SPIR-V Specification Provisional

John Kessenich, Google; Boaz Ouriel, Intel; Raun Krisch, Intel
version 1.1, Revision 3

Table of Contents

Contributors and Acknowledgments

Connor Abbott, Intel

Alexey Bader, Intel

Dan Baker, Oxide Games

Kenneth Benzie, Codeplay

Gordon Brown, Codeplay

Pat Brown, NVIDIA

Diana Po-Yu Chen, MediaTek

Stephen Clarke, Imagination

Patrick Doane, Blizzard Entertainment

Stefanus Du Toit, Google

Tim Foley, Intel

Ben Gaster, Qualcomm

Alexander Galazin, ARM

Christopher Gautier, ARM

Neil Henning, Codeplay

Kerch Holt, NVIDIA

Lee Howes, Qualcomm

Roy Ju, MediaTek

John Kessenich, Google

Daniel Koch, NVIDIA

Ashwin Kolhe, NVIDIA

Raun Krisch, Intel

Graeme Leese, Broadcom

Yuan Lin, NVIDIA

Yaxun Liu, AMD

Timothy Lottes, Epic Games

John McDonald, Valve
David Neto, Google
Boaz Ouriel, Intel
Christophe Riccio, Unity
Andrew Richards, Codeplay
Ian Romanick, Intel
Graham Sellers, AMD
Robert Simpson, Qualcomm
Brian Sumner, AMD
Andrew Woloszyn, Google
Weifeng Zhang, Qualcomm

Note Up-to-date HTML and PDF versions of this specification may be found at the Khronos SPIR-V Registry. (https://www.khronos.org/registry/spir-v/)

1. Introduction

Abstract

SPIR-V is a simple binary intermediate language for graphical shaders and compute kernels. A SPIR-V module contains multiple entry points with potentially shared functions in the entry point's call trees. Each function contains a control-flow graph (CFG) of basic blocks, with optional instructions to express structured control flow. Load/store instructions are used to access declared variables, which includes all input/output (IO). Intermediate results bypassing load/store use static single-assignment (SSA) representation. Data objects are represented logically, with hierarchical type information: There is no flattening of aggregates or assignment to physical register banks, etc. Selectable addressing models establish whether general pointer operations may be used, or if memory access is purely logical.

This document fully defines **SPIR-V**, a Khronos-standard binary intermediate language for representing graphical-shader stages and compute kernels for multiple Khronos APIs.

1.1. Goals

SPIR-V has the following goals:

- Provide a simple binary intermediate language for all functionality appearing in Khronos shaders/kernels.
- Have a concise, transparent, self-contained specification (sections Specification and Binary Form).
- Map easily to other intermediate languages.
- Be the form passed by an API into a driver to set shaders/kernels.
- Can be targeted by new front ends for novel high-level languages.
- Allow the first steps of compilation and reflection to be done offline.
- Be low-level enough to require a reverse-engineering step to reconstruct source code.
- Improve portability by enabling shared tools to generate or operate on it.
- Allow separation of core specification from source-language-specific sets of built-in functions.
- Reduce compile time during application run time. (Eliminating most of the compile time during application run time is not a goal of this intermediate language. Target-specific register allocation and scheduling are still expected to take significant time.)
- Allow some optimizations to be done offline.

1.2. About this document

This document aims to:

- Include everything needed to fully understand, create, and consume SPIR-V. However:
  - Imported sets of instructions (which implement source-specific built-in functions) will need their own specification.
  - Many validation rules are client-API specific, and hence documented with client API and not in this specification.
- Separate expository and specification language. The specification-proper is in Specification and Binary Form.

1.3. Extendability

SPIR-V can be extended by multiple vendors or parties simultaneously:

- Using the **OpExtension** instruction to require new semantics that must be supported. Such new semantics would come from an extension document.
- Reserving (registering) ranges of the token values, as described further below.
- Aided by instruction skipping, also further described below.

Enumeration Token Values. It is easy to extend all the types, storage classes, opcodes, decorations, etc. by adding to the token values.

Registration. Ranges of token values in the Binary Form section can be pre-allocated to numerous vendors/parties. This allows combining multiple independent extensions without conflict. To register ranges, see https://www.khronos.org/registry/spir-v/api/spir-v.xml.

Extended Instructions. Sets of extended instructions can be provided and specified in separate specifications. These help personalize SPIR-V for different source languages or execution environments (client APIs). Multiple sets of extended instructions can be imported without conflict, as the extended instructions are selected by {set id, instruction number} pairs.

Instruction Skipping. Tools are encouraged to skip opcodes for features they are not required to process. This is trivially enabled by the word count in an instruction, which makes it easier to add new instructions without breaking existing tools.

1.4. Debuggability

SPIR-V can decorate, with a text string, virtually anything created in the shader: types, variables, functions, etc. This is required for externally visible symbols, and also allowed for naming the result of any instruction. This can be used to aid in understandability when disassembling or debugging lowered versions of SPIR-V.

Location information (file names, lines, and columns) can be interleaved with the instruction stream to track the origin of each instruction.

1.5. Design Principles

Regularity. All instructions start with a word count. This allows walking a SPIR-V module without decoding each opcode. All instructions have an opcode that dictates for all operands what kind of operand they are. For instructions with a variable number of operands, the number of variable operands is known by subtracting the number of non-variable words from the instruction's word count.

Non Combinatorial. There is no combinatorial type explosion or need for large encode/decode tables for types. Rather, types are parameterized. Image types declare their dimensionality, arrayness, etc. all orthogonally, which greatly simplify code. This is done similarly for other types. It also applies to opcodes. Operations are orthogonal to scalar/vector size, but not to integer vs. floating-point differences.

Modeless. After a given execution model (e.g., pipeline stage) is specified, internal operation is essentially modeless: Generally, it will follow the rule: "same spelling, same semantics", and does not have mode bits that modify semantics. If a change to SPIR-V modifies semantics, it should use a different spelling. This makes consumers of SPIR-V much more robust. There are execution modes declared, but these are generally to affect the way the module interacts with the environment around it, not the internal semantics. Capabilities are also declared, but this is to declare the subset of functionality that is used, not to change any semantics of what is used.

Declarative. SPIR-V declares externally-visible modes like "writes depth", rather than having rules that require deduction from full shader inspection. It also explicitly declares what addressing modes, execution model, extended instruction sets, etc. will be used. See Language Capabilities for more information.

SSA. All results of intermediate operations are strictly SSA. However, declared variables reside in memory and use load/store for access, and such variables can be stored to multiple times.

IO. Some storage classes are for input/output (IO) and, fundamentally, IO will be done through load/store of variables declared in these storage classes.

1.6. Static Single Assignment (SSA)

SPIR-V includes a phi instruction to allow the merging together of intermediate results from split control flow. This allows split control flow without load/store to memory. SPIR-V is flexible in the degree to which load/store is used; it is possible to use control flow with no phi-instructions, while still staying in SSA form, by using memory load/store.

Some storage classes are for IO and, fundamentally, IO will be done through load/store, and initial load and final store can never be eliminated. Other storage classes are shader local and can have their load/store eliminated. It can be considered an optimization to largely eliminate such loads/stores by moving them into intermediate results in SSA form.

1.7. Built-In Variables

SPIR-V identifies built-in variables from a high-level language with an enumerant decoration. This assigns any unusual semantics to the variable. Built-in variables must otherwise be declared with their correct SPIR-V type and treated the same as any other variable.

1.8. Specialization

Specialization enables creating a portable SPIR-V module outside the target execution environment, based on constant values that won't be known until inside the execution environment. For example, to size a fixed array with a constant not known during creation of a module, but known when the module will be lowered to the target architecture.

See Specialization in the next section for more details.

1.9. Example

The SPIR-V form is binary, not human readable, and fully described in Binary Form. This is an example disassembly to give a basic idea of what SPIR-V looks like:

GLSL fragment shader:

```
#version 450

in vec4 color1;
in vec4 multiplier;
noperspective in vec4 color2;
out vec4 color;

struct S {
    bool b;
    vec4 v[5];
    int i;
};

uniform blockName {
    S s;
    bool cond;
};

void main()
{
    vec4 scale = vec4(1.0, 1.0, 2.0, 1.0);

    if (cond)
        color = color1 + s.v[2];
    else
        color = sqrt(color2) * scale;

    for (int i = 0; i < 4; ++i)
        color *= multiplier;
}
```

Corresponding SPIR-V:

```
; Magic:     0x07230203 (SPIR-V)
; Version:   0x00010000 (Version: 1.0.0)
; Generator: 0x00080001 (Khronos Glslang Reference Front End; 1)
; Bound:     63
; Schema:    0

               OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %4 "main" %31 %33 %42 %57
               OpExecutionMode %4 OriginLowerLeft

; Debug information
               OpSource GLSL 450
               OpName %4 "main"
               OpName %9 "scale"
               OpName %17 "S"
               OpMemberName %17 0 "b"
               OpMemberName %17 1 "v"
               OpMemberName %17 2 "i"
               OpName %18 "blockName"
               OpMemberName %18 0 "s"
               OpMemberName %18 1 "cond"
               OpName %20 ""
               OpName %31 "color"
```

```
               OpName %33 "color1"
               OpName %42 "color2"
               OpName %48 "i"
               OpName %57 "multiplier"

; Annotations (non-debug)
               OpDecorate %15 ArrayStride 16
               OpMemberDecorate %17 0 Offset 0
               OpMemberDecorate %17 1 Offset 16
               OpMemberDecorate %17 2 Offset 96
               OpMemberDecorate %18 0 Offset 0
               OpMemberDecorate %18 1 Offset 112
               OpDecorate %18 Block
               OpDecorate %20 DescriptorSet 0
               OpDecorate %42 NoPerspective

; All types, variables, and constants
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2                      ; void ()
          %6 = OpTypeFloat 32                         ; 32-bit float
          %7 = OpTypeVector %6 4                      ; vec4
          %8 = OpTypePointer Function %7              ; function-local vec4*
         %10 = OpConstant %6 1
         %11 = OpConstant %6 2
         %12 = OpConstantComposite %7 %10 %10 %11 %10 ; vec4(1.0, 1.0, 2.0, 1.0)
         %13 = OpTypeInt 32 0                         ; 32-bit int, sign-less
         %14 = OpConstant %13 5
         %15 = OpTypeArray %7 %14
         %16 = OpTypeInt 32 1
         %17 = OpTypeStruct %13 %15 %16
         %18 = OpTypeStruct %17 %13
         %19 = OpTypePointer Uniform %18
         %20 = OpVariable %19 Uniform
         %21 = OpConstant %16 1
         %22 = OpTypePointer Uniform %13
         %25 = OpTypeBool
         %26 = OpConstant %13 0
         %30 = OpTypePointer Output %7
         %31 = OpVariable %30 Output
         %32 = OpTypePointer Input %7
         %33 = OpVariable %32 Input
         %35 = OpConstant %16 0
         %36 = OpConstant %16 2
         %37 = OpTypePointer Uniform %7
         %42 = OpVariable %32 Input
         %47 = OpTypePointer Function %16
         %55 = OpConstant %16 4
         %57 = OpVariable %32 Input

; All functions
          %4 = OpFunction %2 None %3                  ; main()
          %5 = OpLabel
          %9 = OpVariable %8 Function
         %48 = OpVariable %47 Function
               OpStore %9 %12
         %23 = OpAccessChain %22 %20 %21              ; location of cond
         %24 = OpLoad %13 %23                         ; load 32-bit int from cond
         %27 = OpINotEqual %25 %24 %26                ; convert to bool
               OpSelectionMerge %29 None              ; structured if
               OpBranchConditional %27 %28 %41        ; if cond
         %28 = OpLabel                                ; then
         %34 = OpLoad %7 %33
         %38 = OpAccessChain %37 %20 %35 %21 %36      ; s.v[2]
         %39 = OpLoad %7 %38
         %40 = OpFAdd %7 %34 %39
               OpStore %31 %40
               OpBranch %29
```

```
%41 = OpLabel                                   ; else
%43 = OpLoad %7 %42
%44 = OpExtInst %7 %1 Sqrt %43                  ; extended instruction sqrt
%45 = OpLoad %7 %9
%46 = OpFMul %7 %44 %45
      OpStore %31 %46
      OpBranch %29
%29 = OpLabel                                   ; endif
      OpStore %48 %35
      OpBranch %49
%49 = OpLabel
      OpLoopMerge %51 %52 None                  ; structured loop
      OpBranch %53
%53 = OpLabel
%54 = OpLoad %16 %48
%56 = OpSLessThan %25 %54 %55                   ; i < 4 ?
      OpBranchConditional %56 %50 %51           ; body or break
%50 = OpLabel                                   ; body
%58 = OpLoad %7 %57
%59 = OpLoad %7 %31
%60 = OpFMul %7 %59 %58
      OpStore %31 %60
      OpBranch %52
%52 = OpLabel                                   ; continue target
%61 = OpLoad %16 %48
%62 = OpIAdd %16 %61 %21                        ; ++i
      OpStore %48 %62
      OpBranch %49                              ; loop back
%51 = OpLabel                                   ; loop merge point
      OpReturn
      OpFunctionEnd
```

2. Specification

2.1. Language Capabilities

A SPIR-V module is consumed by an execution environment, specified by a client API, that needs to support the features used by that SPIR-V module. Features are classified through capabilities. Capabilities used by a particular SPIR-V module must be declared early in that module with the **OpCapability** instruction. Then:

- A validator can validate that the module uses only its declared capabilities.
- An execution environment is allowed to reject modules declaring capabilities it does not support. (See client API specifications for environment-specific rules.)

All available capabilities and their dependencies form a capability hierarchy, fully listed in the capability section. Only top-level capabilities need to be declared; their dependencies are automatically included.

This (SPIR-V) specification provides capability-specific validation rules, in the validation section. To ensure portability, each client API needs to include the following:

- Which capabilities in the capability section it requires environments to support, and hence allows in SPIR-V modules.
- Required limits, if they are beyond the Universal Limits.
- Any validation requirements specific to the environment that are not tied to specific capabilities, and hence not covered in the SPIR-V specification.

2.2. Terms

2.2.1. Instructions

Word: 32 bits.

<id>: A numerical name; the name used to refer to an object, a type, a function, a label, etc. An *<id>* always consumes one *word*. The *<id>s* defined by a module obey SSA.

Result <id>: Most instructions define a result, named by an *<id>* explicitly provided in the instruction. The *Result <id>* is used as an operand in other instructions to refer to the instruction that defined it.

Literal String: A nul-terminated stream of characters consuming an integral number of *words*. The character set is Unicode in the UTF-8 encoding scheme. The UTF-8 octets (8-bit bytes) are packed four per *word*, following the little-endian convention (i.e., the first octet is in the lowest-order 8 bits of the word). The final word contains the string's nul-termination character (0), and all contents past the end of the string in the final word are padded with 0.

Literal Number: A numeric value consuming one or more *words*. An instruction will determine what type a literal will be interpreted as. When the type's bit width is larger than one word, the literal's low-order words appear first. When the type's bit width is less than 32-bits, the literal's value appears in the low-order bits of the word, and the high-order bits must be 0 for a floating-point type, or 0 for an integer type with *Signedness* of 0, or sign extended when *Signedness* is 1. (Similarly for the remaining bits of widths larger than 32 bits but not a multiple of 32 bits.)

Literal: A *Literal String* or a *Literal Number*.

Operand: A one-*word* argument to an instruction. E.g., it could be an *<id>*, or a (part of a) *literal*. Which form it holds is always explicitly known from the opcode.

Immediate: Operand(s) directly holding a literal value rather than an *<id>*. Immediate values larger than one *word* will consume multiple operands, one per word. That is, operand counting is always done per word, not per immediate.

WordCount: The complete number of *words* taken by an instruction, including the word holding the word count and opcode, and any optional operands. An instruction's word count is the total space taken by the instruction.

Instruction: After a header, a module is simply a linear list of instructions. An instruction contains a *word count*, an opcode, an optional *Result <id>*, an optional *<id>* of the instruction's type, and a variable list of operands. All instruction opcodes and semantics are listed in **Instructions**.

Decoration: Auxiliary information such as built-in variable, stream numbers, invariance, interpolation type, relaxed precision, etc., added to *<id>s* or structure-type members through Decorations. Decorations are enumerated in Decoration in the Binary Form section.

Object: An instantiation of a non-void type, either as the *Result <id>* of an operation, or created through **OpVariable**.

Memory Object: An object created through **OpVariable**. Such an object can die on function exit, if it was a function variable, or exist for the duration of an entry point.

Intermediate Object or *Intermediate Value* or *Intermediate Result:* An object created by an operation (not memory allocated by **OpVariable**) and dying on its last consumption.

Constant Instruction: Either a specialization-constant instruction or a fixed constant instruction: Instructions that start "OpConstant" or "OpSpec".

[a, b]: This square-bracket notation means the range from *a* to *b*, inclusive of *a* and *b*. Parenthesis exclude their end point, so, for example, *(a, b]* means *a* to *b* excluding *a* but including *b*.

2.2.2. Types

Boolean type: The type returned by **OpTypeBool**.

Integer type: Any width signed or unsigned type from **OpTypeInt**. By convention, the lowest-order bit will be referred to as bit-number 0, and the highest-order bit as bit-number *Width* - 1.

Floating-point type: Any width type from **OpTypeFloat**.

Numerical type: An integer type or a floating-point type.

Scalar: A single instance of a *numerical type* or *Boolean type*. Scalars will also be called *components* when being discussed either by themselves or in the context of the contents of a *vector*.

Vector: An ordered homogeneous collection of two or more *scalars*. Vector sizes are quite restrictive and dependent on the execution model.

Matrix: An ordered homogeneous collection of vectors. When vectors are part of a matrix, they will also be called *columns*. Matrix sizes are quite restrictive and dependent on the execution model.

Array: An ordered homogeneous collection of any non-void-type objects. When an object is part of an array, it will also be called an *element*. Array sizes are generally not restricted.

Structure: An ordered heterogeneous collection of any non-void types. When an object is part of a structure, it will also be called a *member*.

Aggregate: A *structure* or an *array*.

Composite: An *aggregate*, a *matrix*, or a *vector*.

Image: A traditional texture or image; SPIR-V has this single name for these. An image type is declared with **OpTypeImage**. An image does not include any information about how to access, filter, or sample it.

Sampler: Settings that describe how to access, filter, or sample an image. Can come either from literal declarations of settings or be an opaque reference to externally bound settings. A sampler does not include an *image*.

Sampled Image: An *image* combined with a *sampler*, enabling filtered accesses of the image's contents.

Concrete Type: A numerical scalar, vector, or matrix type, or **OpTypePointer** when using a **Physical** addressing model, or any aggregate containing only these types.

Abstract Type: An **OpTypeVoid** or **OpTypeBool**, or **OpTypePointer** when using the **Logical** addressing model, or any aggregate type containing any of these.

Opaque Type: A type that is, or contains, or points to, or contains pointers to, any of the following types:

- **OpTypeImage**
- **OpTypeSampler**
- **OpTypeSampledImage**
- **OpTypeOpaque**
- **OpTypeEvent**
- **OpTypeDeviceEvent**
- **OpTypeReserveId**
- **OpTypeQueue**
- **OpTypePipe**
- **OpTypeForwardPointer**
- **OpTypePipeStorage**

- **OpTypeNamedBarrier**

2.2.3. Module

Module: A single unit of SPIR-V. It can contain multiple *entry points*, but only one set of capabilities.

Entry Point: A function in a *module* where execution begins. A single *entry point* is limited to a single *execution model*. An entry point is declared using **OpEntryPoint**.

Execution Model: A graphical-pipeline stage or OpenCL kernel. These are enumerated in Execution Model.

Execution Mode: Modes of operation relating to the interface or execution environment of the module. These are enumerated in Execution Mode. Generally, modes do not change the semantics of instructions within a SPIR-V module.

Vertex Processor: Any stage or execution model that processes vertices: Vertex, tessellation control, tessellation evaluation, and geometry. Explicitly excludes fragment and compute execution models.

2.2.4. Control Flow

Block: A contiguous sequence of instructions starting with an **OpLabel**, ending with a branch instruction, and having no other label or branch instructions.

Branch Instruction: One of the following, used to terminate blocks:

- **OpBranch**
- **OpBranchConditional**
- **OpSwitch**
- **OpKill**
- **OpReturn**
- **OpReturnValue**
- **OpUnreachable**

Dominate: A block *A* dominates a block *B*, where *A* and *B* are in the same function, if every path from the function's entry point to block *B* goes through block *A*.

Post Dominate: A block *B* post dominates a block *A*, where *A* and *B* are in the same function, if every path from *A* to a function-return instruction goes through block *B*.

Control-Flow Graph: The graph formed by a function's blocks and branches. The blocks are the graph's nodes, and the branches the graph's edges.

CFG: Control-flow graph.

Back Edge: If a depth-first traversal is done on a function's CFG, starting from the first block of the function, a *back edge* is a branch to a previously visited block. A *back-edge block* is the block containing such a branch.

Merge Instruction: One of the following, used before a branch instruction to declare structured control flow:

- **OpSelectionMerge**
- **OpLoopMerge**

Header Block: A block containing a merge instruction.

Loop Header: A header block whose merge instruction is an **OpLoopMerge**.

Merge Block: A block declared by the *Merge Block* operand of a merge instruction.

Break Block: A block containing a branch to the *Merge Block* of a loop header's merge instruction.

Continue Block: A block containing a branch to an **OpLoopMerge** instruction's *Continue Target*.

Return Block: A block containing an **OpReturn** or **OpReturnValue** branch.

Invocation: A single execution of an entry point in a SPIR-V module, operating only on the amount of data explicitly exposed by the semantics of the instructions. (Any implicit operation on additional instances of data would comprise additional invocations.) For example, in compute execution models, a single invocation operates only on a single work item, or, in a vertex execution model, a single invocation operates only on a single vertex.

Subgroup: The set of invocations exposed as running concurrently with the current invocation. In compute models, the current workgroup is a superset of the subgroup.

Invocation Group: The complete set of invocations collectively processing a particular compute workgroup or graphical operation, where the scope of a "graphical operation" is implementation dependent, but at least as large as a single point, line, triangle, or patch, and at most as large as a single rendering command, as defined by the client API.

Derivative Group: Defined only for the **Fragment** Execution Model: The set of invocations collectively processing a single point, line, or triangle, including any helper invocations.

Dynamic Instance: Within a single invocation, a single static instruction can be executed multiple times, giving multiple dynamic instances of that instruction. This can happen when the instruction is executed in a loop, or in a function called from multiple call sites, or combinations of multiple of these. Different loop iterations and different dynamic function-call-site chains yield different dynamic instances of such an instruction. Dynamic instances are distinguished by the control-flow path within an invocation, not by which invocation executed it. That is, different invocations of an entry point execute the same dynamic instances of an instruction when they follow the same control-flow path, starting from that entry point.

Dynamically Uniform: An *<id>* is dynamically uniform for a dynamic instance consuming it when its value is the same for all invocations (in the invocation group) that execute that dynamic instance.

Uniform Control Flow: Uniform control flow (or converged control flow) occurs when all invocations in the invocation group or derivative group execute the same control-flow path (and hence the same sequence of dynamic instances of instructions). Uniform control flow is the initial state at the entry point, and lasts until a conditional branch takes different control paths for different invocations (non-uniform or divergent control flow). Such divergence can reconverge, with all the invocations once again executing the same control-flow path, and this re-establishes the existence of uniform control flow. If control flow is uniform upon entry into a header block, and all invocations leave that dynamic instance of the header block's control-flow construct via the header block's declared merge block, then control flow reconverges to be uniform at that merge block.

2.3. Physical Layout of a SPIR-V Module and Instruction

A SPIR-V module is a single linear stream of *words*. The first words are shown in the following table:

Table 1. First Words of Physical Layout

Word Number	Contents
0	**Magic Number**.
1	Version number. The bytes are, high-order to low-order: *0* **|** *Major Number* **|** *Minor Number* **|** *0* Hence, version 1.00 is the value 0x00010000.
2	Generator’s magic number. It is associated with the tool that generated the module. Its value does not affect any semantics, and is allowed to be 0. Using a non-0 value is encouraged, and can be registered with Khronos at https://www.khronos.org/registry/spir-v/api/spir-v.xml.
3	*Bound*; where all *<id>s* in this module are guaranteed to satisfy *0 < id < Bound* *Bound* should be small, smaller is better, with all *<id>* in a module being densely packed and near 0.
4	0 (Reserved for instruction schema, if needed.)
5	First word of instruction stream, see below.

All remaining words are a linear sequence of instructions.

Each instruction is a stream of *words*:

Table 2. Instruction Physical Layout

Instruction Word Number	Contents
0	Opcode: The 16 high-order bits are the *WordCount* of the instruction. The 16 low-order bits are the opcode enumerant.
1	Optional instruction type *<id>* (presence determined by opcode).
.	Optional instruction *Result <id>* (presence determined by opcode).
.	Operand 1 (if needed)
.	Operand 2 (if needed)
…	…
WordCount - 1	Operand *N* (*N* is determined by WordCount minus the 1 to 3 words used for the opcode, instruction type *<id>*, and instruction *Result <id>*).

Instructions are variable length due both to having optional instruction type *<id>* and *Result <id>* words as well as a variable number of operands. The details for each specific instruction are given in the Binary Form section.

2.4. Logical Layout of a Module

The instructions of a SPIR-V module must be in the following order. For sections earlier than function definitions, it is invalid to use instructions other than those indicated.

1. All **OpCapability** instructions.
2. Optional **OpExtension** instructions (extensions to SPIR-V).
3. Optional **OpExtInstImport** instructions.
4. The single required **OpMemoryModel** instruction.
5. All entry point declarations, using **OpEntryPoint**.
6. All execution mode declarations, using **OpExecutionMode**.
7. These debug instructions, which must be in the following order:
    a. all **OpString**, **OpSourceExtension**, **OpSource**, and **OpSourceContinued**, without forward references.
    b. all **OpName** and all **OpMemberName**
8. All annotation instructions:
    a. all decoration instructions (**OpDecorate**, **OpMemberDecorate**, **OpGroupDecorate**, **OpGroupMemberDecorate**, and **OpDecorationGroup**).
9. All type declarations (**OpTypeXXX** instructions), all *constant instructions*, and all global variable declarations (all **OpVariable** instructions whose Storage Class is not **Function**). This is the preferred location for **OpUndef** instructions, though they can also appear in function bodies. All operands in all these instructions must be declared before being used. Otherwise, they can be in any order. This section is the first section to allow use of **OpLine** debug information.
10. All function declarations ("declarations" are functions without a body; there is no forward declaration to a function with a body). A function declaration is as follows.
    a. Function declaration, using **OpFunction**.
    b. Function parameter declarations, using **OpFunctionParameter**.
    c. Function end, using **OpFunctionEnd**.
11. All function definitions (functions with a body). A function definition is as follows.
    a. Function definition, using **OpFunction**.
    b. Function parameter declarations, using **OpFunctionParameter**.
    c. Block
    d. Block
    e. …
    f. Function end, using **OpFunctionEnd**.

Within a function definition:

- A block always starts with an **OpLabel** instruction. This may be immediately preceded by an **OpLine** instruction, but the **OpLabel** is considered as the beginning of the block.
- A block always ends with a branch instruction (see validation rules for more detail).
- All **OpVariable** instructions in a function must have a Storage Class of **Function**.
- All **OpVariable** instructions in a function must be in the first block in the function. These instructions, together with any immediately preceding **OpLine** instructions, must be the first instructions in that block. (Note the validation rules prevent **OpPhi** instructions in the first block of a function.)
- A function definition (starts with **OpFunction**) can be immediately preceded by an **OpLine** instruction.

Forward references (an operand *<id>* that appears before the *Result <id>* defining it) are allowed for:

- Operands that are an **OpFunction**. This allows for recursion and early declaration of entry points.
- Annotation-instruction operands. This is required to fully know everything about a type or variable once it is declared.
- Labels.

- Loops can have forward references to a phi function.
- An **OpTypeForwardPointer** has a forward reference to an **OpTypePointer**.
- An **OpTypeStruct** operand that's a forward reference to the *Pointer Type* operand to an **OpTypeForwardPointer**.
- The list of *<id>* provided in the **OpEntryPoint** instruction.

In all cases, there is enough type information to enable a single simple pass through a module to transform it. For example, function calls have all the type information in the call, phi-functions don't change type, and labels don't have type. The pointer forward reference allows structures to contain pointers to themselves or to be mutually recursive (through pointers), without needing additional type information.

The Validation Rules section lists additional rules that must be satisfied.

2.5. Instructions

Most instructions create a *Result <id>*, as provided in the *Result <id>* field of the instruction. These *Result <id>s* are then referred to by other instructions through their *<id>* operands. All instruction operands are specified in the Binary Form section.

Instructions are explicit about whether they require *immediates*, rather than an *<id>* referring to some other result. This is strictly known just from the opcode.

- An immediate 32-bit (or smaller) integer is always one operand directly holding a 32-bit two's-complement value.
- An immediate 32-bit float is always one operand, directly holding a 32-bit IEEE 754 floating-point representation.
- An immediate 64-bit float is always two operands, directly holding a 64-bit IEEE 754 representation. The low-order 32 bits appear in the first operand.

2.5.1. SSA Form

A module is always in static single assignment (SSA) form. That is, there is always exactly one instruction resulting in any particular *Result <id>*. Storing into variables declared in memory is not subject to this; such stores do not create *Result <id>s*. Accessing declared variables is done through:

- **OpVariable** to allocate an object in memory and create a *Result <id>* that is the name of a pointer to it.
- **OpAccessChain** or **OpInBoundsAccessChain** to create a pointer to a subpart of a *composite* object in memory.
- **OpLoad** through a pointer, giving the loaded object a *Result <id>* that can then be used as an operand in other instructions.
- **OpStore** through a pointer, to write a value. There is no *Result <id>* for an **OpStore**.

OpLoad and **OpStore** instructions can often be eliminated, using *intermediate* results instead. When this happens in multiple control-flow paths, these values need to be merged again at the path's merge point. Use **OpPhi** to merge such values together.

2.6. Entry Point and Execution Model

The **OpEntryPoint** instruction identifies an *entry point* with two key things: an execution model and a function definition. Execution models include **Vertex**, **GLCompute**, etc. (one for each graphical stage), as well as **Kernel** for OpenCL kernels. For the complete list, see Execution Model. An **OpEntryPoint** also supplies a name that can be used externally to identify the entry point, and a declaration of all the **Input** and **Output** variables that form its input/output interface.

The static function call graphs rooted at two entry points are allowed to overlap, so that function definitions and global variable definitions can be shared. The execution model and any execution modes associated with an entry point apply to the entire static function call graph rooted at that entry point. This rule implies that a function appearing in both call graphs of two distinct entry points may behave differently in each case. Similarly, variables whose semantics depend on properties of an entry point, e.g. those using the **Input** Storage Class, may behave differently when used in call graphs rooted in two different entry points.

2.7. Execution Modes

Information like the following is declared with **OpExecutionMode** instructions. For example,

- number of invocations (**Invocations**)
- vertex-order CCW (**VertexOrderCcw**)
- triangle strip generation (**OutputTriangleStrip**)
- number of output vertices (**OutputVertices**)
- etc.

For a complete list, see Execution Mode.

2.8. Types and Variables

Types are built up hierarchically, using **OpTypeXXX** instructions. The *Result <id>* of an **OpTypeXXX** instruction becomes a type *<id>* for future use where type *<id>s* are needed (therefore, **OpTypeXXX** instructions do not have a type *<id>*, like most other instructions do).

The "leaves" to start building with are types like **OpTypeFloat**, **OpTypeInt**, **OpTypeImage**, **OpTypeEvent**, etc. Other types are built up from the *Result <id>* of these. The numerical types are parameterized to specify bit width and signed vs. unsigned.

Higher-level types are then constructed using opcodes like **OpTypeVector**, **OpTypeMatrix**, **OpTypeImage**, **OpTypeArray**, **OpTypeRuntimeArray**, **OpTypeStruct**, and **OpTypePointer**. These are parameterized by number of components, array size, member lists, etc. The image types are parameterized by the return type, dimensionality, arrayness, etc. To do sampling or filtering operations, a type from **OpTypeSampledImage** is used that contains both an image and a sampler. Such a sampled image can be set directly by the API, or combined in a SPIR-V module from an independent image and an independent sampler.

Types are built bottom up: A parameterizing operand in a type must be defined before being used.

Some additional information about the type of an *<id>* can be provided using the decoration instructions (**OpDecorate**, **OpMemberDecorate**, **OpGroupDecorate**, **OpGroupMemberDecorate**, and **OpDecorationGroup**). These can add, for example, **Invariant** to an *<id>* created by another instruction. See the full list of Decorations in the Binary Form section.

Two different type *<id>s* form, by definition, two different types. It is valid to declare multiple aggregate type *<id>s* having the same opcode and operands. This is to allow multiple instances of aggregate types with the same structure to be decorated differently. (Different decorations are not required; two different aggregate type *<id>s* are allowed to have identical declarations and decorations, and will still be two different types.) Non-aggregate types are different: It is invalid to declare multiple type *<id>s* for the same scalar, vector, or matrix type. That is, non-aggregate type declarations must all have different opcodes or operands. (Note that non-aggregate types cannot be decorated in ways that affect their type.)

Variables are declared to be of an already built type, and placed in a Storage Class. Storage classes include **UniformConstant**, **Input**, **Workgroup**, etc. and are fully specified in Storage Class. Variables declared with the **Function** Storage Class can have their lifetime's specified within their function using the **OpLifetimeStart** and **OpLifetimeStop** instructions.

Intermediate results are typed by the instruction's type *<id>*, which must validate with respect to the operation being done.

Built-in variables needing special driver handling (having unique semantics) are declared using **OpDecorate** or **OpMemberDecorate** with the **BuiltIn** Decoration, followed by a **BuiltIn** enumerant. This decoration is applied to a variable or a structure-type member.

2.9. Function Calling

To call a function defined in the current module or a function declared to be imported from another module, use **OpFunctionCall** with an operand that is the *<id>* of the **OpFunction** to call, and the *<id>s* of the arguments to pass. All arguments are passed by value into the called function. This includes pointers, through which a callee object could be modified.

2.10. Extended Instruction Sets

Many operations and/or built-in function calls from high-level languages are represented through *extended instruction sets*. Extended instruction sets will include things like

- trigonometric functions: sin(), cos(), …
- exponentiation functions: exp(), pow(), …

- geometry functions: reflect(), smoothstep(), …
- functions having rich performance/accuracy trade-offs
- etc.

Non-extended instructions, those that are core SPIR-V instructions, are listed in the Binary Form section. Native operations include:

- Basic arithmetic: +, -, *, min(), scalar * vector, etc.
- Texturing, to help with back-end decoding and support special code-motion rules.
- Derivatives, due to special code-motion rules.

Extended instruction sets are specified in independent specifications. They can be referenced (but not specified) in this specification. The separate extended instruction set specification will specify instruction opcodes, semantics, and instruction names.

To use an extended instruction set, first import it by name string using **OpExtInstImport** and giving it a *Result <id>*:

```
<extinst-id> OpExtInstImport "name-of-extended-instruction-set"
```

The "name-of-extended-instruction-set" is a literal string. The standard convention for this string is

```
"<source language name>.<package name>.<version>"
```

For example "GLSL.std.450" could be the name of the core built-in functions for GLSL versions 450 and earlier.

Note There is nothing precluding having two "mirror" sets of instructions with different names but the same opcode values, which could, for example, let modifying just the import statement to change a performance/accuracy trade off.

Then, to call a specific extended instruction, use **OpExtInst**:

```
OpExtInst <extinst-id> instruction-number operand0, operand1, ...
```

Extended instruction-set specifications will provide semantics for each "instruction-number". It is up to the specific specification what the overloading rules are on operand type. The specification must be clear on its semantics, and producers/consumers of it must follow those semantics.

By convention, it is recommended that all external specifications include an **enum** {…} listing all the "instruction-numbers", and a mapping between these numbers and a string representing the instruction name. However, there are no requirements that instruction name strings are provided or mangled.

Note Producing and consuming extended instructions can be done entirely through numbers (no string parsing). An extended instruction set specification provides opcode enumerant values for the instructions, and these will be produced by the front end and consumed by the back end.

2.11. Structured Control Flow

SPIR-V can explicitly declare structured control-flow *constructs* using merge instructions. These explicitly declare a header block before the control flow diverges and a merge block where control flow subsequently converges. These blocks delimit constructs that must nest, and can only be entered and exited in structured ways, as per the following.

Structured control-flow declarations must satisfy the following rules:

- the merge block declared by a header block cannot be a merge block declared by any other header block
- each header block must dominate its merge block, unless the merge block is unreachable in the CFG
- all CFG back edges must branch to a loop header, with each loop header having exactly one back

edge branching to it

- for a given loop, its back-edge block must post dominate the **OpLoopMerge's** *Continue Target*, and that *Continue Target* must dominate that back-edge block

A structured control-flow *construct* is then defined as one of:

- a *selection construct*: the set of blocks dominated by a selection header, minus the set of blocks dominated by the header's merge block
- a *continue construct*: the set of blocks dominated by an **OpLoopMerge's** *Continue Target* and post dominated by the corresponding back-edge block
- a *loop construct*: the set of blocks dominated by a loop header, minus the set of blocks dominated by the loop's merge block, minus the loop's corresponding *continue construct*
- a *case construct*: the set of blocks dominated by an **OpSwitch** *Target* or *Default*, minus the set of blocks dominated by the **OpSwitch's** merge block (this construct is only defined for those **OpSwitch** *Target* or *Default* that are not equal to the **OpSwitch's** corresponding merge block)

The above structured control-flow constructs must satisfy the following rules:

- if a construct contains another header block, then it also contains that header's corresponding merge block
- the only blocks in a construct that can branch outside the construct are
  - a block branching to the construct's merge block
  - a block branching from one *case construct* to another, for the same **OpSwitch**
  - a continue block for the innermost loop it is nested inside of
  - a break block for the innermost loop it is nested inside of
  - a return block
- additionally for switches:
  - an **OpSwitch** block dominates all its defined *case constructs*
  - each *case construct* has at most one branch to another *case construct*
  - each *case construct* is branched to by at most one other *case construct*
  - if *Target T1* branches to *Target T2*, or if *Target T1* branches to the *Default* and the *Default* branches to *Target T2*, then *T1* must immediately precede *T2* in the list of the **OpSwitch** *Target* operands

2.12. Specialization

Specialization is intended for constant objects that will not have known constant values until after initial generation of a SPIR-V module. Such objects are called *specialization constants*.

A SPIR-V module containing specialization constants can consume one or more externally provided *specializations*: A set of final constant values for some subset of the module's *specialization constants*. Applying these final constant values yields a new module having fewer remaining specialization constants. A module also contains default values for any specialization constants that never get externally specialized.

Note	No optimizing transforms are required to make a *specialized* module functionally correct. The specializing transform is straightforward and explicitly defined below.

Note	Ad hoc specializing should not be done through constants (**OpConstant** or **OpConstantComposite**) that get overwritten: A SPIR-V → SPIR-V transform might want to do something irreversible with the value of such a constant, unconstrained from the possibility that its value could be later changed.

Within a module, a *Specialization Constant* is declared with one of these instructions:

- **OpSpecConstantTrue**
- **OpSpecConstantFalse**
- **OpSpecConstant**

- **OpSpecConstantComposite**
- **OpSpecConstantOp**

The literal operands to **OpSpecConstant** are the default numerical specialization constants. Similarly, the "**True**" and "**False**" parts of **OpSpecConstantTrue** and **OpSpecConstantFalse** provide the default Boolean specialization constants. These default values make an external specialization optional. However, such a default constant is applied only after all external specializations are complete, and none contained a specialization for it.

An external specialization is provided as a logical list of pairs. Each pair is a **SpecId** Decoration of a scalar specialization instruction along with its specialization constant. The numeric values are exactly what the operands would be to a corresponding **OpConstant** instruction. Boolean values are true if non-zero and false if zero.

Specializing a module is straightforward. The following specialization-constant instructions can be updated with specialization constants, and replaced in place, leaving everything else in the module exactly the same:

```
       OpSpecConstantTrue -> OpConstantTrue or OpConstantFalse
      OpSpecConstantFalse -> OpConstantTrue or OpConstantFalse
           OpSpecConstant -> OpConstant
  OpSpecConstantComposite -> OpConstantComposite
```

The **OpSpecConstantOp** instruction is specialized by executing the operation and replacing the instruction with the result. The result can be expressed in terms of a constant instruction that is not a specialization-constant instruction. (Note, however, this resulting instruction might not have the same size as the original instruction, so is not a "replaced in place" operation.)

When applying an external specialization, the following (and only the following) must be modified to be non-specialization-constant instructions:

- specialization-constant instructions with values provided by the specialization
- specialization-constant instructions that consume nothing but non-specialization constant instructions (including those that the partial specialization transformed from specialization-constant instructions; these are in order, so it is a single pass to do so)

A full specialization can also be done, when requested or required, in which all specialization-constant instructions will be modified to non-specialization-constant instructions, using the default values where required.

2.13. Linkage

The ability to have partially linked modules and libraries is provided as part of the **Linkage** capability.

By default, functions and global variables are private to a module and cannot be accessed by other modules. However, a module may be written to *export* or *import* functions and global (module scope) variables. Imported functions and global variable definitions are resolved at linkage time. A module is considered to be partially linked if it depends on imported values.

Within a module, imported or exported values are decorated using the **Linkage Attributes** **Decoration**. This decoration assigns the following linkage attributes to decorated values:

- A *Linkage Type*.
- A *name*, which is a *Literal String*, and is used to uniquely identify exported values.

Note When resolving imported functions, the *Function Control* and all *Function Parameter Attributes* are taken from the function definition, and not from the function declaration.

2.14. Relaxed Precision

The **RelaxedPrecision** Decoration allows 32-bit integer and 32-bit floating-point operations to execute with a relaxed precision of somewhere between 16 and 32 bits.

For a floating-point operation, operating at relaxed precision means that the minimum requirements for range and precision are as follows:

- the floating point range may be as small as $(-2^{14}, 2^{14})$
- the floating point magnitude range may be as small as $(2^{-14}, 2^{14})$
- the relative floating point precision may be as small as 2^{-10}

Relative floating-point precision is defined as the worst case (i.e. largest) ratio of the smallest step in relation to the value for all non-zero values:

$\text{Precision}_{relative} = (\text{abs}(v_1 - v_2)_{min} / \text{abs}(v_1))_{max}$ for $v_1 \neq 0$, $v_2 \neq 0$, $v_1 \neq v_2$

For integer operations, operating at relaxed precision means that the operation will be evaluated by an operation in which, for some N, $16 \leq N \leq 32$:

- the operation is executed as though its type were N bits in size, and
- the result is zero or sign extended to 32 bits as determined by the signedness of the result type of the operation.

The **RelaxedPrecision** Decoration can be applied to:

- The *<id>* of a variable, where the variable's type is a scalar, vector, or matrix, or an array of scalar, vector, or matrix. In all cases, the components in the type must be a 32-bit numerical type.
- The *Result <id>* of an instruction that operates on numerical types, meaning the instruction is to operate at relaxed precision.
- The *Result <id>* of an **OpFunction** meaning the function's returned result is at relaxed precision. It cannot be applied to **OpTypeFunction** or to an **OpFunction** whose return type is **OpTypeVoid**.
- A structure-type member (through OpMemberDecorate).

When applied to a variable or structure member, all loads and stores from the decorated object may be treated as though they were decorated with **RelaxedPrecision**. Loads may also be decorated with **RelaxedPrecision**, in which case they are treated as operating at relaxed precision.

All loads and stores involving relaxed precision still read and write 32 bits of data, respectively. Floating-point data read or written in such a manner is written in full 32-bit floating-point format. However, a load or store might reduce the precision (as allowed by **RelaxedPrecision**) of the destination value.

For debugging portability of floating-point operations, **OpQuantizeToF16** may be used to explicitly reduce the precision of a relaxed-precision result to 16-bit precision. (Integer-result precision can be reduced, for example, using left- and right-shift opcodes.)

2.15. Debug Information

Debug information is supplied with:

- Source-code text through **OpString**, **OpSource**, and **OpSourceContinued**.
- Object names through **OpName** and **OpMemberName**.
- Line numbers through **OpLine**.

A module will not lose any semantics when all such instructions are removed.

2.15.1. Function-Name Mangling

There is no functional dependency on how functions are named. Signature-typing information is explicitly provided, without any need for name "unmangling". (Valid modules can be created without inclusion of mangled names.)

By convention, for debugging purposes, modules with **OpSource** *Source Language* of OpenCL use the Itanium name-mangling standard.

2.16. Validation Rules

2.16.1. Universal Validation Rules

All modules must obey the following, or it is an invalid module:

- The stream of instructions must be ordered as described in the Logical Layout section.
- Any use of a feature described by a capability in the capability section requires that capability to be declared, either directly, or as a "depends on" capability on a capability that is declared.

- Non-structure types (scalars, vectors, arrays, etc.) with the same operand parameterization cannot be type aliases. For non-structures, two type *<id>s* match if-and-only-if the types match.
- If the **Logical** addressing model is selected:
  - **OpVariable** cannot allocate an object whose type is a pointer type (that is, it cannot create an object in memory that is itself a pointer and whose result would thus be a pointer to a pointer)
  - A pointer can only be an operand to the following instructions:
    - **OpLoad**
    - **OpStore**
    - **OpAccessChain**
    - **OpInBoundsAccessChain**
    - **OpFunctionCall**
    - **OpImageTexelPointer**
    - **OpCopyMemory**
    - **OpCopyObject**
    - all **OpAtomic** instructions
  - A pointer can be the *Result <id>* of only the following instructions:
    - **OpVariable**
    - **OpAccessChain**
    - **OpInBoundsAccessChain**
    - **OpFunctionParameter**
    - **OpImageTexelPointer**
    - **OpCopyObject**
  - All indexes in **OpAccessChain** and **OpInBoundsAccessChain** that are **OpConstant** with type of **OpTypeInt** with a *signedness* of 1 must not have their sign bit set.
- SSA
  - Each *<id>* must appear exactly once as the *Result <id>* of an instruction.
  - The definition of an SSA *<id>* should dominate all uses of it, with the following exceptions:
    - Function calls may call functions not yet defined. However, note that the function's argument and return types will already be known at the call site.
    - Uses in a phi-function in a loop may consume definitions in the loop that don't dominate the use.
- Entry point and execution model
  - There is at least one **OpEntryPoint** instruction, unless the **Linkage** capability is being used.
  - No function can be targeted by both an **OpEntryPoint** instruction and an **OpFunctionCall** instruction.
- Functions
  - A function declaration (an **OpFunction** with no basic blocks), must have a **Linkage Attributes Decoration** with the **Import Linkage Type**.
  - A function definition (an **OpFunction** with basic blocks) cannot be decorated with the **Import Linkage Type**.
  - A function cannot have both a declaration and a definition (no forward declarations).
- Global (Module Scope) Variables
  - It is illegal to initialize an imported variable. This means that a module-scope **OpVariable** with initialization value cannot be marked with the **Import Linkage Type**.
- Control-Flow Graph (CFG)
  - Blocks exist only within a function.
  - The first block in a function definition is the entry point of that function and cannot be the target of any branch. (Note this means it will have no **OpPhi** instructions.)
  - The order of blocks in a function must satisfy the rule that blocks appear before all blocks they

dominate.
    - Each block starts with a label.
        - A label is made by **OpLabel**.
        - This includes the first block of a function (**OpFunction** is not a label).
        - Labels are used only to form blocks.
    - The last instruction of each block is a branch instruction.
    - Branch instructions can only appear as the last instruction in a block.
    - **OpLabel** instructions can only appear within a function.
    - All branches within a function must be to labels in that function.
- All **OpFunctionCall** *Function* operands are an *<id>* of an **OpFunction** in the same module.
- Data rules
    - Scalar floating-point types can be parameterized only as 32 bit, plus any additional sizes enabled by capabilities.
    - Scalar integer types can be parameterized only as 32 bit, plus any additional sizes enabled by capabilities.
    - Vector types can only be parameterized with numerical types or the OpTypeBool type.
    - Vector types for can only be parameterized as having 2, 3, or 4 components, plus any additional sizes enabled by capabilities.
    - Matrix types can only be parameterized with floating-point types.
    - Matrix types can only be parameterized as having only 2, 3, or 4 columns.
    - Specialization constants (see Specialization) are limited to integers, Booleans, floating-point numbers, and vectors of these.
    - Forward reference operands in an **OpTypeStruct**
        - must be later declared with **OpTypePointer**
        - the type pointed to must be an OpTypeStruct
        - had an earlier **OpTypeForwardPointer** forward reference to the same *<id>*
    - All **OpSampledImage** instructions must be in the same block in which their *Result <id>* are consumed. *Result <id>* from **OpSampledImage** instructions must not appear as operands to **OpPhi** instructions or **OpSelect** instructions, or any instructions other than the image lookup and query instructions specified to take an operand whose type is **OpTypeSampledImage**.
    - Instructions for extracting a scalar image or scalar sampler out of a composite must only use dynamically-uniform indexes. They must be in the same block in which their *Result <id>* are consumed. Such *Result <id>* must not appear as operands to **OpPhi** instructions or **OpSelect** instructions, or any instructions other than the image instructions specified to operate on them.
- Decoration rules
    - The **Aliased** Decoration can only be applied to *intermediate* objects that are pointers to non-void types.
    - The **Linkage Attributes** Decoration cannot be applied to functions targeted by an **OpEntryPoint** instruction.
    - A BuiltIn Decoration can only be applied as follows:
        - When applied to a structure-type member, all members of that structure type must also be decorated with **BuiltIn**. (No allowed mixing of built-in variables and non-built-in variables within a single structure.)
        - When applied to a structure-type member, that structure type cannot be contained as a member of another structure type.
        - There is at most one object per Storage Class that can contain a structure type containing members decorated with **BuiltIn**, consumed per entry-point.
- **OpLoad** and **OpStore** can only consume objects whose type is a pointer.
- A *Result <id>* resulting from an instruction within a function can only be used in that function.
- A function call must have the same number of arguments as the function definition (or declaration) has parameters, and their respective types must match.

- An instruction requiring a specific number of operands must have that many operands. The *word count* must agree.
- Each opcode specifies its own requirements for number and type of operands, and these must be followed.
- Atomic access rules
  - The pointers taken by atomic operation instructions must be a pointer into one of the following Storage Classes:
    - **Uniform** when used with the **BufferBlock** **Decoration**
    - **Workgroup**
    - **CrossWorkgroup**
    - **Generic**
    - **AtomicCounter**
    - **Image**
  - The only instructions that can operate on a pointer to the **AtomicCounter** Storage Class are
    - **OpAtomicLoad**
    - **OpAtomicIIncrement**
    - **OpAtomicIDecrement**
  - All pointers used in atomic operation instructions must be pointers to one of the following:
    - 32-bit scalar integer
    - 64-bit scalar integer

2.16.2. Validation Rules for Shader **Capabilities**

- CFG:
  - Loops must be structured, having an **OpLoopMerge** instruction in their header.
  - Selections must be structured, having an **OpSelectionMerge** instruction in their header.
- Entry point and execution model
  - Each *entry point* in a module, along with its corresponding static call tree within that module, forms a complete pipeline stage.
  - Each **OpEntryPoint** with the **Fragment** Execution Model must have an **OpExecutionMode** for either the **OriginLowerLeft** or the **OriginUpperLeft** Execution Mode. (Exactly one of these is required.)
  - An **OpEntryPoint** with the **Fragment** Execution Model can set at most one of the **DepthGreater**, **DepthLess**, or **DepthUnchanged** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Modes can set at most one of the **SpacingEqual**, **FractionalEven**, or **FractionalOdd** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Models can set at most one of the **Triangles**, **Quads**, or **Isolines** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Models can set at most one of the **VertexOrderCw** or **VertexOrderCcw** **Execution Modes**.
  - An **OpEntryPoint** with the **Geometry** Execution Model must set exactly one of the **InputPoints**, **InputLines**, **InputLinesAdjacency**, **Triangles**, or **TrianglesAdjacency** **Execution Modes**.
  - An **OpEntryPoint** with the **Geometry** Execution Model must set exactly one of the **OutputPoints**, **OutputLineStrip**, or **OutputTriangleStrip** **Execution Modes**.
- Composite objects in the **UniformConstant**, **Uniform**, and **PushConstant** Storage Classes must be explicitly laid out. The following apply to all the aggregate and matrix types describing such an object, recursively through their nested types:
  - Each structure-type member must have an **Offset** Decoration.
  - Each array type must have an **ArrayStride** Decoration.
  - Each structure-type member that is a matrix or array-of-matrices must have be decorated with
    - a **MatrixStride** Decoration, and

        - one of the **RowMajor** or **ColMajor** Decorations.
    - The **ArrayStride**, **MatrixStride**, and **Offset** Decorations must be large enough to hold the size of the objects they affect (that is, specifying overlap is invalid).
    - The **MatrixStride** on a **RowMajor** (**ColMajor**) matrix must be padded to hold a row (column) of 4 components, when the matrix only has 3 columns (rows). In all other uses of **MatrixStride**, no padding is allowed.
- For structure objects in the **Input** and **Output** Storage Classes, the following apply:
    - When applied to structure-type members, the Decorations **Noperspective**, **Flat**, **Patch**, **Centroid**, and **Sample** can only be applied to the top-level members of the structure type. (Nested objects' types cannot be structures whose members are decorated with these decorations.)
- Decorations
    - At most one of **Noperspective** or **Flat** Decorations can be applied to the same object or member.
    - At most one of **Patch**, **Centroid**, or **Sample** Decorations can be applied to the same object or member.
    - At most one of **RowMajor** and **ColMajor** Decorations can be applied to a structure type.
    - At most one of **Block** and **BufferBlock** Decorations can be applied to a structure type.
- All *<id>* used for Scope and Memory Semantics must be of an OpConstant.

2.16.3. Validation Rules for Kernel Capabilities

- The *Signedness* in **OpTypeInt** must always be 0.

2.17. Universal Limits

These quantities are minimum limits for all implementations and validators. Implementations are allowed to support larger quantities. Specific APIs may impose larger minimums. See Language Capabilities.

Validators must either

- inform when these limits are crossed, or
- be explicitly parameterized with larger limits.

Table 3. Limits

Limited Entity	Minimum Limit	
	Decimal	Hexadecimal
Characters in a literal string	65,535	FFFF
Instruction word count	65,535	FFFF
Result *<id>* bound See Physical Layout for the shader-specific bound.	4,194,303	3FFFFF
Control-flow nesting depth Measured per function, in program order, counting the maximum number of **OpBranch**, **OpBranchConditional**, or **OpSwitch** that are seen without yet seeing their corresponding *Merge Block*, as declared by **OpSelectionMerge** or **OpLoopMerge**.	1023	3FF
Global variables (Storage Class other than **Function**)	65,535	FFFF
Local variables (**Function** Storage Class)	524,287	7FFFF
Decorations per target *<id>*	Number of entries in the **Decoration** table.	
Execution modes per entry point	255	FF
Indexes for **OpAccessChain**, **OpInBoundsAccessChain**, **OpPtrAccessChain**, **OpInBoundsPtrAccessChain**, **OpCompositeExtract**, and **OpCompositeInsert**	255	FF
Number of function parameters, per function declaration	255	FF
OpFunctionCall actual arguments	255	FF
OpExtInst actual arguments	255	FF
OpSwitch (literal, label) pairs	16,383	3FFF
OpTypeStruct members	16,383	3FFF
Structure nesting depth	255	FF

2.18. Memory Model

A memory model is chosen using a single **OpMemoryModel** instruction near the beginning of the module. This selects both an addressing model and a memory model.

The **Logical** addressing model means pointers are abstract, having no physical size or numeric value. In this mode, pointers can only be created from existing objects, and they cannot be stored into an object.

The non-**Logical** addressing models allow physical pointers to be formed. **OpVariable** can be used to create objects that hold pointers. These are declared for a specific Storage Class. Pointers for one Storage Class cannot be used to access objects in another Storage Class. However, they can be converted with conversion opcodes. Any particular addressing model must describe the bit width of pointers for each of the storage classes.

2.18.1. Memory Layout

When memory is shared between a SPIR-V module and an API, its contents are transparent, and must be agreed on. For example, the **Offset**, **MatrixStride**, and **ArrayStride** Decorations applied to members of a struct object can partially define how the memory is laid out. In addition, the following are always true, applied recursively as needed, of the offsets within the memory buffer:

- a vector consumes contiguous memory with lower-numbered components appearing in smaller offsets than higher-numbered components, and with component 0 starting at the vector's **Offset** Decoration, if present
- in an array, lower-numbered elements appear at smaller offsets than higher-numbered elements, with element 0 starting at the **Offset** Decoration for the array, if present
- a structure has lower-numbered members appearing at smaller offsets than higher-numbered members, with member 0 starting at the **Offset** Decoration for the structure, if present
- in a matrix, lower-numbered columns appear at smaller offsets than higher-numbered columns, and lower-numbered components within the matrix's vectors appearing at smaller offsets than high-numbered components, with component 0 of column 0 starting at the **Offset** Decoration, if present (the **RowMajor** and **ColMajor** Decorations dictate what is contiguous)

2.18.2. Aliasing

Here, *aliasing* means one of:

- Two or more pointers that point into overlapping parts of the same underlying object. That is, two *intermediates*, both of which are typed pointers, that can be dereferenced (in bounds) such that both dereferences access the same memory.
- Images, buffers, or other externally allocated objects where a function might access the same underlying memory via accesses to two different objects.

How aliasing is managed depends on the **Memory Model**:

- The simple and GLSL memory models can assume that aliasing is generally not present. Specifically, the compiler is free to compile as if aliasing is not present, unless a pointer is explicitly indicated to be an alias. This is indicated by applying the **Aliased Decoration** to an *intermediate* object's *<id>*. Applying **Restrict** is allowed, but has no effect.
- The OpenCL memory models must assume that aliasing is generally present. Specifically, the compiler must compile as if aliasing is present, unless a pointer is explicitly indicated to not alias. This is done by applying the **Restrict Decoration** to an *intermediate* object's *<id>*. Applying **Aliased** is allowed, but has no effect.

It is invalid to apply both **Restrict** and **Aliased** to the same *<id>*.

2.19. Derivatives

Derivatives appear only in the **Fragment** Execution Model. They can be implicit or explicit. Some image instructions consume implicit derivatives, while the derivative instructions compute explicit derivatives. In all cases, derivatives are well defined only if the derivative group has uniform control flow.

2.20. Code Motion

Texturing instructions in the Fragment Execution Model that rely on an implicit derivative cannot be moved into control flow that is not known to be uniform control flow within each derivative group.

3. Binary Form

This section contains the exact form for all instructions, starting with the numerical values for all fields. See Physical Layout for the order words appear in.

3.1. Magic Number

Magic number for a SPIR-V module.

Tip **Endianness:** A module is defined as a stream of words, not a stream of bytes. However, if stored as a stream of bytes (e.g., in a file), the magic number can be used to deduce what endianness to apply to convert the byte stream back to a word stream.

Magic Number
0x07230203

3.2. Source Language

The source language is for debug purposes only, with no semantics that affect the meaning of other parts of the module. Used by **OpSource**.

Source Language	
0	**Unknown**
1	**ESSL**
2	**GLSL**
3	**OpenCL_C**
4	**OpenCL_CPP**

3.3. Execution Model

Used by **OpEntryPoint**.

Execution Model		**Required Capability**
0	**Vertex** Vertex shading stage.	**Shader**
1	**TessellationControl** Tessellation control (or hull) shading stage.	**Tessellation**
2	**TessellationEvaluation** Tessellation evaluation (or domain) shading stage.	**Tessellation**
3	**Geometry** Geometry shading stage.	**Geometry**
4	**Fragment** Fragment shading stage.	**Shader**
5	**GLCompute** Graphical compute shading stage.	**Shader**

	Execution Model	Required Capability
6	**Kernel** Compute kernel.	**Kernel**

3.4. Addressing Model

Used by **OpMemoryModel**.

	Addressing Model	Required Capability
0	**Logical**	
1	**Physical32** Indicates a 32-bit module, where the address width is equal to 32 bits.	**Addresses**
2	**Physical64** Indicates a 64-bit module, where the address width is equal to 64 bits.	**Addresses**

3.5. Memory Model

Used by **OpMemoryModel**.

	Memory Model	Required Capability
0	**Simple** No shared memory consistency issues.	**Shader**
1	**GLSL450** Memory model needed by later versions of GLSL and ESSL. Works across multiple versions.	**Shader**
2	**OpenCL** OpenCL memory model.	**Kernel**

3.6. Execution Mode

Declare the modes an entry point will execute in. Used by **OpExecutionMode**.

	Execution Mode	Required Capability	Extra Operands
0	**Invocations** Number of times to invoke the geometry stage for each input primitive received. The default is to run once for each input primitive. If greater than the target-dependent maximum, it will fail to compile. Only valid with the **Geometry** Execution Model.	**Geometry**	*Literal Number* *Number of invocations*
1	**SpacingEqual** Requests the tessellation primitive generator to divide edges into a collection of equal-sized segments. Only valid with one of the tessellation Execution Models.	**Tessellation**	
2	**SpacingFractionalEven** Requests the tessellation primitive generator	**Tessellation**	

	Execution Mode	Required Capability	Extra Operands
	to divide edges into an even number of equal-length segments plus two additional shorter fractional segments. Only valid with one of the tessellation Execution Models.		
3	**SpacingFractionalOdd** Requests the tessellation primitive generator to divide edges into an odd number of equal-length segments plus two additional shorter fractional segments. Only valid with one of the tessellation Execution Models.	**Tessellation**	
4	**VertexOrderCw** Requests the tessellation primitive generator to generate triangles in clockwise order. Only valid with one of the tessellation Execution Models.	**Tessellation**	
5	**VertexOrderCcw** Requests the tessellation primitive generator to generate triangles in counter-clockwise order. Only valid with one of the tessellation Execution Models.	**Tessellation**	
6	**PixelCenterInteger** Pixels appear centered on whole-number pixel offsets. E.g., the coordinate (0.5, 0.5) appears to move to (0.0, 0.0). Only valid with the **Fragment** Execution Model. If a **Fragment** entry point does not have this set, pixels appear centered at offsets of (0.5, 0.5) from whole numbers	**Shader**	
7	**OriginUpperLeft** Pixel coordinates appear to originate in the upper left, and increase toward the right and downward. Only valid with the **Fragment** Execution Model.	**Shader**	
8	**OriginLowerLeft** Pixel coordinates appear to originate in the lower left, and increase toward the right and upward. Only valid with the **Fragment** Execution Model.	**Shader**	
9	**EarlyFragmentTests** Fragment tests are to be performed before fragment shader execution. Only valid with the **Fragment** Execution Model.	**Shader**	
10	**PointMode** Requests the tessellation primitive generator to generate a point for each distinct vertex in the subdivided primitive, rather than to generate lines or triangles. Only valid with one of the tessellation Execution Models.	**Tessellation**	
11	**Xfb** This stage will run in transform feedback-capturing mode and this module is responsible for describing the transform-feedback setup. See the **XfbBuffer**, **Offset**, and **XfbStride** **Decorations**.	**TransformFeedback**	

Execution Mode	Required Capability	Extra Operands		
12 **DepthReplacing** This mode must be declared if this module potentially changes the fragment's depth. Only valid with the **Fragment** Execution Model.	**Shader**			
14 **DepthGreater** External optimizations may assume depth modifications will leave the fragment's depth as greater than or equal to the fragment's interpolated depth value (given by the *z* component of the **FragCoord** **BuiltIn** decorated variable). Only valid with the **Fragment** Execution Model.	**Shader**			
15 **DepthLess** External optimizations may assume depth modifications leave the fragment's depth less than the fragment's interpolated depth value, (given by the *z* component of the **FragCoord** **BuiltIn** decorated variable). Only valid with the **Fragment** Execution Model.	**Shader**			
16 **DepthUnchanged** External optimizations may assume this stage did not modify the fragment's depth. However, **DepthReplacing** mode must accurately represent depth modification. Only valid with the **Fragment** Execution Model.	**Shader**			
17 **LocalSize** Indicates the work-group size in the *x*, *y*, and *z* dimensions. Only valid with the **GLCompute** or **Kernel** Execution Models.		*Literal Number* *x size*	*Literal Number* *y size*	*Literal Number* *z size*
18 **LocalSizeHint** A hint to the compiler, which indicates the most likely to be used work-group size in the *x*, *y*, and *z* dimensions. Only valid with the **Kernel** Execution Model.	**Kernel**	*Literal Number* *x size*	*Literal Number* *y size*	*Literal Number* *z size*
19 **InputPoints** Stage input primitive is *points*. Only valid with the **Geometry** Execution Model.	**Geometry**			
20 **InputLines** Stage input primitive is *lines*. Only valid with the **Geometry** Execution Model.	**Geometry**			
21 **InputLinesAdjacency** Stage input primitive is *lines adjacency*. Only valid with the **Geometry** Execution Model.	**Geometry**			
22 **Triangles** For a geometry stage, input primitive is *triangles*. For a tessellation stage, requests the tessellation primitive generator to generate triangles. Only valid with the **Geometry** or one of the tessellation Execution Models.	**Geometry**, **Tessellation**			
23 **InputTrianglesAdjacency** Geometry stage input primitive is *triangles adjacency*. Only valid with the **Geometry**	**Geometry**			

Execution Mode	Required Capability	Extra Operands
Execution Model.		
24 **Quads** Requests the tessellation primitive generator to generate *quads*. Only valid with one of the tessellation Execution Models.	**Tessellation**	
25 **Isolines** Requests the tessellation primitive generator to generate *isolines*. Only valid with one of the tessellation Execution Models.	**Tessellation**	
26 **OutputVertices** For a geometry stage, the maximum number of vertices the shader will ever emit in a single invocation. For a tessellation-control stage, the number of vertices in the output patch produced by the tessellation control shader, which also specifies the number of times the tessellation control shader is invoked. Only valid with the **Geometry** or one of the tessellation Execution Models.	**Geometry**, **Tessellation**	*Literal Number* *Vertex count*
27 **OutputPoints** Stage output primitive is *points*. Only valid with the **Geometry** Execution Model.	**Geometry**	
28 **OutputLineStrip** Stage output primitive is *line strip*. Only valid with the **Geometry** Execution Model.	**Geometry**	
29 **OutputTriangleStrip** Stage output primitive is *triangle strip*. Only valid with the **Geometry** Execution Model.	**Geometry**	
30 **VecTypeHint** A hint to the compiler, which indicates that most operations used in the entry point are explicitly vectorized using a particular vector type. The 16 high-order bits of *Vector Type* operand specify the *number of components* of the vector. The 16 low-order bits of *Vector Type* operand specify the *data type* of the vector. These are the legal *data type* values: *0* represents an 8-bit integer value. *1* represents a 16-bit integer value. *2* represents a 32-bit integer value. *3* represents a 64-bit integer value. *4* represents a 16-bit float value. *5* represents a 32-bit float value. *6* represents a 64-bit float value. Only valid with the **Kernel** Execution Model.	**Kernel**	*Literal Number* *Vector type*
31 **ContractionOff** Indicates that floating-point-expressions contraction is disallowed. Only valid with the **Kernel** Execution Model.	**Kernel**	
33 **Initializer** Indicates that this entry point is a module initializer.	**Kernel**	

3.7. Storage Class

Class of storage for declared variables (does not include *intermediate* values). Used by:

- **OpTypePointer**
- **OpTypeForwardPointer**
- **OpVariable**
- **OpGenericCastToPtrExplicit**

	Storage Class	Required Capability
0	**UniformConstant** Shared externally, visible across all functions in all invocations in all work groups. Graphics uniform memory. OpenCL constant memory. Variables declared with this storage class are read-only, and cannot have initializers.	
1	**Input** Input from pipeline. Visible across all functions in the current invocation. Variables declared with this storage class are read-only, and cannot have initializers.	
2	**Uniform** Shared externally, visible across all functions in all invocations in all work groups. Graphics uniform blocks and buffer blocks.	**Shader**
3	**Output** Output to pipeline. Visible across all functions in the current invocation.	**Shader**
4	**Workgroup** Shared across all invocations within a work group. Visible across all functions. The OpenGL "shared" storage qualifier. OpenCL local memory.	
5	**CrossWorkgroup** Visible across all functions of all invocations of all work groups. OpenCL global memory.	
6	**Private** Visible to all functions in the current invocation. Regular global memory.	**Shader**
7	**Function** Visible only within the declaring function of the current invocation. Regular function memory.	
8	**Generic** For generic pointers, which overload the **Function**, **Workgroup**, and **CrossWorkgroup** Storage Classes.	**GenericPointer**
9	**PushConstant** For holding push-constant memory, visible across all functions in all invocations in all work groups. Intended to contain a small bank of values pushed from the API. Variables declared with this storage class are read-only, and cannot have initializers.	**Shader**

3.8. Dim

Dimensionality of an image. Used by **OpTypeImage**.

Dim		Required Capability
0	**1D**	**Sampled1D**
1	**2D**	
2	**3D**	
3	**Cube**	**Shader**
4	**Rect**	**SampledRect**
5	**Buffer**	**SampledBuffer**
6	**SubpassData**	**InputAttachment**

3.9. Sampler Addressing Mode

Addressing mode for creating constant samplers. Used by **OpConstantSampler**.

Sampler Addressing Mode		Required Capability
0	**None** The image coordinates used to sample elements of the image refer to a location inside the image, otherwise the results are undefined.	**Kernel**
1	**ClampToEdge** Out-of-range image coordinates are clamped to the extent.	**Kernel**
2	**Clamp** Out-of-range image coordinates will return a border color.	**Kernel**
3	**Repeat** Out-of-range image coordinates are wrapped to the valid range. Can only be used with normalized coordinates.	**Kernel**
4	**RepeatMirrored** Flip the image coordinate at every integer junction. Can only be used with normalized coordinates.	**Kernel**

3.10. Sampler Filter Mode

Filter mode for creating constant samplers. Used by **OpConstantSampler**.

Sampler Filter Mode		Required Capability
0	**Nearest** Use filter nearest mode when performing a read image operation.	**Kernel**

	Sampler Filter Mode	Required Capability
1	**Linear** Use filter linear mode when performing a read image operation.	**Kernel**

3.11. Image Format

Declarative image format. Used by **OpTypeImage**.

	Image Format	Required Capability
0	**Unknown**	
1	**Rgba32f**	**Shader**
2	**Rgba16f**	**Shader**
3	**R32f**	**Shader**
4	**Rgba8**	**Shader**
5	**Rgba8Snorm**	**Shader**
6	**Rg32f**	**StorageImageExtendedFormats**
7	**Rg16f**	**StorageImageExtendedFormats**
8	**R11fG11fB10f**	**StorageImageExtendedFormats**
9	**R16f**	**StorageImageExtendedFormats**
10	**Rgba16**	**StorageImageExtendedFormats**
11	**Rgb10A2**	**StorageImageExtendedFormats**
12	**Rg16**	**StorageImageExtendedFormats**
13	**Rg8**	**StorageImageExtendedFormats**
14	**R16**	**StorageImageExtendedFormats**
15	**R8**	**StorageImageExtendedFormats**
16	**Rgba16Snorm**	**StorageImageExtendedFormats**
17	**Rg16Snorm**	**StorageImageExtendedFormats**
18	**Rg8Snorm**	**StorageImageExtendedFormats**
19	**R16Snorm**	**StorageImageExtendedFormats**
20	**R8Snorm**	**StorageImageExtendedFormats**
21	**Rgba32i**	**Shader**
22	**Rgba16i**	**Shader**
23	**Rgba8i**	**Shader**
24	**R32i**	**Shader**
25	**Rg32i**	**StorageImageExtendedFormats**
26	**Rg16i**	**StorageImageExtendedFormats**

Image Format		Required Capability
27	**Rg8i**	**StorageImageExtendedFormats**
28	**R16i**	**StorageImageExtendedFormats**
29	**R8i**	**StorageImageExtendedFormats**
30	**Rgba32ui**	**Shader**
31	**Rgba16ui**	**Shader**
32	**Rgba8ui**	**Shader**
33	**R32ui**	**Shader**
34	**Rgb10a2ui**	**StorageImageExtendedFormats**
35	**Rg32ui**	**StorageImageExtendedFormats**
36	**Rg16ui**	**StorageImageExtendedFormats**
37	**Rg8ui**	**StorageImageExtendedFormats**
38	**R16ui**	**StorageImageExtendedFormats**
39	**R8ui**	**StorageImageExtendedFormats**

3.12. Image Channel Order

Image channel order returned by **OpImageQueryOrder**.

Image Channel Order		Required Capability
0	**R**	**Kernel**
1	**A**	**Kernel**
2	**RG**	**Kernel**
3	**RA**	**Kernel**
4	**RGB**	**Kernel**
5	**RGBA**	**Kernel**
6	**BGRA**	**Kernel**
7	**ARGB**	**Kernel**
8	**Intensity**	**Kernel**
9	**Luminance**	**Kernel**
10	**Rx**	**Kernel**
11	**RGx**	**Kernel**
12	**RGBx**	**Kernel**
13	**Depth**	**Kernel**
14	**DepthStencil**	**Kernel**
15	**sRGB**	**Kernel**

	Image Channel Order	Required Capability
16	**sRGBx**	**Kernel**
17	**sRGBA**	**Kernel**
18	**sBGRA**	**Kernel**
19	**ABGR**	**Kernel**

3.13. Image Channel Data Type

Image channel data type returned by **OpImageQueryFormat**.

	Image Channel Data Type	Required Capability
0	**SnormInt8**	**Kernel**
1	**SnormInt16**	**Kernel**
2	**UnormInt8**	**Kernel**
3	**UnormInt16**	**Kernel**
4	**UnormShort565**	**Kernel**
5	**UnormShort555**	**Kernel**
6	**UnormInt101010**	**Kernel**
7	**SignedInt8**	**Kernel**
8	**SignedInt16**	**Kernel**
9	**SignedInt32**	**Kernel**
10	**UnsignedInt8**	**Kernel**
11	**UnsignedInt16**	**Kernel**
12	**UnsignedInt32**	**Kernel**
13	**HalfFloat**	**Kernel**
14	**Float**	**Kernel**
15	**UnormInt24**	**Kernel**
16	**UnormInt101010_2**	**Kernel**

3.14. Image Operands

Additional operands to sampling, or getting texels from, an image. Bits that are set can indicate that another operand follows. If there are multiple following operands indicated, they are ordered: Those indicated by smaller-numbered bits appear first. At least one bit must be set (**None** is invalid).

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by:

- **OpImageSampleImplicitLod**
- **OpImageSampleExplicitLod**
- **OpImageSampleDrefImplicitLod**
- **OpImageSampleDrefExplicitLod**

- **OpImageSampleProjImplicitLod**
- **OpImageSampleProjExplicitLod**
- **OpImageSampleProjDrefImplicitLod**
- **OpImageSampleProjDrefExplicitLod**
- **OpImageFetch**
- **OpImageGather**
- **OpImageDrefGather**
- **OpImageRead**
- **OpImageWrite**
- **OpImageSparseSampleImplicitLod**
- **OpImageSparseSampleExplicitLod**
- **OpImageSparseSampleDrefImplicitLod**
- **OpImageSparseSampleDrefExplicitLod**
- **OpImageSparseSampleProjImplicitLod**
- **OpImageSparseSampleProjExplicitLod**
- **OpImageSparseSampleProjDrefImplicitLod**
- **OpImageSparseSampleProjDrefExplicitLod**
- **OpImageSparseFetch**
- **OpImageSparseGather**
- **OpImageSparseDrefGather**
- **OpImageSparseRead**

Image Operands		Required Capability
0x0	**None**	
0x1	**Bias** A following operand is the bias added to the implicit level of detail. Only valid with implicit-lod instructions. It must be a *floating-point type* scalar. This can only be used with an **OpTypeImage** that has a *Dim* operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	**Shader**
0x2	**Lod** A following operand is the explicit level-of-detail to use. Only valid with explicit-lod instructions. For sampling operations, it must be a *floating-point type* scalar. For queries and fetch operations, it must be an *integer type* scalar. This can only be used with an **OpTypeImage** that has a *Dim* operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	
0x4	**Grad** Two following operands are *dx* followed by *dy*. These are explicit derivatives in the *x* and *y* direction to use in computing level of detail. Each is a scalar or vector containing (*du/dx*[, *dv/dx*] [, *dw/dx*]) and (*du/dy*[, *dv/dy*] [, *dw/dy*]). The number of components of each must equal the number of components in *Coordinate*, minus the *array layer* component, if	

	Image Operands	Required Capability
	present. Only valid with explicit-lod instructions. They must be a scalar or vector of *floating-point type*. This can only be used with an **OpTypeImage** that has an *MS* operand of 0. It is invalid to set both the **Lod** and **Grad** bits.	
0x8	**ConstOffset** A following operand is added to (*u*, *v*, *w*) before texel lookup. It must be an *<id>* of an integer-based *constant instruction* of scalar or vector type. It is a compile-time error if these fall outside a target-dependent allowed range. The number of components must equal the number of components in *Coordinate*, minus the *array layer* component, if present.	
0x10	**Offset** A following operand is added to (*u*, *v*, *w*) before texel lookup. It must be a scalar or vector of *integer type*. It is a compile-time error if these fall outside a target-dependent allowed range. The number of components must equal the number of components in *Coordinate*, minus the *array layer* component, if present.	**ImageGatherExtended**
0x20	**ConstOffsets** A following operand is *Offsets*. *Offsets* must be an *<id>* of a *constant instruction* making an array of size four of vectors of two integer components. Each gathered texel is identified by adding one of these array elements to the (*u*, *v*) sampled location. It is a compile-time error if this falls outside a target-dependent allowed range. Only valid with **OpImageGather** or **OpImageDrefGather**.	
0x40	**Sample** A following operand is the sample number of the sample to use. Only valid with **OpImageFetch**, **OpImageRead**, and **OpImageWrite**. It is invalid to have a **Sample** operand if the underlying **OpTypeImage** has *MS* of 0. It must be an *integer type* scalar.	
0x80	**MinLod** A following operand is the minimum level-of-detail to use when accessing the image. Only valid with **Implicit** instructions and **Grad** instructions. It must be a *floating-point type* scalar. This can only be used with an **OpTypeImage** that has a *Dim* operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	**MinLod**

3.15. FP Fast Math Mode

Enables fast math operations which are otherwise unsafe.

- Only valid on **OpFAdd**, **OpFSub**, **OpFMul**, **OpFDiv**, **OpFRem**, and **OpFMod** instructions.

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

FP Fast Math Mode		Required Capability
0x0	**None**	
0x1	**NotNaN** Assume parameters and result are not NaN.	**Kernel**
0x2	**NotInf** Assume parameters and result are not +/- Inf.	**Kernel**
0x4	**NSZ** Treat the sign of a zero parameter or result as insignificant.	**Kernel**
0x8	**AllowRecip** Allow the usage of reciprocal rather than perform a division.	**Kernel**
0x10	**Fast** Allow algebraic transformations according to real-number associative and distributive algebra. This flag implies all the others.	**Kernel**

3.16. FP Rounding Mode

Associate a rounding mode to a floating-point conversion instruction.

By default

- Conversions from floating-point to integer types use the round-toward-zero rounding mode.
- Conversions to floating-point types use the round-to-nearest-even rounding mode.

FP Rounding Mode		Required Capability
0	**RTE** Round to nearest even.	**Kernel**
1	**RTZ** Round towards zero.	**Kernel**
2	**RTP** Round towards positive infinity.	**Kernel**
3	**RTN** Round towards negative infinity.	**Kernel**

3.17. Linkage Type

Associate a linkage type to functions or global variables. See linkage.

Linkage Type		Required Capability
0	**Export** Accessible by other modules as well.	**Linkage**

	Linkage Type	Required Capability
1	**Import** A declaration of a global variable or a function that exists in another module.	**Linkage**

3.18. Access Qualifier

Defines the access permissions.

Used by **OpTypeImage** and **OpTypePipe**.

	Access Qualifier	Required Capability
0	**ReadOnly** A read-only object.	**Kernel**
1	**WriteOnly** A write-only object.	**Kernel**
2	**ReadWrite** A readable and writable object.	**Kernel**

3.19. Function Parameter Attribute

Adds additional information to the return type and to each parameter of a function.

	Function Parameter Attribute	Required Capability
0	**Zext** Value should be zero extended if needed.	**Kernel**
1	**Sext** Value should be sign extended if needed.	**Kernel**
2	**ByVal** This indicates that the pointer parameter should really be passed by value to the function. Only valid for pointer parameters (not for ret value).	**Kernel**
3	**Sret** Indicates that the pointer parameter specifies the address of a structure that is the return value of the function in the source program. Only applicable to the first parameter which must be a pointer parameters.	**Kernel**
4	**NoAlias** Indicates that the memory pointed to by a pointer parameter is not accessed via pointer values which are not derived from this pointer parameter. Only valid for pointer parameters. Not valid on return values.	**Kernel**
5	**NoCapture** The callee does not make a copy of the pointer parameter into a location that is accessible after returning from the callee. Only valid for pointer parameters. Not valid on return values.	**Kernel**

	Function Parameter Attribute	Required Capability
6	**NoWrite** Can only read the memory pointed to by a pointer parameter. Only valid for pointer parameters. Not valid on return values.	**Kernel**
7	**NoReadWrite** Cannot dereference the memory pointed to by a pointer parameter. Only valid for pointer parameters. Not valid on return values.	**Kernel**

3.20. Decoration

Used by **OpDecorate** and **OpMemberDecorate**.

	Decoration	Required Capability	Extra Operands
0	**RelaxedPrecision** Allow reduced precision operations. To be used as described in Relaxed Precision.	**Shader**	
1	**SpecId** Apply to a scalar specialization constant. Forms the API linkage for setting a specialized value. See specialization.	**Shader**, **Kernel**	*Literal Number* *Specialization Constant ID*
2	**Block** Apply to a structure type to establish it is a non-SSBO-like shader-interface block.	**Shader**	
3	**BufferBlock** Apply to a structure type to establish it is an SSBO-like shader-interface block.	**Shader**	
4	**RowMajor** Applies only to a member of a structure type. Only valid on a matrix or array whose most basic element is a matrix. Indicates that components within a row are contiguous in memory.	**Matrix**	
5	**ColMajor** Applies only to a member of a structure type. Only valid on a matrix or array whose most basic element is a matrix. Indicates that components within a column are contiguous in memory.	**Matrix**	
6	**ArrayStride** Apply to an array type to specify the stride, in bytes, of the array's elements. Must not be applied to anything other than an array type.	**Shader**	*Literal Number* *Array Stride*
7	**MatrixStride** Applies only to a member of a structure type. Only valid on a matrix or array whose most basic element is a matrix. Specifies the stride of rows in a **RowMajor**-decorated matrix, or columns in a **ColMajor**-decorated matrix.	**Matrix**	*Literal Number* *Matrix Stride*
8	**GLSLShared** Apply to a structure type to get GLSL **shared** memory layout.	**Shader**	

	Decoration	Required Capability	Extra Operands
9	**GLSLPacked** Apply to a structure type to get GLSL **packed** memory layout.	**Shader**	
10	**CPacked** Apply to a structure type, to marks it as "packed", indicating that the alignment of the structure is one and that there is no padding between structure members.	**Kernel**	
11	**BuiltIn** Apply to an object or a member of a structure type. Indicates which built-in variable the entity represents. See BuiltIn for more information.		*BuiltIn*
13	**NoPerspective** Apply to an object or a member of a structure type. Indicates that linear, non-perspective correct, interpolation must be used. Only valid for the **Input** and **Output** Storage Classes.	**Shader**	
14	**Flat** Apply to an object or a member of a structure type. Indicates no interpolation will be done. The non-interpolated value will come from a vertex, as described in the API specification. Only valid for the **Input** and **Output** Storage Classes.	**Shader**	
15	**Patch** Apply to an object or a member of a structure type. Indicates a tessellation patch. Only valid for the **Input** and **Output** Storage Classes. Invalid to use on objects or types referenced by non-tessellation Execution Models.	**Tessellation**	
16	**Centroid** Apply to an object or a member of a structure type. When used with multi-sampling rasterization, allows a single interpolation location for an entire pixel. The interpolation location must lie in both the pixel and in the primitive being rasterized. Only valid for the **Input** and **Output** Storage Classes.	**Shader**	
17	**Sample** Apply to an object or a member of a structure type. When used with multi-sampling rasterization, requires per-sample interpolation. The interpolation locations must be the locations of the samples lying in both the pixel and in the primitive being rasterized. Only valid for the **Input** and **Output** Storage Classes.	**SampleRateShading**	
18	**Invariant** Apply to a variable, to indicate expressions computing its value be done invariant with respect to other modules computing the same expressions.	**Shader**	
19	**Restrict** Apply to a variable, to indicate the compiler may compile as if there is no aliasing. See the Aliasing section for more detail.		

Decoration	Required Capability	Extra Operands
20 **Aliased** Apply to a variable, to indicate the compiler is to generate accesses to the variable that work correctly in the presence of aliasing. See the Aliasing section for more detail.		
21 **Volatile** Apply to an object or a member of a structure type. Can only be used for objects declared as storage images (see **OpTypeImage**) or in the **Uniform** Storage Class with the **BufferBlock** Decoration. This indicates the memory holding the variable is volatile memory. Accesses to volatile memory cannot be eliminated, duplicated, or combined with other accesses. The variable cannot be in the **Function** Storage Class.		
22 **Constant** Indicates that a global variable is constant and will **never** be modified. Only allowed on global variables.	**Kernel**	
23 **Coherent** Apply to an object or a member of a structure type. Can only be used for objects declared as storage images (see **OpTypeImage**) or in the **Uniform** Storage Class with the **BufferBlock** Decoration. This indicates the memory backing the object is coherent.		
24 **NonWritable** Apply to an object or a member of a structure type. Can only be used for objects declared as storage images (see **OpTypeImage**) or in the **Uniform** Storage Class with the **BufferBlock** Decoration. This indicates the memory holding the variable is not writable, and that this module does not write to it.		
25 **NonReadable** Apply to an object or a member of a structure type. Can only be used for objects declared as storage images (see **OpTypeImage**) or in the **Uniform** Storage Class with the **BufferBlock** Decoration. This indicates the memory holding the variable is not readable, and that this module does not read from it.		
26 **Uniform** Apply to an object or a member of a structure type. Asserts that the value backing the decorated *<id>* is dynamically uniform, hence the consumer is allowed to assume this is the case.	**Shader**	
28 **SaturatedConversion** Indicates that a conversion to an integer type which is outside the representable range of *Result Type* will be clamped to the nearest representable value of *Result Type*. *NaN* will be converted to *0*. This decoration can only be applied to conversion instructions to integer types, not including the **OpSatConvertUToS** and **OpSatConvertSToU**	**Kernel**	

	Decoration	Required Capability	Extra Operands
	instructions.		
29	**Stream** Apply to an object or a member of a structure type. Indicates the stream number to put an output on. Only valid for the **Output** Storage Class and the **Geometry** Execution Model.	**GeometryStreams**	*Literal Number* *Stream Number*
30	**Location** Apply to a variable or a structure-type member. Forms the main linkage for Storage Class **Input** and **Output** variables: - between the API and vertex-stage inputs, - between consecutive programmable stages, or - between fragment-stage outputs and the API. Also can tag variables or structure-type members in the **UniformConstant** Storage Class for linkage with the API. Only valid for the **Input**, **Output**, and **UniformConstant** Storage Classes.	**Shader**	*Literal Number* *Location*
31	**Component** Apply to an object or a member of a structure type. Indicates which component within a **Location** will be taken by the decorated entity. Only valid for the **Input** and **Output** Storage Classes.	**Shader**	*Literal Number* *Component*
32	**Index** Apply to a variable to identify a blend equation input index, used as described in the API specification. Only valid for the **Output** Storage Class and the **Fragment** Execution Model.	**Shader**	*Literal Number* *Index*
33	**Binding** Apply to a variable. Part of the main linkage between the API and SPIR-V modules for memory buffers, images, etc. See the API specification for more information.	**Shader**	*Literal Number* *Binding Point*
34	**DescriptorSet** Apply to a variable. Part of the main linkage between the API and SPIR-V modules for memory buffers, images, etc. See the API specification for more information.	**Shader**	*Literal Number* *Descriptor Set*
35	**Offset** Apply to a structure-type member. This gives the byte offset of the member relative to the beginning of the structure. Can be used, for example, by both uniform and transform-feedback buffers. It must not cause any overlap of the structure's members, or overflow of a transform-feedback buffer's **XfbStride**.	**Shader**	*Literal Number* *Byte Offset*
36	**XfbBuffer** Apply to an object or a member of a structure type. Indicates which transform-feedback buffer an output is written to. Only valid for the **Output** Storage Classes of *vertex processing* Execution Models.	**TransformFeedback**	*Literal Number* *XFB Buffer Number*
37	**XfbStride** Apply to anything **XfbBuffer** is applied to. Specifies the stride, in bytes, of transform-	**TransformFeedback**	*Literal Number* *XFB Stride*

	Decoration	Required Capability	Extra Operands	
	feedback buffer vertices. If the transform-feedback buffer is capturing any double-precision components, the stride must be a multiple of 8, otherwise it must be a multiple of 4.			
38	**FuncParamAttr** Indicates a function return value or parameter attribute.	**Kernel**	*Function Parameter Attribute* *Function Parameter Attribute*	
39	**FPRoundingMode** Indicates a floating-point rounding mode.	**Kernel**	*FP Rounding Mode* *Floating-Point Rounding Mode*	
40	**FPFastMathMode** Indicates a floating-point fast math flag.	**Kernel**	*FP Fast Math Mode* *Fast-Math Mode*	
41	**LinkageAttributes** Associate linkage attributes to values. Only valid on **OpFunction** or global (module scope) **OpVariable**. See linkage.	**Linkage**	*Literal String* *Name*	*Linkage Type* *Linkage Type*
42	**NoContraction** Apply to an arithmetic instruction to indicate the operation cannot be combined with another instruction to form a single operation. For example, if applied to an **OpFMul**, that multiply can't be combined with an addition to yield a fused multiply-add operation. Furthermore, such operations are not allowed to reassociate; e.g., add(a + add(b+c)) cannot be transformed to add(add(a+b) + c).	**Shader**		
43	**InputAttachmentIndex** Apply to a variable to provide an input-target index (as described in the API specification). Only valid in the **Fragment** Execution Model and for variables of type **OpTypeImage** with a *Dim* operand of **SubpassData**.	**InputAttachment**	*Literal Number* *Attachment Index*	
44	**Alignment** Apply to a pointer. This declares a known minimum alignment the pointer has.	**Kernel**	*Literal Number* *Alignment*	
45	**MaxByteOffset** Apply to a pointer. This declares a known maximum byte offset this pointer will be incremented by from the point of the decoration. This is a guaranteed upper bound when applied to **OpFunctionParameter**.	**Addresses**	*Literal Number* *Max Byte Offset*	

3.21. BuiltIn

Used when **Decoration** is **BuiltIn**. Apply to either

- the result *<id>* of the variable declaration of the built-in variable, or
- a structure-type member, if the built-in is a member of a structure.

As stated per entry below, these have additional semantics and constraints described by the client API.

BuiltIn		Required Capability
0	**Position** Output vertex position from a vertex processing Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
1	**PointSize** Output point size from a vertex processing Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
3	**ClipDistance** Array of clip distances. See Vulkan or OpenGL API specifications for more detail.	**ClipDistance**
4	**CullDistance** Array of clip distances. See Vulkan or OpenGL API specifications for more detail.	**CullDistance**
5	**VertexId** Input vertex ID to a **Vertex** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
6	**InstanceId** Input instance ID to a **Vertex** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
7	**PrimitiveId** Primitive ID in a **Geometry** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Geometry**, **Tessellation**
8	**InvocationId** Invocation ID, input to **Geometry** and **TessellationControl** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Geometry**, **Tessellation**
9	**Layer** Layer output by a **Geometry** Execution Model, input to a **Fragment** Execution Model, for multi-layer framebuffer. See Vulkan or OpenGL API specifications for more detail.	**Geometry**
10	**ViewportIndex** Viewport Index output by a **Geometry** stage, input to a **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**MultiViewport**
11	**TessLevelOuter** Output patch outer levels in a **TessellationControl** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Tessellation**
12	**TessLevelInner** Output patch inner levels in a **TessellationControl** Execution Model. See Vulkan or OpenGL API specifications for	**Tessellation**

	BuiltIn	Required Capability
	more detail.	
13	**TessCoord** Input vertex position in **TessellationEvaluation** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Tessellation**
14	**PatchVertices** Input patch vertex count in a tessellation Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Tessellation**
15	**FragCoord** Coordinates *(x, y, z, 1/w)* of the current fragment, input to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
16	**PointCoord** Coordinates within a *point*, input to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
17	**FrontFacing** Face direction, input to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
18	**SampleId** Input sample number to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**SampleRateShading**
19	**SamplePosition** Input sample position to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**SampleRateShading**
20	**SampleMask** Input or output sample mask to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**SampleRateShading**
22	**FragDepth** Output fragment depth from the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
23	**HelperInvocation** Input whether a helper invocation, to the **Fragment** Execution Model. See Vulkan or OpenGL API specifications for more detail.	**Shader**
24	**NumWorkgroups** Number of workgroups in **GLCompute** or **Kernel** Execution Models. See OpenCL, Vulkan, or OpenGL API specifications for more detail.	
25	**WorkgroupSize** Work-group size in **GLCompute** or **Kernel** Execution Models. See OpenCL, Vulkan, or OpenGL API specifications for more detail.	

3.22. Selection Control

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by **OpSelectionMerge**.

Selection Control	
0x0	**None**
0x1	**Flatten** Strong request, to the extent possible, to remove the control flow for this selection.
0x2	**DontFlatten** Strong request, to the extent possible, to keep this selection as control flow.

3.23. Loop Control

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by **OpLoopMerge**.

Loop Control	
0x0	**None**
0x1	**Unroll** Strong request, to the extent possible, to unroll or unwind this loop.
0x2	**DontUnroll** Strong request, to the extent possible, to keep this loop as a loop, without unrolling.
0x4	**DependencyInfinite** Guarantees that there are no dependencies between loop iterations.
0x8	**DependencyLength** Guarantees that there are no dependencies between a number of loop iterations,specified as a subsequent literal-number operand to the instruction.

3.24. Function Control

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by **OpFunction**.

Function Control	
0x0	**None**
0x1	**Inline** Strong request, to the extent possible, to inline the function.
0x2	**DontInline** Strong request, to the extent possible, to not inline the function.

	Function Control
0x4	**Pure** Compiler can assume this function has no side effect, but might read global memory or read through dereferenced function parameters. Always computes the same result for the same argument values.
0x8	**Const** Compiler can assume this function has no side effects, and will not access global memory or dereference function parameters. Always computes the same result for the same argument values.

3.25. Memory Semantics <id>

Must be an *<id>* of a 32-bit integer scalar that contains a mask. The rest of this description is about that mask.

Memory semantics define memory-order constraints, and on what storage classes those constraints apply to. The memory order constrains the allowed orders in which memory operations in this invocation can made visible to another invocation. The storage classes specify to which subsets of memory these constraints are to be applied. Storage classes not selected are not being constrained.

Despite being a mask and allowing multiple bits to be combined, at most one of the first four (low-order) bits can be set. Requesting both **Acquire** and **Release** semantics is done by setting the **AcquireRelease** bit, not by setting two bits.

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by:

- **OpControlBarrier**
- **OpMemoryBarrier**
- **OpAtomicLoad**
- **OpAtomicStore**
- **OpAtomicExchange**
- **OpAtomicCompareExchange**
- **OpAtomicCompareExchangeWeak**
- **OpAtomicIIncrement**
- **OpAtomicIDecrement**
- **OpAtomicIAdd**
- **OpAtomicISub**
- **OpAtomicSMin**
- **OpAtomicUMin**
- **OpAtomicSMax**
- **OpAtomicUMax**
- **OpAtomicAnd**
- **OpAtomicOr**
- **OpAtomicXor**
- **OpAtomicFlagTestAndSet**
- **OpAtomicFlagClear**
- **OpMemoryNamedBarrier**

Memory Semantics		Required Capability
0x0	**None (Relaxed)**	
0x2	**Acquire** All memory operations provided in program order after this memory operation will execute after this memory operation.	
0x4	**Release** All memory operations provided in program order before this memory operation will execute before this memory operation.	
0x8	**AcquireRelease** Has the properties of both **Acquire** and **Release** semantics. It is used for read-modify-write operations.	
0x10	**SequentiallyConsistent** All observers will see this memory access in the same order with respect to other sequentially-consistent memory accesses from this invocation.	
0x40	**UniformMemory** Apply the memory-ordering constraints to **Uniform** Storage Class memory.	**Shader**
0x80	**SubgroupMemory** Apply the memory-ordering constraints to subgroup memory.	
0x100	**WorkgroupMemory** Apply the memory-ordering constraints to **Workgroup** Storage Class memory.	
0x200	**CrossWorkgroupMemory** Apply the memory-ordering constraints to **CrossWorkgroup** Storage Class memory.	
0x400	**AtomicCounterMemory** Apply the memory-ordering constraints to **AtomicCounter** Storage Class memory.	**AtomicStorage**
0x800	**ImageMemory** Apply the memory-ordering constraints to image contents (types declared by **OpTypeImage**), or to accesses done through pointers to the **Image** Storage Class.	

3.26. Memory Access

Memory access semantics.

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Used by:

- **OpLoad**
- **OpStore**

- **OpCopyMemory**
- **OpCopyMemorySized**

Memory Access	
0x0	**None**
0x1	**Volatile** This access cannot be eliminated, duplicated, or combined with other accesses.
0x2	**Aligned** This access has a known alignment, provided as a literal in the next operand.
0x4	**Nontemporal** Hints that the accessed address is not likely to be accessed again in the near future.

3.27. Scope <id>

Must be an *<id>* of a 32-bit integer scalar that contains a mask. The rest of this description is about that mask.

The execution scope or memory scope of an operation. When used as a memory scope, it specifies the distance of synchronization from the current invocation. When used as an execution scope, it specifies the set of executing invocations taking part in the operation. Used by:

- **OpControlBarrier**
- **OpMemoryBarrier**
- **OpAtomicLoad**
- **OpAtomicStore**
- **OpAtomicExchange**
- **OpAtomicCompareExchange**
- **OpAtomicCompareExchangeWeak**
- **OpAtomicIIncrement**
- **OpAtomicIDecrement**
- **OpAtomicIAdd**
- **OpAtomicISub**
- **OpAtomicSMin**
- **OpAtomicUMin**
- **OpAtomicSMax**
- **OpAtomicUMax**
- **OpAtomicAnd**
- **OpAtomicOr**
- **OpAtomicXor**
- **OpGroupAsyncCopy**
- **OpGroupWaitEvents**
- **OpGroupAll**
- **OpGroupAny**
- **OpGroupBroadcast**
- **OpGroupIAdd**

- **OpGroupFAdd**
- **OpGroupFMin**
- **OpGroupUMin**
- **OpGroupSMin**
- **OpGroupFMax**
- **OpGroupUMax**
- **OpGroupSMax**
- **OpGroupReserveReadPipePackets**
- **OpGroupReserveWritePipePackets**
- **OpGroupCommitReadPipe**
- **OpGroupCommitWritePipe**
- **OpAtomicFlagTestAndSet**
- **OpAtomicFlagClear**
- **OpMemoryNamedBarrier**

Scope	
0	**CrossDevice** Scope crosses multiple devices.
1	**Device** Scope is the current device.
2	**Workgroup** Scope is the current workgroup.
3	**Subgroup** Scope is the current subgroup.
4	**Invocation** Scope is the current Invocation.

3.28. Group Operation

Defines the class of workgroup or subgroup operation. Used by:

- **OpGroupIAdd**
- **OpGroupFAdd**
- **OpGroupFMin**
- **OpGroupUMin**
- **OpGroupSMin**
- **OpGroupFMax**
- **OpGroupUMax**
- **OpGroupSMax**

Group Operation		Required Capability
0	**Reduce** A reduction operation for all values of a specific value X specified by invocations within a workgroup.	**Kernel**
1	**InclusiveScan** A binary operation with an identity *I* and *n* (where *n* is the size of the workgroup) elements[a_0, a_1, … a_{n-1}] resulting in [a_0, (a_0	**Kernel**

Group Operation		Required Capability
	op a_1), …(a_0 op a_1 op … op a_{n-1})]	
2	**ExclusiveScan** A binary operation with an identity *I* and *n* (where *n* is the size of the workgroup) elements[a_0, a_1, … a_{n-1}] resulting in [*I*, a_0, (a_0 op a_1), … (a_0 op a_1 op … op a_{n-2})].	**Kernel**

3.29. Kernel Enqueue Flags

Specify when the child kernel begins execution.

Note: Implementations are not required to honor this flag. Implementations may not schedule kernel launch earlier than the point specified by this flag, however. Used by **OpEnqueueKernel**.

Kernel Enqueue Flags		Required Capability
0	**NoWait** Indicates that the enqueued kernels do not need to wait for the parent kernel to finish execution before they begin execution.	**Kernel**
1	**WaitKernel** Indicates that all work-items of the parent kernel must finish executing and all immediate side effects committed before the enqueued child kernel may begin execution. **Note:** Immediate meaning not side effects resulting from child kernels. The side effects would include stores to global memory and pipe reads and writes.	**Kernel**
2	**WaitWorkGroup** Indicates that the enqueued kernels wait only for the workgroup that enqueued the kernels to finish before they begin execution. **Note:** This acts as a memory synchronization point between work-items in a work-group and child kernels enqueued by work-items in the work-group.	**Kernel**

3.30. Kernel Profiling Info

Specify the profiling information to be queried. Used by **OpCaptureEventProfilingInfo**.

This value is a mask; it can be formed by combining the bits from multiple rows in the table below.

Kernel Profiling Info		Required Capability
0x0	**None**	
0x1	**CmdExecTime** Indicates that the profiling info queried is the execution time.	**Kernel**

3.31. Capability

Capabilities a module can declare it uses. All used capabilities must be declared, either directly or through a dependency: all capabilities that a declared capability depends on are automatically implied.

The **Depends On** column lists the dependencies for each capability. These are the ones implicitly declared. It is not necessary (but allowed) to declare a dependency for a declared capability.

See the capabilities section for more detail. Used by **OpCapability**.

Capability		Depends On
0	**Matrix** Uses **OpTypeMatrix**.	
1	**Shader** Uses **Vertex**, **Fragment**, or **GLCompute** Execution Models.	**Matrix**
2	**Geometry** Uses the **Geometry** Execution Model.	**Shader**
3	**Tessellation** Uses the **TessellationControl** or **TessellationEvaluation** Execution Models.	**Shader**
4	**Addresses** Uses physical addressing, non-logical addressing modes.	
5	**Linkage** Uses partially linked modules and libraries.	
6	**Kernel** Uses the **Kernel** Execution Model.	
7	**Vector16** Uses **OpTypeVector** to declare 8 component or 16 component vectors.	**Kernel**
8	**Float16Buffer** Uses pointers to 16-bit floating-point data types (but doesn't use 16-bit **OpTypeFloat** as a *Result <id>*).	**Kernel**
9	**Float16** Uses **OpTypeFloat** to declare the 16-bit floating-point type.	
10	**Float64** Uses **OpTypeFloat** to declare the 64-bit floating-point type.	
11	**Int64** Uses **OpTypeInt** to declare 64-bit integer types.	
12	**Int64Atomics** Uses atomic instructions on 64-bit integer types.	**Int64**
13	**ImageBasic** Uses **OpTypeImage** or **OpTypeSampler** in a **Kernel**.	**Kernel**

	Capability	Depends On
14	**ImageReadWrite** Uses **OpTypeImage** with the **ReadWrite** *access qualifier*.	**ImageBasic**
15	**ImageMipmap** Uses non-zero **Lod** Image Operands.	**ImageBasic**
17	**Pipes** Uses **OpTypePipe**, **OpTypeReserveId** or *pipe* instructions.	**Kernel**
18	**Groups** Uses *group* instructions.	
19	**DeviceEnqueue** Uses **OpTypeQueue**, **OpTypeDeviceEvent**, and *device side enqueue* instructions.	**Kernel**
20	**LiteralSampler** Samplers are made from literals within the module. See **OpConstantSampler**.	**Kernel**
21	**AtomicStorage** Uses the **AtomicCounter** Storage Class.	**Shader**
22	**Int16** Uses **OpTypeInt** to declare 16-bit integer types.	
23	**TessellationPointSize** Tessellation stage exports point size.	**Tessellation**
24	**GeometryPointSize** Geometry stage exports point size	**Geometry**
25	**ImageGatherExtended** Uses texture gather with non-constant or independent offsets	**Shader**
27	**StorageImageMultisample** Uses multi-sample images for non-sampled images.	**Shader**
28	**UniformBufferArrayDynamicIndexing** **Block**-decorated arrays in uniform storage classes use dynamically uniform indexing.	**Shader**
29	**SampledImageArrayDynamicIndexing** Arrays of sampled images use dynamically uniform indexing.	**Shader**
30	**StorageBufferArrayDynamicIndexing** **BufferBlock**-decorated arrays in uniform storage classes use dynamically uniform indexing.	**Shader**
31	**StorageImageArrayDynamicIndexing** Arrays of non-sampled images are accessed with dynamically uniform indexing.	**Shader**
32	**ClipDistance** Uses the **ClipDistance** BuiltIn.	**Shader**

	Capability	Depends On
33	**CullDistance** Uses the **CullDistance** BuiltIn.	**Shader**
34	**ImageCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in OpTypeImage, without a sampler.	**SampledCubeArray**
35	**SampleRateShading** Uses per-sample rate shading.	**Shader**
36	**ImageRect** Uses the **Rect** Dim without a sampler.	**SampledRect**
37	**SampledRect** Uses the **Rect** Dim with a sampler.	**Shader**
38	**GenericPointer** Uses the **Generic** Storage Class.	**Addresses**
39	**Int8** Uses **OpTypeInt** to declare 8-bit integer types.	**Kernel**
40	**InputAttachment** Uses the **SubpassData** Dim.	**Shader**
41	**SparseResidency** Uses **OpImageSparse…** instructions.	**Shader**
42	**MinLod** Uses the **MinLod** Image Operand.	**Shader**
43	**Sampled1D** Uses the **1D** Dim with a sampler.	
44	**Image1D** Uses the **1D** Dim without a sampler.	**Sampled1D**
45	**SampledCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in OpTypeImage, with a sampler.	**Shader**
46	**SampledBuffer** Uses the **Buffer** Dim without a sampler.	
47	**ImageBuffer** Uses the **Buffer** Dim without a sampler.	**SampledBuffer**
48	**ImageMSArray** An *MS* operand in OpTypeImage indicates multisampled, used without a sampler.	**Shader**
49	**StorageImageExtendedFormats** One of a large set of more advanced image formats are used, namely one of those in the Image Format table listed as requiring this capability.	**Shader**
50	**ImageQuery** The sizes, number of samples, or lod, etc. are queried.	**Shader**

	Capability	Depends On
51	**DerivativeControl** Uses fine or coarse-grained derivatives, e.g., **OpDPdxFine**.	**Shader**
52	**InterpolationFunction** Uses one of the **InterpolateAtCentroid**, **InterpolateAtSample**, or **InterpolateAtOffset** GLSL.std.450 extended instructions.	**Shader**
53	**TransformFeedback** Uses the **Xfb** Execution Mode.	**Shader**
54	**GeometryStreams** Uses multiple numbered streams for geometry-stage output.	**Geometry**
55	**StorageImageReadWithoutFormat** **OpImageRead** can use the **Unknown** Image Format for	**Shader**
56	**StorageImageWriteWithoutFormat** **OpImageWrite** can use the **Unknown** Image Format.	**Shader**
57	**MultiViewport** Multiple viewports are used.	**Geometry**
58	**SubgroupDispatch** Uses subgroup dispatch instructions.	**DeviceEnqueue**
59	**NamedBarrier** Uses **OpTypeNamedBarrier**.	**Kernel**
60	**PipeStorage** Uses **OpTypePipeStorage**.	**Pipes**

3.32. Instructions

Form for each instruction:

Opcode Name Instruction description. *Word Count* is the high-order 16 bits of word 0 of the instruction, holding its total *WordCount*. If the instruction takes a variable number of operands, *Word Count* will also say "+ variable", after stating the minimum size of the instruction. *Opcode* is the low-order 16 bits of word 0 of the instruction, holding its opcode enumerant. *Results*, when present, are any *Result <id>* or *Result Type* created by the instruction. Each one is always 32 bits. *Operands*, when present, are any literals, other instruction's *Result <id>*, etc., consumed by the instruction. Each one is always 32 bits.			Capability **Required Capabilities** (when needed)
Word Count	*Opcode*	*Results*	*Operands*

3.32.1. Miscellaneous Instructions

OpNop This has no semantic impact and can safely be removed from a module.	
1	0

OpUndef Make an intermediate object whose value is undefined. *Result Type* is the type of object to make. Each consumption of *Result <id>* yields an arbitrary, possibly different bit pattern.			
3	1	*<id>* *Result Type*	*Result <id>*

OpSizeOf Computes the run-time size of the type pointed to by *Pointer* *Result Type* must be a 32-bit *integer type* scalar. *Pointer* must point to a concrete type.				Capability: **Addresses**
4	321	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

3.32.2. Debug Instructions

OpSourceContinued

Continue specifying the *Source* text from the previous instruction. This has no semantic impact and can safely be removed from a module.

Continued Source is a continuation of the source text in the previous *Source*.

The previous instruction must be an **OpSource** or an **OpSourceContinued** instruction. As is true for all literal strings, the previous instruction's string was nul terminated. That terminating 0 word from the previous instruction is not part of the source text; the first character of *Continued Source* logically immediately follows the last character of *Source* before its nul.

2 + variable	2	*Literal String* *Continued Source*

OpSource

Document what source language and text this module was translated from. This has no semantic impact and can safely be removed from a module.

Version is the version of the source language. This literal operand is limited to a single word.

File is an **OpString** instruction and is the source-level file name.

Source is the text of the source-level file.

Each client API describes what form the *Version* operand takes, per source language.

3 + variable	3	*Source Language*	*Literal Number* *Version*	Optional *<id>* *File*	Optional *Literal String* *Source*

OpSourceExtension

Document an extension to the source language. This has no semantic impact and can safely be removed from a module.

Extension is a string describing a source-language extension. Its form is dependent on the how the source language describes extensions.

2 + variable	4	*Literal String* *Extension*

OpName

Assign a name string to another instruction's *Result <id>*. This has no semantic impact and can safely be removed from a module.

Target is the *Result <id>* to assign a name to. It can be the *Result <id>* of any other instruction; a variable, function, type, intermediate result, etc.

Name is the string to assign.

3 + variable	5	*<id>* *Target*	*Literal String* *Name*

OpMemberName

Assign a name string to a member of a structure type. This has no semantic impact and can safely be removed from a module.

Type is the *<id>* from an **OpTypeStruct** instruction.

Member is the number of the member to assign in the structure. The first member is member 0, the next is member 1, … This literal operand is limited to a single word.

Name is the string to assign to the member.

4 + variable	6	*<id>* *Type*	*Literal Number* *Member*	*Literal String* *Name*

OpString

Assign a *Result <id>* to a string for use by other debug instructions (see **OpLine** and **OpSource**). This has no semantic impact and can safely be removed from a module. (Removal also requires removal of all instructions referencing *Result <id>*.)

String is the literal string being assigned a *Result <id>*.

3 + variable	7	*Result <id>*	*Literal String* *String*

OpLine

Add source-level location information. This has no semantic impact and can safely be removed from a module.

This location 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 **OpLine** instruction, or the next **OpNoLine** instruction.

File must be an **OpString** instruction and is the source-level file name.

Line is the source-level line number. This literal operand is limited to a single word.

Column is the source-level column number. This literal operand is limited to a single word.

OpLine can generally immediately precede other instructions, with the following exceptions:

- it may not be used until after the annotation instructions,
(see the Logical Layout section)

- cannot be the last instruction in a block, which is defined to end with a branch instruction

- if a branch merge instruction is used, the last **OpLine** in the block must be before its merge instruction

4	8	*<id>* *File*	*Literal Number* *Line*	*Literal Number* *Column*

OpNoLine

Discontinue any source-level location information that might be active from a previous **OpLine** instruction. This has no semantic impact and can safely be removed from a module.

This instruction can only appear after the annotation instructions (see the Logical Layout section). It cannot be the last instruction in a block, or the second-to-last instruction if the block has a merge instruction. There is not a requirement that there is a preceding **OpLine** instruction.

1	317

OpModuleProcessed

Document a process that was applied to a module. This has no semantic impact and can safely be removed from a module.

Process is a string describing a process and/or tool (processor) that did the processing. Its form is dependent on the processor.

2 + variable	330	*Literal String* *Process*

3.32.3. Annotation Instructions

OpDecorate

Add a Decoration to another *<id>*.

Target is the *<id>* to decorate. It can potentially be any *<id>* that is a forward reference. A set of decorations can be grouped together by having multiple **OpDecorate** instructions target the same **OpDecorationGroup** instruction.

3 + variable	71	*<id>* *Target*	*Decoration*	*Literal, Literal, …* See *Decoration*.

OpMemberDecorate

Add a Decoration to a member of a structure type.

Structure type is the *<id>* of a type from **OpTypeStruct**.

Member is the number of the member to decorate in the type. The first member is member 0, the next is member 1, …

4 + variable	72	*<id>* *Structure Type*	*Literal Number* *Member*	*Decoration*	*Literal, Literal, …* See *Decoration*.

OpDecorationGroup

A collector for Decorations from **OpDecorate** instructions. All such **OpDecorate** instructions targeting this **OpDecorationGroup** instruction must precede it. Subsequent **OpGroupDecorate** and **OpGroupMemberDecorate** instructions consume this instruction's *Result <id>* to apply multiple decorations to multiple targets.

2	73	*Result <id>*

OpGroupDecorate

Add a group of Decorations to another *<id>*.

Decoration Group is the *<id>* of an **OpDecorationGroup** instruction.

Targets is a list of *<id>s* to decorate with the groups of decorations.

2 + variable	74	*<id>* *Decoration Group*	*<id>, <id>, …* *Targets*

OpGroupMemberDecorate

Add a group of Decorations to members of structure types.

Decoration Group is the *<id>* of an **OpDecorationGroup** instruction.

Targets is a list of (*<id>*, *Member*) pairs to decorate with the groups of decorations. Each *<id>* in the pair must be a target structure type, and the associated *Member* is the number of the member to decorate in the type. The first member is member 0, the next is member 1, …

2 + variable	75	*<id>* *Decoration Group*	*<id>, literal,* *<id>, literal,* *…* *Targets*

3.32.4. Extension Instructions

OpExtension

Declare use of an extension to SPIR-V. This allows validation of additional instructions, tokens, semantics, etc.

Name is the extension's name string.

2 + variable	10	*Literal String* *Name*

OpExtInstImport

Import an extended set of instructions. It can be later referenced by the *Result <id>*.

Name is the extended instruction-set's name string. There must be an external specification defining the semantics for this extended instruction set.

See Extended Instruction Sets for more information.

3 + variable	11	*Result <id>*	*Literal String* *Name*

OpExtInst

Execute an instruction in an imported set of extended instructions.

Result Type is as defined, per *Instruction*, in the external specification for *Set*.

Set is the result of an **OpExtInstImport** instruction.

Instruction is the enumerant of the instruction to execute within *Set*. This literal operand is limited to a single word. The semantics of the instruction must be defined in the external specification for *Set*.

Operand 1, … are the operands to the extended instruction.

5 + variable	12	*<id>* *Result Type*	*Result <id>*	*<id>* *Set*	*Literal Number* *Instruction*	*<id>, <id>, …* *Operand 1*, *Operand 2*, …

3.32.5. Mode-Setting Instructions

OpMemoryModel

Set addressing model and memory model for the entire module.

Addressing Model selects the module’s **Addressing Model**.

Memory Model selects the module’s memory model, see **Memory Model**.

3	14	*Addressing Model*	*Memory Model*

OpEntryPoint

Declare an entry point and its execution model.

Execution Model is the execution model for the entry point and its static call tree. See Execution Model.

Entry Point must be the *Result <id>* of an **OpFunction** instruction.

Name is a name string for the entry point. A module cannot have two **OpEntryPoint** instructions with the same Execution Model and the same *Name* string.

Interface is a list of *<id>* of global **OpVariable** instructions with either **Input** or **Output** for its Storage Class operand. These declare the input/output interface of the entry point. They could be a subset of the input/output declarations of the module, and a superset of those referenced by the entry point’s static call tree. It is invalid for the entry point’s static call tree to reference such an *<id>* if it was not listed with this instruction.

Interface <id> are forward references. They allow declaration of all variables forming an interface for an entry point, whether or not all the variables are actually used by the entry point.

4 + variable	15	*Execution Model*	*<id>* *Entry Point*	*Literal String* *Name*	*<id>, <id>, …* *Interface*

OpExecutionMode

Declare an execution mode for an entry point.

Entry Point must be the *Entry Point <id>* operand of an **OpEntryPoint** instruction.

Mode is the execution mode. See Execution Mode.

3 + variable	16	*<id>* *Entry Point*	*Execution Mode* *Mode*	*Optional literal(s)* See Execution Mode

OpCapability Declare a capability used by this module. *Capability* is the capability declared by this instruction. There are no restrictions on the order in which capabilities are declared. See the capabilities section for more detail.		
2	17	*Capability* *Capability*

3.32.6. Type-Declaration Instructions

OpTypeVoid Declare the void type.		
2	19	*Result <id>*

OpTypeBool Declare the *Boolean type*. Values of this type can only be either **true** or **false**. There is no physical size or bit pattern defined for these values. If they are stored (in conjunction with **OpVariable**), they can only be used with logical addressing operations, not physical, and only with non-externally visible shader Storage Classes: **Workgroup**, **CrossWorkgroup**, **Private**, and **Function**.		
2	20	*Result <id>*

OpTypeInt Declare a new *integer type*. *Width* specifies how many bits wide the type is. This literal operand is limited to a single word. The bit pattern of a signed integer value is two's complement. *Signedness* specifies whether there are signed semantics to preserve or validate. 0 indicates unsigned, or no signedness semantics 1 indicates signed semantics. In all cases, the type of operation of an instruction comes from the instruction's opcode, not the signedness of the operands.				
4	21	*Result <id>*	*Literal Number* *Width*	*Literal Number* *Signedness*

OpTypeFloat Declare a new *floating-point type*. *Width* specifies how many bits wide the type is. The bit pattern of a floating-point value is as described by the IEEE 754 standard.			
3	22	*Result <id>*	*Literal Number* *Width*

OpTypeVector Declare a new vector type. *Component Type* is the type of each component in the resulting type. It must be a scalar type. *Component Count* is the number of components in the resulting type. It must be at least 2. Components are numbered consecutively, starting with 0.				
4	23	*Result <id>*	*<id>* *Component Type*	*Literal Number* *Component Count*

OpTypeMatrix				Capability: **Matrix**

Declare a new matrix type. *Column Type* is the type of each column in the matrix. It must be vector type. *Column Count* is the number of columns in the new matrix type. It must be at least 2. Matrix columns are numbered consecutively, starting with 0. This is true independently of any Decorations describing the memory layout of a matrix (e.g., **RowMajor** or **MatrixStride**).				
4	24	*Result <id>*	*<id>* *Column Type*	*Literal Number* *Column Count*

OpTypeImage

Declare a new image type. Consumed, for example, by **OpTypeSampledImage**. This type is opaque: values of this type have no defined physical size or bit pattern.

Sampled Type is the type of the components that result from sampling or reading from this image type. Must be a scalar numerical type or **OpTypeVoid**.

Dim is the image dimensionality (Dim).

Depth is whether or not this image is a depth image. (Note that whether or not depth comparisons are actually done is a property of the sampling opcode, not of this type declaration.)
0 indicates not a depth image
1 indicates a depth image
2 means no indication as to whether this is a depth or non-depth image

Arrayed must be one of the following indicated values:
0 indicates non-arrayed content
1 indicates arrayed content

MS must be one of the following indicated values:
0 indicates single-sampled content
1 indicates multisampled content

Sampled indicates whether or not this image will be accessed in combination with a sampler, and must be one of the following values:
0 indicates this is only known at run time, not at compile time
1 indicates will be used with sampler
2 indicates will be used without a sampler (a storage image)

Image Format is the Image Format, which can be **Unknown**, depending on the client API.

If *Dim* is **SubpassData**, *Sampled* must be 2, *Image Format* must be **Unknown**, and the Execution Model must be **Fragment**.

Access Qualifier is an image **Access Qualifier**.

9 + variable	25	*Result <id>*	*<id> Sampled Type*	*Dim*	*Literal Number Depth*	*Literal Number Arrayed*	*Literal Number MS*	*Literal Number Sampled*	*Image Format*	Optional *Access Qualifier*

OpTypeSampler

Declare the sampler type. Consumed by **OpSampledImage**. This type is opaque: values of this type have no defined physical size or bit pattern.

2	26	*Result <id>*

OpTypeSampledImage

Declare a sampled image type, the *Result Type* of **OpSampledImage**, or an externally combined sampler and image. This type is opaque: values of this type have no defined physical size or bit pattern.

Image Type must be an **OpTypeImage**. It is the type of the image in the combined sampler and image type.

3	27	*Result <id>*	*<id>* *Image Type*

OpTypeArray

Declare a new array type: a dynamically-indexable ordered aggregate of elements all having the same type.

Element Type is the type of each element in the array.

Length is the number of elements in the array. It must be at least 1. *Length* must come from a *constant instruction* of an *integer-type* scalar whose value is at least 1.

Array elements are number consecutively, starting with 0.

4	28	*Result <id>*	*<id>* *Element Type*	*<id>* *Length*

OpTypeRuntimeArray Declare a new run-time array type. Its length is not known at compile time. *Element Type* is the type of each element in the array. It must be a concrete type. See **OpArrayLength** for getting the *Length* of an array of this type. Objects of this type can only be created with **OpVariable** using the **Uniform** Storage Class.	Capability: **Shader**

3	29	*Result <id>*	*<id>* *Element Type*

OpTypeStruct

Declare a new structure type: an aggregate of potentially heterogeneous members.

Member N type is the type of member *N* of the structure. The first member is member 0, the next is member 1, …

If an operand is not yet defined, it must be defined by an **OpTypePointer**, where the type pointed to is an **OpTypeStruct**.

2 + variable	30	*Result <id>*	*<id>, <id>, …* *Member 0 type*, *member 1 type*, …

OpTypeOpaque Declare a structure type with no body specified.			*Capability*: **Kernel**
3 + variable	31	*Result <id>*	*Literal String* The name of the opaque type.

OpTypePointer Declare a new pointer type. *Storage Class* is the Storage Class of the memory holding the object pointed to. If there was a forward reference to this type from an **OpTypeForwardPointer**, the *Storage Class* of that instruction must equal the *Storage Class* of this instruction. *Type* is the type of the object pointed to.				
4	32	*Result <id>*	*Storage Class*	*<id>* *Type*

OpTypeFunction Declare a new function type. **OpFunction** will use this to declare the return type and parameter types of a function. **OpFunction** is the only valid use of **OpTypeFunction**. *Return Type* is the type of the return value of functions of this type. It must be a concrete or abstract type, or a pointer to such a type. If the function has no return value, *Return Type* must be **OpTypeVoid**. *Parameter N Type* is the type *<id>* of the type of parameter *N*.				
3 + variable	33	*Result <id>*	*<id>* *Return Type*	*<id>, <id>, …* *Parameter 0 Type*, *Parameter 1 Type*, …

OpTypeEvent Declare an OpenCL event type.		*Capability*: **Kernel**
2	34	*Result <id>*

OpTypeDeviceEvent Declare an OpenCL device-side event type.		*Capability*: **DeviceEnqueue**
2	35	*Result <id>*

OpTypeReserveId Declare an OpenCL reservation id type.		Capability: **Pipes**
2	36	*Result <id>*

OpTypeQueue Declare an OpenCL queue type.		Capability: **DeviceEnqueue**
2	37	*Result <id>*

OpTypePipe Declare an OpenCL pipe type. *Qualifier* is the pipe access qualifier.			Capability: **Pipes**
3	38	*Result <id>*	*Access Qualifier* *Qualifier*

OpTypeForwardPointer Declare the Storage Class for a forward reference to a pointer. *Pointer Type* is a forward reference to the result of an **OpTypePointer**. The type of object the pointer points to is declared by the **OpTypePointer** instruction, not this instruction. Subsequent **OpTypeStruct** instructions can use *Pointer Type* as an operand. *Storage Class* is the Storage Class of the memory holding the object pointed to.			Capability: **Addresses**
3	39	*<id>* *Pointer Type*	*Storage Class*

OpTypePipeStorage Declare the OpenCL pipe-storage type.		Capability: **PipeStorage**
2	322	*Result <id>*

OpTypeNamedBarrier Declare the named-barrier type.		Capability: **NamedBarrier**
2	327	*Result <id>*

3.32.7. Constant-Creation Instructions

OpConstantTrue

Declare a **true** *Boolean-type* scalar constant.

Result Type must be the scalar *Boolean type*.

3	41	*<id>* *Result Type*	*Result <id>*

OpConstantFalse

Declare a **false** *Boolean-type* scalar constant.

Result Type must be the scalar *Boolean type*.

3	42	*<id>* *Result Type*	*Result <id>*

OpConstant

Declare a new *integer-type* or *floating-point-type* scalar constant.

Result Type must be a scalar *integer type* or *floating-point type*.

Value is the bit pattern for the constant. Types 32 bits wide or smaller take one word. Larger types take multiple words, with low-order words appearing first.

3 + variable	43	*<id>* *Result Type*	*Result <id>*	*Literal, Literal, …* *Value*

OpConstantComposite

Declare a new *composite* constant.

Result Type must be a *composite* type, whose top-level members/elements/components/columns have the same type as the types of the *Constituents*. The ordering must be the same between the top-level types in *Result Type* and the *Constituents*.

Constituents will become members of a structure, or elements of an array, or components of a vector, or columns of a matrix. There must be exactly one *Constituent* for each top-level member/element /component/column of the result. The *Constituents* must appear in the order needed by the definition of the *Result Type*. The *Constituents* must all be *<id>s* of other constant declarations or an **OpUndef**.

3 + variable	44	*<id>* *Result Type*	*Result <id>*	*<id>, <id>, …* *Constituents*

OpConstantSampler Declare a new sampler constant. *Result Type* must be **OpTypeSampler**. *Sampler Addressing Mode* is the addressing mode; a literal from Sampler Addressing Mode. *Param* is one of: 0: Non Normalized 1: Normalized *Sampler Filter Mode* is the filter mode; a literal from Sampler Filter Mode.					Capability: **LiteralSampler**	
6	45	*<id>* *Result Type*	*Result <id>*	*Sampler Addressing Mode*	*Literal Number* *Param*	*Sampler Filter Mode*

OpConstantNull Declare a new *null* constant value. The *null* value is type dependent, defined as follows: - Scalar Boolean: **false** - Scalar integer: 0 - Scalar floating point: +0.0 (all bits 0) - All other scalars: Abstract - Composites: Members are set recursively to the null constant according to the null value of their constituent types. *Result Type* must be one of the following types: - Scalar or vector *Boolean type* - Scalar or vector *integer type* - Scalar or vector *floating-point type* - Pointer type - *Event type* - *Device side event type* - *Reservation id type* - *Queue type* - *Composite type*			
3	46	*<id>* *Result Type*	*Result <id>*

OpSpecConstantTrue

Declare a *Boolean-type* scalar specialization constant with a default value of **true**.

This instruction can be specialized to become either an **OpConstantTrue** or **OpConstantFalse** instruction.

Result Type must be the scalar *Boolean type*.

See Specialization.

3	48	*<id>* *Result Type*	*Result <id>*

OpSpecConstantFalse

Declare a *Boolean-type* scalar specialization constant with a default value of **false**.

This instruction can be specialized to become either an **OpConstantTrue** or **OpConstantFalse** instruction.

Result Type must be the scalar *Boolean type*.

See Specialization.

3	49	*<id>* *Result Type*	*Result <id>*

OpSpecConstant

Declare a new *integer-type* or *floating-point-type* scalar specialization constant.

Result Type must be a scalar *integer type* or *floating-point type*.

Value is the bit pattern for the default value of the constant. Types 32 bits wide or smaller take one word. Larger types take multiple words, with low-order words appearing first.

This instruction can be specialized to become an **OpConstant** instruction.

See Specialization.

3 + variable	50	*<id>* *Result Type*	*Result <id>*	*Literal, Literal, …* *Value*

OpSpecConstantComposite

Declare a new *composite* specialization constant.

Result Type must be a *composite* type, whose top-level members/elements/components/columns have the same type as the types of the *Constituents*. The ordering must be the same between the top-level types in *Result Type* and the *Constituents*.

Constituents will become members of a structure, or elements of an array, or components of a vector, or columns of a matrix. There must be exactly one *Constituent* for each top-level member/element /component/column of the result. The *Constituents* must appear in the order needed by the definition of the type of the result. The *Constituents* must be the *<id>* of other specialization constant or constant declarations.

This instruction will be specialized to an **OpConstantComposite** instruction.

See Specialization.

3 + variable	51	*<id>* *Result Type*	*Result <id>*	*<id>, <id>, …* *Constituents*

OpSpecConstantOp

Declare a new specialization constant that results from doing an operation.

Result Type must be the type required by the *Result Type* of *Opcode*.

Opcode must be one of the following opcodes. This literal operand is limited to a single word.
OpSConvert, **OpFConvert**
OpSNegate, **OpNot**
OpIAdd, **OpISub**
OpIMul, **OpUDiv**, **OpSDiv**, **OpUMod**, **OpSRem**, **OpSMod**
OpShiftRightLogical, **OpShiftRightArithmetic**, **OpShiftLeftLogical**
OpBitwiseOr, **OpBitwiseXor**, **OpBitwiseAnd**
OpVectorShuffle, **OpCompositeExtract**, **OpCompositeInsert**
OpLogicalOr, **OpLogicalAnd**, **OpLogicalNot**,
OpLogicalEqual, **OpLogicalNotEqual**
OpSelect
OpIEqual, **OpINotEqual**
OpULessThan, **OpSLessThan**
OpUGreaterThan, **OpSGreaterThan**
OpULessThanEqual, **OpSLessThanEqual**
OpUGreaterThanEqual, **OpSGreaterThanEqual**

If the **Shader** capability was declared, the following opcode is also valid:
OpQuantizeToF16

If the **Kernel** capability was declared, the following opcodes are also valid:
OpConvertFToS, **OpConvertSToF**
OpConvertFToU, **OpConvertUToF**
OpUConvert
OpConvertPtrToU, **OpConvertUToPtr**
OpGenericCastToPtr, **OpPtrCastToGeneric**
OpBitcast
OpFNegate
OpFAdd, **OpFSub**
OpFMul, **OpFDiv**
OpFRem, **OpFMod**
OpAccessChain, **OpInBoundsAccessChain**
OpPtrAccessChain, **OpInBoundsPtrAccessChain**

Operands are the operands required by *opcode*, and satisfy the semantics of *opcode*. In addition, all *Operands* must be the *<id>s* of other constant instructions, or for the **AccessChain** named opcodes, their *Base* is allowed to be a global (module scope) **OpVariable** instruction.

See Specialization.

4 + variable	52	*<id>* *Result Type*	*Result <id>*	*Literal Number* *Opcode*	*<id>, <id>, …* *Operands*

3.32.8. Memory Instructions

OpVariable

Allocate an object in memory, resulting in a pointer to it, which can be used with **OpLoad** and **OpStore**.

Result Type must be an **OpTypePointer**. Its *Type* operand is the type of object in memory.

Storage Class is the Storage Class of the memory holding the object. It cannot be **Generic**.

Initializer is optional. If *Initializer* is present, it will be the initial value of the variable's memory content. *Initializer* must be an *<id>* from a *constant instruction* or a global (module scope) **OpVariable** instruction. *Initializer* must have the same type as the type pointed to by *Result Type*.

4 + variable	59	*<id>* *Result Type*	*Result <id>*	*Storage Class*	Optional *<id>* *Initializer*

OpImageTexelPointer

Form a pointer to a texel of an image. Use of such a pointer is limited to atomic operations.

Result Type must be an **OpTypePointer** whose *Storage Class* operand is **Image**. Its *Type* operand must be a scalar numerical type or **OpTypeVoid**.

Image must have a type of **OpTypePointer** with *Type* **OpTypeImage**. The *Sampled Type* of the type of *Image* must be the same as the *Type* pointed to by *Result Type*. The *Dim* operand of *Type* cannot be **SubpassData**.

Coordinate and *Sample* specify which texel and sample within the image to form a pointer to.

Coordinate must be a scalar or vector of *integer type*. It must have the number of components specified below, given the following *Arrayed* and *Dim* operands of the type of the **OpTypeImage**.

If *Arrayed* is 0:
1D: scalar
2D: 2 components
3D: 3 components
Cube: 3 components
Rect: 2 components
Buffer: scalar

If *Arrayed* is 1:
1D: 2 components
2D: 3 components
Cube: 4 components

Sample must be an *integer type* scalar. It specifies which sample to select at the given coordinate. It must be a valid *<id>* for the value 0 if the **OpTypeImage** has *MS* of 0.

6	60	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	*<id>* *Sample*

OpLoad

Load through a pointer.

Result Type is the type of the loaded object.

Pointer is the pointer to load through. Its type must be an **OpTypePointer** whose *Type* operand is the same as *Result Type*.

Memory Access must be a Memory Access literal. If not present, it is the same as specifying **None**.

4 + variable	61	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	Optional *Memory Access*

OpStore

Store through a pointer.

Pointer is the pointer to store through. Its type must be an **OpTypePointer** whose *Type* operand is the same as the type of *Object*.

Object is the object to store.

Memory Access must be a Memory Access literal. If not present, it is the same as specifying **None**.

3 + variable	62	*<id>* *Pointer*	*<id>* *Object*	Optional *Memory Access*

OpCopyMemory

Copy from the memory pointed to by *Source* to the memory pointed to by *Target*. Both operands must be non-void pointers of the same type. Matching Storage Class is not required. The amount of memory copied is the size of the type pointed to.

Memory Access must be a Memory Access literal. If not present, it is the same as specifying **None**.

3 + variable	63	*<id>* *Target*	*<id>* *Source*	Optional *Memory Access*

OpCopyMemorySized

Copy from the memory pointed to by *Source* to the memory pointed to by *Target*.

Size is the number of bytes to copy. It must have a scalar integer type. If it is a constant instruction, the constant value cannot be 0. It is invalid for both the constant's type to have *Signedness* of 1 and to have the sign bit set. Otherwise, as a run-time value, *Size* is treated as unsigned, and if its value is 0, no memory access will be made.

Memory Access must be a Memory Access literal. If not present, it is the same as specifying **None**.

Capability:
Addresses

4 + variable	64	*<id>* *Target*	*<id>* *Source*	*<id>* *Size*	Optional *Memory Access*

OpAccessChain

Create a pointer into a *composite* object that can be used with **OpLoad** and **OpStore**.

Result Type must be an **OpTypePointer**. Its *Type* operand must be the type reached by walking the *Base's* type hierarchy down to the last provided index in *Indexes*, and its *Storage Class* operand must be the same as the Storage Class of *Base*.

Base must be a pointer, pointing to the base of a composite object.

Indexes walk the type hierarchy to the desired depth, potentially down to scalar granularity. The first index in *Indexes* will select the top-level member/element/component/element of the base composite. All composite constituents use zero-based numbering, as described by their **OpType…** instruction. The second index will apply similarly to that result, and so on. Once any non-composite type is reached, there must be no remaining (unused) indexes. Each of the *Indexes* must:
- be a scalar integer type,
- be an **OpConstant** when indexing into a structure.

4 + variable	65	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>, <id>, …* *Indexes*

OpInBoundsAccessChain

Has the same semantics as **OpAccessChain**, with the addition that the resulting pointer is known to point within the base object.

4 + variable	66	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>, <id>, …* *Indexes*

OpPtrAccessChain

Has the same semantics as **OpAccessChain**, with the addition of the *Element* operand.

Element is used to do the initial dereference of *Base*: *Base* is treated as the address of the first element of an array, and the *Element* element's address is computed to be the base for the *Indexes*, as per **OpAccessChain**. The type of *Base* after being dereferenced with *Element* is still the same as the original type of *Base*.

Note: If *Base* is originally typed to be a pointer an array, and the desired operation is to select an element of that array, **OpAccessChain** should be directly used, as its first *Index* will select the array element.

Capability:
Addresses

5 + variable	67	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Element*	*<id>, <id>, …* *Indexes*

OpArrayLength

Length of a run-time array.

Result Type must be an OpTypeInt with 32-bit *Width* and 0 *Signedness*.

Structure must have a type of **OpTypeStruct** whose last member is a run-time array.

Array member is the last member number of *Structure* and must have a type from **OpTypeRuntimeArray**.

Capability:
Shader

5	68	*<id>* *Result Type*	*Result <id>*	*<id>* *Structure*	*Literal Number* *Array member*

OpGenericPtrMemSemantics

Result is a valid **Memory Semantics** which includes mask bits set for the Storage Class for the specific (non-Generic) Storage Class of *Pointer*.

Pointer must point to **Generic** Storage Class.

Result Type must be an OpTypeInt with 32-bit *Width* and 0 *Signedness*.

Capability:
Kernel

4	69	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpInBoundsPtrAccessChain

Has the same semantics as **OpPtrAccessChain**, with the addition that the resulting pointer is known to point within the base object.

Capability:
Addresses

5 + variable	70	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Element*	*<id>, <id>, …* *Indexes*

3.32.9. Function Instructions

OpFunction

Add a function. This instruction must be immediately followed by one **OpFunctionParameter** instruction per each formal parameter of this function. This function's body or declaration will terminate with the next **OpFunctionEnd** instruction.

The *Result <id>* cannot be used generally by other instructions. It can only be used by **OpFunctionCall**, **OpEntryPoint**, and decoration instructions.

Result Type must be the same as the *Return Type* declared in *Function Type*.

Function Type is the result of an **OpTypeFunction**, which declares the types of the return value and parameters of the function.

5	54	*<id>* *Result Type*	*Result <id>*	*Function Control*	*<id>* *Function Type*

OpFunctionParameter

Declare a formal parameter of the current function.

Result Type is the type of the parameter.

This instruction must immediately follow an **OpFunction** or **OpFunctionParameter** instruction. The order of contiguous **OpFunctionParameter** instructions is the same order arguments will be listed in an **OpFunctionCall** instruction to this function. It is also the same order in which *Parameter Type* operands are listed in the **OpTypeFunction** of the *Function Type* operand for this function's **OpFunction** instruction.

3	55	*<id>* *Result Type*	*Result <id>*

OpFunctionEnd

Last instruction of a function.

1	56

OpFunctionCall

Call a function.

Result Type is the type of the return value of the function. It must be the same as the *Return Type* operand of the *Function Type* operand of the *Function* operand.

Function is an **OpFunction** instruction. This could be a forward reference.

Argument N is the object to copy to parameter *N* of *Function*.

Note: A forward call is possible because there is no missing type information: *Result Type* must match the *Return Type* of the function, and the calling argument types must match the formal parameter types.

4 + variable	57	*<id>* *Result Type*	*Result <id>*	*<id>* *Function*	*<id>, <id>, …* *Argument 0*, *Argument 1*, …

3.32.10. Image Instructions

OpSampledImage

Create a sampled image, containing both a sampler and an image.

Result Type must be the **OpTypeSampledImage** type.

Image is an object whose type is an **OpTypeImage**, whose *Sampled* operand is 0 or 1, and whose *Dim* operand is not **SubpassData**.

Sampler must be an object whose type is **OpTypeSampler**.

5	86	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Sampler*

OpImageSampleImplicitLod

Capability:
Shader

Sample an image with an implicit level of detail.

Result Type must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**).

Sampled Image must be an object whose type is **OpTypeSampledImage**.

Coordinate must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components.

Image Operands encodes what operands follow, as per Image Operands.

This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.

5 + variable	87	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSampleExplicitLod

Sample an image using an explicit level of detail.

Result Type must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**).

Sampled Image must be an object whose type is **OpTypeSampledImage**.

Coordinate must be a scalar or vector of *floating-point type* or *integer type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. Unless the **Kernel** capability is being used, it must be floating point. It may be a vector larger than needed, but all unused components will appear after all used components.

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.

7 + variable	88	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSampleDrefImplicitLod

Capability: **Shader**

Sample an image doing depth-comparison with an implicit level of detail.

Result Type must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**.

Sampled Image must be an object whose type is **OpTypeSampledImage**.

Coordinate must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components.

D_{ref} is the depth-comparison reference value.

Image Operands encodes what operands follow, as per Image Operands.

This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.

6 + variable	89	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSampleDrefExplicitLod

Capability: **Shader**

Sample an image doing depth-comparison using an explicit level of detail.

Result Type must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**.

Sampled Image must be an object whose type is **OpTypeSampledImage**.

Coordinate must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components.

D_{ref} is the depth-comparison reference value.

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.

8 + variable	90	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*<id>* D_{ref}	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSampleProjImplicitLod

Sample an image with with a project coordinate and an implicit level of detail.

Result Type must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**).

Sampled Image must be an object whose type is **OpTypeSampledImage**. The *Dim* operand of the underlying **OpTypeImage** must be **1D**, **2D**, **3D**, or **Rect**, and the *Arrayed* and *MS* operands must be 0.

Coordinate is a floating-point vector containing (*u* [, *v*] [, *w*], *q*), as needed by the definition of *Sampled Image*, with the *q* component consumed for the projective division. That is, the actual sample coordinate will be (*u/q* [, *v/q*] [, *w/q*]), as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components.

Image Operands encodes what operands follow, as per Image Operands.

This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.

Capability:
Shader

5 + variable	91	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSampleProjExplicitLod Sample an image with a project coordinate using an explicit level of detail. *Result Type* must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**). *Sampled Image* must be an object whose type is **OpTypeSampledImage**. The *Dim* operand of the underlying **OpTypeImage** must be **1D**, **2D**, **3D**, or **Rect**, and the *Arrayed* and *MS* operands must be 0. *Coordinate* is a floating-point vector containing (*u* [, *v*] [, *w*], *q*), as needed by the definition of *Sampled Image*, with the *q* component consumed for the projective division. That is, the actual sample coordinate will be (*u/q* [, *v/q*] [, *w/q*]), as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components. *Image Operands* encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.							Capability: **Shader**	
7 + variable	92	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSampleProjDrefImplicitLod Sample an image with a project coordinate, doing depth-comparison, with an implicit level of detail. *Result Type* must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**. *Sampled Image* must be an object whose type is **OpTypeSampledImage**. The *Dim* operand of the underlying **OpTypeImage** must be **1D**, **2D**, **3D**, or **Rect**, and the *Arrayed* and *MS* operands must be 0. *Coordinate* is a floating-point vector containing (*u* [, *v*] [, *w*], *q*), as needed by the definition of *Sampled Image*, with the *q* component consumed for the projective division. That is, the actual sample coordinate will be (*u/q* [, *v/q*] [, *w/q*]), as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components. D_{ref} /*q* is the depth-comparison reference value. *Image Operands* encodes what operands follow, as per Image Operands. This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.							Capability: **Shader**	
6 + variable	93	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSampleProjDrefExplicitLod Sample an image with a project coordinate, doing depth-comparison, using an explicit level of detail. *Result Type* must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**. *Sampled Image* must be an object whose type is **OpTypeSampledImage**. The *Dim* operand of the underlying **OpTypeImage** must be **1D**, **2D**, **3D**, or **Rect**, and the *Arrayed* and *MS* operands must be 0. *Coordinate* is a floating-point vector containing (*u* [, *v*] [, *w*], *q*), as needed by the definition of *Sampled Image*, with the *q* component consumed for the projective division. That is, the actual sample coordinate will be (*u/q* [, *v/q*] [, *w/q*]), as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components. D_{ref} /*q* is the depth-comparison reference value. *Image Operands* encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.								Capability: **Shader**	
8 + variable	94	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* D_{ref}	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageFetch Fetch a single texel from a sampled image. *Result Type* must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**). *Image* must be an object whose type is **OpTypeImage**. Its *Dim* operand cannot be **Cube**, and its *Sampled* operand must be 1. *Coordinate* is an integer scalar or vector containing (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. *Image Operands* encodes what operands follow, as per Image Operands.							
5 + variable	95	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

<table>
<tr><td colspan="7">OpImageGather

Gathers the requested component from four texels.

Result Type must be a vector of four components of floating-point type or integer type. Its components must be the same as Sampled Type of the underlying OpTypeImage (unless that underlying Sampled Type is OpTypeVoid). It has one component per gathered texel.

Sampled Image must be an object whose type is OpTypeSampledImage. Its OpTypeImage must have a Dim of 2D, Cube, or Rect.

Coordinate must be a scalar or vector of floating-point type. It contains (u[, v] … [, array layer]) as needed by the definition of Sampled Image.

Component is the component number that will be gathered from all four texels. It must be 0, 1, 2 or 3.

Image Operands encodes what operands follow, as per Image Operands.</td><td colspan="2">Capability:
Shader</td></tr>
<tr><td>6 + variable</td><td>96</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td><td><id>
Component</td><td>Optional
Image Operands</td><td>Optional
<id>, <id>, …</td></tr>
</table>

<table>
<tr><td colspan="7">OpImageDrefGather

Gathers the requested depth-comparison from four texels.

Result Type must be a scalar of integer type or floating-point type. It must be the same as Sampled Type of the underlying OpTypeImage. It has one component per gathered texel.

Sampled Image must be an object whose type is OpTypeSampledImage. Its OpTypeImage must have a Dim of 2D, Cube, or Rect.

Coordinate must be a scalar or vector of floating-point type. It contains (u[, v] … [, array layer]) as needed by the definition of Sampled Image.

D_{ref} is the depth-comparison reference value.

Image Operands encodes what operands follow, as per Image Operands.</td><td colspan="2">Capability:
Shader</td></tr>
<tr><td>6 + variable</td><td>97</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td><td><id>
D_{ref}</td><td>Optional
Image Operands</td><td>Optional
<id>, <id>, …</td></tr>
</table>

OpImageRead

Read a texel from an image without a sampler.

Result Type must be a scalar or vector of *floating-point type* or *integer type*. Its component type must be the same as *Sampled Type* of the **OpTypeImage** (unless that *Sampled Type* is **OpTypeVoid**).

Image must be an object whose type is **OpTypeImage** with a *Sampled* operand of 0 or 2. If the *Sampled* operand is 2, then some dimensions require a capability; e.g., one of **Image1D**, **ImageRect**, **ImageBuffer**, **ImageCubeArray**, or **ImageMSArray**.

Coordinate is an integer scalar or vector containing non-normalized texel coordinates (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Image*. If the coordinates are outside the image, the memory location that is accessed is undefined.

When the *Image Dim* operand is **SubpassData**, *Coordinate* is relative to the current fragment location. That is, the integer value (rounded down) of the current fragment's window-relative *(x, y)* coordinate is added to *(u, v)*.

When the *Image Dim* operand is not **SubpassData**, the *Image Format* must not be **Unknown**, unless the **StorageImageReadWithoutFormat** Capability was declared.

Image Operands encodes what operands follow, as per Image Operands.

5 + variable	98	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageWrite

Write a texel to an image without a sampler.

Image must be an object whose type is **OpTypeImage** with a *Sampled* operand of 0 or 2. If the *Sampled* operand is 2, then some dimensions require a capability; e.g., one of **Image1D**, **ImageRect**, **ImageBuffer**, **ImageCubeArray**, or **ImageMSArray**. Its *Dim* operand cannot be **SubpassData**.

Coordinate is an integer scalar or vector containing non-normalized texel coordinates (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Image*. If the coordinates are outside the image, the memory location that is accessed is undefined.

Texel is the data to write. Its component type must be the same as *Sampled Type* of the **OpTypeImage** (unless that *Sampled Type* is **OpTypeVoid**).

The *Image Format* must not be **Unknown**, unless the **StorageImageWriteWithoutFormat** Capability was declared.

Image Operands encodes what operands follow, as per Image Operands.

4 + variable	99	*<id>* *Image*	*<id>* *Coordinate*	*<id>* *Texel*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImage

Extract the image from a sampled image.

Result Type must be **OpTypeImage**.

Sampled Image must have type **OpTypeSampledImage** whose *Image Type* is the same as *Result Type*.

4	100	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*

OpImageQueryFormat

Query the image format of an image created with an **Unknown** Image Format.

Result Type must be a scalar integer type. The resulting value is an enumerant from Image Channel Data Type.

Image must be an object whose type is **OpTypeImage**.

Capability:
Kernel

4	101	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*

OpImageQueryOrder

Query the channel order of an image created with an **Unknown** Image Format.

Result Type must be a scalar integer type. The resulting value is an enumerant from Image Channel Order.

Image must be an object whose type is **OpTypeImage**.

Capability:
Kernel

4	102	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*

<table>
<tr><td colspan="5">OpImageQuerySizeLod

Query the dimensions of Image for mipmap level for Level of Detail.

Result Type must be an integer type scalar or vector. The number of components must be
1 for 1D Dim,
2 for 2D, and Cube Dimensionalities,
3 for 3D Dim,
plus 1 more if the image type is arrayed. This vector is filled in with (width [, height] [, depth] [, elements]) where elements is the number of layers in an image array, or the number of cubes in a cube-map array.

Image must be an object whose type is OpTypeImage. Its Dim operand must be one of 1D, 2D, 3D, or Cube, and its MS must be 0. See OpImageQuerySize for querying image types without level of detail.

Level of Detail is used to compute which mipmap level to query, as described in the API specification.</td><td>Capability:
Kernel, ImageQuery</td></tr>
<tr><td>5</td><td>103</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Image</td><td><id>
Level of Detail</td></tr>
</table>

<table>
<tr><td colspan="4">OpImageQuerySize

Query the dimensions of Image, with no level of detail.

Result Type must be an integer type scalar or vector. The number of components must be
1 for Buffer Dim,
2 for 2D and Rect Dimensionalities,
3 for 3D Dim,
plus 1 more if the image type is arrayed. This vector is filled in with (width [, height] [, elements]) where elements is the number of layers in an image array.

Image must be an object whose type is OpTypeImage. Its Dim operand must be one of Rect or Buffer, or if its MS is 1, it can be 2D, or, if its Sampled Type is 0 or 2, it can be 2D or 3D. It cannot be an image with level of detail; there is no implicit level-of-detail consumed by this instruction. See OpImageQuerySizeLod for querying images having level of detail.</td><td>Capability:
Kernel, ImageQuery</td></tr>
<tr><td>4</td><td>104</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Image</td></tr>
</table>

OpImageQueryLod Query the mipmap level and the level of detail for a hypothetical sampling of *Image* at *Coordinate* using an implicit level of detail. *Result Type* must be a two-component floating-point type vector. The first component of the result will contain the mipmap array layer. The second component of the result will contain the implicit level of detail relative to the base level. *Image* must be an object whose type is **OpTypeImage**. Its *Dim* operand must be one of **1D**, **2D**, **3D**, or **Cube**. *Coordinate* must be a scalar or vector of *floating-point type* or *integer type*. It contains (*u*[, *v*] …) as needed by the definition of *Sampled Image*, not including any array layer index. Unless the **Kernel** capability is being used, it must be floating point. If called on an incomplete image, the results are undefined. This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.					Capability: **ImageQuery**
5	105	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*

OpImageQueryLevels Query the number of mipmap levels accessible through *Image*. *Result Type* must be a scalar integer type. The result is the number of mipmap levels, as defined by the API specification. *Image* must be an object whose type is **OpTypeImage**. Its *Dim* operand must be one of **1D**, **2D**, **3D**, or **Cube**.				Capability: **Kernel**, **ImageQuery**
4	106	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*

OpImageQuerySamples Query the number of samples available per texel fetch in a multisample image. *Result Type* must be a scalar integer type. The result is the number of samples. *Image* must be an object whose type is **OpTypeImage**. Its *Dim* operand must be one of **2D** and *MS* of 1.				Capability: **Kernel**, **ImageQuery**
4	107	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*

<table>
<tr><td colspan="6">OpImageSparseSampleImplicitLod

Sample a sparse image with an implicit level of detail.

Result Type must be an OpTypeStruct with two members. The first member’s type must be an integer type scalar. It will hold a Residency Code that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as Sampled Type of the underlying OpTypeImage (unless that underlying Sampled Type is OpTypeVoid).

Sampled Image must be an object whose type is OpTypeSampledImage.

Coordinate must be a scalar or vector of floating-point type. It contains (u[, v] … [, array layer]) as needed by the definition of Sampled Image. It may be a vector larger than needed, but all unused components will appear after all used components.

Image Operands encodes what operands follow, as per Image Operands.

This instruction is only valid in the Fragment Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.</td><td colspan="2">Capability:
SparseResidency</td></tr>
<tr><td>5 + variable</td><td>305</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td><td>Optional
Image Operands</td><td>Optional
<id>, <id>, …</td></tr>
</table>

<table>
<tr><td colspan="7">OpImageSparseSampleExplicitLod

Sample a sparse image using an explicit level of detail.

Result Type must be an OpTypeStruct with two members. The first member’s type must be an integer type scalar. It will hold a Residency Code that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as Sampled Type of the underlying OpTypeImage (unless that underlying Sampled Type is OpTypeVoid).

Sampled Image must be an object whose type is OpTypeSampledImage.

Coordinate must be a scalar or vector of floating-point type or integer type. It contains (u[, v] … [, array layer]) as needed by the definition of Sampled Image. Unless the Kernel capability is being used, it must be floating point. It may be a vector larger than needed, but all unused components will appear after all used components.

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.</td><td colspan="2">Capability:
SparseResidency</td></tr>
<tr><td>7 + variable</td><td>306</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td><td>Image Operands</td><td><id></td><td>Optional
<id>, <id>, …</td></tr>
</table>

<table>
<tr><td colspan="7">OpImageSparseSampleDrefImplicitLod

Sample a sparse image doing depth-comparison with an implicit level of detail.

Result Type must be an OpTypeStruct with two members. The first member's type must be an integer type scalar. It will hold a Residency Code that can be passed to OpImageSparseTexelsResident. The second member must be a scalar of integer type or floating-point type. It must be the same as Sampled Type of the underlying OpTypeImage.

Sampled Image must be an object whose type is OpTypeSampledImage.

Coordinate must be a scalar or vector of floating-point type. It contains (u[, v] … [, array layer]) as needed by the definition of Sampled Image. It may be a vector larger than needed, but all unused components will appear after all used components.

D_{ref} is the depth-comparison reference value.

Image Operands encodes what operands follow, as per Image Operands.

This instruction is only valid in the Fragment Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.</td><td colspan="2">Capability:
SparseResidency</td></tr>
<tr><td>6 + variable</td><td>307</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td><td><id>
D_{ref}</td><td>Optional
Image Operands</td><td>Optional
<id>, <id>, …</td></tr>
</table>

OpImageSparseSampleDrefExplicitLod

Capability:
SparseResidency

Sample a sparse image doing depth-comparison using an explicit level of detail.

Result Type must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It will hold a *Residency Code* that can be passed to **OpImageSparseTexelsResident**. The second member must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**.

Sampled Image must be an object whose type is **OpTypeSampledImage**.

Coordinate must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components.

D_{ref} is the depth-comparison reference value.

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.

8 + variable	308	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*<id>* D_{ref}	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSparseSampleProjImplicitLod Instruction reserved for future use. Use of this instruction is invalid. ~~Sample a sparse image with a projective coordinate and an implicit level of detail.~~						Capability: **SparseResidency**	
5 + variable	309	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseSampleProjExplicitLod Instruction reserved for future use. Use of this instruction is invalid. ~~Sample a sparse image with a projective coordinate using an explicit level of detail.~~							Capability: **SparseResidency**	
7 + variable	310	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSparseSampleProjDrefImplicitLod Instruction reserved for future use. Use of this instruction is invalid. ~~Sample a sparse image with a projective coordinate, doing depth-comparison, with an implicit level of detail.~~							Capability: **SparseResidency**	
6 + variable	311	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseSampleProjDrefExplicitLod Instruction reserved for future use. Use of this instruction is invalid. ~~Sample a sparse image with a projective coordinate, doing depth-comparison, using an explicit level of detail.~~								Capability: **SparseResidency**	
8 + variable	312	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* D_{ref}	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSparseFetch

Fetch a single texel from a sparse image.

Result Type must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It will hold a *Residency Code* that can be passed to **OpImageSparseTexelsResident**. The second member must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**).

Image must be an object whose type is **OpTypeImage**. Its *Dim* operand cannot be **Cube**.

Coordinate is an integer scalar or vector containing (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*.

Image Operands encodes what operands follow, as per Image Operands.

Capability:
SparseResidency

5 + variable	313	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseGather

Gathers the requested component from four texels of a sparse image.

Result Type must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It will hold a *Residency Code* that can be passed to **OpImageSparseTexelsResident**. The second member must be a vector of four components of *floating-point type* or *integer type*. Its components must be the same as *Sampled Type* of the underlying **OpTypeImage** (unless that underlying *Sampled Type* is **OpTypeVoid**). It has one component per gathered texel.

Sampled Image must be an object whose type is **OpTypeSampledImage**. Its **OpTypeImage** must have a *Dim* of **2D**, **Cube**, or **Rect**.

Coordinate must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*.

Component is the component number that will be gathered from all four texels. It must be 0, 1, 2 or 3.

Image Operands encodes what operands follow, as per Image Operands.

Capability:
SparseResidency

6 + variable	314	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* *Component*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseDrefGather Gathers the requested depth-comparison from four texels of a sparse image. *Result Type* must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It will hold a *Residency Code* that can be passed to **OpImageSparseTexelsResident**. The second member must be a scalar of *integer type* or *floating-point type*. It must be the same as *Sampled Type* of the underlying **OpTypeImage**. It has one component per gathered texel. *Sampled Image* must be an object whose type is **OpTypeSampledImage**. Its **OpTypeImage** must have a *Dim* of **2D**, **Cube**, or **Rect**. *Coordinate* must be a scalar or vector of *floating-point type*. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. D_{ref} is the depth-comparison reference value. *Image Operands* encodes what operands follow, as per Image Operands.							Capability: **SparseResidency**	
6 + variable	315	*<id>* *Result Type*	*Result <id>*	*<id>* *Sampled Image*	*<id>* *Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseTexelsResident Translates a *Resident Code* into a Boolean. Result is **false** if any of the texels were in uncommitted texture memory, and **true** otherwise. *Result Type* must be a *Boolean type* scalar. *Resident Code* is a value from an **OpImageSparse…** instruction that returns a resident code.				Capability: **SparseResidency**
4	316	*<id>* *Result Type*	*Result <id>*	*<id>* *Resident Code*

OpImageSparseRead

Read a texel from a sparse image without a sampler.

Result Type must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It will hold a *Residency Code* that can be passed to **OpImageSparseTexelsResident**. The second member must be a scalar or vector of *floating-point type* or *integer type*. Its component type must be the same as *Sampled Type* of the **OpTypeImage** (unless that *Sampled Type* is **OpTypeVoid**).

Image must be an object whose type is **OpTypeImage** with a *Sampled* operand of 2.

Coordinate is an integer scalar or vector containing non-normalized texel coordinates (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Image*. If the coordinates are outside the image, the memory location that is accessed is undefined.

The *Image Format* must not be **Unknown**, unless the **StorageImageReadWithoutFormat** Capability was declared.

Image Operands encodes what operands follow, as per Image Operands.

Capability:
SparseResidency

5 + variable	320	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

3.32.11. Conversion Instructions

OpConvertFToU

Convert (value preserving) from floating point to unsigned integer, with round toward 0.0.

Result Type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

Float Value must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	109	*<id>* *Result Type*	*Result <id>*	*<id>* *Float Value*

OpConvertFToS

Convert (value preserving) from floating point to signed integer, with round toward 0.0.

Result Type must be a scalar or vector of *integer type*.

Float Value must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	110	*<id>* *Result Type*	*Result <id>*	*<id>* *Float Value*

OpConvertSToF

Convert (value preserving) from signed integer to floating point.

Result Type must be a scalar or vector of *floating-point type*.

Signed Value must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	111	*<id>* *Result Type*	*Result <id>*	*<id>* *Signed Value*

OpConvertUToF

Convert (value preserving) from unsigned integer to floating point.

Result Type must be a scalar or vector of *floating-point type*.

Unsigned Value must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	112	*<id>* *Result Type*	*Result <id>*	*<id>* *Unsigned Value*

OpUConvert

Convert (value preserving) unsigned width. This is either a truncate or a zero extend.

Result Type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

Unsigned Value must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*. The component width cannot equal the component width in *Result Type*.

Results are computed per component.

4	113	*<id>* *Result Type*	*Result <id>*	*<id>* *Unsigned Value*

OpSConvert

Convert (value preserving) signed width. This is either a truncate or a sign extend.

Result Type must be a scalar or vector of *integer type*.

Signed Value must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*. The component width cannot equal the component width in *Result Type*.

Results are computed per component.

4	114	*<id>* *Result Type*	*Result <id>*	*<id>* *Signed Value*

OpFConvert

Convert (value preserving) floating-point width.

Result Type must be a scalar or vector of *floating-point type*.

Float Value must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. The component width cannot equal the component width in *Result Type*.

Results are computed per component.

4	115	*<id>* *Result Type*	*Result <id>*	*<id>* *Float Value*

OpQuantizeToF16 Quantize a floating-point value to what is expressible by a 16-bit	Capability: **Shader**

<table>
<tr><td colspan="4">floating-point value.

Result Type must be a scalar or vector of floating-point type. The component width must be 32 bits.

Value is the value to quantize. The type of Value must be the same as Result Type.

If Value is an infinity, the result is the same infinity. If Value is a NaN, the result is a NaN, but not necessarily the same NaN. If Value is positive with a magnitude too large to represent as a 16-bit floating-point value, the result is positive infinity. If Value is negative with a magnitude too large to represent as a 16-bit floating-point value, the result is negative infinity. If the magnitude of Value is too small to represent as a normalized 16-bit floating-point value, the result is 0.

The RelaxedPrecision Decoration has no effect on this instruction.

Results are computed per component.</td><td></td></tr>
<tr><td>4</td><td>116</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Value</td></tr>
</table>

<table>
<tr><td colspan="4">OpConvertPtrToU

Convert a pointer to an unsigned integer type. A Result Type width larger than the width of Pointer will zero extend. A Result Type smaller than the width of Pointer will truncate. For same-width source and result, this is the same as OpBitcast.

Result Type must be a scalar or vector of integer type, whose Signedness operand is 0.</td><td>Capability:
Addresses</td></tr>
<tr><td>4</td><td>117</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td></tr>
</table>

<table>
<tr><td colspan="4">OpSatConvertSToU

Convert a signed integer to unsigned integer. Converted values outside the representable range of Result Type are clamped to the nearest representable value of Result Type.

Result Type must be a scalar or vector of integer type.

Signed Value must be a scalar or vector of integer type. It must have the same number of components as Result Type.

Results are computed per component.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>118</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Signed Value</td></tr>
</table>

OpSatConvertUToS

Capability:
Kernel

Convert an unsigned integer to signed integer. Converted values outside the representable range of *Result Type* are clamped to the nearest representable value of *Result Type*.

Result Type must be a scalar or vector of *integer type*.

Unsigned Value must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	119	*<id>* *Result Type*	*Result <id>*	*<id>* *Unsigned Value*

OpConvertUToPtr

Capability:
Addresses

Convert an integer to pointer. A *Result Type* width smaller than the width of *Integer Value* pointer will truncate. A *Result Type* width larger than the width of *Integer Value* pointer will zero extend.

Result Type must be an **OpTypePointer**. For same-width source and result, this is the same as **OpBitcast**.

4	120	*<id>* *Result Type*	*Result <id>*	*<id>* *Integer Value*

OpPtrCastToGeneric

Capability:
Kernel

Convert a pointer's Storage Class to **Generic**.

Result Type must be an **OpTypePointer**. Its Storage Class must be **Generic**.

Pointer must point to the **Workgroup**, **CrossWorkgroup**, or **Function** Storage Class.

Result Type and *Pointer* must point to the same type.

4	121	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpGenericCastToPtr

Capability:
Kernel

Convert a pointer's Storage Class to a non-**Generic** class.

Result Type must be an **OpTypePointer**. Its Storage Class must be **Workgroup**, **CrossWorkgroup**, or **Function**.

Pointer must point to the **Generic** Storage Class.

Result Type and *Pointer* must point to the same type.

4	122	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpGenericCastToPtrExplicit

Capability:
Kernel

Attempts to explicitly convert *Pointer* to *Storage* storage-class pointer value.

Result Type must be an **OpTypePointer**. Its Storage Class must be *Storage*. *Pointer* must have a type of **OpTypePointer** whose *Type* is the same as the *Type* of *Result Type*.*Pointer* must point to the **Generic** Storage Class. If the cast fails, the instruction result is an **OpConstantNull** pointer in the *Storage* Storage Class. *Storage* must be one of the following literal values from Storage Class: **Workgroup**, **CrossWorkgroup**, or **Function**.					
5	123	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Storage Class* *Storage*

OpBitcast Bit pattern-preserving type conversion. *Result Type* must be an **OpTypePointer**, or a scalar or vector of *numerical-type*. *Operand* must have a type of **OpTypePointer**, or a scalar or vector of *numerical-type*. It must be a different type than *Result Type*. If *Result Type* is a pointer, *Operand* must be a pointer or integer scalar. If *Operand* is a pointer, *Result Type* must be a pointer or integer scalar. If *Result Type* has the same number of components as *Operand*, they must also have the same component width, and results are computed per component. If *Result Type* has a different number of components than *Operand*, the total number of bits in *Result Type* must equal the total number of bits in *Operand*. Let *L* be the type, either *Result Type* or *Operand's* type, that has the larger number of components. Let *S* be the other type, with the smaller number of components. The number of components in *L* must be an integer multiple of the number of components in *S*. The first component (that is, the only or lowest-numbered component) of *S* maps to the first components of *L*, and so on, up to the last component of *S* mapping to the last components of *L*. Within this mapping, any single component of *S* (mapping to multiple components of *L*) maps its lower-ordered bits to the lower-numbered components of *L*.				
4	124	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

3.32.12. Composite Instructions

OpVectorExtractDynamic

Extract a single, dynamically selected, component of a vector.

Result Type must be a scalar type.

Vector must have a type **OpTypeVector** whose *Component Type* is *Result Type*.

Index must be a scalar integer 0-based index of which component of *Vector* to extract.

The value read is undefined if *Index's* value is less than zero or greater than or equal to the number of components in *Vector*.

5	77	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*	*<id>* *Index*

OpVectorInsertDynamic

Make a copy of a vector, with a single, variably selected, component modified.

Result Type must be an **OpTypeVector**.

Vector must have the same type as *Result Type* and is the vector that the non-written components will be copied from.

Component is the value that will be supplied for the component selected by *Index*. It must have the same type as the type of components in *Result Type*.

Index must be a scalar integer 0-based index of which component to modify.

What is written is undefined if *Index's* value is less than zero or greater than or equal to the number of components in *Vector*.

6	78	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*	*<id>* *Component*	*<id>* *Index*

OpVectorShuffle

Select arbitrary components from two vectors to make a new vector.

Result Type must be an **OpTypeVector**. The number of components in *Result Type* must be the same as the number of *Component* operands.

Vector 1 and *Vector 2* must both have vector types, with the same *Component Type* as *Result Type*. They do not have to have the same number of components as *Result Type* or with each other. They are logically concatenated, forming a single vector with *Vector 1's* components appearing before *Vector 2's*. The components of this logical vector are logically numbered with a single consecutive set of numbers from 0 to *N* - 1, where *N* is the total number of components.

Components are these logical numbers (see above), selecting which of the logically numbered components form the result. They can select the components in any order and can repeat components. The first component of the result is selected by the first *Component* operand, the second component of the result is selected by the second *Component* operand, etc. A *Component literal* may also be FFFFFFFF, which means the corresponding result component has no source and is undefined. All *Component literals* must either be FFFFFFFF or in [0, *N* - 1] (inclusive).

Note: A vector "swizzle" can be done by using the vector for both *Vector* operands, or using an **OpUndef** for one of the *Vector* operands.

5 + variable	79	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector 1*	*<id>* *Vector 2*	*Literal, Literal, …* *Components*

OpCompositeConstruct

Construct a new *composite* object from a set of constituent objects that will fully form it.

Result Type must be a *composite* type, whose top-level members/elements/components/columns have the same type as the types of the operands, with one exception. The exception is that for constructing a vector, the operands may also be vectors with the same component type as the *Result Type* component type. When constructing a vector, the total number of components in all the operands must equal the number of components in *Result Type*.

Constituents will become members of a structure, or elements of an array, or components of a vector, or columns of a matrix. There must be exactly one *Constituent* for each top-level member/element /component/column of the result, with one exception. The exception is that for constructing a vector, a contiguous subset of the scalars consumed can be represented by a vector operand instead. The *Constituents* must appear in the order needed by the definition of the type of the result. When constructing a vector, there must be at least two *Constituent* operands.

3 + variable	80	*<id>* *Result Type*	*Result <id>*	*<id>, <id>, …* *Constituents*

OpCompositeExtract

Extract a part of a *composite* object.

Result Type must be the type of object selected by the last provided index. The instruction result is the extracted object.

Composite is the composite to extract from.

Indexes walk the type hierarchy, potentially down to component granularity, to select the part to extract. All indexes must be in bounds. All composite constituents use zero-based numbering, as described by their **OpType…** instruction.

4 + variable	81	*<id>* *Result Type*	*Result <id>*	*<id>* *Composite*	*Literal, Literal, …* *Indexes*

OpCompositeInsert

Make a copy of a *composite* object, while modifying one part of it.

Result Type must be the same type as *Composite*.

Object is the object to use as the modified part.

Composite is the composite to copy all but the modified part from.

Indexes walk the type hierarchy of *Composite* to the desired depth, potentially down to component granularity, to select the part to modify. All indexes must be in bounds. All composite constituents use zero-based numbering, as described by their **OpType…** instruction. The type of the part selected to modify must match the type of *Object*.

5 + variable	82	*<id>* *Result Type*	*Result <id>*	*<id>* *Object*	*<id>* *Composite*	*Literal, Literal, …* *Indexes*

OpCopyObject

Make a copy of *Operand*. There are no dereferences involved.

Result Type must match *Operand* type. There are no other restrictions on the types.

4	83	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpTranspose

Capability:
Matrix

Transpose a matrix.

Result Type must be an **OpTypeMatrix**, where the number of columns and the column size is the reverse of those of the type of *Matrix*.

Matrix must have of type of **OpTypeMatrix**.

4	84	*<id>* *Result Type*	*Result <id>*	*<id>* *Matrix*

3.32.13. Arithmetic Instructions

OpSNegate

Signed-integer subtract of *Operand* from zero.

Result Type must be a scalar or vector of *integer type*.

Operand's type must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*. The component width must equal the component width in *Result Type*.

Results are computed per component.

4	126	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpFNegate

Floating-point subtract of *Operand* from zero.

Result Type must be a scalar or vector of *floating-point type*.

The type of *Operand* must be the same as *Result Type*.

Results are computed per component.

4	127	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpIAdd

Integer addition of *Operand 1* and *Operand 2*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component.

5	128	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFAdd

Floating-point addition of *Operand 1* and *Operand 2*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component.

5	129	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpISub

Integer subtraction of *Operand 2* from *Operand 1*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component.

5	130	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFSub

Floating-point subtraction of *Operand 2* from *Operand 1*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component.

5	131	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpIMul

Integer multiplication of *Operand 1* and *Operand 2*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component.

5	132	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFMul

Floating-point multiplication of *Operand 1* and *Operand 2*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component.

5	133	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUDiv

Unsigned-integer division of *Operand 1* divided by *Operand 2*.

Result Type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	134	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSDiv

Signed-integer division of *Operand 1* divided by *Operand 2*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	135	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFDiv

Floating-point division of *Operand 1* divided by *Operand 2*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	136	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUMod

Unsigned modulo operation of *Operand 1* modulo *Operand 2*.

Result Type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	137	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSRem

Signed remainder operation of *Operand 1* divided by *Operand 2*. The sign of a non-0 result comes from *Operand 1*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	138	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSMod

Signed modulo operation of *Operand 1* modulo *Operand 2*. The sign of a non-0 result comes from *Operand 2*.

Result Type must be a scalar or vector of *integer type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	139	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFRem

Floating-point remainder operation of *Operand 1* divided by *Operand 2*. The sign of a non-0 result comes from *Operand 1*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	140	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFMod

Floating-point remainder operation of *Operand 1* divided by *Operand 2*. The sign of a non-0 result comes from *Operand 2*.

Result Type must be a scalar or vector of *floating-point type*.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component. The resulting value is undefined if *Operand 2* is 0.

5	141	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpVectorTimesScalar

Scale a floating-point vector. *Result Type* must be a vector of *floating-point type*. The type of *Vector* must be the same as *Result Type*. Each component of *Vector* is multiplied by *Scalar*. *Scalar* must have the same type as the *Component Type* in *Result Type*.					
5	142	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*	*<id>* *Scalar*

OpMatrixTimesScalar Scale a floating-point matrix. *Result Type* must be an **OpTypeMatrix** whose *Column Type* is a vector of *floating-point type*. The type of *Matrix* must be the same as *Result Type*. Each component in each column in *Matrix* is multiplied by *Scalar*. *Scalar* must have the same type as the *Component Type* in *Result Type*.					Capability: **Matrix**
5	143	*<id>* *Result Type*	*Result <id>*	*<id>* *Matrix*	*<id>* *Scalar*

OpVectorTimesMatrix Linear-algebraic *Vector X Matrix*. *Result Type* must be a vector of *floating-point type*. *Vector* must be a vector with the same *Component Type* as the *Component Type* in *Result Type*. Its number of components must equal the number of components in each column in *Matrix*. *Matrix* must be a matrix with the same *Component Type* as the *Component Type* in *Result Type*. Its number of columns must equal the number of components in *Result Type*.					Capability: **Matrix**
5	144	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*	*<id>* *Matrix*

OpMatrixTimesVector Linear-algebraic *Vector X Matrix*. *Result Type* must be a vector of *floating-point type*. *Matrix* must be an **OpTypeMatrix** whose *Column Type* is *Result Type*. *Vector* must be a vector with the same *Component Type* as the *Component Type* in *Result Type*. Its number of components must equal the number of columns in *Matrix*.					Capability: **Matrix**
5	145	*<id>* *Result Type*	*Result <id>*	*<id>* *Matrix*	*<id>* *Vector*

OpMatrixTimesMatrix Linear-algebraic multiply of *LeftMatrix* X *RightMatrix*. *Result Type* must be an **OpTypeMatrix** whose *Column Type* is a vector of *floating-point type*. *LeftMatrix* must be a matrix whose *Column Type* is the same as the *Column Type* in *Result Type*. *RightMatrix* must be a matrix with the same *Component Type* as the *Component Type* in *Result Type*. Its number of columns must equal the number of columns in *Result Type*. Its columns must have the same number of components as the number of columns in *LeftMatrix*.					Capability: **Matrix**
5	146	*<id>* *Result Type*	*Result <id>*	*<id>* *LeftMatrix*	*<id>* *RightMatrix*

OpOuterProduct Linear-algebraic outer product of *Vector 1* and *Vector 2*. *Result Type* must be an **OpTypeMatrix** whose *Column Type* is a vector of *floating-point type*. *Vector 1* must have the same type as the *Column Type* in *Result Type*. *Vector 2* must be a vector with the same *Component Type* as the *Component Type* in *Result Type*. Its number of components must equal the number of columns in *Result Type*.					Capability: **Matrix**
5	147	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector 1*	*<id>* *Vector 2*

OpDot Dot product of *Vector 1* and *Vector 2*. *Result Type* must be a *floating-point type* scalar. *Vector 1* and *Vector 2* must have the same type, and their component type must be *Result Type*.					
5	148	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector 1*	*<id>* *Vector 2*

OpIAddCarry

Result is the unsigned integer addition of *Operand 1* and *Operand 2*, including its carry.

Result Type must be from **OpTypeStruct**. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

Operand 1 and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as unsigned integers.

Results are computed per component.

Member 0 of the result gets the low-order bits (full component width) of the addition.

Member 1 of the result gets the high-order (carry) bit of the result of the addition. That is, it gets the value 1 if the addition overflowed the component width, and 0 otherwise.

5	149	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpISubBorrow

Result is the unsigned integer subtraction of *Operand 2* from *Operand 1*, and what it needed to borrow.

Result Type must be from **OpTypeStruct**. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

Operand 1 and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as unsigned integers.

Results are computed per component.

Member 0 of the result gets the low-order bits (full component width) of the subtraction. That is, if *Operand 1* is larger than *Operand 2*, member 0 gets the full value of the subtraction; if *Operand 2* is larger than *Operand 1*, member 0 gets 2^w + *Operand 1* - *Operand 2*, where w is the component width.

Member 1 of the result gets 0 if *Operand 1* ≥ *Operand 2*, and gets 1 otherwise.

5	150	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUMulExtended

Result is the full value of the unsigned integer multiplication of *Operand 1* and *Operand 2*.

Result Type must be from **OpTypeStruct**. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of *integer type*, whose *Signedness* operand is 0.

Operand 1 and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as unsigned integers.

Results are computed per component.

Member 0 of the result gets the low-order bits of the multiplication.

Member 1 of the result gets the high-order bits of the multiplication.

5	151	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSMulExtended

Result is the full value of the signed integer multiplication of *Operand 1* and *Operand 2*.

Result Type must be from **OpTypeStruct**. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of *integer type*.

Operand 1 and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as signed integers.

Results are computed per component.

Member 0 of the result gets the low-order bits of the multiplication.

Member 1 of the result gets the high-order bits of the multiplication.

5	152	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

3.32.14. Bit Instructions

OpShiftRightLogical

Shift the bits in *Base* right by the number of bits specified in *Shift*. The most-significant bits will be zero filled.

Result Type must be a scalar or vector of *integer type*.

The type of each *Base* and *Shift* must be a scalar or vector of *integer type*. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*.

Shift is consumed as an unsigned integer. The result is undefined if *Shift* is greater than the bit width of the components of *Base*.

Results are computed per component.

5	194	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Shift*

OpShiftRightArithmetic

Shift the bits in *Base* right by the number of bits specified in *Shift*. The most-significant bits will be filled with the sign bit from *Base*.

Result Type must be a scalar or vector of *integer type*.

The type of each *Base* and *Shift* must be a scalar or vector of *integer type*. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*.

Shift is treated as unsigned. The result is undefined if *Shift* is greater than the bit width of the components of *Base*.

Results are computed per component.

5	195	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Shift*

OpShiftLeftLogical

Shift the bits in *Base* left by the number of bits specified in *Shift*. The least-significant bits will be zero filled.

Result Type must be a scalar or vector of *integer type*.

The type of each *Base* and *Shift* must be a scalar or vector of *integer type*. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*.

Shift is treated as unsigned. The result is undefined if *Shift* is greater than the bit width of the components of *Base*.

The number of components and bit width of *Result Type* must match those *Base* type. All types must be integer types.

Results are computed per component.

5	196	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Shift*

OpBitwiseOr

Result is 1 if either *Operand 1* or *Operand 2* is 1. Result is 0 if both *Operand 1* and *Operand 2* are 0.

Results are computed per component, and within each component, per bit.

Result Type must be a scalar or vector of *integer type*. The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

5	197	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpBitwiseXor

Result is 1 if exactly one of *Operand 1* or *Operand 2* is 1. Result is 0 if *Operand 1* and *Operand 2* have the same value.

Results are computed per component, and within each component, per bit.

Result Type must be a scalar or vector of *integer type*. The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

5	198	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpBitwiseAnd

Result is 1 if both *Operand 1* and *Operand 2* are 1. Result is 0 if either *Operand 1* or *Operand 2* are 0.

Results are computed per component, and within each component, per bit.

Result Type must be a scalar or vector of *integer type*. The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*.

5	199	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpNot

Complement the bits of *Operand*.

Results are computed per component, and within each component, per bit.

Result Type must be a scalar or vector of *integer type*.

Operand's type must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*. The component width must equal the component width in *Result Type*.

4	200	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpBitFieldInsert

Make a copy of an object, with a modified bit field that comes from another object.

Capability:
Shader

<table>
<tr><td colspan="6">Results are computed per component.

Result Type must be a scalar or vector of integer type.

The type of Base and Insert must be the same as Result Type.

Any result bits numbered outside [Offset, Offset + Count - 1] (inclusive) will come from the corresponding bits in Base.

Any result bits numbered in [Offset, Offset + Count - 1] come, in order, from the bits numbered [0, Count - 1] of Insert.

Count must be an integer type scalar. Count is the number of bits taken from Insert. It will be consumed as an unsigned value. Count can be 0, in which case the result will be Base.

Offset must be an integer type scalar. Offset is the lowest-order bit of the bit field. It will be consumed as an unsigned value.

The resulting value is undefined if Count or Offset or their sum is greater than the number of bits in the result.</td><td colspan="2"></td></tr>
<tr><td>7</td><td>201</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Base</td><td><id>
Insert</td><td><id>
Offset</td><td><id>
Count</td></tr>
</table>

OpBitFieldSExtract

Extract a bit field from an object, with sign extension.

Results are computed per component.

Result Type must be a scalar or vector of *integer type*.

The type of *Base* must be the same as *Result Type*.

If *Count* is greater than 0: The bits of *Base* numbered in [*Offset*, *Offset* + *Count* - 1] (inclusive) become the bits numbered [0, *Count* - 1] of the result. The remaining bits of the result will all be the same as bit *Offset + Count - 1* of *Base*.

Count must be an *integer type* scalar. *Count* is the number of bits extracted from *Base*. It will be consumed as an unsigned value. *Count* can be 0, in which case the result will be 0.

Offset must be an *integer type* scalar. *Offset* is the lowest-order bit of the bit field to extract from *Base*. It will be consumed as an unsigned value.

The resulting value is undefined if *Count* or *Offset* or their sum is greater than the number of bits in the result.

Capability:
Shader

6	202	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Offset*	*<id>* *Count*

OpBitFieldUExtract

Extract a bit field from an object, without sign extension.

The semantics are the same as with **OpBitFieldSExtract** with the exception that there is no sign extension. The remaining bits of the result will all be 0.

Capability:
Shader

6	203	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*	*<id>* *Offset*	*<id>* *Count*

OpBitReverse

Reverse the bits in an object.

Results are computed per component.

Result Type must be a scalar or vector of *integer type*.

The type of *Base* must be the same as *Result Type*.

The bit-number *n* of the result will be taken from bit-number *Width - 1 - n* of *Base*, where *Width* is the **OpTypeInt** operand of the *Result Type*.

Capability:
Shader

4	204	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*

OpBitCount

Count the number of set bits in an object.

Results are computed per component.

Result Type must be a scalar or vector of *integer type*. The components must be wide enough to hold the unsigned *Width* of *Base* as an unsigned value. That is, no sign bit is needed or counted when checking for a wide enough result width.

Base must be a scalar or vector of *integer type*. It must have the same number of components as *Result Type*.

The result is the unsigned value that is the number of bits in *Base* that are 1.

4	205	*<id>* *Result Type*	*Result <id>*	*<id>* *Base*

3.32.15. Relational and Logical Instructions

OpAny

Result is **true** if any component of *Vector* is **true**, otherwise result is **false**.

Result Type must be a *Boolean type* scalar.

Vector must be a vector of *Boolean type*.

4	154	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*

OpAll

Result is **true** if all components of *Vector* are **true**, otherwise result is **false**.

Result Type must be a *Boolean type* scalar.

Vector must be a vector of *Boolean type*.

4	155	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector*

OpIsNan

Result is **true** if *x* is an IEEE NaN, otherwise result is **false**.

Result Type must be a scalar or vector of *Boolean type*.

x must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	156	*<id>* *Result Type*	*Result <id>*	*<id>* *x*

OpIsInf

Result is **true** if *x* is an IEEE Inf, otherwise result is **false**

Result Type must be a scalar or vector of *Boolean type*.

x must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*.

Results are computed per component.

4	157	*<id>* *Result Type*	*Result <id>*	*<id>* *x*

OpIsFinite Result is **true** if *x* is an IEEE finite number, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. Results are computed per component.				Capability: **Kernel**
4	158	*<id>* *Result Type*	*Result <id>*	*<id>* *x*

OpIsNormal Result is **true** if *x* is an IEEE normal number, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. Results are computed per component.				Capability: **Kernel**
4	159	*<id>* *Result Type*	*Result <id>*	*<id>* *x*

OpSignBitSet Result is **true** if *x* has its sign bit set, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. Results are computed per component.				Capability: **Kernel**
4	160	*<id>* *Result Type*	*Result <id>*	*<id>* *x*

OpLessOrGreater Result is **true** if *x* < *y* or *x* > *y*, where IEEE comparisons are used, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. *y* must have the same type as *x*. Results are computed per component.					Capability: **Kernel**
5	161	*<id>* *Result Type*	*Result <id>*	*<id>* *x*	*<id>* *y*

OpOrdered Result is **true** if both *x* == *x* and *y* == *y* are **true**, where IEEE comparison is used, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. *y* must have the same type as *x*. Results are computed per component.					Capability: **Kernel**
5	162	*<id>* *Result Type*	*Result <id>*	*<id>* *x*	*<id>* *y*

OpUnordered Result is **true** if either *x* or *y* is an IEEE NaN, otherwise result is **false**. *Result Type* must be a scalar or vector of *Boolean type*. *x* must be a scalar or vector of *floating-point type*. It must have the same number of components as *Result Type*. *y* must have the same type as *x*. Results are computed per component.					Capability: **Kernel**
5	163	*<id>* *Result Type*	*Result <id>*	*<id>* *x*	*<id>* *y*

OpLogicalEqual

Result is **true** if *Operand 1* and *Operand 2* have the same value. Result is **false** if *Operand 1* and *Operand 2* have different values.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* must be the same as *Result Type*.

The type of *Operand 2* must be the same as *Result Type*.

Results are computed per component.

5	164	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpLogicalNotEqual

Result is **true** if *Operand 1* and *Operand 2* have different values. Result is **false** if *Operand 1* and *Operand 2* have the same value.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* must be the same as *Result Type*.

The type of *Operand 2* must be the same as *Result Type*.

Results are computed per component.

5	165	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpLogicalOr

Result is **true** if either *Operand 1* or *Operand 2* is **true**. Result is **false** if both *Operand 1* and *Operand 2* are **false**.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* must be the same as *Result Type*.

The type of *Operand 2* must be the same as *Result Type*.

Results are computed per component.

5	166	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpLogicalAnd

Result is **true** if both *Operand 1* and *Operand 2* are **true**. Result is **false** if either *Operand 1* or *Operand 2* are **false**.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* must be the same as *Result Type*.

The type of *Operand 2* must be the same as *Result Type*.

Results are computed per component.

5	167	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpLogicalNot

Result is **true** if *Operand* is **false**. Result is **false** if *Operand* is **true**.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand* must be the same as *Result Type*.

Results are computed per component.

4	168	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpSelect

Select between two objects.

Result Type must be a scalar or vector.

The type of *Object 1* must be the same as *Result Type*. *Object 1* is selected as the result if *Condition* is **true**.

The type of *Object 2* must be the same as *Result Type*. *Object 2* is selected as the result if *Condition* is **false**.

Condition must be a scalar or vector of *Boolean type*. It must have the same number of components as *Result Type*.

Results are computed per component.

6	169	*<id>* *Result Type*	*Result <id>*	*<id>* *Condition*	*<id>* *Object 1*	*<id>* *Object 2*

OpIEqual

Integer comparison for equality.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	170	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpINotEqual

Integer comparison for inequality.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	171	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUGreaterThan

Unsigned-integer comparison if *Operand 1* is greater than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	172	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSGreaterThan

Signed-integer comparison if *Operand 1* is greater than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	173	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUGreaterThanEqual

Unsigned-integer comparison if *Operand 1* is greater than or equal to *Operand 2*.

Result Type must be a scalar or vector of Boolean type.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	174	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSGreaterThanEqual

Signed-integer comparison if *Operand 1* is greater than or equal to *Operand 2*.

Result Type must be a scalar or vector of Boolean type.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	175	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpULessThan

Unsigned-integer comparison if *Operand 1* is less than *Operand 2*.

Result Type must be a scalar or vector of Boolean type.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	176	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSLessThan

Signed-integer comparison if *Operand 1* is less than *Operand 2*.

Result Type must be a scalar or vector of Boolean type.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	177	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpULessThanEqual

Unsigned-integer comparison if *Operand 1* is less than or equal to *Operand 2*.

Result Type must be a scalar or vector of Boolean type.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	178	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSLessThanEqual

Signed-integer comparison if *Operand 1* is less than or equal to *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *integer type*. They must have the same component width, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	179	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdEqual

Floating-point comparison for being ordered and equal.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	180	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordEqual

Floating-point comparison for being unordered or equal.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	181	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdNotEqual

Floating-point comparison for being ordered and not equal.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.					
5	182	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordNotEqual

Floating-point comparison for being unordered or not equal.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	183	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdLessThan

Floating-point comparison if operands are ordered and *Operand 1* is less than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	184	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordLessThan

Floating-point comparison if operands are unordered or *Operand 1* is less than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	185	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdGreaterThan

Floating-point comparison if operands are ordered and *Operand 1* is greater than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	186	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordGreaterThan

Floating-point comparison if operands are unordered or *Operand 1* is greater than *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	187	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdLessThanEqual

Floating-point comparison if operands are ordered and *Operand 1* is less than or equal to *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	188	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordLessThanEqual

Floating-point comparison if operands are unordered or *Operand 1* is less than or equal to *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	189	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFOrdGreaterThanEqual

Floating-point comparison if operands are ordered and *Operand 1* is greater than or equal to *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	190	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFUnordGreaterThanEqual

Floating-point comparison if operands are unordered or *Operand 1* is greater than or equal to *Operand 2*.

Result Type must be a scalar or vector of *Boolean type*.

The type of *Operand 1* and *Operand 2* must be a scalar or vector of *floating-point type*. They must have the same type, and they must have the same number of components as *Result Type*.

Results are computed per component.

5	191	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

3.32.16. Derivative Instructions

OpDPdx

Same result as either **OpDPdxFine** or **OpDPdxCoarse** on *P*. Selection of which one is based on external factors.

Result Type must be a scalar or vector of *floating-point type*.

The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of.

This instruction is only valid in the **Fragment** Execution Model.

Capability:
Shader

4	207	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

OpDPdy

Same result as either **OpDPdyFine** or **OpDPdyCoarse** on *P*. Selection of which one is based on external factors.

Result Type must be a scalar or vector of *floating-point type*.

The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of.

This instruction is only valid in the **Fragment** Execution Model.

Capability:
Shader

4	208	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

OpFwidth

Result is the same as computing the sum of the absolute values of **OpDPdx** and **OpDPdy** on *P*.

Result Type must be a scalar or vector of *floating-point type*.

The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of.

This instruction is only valid in the **Fragment** Execution Model.

Capability:
Shader

4	209	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

<table>
<tr><td colspan="4">OpDPdxFine

Result is the partial derivative of P with respect to the window x coordinate.Will use local differencing based on the value of P for the current fragment and its immediate neighbor(s).

Result Type must be a scalar or vector of floating-point type.

The type of P must be the same as Result Type. P is the value to take the derivative of.

This instruction is only valid in the Fragment Execution Model.</td><td>Capability:
DerivativeControl</td></tr>
<tr><td>4</td><td>210</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

<table>
<tr><td colspan="4">OpDPdyFine

Result is the partial derivative of P with respect to the window y coordinate.Will use local differencing based on the value of P for the current fragment and its immediate neighbor(s).

Result Type must be a scalar or vector of floating-point type.

The type of P must be the same as Result Type. P is the value to take the derivative of.

This instruction is only valid in the Fragment Execution Model.</td><td>Capability:
DerivativeControl</td></tr>
<tr><td>4</td><td>211</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

<table>
<tr><td colspan="4">OpFwidthFine

Result is the same as computing the sum of the absolute values of OpDPdxFine and OpDPdyFine on P.

Result Type must be a scalar or vector of floating-point type.

The type of P must be the same as Result Type. P is the value to take the derivative of.

This instruction is only valid in the Fragment Execution Model.</td><td>Capability:
DerivativeControl</td></tr>
<tr><td>4</td><td>212</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

OpDPdxCoarse Result is the partial derivative of *P* with respect to the window *x* coordinate. Will use local differencing based on the value of *P* for the current fragment's neighbors, and will possibly, but not necessarily, include the value of *P* for the current fragment. That is, over a given area, the implementation can compute *x* derivatives in fewer unique locations than would be allowed for **OpDPdxFine**. *Result Type* must be a scalar or vector of *floating-point type*. The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of. This instruction is only valid in the **Fragment** Execution Model.				Capability: **DerivativeControl**
4	213	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

OpDPdyCoarse Result is the partial derivative of *P* with respect to the window *y* coordinate. Will use local differencing based on the value of *P* for the current fragment's neighbors, and will possibly, but not necessarily, include the value of *P* for the current fragment. That is, over a given area, the implementation can compute *y* derivatives in fewer unique locations than would be allowed for **OpDPdyFine**. *Result Type* must be a scalar or vector of *floating-point type*. The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of. This instruction is only valid in the **Fragment** Execution Model.				Capability: **DerivativeControl**
4	214	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

OpFwidthCoarse Result is the same as computing the sum of the absolute values of **OpDPdxCoarse** and **OpDPdyCoarse** on *P*. *Result Type* must be a scalar or vector of *floating-point type*. The type of *P* must be the same as *Result Type*. *P* is the value to take the derivative of. This instruction is only valid in the **Fragment** Execution Model.				Capability: **DerivativeControl**
4	215	*<id>* *Result Type*	*Result <id>*	*<id>* *P*

3.32.17. Control-Flow Instructions

OpPhi

The SSA phi function.

The result is selected based on control flow: If control reached the current block from *Parent i*, *Result Id* gets the value that *Variable i* had at the end of *Parent i*.

Result Type can be any type.

Operands are a sequence of pairs: (*Variable 1*, *Parent 1* block), (*Variable 2*, *Parent 2* block), … Each *Parent i* block is the label of an immediate predecessor in the CFG of the current block. A *Parent i* block must not appear more than once in the operand sequence. All *Variables* must have a type matching *Result Type*.

Within a block, this instruction must appear before all non-**OpPhi** instructions (except for **OpLine**, which can be mixed with **OpPhi**).

3 + variable	245	*<id>* *Result Type*	*Result <id>*	*<id>, <id>, …* *Variable, Parent, …*

OpLoopMerge

Declare a structured loop.

This instruction must immediately precede either an **OpBranch** or **OpBranchConditional** instruction. That is, it must be the second-to-last instruction in its block.

Merge Block is the label of the merge block for this structured loop.

Continue Target is the label of a block targeted for processing a loop "continue".

Loop Control Parameters appear in Loop Control-table order for any *Loop Control* setting that requires such a parameter.

See Structured Control Flow for more detail.

4 + variable	246	*<id>* *Merge Block*	*<id>* *Continue Target*	*Loop Control*	*Literal, Literal, …* *Loop Control Parameters*

OpSelectionMerge

Declare a structured selection.

This instruction must immediately precede either an **OpBranchConditional** or **OpSwitch** instruction. That is, it must be the second-to-last instruction in its block.

Merge Block is the label of the merge block for this structured selection.

See Structured Control Flow for more detail.

3	247	*<id>* *Merge Block*	*Selection Control*

OpLabel

The block label instruction: Any reference to a block is through the *Result <id>* of its label.

Must be the first instruction of any block, and appears only as the first instruction of a block.

2	248	*Result <id>*

OpBranch

Unconditional branch to *Target Label*.

Target Label must be the *Result <id>* of an **OpLabel** instruction in the current function.

This instruction must be the last instruction in a block.

2	249	*<id>* *Target Label*

OpBranchConditional

If *Condition* is **true**, branch to *True Label*, otherwise branch to *False Label*.

Condition must be a *Boolean type* scalar.

True Label must be an **OpLabel** in the current function.

False Label must be an **OpLabel** in the current function.

Branch weights are unsigned 32-bit integer literals. There must be either no *Branch Weights* or exactly two branch weights. If present, the first is the weight for branching to *True Label*, and the second is the weight for branching to *False Label*. The implied probability that a branch is taken is its weight divided by the sum of the two *Branch weights*.

This instruction must be the last instruction in a block.

4 + variable	250	*<id>* *Condition*	*<id>* *True Label*	*<id>* *False Label*	*Literal, Literal, …* *Branch weights*

OpSwitch

Multi-way branch to one of the operand label *<id>*.

Selector must have a type of **OpTypeInt**. *Selector* will be compared for equality to the *Target* literals.

Default must be the *<id>* of a label. If *Selector* does not equal any of the *Target* literals, control flow will branch to the *Default* label *<id>*.

Target must be alternating scalar integer *literals* and the *<id>* of a label. If *Selector* equals a *literal*, control flow will branch to the following *label <id>*. It is invalid for any two *literal* to be equal to each other. If *Selector* does not equal any *literal*, control flow will branch to the *Default* label *<id>*. Each *literal* is interpreted with the type of *Selector*: The bit width of *Selector's* type will be the width of each *literal's* type. If this width is not a multiple of 32-bits, the literals must be sign extended when the **OpTypeInt** *Signedness* is set to 1. (See *Literal Number*.)

This instruction must be the last instruction in a block.

3 + variable	251	*<id>* *Selector*	*<id>* *Default*	*literal, label <id>,* *literal, label <id>,* *…* *Target*

OpKill

Capability:
Shader

Fragment-shader discard.

Ceases all further processing in any invocation that executes it: Only instructions these invocations executed before **OpKill** will have observable side effects. If this instruction is executed in non-uniform control flow, all subsequent control flow is non-uniform (for invocations that continue to execute).

This instruction must be the last instruction in a block.

This instruction is only valid in the **Fragment** Execution Model.

1	252

OpReturn

Return with no value from a function with void return type.

This instruction must be the last instruction in a block.

1	253

OpReturnValue Return a value from a function. *Value* is the value returned, by copy, and must match the *Return Type* operand of the **OpTypeFunction** type of the **OpFunction** body this return instruction is in. This instruction must be the last instruction in a block.		
2	254	*<id>* *Value*

OpUnreachable Declares that this block is not reachable in the CFG. This instruction must be the last instruction in a block.	
1	255

OpLifetimeStart Declare that an object was not defined before this instruction. *Pointer* is a pointer to the object whose lifetime is starting. Its type must be an **OpTypePointer** with Storage Class **Function**. *Size* must be 0 if *Pointer* is a pointer to a non-void type or the **Addresses** capability is not being used. If *Size* is non-zero, it is the number of bytes of memory whose lifetime is starting. Its type must be an *integer type* scalar. It is treated as unsigned; if its type has *Signedness* of 1, its sign bit cannot be set.			Capability: **Kernel**
3	256	*<id>* *Pointer*	*Literal Number* *Size*

OpLifetimeStop Declare that an object is dead after this instruction. *Pointer* is a pointer to the object whose lifetime is ending. Its type must be an **OpTypePointer** with Storage Class **Function**. *Size* must be 0 if *Pointer* is a pointer to a non-void type or the **Addresses** capability is not being used. If *Size* is non-zero, it is the number of bytes of memory whose lifetime is ending. Its type must be an *integer type* scalar. It is treated as unsigned; if its type has *Signedness* of 1, its sign bit cannot be set.			Capability: **Kernel**
3	257	*<id>* *Pointer*	*Literal Number* *Size*

3.32.18. Atomic Instructions

OpAtomicLoad

Atomically load through *Pointer* using the given *Semantics*. All subparts of the value that is loaded will be read atomically with respect to all other atomic accesses to it within *Scope*.

Result Type must be a scalar of *integer type* or *floating-point type*.

Pointer is the pointer to the memory to read. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

6	227	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*

OpAtomicStore

Atomically store through *Pointer* using the given *Semantics*. All subparts of *Value* will be written atomically with respect to all other atomic accesses to it within *Scope*.

Pointer is the pointer to the memory to write. The type it points to must be a scalar of *integer type* or *floating-point type*.

Value is the value to write. The type of *Value* and the type pointed to by *Pointer* must be the same type.

5	228	<id> *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	<id> *Value*

OpAtomicExchange

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* from copying *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be a scalar of *integer type* or *floating-point type*.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	229	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	<id> *Value*

OpAtomicCompareExchange

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by selecting *Value* if *Original Value* equals *Comparator* or selecting *Original Value* otherwise, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

Use *Equal* for the memory semantics of this instruction when *Value* and *Original Value* compare equal.

Use *Unequal* for the memory semantics of this instruction when *Value* and *Original Value* compare unequal. *Unequal* cannot be set to **Release** or **Acquire and Release**. In addition, *Unequal* cannot be set to a stronger memory-order then *Equal*.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*. This type must also match the type of *Comparator*.

9	230	*<id> Result Type*	*Result <id>*	*<id> Pointer*	*Scope <id> Scope*	*Memory Semantics <id> Equal*	*Memory Semantics <id> Unequal*	*<id> Value*	*<id> Comparator*

<table>
<tr><td colspan="8">OpAtomicCompareExchangeWeak

Attempts to do the following:

Perform the following steps atomically with respect to any other atomic accesses within Scope to the same location:
1) load through Pointer to get an Original Value,
2) get a New Value by selecting Value if Original Value equals Comparator or selecting Original Value otherwise, and
3) store the New Value back through Pointer.

The instruction's result is the Original Value.

The weak compare-and-exchange operations may fail spuriously. That is, even when Original Value equals Comparator the comparison can fail and store back the Original Value through Pointer.

Result Type must be an integer type scalar.

Use Equal for the memory semantics of this instruction when Value and Original Value compare equal.

Use Unequal for the memory semantics of this instruction when Value and Original Value compare unequal. Unequal cannot be set to Release or Acquire and Release. In addition, Unequal cannot be set to a stronger memory-order then Equal.

The type of Value must be the same as Result Type. The type of the value pointed to by Pointer must be the same as Result Type. This type must also match the type of Comparator.</td><td colspan="2">Capability:
Kernel</td></tr>
<tr><td>9</td><td>231</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td><td>Scope <id>
Scope</td><td>Memory Semantics <id>
Equal</td><td>Memory Semantics <id>
Unequal</td><td><id>
Value</td><td><id>
Comparator</td></tr>
</table>

OpAtomicIIncrement

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* through integer addition of *1* to *Original Value*, and
3) store the *New Value* back through *Pointer*.

The instruction’s result is the *Original Value*.

Result Type must be an *integer type* scalar. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

6	232	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*

OpAtomicIDecrement

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* through integer subtraction of *1* from *Original Value*, and
3) store the *New Value* back through *Pointer*.

The instruction’s result is the *Original Value*.

Result Type must be an *integer type* scalar. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

6	233	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*

OpAtomicIAdd

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by integer addition of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction’s result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	234	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicISub

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by integer subtraction of *Value* from *Original Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	235	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicSMin

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by finding the smallest signed integer of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	236	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicUMin

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by finding the smallest unsigned integer of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	237	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicSMax

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by finding the largest signed integer of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	238	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicUMax

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by finding the largest unsigned integer of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	239	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicAnd

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by the bitwise AND of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	240	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicOr

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by the bitwise OR of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	241	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicXor

Perform the following steps atomically with respect to any other atomic accesses within *Scope* to the same location:
1) load through *Pointer* to get an *Original Value*,
2) get a *New Value* by the bitwise exclusive OR of *Original Value* and *Value*, and
3) store the *New Value* back through *Pointer*.

The instruction's result is the *Original Value*.

Result Type must be an *integer type* scalar.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

7	242	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Scope*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

<table>
<tr><td colspan="5">OpAtomicFlagTestAndSet

Atomically sets the flag value pointed to by Pointer to the set state.

Pointer must be a pointer to a 32-bit integer type representing an atomic flag.

The instruction’s result is true if the flag was in the set state or false if the flag was in the clear state immediately before the operation.

Result Type must be a Boolean type.

Results are undefined if an atomic flag is modified by an instruction other than OpAtomicFlagTestAndSet or OpAtomicFlagClear</td><td colspan="2">Capability:
Kernel</td></tr>
<tr><td>6</td><td>318</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td><td>Scope <id>
Scope</td><td>Memory Semantics <id>
Semantics</td></tr>
</table>

<table>
<tr><td colspan="4">OpAtomicFlagClear

Atomically sets the flag value pointed to by Pointer to the clear state.

Pointer must be a pointer to a 32-bit integer type representing an atomic flag.

Memory Semantics cannot be Acquire or AcquireRelease

Results are undefined if an atomic flag is modified by an instruction other than OpAtomicFlagTestAndSet or OpAtomicFlagClear</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>319</td><td><id>
Pointer</td><td>Scope <id>
Scope</td><td>Memory Semantics <id>
Semantics</td></tr>
</table>

3.32.19. Primitive Instructions

OpEmitVertex Emits the current values of all output variables to the current output primitive. After execution, the values of all output variables are undefined. This instruction can only be used when only one stream is present.	Capability: **Geometry**
1	218

OpEndPrimitive Finish the current primitive and start a new one. No vertex is emitted. This instruction can only be used when only one stream is present.	Capability: **Geometry**
1	219

OpEmitStreamVertex Emits the current values of all output variables to the current output primitive. After execution, the values of all output variables are undefined. *Stream* must be an *<id>* of a *constant instruction* with a scalar integer type. That constant is the output-primitive stream number. This instruction can only be used when multiple streams are present.		Capability: **GeometryStreams**
2	220	*<id>* *Stream*

OpEndStreamPrimitive Finish the current primitive and start a new one. No vertex is emitted. *Stream* must be an *<id>* of a *constant instruction* with a scalar integer type. That constant is the output-primitive stream number. This instruction can only be used when multiple streams are present.		Capability: **GeometryStreams**
2	221	*<id>* *Stream*

3.32.20. Barrier Instructions

OpControlBarrier

Wait for other invocations of this module to reach the current point of execution.

All invocations of this module within *Execution* scope must reach this point of execution before any invocation will proceed beyond it.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

If *Semantics* is not **None**, this instruction also serves as an **OpMemoryBarrier** instruction, and must also perform and adhere to the description and semantics of an **OpMemoryBarrier** instruction with the same *Memory* and *Semantics* operands. This allows atomically specifying both a control barrier and a memory barrier (that is, without needing two instructions). If *Semantics* is **None**, *Memory* is ignored.

It is only valid to use this instruction with **TessellationControl**, **GLCompute**, or **Kernel** execution models.

When used with the **TessellationControl** execution model, it also implicitly synchronizes the **Output** Storage Class: Writes to **Output** variables performed by any invocation executed prior to a **OpControlBarrier** will be visible to any other invocation after return from that **OpControlBarrier**.

4	224	*Scope <id>* *Execution*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

OpMemoryBarrier

Control the order that memory accesses are observed.

Ensures that memory accesses issued before this instruction will be observed before memory accesses issued after this instruction. This control is ensured only for memory accesses issued by this invocation and observed by another invocation executing within *Memory* scope.

Semantics declares what kind of memory is being controlled and what kind of control to apply.

To execute both a memory barrier and a control barrier, see **OpControlBarrier**.

3	225	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

OpNamedBarrierInitialize

Capability:
NamedBarrier

Declare a new named-barrier object.

Result Type must be the type **OpTypeNamedBarrier**.

Subgroup Count must be a 32-bit *integer type* scalar representing the number of subgroups that must reach the current point of execution.

4	328	*<id>* *Result Type*	*Result <id>*	*<id>* *Subgroup Count*

OpMemoryNamedBarrier

Capability:
NamedBarrier

Wait for other invocations of this module to reach the current point of execution.

Named Barrier must be the type **OpTypeNamedBarrier**.

If *Semantics* is not **None**, this instruction also serves as an **OpMemoryBarrier** instruction, and must also perform and adhere to the description and semantics of an **OpMemoryBarrier** instruction with the same *Memory* and *Semantics* operands. This allows atomically specifying both a control barrier and a memory barrier (that is, without needing two instructions). If *Semantics* **None**, *Memory* is ignored.				
4	329	*<id>* *Named Barrier*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

3.32.21. Group Instructions

OpGroupAsyncCopy

Capability:
Kernel

Perform an asynchronous group copy of *Num Elements* elements from *Source* to *Destination*. The asynchronous copy is performed by all work-items in a group.

This instruction returns an event object that can be used by **OpGroupWaitEvents** to wait for the async copy to finish.

All invocations of this module within *Execution* must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be an **OpTypeEvent** object.

Destination must be a pointer to a scalar or vector of *floating-point type* or *integer type*.

Destination pointer Storage Class must be **Workgroup** or **CrossWorkgroup**.

The type of *Source* must be the same as *Destination*.

When *Destination* pointer Storage Class is **Workgroup**, the *Source* pointer Storage Class must be **CrossWorkgroup**. In this case *Stride* defines the stride in elements when reading from *Source* pointer.

When *Destination* pointer Storage Class is **CrossWorkgroup**, the *Source* pointer Storage Class must be **Workgroup**. In this case *Stride* defines the stride in elements when writing each element to *Destination* pointer.

Stride and *NumElements* must be a 32-bit *integer type* scalar when the addressing model is *Physical32* and 64 bit *integer type* scalar when the *Addressing Model* is *Physical64*.

Event must have a type of **OpTypeEvent**.

Event can be used to associate the copy with a previous copy allowing an event to be shared by multiple copies. Otherwise *Event* should be an **OpConstantNull**.

If *Event* argument is not **OpConstantNull**, the event object supplied in event argument will be returned.

9	259	<id> *Result Type*	*Result <id>*	*Scope <id> Execution*	<id> *Destination*	<id> *Source*	<id> *Num Elements*	<id> *Stride*	<id> *Event*

<table>
<tr><td colspan="4">OpGroupWaitEvents

Wait for events generated by OpGroupAsyncCopy operations to complete. Events List points to Num Events event objects, which will be released after the wait is performed.

All invocations of this module within Execution must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within Execution. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Execution must be Workgroup or Subgroup Scope.

Num Events must be a 32-bit integer type scalar.

Events List must be a pointer to OpTypeEvent.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>260</td><td>Scope <id>
Execution</td><td><id>
Num Events</td><td><id>
Events List</td></tr>
</table>

<table>
<tr><td colspan="5">OpGroupAll

Evaluates a predicate for all invocations in the group,resulting in true if predicate evaluates to true for all invocations in the group, otherwise the result is false.

All invocations of this module within Execution must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within Execution. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be a Boolean type.

Execution must be Workgroup or Subgroup Scope.

Predicate must be a Boolean type.</td><td>Capability:
Groups</td></tr>
<tr><td>5</td><td>261</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Predicate</td></tr>
</table>

OpGroupAny

Evaluates a predicate for all invocations in the group,resulting in **true** if predicate evaluates to **true** for any invocation in the group, otherwise the result is **false**.

All invocations of this module within *Execution* must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be a *Boolean type*.

Execution must be **Workgroup** or **Subgroup** *Scope*.

Predicate must be a *Boolean type*.

Capability: **Groups**

5	262	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Predicate*

OpGroupBroadcast

Return the *Value* of the invocation identified by the local id *LocalId* to all invocations in the group.

All invocations of this module within *Execution* must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be a 32-bit or 64-bit *integer type* or a 16, 32 or 64 *float type* scalar.

Execution must be **Workgroup** or **Subgroup** *Scope*.

The type of *Value* must be the same as *Result Type*.

LocalId must be an integer datatype. It can be a scalar, or a vector with 2 components or a vector with 3 components. *LocalId* must be the same for all invocations in the group.

Capability: **Groups**

6	263	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Value*	*<id>* *LocalId*

OpGroupIAdd An integer add group operation specified for all values of *X* specified by invocations in the group. The identity *I* is 0. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be a 32-bit or 64-bit *integer type* scalar. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	264	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupFAdd A floating-point add group operation specified for all values of *X* specified by invocations in the group. The identity *I* is 0. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be a 16-bit, 32-bit, or 64-bit *floating-point type* scalar. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	265	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

<table>
<tr><td colspan="5">OpGroupFMin

A floating-point minimum group operation specified for all values of X specified by invocations in the group.

The identity I is +INF.

All invocations of this module within Execution must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within Execution. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be a 16-bit, 32-bit, or 64-bit floating-point type scalar.

Execution must be Workgroup or Subgroup Scope.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>266</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

<table>
<tr><td colspan="5">OpGroupUMin

An unsigned integer minimum group operation specified for all values of X specified by invocations in the group.

The identity I is UINT_MAX when X is 32 bits wide and ULONG_MAX when X is 64 bits wide.

All invocations of this module within Execution must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within Execution. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be a 32-bit or 64-bit integer type scalar.

Execution must be Workgroup or Subgroup Scope.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>267</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

OpGroupSMin A signed integer minimum group operation specified for all values of *X* specified by invocations in the group. The identity *I* is INT_MAX when *X* is 32 bits wide and LONG_MAX when *X* is 64 bits wide. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be a 32-bit or 64-bit *integer type* scalar. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	268	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* X

OpGroupFMax A floating-point maximum group operation specified for all values of *X* specified by invocations in the group. The identity *I* is -INF. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be a 16-bit, 32-bit, or 64-bit *floating-point type* scalar. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	269	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* X

OpGroupUMax An unsigned integer maximum group operation specified for all values of *X* specified by invocations in the group. The identity *I* is 0. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be a 32-bit or 64-bit *integer type* scalar. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	270	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* X

OpGroupSMax A signed integer maximum group operation specified for all values of *X* specified by invocations in the group. The identity *I* is INT_MIN when *X* is 32 bits wide and LONG_MIN when *X* is 64 bits wide. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *X* and *Result Type* must be a 32-bit or 64-bit **OpTypeInt** data type. *Execution* must be **Workgroup** or **Subgroup** *Scope*. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	271	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* X

3.32.22. Device-Side Enqueue Instructions

<table>
<tr><td colspan="6">OpEnqueueMarker

Enqueue a marker command to the queue object specified by Queue. The marker command waits for a list of events to complete, or if the list is empty it waits for all previously enqueued commands in Queue to complete before the marker completes.

Result Type must be a 32-bit integer type scalar. A successful enqueue results in the value 0. A failed enqueue results in a non-0 value.

Queue must be of the type OpTypeQueue.

Num Events specifies the number of event objects in the wait list pointed to by Wait Events and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Wait Events specifies the list of wait event objects and must be a pointer to OpTypeDeviceEvent.

Ret Event is a pointer to a device event which gets implicitly retained by this instruction. It must have a type of OpTypePointer to OpTypeDeviceEvent. If Ret Event is set to null this instruction becomes a no-op.</td><td colspan="2">Capability:
DeviceEnqueue</td></tr>
<tr><td>7</td><td>291</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Queue</td><td><id>
Num Events</td><td><id>
Wait Events</td><td><id>
Ret Event</td></tr>
</table>

OpEnqueueKernel

Enqueue the function specified by *Invoke* and the NDRange specified by *ND Range* for execution to the queue object specified by *Queue*.

Result Type must be a 32-bit *integer type* scalar. A successful enqueue results in the value 0. A failed enqueue results in a non-0 value.

Queue must be of the type **OpTypeQueue**.

Flags must be an *integer type* scalar. The content of *Flags* is interpreted as *Kernel Enqueue Flags* mask.

ND Range must have a type of **OpTypeStruct** created by **OpBuildNDRange**.

Num Events specifies the number of event objects in the wait list pointed to by *Wait Events* and must be 32-bit *integer type* scalar, which is treated as an unsigned integer.

Wait Events specifies the list of wait event objects and must be a pointer to **OpTypeDeviceEvent**.

Ret Event must be a pointer to **OpTypeDeviceEvent** which gets implicitly retained by this instruction.

Invoke must be an **OpFunction** whose **OpTypeFunction** operand has:
- *Result Type* must be **OpTypeVoid**.
- The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**.
- An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class.

Param is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar.

Param Size is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

Param Align is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

Each *Local Size* operand corresponds (in order) to one **OpTypePointer** to **Workgroup** Storage Class parameter to the *Invoke* function, and specifies the number of bytes of **Workgroup** storage used to back the pointer during the execution of the *Invoke* function.

Capability:
DeviceEnqueue

13 + variable	292	<id> *Result Type*	*Result <id>*	<id> *Queue*	<id> *Flags*	<id> *ND Range*	<id> *Num Events*	<id> *Wait Events*	<id> *Ret Event*	<id> *Invoke*	<id> *Param*	<id> *Param Size*	<id> *Param Align*	<id>, <id>, … *Local Size*

OpGetKernelNDrangeSubGroupCount Returns the number of subgroups in each workgroup of the dispatch (except for the last in cases where the global size does not divide cleanly into work-groups) given the combination of the passed NDRange descriptor specified by *ND Range* and the function specified by *Invoke*. *Result Type* must be a 32-bit *integer type* scalar. *ND Range* must have a type of **OpTypeStruct** created by **OpBuildNDRange**. *Invoke* must be an **OpFunction** whose **OpTypeFunction** operand has: - *Result Type* must be **OpTypeVoid**. - The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**. - An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class. *Param* is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar. *Param Size* is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer. *Param Align* is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.							Capability: **DeviceEnqueue**	
8	293	*<id>* *Result Type*	*Result <id>*	*<id>* *ND Range*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

OpGetKernelNDrangeMaxSubGroupSize Returns the maximum sub-group size for the function specified by *Invoke* and the NDRange specified by *ND Range*. *Result Type* must be a 32-bit *integer type* scalar. *ND Range* must have a type of **OpTypeStruct** created by **OpBuildNDRange**. *Invoke* must be an **OpFunction** whose **OpTypeFunction** operand has: - *Result Type* must be **OpTypeVoid**. - The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**. - An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class. *Param* is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar. *Param Size* is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer. *Param Align* is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.							Capability: **DeviceEnqueue**	
8	294	*<id>* *Result Type*	*Result <id>*	*<id>* *ND Range*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

OpGetKernelWorkGroupSize Returns the maximum work-group size that can be used to execute the function specified by *Invoke* on the device. *Result Type* must be a 32-bit *integer type* scalar. *Invoke* must be an **OpFunction** whose **OpTypeFunction** operand has: - *Result Type* must be **OpTypeVoid**. - The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**. - An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class. *Param* is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar. *Param Size* is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer. *Param Align* is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.						Capability: **DeviceEnqueue**	
7	295	*<id>* *Result Type*	*Result <id>*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

<table>
<tr><td colspan="6">OpGetKernelPreferredWorkGroupSizeMultiple

Returns the preferred multiple of work-group size for the function specified by Invoke. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size will not fail to enqueue Invoke for execution unless the work-group size specified is larger than the device maximum.

Result Type must be a 32-bit integer type scalar.

Invoke must be an OpFunction whose OpTypeFunction operand has:
- Result Type must be OpTypeVoid.
- The first parameter must have a type of OpTypePointer to an 8-bit OpTypeInt.
- An optional list of parameters, each of which must have a type of OpTypePointer to the Workgroup Storage Class.

Param is the first parameter of the function specified by Invoke and must be a pointer to an 8-bit integer type scalar.

Param Size is the size in bytes of the memory pointed to by Param and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Param Align is the alignment of Param and must be a 32-bit integer type scalar, which is treated as an unsigned integer.</td><td colspan="2">Capability:
DeviceEnqueue</td></tr>
<tr><td>7</td><td>296</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Invoke</td><td><id>
Param</td><td><id>
Param Size</td><td><id>
Param Align</td></tr>
</table>

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

Increments the reference count of the event object specified by Event.

Event must be an event that was produced by OpEnqueueKernel, OpEnqueueMarker or OpCreateUserEvent.</td><td>Capability:
DeviceEnqueue</td></tr>
<tr><td>2</td><td>297</td><td><id>
Event</td></tr>
</table>

OpReleaseEvent Decrements the reference count of the event object specified by *Event*. The event object is deleted once the event reference count is zero, the specific command identified by this event has completed (or terminated) and there are no commands in any device command queue that require a wait for this event to complete. *Event* must be an event that was produced by **OpEnqueueKernel**, **OpEnqueueMarker** or **OpCreateUserEvent**.		Capability: **DeviceEnqueue**
2	298	*<id>* *Event*

OpCreateUserEvent Create a user event. The execution status of the created event is set to a value of 2 (CL_SUBMITTED). *Result Type* must be **OpTypeDeviceEvent**.			Capability: **DeviceEnqueue**
3	299	*<id>* *Result Type*	*Result <id>*

OpIsValidEvent Returns **true** if the event specified by *Event* is a valid event, otherwise result is **false**. *Result Type* must be a *Boolean type*. *Event* must have a type of **OpTypeDeviceEvent**				Capability: **DeviceEnqueue**
4	300	*<id>* *Result Type*	*Result <id>*	*<id>* *Event*

OpSetUserEventStatus Sets the execution status of a user event specified by *Event*.*Status* can be either 0 (CL_COMPLETE) to indicate that this kernel and all its child kernels finished execution successfully, or a negative integer value indicating an error. *Event* must have a type of **OpTypeDeviceEvent** that was produced by **OpCreateUserEvent**. *Status* must have a type of 32-bit **OpTypeInt** treated as a signed integer.			Capability: **DeviceEnqueue**
3	301	*<id>* *Event*	*<id>* *Status*

OpCaptureEventProfilingInfo Captures the profiling information specified by *Profiling Info* for the command associated with the event specified by *Event* in the memory pointed to by *Value*.The profiling information will be available in the memory pointed to by *Value* once the command identified by *Event* has completed. *Event* must have a type of **OpTypeDeviceEvent** that was produced by **OpEnqueueKernel** or **OpEnqueueMarker**. *Profiling Info* must be an *integer type* scalar. The content of *Profiling Info* is interpreted as *Kernel Profiling Info* mask. *Value* must be a pointer to a scalar 8-bit *integer type* in the **CrossWorkgroup** Storage Class. When *Profiling Info* is **CmdExecTime**, *Value* must point to 128-bit memory range. The first 64 bits contain the elapsed time CL_PROFILING_COMMAND_END - CL_PROFILING_COMMAND_START for the command identified by *Event* in nanoseconds. The second 64 bits contain the elapsed time CL_PROFILING_COMMAND_COMPLETE - CL_PROFILING_COMMAND_START for the command identified by *Event* in nanoseconds. **Note:** The behavior of this instruction is undefined when called multiple times for the same event.				Capability: **DeviceEnqueue**
4	302	*<id>* *Event*	*<id>* *Profiling Info*	*<id>* *Value*

OpGetDefaultQueue Returns the default device queue. If a default device queue has not been created, a null queue object is returned. *Result Type* must be an **OpTypeQueue**.			Capability: **DeviceEnqueue**
3	303	*<id>* *Result Type*	*Result <id>*

OpBuildNDRange Given the global work size specified by *GlobalWorkSize*, local work size specified by *LocalWorkSize* and global work offset specified by *GlobalWorkOffset*, builds a 1D, 2D or 3D ND-range descriptor structure and returns it. *Result Type* must be an **OpTypeStruct** with the following ordered list of members, starting from the first to last: 1) 32-bit *integer type* scalar, that specifies the number of dimensions used to specify the global work-items and work-items in the work-group. 2) **OpTypeArray** with 3 elements, where each element is 32-bit *integer type* scalar when the *addressing model* is **Physical32** and 64-bit *integer type* scalar when the *addressing model* is **Physical64**. This member is an array of per-dimension unsigned values that describe the offset used to calculate the global ID of a work-item. 3) **OpTypeArray** with 3 elements, where each element is 32-bit *integer type* scalar when the *addressing model* is **Physical32** and 64-bit *integer type* scalar when the *addressing model* is **Physical64**. This member is an array of per-dimension unsigned values that describe the number of global work-items in the dimensions that will execute the kernel function. 4) **OpTypeArray** with 3 elements, where each element is 32-bit *integer type* scalar when the *addressing model* is **Physical32** and 64-bit *integer type* scalar when the *addressing model* is **Physical64**. This member is an array of per-dimension unsigned values that describe the number of work-items that make up a work-group. *GlobalWorkSize* must be a scalar or an array with 2 or 3 components. Where the type of each element in the array is 32-bit *integer type* scalar when the *addressing model* is **Physical32** or 64-bit *integer type* scalar when the *addressing model* is **Physical64**. The type of *LocalWorkSize* must be the same as *GlobalWorkSize*. The type of *GlobalWorkOffset* must be the same as *GlobalWorkSize*.					Capability: **DeviceEnqueue**	
6	304	*<id>* *Result Type*	*Result <id>*	*<id>* *GlobalWorkSize*	*<id>* *LocalWorkSize*	*<id>* *GlobalWorkOffset*

OpGetKernelLocalSizeForSubgroupCount Returns the 1D local size to enqueue *Invoke* with *Subgroup Count* subgroups per workgroup. *Result Type* must be a 32-bit *integer type* scalar. *Subgroup Count* must be a 32-bit *integer type* scalar. *Invoke* must be an **OpFunction** whose **OpTypeFunction** operand has: - *Result Type* must be **OpTypeVoid**. - The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**.	Capability: **SubgroupDispatch**

- An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class.

Param is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar.

Param Size is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

Param Align is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

8	325	*<id>* *Result Type*	*Result <id>*	*<id>* *Subgroup Count*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

OpGetKernelMaxNumSubgroups

Capability:
SubgroupDispatch

Returns the maximum number of subgroups that can be used to execute *Invoke* on the devce.

Result Type must be a 32-bit *integer type* scalar.

Invoke must be an **OpFunction** whose **OpTypeFunction** operand has:
- *Result Type* must be **OpTypeVoid**.
- The first parameter must have a type of **OpTypePointer** to an 8-bit **OpTypeInt**.
- An optional list of parameters, each of which must have a type of **OpTypePointer** to the **Workgroup** Storage Class.

Param is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit *integer type* scalar.

Param Size is the size in bytes of the memory pointed to by *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

Param Align is the alignment of *Param* and must be a 32-bit *integer type* scalar, which is treated as an unsigned integer.

7	326	*<id>* *Result Type*	*Result <id>*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

3.32.23. Pipe Instructions

OpReadPipe Read a packet from the pipe object specified by *Pipe* into *Pointer*. Result is 0 if the operation is successful and a negative value if the pipe is empty. *Result Type* must be a 32-bit *integer type* scalar. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** *access qualifier*. *Pointer* must have a type of **OpTypePointer** with the same data type as *Pipe* and a **Generic** Storage Class. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.						Capability: **Pipes**	
7	274	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpWritePipe Write a packet from *Pointer* to the pipe object specified by *Pipe*. Result is 0 if the operation is successful and a negative value if the pipe is full. *Result Type* must be a 32-bit *integer type* scalar. *Pipe* must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*. *Pointer* must have a type of **OpTypePointer** with the same data type as *Pipe* and a **Generic** Storage Class. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.						Capability: **Pipes**	
7	275	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpReservedReadPipe Read a packet from the reserved area specified by *Reserve Id* and *Index* of the pipe object specified by *Pipe* into *Pointer*. The reserved pipe entries are referred to by indices that go from 0 … *Num Packets* - 1. Result is 0 if the operation is successful and a negative value otherwise. *Result Type* must be a 32-bit *integer type* scalar. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Index* must be a 32-bit *integer type* scalar, which is treated as an unsigned value. *Pointer* must have a type of **OpTypePointer** with the same data type as *Pipe* and a **Generic** Storage Class. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.								Capability: **Pipes**	
9	276	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Index*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpReservedWritePipe Write a packet from *Pointer* into the reserved area specified by *Reserve Id* and *Index* of the pipe object specified by *Pipe*. The reserved pipe entries are referred to by indices that go from 0 … *Num Packets* - 1. Result is 0 if the operation is successful and a negative value otherwise. *Result Type* must be a 32-bit *integer type* scalar. *Pipe* must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Index* must be a 32-bit *integer type* scalar, which is treated as an unsigned value. *Pointer* must have a type of **OpTypePointer** with the same data type as *Pipe* and a **Generic** Storage Class. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.								Capability: **Pipes**	
9	277	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Index*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpReserveReadPipePackets	Capability:

<table>
<tr><td colspan="6">Reserve Num Packets entries for reading from the pipe object specified by Pipe. Result is a valid reservation ID if the reservation is successful.

Result Type must be an OpTypeReserveId.

Pipe must have a type of OpTypePipe with ReadOnly access qualifier.

Num Packets must be a 32-bit integer type scalar, which is treated as an unsigned value.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td colspan="2">Pipes</td></tr>
<tr><td>7</td><td>278</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pipe</td><td><id>
Num Packets</td><td><id>
Packet Size</td><td><id>
Packet Alignment</td></tr>
</table>

<table>
<tr><td colspan="6">OpReserveWritePipePackets

Reserve num_packets entries for writing to the pipe object specified by Pipe. Result is a valid reservation ID if the reservation is successful.

Pipe must have a type of OpTypePipe with WriteOnly access qualifier.

Num Packets must be a 32-bit OpTypeInt which is treated as an unsigned value.

Result Type must be an OpTypeReserveId.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>7</td><td>279</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pipe</td><td><id>
Num Packets</td><td><id>
Packet Size</td><td><id>
Packet Alignment</td></tr>
</table>

OpCommitReadPipe Indicates that all reads to *Num Packets* associated with the reservation specified by *Reserve Id* and the pipe object specified by *Pipe* are completed. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**
5	280	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpCommitWritePipe Indicates that all writes to *Num Packets* associated with the reservation specified by *Reserve Id* and the pipe object specified by *Pipe* are completed. *Pipe* must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**
5	281	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpIsValidReserveId Return **true** if *Reserve Id* is a valid reservation id and **false** otherwise. *Result Type* must be a *Boolean type*. *Reserve Id* must have a type of **OpTypeReserveId**.				Capability: **Pipes**
4	282	*<id>* *Result Type*	*Result <id>*	*<id>* *Reserve Id*

OpGetNumPipePackets Result is the number of available entries in the pipe object specified by *Pipe*. The number of available entries in a pipe is a dynamic value. The value returned should be considered immediately stale. *Result Type* must be a 32-bit *integer type* scalar, which should be treated as an unsigned value. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** or **WriteOnly** *access qualifier*. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**	
6	283	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpGetMaxPipePackets Result is the maximum number of packets specified when the pipe object specified by *Pipe* was created. *Result Type* must be a 32-bit *integer type* scalar, which should be treated as an unsigned value. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** or **WriteOnly** *access qualifier*. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**	
6	284	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpGroupReserveReadPipePackets Reserve *Num Packets* entries for reading from the pipe object specified by *Pipe* at group level. Result is a valid reservation id if the reservation is successful. The reserved pipe entries are referred to by indices that go from 0 … *Num Packets* - 1. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Result Type* must be an **OpTypeReserveId**. *Execution* must be **Workgroup** or **Subgroup** *Scope*. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** *access qualifier*. *Num Packets* must be a 32-bit *integer type* scalar, which is treated as an unsigned value. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.							Capability: **Pipes**	
8	285	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Pipe*	*<id>* *Num Packets*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpGroupReserveWritePipePackets

Reserve *Num Packets* entries for writing to the pipe object specified by *Pipe* at group level. Result is a valid reservation ID if the reservation is successful.

The reserved pipe entries are referred to by indices that go from 0 … *Num Packets* - 1.

All invocations of this module within *Execution* must reach this point of execution.

This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely.

Result Type must be an **OpTypeReserveId**.

Execution must be **Workgroup** or **Subgroup** *Scope*.

Pipe must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*.

Num Packets must be a 32-bit *integer type* scalar, which is treated as an unsigned value.

Packet Size and *Packet Alignment* must satisfy the following:
- 1 <= *Packet Alignment* <= *Packet Size*.
- *Packet Alignment* must evenly divide *Packet Size*

For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.

Capability:
Pipes

8	286	*<id> Result Type*	*Result <id>*	*Scope <id> Execution*	*<id> Pipe*	*<id> Num Packets*	*<id> Packet Size*	*<id> Packet Alignment*

OpGroupCommitReadPipe A group level indication that all reads to *Num Packets* associated with the reservation specified by *Reserve Id* to the pipe object specified by *Pipe* are completed. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Execution* must be **Workgroup** or **Subgroup** *Scope*. *Pipe* must have a type of **OpTypePipe** with **ReadOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**	
6	287	*Scope <id>* *Execution*	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpGroupCommitWritePipe A group level indication that all writes to *Num Packets* associated with the reservation specified by *Reserve Id* to the pipe object specified by *Pipe* are completed. All invocations of this module within *Execution* must reach this point of execution. This instruction is only guaranteed to work correctly if placed strictly within uniform control flow within *Execution*. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, an invocation may stall indefinitely. *Execution* must be **Workgroup** or **Subgroup** *Scope*. *Pipe* must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.					Capability: **Pipes**	
6	288	*Scope <id>* *Execution*	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpConstantPipeStorage Creates a pipe-storage object. *Result Type* must be **OpTypePipeStorage**. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types. *Capacity* is the minimum number of *Packet Size* blocks the resulting **OpTypePipeStorage** can hold.					Capability: **PipeStorage**	
6	323	*<id>* *Result Type*	*Result <id>*	*Literal Number* *Packet Size*	*Literal Number* *Packet Alignment*	*Literal Number* *Capacity*

OpCreatePipeFromPipeStorage Creates a pipe object from a pipe-storage object. *Result Type* must be **OpTypePipe**. *Pipe Storage* must be a pipe-storage object created from **OpConstantPipeStorage**.	Capability: **PipeStorage**

Qualifier is the pipe access qualifier.				
4	324	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe Storage*

4. Appendix A: Changes

4.1. Changes from Version 0.99, Revision 31

- Added the **PushConstant** Storage Class.
- Added **OpIAddCarry**, **OpISubBorrow**, **OpUMulExtended**, and **OpSMulExtended**.
- Added **OpInBoundsPtrAccessChain**.
- Added the Decoration **NoContraction** to prevent combining multiple operations into a single operation (bug 14396).
- Added sparse texturing (14486):
  - Added **OpImageSparse…** for accessing images that might not be resident.
  - Added **MinLod** functionality for accessing images with a minimum level of detail.
- Added back the **Alignment** Decoration, for the **Kernel** capability (14505).
- Added a **NonTemporal** Memory Access (14566).
- Structured control flow changes:
  - Changed structured loops to have a structured continue *Continue Target* in **OpLoopMerge** (14422).
  - Added rules for how "fall through" works with **OpSwitch** (13579).
  - Added definitions for what is "inside" a structured control-flow construct (14422).
- Added **SubpassData** Dim to support input targets written by a previous subpass as an output target (14304). This is also a Decoration and a Capability, and can be used by some image ops to read the input target.
- Added **OpTypeForwardPointer** to establish the Storage Class of a forward reference to a pointer type (13822).
- Improved Debuggability
  - Changed **OpLine** to not have a target *<id>*, but instead be placed immediately preceding the instruction(s) it is annotating (13905).
  - Added **OpNoLine** to terminate the affect of **OpLine** (13905).
  - Changed **OpSource** to include the source code:
    - Allow multiple occurrences.
    - Be mixed in with the **OpString** instructions.
    - Optionally consume an **OpString** result to say which file it is annotating.
    - Optionally include the source text corresponding to that **OpString**.
    - Included adding **OpSourceContinued** for source text that is too long for a single instruction.
- Added a large number of Capabilities for subsetting functionality (14520, 14453), including 8-bit integer support for OpenCL kernels.
- Added **VertexIndex** and **InstanceIndex** BuiltIn Decorations (14255).
- Added **GenericPointer** capability that allows the ability to use the **Generic** Storage Class (14287).
- Added **IndependentForwardProgress** Execution Mode (14271).
- Added **OpAtomicFlagClear** and **OpAtomicFlagTestAndSet** instructions (14315).
- Changed **OpEntryPoint** to take a list of **Input** and **Output** *<id>* for declaring the entry point's interface.
- Fixed internal bugs
  - 14411 Added missing documentation for mad_sat OpenCL extended instructions (enums existed, just the documentation was missing)
  - 14241 Removed shader capability requirement from **OpImageQueryLevels** and **OpImageQuerySamples**.
  - 14241 Removed unneeded OpImageQueryDim instruction.

- 14241 Filled in *TBD* section for OpAtomicCompareExchangeWeek
- 14366 All **OpSampledImage** must appear before uses of sampled images (and still in the first block of the entry point).
- 14450 DeviceEnqueue capability is required for OpTypeQueue and OpTypeDeviceEvent
- 14363 OpTypePipe is opaque - moved packet size and alignment to opcodes
- 14367 Float16Buffer capability clarified
- 14241 Clarified how **OpSampledImage** can be used
- 14402 Clarified **OpTypeImage** encodings for OpenCL extended instructions
- 14569 Removed mention of non-existent OpFunctionDecl
- 14372 Clarified usage of **OpGenericPtrMemSemantics**
- 13801 Clarified the **SpecId** Decoration is just for constants
- 14447 Changed literal values of Memory Semantic enums to match OpenCL/C++11 atomics, and made the Memory Semantic **None** and **Relaxed** be aliases
- 14637 Removed subgroup scope from **OpGroupAsyncCopy** and **OpGroupWaitEvents**

4.2. Changes from Version 0.99, Revision 32

- Added **UnormInt101010_2** to the Image Channel Data Type table.
- Added place holder for C++11 atomic *Consume* Memory Semantics along with an explicit AcquireRelease memory semantic.
- Fixed internal bugs:
  - 14690 **OpSwitch** *literal* width (and hence number of operands) is determined by the type of *Selector*, and be rigorous about how sub-32-bit literals are stored.
  - 14485 The client API owns the semantics of built-ins that only have "pass through" semantics WRT SPIR-V.
- Fixed public bugs:
  - 1387 Don't describe result type of **OpImageWrite**.

4.3. Changes from Version 1.00, Revision 1

- Adjusted Capabilities:
  - Split geometry-stream functionality into its own **GeometryStreams** capability (14873).
  - Have **InputAttachmentIndex** to depend on **InputAttachment** instead of **Shader** (14797).
  - Merge **AdvancedFormats** and **StorageImageExtendedFormats** into just **StorageImageExtendedFormats** (14824).
  - Require **StorageImageReadWithoutFormat** and **StorageImageWriteWithoutFormat** to read and write storage images with an **Unknown** Image Format.
  - Removed the **ImageSRGBWrite** capability.
- Clarifications
  - **RelaxedPrecision** Decoration can be applied to **OpFunction** (14662).
- Fixed internal bugs:
  - 14797 The literal argument was missing for the **InputAttachmentIndex** Decoration.
  - 14547 Remove the **FragColor** BuiltIn, so that no implicit broadcast is implied.
  - 13292 Make statements about "Volatile" be more consistent with the memory model specification (non-functional change).
  - 14948 Remove image-"Query" overloading on image/sampled-image type and "fetch" on non-sampled images, by adding the **OpImage** instruction to get the image from a sampled image.
  - 14949 Make consistent placement between **OpSource** and **OpSourceExtension** in the logical layout of a module.

  - 14865 Merge **WorkgroupLinearId** with **LocalInvocationId** BuiltIn Decorations.
  - 14806 Include 3D images for **OpImageQuerySize**.
  - 14325 Removed the **Smooth** Decoration.
  - 12771 Make the version word formatted as: "0 | Major Number | Minor Number | 0" in the physical layout.
  - 15035 Allow **OpTypeImage** to use a *Depth* operand of 2 for not indicating a depth or non-depth image.
  - 15009 Split the **OpenCL** Source Language into two: **OpenCL_C** and **OpenCL_CPP**.
  - 14683 **OpSampledImage** instructions can only be the consuming block, for scalars, and directly consumed by an image lookup or query instruction.
  - 14325 mutual exclusion validation rules of Execution Modes and Decorations
  - 15112 add definitions for invocation, dynamically uniform, and uniform control flow.
- Renames
  - **InputTargetIndex** Decoration → **InputAttachmentIndex**
  - **InputTarget** Capability→ **InputAttachment**
  - **InputTarget** Dim → **SubpassData**
  - **WorkgroupLocal** Storage Class → **Workgroup**
  - **WorkgroupGlobal** Storage Class → **CrossWorkgroup**
  - **PrivateGlobal** Storage Class → **Private**
  - **OpAsyncGroupCopy** → **OpGroupAsyncCopy**
  - **OpWaitGroupEvents** → **OpGroupWaitEvents**
  - **InputTriangles** Execution Mode → **Triangles**
  - **InputQuads** Execution Mode → **Quads**
  - **InputIsolines** Execution Mode → **Isolines**

4.4. Changes from Version 1.00, Revision 2

- Updated example at the end of Section 1 to conform to the KHR_vulkan_glsl extension and treat OpTypeBool as an abstract type.
- Adjusted Capabilities:
  - **MatrixStride** depends on **Matrix** (15234).
  - **Sample**, **SampleId**, **SamplePosition**, and **SampleMask** depend on **SampleRateShading** (15234).
  - **ClipDistance** and **CullDistance** BuiltIns depend on, respectively, **ClipDistance** and **CullDistance** (1407, 15234).
  - **ViewportIndex** depends on **MultiViewport** (15234).
  - **AtomicCounterMemory** should be the **AtomicStorage** (15234).
  - **Float16** has no dependencies (15234).
  - **Offset** Decoration should only be for **Shader** (15268).
  - **Generic** Storage Class is supposed to need the **GenericPointer** Capability (14287).
  - Remove capability restriction on the **BuiltIn** Decoration (15248).
- Fixed internal bugs:
  - 15203 Updated description of **SampleMask** BuiltIn to include "Input or output…", not just "Input…"
  - 15225 Include no re-association as a constraint required by the **NoContraction** Decoration.
  - 15210 Clarify **OpPhi** semantics that operand values only come from parent blocks.
  - 15239 Add OpImageSparseRead, which was missing (supposed to be 12 sparse-image instructions, but only 11 got incorporated, this adds the 12th).
  - 15299 Move **OpUndef** back to the Miscellaneous section.

    - 15321 **OpTypeImage** does not have a *Depth* restriction when used with **SubpassData**.
    - 14948 Fix the **Lod** Image Operands to allow both integer and floating-point values.
    - 15275 Clarify specific storage classes allowed for atomic operations under universal validation rules "Atomic access rules".
    - 15501 Restrict **Patch** Decoration to one of the tessellation execution models.
    - 15472 Reserved use of **OpImageSparseSampleProjImplicitLod**, **OpImageSparseSampleProjExplicitLod**, **OpImageSparseSampleProjDrefImplicitLod**, and **OpImageSparseSampleProjDrefExplicitLod**.
    - 15459 Clarify what makes different aggregate types in "Types and Variables".
    - 15426 Don't require **OpQuantizeToF16** to preserve NaN patterns.
    - 15418 Don't set both **Acquire** and **Release** bits in Memory Semantics.
    - 15404 **OpFunction** *Result <id>* can only be used by **OpFunctionCall**, **OpEntryPoint**, and decoration instructions.
    - 15437 Restrict element type for **OpTypeRuntimeArray** by adding a definition of concrete types.
    - 15403 Clarify **OpTypeFunction** can only be consumed by **OpFunction** and functions can only return concrete and abstract types.
- Improved accuracy of the opcode word count in each instruction regarding which operands are optional. For sampling operations with explicit LOD, this included not marking the required LOD operands as optional.
- Clarified that when **NonWritable**, **NonReadable**, **Volatile**, and **Coherent** Decorations are applied to the **Uniform** storage class, the **BufferBlock** decoration must be present.
- Fixed external bugs:
    - 1413 (see internal 15275)
    - 1417 Added definitions for block, dominate, post dominate, CFG, and back edge. Removed use of "dominator tree".

4.5. Changes from Version 1.00, Revision 3

- Added definition of derivative group, and use it to say when derivatives are well defined.

4.6. Changes from Version 1.00, Revision 4

- Expanded the list of instructions that may use or return a pointer in the **Logical** addressing model.
- Added missing ABGR **Image Channel Order**

4.7. Changes from Version 1.00, Revision 5

- Khronos SPIR-V issue #27: Removed **Shader** dependency from **SampledBuffer** and **Sampled1D** Capabilities.
- Khronos SPIR-V issue #56: Clarify that the meaning of "read-only" in the Storage Classes includes not allowing initializers.
- Khronos SPIR-V issue #57: Clarify "modulo" means "remainder" in **OpFMod**'s description.
- Khronos SPIR-V issue #60: **OpControlBarrier** synchronizes **Output** variables when used in tessellation-control shader.
- Public SPIRV-Headers issue #1: Remove the **Shader** capability requirement from the **Input** Storage Class.
- Public SPIRV-Headers issue #10: Don't say the *(u [, v] [, w], q)* has four components, as it can be closed up when the optional ones are missing. Seen in the projective image instructions.
- Public SPIRV-Headers issues #12 and #13 and Khronos SPIR-V issue #65: Allow **OpVariable** as an initializer for another **OpVariable** instruction or the *Base* of an **OpSpecConstantOp** with an **AccessChain** opcode.
- Public SPIRV-Headers issues #14: add **Max** enumerants of 0x7FFFFFFF to each of the non-mask

enums in the C-based header files.

4.8. Changes from Version 1.00, Revision 6

- Khronos SPIR-V issue #63: Be clear that **OpUndef** can be used in sequence 9 (and is preferred to be) of the Logical Layout and can be part of partially-defined **OpConstantComposite**.
- Khronos SPIR-V issue #70: Don't explicitly require operand truncation for integer operations when operating at **RelaxedPrecision**.
- Khronos SPIR-V issue #76: Include **OpINotEqual** in the list of allowed instructions for **OpSpecConstantOp**.
- Khronos SPIR-V issue #79: Remove implication that **OpImageQueryLod** should have a component for the array index.
- Public SPIRV-Headers issue #17: Decorations **Noperspective**, **Flat**, **Patch**, **Centroid**, and **Sample** can apply to a top-level member that is itself a structure, so don't disallow it through restrictions to numeric types.

4.9. Changes from Version 1.00

- Moved version number to SPIR-V 1.1
- New functionality:
  - Bug 14202 named barriers:
    - Added the **NamedBarrier** Capability.
    - Added the instructions: **OpTypeNamedBarrier**, **OpNamedBarrierInitialize**, and **OpMemoryNamedBarrier**.
  - Bug 14201 subgroup dispatch:
    - Added the **SubgroupDispatch** Capability.
    - Added the instructions: **OpGetKernelLocalSizeForSubgroupCount** and **OpGetKernelMaxNumSubgroups**.
    - Added **SubgroupSize** and **SubgroupsPerWorkgroup** Execution Modes.
  - Bug 14441 program-scope pipes:
    - Added the **PipeStorage** Capability.
    - Added Instructions: **OpTypePipeStorage**, **OpConstantPipeStorage**, and **OpCreatePipeFromPipeStorage**.
  - Bug 15434 Added the **OpSizeOf** instruction.
  - Bug 15024 support for OpenCL-C++ ivdep loop attribute:
    - Added **DependencyInfinite** and **DependencyLength** Loop Controls.
    - Updated **OpLoopMerge** to support these.
  - Bug 14022 Added **Initializer** and **Finalizer** and Execution Modes.
  - Bug 15539 Added the **MaxByteOffset** Decoration.
  - Bug 15073 Added the **Kernel** Capability to the **SpecId** Decoration.
  - Bug 14828 Added the **OpModuleProcessed** instruction.
- Fixed internal bugs:
  - Bug 15481 Clarification on alignment and size operands for pipe operands

4.10. Changes from Version 1.1, Revision 1

- Incorporated bug fixes from Revision 6 of Version 1.00 (see section 4.7. Changes from Version 1.00, Revision 5).

4.11. Changes from Version 1.1, Revision 2

- Incorporated bug fixes from Revision 7 of Version 1.00 (see section 4.8. Changes from Version 1.00, Revision 6).

Version 1.1, Revision 3
Last updated 2016-04-14 16:52:03 MDT