OpenCL.DebugInfo.100 Information Extended Instruction Set Specification

Alexey Sotkin, Intel

Version 2.00, Revision 2

December 19, 2019

Copyright © 2014-2019 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast, or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be reformatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group website should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents, or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, SYCL, SPIR, WebGL, EGL, COLLADA, StreamInput, OpenVX, OpenKCam, glTF, OpenKODE, OpenVG, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL and OpenMAX DL are trademarks and WebCL is a certification mark of the Khronos Group Inc. OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Contents

Contributors and Acknowledgments

- Yaxun Liu, AMD
- Brian Sumner, AMD
- Ben Ashbaugh, Intel
- Alexey Bader, Intel
- Raun Krisch, Intel
- Pratik Ashar, Intel
- John Kessenich, Google
- David Neto, Google
- Neil Henning, Codeplay
- Kerch Holt, Nvidia

1 Introduction

This is the specification of **OpenCL.DebugInfo.100** extended instruction set.

The library is imported into a SPIR-V module in the following manner:

```
<extinst-id> OpExtInstImport "OpenCL.DebugInfo.100"
```

The instructions below are capable to convey debug information of the source program.

The design guide lines for these instructions are:

- Sufficient for a backend to generate DWARF debug info for OpenCL C/C++ kernels
- Easy translation between SPIR-V/LLVM
- Clear
- Concise
- Extendable for other languages
- Capable of representing debug information for optimized IR

2 Binary Form

This section contains the semantics of the debug info extended instructions using the **OpExtInst** instruction.

All *Name* operands are id of **OpString** instruction, which represents the name of the entry (type, variable, function. etc) as it appears in the source program.

Result Type of all instructions bellow is id of **OpTypeVoid**

Set operand in all instructions bellow is the result of an **OpExtInstImport** instruction.

All instructions in this extended set has no semantic impact and can be safely removed from the module all at once. Or a single debugging instruction can be removed from the module if all references, to the *Result <id>* of this instruction are replaced with id of DebugInfoNone instruction.

DebugScope, DebugNoScope, DebugDeclare, DebugValue instructions can interleave with instructions within a function

body. All other debugging instructions should be located between section 9 (All type declarations (OpTypeXXX instructions), all constant instructions, and all global variable declarations . . .) and section 10 (All function declaration) per the core SPIR-V specification.

Debug info for source language opaque types is represented by DebugTypeComposite without *Members* operands. *Size* of the composite must be DebugInfoNone and *Name* must start with @ symbol to avoid clashes with user defined names.

3 Enumerations

3.1 Instruction Enumeration

Instruction number	Instruction name
0	DebugInfoNone
1	DebugCompilationUnit
2	DebugTypeBasic
3	DebugTypePointer
4	DebugTypeQualifier
5	DebugTypeArray
6	DebugTypeVector
7	DebugTypedef
8	DebugTypeFunction
9	DebugTypeEnum
10	DebugTypeComposite
11	DebugTypeMember
12	DebugTypeInheritance
13	DebugTypePtrToMember
14	DebugTypeTemplate
15	DebugTypeTemplateParameter
16	DebugTypeTemplateTemplateParameter
17	DebugTypeTemplateParameterPack
18	DebugGlobalVariable
19	DebugFunctionDeclaration
20	DebugFunction
21	DebugLexicalBlock
22	DebugLexicalBlockDiscriminator
23	DebugScope
24	DebugNoScope
25	DebugInlinedAt
26	DebugLocalVariable
27	DebugInlinedVariable
28	DebugDeclare
29	DebugValue
30	DebugOperation
31	DebugExpression
32	DebugMacroDef
33	DebugMacroUndef
34	DebugImportedEntity
35	DebugSource

3.2 Debug Info Flags

Value	Flag Name
1 << 0	**FlagIsProtected**
1 << 1	**FlagIsPrivate**
1<<0 | 1<<1	**FlagIsPublic**
1 << 2	**FlagIsLocal**
1 << 3	**FlagIsDefinition**
1 << 4	**FlagFwdDecl**
1 << 5	**FlagArtificial**
1 << 6	**FlagExplicit**
1 << 7	**FlagPrototyped**
1 << 8	**FlagObjectPointer**
1 << 9	**FlagStaticMember**
1 << 10	**FlagIndirectVariable**
1 << 11	**FlagLValueReference**
1 << 12	**FlagRValueReference**
1 << 13	**FlagIsOptimized**
1 << 14	**FlagIsEnumClass**
1 << 15	**FlagTypePassByValue**
1 << 16	**FlagTypePassByReference**

3.3 Base Type Attribute Encodings

Used by DebugTypeBasic

Encoding code	name
0	**Unspecified**
1	**Address**
2	**Boolean**
3	**Float**
4	**Signed**
5	**SignedChar**
6	**Unsigned**
7	**UnsignedChar**

3.4 Composite Types

Used by DebugTypeComposite

Tag code	name
0	**Class**
1	**Structure**
2	**Union**

3.5 Type Qualifiers

Used by DebugTypeQualifier

Qualifier tag code name	
0	**ConstType**
1	**VolatileType**
2	**RestrictType**
3	**AtomicType**

3.6 Debug Operations

These operations are used to form a DWARF expression. Such expressions provide information about the current location (described by DebugDeclare) or value (described by DebugValue) of a variable. Operations in an expression are to be applied on a stack. Initially the stack contains one element - address or value of the source variable.
Used by DebugExpression

Operation encodings		No. of Operands	Description
0	**Deref**	0	Pops the top stack entry, treats it as an address, pushes the value retrieved from that address.
1	**Plus**	0	Pops the top two entries from the stack, adds them together and push the result.
2	**Minus**	0	Pops the top two entries from the stack, subtracts the former top entry from the former second to top entry and push the result.
3	**PlusUconst**	1	Pops the top stack entry, adds the *addend* operand to it, and pushes the result. The operand must be a single word integer literal.
4	**BitPiece**	2	Describes an object or value which may be contained in part of a register or stored in more than one location. The first operand is *offset* in bit from the location defined by the preceding operation. The second operand is *size* of the piece in bits. The operands must be a single word integer literals.
5	**Swap**	0	Swaps the top two stack values.

Operation encodings		No. of Operands	Description
6	**Xderef**	0	Pops the top two entries from the stack. Treats the former top entry as an address and the former second to top entry as an address space. The value retrieved from the address in the given address space is pushed.
7	**StackValue**	0	Describes an object which doesn't exist in memory but it's value is known and is at the top of the DWARF expression stack.
8	**Constu**	1	Pushes a constant *value* onto the stack. The *value* operand must be a single word integer literal.
9	**Fragment**	2	Has the same semantics as **BitPiece**, but the *offset* operand defines location within the source variable.

3.7 Imported Entities

Used by DebugImportedEntity

Tag code name	
0	**ImportedModule**
1	**ImportedDeclaration**

4 Instructions

4.1 Missing Debugging Information

DebugInfoNone Other instructions can refer to this one in case the debugging information is unknown, not available or not applicable. *Result Type* must be **OpTypeVoid**					
5	12	*<id>* *Result Type*	*Result <id>*	*<id> Set*	0

4.2 Compilation Unit

DebugCompilationUnit

Describe the compilation unit. A SPIR-V module can possibly contain multiple compilation units.

Result Type must be **OpTypeVoid**

Version is version of the SPIRV debug information format.

DWARF Version is version of DWARF standard this specification is compatible with.

Source is a **DebugSource** instruction representing text of the source program.

Language is a source programming language of this particular compilation unit. Possible values of this operand are described in the *Source Language* section of the core SPIR-V specification.

9	12	*<id> Result Type*	*Result <id>*	*<id> Set*	1	*Literal Number Version*	*Literal Number DWARF version*	*<id> Source*	*Language*

DebugSource

Describe the source program. It can be either the primary source file or a file added via `#include` directive

Result Type must be **OpTypeVoid**

File is **OpString** holding the name of the source file including its full path.

Text is **OpString** which contains text of the source program the SPIR-V module is derived from.

6+	12	*<id> Result Type*	*Result <id>*	*<id> Set*	35	*<id> File*	Optional *<id> Text*

4.3 Type instructions

DebugTypeBasic

Describe basic data types.

Result Type must be **OpTypeVoid**

Name is an **OpString** representing the name of the type as it appears in the source program. May be empty.

Size is an **OpConstant** with integral type and its value is amount of storage in bits, needed to hold an instance of the type.

Encoding describes how the base type is encoded.

8	12	*<id> Result Type*	*Result <id>*	*<id> Set*	2	*<id> Name*	*<id> Size*	Encoding

DebugTypePointer

Describe pointer or reference data types.

Result Type must be **OpTypeVoid**

Base Type is *<id>* of debugging instruction which represents the pointee type.

Storage Class is the class of the memory where the pointed object is allocated. Possible values of this operand are described in the *Storage Class* section of the core SPIR-V specification.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

8	12	*<id> Result Type*	*Result <id>*	*<id> Set*	3	*<id> Base Type*	*Storage Class*	*Literal Flags*

DebugTypeQualifier

Describe *const*, *volatile* and *restrict* qualified data types. Types with multiple qualifiers are represented as a sequence of single qualified types.

Result Type must be **OpTypeVoid**

Base Type is debug instruction which represents the type being qualified.

Type Qaulifier is a literal value from the TypeQualifiers table.

7	12	*<id> Result Type*	*Result <id>*	*<id> Set*	4	*<id> Base Type*	Type Qaulifier

DebugTypeArray

Describe array data types

Result Type must be **OpTypeVoid**

Base Type is debugging instruction which describes type of element of the array

Component Count is an **OpConstant** with integral result type, and its value is the number of elements in the corresponding dimension of the array. Number and order of *Component Count* operands must match with number and order of array dimensions as they appear in the source program.

7+	12	*<id> Result Type*	*Result <id>*	*<id> Set*	5	*<id> Base Type*	*<id> Component Count, . . .*

DebugTypeVector

Describe vector data types

Result Type must be **OpTypeVoid**

Base Type is id of debugging instruction which describes type of element of the vector

Component Count is a single *word* literal denoting number of elements in the vector.

7	12	*<id> Result Type*	*Result <id>*	*<id> Set*	6	*<id> Base Type*	*Literal Number Component Count*

DebugTypedef

Describe a C and C++ *typedef declaration*

Result Type must be **OpTypeVoid**

Name is **OpString** which is represents a new name for the *Base Type*

Base Type is a debugging instruction representing the type for which a new name is being declared

Source is a **DebugSource** instruction representing text of the source program containing the typedef declaration.

Line is a single *word* literal denoting the source line number at which the declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the declaration appears on the *Line*.

Parent is a debug instruction which represents the parent lexical scope of the declaration.

11	12	*<id> Result Type*	*Result <id>*	*<id> Set*	7	*<id> Name*	*<id> Base Type*	*<id> Source*	*Literal Number Line*	*Literal Number Column*	*<id> Parent*

DebugTypeFunction

Describe a function type

Result Type must be **OpTypeVoid**

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

Return Type is a debug instruction which represents type of return value of the function. If the function has no return value, this operand is **OpTypeVoid**

Parameter Types are debug instructions which describe type of parameters of the function

7+	12	<id> *Result Type*	*Result* <id>	<id> *Set*	8	*Literal Flags*	<id> *Return Type*	Optional <id>, <id>, . . . *Parameter Types*

DebugTypeEnum

Describe enumeration types

Result Type must be **OpTypeVoid**

Name is an **OpString** holding the name of the enumeration as it appears in the source program.

Underlying Type is a debugging instruction which describes the underlying type of the enum in the source program. If the underlying type is not specified in the source program, this operand must refer to DebugInfoNone.

Source is a **DebugSource** instruction representing text of the source program containing the *enum* declaration.

Line is a single *word* literal denoting the source line number at which the enumeration declaration appears in the *Source*.

Column is a single *word* literal denoting column number at which the first character of the enumeration declaration appears on the *Line*.

Parent is a debug instruction which represents a parent lexical scope.

Size is an **OpConstant** with integral result type, and its value is the number of bits required to hold an instance of the enumeration.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

Enumerators are encoded as trailing pairs of *Value* and corresponding *Name*. *Values* must be id of **OpConstant** instruction, with integer result type. *Name* must be id of **OpString** instruction.

13+	12	<id> *Result Type*	*Result* <id>	<id> *Set*	9	<id> *Name*	<id> *Underlying Type*	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id>, *Parent*	<id> *Size*	*Literal Flags*	<id> *Value*, <id> *Name*, <id> *Value*, <id> *Name*, . . .

DebugTypeComposite

Describe *structure*, *class* and *union* data types

Result Type must be **OpTypeVoid**

Tag is a literal value from the Composite Types table which specifies the kind of the composite type.

Name is an **OpString** holding the name of the type as it appears in the source program

Source is a **DebugSource** instruction representing text of the source program containing the type declaration.

Line is a single *word* literal denoting the source line number at which the type declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the declaration appears on the *Line*

Parent is a debug instruction which represents parent lexical scope. Must be one of the following: DebugCompilationUnit, DebugFunction, DebugLexicalBlock or other DebugTypeComposite

Linkage Name is an **OpString**, holding the linkage name or mangled name of the composite.

Size is an **OpConstant** with integral type and its value is the number of bits required to hold an instance of the composite type.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

Members must be ids of DebugTypeMember, DebugFunction or DebugTypeInheritance.

Note: To represent a source language opaque type this instruction must have no *Members* operands, *Size* operand must be DebugInfoNone and *Name* must start with @ symbol to avoid clashes with user defined names.

14+	12	<id> *Result Type*	*Result* <id>	<id> *Set*	10	<id> *Name*	Tag	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id> *Parent*	<id> *Linkage Name*	<id> *Size*	*Literal Flags*	<id>, <id>, ... *Members*

DebugTypeMember

Describe a data member of a *structure*, *class* or *union*.

Result Type must be **OpTypeVoid**

Name is an **OpString** holding the name of the member as it appears in the source program

Type is a debug type instruction which represents type of the member

Source is a **DebugSource** instruction representing text of the source program containing the member declaration.

Line is a single *word* literal denoting the source line number at which the member declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the member declaration appears on the *Line*.

Parent is a debug instruction which represents a composite type containing this member.

Offset is an **OpConstant** with integral type and its value is offset in bits from the beginning of the *Containig Type*.

Size is an **OpConstant** with integral type and its value is the number of bits the *Base type* occupies within the *Containig Type*.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

Value is an **OpConstant** representing initialization value in case of *const static* qualified member in C++.

14+ 12	<id> *Result Type*	*Result* <id>	<id> *Set*	11	<id> *Name*	<id> *Type*	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id> *Parent*	<id> *Offset*	<id> *Size*	Flags	Optional <id> *Value*

DebugTypeInheritance

Describe inheritance relationship with a parent *class* or *structure*. Result of this instruction should be used as a member of a composite type

Result Type must be **OpTypeVoid**

Child is a debug instruction representing a derived *class* or *struct* in C++.

Parent is a debug instruction representing a class or structure the *Child Type* is derived from.

Offset is an **OpConstant** with integral type and its value is offset of the *Parent Type* in bits in layout of the *Child Type*

Size is an **OpConstant** with integral type and its value is the number of bits the *Parent type* occupies within the *Child Type*.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

10	12	<id> *Result Type*	*Result* <id>	<id> *Set*	12	<id> *Child*	<id> *Parent*	<id> *Offset*	<id> *Size*	Flags

DebugTypePtrToMember

Describe a type of an object that is a pointer to a structure or class member

Result Type must be **OpTypeVoid**

Member Type is a debug instruction representing the type of the member

Parent is a debug instruction, representing a structure or class type.

7	12	<id> *Result Type*	*Result* <id>	<id> *Set*	13	<id> *Member Type*	<id> *Parent*

4.4 Templates

DebugTypeTemplate

Describe an instantiated template of *class*, *struct* or *function* in C++.

Result Type must be **OpTypeVoid**

Target is a debug instruction representing class, struct or function which has template parameter(s).

Parameters are debug instructions representing the template parameters for this particular instantiation.

7	12	<id> *Result Type*	*Result* <id>	<id> *Set*	14	<id> *Target*	<id>... *Parameters*

DebugTypeTemplateParameter

Describe a formal parameter of a C++ template instantiation.

Result Type must be **OpTypeVoid**

Name is an **OpString** holding the name of the template parameter

Actual Type is a debug instruction representing the actual type of the formal parameter for this particular instantiation.

If this instruction describes a template value parameter, the *Value* is represented by an **OpConstant** with integer result type. For template type parameter *Value* operand must not be used

Source is a **DebugSource** instruction representing text of the source program containing the template instantiation.

Line is a single *word* literal denoting the source line number at which the template parameter declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the template parameter declaration appears on the *Line*

11	12	*<id> Result Type*	*Result <id>*	*<id> Set*	15	*<id> Name*	*<id> Actual Type*	*<id> Value*	*<id> Source*	*Literal Number Line*	*Literal Number Column*

DebugTypeTemplateTemplateParameter

Describe a template template parameter of a C++ template instantiation.

Result Type must be **OpTypeVoid**

Name is an **OpString** holding the name of the template template parameter

Template Name is an **OpString** holding the name of the template used as template parameter in this particular instantiation.

Source is a **DebugSource** instruction representing text of the source program containing the template instantiation.

Line is a single *word* literal denoting the source line number at which the template template parameter declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the template template parameter declaration appears on the *Line*

10	12	*<id> Result Type*	*Result <id>*	*<id> Set*	16	*<id> Name*	*<id> Template Name*	*<id> Source*	*Literal Number Line*	*Literal Number Column*

DebugTypeTemplateParameterPack

Describe expanded template parameter pack in a variadic template instantiation in C++

Result Type must be **OpTypeVoid**

Name is an **OpString** holding the name of the template parameter pack

Source is a **DebugSource** instruction representing text of the source program containing the template instantiation.

Line is a single *word* literal denoting the source line number at which the template parameter pack declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the template parameter pack declaration appears on the *Line*

Template parameters are DebugTypeTemplateParameters describing the expanded parameter pack in the variadic template instantiation

10+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	17	<*id*> *Name*	<*id*> *Source*	*Literal Number Line*	*Literal Number Column*	<*id*>... *Template parameters*

4.5 Global Variables

DebugGlobalVariable

Describe a global variable.

Result Type must be **OpTypeVoid**

Name is an **OpString**, holding the name of the variable as it appears in the source program

Type is a debug instruction which represents type of the variable.

Source is a **DebugSource** instruction representing text of the source program containing the global variable declaration.

Line is a single *word* literal denoting the source line number at which the global variable declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the global variable declaration appears on the *Line*

Parent is a debug instruction which represents parent lexical scope. Must be one of the following: DebugCompilationUnit, DebugFunction, DebugLexicalBlock or DebugTypeComposite

Linkage Name is an **OpString**, holding the linkage name of the variable.

Variable is id of the global variable or constant which is described by this instruction. If the variable is optimized out, this operand must be DebugInfoNone.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

If the global variable represents a defining declaration for C++ static data member of a structure, class or union, the optional *Static Member Declaration* operand refers to the debugging type of the previously declared variable, i.e. DebugTypeMember

14+ 12	<id> *Result Type*	*Result* <id>	<id> *Set*	18	<id> *Name*	<id> *Type*	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id> *Parent*	<id> *Linkage Name*	<id> *Variable*	Flags	Optional <id> *Static Member Declaration*

4.6 Functions

DebugFunctionDeclaration

Describe function or method declaration.

Result Type must be **OpTypeVoid**

Name is an **OpString**, holding the name of the function as it appears in the source program

Type is an DebugTypeFunction instruction which represents type of the function.

Source is a **DebugSource** instruction representing text of the source program containing the function declaration.

Line is a single *word* literal denoting the source line number at which the function declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the function declaration appears on the *Line*

Parent is a debug instruction which represents parent lexical scope.

Linkage Name is an **OpString**, holding the linkage name of the function

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

13	12	<id> *Result Type*	*Result* <id>	<id> *Set*	19	<id> *Name*	<id> *Type*	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id> *Parent*	<id> *Linkage Name*	Flags

DebugFunction

Describe function or method definition.

Result Type must be **OpTypeVoid**

Name is an **OpString**, holding the name of the function as it appears in the source program

Type is an DebugTypeFunction instruction which represents type of the function.

Source is a **DebugSource** instruction representing text of the source program containing the function definition.

Line is a single *word* literal denoting the source line number at which the function declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the function declaration appears on the *Line*

Parent is a debug instruction which represents parent lexical scope.

Linkage Name is an **OpString**, holding the linkage name of the function

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

Scope Line a single *word* literal denoting line number in the source program at which the function scope begins.

Function is an **OpFunction** which is described by this instruction.

Declaration is DebugFunctionDeclaration which represents non-defining declaration of the function.

15+ 12	<id> *Result Type*	<id> *Result*	<id> *Set*	20	<id> *Name*	<id> *Type*	<id> *Source*	*Literal Number Line*	*Literal Number Column*	<id> *Parent*	<id> *Linkage Name*	Flags	*Literal Number Scope Line*	<id> *Function*	Optional <id> *Declaration*

4.7 Location Information

DebugLexicalBlock

Describe a lexical block in the source program.

Result Type must be **OpTypeVoid**

Source is a **DebugSource** instruction representing text of the source program containing the lexical block.

Line is a single *word* literal denoting the source line number at which the lexical block begins in the *Source*

Column is a single *word* literal denoting column number at which the lexical block begins.

Parent is a debug instructions describing the scope containing the current scope. Entities in the global scope should have *Parent* referring to DebugCompilationUnit.

Presence of the *Name* operand indicates that this instruction represents a C++ namespace. This operand refers to **OpString** holding the name of the namespace. For anonymous C++ namespaces the name must be an empty string.

9+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	21	<*id*> *Source*	*Literal Number Line*	*Literal Number Column*	<*id*> *Parent*	Optional <*id*> *Name*

DebugLexicalBlockDiscriminator

Distinguish lexical blocks on a single line in the source program.

Result Type must be **OpTypeVoid**

Source is a **DebugSource** instruction representing text of the source program containing the lexical block.

Parent is a debug instructions describing the scope containing the current scope.

Discriminator is a single *word* literal denoting DWARF discriminator value for instructions in the lexical block.

8	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	22	<*id*> *Source*	*Literal Number Discriminator*	<*id*> *Parent*

DebugScope

Provide information about source-level scope. This scope information applies to the instructions physically following this instruction, up to the first occurrence of any of the following: the next end of block, the next **DebugScope** instruction, or the next **DebugNoScope** instruction.

Result Type must be **OpTypeVoid**

Scope is a debugging instruction which describes source-level scope.

Inlined is an DebugInlinedAt instruction, which represents source-level scope and line number at which all instructions from the current scope were inlined.

6+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	23	<*id*> *Scope*	Optional <*id*> *Inlined At*

DebugNoScope

Discontinue previously declared by **DebugScope** source-level scope.

Result Type must be **OpTypeVoid**

5	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	24

DebugInlinedAt

Represent source-level scope and line number for the range of inlined instructions grouped together by an DebugScope instruction.

Result Type must be **OpTypeVoid**

Line is a single *word* literal denoting the line number in the source file where the range of instructions were inlined.

Scope is a debug instruction representing a source-level scope at which the range of instructions were inlined.

Inlined is a debug instruction representing the next level of inlining in case of recursive inlining.

7+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	25	*Literal Number Line*	<*id*> *Scope*	Optional <*id*> *Inlined*

4.8 Local Variables

DebugLocalVariable

Describe a local variable.

Result Type must be **OpTypeVoid**

Name is an **OpString**, holding the name of the variable as it appears in the source program

Type is a debugging instruction which represents type of the local variable.

Source is a **DebugSource** instruction representing text of the source program containing the local variable declaration.

Line is a single *word* literal denoting the source line number at which the local variable declaration appears in the *Source*

Column is a single *word* literal denoting column number at which the first character of the local variable declaration appears on the *Line*

Parent id of a debug instruction which represents parent lexical scope.

Flags is a single *word* literal formed by bitwise OR-ing values from the Debug Info Flags table.

If *ArgNumber* operand is present, this instruction represents a function formal parameter.

12+	12	<*id*> *Result Type*	*Result <id>*	<*id*> *Set*	26	<*id*> *Name*	<*id*> *Type*	<*id*> *Source*	*Literal Number Line*	*Literal Number Column*	<*id*> *Parent*	*Literal Flags*	Optional *Literal Number ArgNumber*

DebugInlinedVariable

Describe an inlined local variable.

Result Type must be **OpTypeVoid**

Variable is a debug instruction representing a local variable which is inlined.

Inlined is an DebugInlinedAt instruction representing the inline location.

7+	12	<*id*> *Result Type*	*Result <id>*	<*id*> *Set*	27	<*id*> *Variable*	<*id*> *Inlined*

DebugDeclare

Define point of declaration of a local variable.

Result Type must be **OpTypeVoid**

Local Variable must be an id of DebugLocalVariable

Variable must be an id of **OpVariable** instruction which defines the local variable.

Expression must be an id of a DebugExpression instruction.

8	12	*<id> Result Type*	*Result <id>*	*<id> Set*	28	*<id> Local Variable*	*<id> Variable*	*<id> Expression*

DebugValue

Represent changing of value of a local variable.

Result Type must be **OpTypeVoid**

Local Variable must be an id of DebugLocalVariable

Value is id of instruction, result of which is the new value of the *Local Variable*.

Expression is id of an DebugExpression instruction.

Indexes have the same semantics as corresponding operand(s) of **OpAccessChain**.

8+	12	*<id> Result Type*	*Result <id>*	*<id> Set*	29	*<id> Local Variable*	*<id> Value*	*<id> Expression*	*<id>, <id>, ... Indexes*

DebugOperation

Represent DWARF operation, that operate on a stack of values.

Result Type must be **OpTypeVoid**

Operation is a DWARF operation from the DWARF Operations table.

Operands are zero or more single *word* literals the *Operation* operates on.

6+	12	*<id> Result Type*	*Result <id>*	*<id> Set*	30	OpCode	Optional *Literal Operands ...*

DebugExpression

Represent DWARF expressions, which describe how to compute a value or name location during debugging of a program. They are expressed in terms of DWARF operations that operate on a stack of values.

Result Type must be **OpTypeVoid**

Operation is zero or more ids of DebugOperation.

5+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	31	Optional <*id*>... *Operation*

4.9 Macros

DebugMacroDef

Represents a macro definition

Result Type must be **OpTypeVoid**

Source is id of **OpString**, which contains the name of the file which contains definition of the macro.

Line is line number in the source file at which the macro is defined. If *Line* is zero the macro definition is provided by compiler's command line argument.

Name is id of **OpString**, which contains the name of the macro as it appears in the source program. In the case of a function-like macro definition, no whitespace characters appear between the name of the defined macro and the following left parenthesis. Formal parameters are separated by a comma without any whitespace. Right parenthesis terminates the formal parameter list

Value is id of **OpString**, which contians text with definition of the macro.

7+	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	32	<*id*> *Source*	*Literal Number Line*	<*id*> *Name*	Optional *Value*

DebugMacroUndef

Discontinue previous macro definition.

Result Type must be **OpTypeVoid**

Source is id of **OpString**, which contains the name of the file in which the macro is undefined

Line is line number in the source program at which the macro is rendered as undefined

Macro is id of DebugMacroDef which represent the macro to be undefined

8	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	33	<*id*> *Source*	*Literal Number Line*	<*id*> *Macro*

4.10 Imported Entities

DebugImportedEntity

Represents a C++ namespace *using-directive*, namespace alias or *using-declaration*.

Name is an **OpString**, holding the name or alias for the imported entity.

Tag is a literal value from the Imported Entities table which specifies the kind of the imported entity.

Source is a **DebugSource** instruction representing text of the source program the *Entity* is being imported from.

Entity is a debug instruction representing a namespace or declaration is being imported.

Line is a single *word* literal denoting the source line number at which the *using* declaration appears in the *Source*.

Column is a single *word* literal denoting column number at which the first character of the *using* declaration appears on the *Line*.

Parent is id of a debug instruction which represents the parent lexical scope.

12	12	<*id*> *Result Type*	*Result* <*id*>	<*id*> *Set*	34	<*id*> *Name*	*Literal Tag*	<*id*> *Source*	<*id*> *Entity*	*Literal Number Line*	*Literal Number Column*	<*id*> *Parent*

5 Validation Rules

None.

6 Issues

1. Does the ABI used for the OpenCL C 2.0 blocks feature have to be declared somewhere else in the module?
   RESOLVED: No. Block ABI is out of scope for this specification.

7 Revision History

Rev	Date	Author	Changes
0.99 Rev 1	2016-11-25	Alexey Sotkin	**Initial revision**
0.99 Rev 2	2016-12-08	Alexey Sotkin	Added details for the type instructions
0.99 Rev 3	2016-12-14	Alexey Sotkin	Added details for the rest of instructions
0.99 Rev 4	2016-12-21	Alexey Sotkin	Applied comments after review
0.99 Rev 5	2017-03-22	Alexey Sotkin	Format the specification as extended instruction set
0.99 Rev 6	2017-04-21	Alexey Sotkin	Adding File and Line operands

Rev	Date	Author	Changes
0.99 Rev 7	2017-06-05	Alexey Sotkin	Moving Flags to operands. Adding several new instructions.
0.99 Rev 8	2017-08-31	Alexey Sotkin	Replacing File operand by Source operand. Fixing typos. Formatting
0.99 Rev 9	2017-09-05	Alexey Sotkin	Clarifying representation of opaque types
0.99 Rev 10	2017-09-13	Alexey Sotkin	Support of multidimensional arrays. Adding DebugFunctionDeclaration. Updating debug operations.
0.99 Rev 11	2017-12-13	Alexey Sotkin	Removing "Op" prefix
0.99 Rev 12	2017-12-13	Alexey Sotkin	Changing style of enum tokens to CamelCase
1.00 Rev 1	2017-12-14	David Neto	Approved by SPIR WG on 2017-09-22. Change to 1.00 Rev 1
2.00 Rev 1	2018-12-05	Alexey Sotkin	Changing the name string in **OpExtInstImport** instruction. Adding DebugSource and DebugImportedEntity instructions. Adding *AtomicType* to the Type Qualifiers table. Adding *FlagIsEnumClass*, *FlagTypePassByValue*, *FlagTypePassByReference* to the Debug Info Flags table. Adding *Fragment* to the Debug Operations table. Adding *Linkage Name* operand to the DebugTypeComposite instruction. Adding *Flags* operand to the DebugTypeFunction and DebugLocalVariable instructions. Adding *Language* operand to the DebugCompilationUnit instruction.
2.00 Rev.2	2018-12-19	Alexey Sotkin	Added description of DebugOperations. Fixed minor typos and grammatical errors.