temp[0]) ? 1.0 : 0.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == ALWAYS)
    {
            temp[0] = 1.0;
    }
    else if (AS_TEX_SHADOW_COMPARE_FUNC_N(stage) == NEVER)
    {
            temp[0] = 0.0;
    }

    # max texel temp.x failval
    texel = (temp[0] > failVal) ? temp[0] : failVal;

}
```

```
else if(AS_TEX_SHADOW_FILTER_N(stage) == BEST)
{

Largest number of samples that can be supported in implementation;
}


if (AS_TEX_SHADOW_USAGE_N(stage) == ALPHA)
{
    v[0] = 0.0;
    v[1] = 0.0;
    v[2] = 0.0;
    v[3] = texel;
}
else if (AS_TEX_SHADOW_USAGE_N(stage) == LUMINANCE)
{
    v[0] = texel;
    v[1] = texel;
    v[2] = texel;
    v[3] = 1.0;
}
else if (AS_TEX_SHADOW_USAGE_N(stage) == INTENSITY)
{
    v[0] = texel;
    v[1] = texel;
    v[2] = texel;
    v[3] = texel;
}
else if (AS_TEX_SHADOW_USAGE_N(stage) == NONE)
{
    v[0] = 0.0;
    v[1] = 0.0;
    v[2] = 0.0;
    v[3] = 1.0;
}
WriteResult(v, dst);
```

Description This instruction, which declares properties of a texture stage, must be issued on each resource that is used in a ld or sample instruction. The value of `stage` corresponds to the stage/unit defined in state.
There can be at most one `DECLRES` per value.
A shader cannot use this instruction on the same stage more than once.
If type is IL_USAGE_PIXTEX_UNKNOWN, this instruction indicates that the texture type of the texture on the stage/unit indicated by stage is not known at shader create time and is determined at shader runtime based upon a setting in `STATE`.
If `coordmode` is set to IL_TEXCOORDMODE_NORMALIZED, this instruction indicates that the texture coordinate is normalized in any subsequent `TEXLD`, `TEXLDB`, or `LOD` instruction for the texture on stage.
If `coordmode` is set to IL_TEXCOORDMODE_UNNORMALIZED, this instruction indicates that the texture coordinate is not normalized in any subsequent `TEXLD`, `TEXLDD`, `TEXLDB`, `TEXLDMS`, `TEXWEIGHT`, `PROJECT`, or `LOD` instruction for the texture on stage when the wrap mode set in state for the stage/unit is set to clamp-to-border, clamp-half-way-to-border, or clamp-to-edge. In this case, the x texture coordinate ranges from 0.0 to the width of the texture, the y texture coordinate ranges from 0.0 to the height of the texture, and the z texture coordinate ranges from 0.0 to the depth of the texture.
If `coordmode` is set to IL_TEXCOORDMODE_UNKNOWN, external state is used to determine if the texture coordinate used in any subsequent `TEXLD`, `TEXLDD`, `TEXLDB`, `TEXLDMS`, `TEXWEIGHT`, `PROJECT`, or `LOD` instructions are normalized at shader run time.
This instruction must occur before the first executable instruction of a shader. It can only occur after a `COMMENT`, `DCLDEF`, `DCLPP`, `DCLPI`, `DCLPIN`, `DCLVOUT`, `DCLV`, `DCLARRAY`, `DEF`, `DEFB`, `NOP`, or another `DCLPT` instruction.

Opcode

Ordinal	**Token**
1	IL_Opcode token with code set to IL_OP_DCLRES. The five bits of the control field must be set to a valid value.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
IL_OP_DCLPT																								type			coor-mode		0		

2	Integer ID for the resource.

Appendix B
IL Enumerations Sequence

This appendix specifies the order of the IL enumerations as expected by the compiler. Copy the following code, and paste it into a header file that is to be included in your project.

```
#define IL_OP_DMAX IL_OP_D_MAX
#define IL_OP_DMIN IL_OP_D_MIN

#define IL_MINOR_VERSION 0
#define IL_MAJOR_VERSION 2

// comments in this enum will be added to the il spec
enum IL_Shader_Type
{
  IL_SHADER_VERTEX, // This code describes a pixel shader
  IL_SHADER_PIXEL,  // This code describes a vertex shader
  IL_SHADER_GEOMETRY, // This code describes a geometry shader(R6xx and
later)
  IL_SHADER_COMPUTE,  // This code describes a compute shader (r7xx and
later)
  IL_SHADER_HULL, //This code describes a hull shader (Rxx and later)
  IL_SHADER_DOMAIN, //This code describes a domain shader (Rxx and later)
  IL_SHADER_LAST  /* dimension the enumeration */
};


// comments in this enum will be added to the il spec
enum IL_Language_Type
{
  IL_LANG_GENERIC, // allows for language specific overrides
  IL_LANG_OPENGL,  // any flavor of opengl
  IL_LANG_DX8_PS,  // direct x 1.x pixel shader
  IL_LANG_DX8_VS,  // direct x 1.x vertex shader
  IL_LANG_DX9_PS,  // direct x 2.x and 3.x pixel shader
  IL_LANG_DX9_VS,  // direct x 2.x and 3.x vertex shader
  IL_LANG_DX10_PS, // direct x 4.x pixel shader
  IL_LANG_DX10_VS, // direct x 4.x vertex shader
  IL_LANG_DX10_GS, // direct x 4.x geometry shader
  IL_LANG_DX11_PS, // direct x 5.x pixel shader
  IL_LANG_DX11_VS, // direct x 5.x vertex shader
  IL_LANG_DX11_GS, // direct x 5.x Geometry shader
  IL_LANG_DX11_CS, // direct x 5.x Compute shader
  IL_LANG_DX11_HS, // direct x 5.x Hull shader
  IL_LANG_DX11_DS, // direct x 5.x Domain shader
  IL_LANG_LAST    /* dimension the enumeration */
};

enum ILOpCode
{
  IL_OP_UNKNOWN,
  IL_OP_ABS,
  IL_OP_ACOS,
  IL_OP_ADD,
  IL_OP_ASIN,
  IL_OP_ATAN,
  IL_OP_BREAK,
  IL_OP_BREAKC,
IL_OP_CALL,
IL_OP_CALLNZ,
IL_OP_CLAMP,
IL_OP_CLG,
IL_OP_CMOV,
IL_OP_CMP,
IL_OP_COLORCLAMP,
IL_OP_COMMENT,
IL_OP_CONTINUE,
IL_OP_CONTINUEC,
IL_OP_COS,
IL_OP_CRS,
IL_OP_DCLARRAY,
IL_OP_DCLDEF,
IL_OP_DCLPI,
IL_OP_DCLPIN,
IL_OP_DCLPP,
IL_OP_DCLPT,
IL_OP_DCLV,
IL_OP_DCLVOUT,
IL_OP_DEF,
IL_OP_DEFB,
IL_OP_DET,
IL_OP_DIST,
IL_OP_DIV,
IL_OP_DP2ADD,
IL_OP_DP3,
IL_OP_DP4,
IL_OP_DST,
IL_OP_DSX,
IL_OP_DSY,
IL_OP_ELSE,
IL_OP_END,
IL_OP_ENDIF,
IL_OP_ENDLOOP,
IL_OP_ENDMAIN,
IL_OP_EXN,
IL_OP_EXP,
IL_OP_EXPP,
IL_OP_FACEFORWARD,
IL_OP_FLR,
IL_OP_FRC,
IL_OP_FUNC,
IL_OP_FWIDTH,
IL_OP_IFC,
IL_OP_IFNZ,
IL_OP_INITV,
IL_OP_KILL,
IL_OP_LEN,
IL_OP_LIT,
IL_OP_LN,
IL_OP_LOD,
IL_OP_LOG,
IL_OP_LOGP,
IL_OP_LOOP,
IL_OP_LRP,
IL_OP_MAD,
IL_OP_MAX,
IL_OP_MEMEXPORT,
IL_OP_MEMIMPORT,
IL_OP_MIN,
IL_OP_MMUL,
IL_OP_MOD,
IL_OP_MOV,
IL_OP_MUL,
IL_OP_NOISE,
IL_OP_NOP,
IL_OP_NRM,
IL_OP_PIREDUCE,
IL_OP_POW,
IL_OP_PRECOMP,
IL_OP_PROJECT,
IL_OP_RCP,
IL_OP_REFLECT,
IL_OP_RET,
IL_OP_RND,
IL_OP_RSQ,
IL_OP_SET,
IL_OP_SGN,
IL_OP_SIN,
IL_OP_SINCOS,
IL_OP_SQRT,
IL_OP_SUB,
IL_OP_TAN,
IL_OP_TEXLD,
IL_OP_TEXLDB,
IL_OP_TEXLDD,
IL_OP_TEXLDMS,
IL_OP_TEXWEIGHT,
IL_OP_TRANSPOSE,
IL_OP_TRC,
IL_OP_DXSINCOS,
IL_OP_BREAK_LOGICALZ,
IL_OP_BREAK_LOGICALNZ,
IL_OP_CALL_LOGICALZ,
IL_OP_CALL_LOGICALNZ,
IL_OP_CASE,
IL_OP_CONTINUE_LOGICALZ,
IL_OP_CONTINUE_LOGICALNZ,
IL_OP_DEFAULT,
IL_OP_ENDSWITCH,
IL_OP_ENDFUNC,
IL_OP_IF_LOGICALZ,
IL_OP_IF_LOGICALNZ,
IL_OP_WHILE,
IL_OP_SWITCH,
IL_OP_RET_DYN,
IL_OP_RET_LOGICALZ,
IL_OP_RET_LOGICALNZ,
IL_DCL_CONST_BUFFER,
IL_DCL_INDEXED_TEMP_ARRAY,
IL_DCL_INPUT_PRIMITIVE,
IL_DCL_LITERAL,
IL_DCL_MAX_OUTPUT_VERTEX_COUNT,
IL_DCL_ODEPTH,
IL_DCL_OUTPUT_TOPOLOGY,
IL_DCL_OUTPUT,
IL_DCL_INPUT,
IL_DCL_VPRIM,
IL_DCL_RESOURCE,
IL_OP_CUT,
IL_OP_DISCARD_LOGICALZ,
IL_OP_DISCARD_LOGICALNZ,
IL_OP_EMIT,
IL_OP_EMIT_THEN_CUT,
IL_OP_LOAD,
IL_OP_RESINFO,
IL_OP_SAMPLE,
IL_OP_SAMPLE_B,
IL_OP_SAMPLE_G,
IL_OP_SAMPLE_L,
IL_OP_SAMPLE_C,
IL_OP_SAMPLE_C_LZ,
IL_OP_I_NOT,
IL_OP_I_OR,
IL_OP_I_XOR,
IL_OP_I_ADD,
IL_OP_I_MAD,
IL_OP_I_MAX,
IL_OP_I_MIN,
IL_OP_I_MUL,
IL_OP_I_MUL_HIGH,
  IL_OP_I_EQ,
  IL_OP_I_GE,
  IL_OP_I_LT,
  IL_OP_I_NEGATE,
  IL_OP_I_NE,
  IL_OP_I_SHL,
  IL_OP_I_SHR,
  IL_OP_U_SHR,
  IL_OP_U_DIV,
  IL_OP_U_MOD,
  IL_OP_U_MAD,
  IL_OP_U_MAX,
  IL_OP_U_MIN,
  IL_OP_U_LT,
  IL_OP_U_GE,
  IL_OP_U_MUL,
  IL_OP_U_MUL_HIGH,
  IL_OP_FTOI,
  IL_OP_FTOU,
  IL_OP_ITOF,
  IL_OP_UTOF,
  IL_OP_AND,
  IL_OP_CMOV_LOGICAL,
  IL_OP_EQ,
  IL_OP_EXP_VEC,
  IL_OP_GE,
  IL_OP_LOG_VEC,
  IL_OP_LT,
  IL_OP_NE,
  IL_OP_ROUND_NEAR,
  IL_OP_ROUND_NEG_INF,
  IL_OP_ROUND_PLUS_INF,
  IL_OP_ROUND_ZERO,
  IL_OP_RSQ_VEC,
  IL_OP_SIN_VEC,
  IL_OP_COS_VEC,
  IL_OP_SQRT_VEC,
  IL_OP_DP2,
  IL_OP_INV_MOV,
  IL_OP_SCATTER,
  IL_OP_D_FREXP,
  IL_OP_D_ADD,
  IL_OP_D_MUL,
  IL_OP_D_2_F,
  IL_OP_F_2_D,
  IL_OP_D_LDEXP,
  IL_OP_D_FRAC,
  IL_OP_D_MULADD,
  IL_OP_FETCH4,
  IL_OP_SAMPLEINFO,
  IL_OP_GETLOD,  // the dx10.1 version of lod
  IL_DCL_PERSIST,
  IL_OP_DNE,
  IL_OP_DEQ,
  IL_OP_DGE,
  IL_OP_DLT,
  IL_OP_SAMPLEPOS,
  IL_OP_D_DIV,
  IL_OP_DCL_SHARED_TEMP,
  IL_OP_INIT_SR, // a special init inst for 7xx CS
  IL_OP_INIT_SR_HELPER, // an internal IL OP, only used by SC
  IL_OP_DCL_NUM_THREAD_PER_GROUP, // thread group = thread block
  IL_OP_DCL_TOTAL_NUM_THREAD_GROUP,
  IL_OP_DCL_LDS_SIZE_PER_THREAD,
  IL_OP_DCL_LDS_SHARING_MODE,
  IL_OP_LDS_READ_VEC,  // R7xx LDS
  IL_OP_LDS_WRITE_VEC,
  IL_OP_FENCE,  // R7xx/Evergreen fence
  IL_OP_LDS_LOAD_VEC, // DX10_CS: DX11 style LDS instruction in vector (4
dwords)
  IL_OP_LDS_STORE_VEC, // DX10_CS: DX11 style LDS instruction in vector (4
dwords)
  IL_OP_DCL_UAV, // Evergreen UAV
  IL_OP_DCL_RAW_UAV,
  IL_OP_DCL_STRUCT_UAV,
  IL_OP_UAV_LOAD,
  IL_OP_UAV_RAW_LOAD,
  IL_OP_UAV_STRUCT_LOAD,
  IL_OP_UAV_STORE,
  IL_OP_UAV_RAW_STORE,
  IL_OP_UAV_STRUCT_STORE,
  IL_OP_DCL_ARENA_UAV, // 8xx/OpenCL Arena UAV
  IL_OP_UAV_ARENA_LOAD,
  IL_OP_UAV_ARENA_STORE,
  IL_OP_UAV_ADD,
  IL_OP_UAV_SUB,
  IL_OP_UAV_RSUB,
  IL_OP_UAV_MIN,
  IL_OP_UAV_MAX,
  IL_OP_UAV_UMIN,
  IL_OP_UAV_UMAX,
  IL_OP_UAV_AND,
  IL_OP_UAV_OR,
  IL_OP_UAV_XOR,
  IL_OP_UAV_CMP,
  IL_OP_UAV_READ_ADD,
  IL_OP_UAV_READ_SUB,
  IL_OP_UAV_READ_RSUB,
  IL_OP_UAV_READ_MIN,
  IL_OP_UAV_READ_MAX,
  IL_OP_UAV_READ_UMIN,
  IL_OP_UAV_READ_UMAX,
  IL_OP_UAV_READ_AND,
  IL_OP_UAV_READ_OR,
  IL_OP_UAV_READ_XOR,
  IL_OP_UAV_READ_XCHG,
  IL_OP_UAV_READ_CMP_XCHG,
  IL_OP_APPEND_BUF_ALLOC, // Evergreen Append buf aloc/consum
  IL_OP_APPEND_BUF_CONSUME,
  IL_OP_DCL_RAW_SRV, // Evergreen SRV
  IL_OP_DCL_STRUCT_SRV,
  IL_OP_SRV_RAW_LOAD,
  IL_OP_SRV_STRUCT_LOAD,
  IL_DCL_LDS,         // Evergreen LDS
  IL_DCL_STRUCT_LDS,
  IL_OP_LDS_LOAD,
  IL_OP_LDS_STORE,
  IL_OP_LDS_ADD,
  IL_OP_LDS_SUB,
  IL_OP_LDS_RSUB,
  IL_OP_LDS_MIN,
  IL_OP_LDS_MAX,
  IL_OP_LDS_UMIN,
  IL_OP_LDS_UMAX,
  IL_OP_LDS_AND,
  IL_OP_LDS_OR,
  IL_OP_LDS_XOR,
  IL_OP_LDS_CMP,
  IL_OP_LDS_READ_ADD,
  IL_OP_LDS_READ_SUB,
  IL_OP_LDS_READ_RSUB,
  IL_OP_LDS_READ_MIN,
  IL_OP_LDS_READ_MAX,
  IL_OP_LDS_READ_UMIN,
  IL_OP_LDS_READ_UMAX,
  IL_OP_LDS_READ_AND,
  IL_OP_LDS_READ_OR,
  IL_OP_LDS_READ_XOR,
  IL_OP_LDS_READ_XCHG,
  IL_OP_LDS_READ_CMP_XCHG,
  IL_OP_CUT_STREAM,
  IL_OP_EMIT_STREAM,
IL_OP_EMIT_THEN_CUT_STREAM,
IL_OP_SAMPLE_C_L,
IL_OP_SAMPLE_C_G,
IL_OP_SAMPLE_C_B,
IL_OP_I_COUNTBITS,
IL_OP_I_FIRSTBIT,
IL_OP_I_CARRY,
IL_OP_I_BORROW,
IL_OP_I_BIT_EXTRACT,
IL_OP_U_BIT_EXTRACT,
IL_OP_U_BIT_REVERSE,
IL_DCL_NUM_ICP,
IL_DCL_NUM_OCP,
IL_DCL_NUM_INSTANCES,
IL_OP_HS_CP_PHASE,
IL_OP_HS_FORK_PHASE,
IL_OP_HS_JOIN_PHASE,
IL_OP_ENDPHASE,
IL_DCL_TS_DOMAIN,
IL_DCL_TS_PARTITION,
IL_DCL_TS_OUTPUT_PRIMITIVE,
IL_DCL_MAX_TESSFACTOR,
IL_OP_DCL_FUNCTION_BODY,
IL_OP_DCL_FUNCTION_TABLE,
IL_OP_DCL_INTERFACE_PTR,
IL_OP_FCALL,
IL_OP_U_BIT_INSERT,
IL_OP_BUFINFO,
IL_OP_FETCH4_C,
IL_OP_FETCH4_PO,
IL_OP_FETCH4_PO_C,
IL_OP_D_MAX,
IL_OP_D_MIN,
IL_OP_F_2_F16,
IL_OP_F16_2_F,
IL_OP_UNPACK0,
IL_OP_UNPACK1,
IL_OP_UNPACK2,
IL_OP_UNPACK3,
IL_OP_BIT_ALIGN,
IL_OP_BYTE_ALIGN,
IL_OP_U4LERP,
IL_OP_SAD,
IL_OP_SAD_HI,
IL_OP_SAD_4,
IL_OP_F_2_U4,
IL_OP_EVAL_SNAPPED,
IL_OP_EVAL_SAMPLE_INDEX,
IL_OP_EVAL_CENTROID,
IL_OP_D_MOV,
IL_OP_D_MOVC,
IL_OP_D_SQRT,
IL_OP_D_RCP,
IL_OP_D_RSQ,
IL_OP_MACRODEF,
IL_OP_MACROEND,
IL_OP_MACROCALL,
IL_DCL_STREAM,
IL_DCL_GLOBAL_FLAGS,
IL_OP_RCP_VEC,
IL_OP_LOAD_FPTR,
IL_DCL_MAX_THREAD_PER_GROUP,
IL_OP_PREFIX,
// GDS
IL_DCL_GDS,
IL_DCL_STRUCT_GDS,
IL_OP_GDS_LOAD,
IL_OP_GDS_STORE,
IL_OP_GDS_ADD,
IL_OP_GDS_SUB,
IL_OP_GDS_RSUB,
    IL_OP_GDS_INC,
    IL_OP_GDS_DEC,
    IL_OP_GDS_MIN,
    IL_OP_GDS_MAX,
    IL_OP_GDS_UMIN,
    IL_OP_GDS_UMAX,
    IL_OP_GDS_AND,
    IL_OP_GDS_OR,
    IL_OP_GDS_XOR,
    IL_OP_GDS_MSKOR,
    IL_OP_GDS_CMP_STORE,
    IL_OP_GDS_READ_ADD,
    IL_OP_GDS_READ_SUB,
    IL_OP_GDS_READ_RSUB,
    IL_OP_GDS_READ_INC,
    IL_OP_GDS_READ_DEC,
    IL_OP_GDS_READ_MIN,
    IL_OP_GDS_READ_MAX,
    IL_OP_GDS_READ_UMIN,
    IL_OP_GDS_READ_UMAX,
    IL_OP_GDS_READ_AND,
    IL_OP_GDS_READ_OR,
    IL_OP_GDS_READ_XOR,
    IL_OP_GDS_READ_MSKOR,
    IL_OP_GDS_READ_XCHG,
    IL_OP_GDS_READ_CMP_XCHG,
    IL_OP_U_MAD24,
    IL_OP_U_MUL24,
    IL_OP_FMA,
    IL_OP_UAV_UINC,
    IL_OP_UAV_UDEC,
    IL_OP_I_MAD24,
    IL_OP_I_MUL24,
    IL_OP_UAV_READ_UINC,
    IL_OP_UAV_READ_UDEC,
    IL_OP_LDS_LOAD_BYTE,
    IL_OP_LDS_LOAD_SHORT,
    IL_OP_LDS_LOAD_UBYTE,
    IL_OP_LDS_LOAD_USHORT,
    IL_OP_LDS_STORE_BYTE,
    IL_OP_LDS_STORE_SHORT,
    IL_OP_UAV_BYTE_LOAD,
    IL_OP_UAV_SHORT_LOAD,
    IL_OP_UAV_UBYTE_LOAD,
    IL_OP_UAV_USHORT_LOAD,
    IL_OP_UAV_BYTE_STORE,
    IL_OP_UAV_SHORT_STORE,
    IL_OP_I64_ADD,
    IL_OP_I64_EQ,
    IL_OP_I64_GE,
    IL_OP_I64_LT,
    IL_OP_I64_MAX,
    IL_OP_I64_MIN,
    IL_OP_I64_NE,
    IL_OP_I64_NEGATE,
    IL_OP_I64_SHL,
    IL_OP_I64_SHR,
    IL_OP_U64_GE,
    IL_OP_U64_LT,
    IL_OP_U64_MAX,
    IL_OP_U64_MIN,
    IL_OP_U64_SHR,
    IL_OP_DCL_TYPED_UAV,
    IL_OP_DCL_TYPELESS_UAV,
    IL_OP_I_MUL24_HIGH,
    IL_OP_U_MUL24_HIGH,
    IL_OP_LDS_INC,
    IL_OP_LDS_DEC,
    IL_OP_LDS_READ_INC,
    IL_OP_LDS_READ_DEC,
    IL_OP_LDS_MSKOR,
  IL_OP_LDS_READ_MSKOR,
  IL_OP_F_2_F16_NEAR,
  IL_OP_F_2_F16_NEG_INF,
  IL_OP_F_2_F16_PLUS_INF,
  IL_OP_I64_MUL,
  IL_OP_U64_MUL,
  IL_OP_LDEXP,
  IL_OP_FREXP_EXP,
  IL_OP_FREXP_MANT,
  IL_OP_D_FREXP_EXP,
  IL_OP_D_FREXP_MANT,
  IL_OP_DTOI,
  IL_OP_DTOU,
  IL_OP_ITOD,
  IL_OP_UTOD,
  IL_OP_FTOI_RPI,
  IL_OP_FTOI_FLR,
  IL_OP_MIN3,
  IL_OP_MED3,
  IL_OP_MAX3,
  IL_OP_I_MIN3,
  IL_OP_I_MED3,
  IL_OP_I_MAX3,
  IL_OP_U_MIN3,
  IL_OP_U_MED3,
  IL_OP_U_MAX3,
  IL_OP_CLASS,
  IL_OP_D_CLASS,
  IL_OP_SAMPLE_RETURN_CODE,
  IL_OP_CU_ID,
  IL_OP_WAVE_ID,
  IL_OP_I64_SUB,
  IL_OP_I64_DIV,
  IL_OP_U64_DIV,
  IL_OP_I64_MOD,
  IL_OP_U64_MOD,
  IL_DCL_GWS_THREAD_COUNT,
  IL_DCL_SEMAPHORE,
  IL_OP_SEMAPHORE_INIT,
  IL_OP_SEMAPHORE_SIGNAL,
  IL_OP_SEMAPHORE_WAIT,
  IL_OP_DIV_SCALE,
  IL_OP_DIV_FMAS,
  IL_OP_DIV_FIXUP,
  IL_OP_D_DIV_SCALE,
  IL_OP_D_DIV_FMAS,
  IL_OP_D_DIV_FIXUP,
  IL_OP_D_TRIG_PREOP,
  IL_OP_MSAD_U8,
  IL_OP_QSAD_U8,
  IL_OP_MQSAD_U8,
  IL_OP_LAST  /* dimension the enumeration */
};

// \todo remove this define once sc is promoted to dxx and dx=>il has been
updated.
#define IL_OP_I_BIT_INSERT IL_OP_U_BIT_INSERT

// comments in this enum will be added to the il spec
enum ILRegType
{
  IL_REGTYPE_CONST_BOOL,  // single bit boolean constant
  IL_REGTYPE_CONST_FLOAT,
  IL_REGTYPE_CONST_INT,
  IL_REGTYPE_ADDR,
  IL_REGTYPE_TEMP,
  IL_REGTYPE_VERTEX,
  IL_REGTYPE_INDEX,
  IL_REGTYPE_OBJECT_INDEX,
  IL_REGTYPE_BARYCENTRIC_COORD,
  IL_REGTYPE_PRIMITIVE_INDEX,
  IL_REGTYPE_QUAD_INDEX,
  IL_REGTYPE_VOUTPUT,
  IL_REGTYPE_PINPUT,
  IL_REGTYPE_SPRITE,
  IL_REGTYPE_POS,
  IL_REGTYPE_INTERP,
  IL_REGTYPE_FOG,
  IL_REGTYPE_TEXCOORD,
  IL_REGTYPE_PRICOLOR,
  IL_REGTYPE_SECCOLOR,
  IL_REGTYPE_SPRITECOORD,
  IL_REGTYPE_FACE,
  IL_REGTYPE_WINCOORD,
  IL_REGTYPE_PRIMCOORD,
  IL_REGTYPE_PRIMTYPE,
  IL_REGTYPE_PCOLOR,
  IL_REGTYPE_DEPTH,
  IL_REGTYPE_STENCIL,
  IL_REGTYPE_CLIP,
  IL_REGTYPE_VPRIM,
  IL_REGTYPE_ITEMP,
  IL_REGTYPE_CONST_BUFF,
  IL_REGTYPE_LITERAL,
  IL_REGTYPE_INPUT,
  IL_REGTYPE_OUTPUT,
  IL_REGTYPE_IMMED_CONST_BUFF,
  IL_REGTYPE_OMASK,
  IL_REGTYPE_PERSIST,
  IL_REGTYPE_GLOBAL,
  IL_REGTYPE_PS_OUT_FOG,
  IL_REGTYPE_SHARED_TEMP,
  IL_REGTYPE_THREAD_ID_IN_GROUP, // 3-dim
  IL_REGTYPE_THREAD_ID_IN_GROUP_FLAT,  // 1-dim
  IL_REGTYPE_ABSOLUTE_THREAD_ID, // 3-dim
  IL_REGTYPE_ABSOLUTE_THREAD_ID_FLAT, // 1-dim
  IL_REGTYPE_THREAD_GROUP_ID,  // 3-dim
  IL_REGTYPE_THREAD_GROUP_ID_FLAT, // 1-dim
  IL_REGTYPE_GENERIC_MEM, // generic memory type, used w/mask of lds_write
  IL_REGTYPE_INPUTCP,     // tessellation, input control-point register
  IL_REGTYPE_PATCHCONST,  // tessellation, patch constants
  IL_REGTYPE_DOMAINLOCATION, // domain shader, domain location
  IL_REGTYPE_OUTPUTCP,    // tessellation, output control-point register
  IL_REGTYPE_OCP_ID,      // tessellation, output control-point id
  IL_REGTYPE_SHADER_INSTANCE_ID, // tessellation hs fork/join instance id
or gs instance id
  IL_REGTYPE_THIS,
  IL_REGTYPE_EDGEFLAG, //edge flag
  IL_REGTYPE_DEPTH_LE, //dx11 conservative depth guaranteed to be <=
raster depth
  IL_REGTYPE_DEPTH_GE, //dx11 conservative depth guaranteed to be >=
raster depth
  IL_REGTYPE_INPUT_COVERAGE_MASK, //dx11 ps input coverage mask
  IL_REGTYPE_TIMER,
  IL_REGTYPE_LINE_STIPPLE, // Evergreen+, anti-aliased line stipple
  IL_REGTYPE_INPUT_ARG,  // macro processor input
  IL_REGTYPE_OUTPUT_ARG, // macro processor output
  IL_REGTYPE_LAST,       // Must be < 64, we only have 6 bits for encoding
the reg type
                         // SoftIL requires IL_REGTYPE_LAST <= 63
                         // We cannot add any more IL_REGTYPE without
using extended field
};

enum ILMatrix
{
  IL_MATRIX_4X4,
  IL_MATRIX_4X3,
  IL_MATRIX_3X4,
  IL_MATRIX_3X3,
  IL_MATRIX_3X2,
  IL_MATRIX_LAST      /* dimension the enumeration */
```

```
};

enum ILDivComp
{
  IL_DIVCOMP_NONE,    /* None */
  IL_DIVCOMP_Y,       /* Divide the x component by y */
  IL_DIVCOMP_Z,       /* Divide the x and y components by z */
  IL_DIVCOMP_W,       /* Divide the x, y, and z components by w */
  IL_DIVCOMP_UNKNOWN, /* Divide each component by the value of AS_**** */
  IL_DIVCOMP_LAST
};

enum ILZeroOp
{
  IL_ZEROOP_FLTMAX,
  IL_ZEROOP_0,
  IL_ZEROOP_INFINITY,
  IL_ZEROOP_INF_ELSE_MAX,
  IL_ZEROOP_LAST    /* dimension the enumeration */
};

enum ILModDstComponent
{
  IL_MODCOMP_NOWRITE, // do not write this component
  IL_MODCOMP_WRITE,   // write the result to this component
  IL_MODCOMP_0,      // force the component to float 0.0
  IL_MODCOMP_1,      // force the component to float 1.0
  IL_MODCOMP_LAST     /* dimension the enumeration */
};

enum ILComponentSelect
{
  IL_COMPSEL_X_R, //select the 1st component (x/red) for the channel.
  IL_COMPSEL_Y_G, //select the 2nd component (y/green) for the channel.
  IL_COMPSEL_Z_B, // select the 3rd component (z/blue) for the channel.
  IL_COMPSEL_W_A, //select the 4th component (w/alpha) for the channel.
  IL_COMPSEL_0,  //Force this channel to 0.0
  IL_COMPSEL_1, //Force this channel to 1.0
  IL_COMPSEL_LAST     /* dimension the enumeration */
};

enum ILShiftScale
{
  IL_SHIFT_NONE = 0,
  IL_SHIFT_X2,
  IL_SHIFT_X4,
  IL_SHIFT_X8,
  IL_SHIFT_D2,
  IL_SHIFT_D4,
  IL_SHIFT_D8,
  IL_SHIFT_LAST       /* dimension the enumeration */
};

enum ILRelOp
{
  IL_RELOP_NE,  /* != */
  IL_RELOP_EQ,  /* == */
  IL_RELOP_GE,  /* >= */
  IL_RELOP_GT,  /* > */
  IL_RELOP_LE,  /* <= */
  IL_RELOP_LT,  /* < */
  IL_RELOP_LAST /* dimension the enumeration */
};

enum ILDefaultVal
{
  IL_DEFVAL_NONE = 0,
  IL_DEFVAL_0,
  IL_DEFVAL_1,
  IL_DEFVAL_LAST /* dimension the enumeration */
};
```

```
enum ILImportComponent
{
  IL_IMPORTSEL_UNUSED = 0,
  IL_IMPORTSEL_DEFAULT0,
  IL_IMPORTSEL_DEFAULT1,
  IL_IMPORTSEL_UNDEFINED,
  IL_IMPORTSEL_LAST  /*  dimension the enum */
};

enum ILImportUsage
{
  IL_IMPORTUSAGE_POS = 0,
  IL_IMPORTUSAGE_POINTSIZE,
  IL_IMPORTUSAGE_COLOR,
  IL_IMPORTUSAGE_BACKCOLOR,
  IL_IMPORTUSAGE_FOG,
  IL_IMPORTUSAGE_PIXEL_SAMPLE_COVERAGE,
  IL_IMPORTUSAGE_GENERIC,
  IL_IMPORTUSAGE_CLIPDISTANCE,
  IL_IMPORTUSAGE_CULLDISTANCE,
  IL_IMPORTUSAGE_PRIMITIVEID,
  IL_IMPORTUSAGE_VERTEXID,
  IL_IMPORTUSAGE_INSTANCEID,
  IL_IMPORTUSAGE_ISFRONTFACE,
  IL_IMPORTUSAGE_LOD,
  IL_IMPORTUSAGE_COLORING,
  IL_IMPORTUSAGE_NODE_COLORING,
  IL_IMPORTUSAGE_NORMAL,
  IL_IMPORTUSAGE_RENDERTARGET_ARRAY_INDEX,
  IL_IMPORTUSAGE_VIEWPORT_ARRAY_INDEX,
  IL_IMPORTUSAGE_UNDEFINED,
  IL_IMPORTUSAGE_SAMPLE_INDEX,
  IL_IMPORTUSAGE_EDGE_TESSFACTOR,
  IL_IMPORTUSAGE_INSIDE_TESSFACTOR,
  IL_IMPORTUSAGE_DETAIL_TESSFACTOR,
  IL_IMPORTUSAGE_DENSITY_TESSFACTOR,
  IL_IMPORTUSAGE_LAST  /*  dimension the enum */
};

enum ILCmpVal
{
  IL_CMPVAL_0_0 = 0, /* compare vs. 0.0  */
  IL_CMPVAL_0_5,     /* compare vs. 0.5  */
  IL_CMPVAL_1_0,     /* compare vs. 1.0  */
  IL_CMPVAL_NEG_0_5, /* compare vs. -0.5 */
  IL_CMPVAL_NEG_1_0, /* compare vs. -1.0 */
  IL_CMPVAL_LAST     /* dimension the enumeration */
};

// dependent upon this table:
// - dx10interpreter.cpp - table that maps dx names to il
// - iltables.cpp - il_pixtex_props_table

enum ILPixTexUsage
{
  // dx9 only il allows 0-7 values for dclpt
  IL_USAGE_PIXTEX_UNKNOWN = 0,
  IL_USAGE_PIXTEX_1D,
  IL_USAGE_PIXTEX_2D,
  IL_USAGE_PIXTEX_3D,
  IL_USAGE_PIXTEX_CUBEMAP,
  IL_USAGE_PIXTEX_2DMSAA,
  // dx10 formats after this point
  IL_USAGE_PIXTEX_4COMP,
  IL_USAGE_PIXTEX_BUFFER,
  IL_USAGE_PIXTEX_1DARRAY,
  IL_USAGE_PIXTEX_2DARRAY,
  IL_USAGE_PIXTEX_2DARRAYMSAA,
  IL_USAGE_PIXTEX_2D_PLUS_W, // Pele hardware feature
  IL_USAGE_PIXTEX_CUBEMAP_PLUS_W, // Pele hardware feature
  // dx 10.1
  IL_USAGE_PIXTEX_CUBEMAP_ARRAY,
  IL_USAGE_PIXTEX_LAST    /* dimension the enumeration */
};

enum ILTexCoordMode
{
  IL_TEXCOORDMODE_UNKNOWN = 0,
  IL_TEXCOORDMODE_NORMALIZED,
  IL_TEXCOORDMODE_UNNORMALIZED,
  IL_TEXCOORDMODE_LAST   /* dimension the enumeration */
};

enum ILElementFormat
{
  IL_ELEMENTFORMAT_UNKNOWN = 0,
  IL_ELEMENTFORMAT_SNORM,
  IL_ELEMENTFORMAT_UNORM,
  IL_ELEMENTFORMAT_SINT,
  IL_ELEMENTFORMAT_UINT,
  IL_ELEMENTFORMAT_FLOAT,
  IL_ELEMENTFORMAT_SRGB,
  IL_ELEMENTFORMAT_MIXED,
  IL_ELEMENTFORMAT_LAST
};

enum ILTexShadowMode
{
  IL_TEXSHADOWMODE_NEVER = 0,
  IL_TEXSHADOWMODE_Z,
  IL_TEXSHADOWMODE_UNKNOWN,
  IL_TEXSHADOWMODE_LAST /* dimension the enumeration */
};

enum ILTexFilterMode
{
  IL_TEXFILTER_UNKNOWN = 0,
  IL_TEXFILTER_POINT,
  IL_TEXFILTER_LINEAR,
  IL_TEXFILTER_ANISO,
  IL_TEXFILTER_LAST    /* dimension the enumeration */
};

enum ILMipFilterMode
{
  IL_MIPFILTER_UNKNOWN = 0,
  IL_MIPFILTER_POINT,
  IL_MIPFILTER_LINEAR,
  IL_MIPFILTER_BASE,
  IL_MIPFILTER_LAST    /* dimension the enumeration */
};

enum ILAnisoFilterMode
{
  IL_ANISOFILTER_UNKNOWN = 0,
  IL_ANISOFILTER_DISABLED,
  IL_ANISOFILTER_MAX_1_TO_1,
  IL_ANISOFILTER_MAX_2_TO_1,
  IL_ANISOFILTER_MAX_4_TO_1,
  IL_ANISOFILTER_MAX_8_TO_1,
  IL_ANISOFILTER_MAX_16_TO_1,
  IL_ANISOFILTER_LAST   /* dimension the enumeration */
};

enum ILNoiseType
{
  IL_NOISETYPE_PERLIN1D = 0,
  IL_NOISETYPE_PERLIN2D,
  IL_NOISETYPE_PERLIN3D,
  IL_NOISETYPE_PERLIN4D,
  IL_NOISETYPE_LAST    /* dimension the enumeration */
```

```
};

enum ILAddressing
{
  IL_ADDR_ABSOLUTE = 0,
  IL_ADDR_RELATIVE,
  IL_ADDR_REG_RELATIVE,
  IL_ADDR_LAST    /* dimension the enumeration */
};

enum ILInterpMode
{
  IL_INTERPMODE_NOTUSED = 0,
  IL_INTERPMODE_CONSTANT,
  IL_INTERPMODE_LINEAR,
  IL_INTERPMODE_LINEAR_CENTROID,
  IL_INTERPMODE_LINEAR_NOPERSPECTIVE,
  IL_INTERPMODE_LINEAR_NOPERSPECTIVE_CENTROID,
  IL_INTERPMODE_LINEAR_SAMPLE,
  IL_INTERPMODE_LINEAR_NOPERSPECTIVE_SAMPLE,
  IL_INTERPMODE_LAST    /* dimension the enumeration */
};

// types for scatter
enum IL_SCATTER
{
  IL_SCATTER_BY_PIXEL,
  IL_SCATTER_BY_QUAD
};

// types that can be input to a gemetry shader

enum IL_TOPOLOGY
{
  IL_TOPOLOGY_POINT,
  IL_TOPOLOGY_LINE,
  IL_TOPOLOGY_TRIANGLE,
  IL_TOPOLOGY_LINE_ADJ,
  IL_TOPOLOGY_TRIANGLE_ADJ,
  IL_TOPOLOGY_PATCH1,
  IL_TOPOLOGY_PATCH2,
  IL_TOPOLOGY_PATCH3,
  IL_TOPOLOGY_PATCH4,
  IL_TOPOLOGY_PATCH5,
  IL_TOPOLOGY_PATCH6,
  IL_TOPOLOGY_PATCH7,
  IL_TOPOLOGY_PATCH8,
  IL_TOPOLOGY_PATCH9,
  IL_TOPOLOGY_PATCH10,
  IL_TOPOLOGY_PATCH11,
  IL_TOPOLOGY_PATCH12,
  IL_TOPOLOGY_PATCH13,
  IL_TOPOLOGY_PATCH14,
  IL_TOPOLOGY_PATCH15,
  IL_TOPOLOGY_PATCH16,
  IL_TOPOLOGY_PATCH17,
  IL_TOPOLOGY_PATCH18,
  IL_TOPOLOGY_PATCH19,
  IL_TOPOLOGY_PATCH20,
  IL_TOPOLOGY_PATCH21,
  IL_TOPOLOGY_PATCH22,
  IL_TOPOLOGY_PATCH23,
  IL_TOPOLOGY_PATCH24,
  IL_TOPOLOGY_PATCH25,
  IL_TOPOLOGY_PATCH26,
  IL_TOPOLOGY_PATCH27,
  IL_TOPOLOGY_PATCH28,
  IL_TOPOLOGY_PATCH29,
  IL_TOPOLOGY_PATCH30,
  IL_TOPOLOGY_PATCH31,
  IL_TOPOLOGY_PATCH32,
  IL_TOPOLOGY_LAST  /* dimension the enumeration */
};

enum IL_OUTPUT_TOPOLOGY
{
  IL_OUTPUT_TOPOLOGY_POINT_LIST,
  IL_OUTPUT_TOPOLOGY_LINE_STRIP,
  IL_OUTPUT_TOPOLOGY_TRIANGLE_STRIP,
  IL_OUTPUT_TOPOLOGY_LAST  /* dimension the enumeration */
};

// for R7xx Compute shader
enum IL_LDS_SHARING_MODE
{
    IL_LDS_SHARING_MODE_RELATIVE = 0, // for wavefront_rel
    IL_LDS_SHARING_MODE_ABSOLUTE,  // for wavefront_abs
    IL_LDS_SHARING_MODE_LAST
};

// for OpenCL Arena UAV load/store and others
enum IL_LOAD_STORE_DATA_SIZE
{
    IL_LOAD_STORE_DATA_SIZE_DWORD = 0, // dword, 32 bits
    IL_LOAD_STORE_DATA_SIZE_SHORT,   // short, 16 bits
    IL_LOAD_STORE_DATA_SIZE_BYTE,   // byte, 8 bits
    IL_LOAD_STORE_DATA_SIZE_LAST
};

enum IL_UAV_ACCESS_TYPE
{
    IL_UAV_ACCESS_TYPE_RW = 0,    // read_write
    IL_UAV_ACCESS_TYPE_RO,        // read_only
    IL_UAV_ACCESS_TYPE_WO,        // write_only
    IL_UAV_ACCESS_TYPE_PRIVATE,   // private
    IL_UAV_ACCESS_TYPE_LAST
};

// type of firstbit
enum IL_FIRSTBIT_TYPE
{
  IL_FIRSTBIT_TYPE_LOW_UINT,
  IL_FIRSTBIT_TYPE_HIGH_UINT,
  IL_FIRSTBIT_TYPE_HIGH_INT
};

enum ILTsDomain
{
  IL_TS_DOMAIN_ISOLINE = 0,
  IL_TS_DOMAIN_TRI = 1,
  IL_TS_DOMAIN_QUAD = 2,
  IL_TS_DOMAIN_LAST,
};

enum ILTsPartition
{
  IL_TS_PARTITION_INTEGER,
  IL_TS_PARTITION_POW2,
  IL_TS_PARTITION_FRACTIONAL_ODD,
  IL_TS_PARTITION_FRACTIONAL_EVEN,
  IL_TS_PARTITION_LAST,
};

enum ILTsOutputPrimitive
{
  IL_TS_OUTPUT_POINT,
  IL_TS_OUTPUT_LINE,
  IL_TS_OUTPUT_TRIANGLE_CW,     // clockwise
  IL_TS_OUTPUT_TRIANGLE_CCW,    // counter clockwise
  IL_TS_OUTPUT_LAST,
};
```

```
//Granularity of gradient
enum IL_DERIV_GRANULARITY
{
  IL_DERIVE_COARSE = 0,
  IL_DERIVE_FINE = 0x80
};

enum IL_IEEE_CONTROL
{
  IL_IEEE_IGNORE = 0,
  IL_IEEE_PRECISE = 0x1
};

enum IL_UAV_READ_TYPE
{
  IL_UAV_SC_DECIDE = 0,
  IL_UAV_FORCE_CACHED = 1,
  IL_UAV_FORCE_UNCACHED = 2
};

#ifdef __cplusplus
}
```

Appendix C ASIC- and Model-Specific Restrictions

This appendix describes the restrictions for specific IL instructions and registers.

C.1 IL_Opcode Restrictions

Certain il_opcodes are not supported on all chips.

Opcodes not supported on R3/400 series chips:

- `IL_OP_CONTINUE`
- `IL_OP_LOD`
- All integer opcodes.
- Fetch opcodes are not supported in vertex shaders.

Opcodes not supported on R500 series chips:

- All integer opcodes

C.2 ShaderModel Restrictions

This document covers two shader models that are not compatible with each other. These first, SM40, corresponds to DirectX shader model 4.0 or later. The second, SM30, corresponds to DirectX shader model 3.0 and earlier. Most of the IL instructions and registers can be used in both shader models; however, there are some that can be used in one, but not the other. Table C-1 lists the instructions valid only for SM40. Table C-2 lists the registers valid only for SM40. Table C-3 lists the instructions valid only for SM30. Table C-4 lists the registers valid only for SM30. It is illegal to use instructions and registers from these two groups in the same program. With the exception of PS and VS, all other shader types are expected to obey SM40 restrictions.

The SM40 is the preferred shader model.

Table C-1 Instructions for SM40 Only

IL_DCL_CONST_BUFFER	IL_OP_EMIT_THEN_CUT_STREAM	IL_OP_LOAD
IL_DCL_GDS		
IL_DCL_GLOBAL_FLAGS	IL_OP_ENDPHASE	IL_OP_RESINFO
IL_DCL_INDEXED_TEMP_ARRAY	IL_OP_EVAL_CENTROID	IL_OP_SAMPLE
IL_DCL_INPUT	IL_OP_EVAL_SAMPLE_INDEX	IL_OP_SAMPLE_B
IL_DCL_INPUT_PRIMITIVE	IL_OP_EVAL_SNAPPED	IL_OP_SAMPLE_C
IL_DCL_LDS	IL_OP_FCALL	IL_OP_SAMPLE_C_B
IL_DCL_LITERAL	IL_OP_FENCE	IL_OP_SAMPLE_C_G
IL_DCL_MAX_OUTPUT_VERTEX_COUNT	IL_OP_FETCH4	IL_OP_SAMPLE_C_L
IL_DCL_MAX_TESSFACTOR	IL_OP_FETCH4_C	IL_OP_SAMPLE_C_LZ
IL_DCL_NUM_ICP	IL_OP_FETCH4_PO	IL_OP_SAMPLE_G
IL_DCL_NUM_INSTANCES	IL_OP_FETCH4_PO_C	IL_OP_SAMPLE_L
IL_DCL_NUM_OCP	IL_OP_GDS_ADD	
IL_DCL_ODEPTH	IL_OP_GDS_AND	
IL_DCL_OUTPUT	IL_OP_GDS_CMP_STORE	
IL_DCL_OUTPUT_TOPOLOGY	IL_OP_GDS_DEC	
IL_DCL_RESOURCE	IL_OP_GDS_INC	
IL_DCL_STREAM	IL_OP_GDS_MAX	
IL_DCL_STRUCT_GDS	IL_OP_GDS_MIN	
IL_DCL_STRUCT_LDS	IL_OP_GDS_MSKOR	
IL_DCL_TS_DOMAIN	IL_OP_GDS_OR	
IL_DCL_TS_OUTPUT_PRIMITIVE	IL_OP_GDS_READ_ADD	
	IL_OP_GDS_READ_AND	
	IL_OP_GDS_READ_CMP_XCHG	
	IL_OP_GDS_READ_DEC	
	IL_OP_GDS_READ_INC	
	IL_OP_GDS_READ_MAX	
	IL_OP_GDS_READ_MIN	
	IL_OP_GDS_READ_MSKOR	
	IL_OP_GDS_READ_OR	
	IL_OP_GDS_READ_RSUB	
	IL_OP_GDS_READ_SUB	

Table C-1 Instructions for SM40 Only (Cont.)

	IL_OP_GDS_READ_UMAX	
	IL_OP_GDS_READ_UMIN	
	IL_OP_GDS_READ_XCHG	
	IL_OP_GDS_READ_XOR	
	IL_OP_GDS_RSUB	
	IL_OP_GDS_STORE	
	IL_OP_GDS_SUB	
	IL_OP_GDS_UMAX	
	IL_OP_GDS_UMIN	
	IL_OP_GDS_XOR	
	IL_OP_GETLOD	IL_OP_SAMPLEINFO
	IL_OP_HS_CP_PHASE	IL_OP_SRV_RAW_LOAD
	IL_OP_HS_FORK_PHASE	IL_OP_SRV_STRUCT_LOAD
	IL_OP_HS_JOIN_PHASE	IL_OP_U_BIT_INSERT
	IL_OP_INIT_SR	IL_OP_UAV_ADD
	IL_OP_INIT_SR_HELPER	IL_OP_UAV_AND
	IL_OP_LDS_ADD	IL_OP_UAV_ARENA_LOAD
	IL_OP_LDS_AND	IL_OP_UAV_ARENA_STORE
	IL_OP_LDS_CMP	IL_OP_UAV_CMP
	IL_OP_LDS_DEC	
	IL_OP_LDS_INC	
IL_DCL_TS_PARTITION	IL_OP_LDS_LOAD	IL_OP_UAV_LOAD
	IL_OP_LDS_LOAD_BYTE	
	IL_OP_LDS_LOAD_SHORT	
	IL_OP_LDS_LOAD_UBYTE	
	IL_OP_LDS_LOAD_USHORT	
IL_DCL_VPRIM	IL_OP_LDS_LOAD_VEC	IL_OP_UAV_MAX
IL_OP_APPEND_BUF_ALLOC	IL_OP_LDS_MAX	IL_OP_UAV_MIN
IL_OP_APPEND_BUF_CONSUME	IL_OP_LDS_MIN	IL_OP_UAV_OR
IL_OP_BUFINFO	IL_OP_LDS_OR	IL_OP_UAV_RAW_LOAD
IL_OP_CUT	IL_OP_LDS_READ_ADD	IL_OP_UAV_RAW_STORE
IL_OP_CUT_STREAM	IL_OP_LDS_READ_AND	IL_OP_UAV_READ_ADD

Table C-1 Instructions for SM40 Only (Cont.)

IL_OP_DCL_ARENA_UAV	IL_OP_LDS_READ_CMP_XCHG	IL_OP_UAV_READ_AND
	IL_OP_LDS_READ_DEC	
	IL_OP_LDS_READ_INC	
IL_OP_DCL_FUNCTION_BODY	IL_OP_LDS_READ_MAX	IL_OP_UAV_READ_CMP_XCHG
IL_OP_DCL_FUNCTION_TABLE	IL_OP_LDS_READ_MIN	IL_OP_UAV_READ_MAX
IL_OP_DCL_INTERFACE_PTR	IL_OP_LDS_READ_OR	IL_OP_UAV_READ_MIN
IL_OP_DCL_LDS_SHARING_MODE	IL_OP_LDS_READ_RSUB	IL_OP_UAV_READ_OR
IL_OP_DCL_LDS_SIZE_PER_THREAD	IL_OP_LDS_READ_SUB	IL_OP_UAV_READ_RSUB
IL_OP_DCL_NUM_THREAD_PER_GROUP	IL_OP_LDS_READ_UMAX	IL_OP_UAV_READ_SUB
IL_OP_DCL_RAW_SRV	IL_OP_LDS_READ_UMIN	IL_OP_UAV_READ_UMAX
IL_OP_DCL_RAW_UAV	IL_OP_LDS_READ_VEC	IL_OP_UAV_READ_UMIN
IL_OP_DCL_SHARED_TEMP	IL_OP_LDS_READ_XCHG	IL_OP_UAV_READ_XCHG
IL_OP_DCL_STRUCT_SRV	IL_OP_LDS_READ_XOR	IL_OP_UAV_READ_XOR
IL_OP_DCL_STRUCT_UAV	IL_OP_LDS_RSUB	IL_OP_UAV_RSUB
IL_OP_DCL_TOTAL_NUM_THREAD_GROUP	IL_OP_LDS_STORE	IL_OP_UAV_STORE
	IL_OP_LDS_STORE_BYTE	
	IL_OP_LDS_STORE_SHORT	
IL_OP_DCL_UAV	IL_OP_LDS_STORE_VEC	IL_OP_UAV_STRUCT_LOAD
IL_OP_DISCARD_LOGICALNZ	IL_OP_LDS_SUB	IL_OP_UAV_STRUCT_STORE
IL_OP_DISCARD_LOGICALZ	IL_OP_LDS_UMAX	IL_OP_UAV_SUB
		IL_OP_UAV_UDEC
		IL_OP_UAV_UINC
IL_OP_EMIT	IL_OP_LDS_UMIN	IL_OP_UAV_UMAX
IL_OP_EMIT_STREAM	IL_OP_LDS_WRITE_VEC	IL_OP_UAV_UMIN
IL_OP_EMIT_THEN_CUT	IL_OP_LDS_XOR	IL_OP_UAV_XOR

Table C-2 Registers for SM40 Only

IL_REGTYPE_ABSOLUTE_THREAD_ID	IL_REGTYPE_OCP_ID
IL_REGTYPE_ABSOLUTE_THREAD_ID_FLAT	IL_REGTYPE_OMASK
IL_REGTYPE_CONST_BUFF	IL_REGTYPE_OUTPUT
IL_REGTYPE_DEPTH_GE	IL_REGTYPE_OUTPUTCP
IL_REGTYPE_DEPTH_LE	IL_REGTYPE_PATCHCONST
IL_REGTYPE_DOMAINLOCATION	IL_REGTYPE_SHADER_INSTANCE_ID
IL_REGTYPE_GENERIC_MEM	IL_REGTYPE_SHARED_TEMP
IL_REGTYPE_IMMED_CONST_BUFF	IL_REGTYPE_THIS
IL_REGTYPE_INPUT	IL_REGTYPE_THREAD_GROUP_ID
IL_REGTYPE_INPUT_COVERAGE_MASK	IL_REGTYPE_THREAD_GROUP_ID_FLAT
IL_REGTYPE_INPUTCP	IL_REGTYPE_THREAD_ID_IN_GROUP
IL_REGTYPE_ITEMP	IL_REGTYPE_THREAD_ID_IN_GROUP_FLAT
IL_REGTYPE_LITERAL	IL_REGTYPE_TIMER

Table C-3 Instructions for SM30 Only

IL_OP_COLORCLAMP	IL_OP_DCLVOUT	IL_OP_NOISE
IL_OP_DCLDEF	IL_OP_DEF	IL_OP_TEXLD
IL_OP_DCLPI	IL_OP_DEFB	IL_OP_TEXLDB
IL_OP_DCLPIN	IL_OP_DET	IL_OP_TEXLDD
IL_OP_DCLPP	IL_OP_DIST	IL_OP_TEXLDMS
IL_OP_DCLPT	IL_OP_MEMEXPORT	IL_OP_TEXWEIGHT
IL_OP_DCLV	IL_OP_MEMIMPORT	IL_OP_TRANSPOSE

Table C-4 Registers for SM30 Only

IL_REGTYPE_ADDR	IL_REGTYPE_PINPUT
IL_REGTYPE_CLIP	IL_REGTYPE_POS
IL_REGTYPE_CONST_BOOL	IL_REGTYPE_PRICOLOR
IL_REGTYPE_CONST_FLOAT	IL_REGTYPE_SECCOLOR

Table C-4 Registers for SM30 Only (Cont.)

IL_REGTYPE_CONST_INT	IL_REGTYPE_SPRITE
IL_REGTYPE_EDGEFLAG	IL_REGTYPE_SPRITECOORD
IL_REGTYPE_FACE	IL_REGTYPE_TEXCOORD
IL_REGTYPE_FOG	IL_REGTYPE_VERTEX
IL_REGTYPE_INTERP	IL_REGTYPE_VOUTPUT
IL_REGTYPE_PCOLOR	IL_REGTYPE_WINCOORD

C.3 Instruction Restrictions

Within each shader models, not all instructions can be used together. Note the following guidelines.

- `DCL_NUM_THREAD_PER_GROUP` and `DCL_MAX_THREAD_PER_GROUP` cannot be used in the same program.
- `DCL_STREAM`, `EMIT_STREAM`, `CUT_STREAM`, `EMIT_THEN_CUT_STREAM` are not compatible with `EMIT`, `CUT`, `EMIT_THEN_CUT`.
- Scatter operations and UAV operations cannot be used in the same program.
- `DCLPI` and `DCLV` are not compitable with `DCLPIN` and `DCLVOUT`.

Appendix D Device Validity for IL Instructions

Instruction Name	Valid Devices	Page Number
Flow Control Instructions		
BREAK	Valid for all GPUs.	page 7-13
BREAKC	Valid for all GPUs.	page 7-13
BREAK_LOGICALNZ, BREAK_LOGICALZ	Valid for all GPUs.	page 7-14
CALL	Valid for all GPUs.	page 7-14
CALLNZ	Valid for all GPUs.	page 7-15
CALL_LOGICALNZ	Valid for all GPUs.	page 7-16
CALL_LOGICALZ	Valid for all GPUs.	page 7-17
CASE	Valid for R6XX GPUs and later.	page 7-18
CONTINUE	Valid for R6XX GPUs and later.	page 7-18
CONTINUEC	Valid for R6XX GPUs and later.	page 7-19
CONTINUE_LOGICALNZ, CONTINUE_LOGICALZ	Valid for R6XX GPUs and later.	page 7-19
DEFAULT	Valid for R6XX GPUs and later.	page 7-19
ELSE	Valid for all GPUs.	page 7-20
END	Valid for all GPUs.	page 7-21
ENDFUNC	Valid for all GPUs.	page 7-21
ENDIF	Valid for all GPUs.	page 7-22
ENDLOOP	Valid for all GPUs.	page 7-22
ENDMAIN	Valid for all GPUs.	page 7-23
ENDPHASE	Valid for 8XX GPUs and later.	page 7-23
ENDSWITCH	Valid for R6XX GPUs and later.	page 7-24
FUNC	Valid for all GPUs.	page 7-24
HS_CP_PHASE	Valid for Evergreen GPUs and later.	page 7-25
HS_FORK_PHASE	Valid for Evergreen GPUs and later.	page 7-25
HS_JOIN_PHASE	Valid for Evergreen GPUs and later.	page 7-26
IF_LOGICALNZ, IF_LOGICALZ	Valid for R6XX GPUs and later.	page 7-26
IFC	Valid for all GPUs.	page 7-27
IFNZ	Valid for all GPUs.	page 7-27
LOOP	Valid for all GPUs.	page 7-28
RET	Valid for all GPUs.	page 7-29

Instruction Name	Valid Devices	Page Number
RET_DYN	Valid for R600 GPUs and later.	page 7-29
RET_LOGICALNZ, RET_LOGICALZ	Valid for R6XX GPUs and later.	page 7-30
SWITCH	Valid for R6XX GPUs and later.	page 7-30
WHILELOOP	Valid for R600 GPUs and later.	page 7-31
Declaration and Initialization Instructions		
DCL_CB	Valid for R600 GPUs and later.	page 7-32
DCL_GLOBAL_FLAGS flag1 | flag2 | ...	The `refactoring_allowed` parameter is valid for R6XX GPUs and later. The `force_early_depth_stencil` parameter is valid for Evergreen GPUs and later. The `enable_raw_structured_buffers` parameter is valid for Evergreen GPUs and later. The `enable_double_precision_float_ops` parameter is valid for Evergreen GPUs and later.	page 7-33
DCL_INDEXED_TEMP_ARRAY	Valid for R600 GPUs and later.	page 7-34
DCL_INPUT	Valid for R600 GPUs and later.	page 7-35
DCL_INPUTPRIMITIVE	Valid for R600 GPUs and later.	page 7-37
DCL_LDS_SHARING_MODE	Valid only for R7XX GPUs.	page 7-37
DCL_LDS_SIZE_PER_THREAD	Valid only for R7XX GPUs.	page 7-38
DCL_LITERAL	Valid for R600 GPUs and later.	page 7-39
DCL_MAX_OUTPUT_VERTEX_COUNT	Valid for R600 GPUs and later.	page 7-40
DCL_MAX_TESSFACTOR	Valid for Evergreen GPUs and later.	page 7-41
DCL_MAX_THREAD_PER_GROUP	Valid for Evergreen GPUs and later.	page 7-41
DCL_NUM_ICP	Valid for Evergreen GPUs and later.	page 7-42
DCL_NUM_INSTANCES	Valid for Evergreen GPUs and later.	page 7-42
DCL_NUM_OCP	Valid for Evergreen GPUs and later.	page 7-43
DCL_NUM_THREAD_PER_GROUP	Valid for R7XX GPUs and later.	page 7-43
DCL_ODEPTH	Valid for R600 GPUs and later.	page 7-44
DCL_OUTPUT	Valid for R600 GPUs and later.	page 7-45
DCL_OUTPUT_TOPOLOGY	Valid for R600 GPUs and later.	page 7-46
DCL_PERSISTENT	Valid for R670 GPUs only.	page 7-46
DCL_RESOURCE	Valid for R600 and later; valid for shader model 4 (SM4) and later.	page 7-47
DCL_SHARED_TEMP	Valid for R700 GPUs and later.	page 7-48
DCL_STREAM	Valid for Evergreen GPUs and later.	page 7-49
DCL_TOTAL_NUM_THREAD_GROUP	Valid for R7XX GPUs and later.	page 7-50
DCL_TS_DOMAIN	Valid for Evergreen GPUs and later.	page 7-50
DCL_TS_OUTPUT_PRIMITIVE	Valid for RXX GPUs and later.	page 7-51
DCL_TS_PARTITION	Valid for Evergreen GPUs and later.	page 7-51
DCL_VPRIM	Valid for R600 GPUs and later.	page 7-52
DCLARRAY	Valid for all GPUs.	page 7-52
DCLDEF	Valid for all GPUs.	page 7-53

Instruction Name	Valid Devices	Page Number
DCLPI	Valid for all GPUs.	page 7-54
DCLPIN	Valid for all GPUs.	page 7-56
DCLPP	Valid for all GPUs.	page 7-58
DCLPT	Valid for all GPUs.	page 7-59
DCLV	Valid for all GPUs.	page 7-60
DCLVOUT	Valid for all GPUs.	page 7-62
DEF	Valid for all GPUs.	page 7-64
DEFB	Valid for R600 GPUs and later.	page 7-65
INIT_SHARED_REGISTERS	Valid for R7XX GPUs and later.	page 7-66
INITV	Valid for all GPUs.	page 7-67
Input/Output Instructions		
BUFINFO	Valid for Evergreen GPUs and later.	page 7-68
CUT	Valid for R600 GPUs and later.	page 7-68
CUT_STREAM	Valid for Evergreen GPUs and later.	page 7-69
DISCARD_LOGICALNZ, DISCARD_LOGICALZ	Valid for R600 GPUs and later.	page 7-69
EMIT	Valid for R600 GPUs and later.	page 7-70
EMIT_STREAM	Valid for Evergreen GPUs and later.	page 7-70
EMIT_THEN_CUT	Valid for R600 GPUs and later.	page 7-71
EMIT_THEN_CUT_STREAM	Valid for Evergreen GPUs and later.	page 7-71
EVAL_CENTROID	Valid for Evergreen GPUs and later.	page 7-72
EVAL_SAMPLE_INDEX	Valid for Evergreen GPUs and later.	page 7-72
EVAL_SNAPPED	Valid for Evergreen GPUs and later.	page 7-73
FENCE	Valid for R7XX GPUs and later.	page 7-74
FETCH4	The first syntax example is valid for R5XX GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-76
FETCH4_PO_C	Valid for Evergreen GPUs and later. The second syntax example supports indexing.	page 7-77
FETCH4C	Valid for Evergreen GPUs and later. The second syntax example supports indexing.	page 7-78
FETCH4po	Valid for Evergreen GPUs and later. The second syntax example supports indexing.	page 7-79
KILL	Valid for all GPUs.	page 7-80
LDS_READ_VEC	Valid only for R7XX GPUs.	page 7-81
LDS_WRITE_VEC	Valid for R7XX GPUs only.	page 7-82
LOAD	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later.	page 7-83
LOAD_FPTR	Valid for Evergreen GPUs and later.	page 7-85
LOD	Valid for R600 GPUs and later.	page 7-86
MEMEXPORT	Valid for R670 GPUs and later.	page 7-87
MEMIMPORT	Valid for R670 GPUs and later.	page 7-88

Instruction Name	Valid Devices	Page Number
RESINFO	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later.	page 7-89
SAMPLE	The first syntax example above is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second syntax example supports indexing.	page 7-90
SAMPLE_B	The first syntax example above is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second one also supports indexing.	page 7-92
SAMPLE_C	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing. Source src0 is the index; source src1.x has the reference value.	page 7-93
SAMPLE_C_B	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-94
SAMPLE_C_G	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing. For R7xx and later GPUs, this instruction works on all resource types, other than buffers. For R6xx GPUs, this instruction works on non-array resource types, other than buffers. This instruction produces undefined results if it is used on an unsupported format.	page 7-95
SAMPLE_C_L	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-96
SAMPLE_C_LZ	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-97
SAMPLE_G	The first syntax example (not indexed) is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-98
SAMPLE_L	The first syntax example is valid for R600 GPUs and later. The second is valid for Evergreen GPUs and later. The second also supports indexing.	page 7-99
SAMPLEINFO	The first syntax example above is valid for R670 GPUs and later. The second syntax example above is valid for Evergreen GPUs and later.	page 7-100
SAMPLEPOS	The first syntax example above is valid for R670 GPUs and later. The second syntax example above is valid for Evergreen GPUs and later.	page 7-101
SCATTER	Valid for R520 GPUs only.	page 7-102
TEXLD	Valid for all GPUs.	page 7-103
TEXLDB	Valid for all GPUs.	page 7-106
TEXLDD	Valid for all GPUs.	page 7-110
TEXLDMS	Valid for R6XX GPUs and later.	page 7-113

Instruction Name	Valid Devices	Page Number
TEXWEIGHT	Valid for all GPUs.	page 7-115
Integer Arithmetic Instructions		
I64_ADD	Valid for Evergreen GPUs and later.	page 7-116
I64EQ, I64GE, I64LT, I64NE	All these instructions are valid for Evergreen GPUs and later.	page 7-117
I64MAX, I64MIN	Both instructions are valid for Evergreen GPUs and later.	page 7-118
I64NEGATE	Valid for Evergreen GPUs and later.	page 7-119
I64SHL, I64SHR	Valid for Evergreen GPUs and later.	page 7-120
IADD	Valid for R600 GPUs and later.	page 7-121
IAND	Valid for R600 GPUs and later.	page 7-121
IBORROW	Valid for Evergreen GPUs and later.	page 7-122
ICARRY	Valid for Evergreen GPUs and later.	page 7-122
IEQ, IGE, ILT, INE	All these instructions are valid for R600 GPUs and later.	page 7-123
IMAD	Valid for R600 GPUs and later.	page 7-124
IMAD24	Valid for Northern Islands GPUs and later.	page 7-124
IMAX, IMIN	Both instructions are valid for R600 GPUs and later.	page 7-125
IMUL	Valid for R600 GPUs and later.	page 7-126
IMUL_HIGH	Valid for R600 GPUs and later.	page 7-126
IMUL24	Valid for Northern Islands GPUs and later.	page 7-127
IMUL24_HIGH	Valid for Northern Islands GPUs and later.	page 7-127
INEGATE	Valid for R600 GPUs and later.	page 7-128
INOT	Valid for R600 GPUs and later.	page 7-128
IOR, IXOR	These instructions are valid for R600 GPUs and later.	page 7-129
ISHL, ISHR	Valid for R600 GPUs and later.	page 7-130
Unsigned Integer Operations		
U64MAX, U64MIN	Both instructions are valid for Evergreen GPUs and later.	page 7-131
UDIV	Valid for R600 GPUs and later.	page 7-132
U64GE, U64LT	These instructions are valid for Evergreen GPUs and later.	page 7-132
U64SHR	Valid for Evergreen GPUs and later.	page 7-133
UGE, ULT	These instructions are valid for R600 GPUs and later.	page 7-133
UMAD	Valid for R600 GPUs and later.	page 7-134
UMAD24	Valid for Evergreen GPUs and later.	page 7-134
UMAX, UMIN	Both instructions are valid for R600 GPUs and later.	page 7-135
UMOD	Valid for R600 GPUs and later.	page 7-136
UMUL	Valid for R600 GPUs and later.	page 7-136
UMUL_HIGH	Valid for R600 GPUs and later.	page 7-137
UMUL24	Valid for Evergreen GPUs and later.	page 7-137

Instruction Name	Valid Devices	Page Number
UMUL24_high	Valid for Evergreen GPUs and later.	page 7-138
USHR	Valid for R600 GPUs and later.	page 7-138
Bit Operations		
IBIT_EXTRACT	Valid for Evergreen GPUs and later.	page 7-139
ICOUNTBITS	Valid for Evergreen GPUs and later.	page 7-140
IFIRSTBIT	Valid for Evergreen GPUs and later.	page 7-140
UBIT_EXTRACT	Valid for Evergreen GPUs and later.	page 7-141
UBIT_INSERT	Valid for Evergreen GPUs and later.	page 7-142
UBIT_REVERSE	Valid for Evergreen GPUs and later.	page 7-142
Conversion Instructions		
D2F	Valid for all GPUs that support double precision.	page 7-143
F2D	Valid for all GPUs that support double precision.	page 7-144
F_2_F16	Valid for Evergreen GPUs and later with double-precision floating-point.	page 7-144
F16_2_F	Valid for Evergreen GPUs and later with double-precision floating-point.	page 7-145
FTOI	Valid for R600 GPUs and later.	page 7-145
FTOU	Valid for R600 GPUs and later.	page 7-146
ITOF	Valid for R600 GPUs and later.	page 7-146
UTOF	Valid for R600 GPUs and later.	page 7-147
Float Instructions		
ABS	Valid for all GPUs.	page 7-148
ACOS	Valid for all GPUs.	page 7-148
ADD	Valid for R600 GPUs and later.	page 7-149
AND	Valid for R6XX GPUs and later.	page 7-149
ASIN	Valid for all GPUs.	page 7-150
ATAN	Valid for all GPUs.	page 7-151
CLAMP	Valid for all GPUs.	page 7-152
CMOV	Valid for all GPUs.	page 7-153
CMOV_LOGICAL	Valid for R600 GPUs and later.	page 7-154
CMP	Valid on all GPUs.	page 7-155
COLORCLAMP	Valid for all GPUs.	page 7-156
COS	Valid for all GPUs.	page 7-157
COS_VEC	Valid for R600 GPUs and later.	page 7-157
CRS	Valid for all GPUs.	page 7-158
DET	Valid for all GPUs.	page 7-159
DIST	Valid for all GPUs.	page 7-160
DIV	Valid for all GPUs.	page 7-161
DP2	Valid for all GPUs.	page 7-162
DP2ADD	Valid for R600 GPUs and later.	page 7-163
DP3	Valid for all GPUs.	page 7-164
DP4	Valid for all GPUs.	page 7-165

Instruction Name	Valid Devices	Page Number
DST	Valid for all GPUs.	page 7-166
DSX	Valid for all GPUs.	page 7-167
DSY	Valid for all GPUs.	page 7-168
DXSINCOS	Valid for all GPUs.	page 7-169
EXN	Valid for all GPUs.	page 7-170
EQ	Valid for R600 GPUs and later.	page 7-170
EXP	Valid for all GPUs.	page 7-171
EXP_VEC	Valid for R600 GPUs and later.	page 7-171
EXPP	Valid for all GPUs.	page 7-172
FACEFORWARD	Valid for all GPUs.	page 7-172
FLR	Valid for all GPUs.	page 7-173
FMA	Valid for Evergreen GPUs and later with double precision capability.	page 7-173
FRC	Valid for all GPUs.	page 7-174
FWIDTH	Valid for all GPUs.	page 7-175
GE	Valid for R600 GPUs and later.	page 7-175
INV_MOV	Valid for R600 GPUs and later.	page 7-176
LEN	Valid for all GPUs.	page 7-176
LIT	Valid for all GPUs.	page 7-177
LN	Valid for all GPUs.	page 7-178
LOG	Valid for all GPUs.	page 7-179
LOG_VEC	Valid for R600 GPUs and later.	page 7-180
LOGP	Valid for all GPUs.	page 7-181
LRP	Valid for all GPUs.	page 7-182
LT	Valid for R6XX GPUs and later.	page 7-183
MAD	Valid for R600 GPUs and later.	page 7-184
MAX	Valid for R600 GPUs and later that support the IEEE flag.	page 7-185
MIN	Valid for R600 GPUs and later that support the IEEE flag.	page 7-186
MMUL	Valid for all GPUs.	page 7-187
MOD	Valid for all GPUs.	page 7-188
MOV	Valid for all GPUs.	page 7-189
MUL	Valid for all GPUs. For R600 GPUs, IEEE flag was added.	page 7-189
NE	Valid for R600 GPUs and later.	page 7-190
NOISE	Valid for all GPUs.	page 7-191
NRM	Valid for all GPUs.	page 7-192
PIREDUCE	Valid for all GPUs.	page 7-193
POW	Valid for all GPUs.	page 7-194
POWER	Valid for all GPUs.	page 7-195
RCP	Valid for all GPUs.	page 7-196

Instruction Name	Valid Devices	Page Number
REFLECT	Valid for all GPUs.	page 7-197
RND	Valid for R600 GPUs and later.	page 7-198
ROUND_NEAREST	Valid for all GPUs.	page 7-198
ROUND_NEG_INF	Valid for R600 GPUs and later.	page 7-199
ROUND_PLUS_INF	Valid for R600 GPUs and later.	page 7-199
ROUND_ZERO	Valid for R600 GPUs and later.	page 7-200
RSQ	Valid for all GPUs.	page 7-201
RSQ_VEC	Valid for R600 GPUs and later.	page 7-202
SET	Valid for all GPUs.	page 7-202
SGN	Valid for all GPUs.	page 7-203
SIN	Valid for R600 GPUs and later.	page 7-204
SINCOS	Valid for all GPUs.	page 7-205
SIN_VEC	Valid for R600 GPUs and later.	page 7-205
SQRT	Valid for all GPUs.	page 7-206
SQRT_VEC	Valid for R600 GPUs and later.	page 7-206
SUB	Valid for all GPUs.	page 7-207
TAN	Valid for all GPUs.	page 7-208
TRANSPOSE	Valid for all GPUs.	page 7-209
Double-Precision Instructions		
DADD	Valid for all GPUs that support double floating-point operations.	page 7-210
D_DIV	Valid for all GPUs that support double floating-points.	page 7-211
D_EQ	Valid for R670 GPUs and later that support double floating-points.	page 7-211
D_FRAC	Valid for all GPUs that support double floating-points.	page 7-212
D_FREXP	Valid for all GPUs that support double floating-points.	page 7-212
D_GE	Valid for R670 GPUs and later.	page 7-213
D_LDEXP	Valid for all GPUs that support double floating-points.	page 7-214
D_LT	Valid for all GPUs from the R6XX series that support double floating-point.	page 7-215
D_MAX	Valid for Evergreen GPUs and later that support double-precision floating-points.	page 7-216
D_MIN	Valid for Evergreen GPUs and later that support double-precision floating-points.	page 7-216
D_MOV	Valid for Evergreen GPUs and later that support double floating-point.	page 7-217
D_MOVC	Valid for Evergreen GPUs and later that support double floating-point.	page 7-217
D_MUL	Valid for all GPUs that support double floating-points.	page 7-218
D_MULADD	Valid for all GPUs that support double floating-points.	page 7-219
D_NE	Valid for R670 GPUs and later that support double floating-points	page 7-219
D_RCP	Valid for Evergreen GPUs and later that support double floating-point.	page 7-220

Instruction Name	Valid Devices	Page Number
D_RSQ	Valid for Evergreen GPUs and later that support double floating-point.	page 7-221
D_SQRT	Valid for Evergreen GPUs and later that support double floating-points.	page 7-221
Multi-Media Instructions		
BitAlign	Valid for Evergreen GPUs and later.	page 7-222
ByteAlign	Valid for Evergreen GPUs and later.	page 7-222
F_2_u4	Valid for Evergreen GPUs and later.	page 7-223
SAD	Valid for Evergreen GPUs and later.	page 7-223
SAD_HI	Valid for Evergreen GPUs and later.	page 7-224
SAD4	Valid for Evergreen GPUs and later.	page 7-224
U4LERP	Valid for Evergreen GPUs and later.	page 7-225
Unpack0	Valid for Evergreen GPUs and later.	page 7-225
Unpack1	Valid for Evergreen GPUs and later.	page 7-226
Unpack2	Valid for Evergreen GPUs and later.	page 7-226
Unpack3	Valid for Evergreen GPUs and later.	page 7-227
Miscellaneous Special Instructions		
COMMENT	Valid for all GPUs.	page 7-228
NOP	Valid for all GPUs.	page 7-228
Evergreen GPU Series Memory Controls		
APPEND_BUF_ALLOC	Valid for Evergreen GPUs and later.	page 7-229
APPEND_BUF_CONSUME	Valid for Evergreen GPUs and later.	page 7-230
DCL_ARENA_UAV	Valid only for Evergreen and Northern Islands GPUs.	page 7-231
DCL_LDS	Valid for R7XX GPUs and later.	page 7-232
DCL_RAW_SRV	Valid for R7XX GPUs and later.	page 7-232
DCL_RAW_UAV	Valid for R700 GPUs and later.	page 7-233
DCL_STRUCT_LDS	Valid for R7XX GPUs and later.	page 7-233
DCL_STRUCT_SRV	Valid for R7XX GPUs and later.	page 7-234
DCL_STRUCT_UAV	Valid for R700 GPUs and later.	page 7-235
DCL_UAV	Valid for Evergreen GPUs and later.	page 7-236
LDS Instructions		
LDS_ADD	Valid for Evergreen GPUs and later.	page 7-237
LDS_AND	Valid for Evergreen GPUs and later.	page 7-238
LDS_CMP	Valid for Evergreen GPUs and later.	page 7-238
LDS_DEC	Valid for Evergreen GPUs and later.	page 7-239
LDS_INC	Valid for Evergreen GPUs and later.	page 7-239
LDS_LOAD	Valid for R700 GPUs and later.	page 7-240
LDS_LOAD_BYTE	Valid for Evergreen GPUs and later.	page 7-240
LDS_LOAD_SHORT	Valid for Evergreen GPUs and later.	page 7-241
LDS_LOAD_UBYTE	Valid for Evergreen GPUs and later.	page 7-241
LDS_LOAD_USHORT	Valid for Evergreen GPUs and later.	page 7-242
LDS_LOAD_VEC	Valid for R700 GPUs and later.	page 7-243

Instruction Name	Valid Devices	Page Number
LDS_MAX	Valid for Evergreen GPUs and later.	page 7-244
LDS_MIN	Valid for Evergreen GPUs and later.	page 7-244
LDS_MSKOR	Valid for Evergreen GPUs and later.	page 7-245
LDS_OR	Valid for Evergreen GPUs and later.	page 7-245
LDS_READ_ADD	Valid for Evergreen GPUs and later.	page 7-246
LDS_READ_AND	Valid for Evergreen GPUs and later.	page 7-247
LDS_READ_CMP_XCHG	Valid for Evergreen GPUs and later.	page 7-248
LDS_READ_MAX	Valid for Evergreen GPUs and later.	page 7-249
LDS_READ_MIN	Valid for Evergreen GPUs and later.	page 7-250
LDS_READ_OR	Valid for Evergreen GPUs and later.	page 7-251
LDS_READ_RSUB	Valid for Evergreen GPUs and later.	page 7-252
LDS_READ_SUB	Valid for Evergreen GPUs and later.	page 7-253
LDS_READ_UMAX	Valid for Evergreen GPUs and later.	page 7-254
LDS_READ_UMIN	Valid for Evergreen GPUs and later.	page 7-255
LDS_READ_XCHG	Valid for Evergreen GPUs and later.	page 7-256
LDS_READ_XOR	Valid for Evergreen GPUs and later.	page 7-257
LDS_RSUB	Valid for Evergreen GPUs and later.	page 7-257
LDS_STORE	Valid for R700 GPUs and later.	page 7-258
LDS_STORE_BYTE	Valid for Evergreen GPUs and later.	page 7-258
LDS_STORE_SHORT	Valid for Evergreen GPUs and later.	page 7-259
LDS_STORE_VEC	Valid for R700 GPUs and later.	page 7-260
LDS_SUB	Valid for Evergreen GPUs and later.	page 7-261
LDS_UMAX	Valid for Evergreen GPUs and later.	page 7-261
LDS_UMIN	Valid for Evergreen GPUs and later.	page 7-262
LDS_XOR	Valid for Evergreen GPUs and later.	page 7-262
SRV_RAW_LOAD	Valid for R7XX GPUs and later.	page 7-263
SRV_STRUCT_LOAD	Valid for R7XX GPUs and later.	page 7-264
UAV_ADD	Valid for Evergreen GPUs and later.	page 7-265
UAV_AND	Valid for Evergreen GPUs and later.	page 7-266
UAV_ARENA_LOAD	Valid only for Evergreen and Northern Islands GPUs.	page 7-267
UAV_ARENA_STORE	Valid only for Evergreen and Northern Islands GPUs.	page 7-268
UAV_CMP	Valid for Evergreen GPUs and later.	page 7-269
UAV_LOAD	Valid for Evergreen GPUs and later.	page 7-270
UAV_MAX	Valid for Evergreen GPUs and later.	page 7-271
UAV_MIN	Valid for Evergreen GPUs and later.	page 7-272
UAV_OR	Valid for Evergreen GPUs and later.	page 7-273
UAV_RAW_LOAD	Valid for R700 GPUs and later.	page 7-274
UAV_RAW_STORE	Valid for R700 GPUs and later. For R7XX GPUs, only a single UAV is allowed.	page 7-275
UAV_READ_ADD	Valid for Evergreen GPUs and later.	page 7-276
UAV_READ_AND	Valid for Evergreen GPUs and later.	page 7-277

Instruction Name	Valid Devices	Page Number
UAV_READ_CMP_XCHG	Valid for Evergreen GPUs and later.	page 7-278
UAV_READ_MAX	Valid for Evergreen GPUs and later.	page 7-279
UAV_READ_MIN	Valid for Evergreen GPUs and later.	page 7-280
UAV_READ_OR	Valid for Evergreen GPUs and later.	page 7-281
UAV_READ_RSUB	Valid for Evergreen GPUs and later.	page 7-282
UAV_READ_SUB	Valid for Evergreen GPUs and later.	page 7-283
UAV_READ_UDEC	Valid for Evergreen GPUs and later.	page 7-284
UAV_READ_UINC	Valid for Evergreen GPUs and later.	page 7-285
UAV_READ_UMAX	Valid for Evergreen GPUs and later.	page 7-286
UAV_READ_UMIN	Valid for Evergreen GPUs and later.	page 7-287
UAV_READ_XCHG	Valid for Evergreen GPUs and later.	page 7-288
UAV_READ_XOR	Valid for Evergreen GPUs and later.	page 7-289
UAV_RSUB	Valid for Evergreen GPUs and later.	page 7-290
UAV_STORE	Valid for Evergreen GPUs and later.	page 7-291
UAV_STRUCT_LOAD	Valid for R700 GPUs and later.	page 7-292
UAV_STRUCT_STORE	Valid for R7XX GPUs and later.	page 7-293
UAV_SUB	Valid for Evergreen GPUs and later.	page 7-294
UAV_UDEC	Valid for Evergreen GPUs and later.	page 7-295
UAV_UINC	Valid for Evergreen GPUs and later.	page 7-296
UAV_UMAX	Valid for Evergreen GPUs and later.	page 7-297
UAV_UMIN	Valid for Evergreen GPUs and later.	page 7-298
UAV_XOR	Valid for Evergreen GPUs and later.	page 7-299
GDS Instructions		
DCL_GDS	Valid for Evergreen GPUs and later.	page 7-300
DCL_STRUCT_GDS	Valid for Evergreen GPUs and later.	page 7-301
GDS_ADD	Valid for Evergreen GPUs and later.	page 7-302
GDS_AND	Valid for Evergreen GPUs and later.	page 7-303
GDS_CMP_STORE	Valid for Evergreen GPUs and later.	page 7-304
GDS_DEC	Valid for Evergreen GPUs and later.	page 7-305
GDS_INC	Valid for Evergreen GPUs and later.	page 7-306
GDS_LOAD	Valid for Evergreen GPUs and later.	page 7-307
GDS_MAX	Valid for Evergreen GPUs and later.	page 7-307
GDS_MIN	Valid for Evergreen GPUs and later.	page 7-308
GDS_MSKOR	Valid for Evergreen GPUs and later.	page 7-309
GDS_OR	Valid for Evergreen GPUs and later.	page 7-310
GDS_READ_ADD	Valid for Evergreen GPUs and later.	page 7-311
GDS_READ_AND	Valid for Evergreen GPUs and later.	page 7-312
GDS_READ_CMP_XCHG	Valid for Evergreen GPUs and later.	page 7-313
GDS_READ_DEC	Valid for Evergreen GPUs and later.	page 7-314
GDS_READ_INC	Valid for Evergreen GPUs and later.	page 7-315
GDS_READ_MAX	Valid for Evergreen GPUs and later.	page 7-316

Instruction Name	Valid Devices	Page Number
GDS_READ_MIN	Valid for Evergreen GPUs and later.	page 7-317
GDS_READ_MSKOR	Valid for Evergreen GPUs and later.	page 7-318
GDS_READ_OR	Valid for Evergreen GPUs and later.	page 7-319
GDS_READ_RSUB	Valid for Evergreen GPUs and later.	page 7-320
GDS_READ_SUB	Valid for Evergreen GPUs and later.	page 7-321
GDS_READ_UMAX	Valid for Evergreen GPUs and later.	page 7-322
GDS_READ_UMIN	Valid for Evergreen GPUs and later.	page 7-323
GDS_READ_XOR	Valid for Evergreen GPUs and later.	page 7-324
GDS_RSUB	Valid for Evergreen GPUs and later.	page 7-325
GDS_STORE	Valid for Evergreen GPUs and later.	page 7-326
GDS_SUB	Valid for Evergreen GPUs and later.	page 7-327
GDS_UMAX	Valid for Evergreen GPUs and later.	page 7-328
GDS_UMIN	Valid for Evergreen GPUs and later.	page 7-329
GDS_XOR	Valid for Evergreen GPUs and later.	page 7-330
Virtual Function/Interface Support		
DCL_FUNCTION_BODY	Valid for Evergreen GPUs and later.	page 7-331
DCL_FUNCTION_TABLE	Valid for Evergreen GPUs and later.	page 7-331
DCL_INTERFACE_PTR	Valid for Evergreen GPUs and later.	page 7-332
FCALL	Valid for Evergreen GPUs and later.	page 7-333
Macro Processor Support		
MACRODEF	Valid for all GPUs.	page 7-336
MACROEND	Valid for all GPUs.	page 7-337
MCALL	Valid for all GPUs.	page 7-338

Glossary of Terms

Term	**Description**
*	Any number of alphanumeric characters in the name of a microcode format, microcode parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{BUF, SWIZ}	One of the multiple options listed. In this case, the string *BUF* or the string *SWIZ*.
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
0x	Indicates that the following is a hexadecimal number.
1011b	A binary value, in this example a 4-bit value.
29'b0	29 bits with the value 0.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
ABI	Application Binary Interface.
absolute	A displacement that references the base of a code segment, rather than an instruction pointer. See relative.
active mask	A 1-bit-per-pixel mask that controls which pixels in a "quad" are really running. Some pixels might not be running if the current "primitive" does not cover the whole quad. A mask can be updated with a `PRED_SET*` ALU instruction, but updates do not take effect until the end of the ALU clause.
address stack	A stack that contains only addresses (no other state). Used for flow control. Popping the address stack overrides the instruction address field of a flow control instruction. The address stack is only modified if the flow control instruction decides to jump.
ACML	AMD Core Math Library. Includes implementations of the full BLAS and LAPACK routines, FFT, Math transcendental and Random Number Generator routines, stream processing backend for load balancing of computations between the CPU and GPU compute device.
aL (also AL)	Loop register. A three-component vector (x, y and z) used to count iterations of a loop.
allocate	To reserve storage space for data in an output buffer ("scratch buffer," "ring buffer," "stream buffer," or "reduction buffer") or for data in an input buffer ("scratch buffer" or "ring buffer") before exporting (writing) or importing (reading) data or addresses to, or from that buffer. Space is allocated only for data, not for addresses. After allocating space in a buffer, an "export" operation can be done.

Term	**Description**
ALU	Arithmetic Logic Unit. Responsible for arithmetic operations like addition, subtraction, multiplication, division, and bit manipulation on integer and floating point values. In stream computing, these are known as *stream cores*. ALU.[X,Y,Z,W] - an ALU that can perform four vector operations in which the four operands (integers or single-precision floating point values) do not have to be related. It performs “SIMD” operations. Thus, although the four operands need not be related, all four operations execute the same instruction. ALU.Trans - (not relevant on HD 6900 and later devices) An ALU unit that can perform one ALU.Trans (transcendental, scalar) operation, or advanced integer operation, on one integer or single-precision floating-point value, and replicate the result. A single instruction can co-issue four ALU.Trans operations to an ALU.[X,Y,Z,W] unit and one (possibly complex) operation to an ALU.Trans unit, which can then replicate its result across all four component being operated on in the associated ALU.[X,Y,Z,W] unit.
AMD APP KernelAnalyzer	A performance profiling tool for developing, debugging, and profiling stream kernels using high-level stream computing languages.
AR	Address register.
ATI Stream™ SDK	A complete software development suite for developing applications for AMD Accelerated Parallel Processing compute devices. Currently, the ATI Stream SDK includes OpenCL and CAL.
aTid	Absolute work-item ID (formerly *thread ID*). It is the ordinal count of all work-items being executed (in a draw call).
b	A bit, as in *1Mb* for one megabit, or *lsb* for least-significant bit.
B	A byte, as in *1MB* for one megabyte, or *LSB* for least-significant byte.
BLAS	Basic Linear Algebra Subroutines.
border color	Four 32-bit floating-point numbers (XYZW) specifying the border color.
branch granularity	The number of work-items executed during a branch. For AMD GPUs, branch granularity is equal to wavefront granularity.
burst mode	The limited write combining ability. See write combining.
byte	Eight bits.
cache	A read-only or write-only on-chip or off-chip storage space.
CAL	Compute Abstraction Layer. A device-driver library that provides a forward-compatible interface to AMD Accelerated Parallel Processing compute devices. This lower-level API gives users direct control over the hardware: they can directly open devices, allocate memory resources, transfer data and initiate kernel execution. CAL also provides a JIT compiler for AMD IL.
CF	Control Flow.
cfile	Constant file or constant register.
channel	A component in a vector.
clamp	To hold within a stated range.
clause	A group of instructions that are of the same type (all stream core, all fetch, etc.) executed as a group. A clause is part of a CAL program written using the compute device ISA. Executed without pre-emption.

Term	**Description**
clause size	The total number of slots required for an stream core clause.
clause temporaries	Temporary values stored at GPR that do not need to be preserved past the end of a clause.
clear	To write a bit-value of 0. Compare “set”.
command	A value written by the host processor directly to the GPU compute device. The commands contain information that is not typically part of an application program, such as setting configuration registers, specifying the data domain on which to operate, and initiating the start of data processing.
command processor	A logic block in the R700 (HD4000-family of devices) that receives host commands, interprets them, and performs the operations they indicate.
component	(1) A 32-bit piece of data in a “vector”. (2) A 32-bit piece of data in an array. (3) One of four data items in a 4-component register.
compute device	A parallel processor capable of executing multiple work-items of a kernel in order to process streams of data.
compute kernel	Similar to a pixel shader, but exposes data sharing and synchronization.
compute shader	Similar to a pixel shader, but exposes data sharing and synchronization.
compute unit pipeline	A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a compute unit pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
constant buffer	Off-chip memory that contains constants. A constant buffer can hold up to 1024 four-component vectors. There are fifteen constant buffers, referenced as cb0 to cb14. An immediate constant buffer is similar to a constant buffer. However, an immediate constant buffer is defined within a kernel using special instructions. There are fifteen immediate constant buffers, referenced as icb0 to icb14.
constant cache	A constant cache is a hardware object (off-chip memory) used to hold data that remains unchanged for the duration of a kernel (constants). “Constant cache” is a general term used to describe constant registers, constant buffers or immediate constant buffers.
constant file	Same as constant register.
constant index register	Same as “AR” register.
constant registers	On-chip registers that contain constants. The registers are organized as four 32-bit component of a vector. There are 256 such registers, each one 128-bits wide.
constant waterfalling	Relative addressing of a constant file. See waterfalling.
context	A representation of the state of a CAL device.
core clock	See engine clock. The clock at which the GPU compute device stream core runs.
CPU	Central Processing Unit. Also called host. Responsible for executing the operating system and the main part of the application. The CPU provides data and instructions to the GPU compute device.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.

Term	**Description**
CS	Compute shader; commonly referred to as a compute kernel. A shader type, analogous to VS/PS/GS/ES.
CTM	Close-to-Metal. A thin, HW/SW interface layer. This was the predecessor of the AMD CAL.
DC	Data Copy Shader.
device	A *device* is an entire AMD Accelerated Parallel Processing compute device.
DMA	Direct-memory access. Also called DMA engine. Responsible for independently transferring data to, and from, the GPU compute device's local memory. This allows other computations to occur in parallel, increasing overall system performance.
double word	Dword. Two words, or four bytes, or 32 bits.
double quad word	Eight words, or 16 bytes, or 128 bits. Also called "octword."
domain of execution	A specified rectangular region of the output buffer to which work-items are mapped.
DPP	Data-Parallel Processor.
dst.X	The X "slot" of an destination operand.
dword	Double word. Two words, or four bytes, or 32 bits.
element	A component in a vector.
engine clock	The clock driving the stream core and memory fetch units on the GPU compute device.
enum(7)	A seven-bit field that specifies an enumerated set of decimal values (in this case, a set of up to 27 values). The valid values can begin at a value greater than, or equal to, zero; and the number of valid values can be less than, or equal to, the maximum supported by the field.
event	A token sent through a pipeline that can be used to enforce synchronization, flush caches, and report status back to the host application.
export	To write data from GPRs to an output buffer (scratch, ring, stream, frame or global buffer, or to a register), or to read data from an input buffer (a "scratch buffer" or "ring buffer") to GPRs. The term "export" is a partial misnomer because it performs both input and output functions. Prior to exporting, an allocation operation must be performed to reserve space in the associated buffer.
FC	Flow control.
FFT	Fast Fourier Transform.
flag	A bit that is modified by a CF or stream core operation and that can affect subsequent operations.
FLOP	Floating Point Operation.
flush	To writeback and invalidate cache data.
FMA	Fused multiply add.
frame	A single two-dimensional screenful of data, or the storage space required for it.
frame buffer	Off-chip memory that stores a frame. Sometimes refers to the all of the GPU memory (excluding local memory and caches).

Term	**Description**
FS	Fetch subroutine. A global program for fetching vertex data. It can be called by a “vertex shader” (VS), and it runs in the same work-item context as the vertex program, and thus is treated for execution purposes as part of the vertex program. The FS provides driver independence between the process of fetching data required by a VS, and the VS itself. This includes having a semantic connection between the outputs of the fetch process and the inputs of the VS.
function	A subprogram called by the main program or another function within an AMD IL stream. Functions are delineated by `FUNC` and `ENDFUNC`.
gather	Reading from arbitrary memory locations by a work-item.
gather stream	Input streams are treated as a memory array, and data elements are addressed directly.
global buffer	GPU memory space containing the arbitrary address locations to which uncached kernel outputs are written. Can be read either cached or uncached. When read in uncached mode, it is known as mem-import. Allows applications the flexibility to read from and write to arbitrary locations in input buffers and output buffers, respectively.
global memory	Memory for reads/writes between work-items. On HD Radeon 5XXX series devices and later, atomic operations can be used to synchronize memory operations.
GPGPU	General-purpose compute device. A GPU compute device that performs general-purpose calculations.
GPR	General-purpose register. GPRs hold vectors of either four 32-bit IEEE floating-point, or four 8-, 16-, or 32-bit signed or unsigned integer or two 64-bit IEEE double precision data components (values). These registers can be indexed, and consist of an on-chip part and an off-chip part, called the “scratch buffer,” in memory.
GPU	Graphics Processing Unit. An integrated circuit that renders and displays graphical images on a monitor. Also called Graphics Hardware, Compute Device, and Data Parallel Processor.
GPU engine clock frequency	Also called 3D engine speed.
GPU compute device	A parallel processor capable of executing multiple work-items of a kernel in order to process streams of data.
GS	Geometry Shader.
HAL	Hardware Abstraction Layer.
host	Also called CPU.
iff	If and only if.
IL	Intermediate Language. In this manual, the AMD version: AMD IL. A pseudo-assembly language that can be used to describe kernels for GPU compute devices. AMD IL is designed for efficient generalization of GPU compute device instructions so that programs can run on a variety of platforms without having to be rewritten for each platform.
in flight	A work-item currently being processed.
instruction	A computing function specified by the *code* field of an IL_OpCode token. Compare “opcode”, “operation”, and “instruction packet”.
instruction packet	A group of tokens starting with an IL_OpCode token that represent a single AMD IL instruction.

Term	Description
int(2)	A 2-bit field that specifies an integer value.
ISA	Instruction Set Architecture. The complete specification of the interface between computer programs and the underlying computer hardware.
kcache	A memory area containing “waterfall” (off-chip) constants. The cache lines of these constants can be locked. The “constant registers” are the 256 on-chip constants.
kernel	A user-developed program that is run repeatedly on a stream of data. A parallel function that operates on every element of input streams. A device program is one type of kernel. Unless otherwise specified, an AMD Accelerated Parallel Processing compute device program is a kernel composed of a main program and zero or more functions. Also called Shader Program. This is not to be confused with an OS kernel, which controls hardware.
LAPACK	Linear Algebra Package.
LDS	Local Data Share. Part of local memory. These are read/write registers that support sharing between all work-items in a work-group. Synchronization is required.
LERP	Linear Interpolation.
local memory fetch units	Dedicated hardware that a) processes fetch instructions, b) requests data from the memory controller, and c) loads registers with data returned from the cache. They are run at stream core or engine clock speeds. Formerly called texture units.
LOD	Level Of Detail.
loop index	A register initialized by software and incremented by hardware on each iteration of a loop.
lsb	Least-significant bit.
LSB	Least-significant byte.
MAD	Multiply-Add. A fused instruction that both multiplies and adds.
mask	(1) To prevent from being seen or acted upon. (2) A field of bits used for a control purpose.
MBZ	Must be zero.
mem-export	An AMD IL term random writes to the global buffer.
mem-import	Uncached reads from the global buffer.
memory clock	The clock driving the memory chips on the GPU compute device.
microcode format	An encoding format whose fields specify instructions and associated parameters. Microcode formats are used in sets of two or four. For example, the two mnemonics, `CF_WORD[0,1]` indicate a microcode-format pair, `CF_WORD0` and `CF_WORD1`.
MIMD	Multiple Instruction Multiple Data. – Multiple SIMD units operating in parallel (Multi-Processor System) – Distributed or shared memory
MRT	Multiple Render Target. One of multiple areas of local GPU compute device memory, such as a “frame buffer”, to which a graphics pipeline writes data.
MSAA	Multi-Sample Anti-Aliasing.
msb	Most-significant bit.

Term	**Description**
MSB	Most-significant byte.
neighborhood	A group of four work-items in the same wavefront that have consecutive work-item IDs (Tid). The first Tid must be a multiple of four. For example, work-items with Tid = 0, 1, 2, and 3 form a neighborhood, as do work-items with Tid = 12, 13, 14, and 15.
normalized	A numeric value in the range [a, b] that has been converted to a range of 0.0 to 1.0 using the formula: normalized value = value/ (b–a+ 1)
oct word	Eight words, or 16 bytes, or 128 bits. Same as "double quad word". Also referred to as octa word.
opcode	The numeric value of the *code* field of an "instruction".
opcode token	A 32-bit value that describes the operation of an instruction.
operation	The function performed by an "instruction".
PaC	Parameter Cache.
PCI Express	A high-speed computer expansion card interface used by modern graphics cards, GPU compute devices and other peripherals needing high data transfer rates. Unlike previous expansion interfaces, PCI Express is structured around point-to-point links. Also called PCIe.
PoC	Position Cache.
pop	Write "stack" entries to their associated hardware-maintained control-flow state. The `POP_COUNT` field of the `CF_WORD1` microcode format specifies the number of stack entries to pop for instructions that pop the stack. Compare "push."
pre-emption	The act of temporarily interrupting a task being carried out on a computer system, without requiring its cooperation, with the intention of resuming the task at a later time.
processor	Unless otherwise stated, the AMD Accelerated Parallel Processing compute device.
program	Unless otherwise specified, a program is a set of instructions that can run on the AMD Accelerated Parallel Processing compute device. A device program is a type of kernel.
PS	Pixel Shader, aka pixel kernel.
push	Read hardware-maintained control-flow state and write their contents onto the stack. Compare pop.
PV	Previous vector register. It contains the previous four-component vector result from a ALU.[X,Y,Z,W] unit within a given clause.
quad	For a compute kernel, this consists of four consecutive work-items. For pixel and other shaders, this is a group of 2x2 work-items in the NDRange. Always processed together.
rasterization	The process of mapping work-items from the domain of execution to the SIMD engine. This term is a carryover from graphics, where it refers to the process of turning geometry, such as triangles, into pixels.
rasterization order	The order of the work-item mapping generated by rasterization.
RAT	Random Access Target. Same as UAV. Allows, on DX11 hardware, writes to, and reads from, any arbitrary location in a buffer.
RB	Ring Buffer.

Term	**Description**
register	For a GPU, this is a 128-bit address mapped memory space consisting of four 32-bit components.
relative	Referencing with a displacement (also called offset) from an index register or the loop index, rather than from the base address of a program (the first control flow [CF] instruction).
render backend unit	The hardware units in a processing element responsible for writing the results of a kernel to output streams by writing the results to an output cache and transferring the cache data to memory.
resource	A block of memory used for input to, or output from, a kernel.
ring buffer	An on-chip buffer that indexes itself automatically in a circle.
Rsvd	Reserved.
sampler	A structure that contains information necessary to access data in a resource. Also called Fetch Unit.
SC	Shader Compiler.
scalar	A single data component, unlike a vector which contains a set of two or more data elements.
scatter	Writes (by uncached memory) to arbitrary locations.
scatter write	Kernel outputs to arbitrary address locations. Must be uncached. Must be made to a memory space known as the global buffer.
scratch buffer	A variable-sized space in off-chip-memory that stores some of the “GPRs”.
set	To write a bit-value of 1. Compare “clear”.
shader processor	Pre-OpenCL term that is now deprecated. Also called thread processor.
shader program	User developed program. Also called kernel.
SIMD	Pre-OpenCL term that is now deprecated. Single instruction multiple data unit. – Each SIMD receives independent stream core instructions. – Each SIMD applies the instructions to multiple data elements. Now called a compute unit.
SIMD Engine	Pre-OpenCL term that is now deprecated. A collection of thread processors, each of which executes the same instruction each cycle.
SIMD pipeline	In OpenCL terminology: compute unit pipeline. Pre-OpenCL term that is now deprecated. A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a SIMD pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
Simultaneous Instruction Issue	Input, output, fetch, stream core, and control flow per SIMD engine.
slot	A position, in an “instruction group,” for an “instruction” or an associated literal constant. An ALU instruction group consists of one to seven slots, each 64 bits wide. All ALU instructions occupy one slot, except double-precision floating-point instructions, which occupy either two or four slots. The size of an ALU clause is the total number of slots required for the clause.
SPU	Shader processing unit.

Term	**Description**
SR	Globally shared registers. These are read/write registers that support sharing between all wavefronts in a SIMD (not a work-group). The sharing is column sharing, so work-items with the same work-item ID within the wavefront can share data. All operations on SR are atomic.
src0, src1, etc.	In floating-point operation syntax, a 32-bit source operand. Src0_64 is a 64-bit source operand.
stage	A sampler and resource pair.
stream	A collection of data elements of the same type that can be operated on in parallel.
stream buffer	A variable-sized space in off-chip memory that stores an instruction stream. It is an output-only buffer, configured by the host processor. It does not store inputs from off-chip memory to the processor.
stream core	The fundamental, programmable computational units, responsible for performing integer, single, precision floating point, double precision floating point, and transcendental operations. They execute VLIW instructions for a particular work-item. Each processing element handles a single instruction within the VLIW instruction.
stream operator	A node that can restructure data.
swizzling	To copy or move any component in a source vector to any element-position in a destination vector. Accessing elements in any combination.
thread	Pre-OpenCL term that is now deprecated. One invocation of a kernel corresponding to a single element in the domain of execution. An instance of execution of a shader program on an ALU. Each thread has its own data; multiple threads can share a single program counter. Generally, in OpenCL terms, there is a one-to-one mapping of work-items to threads.
thread group	Pre-OpenCL term that is now deprecated. It contains one or more thread blocks. Threads in the same thread-group but different thread-blocks might communicate to each through global per-SIMD shared memory. This is a concept mainly for global data share (GDS). A thread group can contain one or more wavefronts, the last of which can be a partial wavefront. All wavefronts in a thread group can run on only one SIMD engine; however, multiple thread groups can share a SIMD engine, if there are enough resources. Generally, in OpenCL terms, there is a one-to-one mapping of work-groups to thread groups.
thread processor	Pre-OpenCL term that is now deprecated. The hardware units in a SIMD engine responsible for executing the threads of a kernel. It executes the same instruction per cycle. Each thread processor contains multiple stream cores. Also called shader processor.
thread-block	Pre-OpenCL term that is now deprecated. A group of threads which might communicate to each other through local per SIMD shared memory. It can contain one or more wavefronts (the last wavefront can be a partial wavefront). A thread-block (all its wavefronts) can only run on one SIMD engine. However, multiple thread blocks can share a SIMD engine, if there are enough resources to fit them in.
Tid	Work-item id (previously called a *thread id*) within a thread block. An integer number from 0 to Num_threads_per_block-1
token	A 32-bit value that represents an independent part of a stream or instruction.
UAV	Unordered Access View. Same as random access target (RAT). They allow compute shaders to store results in (or write results to) a buffer at any arbitrary location. On DX11 hardware, UAVs can be created from buffers and textures. On DX10 hardware, UAVs cannot be created from typed resources (textures).

Term	**Description**
uncached read/write unit	The hardware units in a GPU compute device responsible for handling uncached read or write requests from local memory on the GPU compute device.
vector	(1) A set of up to four related values of the same data type, each of which is an element. For example, a vector with four elements is known as a “4-vector” and a vector with three elements is known as a “3-vector”. (2) See “AR”. (3) See ALU.[X,Y,Z,W].
VLIW design	Very Long Instruction Word. – Co-issued up to 6 operations (5 stream cores + 1 FC); where FC = flow control. – 1.25 Machine Scalar operation per clock for each of 64 data elements – Independent scalar source and destination addressing
vTid	Work-item ID (formerly *thread ID*) within a work-group.
waterfall	To use the address register (AR) for indexing the GPRs. Waterfall behavior is determined by a “configuration registers.”
wavefront	Group of work-items executed together on a single SIMD engine. Composed of quads. A full wavefront contains 64 work-items; a wavefront with fewer than 64 work-items is called a partial wavefront. Wavefronts that have fewer than a full set of work-items are called partial wavefronts. For the HD4000-family of devices, there are 64. 32, 16 work-items in a full wavefront. Work-items within a wavefront execute in lockstep.
write combining	Combining several smaller writes to memory into a single larger write to minimize any overhead associated with write commands.

Index