Debug Information Extended Instruction Set Specification

Alexey Sotkin, Intel

Version 1.00, Revision 1
# Table of Contents

1. Introduction ........................................................................................................... 3
2. Binary Form ............................................................................................................. 4
3. Enumerations .......................................................................................................... 5
   3.1. Instruction Enumeration ................................................................................. 5
   3.2. Debug Info Flags ............................................................................................. 6
   3.3. Base Type Attribute Encodings ..................................................................... 6
   3.4. Composite Types ............................................................................................ 7
   3.5. Type Qualifiers ............................................................................................... 7
   3.6. Debug Operations ........................................................................................... 7
4. Instructions .............................................................................................................. 9
   4.1. Absent Debugging Information ....................................................................... 9
   4.2. Compilation Unit ........................................................................................... 9
   4.3. Type instructions .......................................................................................... 9
   4.4. Templates ....................................................................................................... 15
   4.5. Global Variables ........................................................................................... 17
   4.6. Functions ...................................................................................................... 18
   4.7. Location Information .................................................................................... 20
   4.8. Local Variables ............................................................................................. 22
   4.9. Macros .......................................................................................................... 24
5. Validation Rules ...................................................................................................... 26
6. Issues ...................................................................................................................... 27
7. Revision History ...................................................................................................... 28
Contributors and Acknowledgments

• Yaxun Liu, AMD
• Brian Sumner, AMD
• Ben Ashbaugh, Intel
• Alexey Bader, Intel
• Raun Krisch, Intel
• John Kessenich, Google
• David Neto, Google
• Neil Henning, Codeplay
• Kerch Holt, Nvidia
Chapter 1. Introduction

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

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

```
<extinst-id> OpExtInstImport "DebugInfo"
```

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

The design guidelines for these instructions are:

- Sufficient for a backend to generate DWARF4 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
Chapter 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.
Chapter 3. Enumerations

3.1. Instruction Enumeration

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>Instruction name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DebugInfoNone</td>
</tr>
<tr>
<td>1</td>
<td>DebugCompilationUnit</td>
</tr>
<tr>
<td>2</td>
<td>DebugTypeBasic</td>
</tr>
<tr>
<td>3</td>
<td>DebugTypePointer</td>
</tr>
<tr>
<td>4</td>
<td>DebugTypeQualifier</td>
</tr>
<tr>
<td>5</td>
<td>DebugTypeArray</td>
</tr>
<tr>
<td>6</td>
<td>DebugTypeVector</td>
</tr>
<tr>
<td>7</td>
<td>DebugTypedef</td>
</tr>
<tr>
<td>8</td>
<td>DebugTypeFunction</td>
</tr>
<tr>
<td>9</td>
<td>DebugTypeEnum</td>
</tr>
<tr>
<td>10</td>
<td>DebugTypeComposite</td>
</tr>
<tr>
<td>11</td>
<td>DebugTypeMember</td>
</tr>
<tr>
<td>12</td>
<td>DebugTypeInheritance</td>
</tr>
<tr>
<td>13</td>
<td>DebugTypePtrToMember</td>
</tr>
<tr>
<td>14</td>
<td>DebugTypeTemplate</td>
</tr>
<tr>
<td>15</td>
<td>DebugTypeTemplateParameter</td>
</tr>
<tr>
<td>16</td>
<td>DebugTypeTemplateTemplateParameter</td>
</tr>
<tr>
<td>17</td>
<td>DebugTypeTemplateParameterPack</td>
</tr>
<tr>
<td>18</td>
<td>DebugGlobalVariable</td>
</tr>
<tr>
<td>19</td>
<td>DebugFunctionDeclaration</td>
</tr>
<tr>
<td>20</td>
<td>DebugFunction</td>
</tr>
<tr>
<td>21</td>
<td>DebugLexicalBlock</td>
</tr>
<tr>
<td>22</td>
<td>DebugLexicalBlockDiscriminator</td>
</tr>
<tr>
<td>23</td>
<td>DebugScope</td>
</tr>
<tr>
<td>24</td>
<td>DebugNoScope</td>
</tr>
<tr>
<td>25</td>
<td>DebugInlinedAt</td>
</tr>
</tbody>
</table>
### 3.2. Debug Info Flags

<table>
<thead>
<tr>
<th>Value</th>
<th>Flag Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;&lt; 0</td>
<td>FlagIsProtected</td>
</tr>
<tr>
<td>1 &lt;&lt; 1</td>
<td>FlagIsPrivate</td>
</tr>
<tr>
<td>1 &lt;&lt; 2</td>
<td>FlagIsLocal</td>
</tr>
<tr>
<td>1 &lt;&lt; 3</td>
<td>FlagIsDefinition</td>
</tr>
<tr>
<td>1 &lt;&lt; 4</td>
<td>FlagFwdDecl</td>
</tr>
<tr>
<td>1 &lt;&lt; 5</td>
<td>FlagArtificial</td>
</tr>
<tr>
<td>1 &lt;&lt; 6</td>
<td>FlagExplicit</td>
</tr>
<tr>
<td>1 &lt;&lt; 7</td>
<td>FlagPrototyped</td>
</tr>
<tr>
<td>1 &lt;&lt; 8</td>
<td>FlagObjectPointer</td>
</tr>
<tr>
<td>1 &lt;&lt; 9</td>
<td>FlagStaticMember</td>
</tr>
<tr>
<td>1 &lt;&lt; 10</td>
<td>FlagIndirectVariable</td>
</tr>
<tr>
<td>1 &lt;&lt; 11</td>
<td>FlagLValueReference</td>
</tr>
<tr>
<td>1 &lt;&lt; 12</td>
<td>FlagRValueReference</td>
</tr>
<tr>
<td>1 &lt;&lt; 13</td>
<td>FlagIsOptimized</td>
</tr>
</tbody>
</table>

### 3.3. Base Type Attribute Encodings

Used by `DebugTypeBasic`
### 3.4. Composite Types

**Used by** `DebugTypeComposite`

<table>
<thead>
<tr>
<th>Tag code name</th>
<th>0</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Structure</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Union</td>
<td></td>
</tr>
</tbody>
</table>

### 3.5. Type Qualifiers

**Used by** `DebugTypeQualifier`

<table>
<thead>
<tr>
<th>Qualifier tag code name</th>
<th>0</th>
<th>ConstType</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VolatileType</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RestrictType</td>
<td></td>
</tr>
</tbody>
</table>

### 3.6. Debug Operations

**Used by** `DebugExpression`

<table>
<thead>
<tr>
<th>Operation encodings</th>
<th>No. of Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Deref</td>
</tr>
<tr>
<td>1</td>
<td>Plus</td>
</tr>
<tr>
<td>2</td>
<td>Minus</td>
</tr>
<tr>
<td>3</td>
<td>PlusUconst</td>
</tr>
<tr>
<td>Operation encodings</td>
<td>No. of Operands</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>4 BitPiece</td>
<td>2</td>
</tr>
<tr>
<td>5 Swap</td>
<td>0</td>
</tr>
<tr>
<td>6 Xderef</td>
<td>0</td>
</tr>
<tr>
<td>7 StackValue</td>
<td>0</td>
</tr>
<tr>
<td>8 Constu</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 4. Instructions

4.1. Absent 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 compilation unit.

*Result Type* must be **OpTypeVoid**

*Source* is an **OpSource** providing text of the primary source program this module was derived from.

*Version* is version of SPIRV debug information specification.

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

| 8 | 12 | <id> Result Type | Result <id> | <id> Set | 1 | <id> Source | Literal Number Version | Literal Number DWARF version |

4.3. Type instructions

DebugTypeBasic

Describe basic data types.

*Result Type* must be **OpTypeVoid**

*Name* represents 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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th><code>&lt;id&gt;</code> Result Type</th>
<th>Result <code>&lt;id&gt;</code></th>
<th><code>&lt;id&gt;</code> Set</th>
<th>3</th>
<th><code>&lt;id&gt;</code> Base Type</th>
<th>Storage Class</th>
<th>Literal Flags</th>
</tr>
</thead>
</table>

**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 Qualifier* is a literal value from the **TypeQualifiers** table.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th><code>&lt;id&gt;</code> Result Type</th>
<th>Result <code>&lt;id&gt;</code></th>
<th><code>&lt;id&gt;</code> Set</th>
<th>4</th>
<th><code>&lt;id&gt;</code> Base Type</th>
<th>Type Qualifier</th>
</tr>
</thead>
</table>

**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.

<table>
<thead>
<tr>
<th>7</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>&lt;id&gt; Set</th>
<th>6</th>
<th>&lt;id&gt; Base Type</th>
<th>Literal Number Component Count</th>
</tr>
</thead>
</table>

### 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 an **OpSource** providing text of the primary source program this module was derived from.

*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.

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>&lt;id&gt; Set</th>
<th>7</th>
<th>&lt;id&gt; Name</th>
<th>&lt;id&gt; Base Type</th>
<th>&lt;id&gt; Source</th>
<th>Literal Number Line</th>
<th>Literal Number Column</th>
<th>&lt;id&gt; Parent</th>
</tr>
</thead>
</table>

### DebugTypeFunction

Describe a function type

*Result Type* must be **OpTypeVoid**

*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

<table>
<thead>
<tr>
<th>6+</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>&lt;id&gt; Set</th>
<th>8</th>
<th>&lt;id&gt; Return Type</th>
<th>&lt;id&gt;, &lt;id&gt;, … Parameter Types</th>
</tr>
</thead>
</table>
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 an OpSource providing text of the primary source program this module was derived from.

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.
DebugTypeComposite

Describe structure, class and union data types

Result Type must be OpTypeVoid

Tag specifies the kind of composite type

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

Source is an OpSource providing text of the primary source program this module was derived from.

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

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.
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 an **OpSource** providing text of the primary source program this module was derived from.

*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++.

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>2</th>
<th>&lt;id&gt; Result Type &lt;id&gt;</th>
<th>Result Set &lt;id&gt;</th>
<th>&lt;id&gt; Name</th>
<th>&lt;id&gt; Type</th>
<th>&lt;id&gt; Source Literal Number Line</th>
<th>Literal Number Column</th>
<th>&lt;id&gt; Parent Offset</th>
<th>&lt;id&gt; Size</th>
<th>Flags</th>
<th>Optional &lt;id&gt; Value</th>
</tr>
</thead>
</table>

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.
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.

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.
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 an **OpSource** providing text of the primary source program this module was derived from.

*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*.

---

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 an **OpSource** providing text of the primary source program this module was derived from.

*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*.
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 an *OpSource* providing text of the primary source program this module was derived from.

*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

<table>
<thead>
<tr>
<th>10</th>
<th>12</th>
<th>&lt;id&gt;</th>
<th>Result Type</th>
<th>&lt;id&gt;</th>
<th>Result</th>
<th>&lt;id&gt;</th>
<th>Set</th>
<th>17</th>
<th>&lt;id&gt;</th>
<th>Name</th>
<th>&lt;id&gt;</th>
<th>Source</th>
<th>Literal Number Line</th>
<th>Literal Number Column</th>
<th>&lt;id&gt;…</th>
<th>Template parameters</th>
</tr>
</thead>
</table>

4.5. Global Variables
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 an **OpSource** providing text of the primary source program this module was derived from.

*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**.

| 1 | 1 2 | 1 8 | 1 4 | 1 8 | <id> Res <id> Set Name <id> Type <id> Source Literal Number Line Literal Number Column <id> Parent <id> Linkage Name <id> Variable <id> Flags Option al <id> Static Memb er 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 an **OpSource** providing text of the primary source program this module was derived from.

*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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Resiult Type</th>
<th>&lt;id&gt; Set</th>
<th>1</th>
<th>9</th>
<th>Name</th>
<th>&lt;id&gt; Type</th>
<th>Literal Number Line</th>
<th>Literal Number Column</th>
<th>&lt;id&gt; Parent</th>
<th>&lt;id&gt; Linkage Name</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>&lt;id&gt;</td>
<td>Result</td>
<td>&lt;id&gt;</td>
<td>Name</td>
<td>&lt;id&gt; Type</td>
<td>&lt;id&gt; Source</td>
<td>1</td>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DebugFunction
Describe function or method definition or 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 an OpSource providing text of the primary source program this module was derived from.

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.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>2</th>
<th>&lt;id&gt; Result Type</th>
<th>&lt;id&gt; Res</th>
<th>&lt;id&gt; Set</th>
<th>&lt;id&gt; Name</th>
<th>&lt;id&gt; Type</th>
<th>&lt;id&gt; Source Line</th>
<th>&lt;id&gt; Literal Number Column</th>
<th>&lt;id&gt; Parent Linkage Name</th>
<th>Flags</th>
<th>&lt;id&gt; Function Scope Line</th>
<th>Optional &lt;id&gt; Declaration</th>
</tr>
</thead>
</table>

4.7. Location Information
**DebugLexicalBlock**

Describe a lexical block in the source program.

*Result Type* must be **OpTypeVoid**

*Source* is an **OpSource** providing text of the primary source program this module was derived from.

*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.

<table>
<thead>
<tr>
<th>9+</th>
<th>12</th>
<th>Result Type</th>
<th>Result Set</th>
<th>Source</th>
<th>Literal Line</th>
<th>Literal Column</th>
<th>Parent</th>
<th>Optional Name</th>
</tr>
</thead>
</table>

**DebugLexicalBlockDiscriminator**

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

*Result Type* must be **OpTypeVoid**

*Source* is an **OpSource** providing text of the primary source program this module was derived from.

*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.

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
<th>Result Type</th>
<th>Result Set</th>
<th>Source</th>
<th>Literal Discriminator</th>
<th>Parent</th>
</tr>
</thead>
</table>

**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.
**DebugNoScope**

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

*Result Type* must be `OpTypeVoid`.

**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.

**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 an **OpSource** providing text of the primary source program this module was derived from.

*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.

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

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>7+</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>27</td>
</tr>
</tbody>
</table>

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.
**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`.

**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.

**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`.

**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 contains text with definition of the macro.

<table>
<thead>
<tr>
<th>7+</th>
<th>12</th>
<th>&lt;id&gt; Result Type</th>
<th>Result &lt;id&gt;</th>
<th>&lt;id&gt; Set</th>
<th>32</th>
<th>&lt;id&gt; Source</th>
<th>Literal Number Line</th>
<th>&lt;id&gt; Name</th>
<th>Optional Value</th>
</tr>
</thead>
</table>

**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 |
|-----|----|------------------|-------------|---------|----|-------------|---------------------|---------|----------------|

25
Chapter 5. Validation Rules

None.
Chapter 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.
<table>
<thead>
<tr>
<th>Rev</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.99</td>
<td>2016-11-25</td>
<td>Alexey Sotkin</td>
<td>Initial revision</td>
</tr>
<tr>
<td>0.99</td>
<td>2016-12-08</td>
<td>Alexey Sotkin</td>
<td>Added details for the type instructions</td>
</tr>
<tr>
<td>0.99</td>
<td>2016-12-14</td>
<td>Alexey Sotkin</td>
<td>Added details for the rest of instructions</td>
</tr>
<tr>
<td>0.99</td>
<td>2016-12-21</td>
<td>Alexey Sotkin</td>
<td>Applied comments after review</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-03-22</td>
<td>Alexey Sotkin</td>
<td>Format the specification as extended instruction set</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-04-21</td>
<td>Alexey Sotkin</td>
<td>Adding File and Line operands</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-06-05</td>
<td>Alexey Sotkin</td>
<td>Moving Flags to operands. Adding several new instructions.</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-08-31</td>
<td>Alexey Sotkin</td>
<td>Replacing File operand by Source operand. Fixing typos. Formatting</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-09-05</td>
<td>Alexey Sotkin</td>
<td>Clarifying representation of opaque types</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-12-13</td>
<td>Alexey Sotkin</td>
<td>Removing &quot;Op&quot; prefix</td>
</tr>
<tr>
<td>0.99</td>
<td>2017-12-13</td>
<td>Alexey Sotkin</td>
<td>Changing style of enum tokens to CamelCase</td>
</tr>
<tr>
<td>1.00</td>
<td>2017-12-14</td>
<td>David Neto</td>
<td>Approved by SPIR WG on 2017-09-22. Change to 1.00 Rev 1</td>
</tr>
</tbody>
</table>