SPIR-V Specification

John Kessenich, Google, Boaz Ouriel, Intel, Raun Krisch, Intel

Version 1.6, Revision 1: Unified

Table of Contents

Contributors and Acknowledgments

Connor Abbott, Intel

Ben Ashbaugh, Intel

Alexey Bader, Intel

Alan Baker, Google

Dan Baker, Oxide Games

Kenneth Benzie, Codeplay

Stuart Brady, Arm

Gordon Brown, Codeplay

Pat Brown, NVIDIA

Diana Po-Yu Chen, MediaTek

Stephen Clarke, Imagination

Patrick Doane, Blizzard Entertainment

Alastair Donaldson, Google

Yuehai Du, Qualcomm

Stefanus Du Toit, Google

Gregory Fischer, LunarG

Theresa Foley, Intel

Spencer Fricke, Samsung

Ben Gaster, Qualcomm

Alexander Galazin, ARM

Christopher Gautier, ARM

Tobias Hector, AMD

Nicolai Hahnle, AMD

Neil Henning, AMD

Kerch Holt, NVIDIA

Lee Howes, Qualcomm

Roy Ju, MediaTek

Baldur Karlsson, Valve

Ronan Keryell, Xilinx

John Kessenich, Google

Daniel Koch, NVIDIA

Ashwin Kolhe, NVIDIA

Raun Krisch, Intel

Graeme Leese, Broadcom

Yuan Lin, NVIDIA

Yaxun Liu, AMD

Victor Lomuller, Codeplay

Timothy Lottes, Epic Games

John McDonald, Valve

Mariusz Merecki, Intel

David Neto, Google

Boaz Ouriel, Intel

Kevin Petit, Arm

Robert Quill, Imagination Technologies

Christophe Riccio, Unity

Andrew Richards, Codeplay

Ian Romanick, Intel

Graham Sellers, AMD

Simon Waters, Samsung

Robert Simpson, Qualcomm

Bartosz Sochacki, Intel

Nikos Stavropoulos, Think Silicon

Brian Sumner, AMD

Andrew Woloszyn, Google

Ruihao Zhang, Qualcomm

Weifeng Zhang, Qualcomm

Chapter 1. Introduction

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/)

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 client APIs.

This is a unified specification, specifying all versions since and including version 1.0.

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 a client API into a driver to set shaders/kernels.
- Support multiple execution environments, specified by client APIs.
- 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.
- 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. Execution Environment and Client API

SPIR-V is adaptable to multiple execution environments: A SPIR-V module is consumed by an execution environment, as specified by a client API. The full set of rules needed to consume SPIR-V in a particular environment comes from the combination of SPIR-V and that environment's client API specification. The client API specifies its SPIR-V execution environment as well as extra rules, limitations, capabilities, etc. required by the form of SPIR-V it can validly consume.

1.3. About This Document

This document aims to:

- Specify everything needed to create and consume non-extended SPIR-V, minus:
  - Extended instruction sets, which are imported and come with their own specifications.
  - Client API-specific rules, which are documented in client API specifications.
- Separate expository and specification language. The specification-proper is in Specification and Binary Form.

1.3.1. Versioning

The specification covers multiple versions of SPIR-V, as described in the unified section. It has followed a *Major*.*Minor*.*Revision* versioning scheme, with the specification's stated version being the most recent version of SPIR-V.

Major and *Minor* (but not *Revision*) are declared within a SPIR-V module.

Major is reserved for future use and has been fixed at 1. *Minor* changes have signified additions, deprecation, and removal of features. *Revision* changes have included clarifications, bug fixes, and deprecation (but not removal) of existing features.

1.4. Extendability

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

- Using the **OpExtension** instruction to add semantics, which are described in an extension specification.
- 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, use the https://github.com/KhronosGroup/SPIRV-Headers repository, and submit pull requests against the include/spirv/spir-v.xml file.

Extended Instructions. Sets of extended instructions can be provided and specified in separate specifications. 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.

1.5. 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.

1.6. 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 follows 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 generally affect the way the module interacts with its execution environment, not its 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 is done through load/store of variables declared in these storage classes.

1.7. 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 is done through load/store, and initial load and final store won’t 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.

1.8. 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 are otherwise declared with their correct SPIR-V type and treated the same as any other variable.

1.9. Specialization

Specialization enables offline creation of a portable SPIR-V module based on constant values that won't be known until a later point in time. 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.

1.10. 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
```

Chapter 2. Specification

2.1. Language Capabilities

A SPIR-V module is consumed 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 are declared early in that module with the **OpCapability** instruction. Then:

- A validator can validate that the module uses only its declared capabilities.
- A client API is allowed to reject modules declaring capabilities it does not support.

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

If an instruction, enumerant, or other feature specifies multiple enabling capabilities, only one such capability needs to be declared to use the feature. This declaration does not itself imply anything about the presence of the other enabling capabilities: The execution environment needs to support only the declared capability.

The SPIR-V specification provides universal capability-specific validation rules, in the validation section. Additionally, each client API includes the following:

- Which capabilities in the capability section it supports or requires, and hence allows in a SPIR-V module.
- Any additional validation rules it has beyond those specified by the SPIR-V specification.
- Required limits, if they are beyond the Universal Limits.

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: An immediate value, not an *<id>*. Literals larger than one *word* consume multiple operands, one per word. An instruction states what type the literal will be interpreted as. A string is interpreted as a nul-terminated stream of characters. All string comparisons are case sensitive. 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. For a numeric literal, the lower-order words appear first. If a numeric type's bit width is less than 32-bits, the value appears in the low-order bits of the word, and the high-order bits must be 0 for a floating-point type or integer type with *Signedness* of 0, or sign extended for an integer type with a *Signedness* of 1 (similarly for the remaining bits of widths larger than 32 bits but not a multiple of 32 bits).

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

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 exists only for the duration of a function if it is a function variable, and otherwise exists for the duration of an invocation.

Memory Object Declaration: An **OpVariable**, or an **OpFunctionParameter** of pointer type, or the contents of an **OpVariable** that holds either a pointer to the **PhysicalStorageBuffer** storage class or an array of such pointers.

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 non-specialization 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*. Parentheses exclude their end point, so, for example, *(a, b]* means *a* to *b* excluding *a* but including *b*.

Non-Semantic Instruction: An instruction that has no semantic impact, and that can be safely removed from the module.

2.2.2. Types

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

Integer type: Any width signed or unsigned type from **OpTypeInt**. By convention, the lowest-order bit is 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 are also 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. The vectors forming a matrix are also called its *columns*. Matrix sizes are quite restrictive and dependent on the execution model.

Array: An ordered homogeneous aggregate of any non-void-type objects. The objects forming an array are also called its *elements*. Array sizes are generally not restricted.

Structure: An ordered heterogeneous aggregate of any non-void types. The objects forming a structure are also called its *members*.

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. Comes either from literal declarations of settings or from 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.

Physical Pointer Type: An **OpTypePointer** whose *Storage Class* uses physical addressing according to the addressing model.

Logical Pointer Type: A pointer type that is not a physical pointer type.

Concrete Type: A numerical scalar, vector, or matrix type, or physical pointer type, or any aggregate containing only these types.

Abstract Type: An **OpTypeVoid** or **OpTypeBool**, or logical pointer type, 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**

Variable pointer: A pointer of logical pointer type that results from one of the following instructions:

- **OpSelect**
- **OpPhi**
- **OpFunctionCall**
- **OpPtrAccessChain**
- **OpLoad**
- **OpConstantNull**

Additionally, any **OpAccessChain**, **OpInBoundsAccessChain**, or **OpCopyObject** that takes a variable pointer as an operand also produces a variable pointer. An **OpFunctionParameter** of pointer type is a variable pointer if any **OpFunctionCall** to the function statically passes a variable pointer as the value of the parameter.

2.2.3. Computation

Remainder: When dividing a by b, a *remainder* r is defined to be a value that satisfies $r + q \times b = a$ where q is a whole number and $|r| < |b|$.

2.2.4. 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.

2.2.5. Control Flow

Block: A contiguous sequence of instructions starting with an **OpLabel**, ending with a termination instruction. A *block* has no additional label or termination instructions.

Branch Instruction: One of the following, used as a termination instruction:

- **OpBranch**
- **OpBranchConditional**
- **OpSwitch**
- **OpReturn**
- **OpReturnValue**

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

- any branch instruction
- **OpKill**
- **OpTerminateInvocation**
- **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* includes block *A*. *A strictly dominates B* only if *A dominates B* and *A* and *B* are different blocks.

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: A branch is a *back edge* if there is a depth-first search starting at the entry block of the CFG where the branch branches to one of its ancestors. A *back-edge block* is a block containing such a branch instruction.
Note: For a given function, if all its loops are structured, then each back edge corresponds to exactly one loop header, and vice versa. So the set of back-edges in the function is unique, regardless of the depth-first search used to find them. This is equivalent to the function's CFG being reducible.

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.

Quad: The execution environment can partition invocations into *quads*, where invocations within a quad can synchronize and share data with each other efficiently. See the client API specification for more details.

Quad index: The index of an invocation in a quad.

Subgroup: Invocations are partitioned into subgroups, where invocations within a subgroup can synchronize and share data with each other efficiently. 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 derivatives, which is at most as large as a single point, line, or triangle, including any helper invocations, as defined by the client API.

Tangled Instruction: One of:

- Group and subgroup instructions

- Non-uniform instructions
- **OpControlBarrier**
- Derivative instructions
- Image instructions that consume an implicit derivative

Tangled instructions communicate between 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 if 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.

Additionally, a single dynamic instance may be executed by multiple invocations. Only tangled instructions are required to execute the dynamic instance as if all invocations that communicate together and share the same dynamic instance execute simultaneously. Invocations that execute the same dynamic instance of an instruction will continue to execute the same dynamic instances as long as they follow the same control flow path. A dynamic instance of an instruction, tangled or not, is executed by one or more invocations.

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

Uniform Control Flow: Uniform control flow (or converged control flow) occurs if all invocations (in the invocation group, unless otherwise stated) execute the same dynamic instance of an instruction. 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 structured loop or selection, and all invocations leave that loop or selection via the header block's declared merge block, then control flow reconverges to be uniform at that merge block.

2.2.6. Validity and Defined Behavior

Most SPIR-V rules are expressed statically. These *statically expressed rules* are based on what can be seen with a direct static examination of the module in the specific places the rule says to look. These are expressed using terms like *must*, *must not*, *valid*, *not valid*, and *invalid*. Such rules establish whether the module is classified as valid or not valid, which in turn provides terms that tools may use in labeling and describing modules they process. A module is valid only if it does not violate any of these statically expressed rules. Such rules might not be considered violated if a specialization constant is involved, as described in the specialization constant section.

Some SPIR-V rules say that *behavior is not defined*, that something results in *undefined behavior*, or that *behavior is defined* only under some circumstances. These all refer only to something that happens dynamically while an invocation of a shader or kernel executes.

An invocation having undefined behavior is independent of a module being valid. Tools containing smart transforms may be able to deduce from a static module that behavior will be undefined if some part were to be executed. However, this does not allow the tool to classify the module as invalid.

Sometimes, SPIR-V refers to the client API to specify what is statically valid or dynamically defined for a specific situation, in which case those rules come from the client API's execution environment. Otherwise, a SPIR-V client API can define an execution environment that adds additional statically expressed rules, further constraining what SPIR-V itself said was valid. However, a client cannot remove any such statically expressed rules. A client will not remove any undefined behavior specified by SPIR-V.

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 1.3 is the value 0x00010300.
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://github.com/KhronosGroup/SPIRV-Headers.
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)

Instruction Word Number	Contents
…	…
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** or **OpExecutionModeId**.
7. These debug instructions, which must be grouped in the following order:
   a. All **OpString**, **OpSourceExtension**, **OpSource**, and **OpSourceContinued**, without forward references.
   b. All **OpName** and all **OpMemberName**.
   c. All **OpModuleProcessed** instructions.
8. All annotation instructions:
   a. All decoration instructions.
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:
   a. **OpLine** and **OpNoLine** debug information.
   b. Non-semantic instructions with **OpExtInst**.
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 termination 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 intermixed **OpLine** and **OpNoLine** 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.
- **OpPhi** can contain forward references.
- **OpTypeForwardPointer**:
  - An **OpTypeForwardPointer** *Pointer Type* is a forward reference to an **OpTypePointer**.
  - Subsequent consumption of an **OpTypeForwardPointer** *Pointer Type* can be a forward reference.
- The list of *<id>* provided in the **OpEntryPoint** instruction.
- **OpExecutionModeId**.

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.

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 an operand is (or is part of) a self-contained literal or an *<id>* referring to another instruction's result. While an *<id>* always takes one operand, one literal takes one or more operands. Some common examples of literals:

- A literal 32-bit (or smaller) integer is always one operand directly holding a 32-bit two's-complement value.

- A literal 32-bit float is always one operand, directly holding a 32-bit IEEE 754 floating-point representation.
- A literal 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. If 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 if 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 their sampling result 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 client 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 invalid to declare multiple non-aggregate, non-pointer type *<id>s* having the same opcode and operands. 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.) Pointer types are also allowed to have multiple *<id>s* for the same opcode and operands, to allow for differing decorations (e.g., **Volatile**) or different decoration values (e.g., different *Array Stride* values for the **ArrayStride**). If new pointers are formed, their types must be decorated as needed, so the consumer knows how to generate an access through the pointer.

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 is constrained by each instruction's description.

Built-in variables have special semantics and are declared using **OpDecorate** or **OpMemberDecorate** with the **BuiltIn** Decoration, followed by a **BuiltIn** enumerant. See the BuiltIn section for details on what can be decorated as a built-in variable.

2.8.1. Unsigned Versus Signed Integers

The integer type, **OpTypeInt**, is parameterized not only with a size, but also with signedness. There are two different ways to think about signedness in SPIR-V, both are internally consistent and acceptable:

1. As if all integers are "signless", meaning they are neither signed nor unsigned: All **OpTypeInt** instructions select a signedness of 0 to conceptually mean "no sign" (rather than "unsigned"). This is useful if translating from a language that does not distinguish between signed and unsigned types. The

type of operation (signed or unsigned) to perform is always selected by the choice of opcode.

2. As if some integers are signed, and some are unsigned: Some **OpTypeInt** instructions select signedness of 0 to mean "unsigned" and some select signedness of 1 to mean "signed". This is useful if signedness matters to external interface, or if targeting a higher-level language that cares about types being signed and unsigned. The type of operation (signed or unsigned) to perform is still always selected by the choice of opcode, but a small amount of validation can be done where it is non-sensible to use a signed type.

Note in both cases all signed and unsigned operations always work on unsigned types, and the semantics of operation come from the opcode. SPIR-V does not know which way is being used; it is set up to support both ways of thinking.

Note that while SPIR-V aims to not assign semantic meaning to the signedness bit in choosing how to operate on values, there are a few cases known to do this, all confined to modules declaring the **Shader** capability:

- validation for consistency checking for front ends for directly contradictory usage, where explicitly indicated in this specification
- interfaces that might require widening of an input value, and otherwise don’t know whether to sign extend or zero extend, including the following bullet
- an image read that might require widening of an operand, in versions where the **SignExtend** and **ZeroExtend** image operands are not available (if available, these operands are the supported way to communicate this).

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 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, not in this specification. The separate extended instruction set specification specifies 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"
```

Where "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 provide semantics for each "instruction-number". It is up to the specific specification what the overloading rules are on operand type. The specification will 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 are 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. (Control flow may partially or fully reconverge before reaching the merge block so long as it converges by the time the merge block is reached.) These blocks delimit constructs that must nest, and must 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 must not be a merge block declared by any other header block
- each header block must strictly 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 header, its **OpLoopMerge** *Continue Target*, and corresponding back-edge block:
  - the *loop header* must dominate the *Continue Target*, unless the *Continue Target* is unreachable in the CFG
  - the *Continue Target* must dominate the back-edge block
  - the back-edge block must post dominate the *Continue Target*

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

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

Furthermore, these structured control-flow constructs are additionally defined to exclude all outer constructs' continue constructs and exclude all blocks dominated by all outer constructs' merge blocks.

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

- if a construct contains another header block, it also contains that header's corresponding merge block if that merge block is reachable in the CFG
- a continue construct must include its loop's back-edge block
- a break block is valid only for the innermost loop it is nested inside of
- a continue block is valid only for the innermost loop it is nested inside of
- a branch to an outer **OpSwitch** merge block is
  - valid only for the innermost **OpSwitch** the branch is nested inside of
  - not valid if it is nested in a loop that is nested in that **OpSwitch**
- a branch from one case construct to another must be for the same **OpSwitch**
- all branches into a construct from reachable blocks outside the construct must be to the header 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
  - none of the *Targets* nor the *Default* may be the declared merge block or *Continue Target* of a loop the switch is nested within

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. These can be replaced in place, leaving everything else in the module exactly the same:

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

Note that the **OpSpecConstantOp** instruction is not one that can be updated with a specialization constant.

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) will 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.

If a statically expressed rule would be broken due to the value of a constant, and that constant is a specialization constant, then that rule is not violated. (Consequently, specialization-constant default values are not relevant to the validity of the module.)

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*, interpreted is a literal string, 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 includes 0.0 and $[2^{-14}, 2^{14})$
- the relative floating point precision may be as small as 2^{-10}

The range notation here means the largest required magnitude is half of the relative precision less than the value given.

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 in the required range:

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

It is therefore twice the maximum rounding error when converting from a real number. Subnormal numbers may be supported and may have lower relative precision.

For integer operations, operating at relaxed precision means that the operation is evaluated by an operation in which, for some N, 16 ¬ N ¬ 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 must only be applied to:

- The *<id>* of an **OpVariable**, where it refers to the value of the variable.
- The *<id>* of an **OpFunctionParameter**, where it refers to the value of the parameter.
- The *Result <id>* of an instruction that reads or filters from an image. E.g. **OpImageSampleExplicitLod**, meaning the instruction is to operate at relaxed precision.
- The *Result <id>* of an **OpFunction**, where it refers to the value returned by the function.
- A structure-type member (through OpMemberDecorate).
- The *Result <id>* of an **OpFunctionCall**, where it refers to the result of the function call.
- The *Result <id>* of other instructions that operate on numerical types, meaning the instruction is to operate at relaxed precision. The instruction's operands may also be truncated to the relaxed precision.

In all cases, the types of the values that the **RelaxedPrecision** Decoration refers to must be:

- a scalar, vector, or matrix, or array of scalars, vectors, or matrices, and all the components in the types must be a 32-bit numerical type,
- a pointer to such a type, where it refers to the value pointed to.

The values that the **RelaxedPrecision** Decoration refers to can be truncated to relaxed precision.

When applied to a variable, function parameter, 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.)

For image-sampling operations, decorations can appear on both the sampling instruction and the image variable being sampled. If either is decorated, they both should be decorated, and if both are decorated their decorations must match. If only one is decorated, the sampling instruction can behave either as if both were decorated or neither were decorated.

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** and **OpNoLine**.

A module does 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".

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

- When using **OpBitcast** to convert pointers to/from vectors of integers, only vectors of 32-bit integers are allowed.
- If neither the **VariablePointers** nor **VariablePointersStorageBuffer** capabilities are declared, the following rules apply to logical pointer types:
  - **OpVariable** must not allocate an object whose type is or contains a logical pointer type.
  - It is invalid for a pointer to be an operand to any instruction other than:
    - **OpLoad**
    - **OpStore**
    - **OpAccessChain**
    - **OpInBoundsAccessChain**
    - **OpFunctionCall**
    - **OpImageTexelPointer**
    - **OpCopyMemory**
    - **OpCopyObject**
    - all **OpAtomic** instructions
    - extended instruction-set instructions that are explicitly identified as taking pointer operands
  - It is invalid for a pointer to be the *Result <id>* of any instruction other than:
    - **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.
  - Any pointer operand to an **OpFunctionCall** must point into one of the following storage classes:
    - **UniformConstant**
    - **Function**
    - **Private**

    - **Workgroup**
    - **AtomicCounter**
  - Any pointer operand to an **OpFunctionCall** must be
    - a memory object declaration, or
    - a pointer to an element in an array that is a memory object declaration, where the element type is **OpTypeSampler** or **OpTypeImage**.
  - The instructions **OpPtrEqual** and **OpPtrNotEqual** must not be used.
- If the **VariablePointers** or **VariablePointersStorageBuffer** capability is declared, the following are additionally allowed for logical pointer types, while other prohibitions remain:
  - If **OpVariable** allocates an object whose type is or contains a logical pointer type, the *Storage Class* operand of the **OpVariable** must be one of the following:
    - **Function**
    - **Private**
  - If a pointer is the *Object* operand of **OpStore** or result of **OpLoad**, the storage class the pointer is stored to or loaded from must be one of the following:
    - **Function**
    - **Private**
  - A pointer type can be the:
    - *Result Type* of **OpFunction**
    - *Result Type* of **OpFunctionCall**
    - *Return Type* of **OpTypeFunction**
  - A pointer can be a *variable pointer*
  - A pointer can be an operand to one of:
    - **OpReturnValue**
    - **OpPtrAccessChain**
    - **OpPtrEqual**
    - **OpPtrNotEqual**
    - **OpPtrDiff**
  - A *variable pointer* must point to one of the following storage classes:
    - **StorageBuffer**
    - **Workgroup** (if the **VariablePointers** capability is declared)
  - If the **VariablePointers** capability is not declared, a variable pointer must be selected from pointers pointing into the same structure or be **OpConstantNull**.
  - A pointer operand to **OpFunctionCall** can point into the storage class:
    - **StorageBuffer**
  - For pointer operands to **OpFunctionCall**, the memory object declaration-restriction is removed for the following storage classes:
    - **StorageBuffer**
    - **Workgroup**
  - The instructions **OpPtrEqual** and **OpPtrNotEqual** can be used only if the *Storage Class* of the

operands' **OpTypePointer** declaration is

    · **StorageBuffer** if the **VariablePointersStorageBuffer** capability is explicitly or implicitly declared, whether or not operands point into the same buffer, or

    · **Workgroup**, which can be used only if the **VariablePointers** capability was declared.

- A *variable pointer* must not:
  - be an operand to an **OpArrayLength** instruction
  - point to an object that is or contains an **OpTypeMatrix**
  - point to a column, or a component in a column, within an **OpTypeMatrix**
- Memory model
  - If **OpLoad**, **OpStore**, **OpCopyMemory**, or **OpCopyMemorySized** use **MakePointerAvailable** or **MakePointerVisible**, the optional scope operand must be present.
  - If **OpImageRead**, **OpImageSparseRead**, or **OpImageWrite** use **MakeTexelAvailable** or **MakeTexelVisible**, the optional scope operand must be present.
  - Memory accesses that use **NonPrivatePointer** must use pointers in the **Uniform**, **Workgroup**, **CrossWorkgroup**, **Generic**, **Image**, or **StorageBuffer** storage classes.
  - If the **Vulkan** memory model is declared and any instruction uses **Device** scope, the **VulkanMemoryModelDeviceScope** capability must be declared.
- Physical storage buffer
  - If the addressing model is not **PhysicalStorageBuffer64**, then the **PhysicalStorageBuffer** storage class must not be used.
  - **OpVariable** must not use the **PhysicalStorageBuffer** storage class.
  - If the type an **OpVariable** points to is a pointer (or array of pointers) in the **PhysicalStorageBuffer** storage class, the **OpVariable** must be decorated with exactly one of **AliasedPointer** or **RestrictPointer**.
  - If an **OpFunctionParameter** is a pointer (or array of pointers) in the **PhysicalStorageBuffer** storage class, the function parameter must be decorated with exactly one of **Aliased** or **Restrict**.
  - If an **OpFunctionParameter** is a pointer (or array of pointers) and the type it points to is a pointer in the **PhysicalStorageBuffer** storage class, the function parameter must be decorated with exactly one of **AliasedPointer** or **RestrictPointer**.
  - Any pointer value whose storage class is **PhysicalStorageBuffer** and that points to a matrix, an array of matrices, or a row or element of a matrix must be the result of an **OpAccessChain** or **OpPtrAccessChain** instruction whose *Base* operand is a structure type (or recursively must be the result of a sequence of only access chains from a structure to the final value). Such a pointer must only be used as the *Pointer* operand to **OpLoad** or **OpStore**.
  - The result of **OpConstantNull** must not be a pointer into the **PhysicalStorageBuffer** storage class.
  - Operands to **OpPtrEqual**, **OpPtrNotEqual**, and **OpPtrDiff** must not be pointers into the **PhysicalStorageBuffer** storage class.
- 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 operand and return types are already known at the call site.

    - An **OpPhi** can consume definitions that do not dominate it.

- Entry Point
  - There is at least one **OpEntryPoint** instruction, unless the **Linkage** capability is being used.
  - It is invalid for any function to be targeted by both an **OpEntryPoint** instruction and an **OpFunctionCall** instruction.
  - Each **OpEntryPoint** must not set more than one of the **DenormFlushToZero** or **DenormPreserve** execution modes for any given *Target Width*.
  - Each **OpEntryPoint** must not set more than one of the **RoundingModeRTE** or **RoundingModeRTZ** execution modes for any given *Target Width*.
  - Each **OpEntryPoint** must contain at most one of **LocalSize**, **LocalSizeId**, **LocalSizeHint**, or **LocalSizeHintId** **Execution Modes**.
- 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) must not be decorated with the **Import Linkage Type**.
  - A function must not have both a declaration and a definition (no forward declarations).
- Global (Module Scope) Variables
  - A module-scope **OpVariable** with an *Initializer* operand must not be decorated 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 must not be the target of any branch. (Note this means it has 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 termination instruction.
  - Each Termination instruction must be the last instruction in a block.
  - Each **OpLabel** instruction must be 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 must be parameterized only as 32 bit, plus any additional sizes enabled by capabilities.
  - Scalar integer types must be parameterized only as 32 bit, plus any additional sizes enabled by capabilities.
  - Vector types must be parameterized only with numerical types or the OpTypeBool type.

- Vector types must be parameterized only with 2, 3, or 4 components, plus any additional sizes enabled by capabilities.
- Matrix types must be parameterized only with floating-point types.
- Matrix types must be parameterized only with 2, 3, or 4 columns.
- Specialization constants (see Specialization) are limited to integers, Booleans, floating-point numbers, and vectors of these.
- 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**.
- If instructions dereference a composite to get an image or a sampler, behavior is undefined unless all the dereferencing *Indexes* are dynamically-uniform. Such instructions 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.
- The capabilities **StorageBuffer16BitAccess**, **UniformAndStorageBuffer16BitAccess**, **StoragePushConstant16**, and **StorageInputOutput16** do not generally add 16-bit operations. Rather, they add only the following specific abilities:
  - An **OpTypePointer** pointing to a 16-bit scalar, a 16-bit vector, or a composite containing a 16-bit member can be used as the result type of **OpVariable**, or **OpAccessChain**, or **OpInBoundsAccessChain**.
  - **OpLoad** can load 16-bit scalars, 16-bit vectors, and 16-bit matrices.
  - **OpStore** can store 16-bit scalars, 16-bit vectors, and 16-bit matrices.
  - **OpCopyObject** can be used for 16-bit scalars or composites containing 16-bit members.
  - 16-bit scalars or 16-bit vectors can be used as operands to a width-only conversion instruction to another allowed type (**OpFConvert**, **OpSConvert**, or **OpUConvert**), and can be produced as results of a width-only conversion instruction from another allowed type.
  - A structure containing a 16-bit member can be an operand to **OpArrayLength**.
- The capabilities **StorageBuffer8BitAccess**, **UniformAndStorageBuffer8BitAccess**, and **StoragePushConstant8**, do not generally add 8-bit operations. Rather, they add only the following specific abilities:
  - An **OpTypePointer** pointing to an 8-bit scalar, an 8-bit vector, or a composite containing an 8-bit member can be used as the result type of **OpVariable**, or **OpAccessChain**, or **OpInBoundsAccessChain**.
  - **OpLoad** can load 8-bit scalars and vectors.
  - **OpStore** can store 8-bit scalars and 8-bit vectors.
  - **OpCopyObject** can be used for 8-bit scalars or composites containing 8-bit members.
  - 8-bit scalars and vectors can be used as operands to a width-only conversion instruction to another allowed type (**OpSConvert**, or **OpUConvert**), and can be produced as results of a width-only conversion instruction from another allowed type.
  - A structure containing an 8-bit member can be an operand to **OpArrayLength**.

- Decoration rules
  - The **Linkage Attributes** Decoration must not be applied to functions targeted by an **OpEntryPoint** instruction.
  - A BuiltIn Decoration must be applied only as follows:

        - If 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.)
        - If applied to a structure-type member, that structure type must not be contained as a member of another structure type.
        - There must be no more than one object per Storage Class that contains a structure type containing members decorated with **BuiltIn**, consumed per entry-point.

- **OpLoad** and **OpStore** must consume only objects whose type is a pointer.
- A *Result <id>* resulting from an instruction within a function must be used only 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**
        - **StorageBuffer**
        - **PhysicalStorageBuffer**
        - **Workgroup**
        - **CrossWorkgroup**
        - **Generic**
        - **AtomicCounter**
        - **Image**
        - **Function**
- It is invalid to have a construct that uses the **StorageBuffer** Storage Class and a construct that uses the **Uniform** Storage Class with the **BufferBlock** Decoration in the same SPIR-V module.
- All **XfbStride** **Decorations** must be the same for all objects decorated with the same **XfbBuffer** *XFB Buffer Number*.
- All **Stream** **Decorations** must be the same for all objects decorated with the same **XfbBuffer** *XFB Buffer Number*.
- If the workgroup size is statically specified (using the LocalSize, LocalSizeId execution modes, or the WorkgroupSize BuiltIn), the product of all workgroup size dimensions must not be zero.

2.16.2. Validation Rules for Shader Capabilities

- CFG:
    - Loops must be structured. That is, the target basic block of a back edge must contain an **OpLoopMerge** instruction.
    - Selections must be structured. That is, an **OpSelectionMerge** instruction is required to precede:
        - an **OpSwitch** instruction

- an **OpBranchConditional** instruction that has different *True Label* and *False Label* operands where neither are declared merge blocks or *Continue Targets*.

- 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 must not set more than one of the **DepthGreater**, **DepthLess**, or **DepthUnchanged** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Models must not set more than one of the **SpacingEqual**, **SpacingFractionalEven**, or **SpacingFractionalOdd** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Models must not set more than one of the **Triangles**, **Quads**, or **Isolines** **Execution Modes**.
  - An **OpEntryPoint** with one of the **Tessellation** Execution Models must not set more than 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 **StorageBuffer**, **PhysicalStorageBuffer**, **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, unless it is an array that contains a structure decorated with **Block** or **BufferBlock**, in which case it must not have an **ArrayStride** decoration.
  - Each structure-type member that is a matrix or array-of-matrices must 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). Each **ArrayStride** and **MatrixStride** must be greater than zero, and it is invalid for two members of a given structure to be assigned the same **Offset**.
  - Each **OpPtrAccessChain** must have a *Base* whose type is decorated with **ArrayStride**.
  - If an array-element pointer is derived from an array (e.g., using **OpAccessChain**), and the resulting element-pointer type is decorated with **ArrayStride**, its *Array Stride* must match the *Array Stride* of the array's type. If the array's type is not decorated with **ArrayStride**, the derived array-element pointer also must not be decorated with **ArrayStride**.

- For structure objects in the **Input** and **Output** Storage Classes, the following apply:
  - If applied to structure-type members, the decorations **Noperspective**, **Flat**, **Patch**, **Centroid**, and **Sample** must be applied only to the top-level members of the structure type. (Nested objects' types must not be structures whose members are decorated with these decorations.)

- Type Rules
  - All declared types are restricted to those types that are, or are contained within, valid types for an **OpVariable** *Result Type* or an **OpTypeFunction** *Return Type*.

- Aggregate types for *intermediate objects* are restricted to those types that are a valid *Type* of an **OpVariable** *Result Type* in the global storage classes.

- Decorations
  - It is invalid to apply more than one of **Noperspective** or **Flat** decorations to the same object or member.
  - It is invalid to apply more than one of **Patch**, **Centroid**, or **Sample** decorations to the same object or member.
  - It is invalid to apply more than one of **Block** and **BufferBlock** decorations to a structure type.
  - **Block** and **BufferBlock** decorations must not decorate a structure type that is nested at any level inside another structure type decorated with **Block** or **BufferBlock**.
  - The **FPRoundingMode** decoration must be applied only to a width-only conversion instruction whose only uses are *Object* operands of **OpStore** instructions storing through a pointer to a 16-bit floating-point object in the **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, or **Output** Storage Classes.
- All <*id*> used for Scope <*id*> and Memory Semantics <*id*> must be of an OpConstant.
- Atomic access rules
  - The pointers taken by atomic operation instructions are further restricted to not point into the **Function** storage class.

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. Client APIs may impose larger minimums. See Language Capabilities.

Validators inform when these limits (or explicitly parameterized limits) are crossed.

Table 3. Limits

Limited Entity	**Minimum Limit**	
	Decimal	**Hexadecimal**
Characters in a literal string	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 must be created only from existing objects, and they must not be stored into an object, unless additional capabilities, e.g., **VariablePointers**, are declared to add such functionality.

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 must not be used to access objects in another Storage Class. However, they can be converted with conversion opcodes. Any particular addressing model describes the bit width of pointers for each of the storage classes.

2.18.1. Memory Layout

Offset, **MatrixStride**, and **ArrayStride** Decorations partially define how a memory buffer is laid out. In addition, the following also define layout of a memory buffer, applied recursively as needed:

- 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
- 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

Two memory object declarations are said to *alias* if they can be accessed (in bounds) such that both accesses address the same memory locations. If two memory operations access the same locations, and at least one of them performs a write, the memory consistency model specified by the client API defines the results based on the ordering of the accesses.

How aliasing is managed depends on the memory model:

- The **Simple**, **GLSL**, and **Vulkan** memory models can assume that aliasing is generally not present between the memory object declarations. Specifically, the consumer is free to assume aliasing is not present between memory object declarations, unless the memory object declarations explicitly indicate they alias. Aliasing is indicated by applying the **Aliased** decoration to a memory object declaration's *<id>*, for **OpVariable** and **OpFunctionParameter**. Applying **Restrict** is allowed, but has no effect. For variables holding **PhysicalStorageBuffer** pointers, applying the **AliasedPointer** decoration on the **OpVariable** indicates that the **PhysicalStorageBuffer** pointers are potentially aliased. Applying **RestrictPointer** is allowed, but has no effect. Variables holding **PhysicalStorageBuffer** pointers must be decorated as either **AliasedPointer** or **RestrictPointer**. Only those memory object declarations decorated with **Aliased** or **AliasedPointer** may alias each other.
- The **OpenCL** memory model assumes that memory object declarations might alias each other. An implementation may assume that memory object declarations decorated with **Restrict** will not alias any other memory object declaration. Applying **Aliased** is allowed, but has no effect.

The **Aliased** decoration can be used to express that certain memory object declarations may alias. Referencing the following table, a memory object declaration *P* may alias another declared pointer *Q* if within a single row:

- *P* is an instruction with opcode and storage class from the first pair of columns, and

- *Q* is an instruction with opcode and storage class from the second pair of columns.

First Storage Class	First Instruction(s)	Second Instructions	Second Storage Classes
CrossWorkgroup	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**CrossWorkgroup**, **Generic**
Function	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Function**, **Generic**
Function	**OpVariable**	**OpFunctionParameter**	**Function**, **Generic**
Generic	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**CrossWorkgroup**, **Function**, **Generic**, **Workgroup**
Image	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, **UniformConstant**
Output	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Output**
Private	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Private**
StorageBuffer	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, **UniformConstant**
PhysicalStorageBuffer	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, **UniformConstant**
Uniform	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, **UniformConstant**
UniformConstant	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, **UniformConstant**
Workgroup	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Workgroup**, **Generic**
Workgroup	**OpVariable**	**OpFunctionParameter**	**Workgroup**, **Generic**

In addition to the above table, memory object declarations in the **CrossWorkgroup**, **Function**, **Input**, **Output**, **Private**, or **Workgroup** storage classes must also have matching pointee types for aliasing to be present. In all other cases the decoration is ignored.

Because aliasing, as described above, only applies to memory object declarations, a consumer does not make any assumptions about whether or not memory regions of non memory object declarations overlap. As such, a consumer needs to perform dependency analysis on non memory object declarations if it wishes to reorder instructions affecting memory. Behavior is undefined if operations on two memory object declarations access the same memory location, with at least one of them performing a write, and at least one of the memory object declarations does not have the **Aliased** decoration.

For the **PhysicalStorageBuffer** storage class, **OpVariable** is understood to mean the **PhysicalStorageBuffer** pointer value(s) stored in the variable. An **Aliased PhysicalStorageBuffer** pointer stored in a **Function** variable can alias with other variables in the same function, global variables, or function parameters.

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

2.18.3. Null pointers

A "null pointer" can be formed from an **OpConstantNull** instruction with a pointer result type. The resulting pointer value is abstract, and will not equal the pointer value formed from any declared object or access chain into a declared object. Behavior is undefined if a load or store through **OpConstantNull** is executed.

2.19. Derivatives

Derivatives appear only in the **Fragment** Execution Model. They are either implicit or explicit. Some image instructions consume implicit derivatives, while the derivative instructions compute explicit derivatives. In all cases, derivatives are well defined when the derivative group has uniform control flow, otherwise see the client API specification for what behavior is allowed.

2.20. Code Motion

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

2.21. Deprecation

A feature may be marked as deprecated by a version of the specification or extension to the specification. Features marked as deprecated in one version of the specification are still present in that version, but future versions may reduce their support or completely remove them. Deprecating before removing allows applications time to transition away from the deprecated feature. Once the feature is removed, all tokens used exclusively by that feature will be reserved and any use of those tokens will become invalid.

2.22. Unified Specification

This document specifies all versions of **SPIR-V**.

There are three kinds of entries in the tables of enumerated tokens:

- **Reservation:** These say Reserved in the enabling capabilities. They often contain token names only, lacking a semantic description. They are invalid **SPIR-V** for any version, serving only to reserve the tokens. They may identify enabling capabilities and extensions, in which case any listed extensions might add the tokens. See the listed extensions for additional information.
- **Conditional:** These say Missing before or Missing after in the enabling capabilities. They are invalid **SPIR-V** for the missing versions. They may identify enabling capabilities and extensions, in which case any listed extensions might add the tokens for some of the missing versions. See the listed extensions for additional information. For versions not identified as missing, the tokens are valid **SPIR-V**, subject to any listed enabling capabilities.

- **Universal:** These have no mention of what version they are missing in, or of being reserved. They are valid in all versions of **SPIR-V**.

2.23. Uniformity

SPIR-V has multiple notions of uniformity of values. A *Result <id>* decorated as **Uniform** (for a particular scope) is a contract that all invocations within that scope compute the same value for that result, for a given dynamic instance of an instruction. This is useful to enable implementations to store results in a scalar register file (*scalarization*), for example. Results are assumed not to be uniform unless decorated as such.

An *<id>* is defined to be dynamically uniform for a dynamic instance of an instruction if all invocations (in an invocation group) that execute the dynamic instance have the same value for that *<id>*. This is not something that is explicitly decorated, it is just a property that arises. This property is assumed to hold for operands of certain instructions, such as the *Image* operand of image instructions, unless that operand is decorated as **NonUniform**. Some implementations require more complex instruction expansions to handle non-dynamically uniform values in certain instructions, and thus it is mandatory for certain operands to be decorated as **NonUniform** if they are not guaranteed to be dynamically uniform.

While the names may suggest otherwise, nothing forbids an *<id>* from being decorated as both **Uniform** and **NonUniform**. Because *dynamically uniform* is at a larger scope (invocation group) than the default **Uniform** scope (subgroup), it is even possible for the *<id>* to be uniform at the subgroup scope but not dynamically uniform.

Chapter 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**
5	**HLSL**
6	**CPP_for_OpenCL**

3.3. Execution Model

Used by **OpEntryPoint**.

Execution Model		**Enabling Capabilities**
0	**Vertex** Vertex shading stage.	**Shader**
1	**TessellationControl** Tessellation control (or hull) shading stage.	**Tessellation**
2	**TessellationEvaluation** Tessellation evaluation (or domain) shading stage.	**Tessellation**

Execution Model		Enabling Capabilities
3	**Geometry** Geometry shading stage.	**Geometry**
4	**Fragment** Fragment shading stage.	**Shader**
5	**GLCompute** Graphical compute shading stage.	**Shader**
6	**Kernel** Compute kernel.	**Kernel**
5267	**TaskNV**	**MeshShadingNV** Reserved.
5268	**MeshNV**	**MeshShadingNV** Reserved.
5313	**RayGenerationNV**	**RayTracingNV**, **RayTracingKHR** Reserved.
5313	**RayGenerationKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.
5314	**IntersectionNV**	**RayTracingNV**, **RayTracingKHR** Reserved.
5314	**IntersectionKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.
5315	**AnyHitNV**	**RayTracingNV**, **RayTracingKHR** Reserved.
5315	**AnyHitKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.
5316	**ClosestHitNV**	**RayTracingNV**, **RayTracingKHR** Reserved.
5316	**ClosestHitKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.
5317	**MissNV**	**RayTracingNV**, **RayTracingKHR** Reserved.

Execution Model		Enabling Capabilities
5317	**MissKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.
5318	**CallableNV**	**RayTracingNV**, **RayTracingKHR** Reserved.
5318	**CallableKHR**	**RayTracingNV**, **RayTracingKHR** Reserved.

3.4. Addressing Model

Used by **OpMemoryModel**.

Addressing Model		Enabling Capabilities
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**
5348	**PhysicalStorageBuffer64** Indicates that pointers with a storage class of **PhysicalStorageBuffer** are physical pointer types with an address width of 64 bits, while pointers to all other storage classes are logical.	**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extensions: **SPV_EXT_physical_storage_buffer**, **SPV_KHR_physical_storage_buffer**
5348	**PhysicalStorageBuffer64EXT**	**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extension: **SPV_EXT_physical_storage_buffer**

3.5. Memory Model

Used by **OpMemoryModel**.

Memory Model		Enabling Capabilities
0	**Simple** No shared memory consistency issues.	**Shader**

	Memory Model	Enabling Capabilities
1	**GLSL450** Memory model needed by later versions of GLSL and ESSL. Works across multiple versions.	**Shader**
2	**OpenCL** OpenCL memory model.	**Kernel**
3	**Vulkan** **Vulkan memory model**, as specified by the client API. This memory model must be declared if and only if the **VulkanMemoryModel** capability is declared.	**VulkanMemoryModel** Missing before **version 1.5**.
3	**VulkanKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**

3.6. Execution Mode

Declare the modes an entry point executes in.

Used by **OpExecutionMode** and **OpExecutionModeId**.

	Execution Mode	Extra Operands	Enabling Capabilities
0	**Invocations** *Number of invocations* is an unsigned 32-bit integer number of times to invoke the geometry stage for each input primitive received. The default is to run once for each input primitive. It is invalid to specify a value greater than the target-dependent maximum. Only valid with the **Geometry** Execution Model.	*Literal* *Number of invocations*	**Geometry**
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**

	Execution Mode	Extra Operands	Enabling Capabilities
2	**SpacingFractionalEven** Requests the tessellation primitive generator 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.		**Tessellation**
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** The coordinates decorated by **FragCoord** appear to originate in the upper left, and increase toward the right and downward. Only valid with the **Fragment** Execution Model.		**Shader**

	Execution Mode	Extra Operands	Enabling Capabilities
8	**OriginLowerLeft** The coordinates decorated by **FragCoord** 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 runs in transform feedback-capturing mode and this module is responsible for describing the transform-feedback setup. See the **XfbBuffer**, **Offset**, and **XfbStride** **Decorations**.		**TransformFeedback**
12	**DepthReplacing** This mode declares that this entry point dynamically writes the **FragDepth**-decorated variable. Behavior is undefined if this mode is declared and an invocation does not write to **FragDepth**, or vice versa. Only valid with the **Fragment** Execution Model.		**Shader**
14	**DepthGreater** Indicates that per-fragment tests may assume that any **FragDepth** built in-decorated value written by the shader is greater-than-or-equal to the fragment's interpolated depth value (given by the *z* component of the **FragCoord** built in-decorated variable). Other stages of the pipeline use the written value as normal. Only valid with the **Fragment** execution model.		**Shader**

	Execution Mode	Extra Operands			Enabling Capabilities
15	**DepthLess** Indicates that per-fragment tests may assume that any **FragDepth** built in-decorated value written by the shader is less-than-or-equal to the fragment's interpolated depth value (given by the *z* component of the **FragCoord** built in-decorated variable). Other stages of the pipeline use the written value as normal. Only valid with the **Fragment** execution model.				**Shader**
16	**DepthUnchanged** Indicates that per-fragment tests may assume that any **FragDepth** built in-decorated value written by the shader is the same as the fragment's interpolated depth value (given by the *z* component of the **FragCoord** built in -decorated variable). Other stages of the pipeline use the written value as normal. Only valid with the **Fragment** execution model.				**Shader**
17	**LocalSize** Indicates the work-group size in the *x*, *y*, and *z* dimensions. *x size*, *y size*, and *z size* are unsigned 32-bit integers. Only valid with the **GLCompute** or **Kernel** Execution Models.	*Literal* *x size*	*Literal* *y size*	*Literal* *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. *x size*, *y size*, and *z size* are unsigned 32-bit integers. Only valid with the **Kernel** Execution Model.	*Literal* *x size*	*Literal* *y size*	*Literal* *z size*	**Kernel**
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**

Execution Mode		Extra Operands	Enabling Capabilities
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** Execution Model.		**Geometry**
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** *Vertex Count* is an unsigned 32-bit integer. For a geometry stage, it is the maximum number of vertices the shader will ever emit in a single invocation. For a tessellation-control stage, it is 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.	*Literal* *Vertex count*	**Geometry**, **Tessellation**, **MeshShadingNV**
27	**OutputPoints** Stage output primitive is *points*. Only valid with the **Geometry** Execution Model.		**Geometry**, **MeshShadingNV**

	Execution Mode	Extra Operands	Enabling Capabilities
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 the *Vector Type* operand specify the *number of components* of the vector. The 16 low-order bits of the *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.	*Literal* *Vector type*	**Kernel**
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** Missing before **version 1.1**.
34	**Finalizer** Indicates that this entry point is a module finalizer.		**Kernel** Missing before **version 1.1**.

	Execution Mode	Extra Operands			Enabling Capabilities
35	**SubgroupSize** Indicates that this entry point requires the specified *Subgroup Size*. *Subgroup Size* is an unsigned 32-bit integer.	*Literal* *Subgroup Size*			**SubgroupDispatch** Missing before **version 1.1**.
36	**SubgroupsPerWorkgroup** Indicates that this entry point requires the specified number of *Subgroups Per Workgroup*. *Subgroups Per Workgroup* is an unsigned 32-bit integer.	*Literal* *Subgroups Per Workgroup*			**SubgroupDispatch** Missing before **version 1.1**.
37	**SubgroupsPerWorkgroupId** Same as the **SubgroupsPerWorkgroup** mode, but using an *<id>* operand instead of a literal. The operand is consumed as unsigned and must be an *integer type* scalar.	*<id>* *Subgroups Per Workgroup*			**SubgroupDispatch** Missing before **version 1.2**.
38	**LocalSizeId** Same as the **LocalSize** Mode, but using *<id>* operands instead of literals. The operands are consumed as unsigned and each must be an *integer type* scalar.	*<id>* *x size*	*<id>* *y size*	*<id>* *z size*	Missing before **version 1.2**.
39	**LocalSizeHintId** Same as the **LocalSizeHint** Mode, but using *<id>* operands instead of literals. The operands are consumed as unsigned and each must be an *integer type* scalar.	*<id>* *x size hint*	*<id>* *y size hint*	*<id>* *z size hint*	**Kernel** Missing before **version 1.2**.
4421	**SubgroupUniformControlFlowKHR**				**Shader** Reserved. Also see extension: **SPV_KHR_subgroup_uniform_control_flow**
4446	**PostDepthCoverage**				**SampleMaskPostDepthCoverage** Reserved. Also see extension: **SPV_KHR_post_depth_coverage**

Execution Mode		Extra Operands	Enabling Capabilities
4459	**DenormPreserve** Any denormalized value input into a shader or potentially generated by any instruction in a shader is preserved. Denormalized values obtained via unpacking an integer into a vector of values with smaller bit width and interpreting those values as floating-point numbers is preserved. Only affects instructions operating on a floating-point type whose component width is *Target Width*. *Target Width* is an unsigned 32-bit integer.	*Literal* *Target Width*	**DenormPreserve** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4460	**DenormFlushToZero** Any denormalized value input into a shader or potentially generated by any instruction in a shader is flushed to zero. Denormalized values obtained via unpacking an integer into a vector of values with smaller bit width and interpreting those values as floating-point numbers is flushed to zero. Only affects instructions operating on a floating-point type whose component width is *Target Width*. *Target Width* is an unsigned 32-bit integer.	*Literal* *Target Width*	**DenormFlushToZero** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4461	**SignedZeroInfNanPreserve** The implementation does not perform optimizations on floating-point instructions that do not preserve sign of a zero, or assume that operands and results are not NaNs or infinities. Bit patterns for NaNs might not be preserved. Only affects instructions operating on a floating-point type whose component width is *Target Width*. *Target Width* is an unsigned 32-bit integer.	*Literal* *Target Width*	**SignedZeroInfNanPreserve** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**

Execution Mode		Extra Operands	Enabling Capabilities
4462	**RoundingModeRTE** The default rounding mode for floating-point arithmetic and conversions instructions is round to nearest even. If an instruction is decorated with **FPRoundingMode** or defines a rounding mode in its description, that rounding mode is applied and **RoundingModeRTE** is ignored. Only affects instructions operating on a floating-point type whose component width is *Target Width*. *Target Width* is an unsigned 32-bit integer.	*Literal* *Target Width*	**RoundingModeRTE** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4463	**RoundingModeRTZ** The default rounding mode for floating-point arithmetic and conversions instructions is round toward zero. If an instruction is decorated with **FPRoundingMode** or defines a rounding mode in its description, that rounding mode is applied and **RoundingModeRTZ** is ignored. Only affects instructions operating on a floating-point type whose component width is *Target Width*. *Target Width* is an unsigned 32-bit integer.	*Literal* *Target Width*	**RoundingModeRTZ** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
5027	**StencilRefReplacingEXT**		**StencilExportEXT** Reserved. Also see extension: **SPV_EXT_shader_stencil_export**
5269	**OutputLinesNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5270	**OutputPrimitivesNV**	*Literal* *Primitive count*	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**

Execution Mode		Extra Operands	Enabling Capabilities
5289	**DerivativeGroupQuadsNV**		**ComputeDerivativeGroupQuadsNV** Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5290	**DerivativeGroupLinearNV**		**ComputeDerivativeGroupLinearNV** Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5298	**OutputTrianglesNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5366	**PixelInterlockOrderedEXT**		**FragmentShaderPixelInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5367	**PixelInterlockUnorderedEXT**		**FragmentShaderPixelInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5368	**SampleInterlockOrderedEXT**		**FragmentShaderSampleInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**

	Execution Mode	Extra Operands			Enabling Capabilities
5369	**SampleInterlockUnorderedEXT**				**FragmentShaderSampleInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5370	**ShadingRateInterlockOrderedEXT**				**FragmentShaderShadingRateInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5371	**ShadingRateInterlockUnorderedEXT**				**FragmentShaderShadingRateInterlockEXT** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5618	**SharedLocalMemorySizeINTEL**	*Literal* *Size*			**VectorComputeINTEL** Reserved.
5620	**RoundingModeRTPINTEL**	*Literal* *Target Width*			**RoundToInfinityINTEL** Reserved.
5621	**RoundingModeRTNINTEL**	*Literal* *Target Width*			**RoundToInfinityINTEL** Reserved.
5622	**FloatingPointModeALTINTEL**	*Literal* *Target Width*			**RoundToInfinityINTEL** Reserved.
5623	**FloatingPointModeIEEEINTEL**	*Literal* *Target Width*			**RoundToInfinityINTEL** Reserved.
5893	**MaxWorkgroupSizeINTEL**	*Literal* *max_x_size*	*Literal* *max_y_size*	*Literal* *max_z_size*	**KernelAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_kernel_attributes**

Execution Mode		Extra Operands	Enabling Capabilities
5894	**MaxWorkDimINTEL**	*Literal* *max_dimensions*	**KernelAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_kernel_attributes**
5895	**NoGlobalOffsetINTEL**		**KernelAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_kernel_attributes**
5896	**NumSIMDWorkitemsINTEL**	*Literal* *vector_width*	**FPGAKernelAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_kernel_attributes**
5903	**SchedulerTargetFmaxMhzINTEL**	*Literal* *target_fmax*	**FPGAKernelAttributesINTEL** Reserved.

3.7. Storage Class

Class of storage for declared variables. Intermediate values do not form a storage class, and unless stated otherwise, storage class-based restrictions are not restrictions on intermediate objects and their types.

Used by:

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

Storage Class		Enabling Capabilities
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. They may have initializers, as allowed by the client API.	
1	**Input** Input from pipeline. Visible across all functions in the current invocation. Variables declared with this storage class are read-only, and must not have initializers.	

Storage Class		Enabling Capabilities
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**, **VectorComputeINTEL**
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 client API. Variables declared with this storage class are read-only, and must not have initializers.	**Shader**
10	**AtomicCounter** For holding atomic counters. Visible across all functions of the current invocation. Atomic counter-specific memory.	**AtomicStorage**
11	**Image** For holding image memory.	
12	**StorageBuffer** Shared externally, readable and writable, visible across all functions in all invocations in all work groups. Graphics storage buffers (buffer blocks).	**Shader** Missing before **version 1.3**. Also see extensions: **SPV_KHR_storage_buffer_storage_class**, **SPV_KHR_variable_pointers**

	Storage Class	Enabling Capabilities
5328	**CallableDataNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5328	**CallableDataKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5329	**IncomingCallableDataNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5329	**IncomingCallableDataKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5338	**RayPayloadNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5338	**RayPayloadKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5339	**HitAttributeNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5339	**HitAttributeKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**

Storage Class		Enabling Capabilities
5342	**IncomingRayPayloadNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5342	**IncomingRayPayloadKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5343	**ShaderRecordBufferNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5343	**ShaderRecordBufferKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5349	**PhysicalStorageBuffer** Shared externally, readable and writable, visible across all functions in all invocations in all work groups. Graphics storage buffers using physical addressing.	**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extensions: **SPV_EXT_physical_storage_buffer**, **SPV_KHR_physical_storage_buffer**
5349	**PhysicalStorageBufferEXT**	**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extension: **SPV_EXT_physical_storage_buffer**
5605	**CodeSectionINTEL**	**FunctionPointersINTEL** Reserved. Also see extension: **SPV_INTEL_function_pointers**

Storage Class		Enabling Capabilities
5936	**DeviceOnlyINTEL**	**USMStorageClassesINTEL** Reserved. Also see extension: **SPV_INTEL_usm_storage_classes**
5937	**HostOnlyINTEL**	**USMStorageClassesINTEL** Reserved. Also see extension: **SPV_INTEL_usm_storage_classes**

3.8. Dim

Dimensionality of an image. The listed **Array** capabilities are required if the type's *Arrayed* operand is 1. The listed **Image** capabilities are required if the type's *Sampled* operand is 2.

Used by **OpTypeImage**.

Dim		Enabling Capabilities
0	**1D**	**Sampled1D**, **Image1D**
1	**2D**	**Shader**, **Kernel**, **ImageMSArray**
2	**3D**	
3	**Cube**	**Shader**, **ImageCubeArray**
4	**Rect**	**SampledRect**, **ImageRect**
5	**Buffer**	**SampledBuffer**, **ImageBuffer**
6	**SubpassData**	**InputAttachment**

3.9. Sampler Addressing Mode

Addressing mode for creating constant samplers.

Used by **OpConstantSampler**.

Sampler Addressing Mode		Enabling Capabilities
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**

Sampler Addressing Mode		Enabling Capabilities
2	**Clamp** Out-of-range image coordinates result in a border color.	**Kernel**
3	**Repeat** Out-of-range image coordinates are wrapped to the valid range. Must only be used with normalized coordinates.	**Kernel**
4	**RepeatMirrored** Flip the image coordinate at every integer junction. Must only be used with normalized coordinates.	**Kernel**

3.10. Sampler Filter Mode

Filter mode for creating constant samplers.

Used by **OpConstantSampler**.

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

Image Format		Enabling Capabilities
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
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
40	R64ui	Int64ImageEXT
41	R64i	Int64ImageEXT

3.12. Image Channel Order

The image channel orders that result from **OpImageQueryOrder**.

Image Channel Order		Enabling Capabilities
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**
16	**sRGBx**	**Kernel**
17	**sRGBA**	**Kernel**
18	**sBGRA**	**Kernel**
19	**ABGR**	**Kernel**

3.13. Image Channel Data Type

Image channel data types that result from **OpImageQueryFormat**.

Image Channel Data Type		Enabling Capabilities
0	**SnormInt8**	**Kernel**
1	**SnormInt16**	**Kernel**
2	**UnormInt8**	**Kernel**
3	**UnormInt16**	**Kernel**
4	**UnormShort565**	**Kernel**

Image Channel Data Type		Enabling Capabilities
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

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

Provides additional operands to sampling, or getting texels from, an image. Bits that are set indicate whether an additional operand follows, as described by the table. 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).

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**
- **OpImageSampleFootprintNV**

Image Operands		Enabling Capabilities
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 must 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 fetch operations, it must be an *integer type* scalar. This must 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 present. Only valid with explicit-lod instructions. They must be a scalar or vector of *floating-point type*. This must only be used with an **OpTypeImage** that has an *MS* operand of 0. It is invalid to set both the **Lod** and **Grad** bits.	

Image Operands		Enabling Capabilities
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 invalid for these to be 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. Not valid with the **Cube** dimension. An instruction must specify at most one of the **ConstOffset**, **Offset**, and **ConstOffsets** image operands.	
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 invalid for these to be 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. Not valid with the **Cube** dimension. An instruction must specify at most one of the **ConstOffset**, **Offset**, and **ConstOffsets** image operands.	**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 invalid for these to be outside a target-dependent allowed range. Only valid with **OpImageGather** or **OpImageDrefGather**. Not valid with the **Cube** dimension. An instruction must specify at most one of the **ConstOffset**, **Offset**, and **ConstOffsets** image operands.	**ImageGatherExtended**
0x40	**Sample** A following operand is the sample number of the sample to use. Only valid with **OpImageFetch**, **OpImageRead**, **OpImageWrite**, **OpImageSparseFetch**, and **OpImageSparseRead**. The **Sample** operand must be used if and only if the underlying **OpTypeImage** has *MS* of 1. It must be an *integer type* scalar.	

Image Operands		Enabling Capabilities
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 must 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**
0x100	**MakeTexelAvailable** Perform an availability operation on the texel locations after the store. A following operand is the memory scope that controls the availability operation. Requires **NonPrivateTexel** to also be set. Only valid with **OpImageWrite**.	**VulkanMemoryModel** Missing before **version 1.5**.
0x100	**MakeTexelAvailableKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x200	**MakeTexelVisible** Perform a visibility operation on the texel locations before the load. A following operand is the memory scope that controls the visibility operation. Requires **NonPrivateTexel** to also be set. Only valid with **OpImageRead** and **OpImageSparseRead**.	**VulkanMemoryModel** Missing before **version 1.5**.
0x200	**MakeTexelVisibleKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x400	**NonPrivateTexel** The image access obeys inter-thread ordering, as specified by the client API.	**VulkanMemoryModel** Missing before **version 1.5**.
0x400	**NonPrivateTexelKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x800	**VolatileTexel** This access cannot be eliminated, duplicated, or combined with other accesses.	**VulkanMemoryModel** Missing before **version 1.5**.

Image Operands		Enabling Capabilities
0x800	**VolatileTexelKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x1000	**SignExtend** The texel value is converted to the target value via sign extension. Only valid if the result type is a scalar or vector of *integer type*.	Missing before **version 1.4**.
0x2000	**ZeroExtend** The texel value is converted to the target value via zero extension. Only valid if the result type is a scalar or vector of *integer type* with signedness of 0.	Missing before **version 1.4**.
0x4000	**Nontemporal** Hints that the accessed texels are not likely to be accessed again in the near future.	Missing before **version 1.6**.
0x10000	**Offsets**	

3.15. FP Fast Math Mode

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

Enables fast math operations which are otherwise unsafe.

Only valid on

- **OpFAdd**, **OpFSub**, **OpFMul**, **OpFDiv**, **OpFRem**, and **OpFMod** instructions
- Missing before **version 1.6**:
  - the **OpFNegate** instruction
  - the **OpOrdered**, **OpUnordered**, **OpFOrdEqual**, **OpFUnordEqual**, **OpFOrdNotEqual**, **OpFUnordNotEqual**, **OpFOrdLessThan**, **OpFUnordLessThan**, **OpFOrdGreaterThan**, **OpFUnordGreaterThan**, **OpFOrdLessThanEqual**, **OpFUnordLessThanEqual**, **OpFOrdGreaterThanEqual**, and **OpFUnordGreaterThanEqual** instructions
  - **OpExtInst** extended instructions, where expressly permitted by the extended instruction set in use.

FP Fast Math Mode		Enabling Capabilities
0x0	**None**	
0x1	**NotNaN** Assume parameters and result are not NaN.	
0x2	**NotInf** Assume parameters and result are not +/- Inf.	

FP Fast Math Mode		Enabling Capabilities
0x4	**NSZ** Treat the sign of a zero parameter or result as insignificant.	
0x8	**AllowRecip** Allow the usage of reciprocal rather than perform a division.	
0x10	**Fast** Allow algebraic transformations according to real-number associative and distributive algebra. This flag implies all the others.	
0x10000	**AllowContractFastINTEL**	**FPFastMathModeINTEL** Reserved.
0x20000	**AllowReassocINTEL**	**FPFastMathModeINTEL** Reserved.

3.16. FP Rounding Mode

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

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

3.17. Linkage Type

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

Linkage Type		Enabling Capabilities
0	**Export** Accessible by other modules as well.	**Linkage**
1	**Import** A declaration of a global variable or a function that exists in another module.	**Linkage**

Linkage Type		Enabling Capabilities
2	**LinkOnceODR**	**Linkage** Reserved. Also see extension: **SPV_KHR_linkonce_odr**

3.18. Access Qualifier

Defines the access permissions.

Used by **OpTypeImage**, **OpTypePipe**, and **OpTypeBufferSurfaceINTEL**.

Access Qualifier		Enabling Capabilities
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		Enabling Capabilities
0	**Zext** Zero extend the value, if needed.	**Kernel**
1	**Sext** Sign extend the value, if needed.	**Kernel**
2	**ByVal** Pass the parameter by value to the function. Only valid for pointer parameters (not for ret value).	**Kernel**
3	**Sret** The parameter is 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 parameter.	**Kernel**
4	**NoAlias** The memory pointed to by a pointer parameter is not accessed via pointer values that are not derived from this pointer parameter. Only valid for pointer parameters. Not valid on return values.	**Kernel**

Function Parameter Attribute		Enabling Capabilities
5	**NoCapture** The parameter is not copied into a location that is accessible after returning from the callee. Only valid for pointer parameters. Not valid on return values.	**Kernel**
6	**NoWrite** The parameter is not used to write to the memory pointed to. Only valid for pointer parameters. Not valid on return values.	**Kernel**
7	**NoReadWrite** The parameter is not dereferenced, either to read or write the memory pointed to. Only valid for pointer parameters. Not valid on return values.	**Kernel**

3.20. Decoration

Used by:

- **OpDecorate**
- **OpMemberDecorate**
- **OpDecorateId**
- **OpDecorateString**
- **OpDecorateStringGOOGLE**
- **OpMemberDecorateString**
- **OpMemberDecorateStringGOOGLE**

Decoration		Extra Operands	Enabling Capabilities
0	**RelaxedPrecision** Allow reduced precision operations. To be used as described in Relaxed Precision.		**Shader**
1	**SpecId** Apply only to a scalar specialization constant. *Specialization Constant ID* is an unsigned 32-bit integer forming the external linkage for setting a specialized value. See specialization.	*Literal Specialization Constant ID*	**Shader**, **Kernel**
2	**Block** Apply only to a structure type to establish it is a memory interface block.		**Shader**

Decoration		Extra Operands	Enabling Capabilities
3	**BufferBlock** Deprecated (use **Block**-decorated **StorageBuffer** Storage Class objects). Apply only to a structure type to establish it is a memory interface block. When the type is used for a variable in the **Uniform** Storage Class the memory interface is a **StorageBuffer**-like interface, distinct from those variables decorated with **Block**. In all other Storage Classes the decoration is meaningless.		**Shader** Missing after **version 1.3**.
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. Must not be used with **ColMajor** on the same matrix or matrix aggregate.		**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. Must not be used with **RowMajor** on the same matrix or matrix aggregate.		**Matrix**
6	**ArrayStride** Apply to an array type to specify the stride, in bytes, of the array's elements. Can also apply to a pointer type to an array element. *Array Stride* is an unsigned 32-bit integer specifying the stride of the array that the element resides in. Must not be applied to any other type.	*Literal* *Array Stride*	**Shader**
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. *Matrix Stride* is an unsigned 32-bit integer specifying the stride of the rows in a **RowMajor** -decorated matrix or columns in a **ColMajor**-decorated matrix.	*Literal* *Matrix Stride*	**Matrix**

Decoration		Extra Operands	Enabling Capabilities
8	**GLSLShared** Apply only to a structure type to get GLSL **shared** memory layout.		**Shader**
9	**GLSLPacked** Apply only to a structure type to get GLSL **packed** memory layout.		**Shader**
10	**CPacked** Apply only 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** Indicates which built-in variable an object represents. See BuiltIn for more information.	*BuiltIn*	
13	**NoPerspective** Must only be used on a memory object declaration or a member of a structure type. Requests linear, non-perspective correct, interpolation. Only valid for the **Input** and **Output** Storage Classes.		**Shader**
14	**Flat** Must only be used on a memory object declaration or a member of a structure type. Indicates no interpolation is done. The non-interpolated value comes from a vertex, as specified by the client API. Only valid for the **Input** and **Output** Storage Classes.		**Shader**
15	**Patch** Must only be used on a memory object declaration 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**

Decoration		Extra Operands	Enabling Capabilities
16	**Centroid** Must only be used on a memory object declaration or a member of a structure type. If used with multi-sampling rasterization, allows a single interpolation location for an entire pixel. The interpolation location lies in both the pixel and in the primitive being rasterized. Only valid for the **Input** and **Output** Storage Classes.		**Shader**
17	**Sample** Must only be used on a memory object declaration or a member of a structure type. If used with multi-sampling rasterization, requires per-sample interpolation. The interpolation locations are 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 only to a variable or member of a block-decorated structure type to indicate that expressions computing its value be computed invariantly with respect to other shaders computing the same expressions.		**Shader**
19	**Restrict** Apply only to a memory object declaration, to indicate the compiler may compile as if there is no aliasing. See the Aliasing section for more detail.		
20	**Aliased** Apply only to a memory object declaration, 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.		

	Decoration	Extra Operands	Enabling Capabilities
21	**Volatile** Must be applied only to memory object declarations or members of a structure type. Any such memory object declaration, or any memory object declaration that contains such a structure type, must be one of: - A storage image (see **OpTypeImage**). - A block in the **StorageBuffer** storage class, 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. Volatile applies only to a single invocation and does not guarantee each invocation performs the access. **Volatile** is not allowed if the declared memory model is **Vulkan**. The memory operand bit **Volatile**, the image operand bit **VolatileTexel**, or the memory semantic bit **Volatile** can be used instead.		
22	**Constant** Indicates that a global variable is constant and **never** modified. Only allowed on global variables.		**Kernel**

Decoration		Extra Operands	Enabling Capabilities
23	**Coherent** Must be applied only to memory object declarations or members of a structure type. Any such memory object declaration, or any memory object declaration that contains such a structure type, must be one of: - A storage image (see **OpTypeImage**). - A block in the **StorageBuffer** storage class, or in the **Uniform** storage class with the **BufferBlock** decoration. This indicates the memory backing the object is coherent. **Coherent** is not allowed if the declared memory model is **Vulkan**. The memory operand bits **MakePointerAvailable** and **MakePointerVisible** or the image operand bits **MakeTexelAvailable** and **MakeTexelVisible** can be used instead.		
24	**NonWritable** Must be applied only to memory object declarations or members of a structure type. Any such memory object declaration, or any memory object declaration that contains such a structure type, must be one of: - A storage image (see **OpTypeImage**). - A block in the **StorageBuffer** storage class, or in the **Uniform** storage class with the **BufferBlock** decoration. - Missing before **version 1.4**: An object in the **Private** or **Function** storage classes. This indicates that this module does not write to the memory holding the variable. It does not prevent the use of initializers on a declaration.		

	Decoration	Extra Operands	Enabling Capabilities
25	**NonReadable** Must be applied only to memory object declarations or members of a structure type. Any such memory object declaration, or any memory object declaration that contains such a structure type, must be one of: - A storage image (see **OpTypeImage**). - A block in the **StorageBuffer** storage class, or in the **Uniform** storage class with the **BufferBlock** decoration. This indicates that this module does not read from the memory holding the variable. For image variables, it does not prevent query operations from reading metadata associated with the image.		
26	**Uniform** Apply only to an object. Asserts that, for each dynamic instance of the instruction that computes the result, all active invocations in the invocation's **Subgroup** scope compute the same result value.		**Shader**, **UniformDecoration**
27	**UniformId** Apply only to an object. Asserts that, for each dynamic instance of the instruction that computes the result, all active invocations in the *Execution* scope compute the same result value. *Execution* must not be **Invocation**.	*Scope <id>* *Execution*	**Shader**, **UniformDecoration** Missing before **version 1.4**.
28	**SaturatedConversion** Indicates that a conversion to an integer type which is outside the representable range of *Result Type* is clamped to the nearest representable value of *Result Type*. *NaN* is converted to *0*. This decoration must be applied only to conversion instructions to integer types, not including the **OpSatConvertUToS** and **OpSatConvertSToU** instructions.		**Kernel**

Decoration		Extra Operands	Enabling Capabilities
29	**Stream** Must only be used on a memory object declaration or a member of a structure type. *Stream Number* is an unsigned 32-bit integer indicating the stream number to put an output on. Only valid for the **Output** Storage Class and the **Geometry** Execution Model.	*Literal* *Stream Number*	**GeometryStreams**
30	**Location** Apply only to a variable or a structure-type member. *Location* is an unsigned 32-bit integer that forms the main linkage for Storage Class **Input** and **Output** variables: - between the client API and vertex-stage inputs, - between consecutive programmable stages, or - between fragment-stage outputs and the client API. It can also tag variables or structure-type members in the **UniformConstant** Storage Class for linkage with the client API. Only valid for the **Input**, **Output**, and **UniformConstant** Storage Classes.	*Literal* *Location*	**Shader**
31	**Component** Must only be used on a memory object declaration or a member of a structure type. *Component* is an unsigned 32-bit integer indicating which component within a **Location** is taken by the decorated entity. Only valid for the **Input** and **Output** Storage Classes.	*Literal* *Component*	**Shader**
32	**Index** Apply only to a variable. *Index* is an unsigned 32-bit integer identifying a blend equation input index, used as specified by the client API. Only valid for the **Output** Storage Class and the **Fragment** Execution Model.	*Literal* *Index*	**Shader**
33	**Binding** Apply only to a variable.*Binding Point* is an unsigned 32-bit integer forming part of the linkage between the client API and SPIR-V memory buffers, images, etc. See the client API specification for more detail.	*Literal* *Binding Point*	**Shader**

	Decoration	Extra Operands	Enabling Capabilities
34	**DescriptorSet** Apply only to a variable.*Descriptor Set* is an unsigned 32-bit integer forming part of the linkage between the client API and SPIR-V memory buffers, images, etc. See the client API specification for more detail.	*Literal* *Descriptor Set*	**Shader**
35	**Offset** Apply only to a structure-type member. *Byte Offset* is an unsigned 32-bit integer. It dictates the byte offset of the member relative to the beginning of the structure. It 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**.	*Literal* *Byte Offset*	**Shader**
36	**XfbBuffer** Must only be used on a memory object declaration or a member of a structure type. *XFB Buffer* is an unsigned 32-bit integer indicating which transform-feedback buffer an output is written to. Only valid for the **Output** Storage Classes of *vertex processing* Execution Models.	*Literal* *XFB Buffer Number*	**TransformFeedback**
37	**XfbStride** Apply to anything **XfbBuffer** is applied to. *XFB Stride* is an unsigned 32-bit integer specifying the stride, in bytes, of transform-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.	*Literal* *XFB Stride*	**TransformFeedback**
38	**FuncParamAttr** Indicates a function return value or parameter attribute.	*Function Parameter Attribute* *Function Parameter Attribute*	**Kernel**
39	**FPRoundingMode** Indicates a floating-point rounding mode.	*FP Rounding Mode* *Floating-Point Rounding Mode*	

	Decoration	Extra Operands		Enabling Capabilities
40	**FPFastMathMode** Indicates a floating-point fast math flag.	*FP Fast Math Mode* *Fast-Math Mode*		**Kernel**
41	**LinkageAttributes** Associate linkage attributes to values. *Name* is a string specifying what name the *Linkage Type* applies to. Only valid on **OpFunction** or global (module scope) **OpVariable**. See linkage.	*Literal* *Name*	*Linkage Type* *Linkage Type*	**Linkage**
42	**NoContraction** Apply only 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 only to a variable. *Attachment Index* is an unsigned 32-bit integer providing an input-target index (as specified by the client API). Only valid in the **Fragment** Execution Model and for variables of type **OpTypeImage** with a *Dim* operand of **SubpassData**.	*Literal* *Attachment Index*		**InputAttachment**
44	**Alignment** Apply only to a pointer. *Alignment* is an unsigned 32-bit integer declaring a known minimum alignment the pointer has.	*Literal* *Alignment*		**Kernel**
45	**MaxByteOffset** Apply only to a pointer.*Max Byte Offset* is an unsigned 32-bit integer declaring 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**.	*Literal* *Max Byte Offset*		**Addresses** Missing before **version 1.1**.

Decoration		Extra Operands	Enabling Capabilities
46	**AlignmentId** Same as the **Alignment** decoration, but using an *<id>* operand instead of a literal. The operand is consumed as unsigned and must be an *integer type* scalar.	*<id>* *Alignment*	**Kernel** Missing before **version 1.2**.
47	**MaxByteOffsetId** Same as the **MaxByteOffset** decoration, but using an *<id>* operand instead of a literal. The operand is consumed as unsigned and must be an *integer type* scalar.	*<id>* *Max Byte Offset*	**Addresses** Missing before **version 1.2**.
4469	**NoSignedWrap** Apply to an instruction to indicate that it does not cause signed integer wrapping to occur, in the form of overflow or underflow. It must decorate only the following instructions: - **OpIAdd** - **OpISub** - **OpIMul** - **OpShiftLeftLogical** - **OpSNegate** - **OpExtInst** for instruction numbers specified in the extended instruction-set specifications as accepting this decoration. If an instruction decorated with **NoSignedWrap** does overflow or underflow, behavior is undefined.		Missing before **version 1.4**. Also see extension: **SPV_KHR_no_integer_wrap_decoration**

Decoration		Extra Operands	Enabling Capabilities
4470	**NoUnsignedWrap** Apply to an instruction to indicate that it does not cause unsigned integer wrapping to occur, in the form of overflow or underflow. It must decorate only the following instructions: - **OpIAdd** - **OpISub** - **OpIMul** - **OpShiftLeftLogical** - **OpExtInst** for instruction numbers specified in the extended instruction-set specifications as accepting this decoration. If an instruction decorated with **NoUnsignedWrap** does overflow or underflow, behavior is undefined.		Missing before **version 1.4**. Also see extension: **SPV_KHR_no_integer_wrap_decoration**
4999	**ExplicitInterpAMD**		Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
5248	**OverrideCoverageNV**		**SampleMaskOverrideCoverageNV** Reserved. Also see extension: **SPV_NV_sample_mask_override_coverage**
5250	**PassthroughNV**		**GeometryShaderPassthroughNV** Reserved. Also see extension: **SPV_NV_geometry_shader_passthrough**
5252	**ViewportRelativeNV**		**ShaderViewportMaskNV** Reserved.
5256	**SecondaryViewportRelativeNV**	*Literal* *Offset*	**ShaderStereoViewNV** Reserved. Also see extension: **SPV_NV_stereo_view_rendering**

Decoration		Extra Operands	Enabling Capabilities
5271	**PerPrimitiveNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5272	**PerViewNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5273	**PerTaskNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5285	**PerVertexKHR**		**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5285	**PerVertexNV**		**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5300	**NonUniform** Apply only to an object. Asserts that the value backing the decorated <*id*> is not dynamically uniform. See the client API specification for more detail.		**ShaderNonUniform** Missing before **version 1.5**.

Decoration		Extra Operands	Enabling Capabilities
5300	**NonUniformEXT**		**ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5355	**RestrictPointer** Apply only to a memory object declaration, to indicate the compiler may compile as if there is no aliasing of the pointer stored in the variable. See the aliasing section for more detail.		**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extensions: **SPV_EXT_physical_storage_buffer**, **SPV_KHR_physical_storage_buffer**
5355	**RestrictPointerEXT**		**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extension: **SPV_EXT_physical_storage_buffer**
5356	**AliasedPointer** Apply only to a memory object declaration, to indicate the compiler is to generate accesses to the pointer stored in the variable that work correctly in the presence of aliasing. See the aliasing section for more detail.		**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extensions: **SPV_EXT_physical_storage_buffer**, **SPV_KHR_physical_storage_buffer**
5356	**AliasedPointerEXT**		**PhysicalStorageBufferAddresses** Missing before **version 1.5**. Also see extension: **SPV_EXT_physical_storage_buffer**
5398	**BindlessSamplerNV**		**BindlessTextureNV** Reserved.
5399	**BindlessImageNV**		**BindlessTextureNV** Reserved.
5400	**BoundSamplerNV**		**BindlessTextureNV** Reserved.
5401	**BoundImageNV**		**BindlessTextureNV** Reserved.

Decoration		Extra Operands	Enabling Capabilities
5599	**SIMTCallINTEL**	*Literal* *N*	**VectorComputeINTEL** Reserved.
5602	**ReferencedIndirectlyINTEL**		**IndirectReferencesINTEL** Reserved. Also see extension: **SPV_INTEL_function_pointers**
5607	**ClobberINTEL**	*Literal* *Register*	**AsmINTEL** Reserved.
5608	**SideEffectsINTEL**		**AsmINTEL** Reserved.
5624	**VectorComputeVariableINTEL**		**VectorComputeINTEL** Reserved.
5625	**FuncParamIOKindINTEL**	*Literal* *Kind*	**VectorComputeINTEL** Reserved.
5626	**VectorComputeFunctionINTEL**		**VectorComputeINTEL** Reserved.
5627	**StackCallINTEL**		**VectorComputeINTEL** Reserved.
5628	**GlobalVariableOffsetINTEL**	*Literal* *Offset*	**VectorComputeINTEL** Reserved.
5634	**CounterBuffer** The *<id>* of a counter buffer associated with the decorated buffer. It must decorate only a variable in the **Uniform** storage class. *Counter Buffer* must be a variable in the **Uniform** storage class.	*<id>* *Counter Buffer*	Missing before **version 1.4**.
5634	**HlslCounterBufferGOOGLE**	*<id>* *Counter Buffer*	Reserved. Also see extension: **SPV_GOOGLE_hlsl_functionality1**

Decoration		Extra Operands		Enabling Capabilities
5635	**UserSemantic** *Semantic* is a string describing a user-defined semantic intent of what it decorates. User-defined semantics are case insensitive. It must decorate only a variable or a member of a structure type. If decorating a variable, it must be in the **Input** or **Output** storage classes.	*Literal* *Semantic*		Missing before **version 1.4**.
5635	**HlslSemanticGOOGLE**	*Literal* *Semantic*		Reserved. Also see extension: **SPV_GOOGLE_hlsl_functionality1**
5636	**UserTypeGOOGLE**	*Literal* *User Type*		Reserved. Also see extension: **SPV_GOOGLE_user_type**
5822	**FunctionRoundingModeINTEL**	*Literal* *Target Width*	*FP Rounding Mode* *FP Rounding Mode*	**FunctionFloatControlINTEL** Reserved.
5823	**FunctionDenormModeINTEL**	*Literal* *Target Width*	*Reserved FP Denorm Mode* *FP Denorm Mode*	**FunctionFloatControlINTEL** Reserved.
5825	**RegisterINTEL**			**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5826	**MemoryINTEL**	*Literal* *Memory Type*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5827	**NumbanksINTEL**	*Literal* *Banks*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**

Decoration		Extra Operands		Enabling Capabilities
5828	**BankwidthINTEL**	*Literal* *Bank Width*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5829	**MaxPrivateCopiesINTEL**	*Literal* *Maximum Copies*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5830	**SinglepumpINTEL**			**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5831	**DoublepumpINTEL**			**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5832	**MaxReplicatesINTEL**	*Literal* *Maximum Replicates*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5833	**SimpleDualPortINTEL**			**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5834	**MergeINTEL**	*Literal* *Merge Key*	*Literal* *Merge Type*	**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5835	**BankBitsINTEL**	*Literal* *Bank Bits*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**

	Decoration	Extra Operands		Enabling Capabilities
5836	**ForcePow2DepthINTEL**	*Literal* *Force Key*		**FPGAMemoryAttributesINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5899	**BurstCoalesceINTEL**			**FPGAMemoryAccessesINTEL** Reserved.
5900	**CacheSizeINTEL**	*Literal* *Cache Size in bytes*		**FPGAMemoryAccessesINTEL** Reserved.
5901	**DontStaticallyCoalesceINTEL**			**FPGAMemoryAccessesINTEL** Reserved.
5902	**PrefetchINTEL**	*Literal* *Prefetcher Size in bytes*		**FPGAMemoryAccessesINTEL** Reserved.
5905	**StallEnableINTEL**			**FPGAClusterAttributesINTEL** Reserved.
5907	**FuseLoopsInFunctionINTEL**			**LoopFuseINTEL** Reserved.
5921	**BufferLocationINTEL**	*Literal* *Buffer Location ID*		**FPGABufferLocationINTEL** Reserved.
5944	**IOPipeStorageINTEL**	*Literal* *IO Pipe ID*		**IOPipesINTEL** Reserved.
6080	**FunctionFloatingPointModeINTEL**	*Literal* *Target Width*	*Reserved FP Operation Mode* *FP Operation Mode*	**FunctionFloatControlINTEL** Reserved.
6085	**SingleElementVectorINTEL**			**VectorComputeINTEL** Reserved.
6087	**VectorComputeCallableFunctionINTEL**			**VectorComputeINTEL** Reserved.

Decoration		Extra Operands	Enabling Capabilities
6140	**MediaBlockIOINTEL**		**VectorComputeINTEL** Reserved.

3.21. BuiltIn

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

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

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

For all the declarations of all the global variables and constants statically referenced by the entry-point's call tree, within any specific storage class it is invalid to decorate with a specific **BuiltIn** more than once.

BuiltIn		Enabling Capabilities
0	**Position** Output vertex position from a vertex processing Execution Model. See the client API specification for more detail.	**Shader**
1	**PointSize** Output point size from a vertex processing Execution Model. See the client API specification for more detail.	**Shader**
3	**ClipDistance** Array of clip distances. See the client API specification for more detail.	**ClipDistance**
4	**CullDistance** Array of clip distances. See the client API specification for more detail.	**CullDistance**
5	**VertexId** Input vertex ID to a **Vertex** Execution Model. See the client API specification for more detail.	**Shader**
6	**InstanceId** Input instance ID to a **Vertex** Execution Model. See the client API specification for more detail.	**Shader**
7	**PrimitiveId** Primitive ID in a **Geometry** Execution Model. See the client API specification for more detail.	**Geometry**, **Tessellation**, **RayTracingNV**, **RayTracingKHR**, **MeshShadingNV**
8	**InvocationId** Invocation ID, input to **Geometry** and **TessellationControl** Execution Model. See the client API specification for more detail.	**Geometry**, **Tessellation**

	BuiltIn	Enabling Capabilities
9	**Layer** Layer selection for multi-layer framebuffer. See the client API specification for more detail. The **Geometry** capability allows for a **Layer** output by a **Geometry** Execution Model, input to a **Fragment** Execution Model. The **ShaderLayer** capability allows for **Layer** output by a **Vertex** or **Tessellation** Execution Model.	**Geometry**, **ShaderLayer**, **ShaderViewportIndexLayerEXT**, **MeshShadingNV**
10	**ViewportIndex** Viewport selection for viewport transformation when using multiple viewports. See the client API specification for more detail. The **MultiViewport** capability allows for a **ViewportIndex** output by a **Geometry** Execution Model, input to a **Fragment** Execution Model. The **ShaderViewportIndex** capability allows for a **ViewportIndex** output by a **Vertex** or **Tessellation** Execution Model.	**MultiViewport**, **ShaderViewportIndex**, **ShaderViewportIndexLayerEXT**, **MeshShadingNV**
11	**TessLevelOuter** Output patch outer levels in a **TessellationControl** Execution Model. See the client API specification for more detail.	**Tessellation**
12	**TessLevelInner** Output patch inner levels in a **TessellationControl** Execution Model. See the client API specification for more detail.	**Tessellation**
13	**TessCoord** Input vertex position in **TessellationEvaluation** Execution Model. See the client API specification for more detail.	**Tessellation**
14	**PatchVertices** Input patch vertex count in a tessellation Execution Model. See the client API specification for more detail.	**Tessellation**
15	**FragCoord** Coordinates *(x, y, z, 1/w)* of the current fragment, input to the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**
16	**PointCoord** Coordinates within a *point*, input to the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**

BuiltIn		Enabling Capabilities
17	**FrontFacing** Face direction, input to the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**
18	**SampleId** Input sample number to the **Fragment** Execution Model. See the client API specification for more detail.	**SampleRateShading**
19	**SamplePosition** Input sample position to the **Fragment** Execution Model. See the client API specification for more detail.	**SampleRateShading**
20	**SampleMask** Input or output sample mask to the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**
22	**FragDepth** Output fragment depth from the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**
23	**HelperInvocation** Input whether a helper invocation, to the **Fragment** Execution Model. See the client API specification for more detail.	**Shader**
24	**NumWorkgroups** Number of workgroups in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	
25	**WorkgroupSize** Deprecated (use **LocalSizeId** Execution Mode instead). Work-group size in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	
26	**WorkgroupId** Work-group ID in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	
27	**LocalInvocationId** Local invocation ID in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	
28	**GlobalInvocationId** Global invocation ID in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	

	BuiltIn	Enabling Capabilities
29	**LocalInvocationIndex** Local invocation index in **GLCompute** Execution Models. See the client API specification for more detail. Work-group Linear ID in **Kernel** Execution Models. See the client API specification for more detail.	
30	**WorkDim** Work dimensions in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
31	**GlobalSize** Global size in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
32	**EnqueuedWorkgroupSize** Enqueued work-group size in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
33	**GlobalOffset** Global offset in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
34	**GlobalLinearId** Global linear ID in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
36	**SubgroupSize** Subgroup size. See the client API specification for more detail.	**Kernel**, **GroupNonUniform**, **SubgroupBallotKHR**
37	**SubgroupMaxSize** Subgroup maximum size in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
38	**NumSubgroups** Number of subgroups in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**, **GroupNonUniform**
39	**NumEnqueuedSubgroups** Number of enqueued subgroups in **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**
40	**SubgroupId** Subgroup ID in **GLCompute** or **Kernel** Execution Models. See the client API specification for more detail.	**Kernel**, **GroupNonUniform**
41	**SubgroupLocalInvocationId** Subgroup local invocation ID. See the client API specification for more detail.	**Kernel**, **GroupNonUniform**, **SubgroupBallotKHR**

BuiltIn		Enabling Capabilities
42	**VertexIndex** Vertex index. See the client API specification for more detail.	**Shader**
43	**InstanceIndex** Instance index. See the client API specification for more detail.	**Shader**
4416	**SubgroupEqMask** Subgroup invocations bitmask where bit index == **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4416	**SubgroupEqMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
4417	**SubgroupGeMask** Subgroup invocations bitmask where bit index >= **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4417	**SubgroupGeMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
4418	**SubgroupGtMask** Subgroup invocations bitmask where bit index > **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4418	**SubgroupGtMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
4419	**SubgroupLeMask** Subgroup invocations bitmask where bit index <= **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4419	**SubgroupLeMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**

	BuiltIn	Enabling Capabilities
4420	**SubgroupLtMask** Subgroup invocations bitmask where bit index < **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4420	**SubgroupLtMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
4424	**BaseVertex** Base vertex component of vertex ID. See the client API specification for more detail.	**DrawParameters** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_draw_parameters**
4425	**BaseInstance** Base instance component of instance ID. See the client API specification for more detail.	**DrawParameters** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_draw_parameters**
4426	**DrawIndex** Contains the index of the draw currently being processed. See the client API specification for more detail.	**DrawParameters**, **MeshShadingNV** Missing before **version 1.3**. Also see extensions: **SPV_KHR_shader_draw_parameters**, **SPV_NV_mesh_shader**
4432	**PrimitiveShadingRateKHR**	**FragmentShadingRateKHR** Reserved. Also see extension: **SPV_KHR_fragment_shading_rate**
4438	**DeviceIndex** Input device index of the logical device. See the client API specification for more detail.	**DeviceGroup** Missing before **version 1.3**. Also see extension: **SPV_KHR_device_group**
4440	**ViewIndex** Input view index of the view currently being rendered to. See the client API specification for more detail.	**MultiView** Missing before **version 1.3**. Also see extension: **SPV_KHR_multiview**

BuiltIn		Enabling Capabilities
4444	**ShadingRateKHR**	**FragmentShadingRateKHR** Reserved. Also see extension: **SPV_KHR_fragment_shading_rate**
4992	**BaryCoordNoPerspAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4993	**BaryCoordNoPerspCentroidAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4994	**BaryCoordNoPerspSampleAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4995	**BaryCoordSmoothAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4996	**BaryCoordSmoothCentroidAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4997	**BaryCoordSmoothSampleAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
4998	**BaryCoordPullModelAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**
5014	**FragStencilRefEXT**	**StencilExportEXT** Reserved. Also see extension: **SPV_EXT_shader_stencil_export**

BuiltIn		Enabling Capabilities
5253	**ViewportMaskNV**	**ShaderViewportMaskNV**, **MeshShadingNV** Reserved. Also see extensions: **SPV_NV_viewport_array2**, **SPV_NV_mesh_shader**
5257	**SecondaryPositionNV**	**ShaderStereoViewNV** Reserved. Also see extension: **SPV_NV_stereo_view_rendering**
5258	**SecondaryViewportMaskNV**	**ShaderStereoViewNV** Reserved. Also see extension: **SPV_NV_stereo_view_rendering**
5261	**PositionPerViewNV**	**PerViewAttributesNV**, **MeshShadingNV** Reserved. Also see extensions: **SPV_NVX_multiview_per_view_attributes**, **SPV_NV_mesh_shader**
5262	**ViewportMaskPerViewNV**	**PerViewAttributesNV**, **MeshShadingNV** Reserved. Also see extensions: **SPV_NVX_multiview_per_view_attributes**, **SPV_NV_mesh_shader**
5264	**FullyCoveredEXT**	**FragmentFullyCoveredEXT** Reserved. Also see extension: **SPV_EXT_fragment_fully_covered**
5274	**TaskCountNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**

BuiltIn		Enabling Capabilities
5275	**PrimitiveCountNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5276	**PrimitiveIndicesNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5277	**ClipDistancePerViewNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5278	**CullDistancePerViewNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5279	**LayerPerViewNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5280	**MeshViewCountNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5281	**MeshViewIndicesNV**	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5286	**BaryCoordKHR**	**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**

BuiltIn		Enabling Capabilities
5286	**BaryCoordNV**	**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5287	**BaryCoordNoPerspKHR**	**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5287	**BaryCoordNoPerspNV**	**FragmentBarycentricNV**, **FragmentBarycentricKHR** Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5292	**FragSizeEXT**	**FragmentDensityEXT**, **ShadingRateNV** Reserved. Also see extensions: **SPV_EXT_fragment_invocation_density**, **SPV_NV_shading_rate**
5292	**FragmentSizeNV**	**ShadingRateNV**, **FragmentDensityEXT** Reserved. Also see extensions: **SPV_NV_shading_rate**, **SPV_EXT_fragment_invocation_density**
5293	**FragInvocationCountEXT**	**FragmentDensityEXT**, **ShadingRateNV** Reserved. Also see extensions: **SPV_EXT_fragment_invocation_density**, **SPV_NV_shading_rate**

BuiltIn		Enabling Capabilities
5293	**InvocationsPerPixelNV**	**ShadingRateNV**, **FragmentDensityEXT** Reserved. Also see extensions: **SPV_NV_shading_rate**, **SPV_EXT_fragment_invocation_density**
5319	**LaunchIdNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5319	**LaunchIdKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5320	**LaunchSizeNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5320	**LaunchSizeKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5321	**WorldRayOriginNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5321	**WorldRayOriginKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5322	**WorldRayDirectionNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**

BuiltIn		Enabling Capabilities
5322	**WorldRayDirectionKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5323	**ObjectRayOriginNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5323	**ObjectRayOriginKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5324	**ObjectRayDirectionNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5324	**ObjectRayDirectionKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5325	**RayTminNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5325	**RayTminKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5326	**RayTmaxNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**

BuiltIn		Enabling Capabilities
5326	**RayTmaxKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5327	**InstanceCustomIndexNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5327	**InstanceCustomIndexKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5330	**ObjectToWorldNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5330	**ObjectToWorldKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5331	**WorldToObjectNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5331	**WorldToObjectKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5332	**HitTNV**	**RayTracingNV** Reserved. Also see extension: **SPV_NV_ray_tracing**

BuiltIn		Enabling Capabilities
5333	**HitKindNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5333	**HitKindKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5334	**CurrentRayTimeNV**	**RayTracingMotionBlurNV** Reserved. Also see extension: **SPV_NV_ray_tracing_motion_blur**
5351	**IncomingRayFlagsNV**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5351	**IncomingRayFlagsKHR**	**RayTracingNV**, **RayTracingKHR** Reserved. Also see extensions: **SPV_NV_ray_tracing**, **SPV_KHR_ray_tracing**
5352	**RayGeometryIndexKHR**	**RayTracingKHR** Reserved. Also see extension: **SPV_KHR_ray_tracing**
5374	**WarpsPerSMNV**	**ShaderSMBuiltinsNV** Reserved. Also see extension: **SPV_NV_shader_sm_builtins**
5375	**SMCountNV**	**ShaderSMBuiltinsNV** Reserved. Also see extension: **SPV_NV_shader_sm_builtins**

BuiltIn		Enabling Capabilities
5376	**WarpIDNV**	**ShaderSMBuiltinsNV** Reserved. Also see extension: **SPV_NV_shader_sm_builtins**
5377	**SMIDNV**	**ShaderSMBuiltinsNV** Reserved. Also see extension: **SPV_NV_shader_sm_builtins**

3.22. Selection Control

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

Bits that are set indicate whether an additional operand follows, as described by the table. If there are multiple following operands indicated, they are ordered: Those indicated by smaller-numbered bits appear first.

Used by **OpLoopMerge**.

Loop Control		Enabling Capabilities
0x0	**None**	
0x1	**Unroll** Strong request, to the extent possible, to unroll or unwind this loop. This must not be used with the **DontUnroll** bit.	

Loop Control		Enabling Capabilities
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.	Missing before **version 1.1**.
0x8	**DependencyLength** Guarantees that there are no dependencies between a number of loop iterations. The dependency length is specified in a subsequent unsigned 32-bit integer literal operand.	Missing before **version 1.1**.
0x10	**MinIterations** Unchecked assertion that the loop executes at least a given number of iterations. The iteration count is specified in a subsequent unsigned 32-bit integer literal operand.	Missing before **version 1.4**.
0x20	**MaxIterations** Unchecked assertion that the loop executes at most a given number of iterations. The iteration count is specified in a subsequent unsigned 32-bit integer literal operand.	Missing before **version 1.4**.
0x40	**IterationMultiple** Unchecked assertion that the loop executes a multiple of a given number of iterations. The number is specified in a subsequent unsigned 32-bit integer literal operand. It must be greater than 0.	Missing before **version 1.4**.
0x80	**PeelCount** Request that the loop be peeled by a given number of loop iterations. The peel count is specified in a subsequent unsigned 32-bit integer literal operand. This must not be used with the **DontUnroll** bit.	Missing before **version 1.4**.
0x100	**PartialCount** Request that the loop be partially unrolled by a given number of loop iterations. The unroll count is specified in a subsequent unsigned 32-bit integer literal operand. This must not be used with the **DontUnroll** bit.	Missing before **version 1.4**.
0x10000	**InitiationIntervalINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**

Loop Control		Enabling Capabilities
0x20000	**MaxConcurrencyINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x40000	**DependencyArrayINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x80000	**PipelineEnableINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x100000	**LoopCoalesceINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x200000	**MaxInterleavingINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x400000	**SpeculatedIterationsINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**
0x800000	**NoFusionINTEL**	**FPGALoopControlsINTEL** Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**

3.24. Function Control

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

Used by **OpFunction**.

Function Control		Enabling Capabilities
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.	
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 when called with the same argument values and the same global state.	
0x8	**Const** Compiler assumes this function has no side effects, and does not access global memory or dereference function parameters. Always computes the same result for the same argument values.	
0x10000	**OptNoneINTEL**	**OptNoneINTEL** Reserved.

3.25. Memory Semantics <id>

The <*id*>'s value is a mask; it can be formed by combining the bits from multiple rows in the table below.

The value’s type must be a 32-bit integer scalar. This value is expected to be formed only from the bits in the table below, where at most one of these four bits can be set: **Acquire**, **Release**, **AcquireRelease**, or **SequentiallyConsistent**. If validation rules or the client API require a constant <*id*>, it is invalid for the value to not be formed this expected way. If non-constant <*id*> are allowed, behavior is undefined when the value is not formed this expected way.

Requesting both **Acquire** and **Release** semantics is done by setting the **AcquireRelease** bit, not by setting two bits.

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 are 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.

Used by:

- **OpControlBarrier**
- **OpMemoryBarrier**
- **OpAtomicLoad**
- **OpAtomicStore**

- OpAtomicExchange
- OpAtomicCompareExchange
- OpAtomicCompareExchangeWeak
- OpAtomicIIncrement
- OpAtomicIDecrement
- OpAtomicIAdd
- OpAtomicISub
- OpAtomicSMin
- OpAtomicUMin
- OpAtomicSMax
- OpAtomicUMax
- OpAtomicAnd
- OpAtomicOr
- OpAtomicXor
- OpAtomicFlagTestAndSet
- OpAtomicFlagClear
- OpMemoryNamedBarrier
- OpAtomicFMinEXT
- OpAtomicFMaxEXT
- OpAtomicFAddEXT

Memory Semantics		Enabling Capabilities
0x0	**None (Relaxed)**	
0x2	**Acquire** On an atomic instruction, orders memory operations provided in program order after this atomic instruction against this atomic instruction. On a barrier, orders memory operations provided in program order after this barrier against atomic instructions before this barrier. See the client API specification for more detail.	
0x4	**Release** On an atomic instruction, orders memory operations provided in program order before this atomic instruction against this atomic instruction. On a barrier, orders memory operations provided in program order before this barrier against atomic instructions after this barrier. See the client API specification for more detail.	

Memory Semantics		Enabling Capabilities
0x8	**AcquireRelease** Has the properties of both **Acquire** and **Release** semantics. It is used for read-modify-write operations.	
0x10	**SequentiallyConsistent** All observers see this memory access in the same order with respect to other sequentially-consistent memory accesses from this invocation. If the declared memory model is **Vulkan**, **SequentiallyConsistent** must not be used.	
0x40	**UniformMemory** Apply the memory-ordering constraints to **StorageBuffer**, **PhysicalStorageBuffer**, or **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.	
0x1000	**OutputMemory** Apply the memory-ordering constraints to **Output** storage class memory.	**VulkanMemoryModel** Missing before **version 1.5**.
0x1000	**OutputMemoryKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x2000	**MakeAvailable** Perform an availability operation on all references in the selected storage classes.	**VulkanMemoryModel** Missing before **version 1.5**.

Memory Semantics		Enabling Capabilities
0x2000	**MakeAvailableKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x4000	**MakeVisible** Perform a visibility operation on all references in the selected storage classes.	**VulkanMemoryModel** Missing before **version 1.5**.
0x4000	**MakeVisibleKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x8000	**Volatile** This access cannot be eliminated, duplicated, or combined with other accesses.	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**

3.26. Memory Operands

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

Provides additional operands to the listed memory instructions. Bits that are set indicate whether an additional operand follows, as described by the table. If there are multiple following operands indicated, they are ordered: Those indicated by smaller-numbered bits appear first. An instruction needing two masks must first provide the first mask followed by the first mask's additional operands, and then provide the second mask followed by the second mask's additional operands.

Used by:

- **OpLoad**
- **OpStore**
- **OpCopyMemory**
- **OpCopyMemorySized**
- **OpCooperativeMatrixLoadNV**
- **OpCooperativeMatrixStoreNV**

Memory Operands		Enabling Capabilities
0x0	**None**	
0x1	**Volatile** This access cannot be eliminated, duplicated, or combined with other accesses.	

Memory Operands		Enabling Capabilities
0x2	**Aligned** This access has a known alignment. The alignment is specified in a subsequent unsigned 32-bit integer literal operand. Valid values are defined by the execution environment.	
0x4	**Nontemporal** Hints that the accessed address is not likely to be accessed again in the near future.	
0x8	**MakePointerAvailable** Perform an availability operation on the locations pointed to by the pointer operand, after a store. A following operand is the memory scope for the availability operation. Requires **NonPrivatePointer** to also be set. Not valid with **OpLoad**.	**VulkanMemoryModel** Missing before **version 1.5**.
0x8	**MakePointerAvailableKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x10	**MakePointerVisible** Perform a visibility operation on the locations pointed to by the pointer operand, before a load. A following operand is the memory scope for the visibility operation. Requires **NonPrivatePointer** to also be set. Not valid with **OpStore**.	**VulkanMemoryModel** Missing before **version 1.5**.
0x10	**MakePointerVisibleKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
0x20	**NonPrivatePointer** The memory access obeys inter-thread ordering, as specified by the client API.	**VulkanMemoryModel** Missing before **version 1.5**.
0x20	**NonPrivatePointerKHR**	**VulkanMemoryModel** Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**

3.27. Scope <id>

Must be an <*id*> of a 32-bit integer scalar. Its value is expected to be one of the values in the table below. If validation rules or the client API require a constant <*id*>, it is invalid for it to not be one of these values. If non-constant <*id*> are allowed, behavior is undefined if <*id*> is not one of these values.

If labeled as a memory scope, it specifies the distance of synchronization from the current invocation. If labeled as an execution scope, it specifies the set of executing invocations taking part in the operation. Other usages (neither memory nor execution) of scope are possible, and each such usage defines what scope means in its context.

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
- OpGroupNonUniformElect
- OpGroupNonUniformAll
- OpGroupNonUniformAny
- OpGroupNonUniformAllEqual
- OpGroupNonUniformBroadcast
- OpGroupNonUniformBroadcastFirst
- OpGroupNonUniformBallot
- OpGroupNonUniformInverseBallot
- OpGroupNonUniformBallotBitExtract
- OpGroupNonUniformBallotBitCount
- OpGroupNonUniformBallotFindLSB
- OpGroupNonUniformBallotFindMSB
- OpGroupNonUniformShuffle
- OpGroupNonUniformShuffleXor
- OpGroupNonUniformShuffleUp
- OpGroupNonUniformShuffleDown
- OpGroupNonUniformIAdd
- OpGroupNonUniformFAdd
- OpGroupNonUniformIMul
- OpGroupNonUniformFMul
- OpGroupNonUniformSMin
- OpGroupNonUniformUMin
- OpGroupNonUniformFMin
- OpGroupNonUniformSMax
- OpGroupNonUniformUMax
- OpGroupNonUniformFMax
- OpGroupNonUniformBitwiseAnd
- OpGroupNonUniformBitwiseOr
- OpGroupNonUniformBitwiseXor
- OpGroupNonUniformLogicalAnd
- OpGroupNonUniformLogicalOr
- OpGroupNonUniformLogicalXor

- OpGroupNonUniformQuadBroadcast
- OpGroupNonUniformQuadSwap
- OpGroupIAddNonUniformAMD
- OpGroupFAddNonUniformAMD
- OpGroupFMinNonUniformAMD
- OpGroupUMinNonUniformAMD
- OpGroupSMinNonUniformAMD
- OpGroupFMaxNonUniformAMD
- OpGroupUMaxNonUniformAMD
- OpGroupSMaxNonUniformAMD
- OpReadClockKHR
- OpTypeCooperativeMatrixNV
- OpAtomicFMinEXT
- OpAtomicFMaxEXT
- OpAtomicFAddEXT

Scope		Enabling Capabilities
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.	
5	**QueueFamily** Scope is the current queue family.	**VulkanMemoryModel** Missing before **version 1.5**.
5	**QueueFamilyKHR**	**VulkanMemoryModel** Missing before **version 1.5**.
6	**ShaderCallKHR**	**RayTracingKHR** Reserved.

3.28. Group Operation

Defines the class of workgroup or subgroup operation.

Used by:

- OpGroupIAdd
- OpGroupFAdd
- OpGroupFMin
- OpGroupUMin
- OpGroupSMin
- OpGroupFMax
- OpGroupUMax
- OpGroupSMax
- OpGroupNonUniformBallotBitCount
- OpGroupNonUniformIAdd
- OpGroupNonUniformFAdd
- OpGroupNonUniformIMul
- OpGroupNonUniformFMul
- OpGroupNonUniformSMin
- OpGroupNonUniformUMin
- OpGroupNonUniformFMin
- OpGroupNonUniformSMax
- OpGroupNonUniformUMax
- OpGroupNonUniformFMax
- OpGroupNonUniformBitwiseAnd
- OpGroupNonUniformBitwiseOr
- OpGroupNonUniformBitwiseXor
- OpGroupNonUniformLogicalAnd
- OpGroupNonUniformLogicalOr
- OpGroupNonUniformLogicalXor
- OpGroupIAddNonUniformAMD
- OpGroupFAddNonUniformAMD
- OpGroupFMinNonUniformAMD
- OpGroupUMinNonUniformAMD
- OpGroupSMinNonUniformAMD
- OpGroupFMaxNonUniformAMD
- OpGroupUMaxNonUniformAMD
- OpGroupSMaxNonUniformAMD

Group Operation		Enabling Capabilities
0	**Reduce** A reduction operation for all values of a specific value X specified by invocations within a workgroup.	**Kernel**, **GroupNonUniformArithmetic**, **GroupNonUniformBallot**

Group Operation		Enabling Capabilities
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 op a_1), …(a_0 op a_1 op … op a_{n-1})]	**Kernel**, **GroupNonUniformArithmetic**, **GroupNonUniformBallot**
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**, **GroupNonUniformArithmetic**, **GroupNonUniformBallot**
3	**ClusteredReduce**	**GroupNonUniformClustered** Missing before **version 1.3**.
6	**PartitionedReduceNV**	**GroupNonUniformPartitionedNV** Reserved. Also see extension: **SPV_NV_shader_subgroup_partitioned**
7	**PartitionedInclusiveScanNV**	**GroupNonUniformPartitionedNV** Reserved. Also see extension: **SPV_NV_shader_subgroup_partitioned**
8	**PartitionedExclusiveScanNV**	**GroupNonUniformPartitionedNV** Reserved. Also see extension: **SPV_NV_shader_subgroup_partitioned**

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		Enabling Capabilities
0	**NoWait** Indicates that the enqueued kernels do not need to wait for the parent kernel to finish execution before they begin execution.	**Kernel**

Kernel Enqueue Flags		Enabling Capabilities
1	**WaitKernel** Indicates that all work-items of the parent kernel finish executing and all immediate side effects committed before the enqueued child kernel begins 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

The <*id*>'s value is a mask; it can be formed by combining the bits from multiple rows in the table below.

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

Kernel Profiling Info		Enabling Capabilities
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 need to be declared, either explicitly with **OpCapability** or implicitly through the **Implicitly Declares** column: If a capability defined with statically expressed rules is used, it is invalid to not declare it. If a capability defined in terms of dynamic behavior is used, behavior is undefined unless the capability is declared. The **Implicitly Declares** column lists additional capabilities that are all implicitly declared when the **Capability** entry is explicitly or implicitly declared. It is not necessary, but allowed, to explicitly declare an implicitly declared capability.

See the capabilities section for more detail.

Used by **OpCapability**.

	Capability	Implicitly Declares
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** Allows a 16-bit **OpTypeFloat** instruction for creating an **OpTypePointer** to a 16-bit float. Pointers to a 16-bit float must not be dereferenced, unless specifically allowed by a specific instruction. All other uses of 16-bit **OpTypeFloat** are disallowed.	**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**
14	**ImageReadWrite** Uses **OpTypeImage** with the **ReadWrite** *access qualifier* in a kernel.	**ImageBasic**

	Capability	Implicitly Declares
15	**ImageMipmap** Uses non-zero **Lod** Image Operands in a kernel.	**ImageBasic**
17	**Pipes** Uses **OpTypePipe**, **OpTypeReserveId** or *pipe* instructions.	**Kernel**
18	**Groups** Uses common group instructions.	Also see extension: **SPV_AMD_shader_ballot**
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, allowing use of only the **OpAtomicLoad**, **OpAtomicIIncrement**, and **OpAtomicIDecrement** instructions.	**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** An *MS* operand in OpTypeImage indicates multisampled, used with an **OpTypeImage** having *Sampled* == 2.	**Shader**
28	**UniformBufferArrayDynamicIndexing** **Block**-decorated arrays in uniform storage classes use dynamically uniform indexing.	**Shader**
29	**SampledImageArrayDynamicIndexing** Arrays of sampled images, samplers, or images with *Sampled* = 0 or 1 use dynamically uniform indexing.	**Shader**
30	**StorageBufferArrayDynamicIndexing** Arrays in the **StorageBuffer** Storage Class, or **BufferBlock**-decorated arrays, use dynamically uniform indexing.	**Shader**

Capability		Implicitly Declares
31	**StorageImageArrayDynamicIndexing** Arrays of images with *Sampled* = 2 are accessed with dynamically uniform indexing.	**Shader**
32	**ClipDistance** Uses the **ClipDistance** BuiltIn.	**Shader**
33	**CullDistance** Uses the **CullDistance** BuiltIn.	**Shader**
34	**ImageCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in **OpTypeImage**, with an **OpTypeImage** having *Sampled* == 2.	**SampledCubeArray**
35	**SampleRateShading** Uses per-sample rate shading.	**Shader**
36	**ImageRect** Uses the **Rect** Dim with an **OpTypeImage** having *Sampled* == 2.	**SampledRect**
37	**SampledRect** Uses the **Rect** Dim with an **OpTypeImage** having *Sampled* == 0 or 1.	**Shader**
38	**GenericPointer** Uses the **Generic** Storage Class.	**Addresses**
39	**Int8** Uses **OpTypeInt** to declare 8-bit integer types.	
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 an **OpTypeImage** having *Sampled* == 0 or 1.	
44	**Image1D** Uses the **1D** Dim with an **OpTypeImage** having *Sampled* == 2.	**Sampled1D**
45	**SampledCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in OpTypeImage, with an **OpTypeImage** having *Sampled* == 0 or 1.	**Shader**
46	**SampledBuffer** Uses the **Buffer** Dim with an **OpTypeImage** having *Sampled* == 0 or 1.	

	Capability	Implicitly Declares
47	**ImageBuffer** Uses the **Buffer** Dim with an **OpTypeImage** having *Sampled* == 2.	**SampledBuffer**
48	**ImageMSArray** An *MS* operand in OpTypeImage indicates multisampled, used with an **OpTypeImage** having *Sampled* == 2 and *Arrayed* == 1.	**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**
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.	**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** Missing before **version 1.1**.
59	**NamedBarrier** Uses **OpTypeNamedBarrier**.	**Kernel** Missing before **version 1.1**.
60	**PipeStorage** Uses **OpTypePipeStorage**.	**Pipes** Missing before **version 1.1**.

	Capability	Implicitly Declares
61	**GroupNonUniform**	Missing before **version 1.3**.
62	**GroupNonUniformVote**	**GroupNonUniform** Missing before **version 1.3**.
63	**GroupNonUniformArithmetic**	**GroupNonUniform** Missing before **version 1.3**.
64	**GroupNonUniformBallot**	**GroupNonUniform** Missing before **version 1.3**.
65	**GroupNonUniformShuffle**	**GroupNonUniform** Missing before **version 1.3**.
66	**GroupNonUniformShuffleRelative**	**GroupNonUniform** Missing before **version 1.3**.
67	**GroupNonUniformClustered**	**GroupNonUniform** Missing before **version 1.3**.
68	**GroupNonUniformQuad**	**GroupNonUniform** Missing before **version 1.3**.
69	**ShaderLayer**	Missing before **version 1.5**.
70	**ShaderViewportIndex**	Missing before **version 1.5**.
71	**UniformDecoration** Uses the **Uniform** or **UniformId** decoration	Missing before **version 1.6**.
4422	**FragmentShadingRateKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_fragment_shading_rate**
4423	**SubgroupBallotKHR**	Reserved. Also see extension: **SPV_KHR_shader_ballot**
4427	**DrawParameters**	**Shader** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_draw_parameters**

	Capability	Implicitly Declares
4428	**WorkgroupMemoryExplicitLayoutKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_workgroup_memory_explicit_layout**
4429	**WorkgroupMemoryExplicitLayout8BitAccessKHR**	**WorkgroupMemoryExplicitLayoutKHR** Reserved. Also see extension: **SPV_KHR_workgroup_memory_explicit_layout**
4430	**WorkgroupMemoryExplicitLayout16BitAccessKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_workgroup_memory_explicit_layout**
4431	**SubgroupVoteKHR**	Reserved. Also see extension: **SPV_KHR_subgroup_vote**
4433	**StorageBuffer16BitAccess** Uses 16-bit **OpTypeFloat** and **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **StorageBuffer** storage class, the **PhysicalStorageBuffer** storage class, or the **Uniform** storage class with the **BufferBlock** decoration.	Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4433	**StorageUniformBufferBlock16**	Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4434	**UniformAndStorageBuffer16BitAccess** Uses 16-bit **OpTypeFloat** and **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **StorageBuffer** storage class, the **PhysicalStorageBuffer** storage class, or the **Uniform** storage class.	**StorageBuffer16BitAccess**, **StorageUniformBufferBlock16** Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**

Capability		Implicitly Declares
4434	**StorageUniform16**	**StorageBuffer16BitAccess**, **StorageUniformBufferBlock16** Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4435	**StoragePushConstant16** Uses 16-bit **OpTypeFloat** and **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **PushConstant** storage class.	Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4436	**StorageInputOutput16** Uses 16-bit **OpTypeFloat** and **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **Output** storage class.	Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4437	**DeviceGroup**	Missing before **version 1.3**. Also see extension: **SPV_KHR_device_group**
4439	**MultiView**	**Shader** Missing before **version 1.3**. Also see extension: **SPV_KHR_multiview**
4441	**VariablePointersStorageBuffer** Allow *variable pointers*, each confined to a single **Block**-decorated struct in the **StorageBuffer** storage class.	**Shader** Missing before **version 1.3**. Also see extension: **SPV_KHR_variable_pointers**
4442	**VariablePointers** Allow *variable pointers*.	**VariablePointersStorageBuffer** Missing before **version 1.3**. Also see extension: **SPV_KHR_variable_pointers**
4445	**AtomicStorageOps**	Reserved. Also see extension: **SPV_KHR_shader_atomic_counter_ops**
4447	**SampleMaskPostDepthCoverage**	Reserved. Also see extension: **SPV_KHR_post_depth_coverage**

Capability		Implicitly Declares
4448	**StorageBuffer8BitAccess** Uses 8-bit **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **StorageBuffer** storage class or the **PhysicalStorageBuffer** storage class.	Missing before **version 1.5**. Also see extension: **SPV_KHR_8bit_storage**
4449	**UniformAndStorageBuffer8BitAccess** Uses 8-bit **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **StorageBuffer** storage class, the **PhysicalStorageBuffer** storage class, or the **Uniform** storage class.	**StorageBuffer8BitAccess** Missing before **version 1.5**. Also see extension: **SPV_KHR_8bit_storage**
4450	**StoragePushConstant8** Uses 8-bit **OpTypeInt** instructions for creating scalar, vector, and composite types that become members of a block residing in the **PushConstant** storage class.	Missing before **version 1.5**. Also see extension: **SPV_KHR_8bit_storage**
4464	**DenormPreserve** Uses the **DenormPreserve** execution mode.	Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4465	**DenormFlushToZero** Uses the **DenormFlushToZero** execution mode.	Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4466	**SignedZeroInfNanPreserve** Uses the **SignedZeroInfNanPreserve** execution mode.	Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4467	**RoundingModeRTE** Uses the **RoundingModeRTE** execution mode.	Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4468	**RoundingModeRTZ** Uses the **RoundingModeRTZ** execution mode.	Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4471	**RayQueryProvisionalKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_ray_query**
4472	**RayQueryKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_ray_query**

Capability		Implicitly Declares
4478	**RayTraversalPrimitiveCullingKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved. Also see extensions: **SPV_KHR_ray_query**, **SPV_KHR_ray_tracing**
4479	**RayTracingKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_ray_tracing**
5008	**Float16ImageAMD**	**Shader** Reserved. Also see extension: **SPV_AMD_gpu_shader_half_float_fetch**
5009	**ImageGatherBiasLodAMD**	**Shader** Reserved. Also see extension: **SPV_AMD_texture_gather_bias_lod**
5010	**FragmentMaskAMD**	**Shader** Reserved. Also see extension: **SPV_AMD_shader_fragment_mask**
5013	**StencilExportEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_shader_stencil_export**
5015	**ImageReadWriteLodAMD**	**Shader** Reserved. Also see extension: **SPV_AMD_shader_image_load_store_lod**
5016	**Int64ImageEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_shader_image_int64**

	Capability	Implicitly Declares
5055	**ShaderClockKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_shader_clock**
5249	**SampleMaskOverrideCoverageNV**	**SampleRateShading** Reserved. Also see extension: **SPV_NV_sample_mask_override_coverage**
5251	**GeometryShaderPassthroughNV**	**Geometry** Reserved. Also see extension: **SPV_NV_geometry_shader_passthrough**
5254	**ShaderViewportIndexLayerEXT**	**MultiViewport** Reserved. Also see extension: **SPV_EXT_shader_viewport_index_layer**
5254	**ShaderViewportIndexLayerNV**	**MultiViewport** Reserved. Also see extension: **SPV_NV_viewport_array2**
5255	**ShaderViewportMaskNV**	**ShaderViewportIndexLayerNV** Reserved. Also see extension: **SPV_NV_viewport_array2**
5259	**ShaderStereoViewNV**	**ShaderViewportMaskNV** Reserved. Also see extension: **SPV_NV_stereo_view_rendering**
5260	**PerViewAttributesNV**	**MultiView** Reserved. Also see extension: **SPV_NVX_multiview_per_view_attributes**

	Capability	Implicitly Declares
5265	**FragmentFullyCoveredEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_fragment_fully_covered**
5266	**MeshShadingNV**	**Shader** Reserved. Also see extension: **SPV_NV_mesh_shader**
5282	**ImageFootprintNV**	Reserved. Also see extension: **SPV_NV_shader_image_footprint**
5284	**FragmentBarycentricKHR**	Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5284	**FragmentBarycentricNV**	Reserved. Also see extensions: **SPV_NV_fragment_shader_barycentric**, **SPV_KHR_fragment_shader_barycentric**
5288	**ComputeDerivativeGroupQuadsNV**	Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5291	**FragmentDensityEXT**	**Shader** Reserved. Also see extensions: **SPV_EXT_fragment_invocation_density**, **SPV_NV_shading_rate**
5291	**ShadingRateNV**	**Shader** Reserved. Also see extensions: **SPV_NV_shading_rate**, **SPV_EXT_fragment_invocation_density**
5297	**GroupNonUniformPartitionedNV**	Reserved. Also see extension: **SPV_NV_shader_subgroup_partitioned**

	Capability	Implicitly Declares
5301	**ShaderNonUniform** Uses the **NonUniform** decoration on a variable or instruction.	**Shader** Missing before **version 1.5**.
5301	**ShaderNonUniformEXT**	**Shader** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5302	**RuntimeDescriptorArray** Uses arrays of resources which are sized at run-time.	**Shader** Missing before **version 1.5**.
5302	**RuntimeDescriptorArrayEXT**	**Shader** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5303	**InputAttachmentArrayDynamicIndexing** Arrays of **InputAttachment**s use dynamically uniform indexing.	**InputAttachment** Missing before **version 1.5**.
5303	**InputAttachmentArrayDynamicIndexingEXT**	**InputAttachment** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5304	**UniformTexelBufferArrayDynamicIndexing** Arrays of **SampledBuffer**s use dynamically uniform indexing.	**SampledBuffer** Missing before **version 1.5**.
5304	**UniformTexelBufferArrayDynamicIndexingEXT**	**SampledBuffer** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5305	**StorageTexelBufferArrayDynamicIndexing** Arrays of **ImageBuffer**s use dynamically uniform indexing.	**ImageBuffer** Missing before **version 1.5**.
5305	**StorageTexelBufferArrayDynamicIndexingEXT**	**ImageBuffer** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**

Capability		Implicitly Declares
5306	**UniformBufferArrayNonUniformIndexing** **Block**-decorated arrays in uniform storage classes use non-uniform indexing.	**ShaderNonUniform** Missing before **version 1.5**.
5306	**UniformBufferArrayNonUniformIndexingEXT**	**ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5307	**SampledImageArrayNonUniformIndexing** Arrays of sampled images use non-uniform indexing.	**ShaderNonUniform** Missing before **version 1.5**.
5307	**SampledImageArrayNonUniformIndexingEXT**	**ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5308	**StorageBufferArrayNonUniformIndexing** Arrays in the **StorageBuffer** storage class or **BufferBlock**-decorated arrays use non-uniform indexing.	**ShaderNonUniform** Missing before **version 1.5**.
5308	**StorageBufferArrayNonUniformIndexingEXT**	**ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5309	**StorageImageArrayNonUniformIndexing** Arrays of non-sampled images use non-uniform indexing.	**ShaderNonUniform** Missing before **version 1.5**.
5309	**StorageImageArrayNonUniformIndexingEXT**	**ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5310	**InputAttachmentArrayNonUniformIndexing** Arrays of **InputAttachment**s use non-uniform indexing.	**InputAttachment**, **ShaderNonUniform** Missing before **version 1.5**.
5310	**InputAttachmentArrayNonUniformIndexingEXT**	**InputAttachment**, **ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**

Capability		Implicitly Declares
5311	**UniformTexelBufferArrayNonUniformIndexing** Arrays of **SampledBuffer**s use non-uniform indexing.	**SampledBuffer**, **ShaderNonUniform** Missing before **version 1.5**.
5311	**UniformTexelBufferArrayNonUniformIndexingEXT**	**SampledBuffer**, **ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5312	**StorageTexelBufferArrayNonUniformIndexing** Arrays of **ImageBuffer**s use non-uniform indexing.	**ImageBuffer**, **ShaderNonUniform** Missing before **version 1.5**.
5312	**StorageTexelBufferArrayNonUniformIndexingEXT**	**ImageBuffer**, **ShaderNonUniform** Missing before **version 1.5**. Also see extension: **SPV_EXT_descriptor_indexing**
5340	**RayTracingNV**	**Shader** Reserved. Also see extension: **SPV_NV_ray_tracing**
5341	**RayTracingMotionBlurNV**	**Shader** Reserved. Also see extension: **SPV_NV_ray_tracing_motion_blur**
5345	**VulkanMemoryModel** Uses the **Vulkan** memory model. This capability must be declared if and only if the **Vulkan** memory model is declared.	Missing before **version 1.5**.
5345	**VulkanMemoryModelKHR**	Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**
5346	**VulkanMemoryModelDeviceScope** Uses **Device** scope with any instruction when the **Vulkan** memory model is declared.	Missing before **version 1.5**.
5346	**VulkanMemoryModelDeviceScopeKHR**	Missing before **version 1.5**. Also see extension: **SPV_KHR_vulkan_memory_model**

Capability		Implicitly Declares
5347	**PhysicalStorageBufferAddresses** Uses physical addressing on storage buffers.	**Shader** Missing before **version 1.5**. Also see extensions: **SPV_EXT_physical_storage_buffer**, **SPV_KHR_physical_storage_buffer**
5347	**PhysicalStorageBufferAddressesEXT**	**Shader** Missing before **version 1.5**. Also see extension: **SPV_EXT_physical_storage_buffer**
5350	**ComputeDerivativeGroupLinearNV**	Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5353	**RayTracingProvisionalKHR**	**Shader** Reserved. Also see extension: **SPV_KHR_ray_tracing**
5357	**CooperativeMatrixNV**	**Shader** Reserved. Also see extension: **SPV_NV_cooperative_matrix**
5363	**FragmentShaderSampleInterlockEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5372	**FragmentShaderShadingRateInterlockEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5373	**ShaderSMBuiltinsNV**	**Shader** Reserved. Also see extension: **SPV_NV_shader_sm_builtins**

Capability		Implicitly Declares
5378	**FragmentShaderPixelInterlockEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_fragment_shader_interlock**
5379	**DemoteToHelperInvocation**	**Shader** Missing before **version 1.6**.
5379	**DemoteToHelperInvocationEXT**	**Shader** Missing before **version 1.6**. Also see extension: **SPV_EXT_demote_to_helper_invocation**
5390	**BindlessTextureNV**	Reserved. Also see extension: **SPV_NV_bindless_texture**
5568	**SubgroupShuffleINTEL**	Reserved. Also see extension: **SPV_INTEL_subgroups**
5569	**SubgroupBufferBlockIOINTEL**	Reserved. Also see extension: **SPV_INTEL_subgroups**
5570	**SubgroupImageBlockIOINTEL**	Reserved. Also see extension: **SPV_INTEL_subgroups**
5579	**SubgroupImageMediaBlockIOINTEL**	Reserved. Also see extension: **SPV_INTEL_media_block_io**
5582	**RoundToInfinityINTEL**	Reserved. Also see extension: **SPV_INTEL_float_controls2**
5583	**FloatingPointModeINTEL**	Reserved. Also see extension: **SPV_INTEL_float_controls2**
5584	**IntegerFunctions2INTEL**	**Shader** Reserved. Also see extension: **SPV_INTEL_shader_integer_functions2**

Capability		Implicitly Declares
5603	**FunctionPointersINTEL**	Reserved. Also see extension: **SPV_INTEL_function_pointers**
5604	**IndirectReferencesINTEL**	Reserved. Also see extension: **SPV_INTEL_function_pointers**
5606	**AsmINTEL**	Reserved. Also see extension: **SPV_INTEL_inline_assembly**
5612	**AtomicFloat32MinMaxEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float_min_max**
5613	**AtomicFloat64MinMaxEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float_min_max**
5616	**AtomicFloat16MinMaxEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float_min_max**
5617	**VectorComputeINTEL**	**VectorAnyINTEL** Reserved. Also see extension: **SPV_INTEL_vector_compute**
5619	**VectorAnyINTEL**	Reserved. Also see extension: **SPV_INTEL_vector_compute**
5629	**ExpectAssumeKHR**	Reserved. Also see extension: **SPV_KHR_expect_assume**
5696	**SubgroupAvcMotionEstimationINTEL**	Reserved. Also see extension: **SPV_INTEL_device_side_avc_motion_estimation**

Capability		Implicitly Declares
5697	**SubgroupAvcMotionEstimationIntraINTEL**	Reserved. Also see extension: **SPV_INTEL_device_side_avc_motion_estimation**
5698	**SubgroupAvcMotionEstimationChromaINTEL**	Reserved. Also see extension: **SPV_INTEL_device_side_avc_motion_estimation**
5817	**VariableLengthArrayINTEL**	Reserved. Also see extension: **SPV_INTEL_variable_length_array**
5821	**FunctionFloatControlINTEL**	Reserved. Also see extension: **SPV_INTEL_float_controls2**
5824	**FPGAMemoryAttributesINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_memory_attributes**
5837	**FPFastMathModeINTEL**	**Kernel** Reserved. Also see extension: **SPV_INTEL_fp_fast_math_mode**
5844	**ArbitraryPrecisionIntegersINTEL**	Reserved. Also see extension: **SPV_INTEL_arbitrary_precision_integers**
5845	**ArbitraryPrecisionFloatingPointINTEL**	Reserved. Also see extension: **SPV_INTEL_arbitrary_precision_floating_point**
5886	**UnstructuredLoopControlsINTEL**	Reserved. Also see extension: **SPV_INTEL_unstructured_loop_controls**
5888	**FPGALoopControlsINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_loop_controls**

Capability		Implicitly Declares
5892	**KernelAttributesINTEL**	Reserved. Also see extension: **SPV_INTEL_kernel_attributes**
5897	**FPGAKernelAttributesINTEL**	Reserved. Also see extension: **SPV_INTEL_kernel_attributes**
5898	**FPGAMemoryAccessesINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_memory_accesses**
5904	**FPGAClusterAttributesINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_cluster_attributes**
5906	**LoopFuseINTEL**	Reserved. Also see extension: **SPV_INTEL_loop_fuse**
5920	**FPGABufferLocationINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_buffer_location**
5922	**ArbitraryPrecisionFixedPointINTEL**	Reserved. Also see extension: **SPV_INTEL_arbitrary_precision_fixed_point**
5935	**USMStorageClassesINTEL**	Reserved. Also see extension: **SPV_INTEL_usm_storage_classes**
5943	**IOPipesINTEL**	Reserved. Also see extension: **SPV_INTEL_io_pipes**
5945	**BlockingPipesINTEL**	Reserved. Also see extension: **SPV_INTEL_blocking_pipes**
5948	**FPGARegINTEL**	Reserved. Also see extension: **SPV_INTEL_fpga_reg**
6016	**DotProductInputAll** Uses vector of any integer type as input to the dot product instructions	Missing before **version 1.6**.

Capability		Implicitly Declares
6016	**DotProductInputAllKHR**	Missing before **version 1.6**. Also see extension: **SPV_KHR_integer_dot_product**
6017	**DotProductInput4x8Bit** Uses vectors of four components of 8-bit integer type as inputs to the dot product instructions	**Int8** Missing before **version 1.6**.
6017	**DotProductInput4x8BitKHR**	**Int8** Missing before **version 1.6**. Also see extension: **SPV_KHR_integer_dot_product**
6018	**DotProductInput4x8BitPacked** Uses 32-bit integer scalars packing 4-component vectors of 8-bit integers as inputs to the dot product instructions	Missing before **version 1.6**.
6018	**DotProductInput4x8BitPackedKHR**	Missing before **version 1.6**. Also see extension: **SPV_KHR_integer_dot_product**
6019	**DotProduct** Uses dot product instructions	Missing before **version 1.6**.
6019	**DotProductKHR**	Missing before **version 1.6**. Also see extension: **SPV_KHR_integer_dot_product**
6025	**BitInstructions**	Reserved. Also see extension: **SPV_KHR_bit_instructions**
6033	**AtomicFloat32AddEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float_add**
6034	**AtomicFloat64AddEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float_add**
6089	**LongConstantCompositeINTEL**	Reserved. Also see extension: **SPV_INTEL_long_constant_composite**

Capability		Implicitly Declares
6094	**OptNoneINTEL**	Reserved. Also see extension: **SPV_INTEL_optnone**
6095	**AtomicFloat16AddEXT**	Reserved. Also see extension: **SPV_EXT_shader_atomic_float16_add**
6114	**DebugInfoModuleINTEL**	Reserved. Also see extension: **SPV_INTEL_debug_module**

3.32. Reserved Ray Flags

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

Reserved Ray Flags		Enabling Capabilities
0x0	**None**	
0x1	**OpaqueKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x2	**NoOpaqueKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x4	**TerminateOnFirstHitKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x8	**SkipClosestHitShaderKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x10	**CullBackFacingTrianglesKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x20	**CullFrontFacingTrianglesKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x40	**CullOpaqueKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.
0x80	**CullNoOpaqueKHR**	**RayQueryKHR**, **RayTracingKHR** Reserved.

Reserved Ray Flags		Enabling Capabilities
0x100	**SkipTrianglesKHR**	**RayTraversalPrimitiveCullingKHR** Reserved.
0x200	**SkipAABBsKHR**	**RayTraversalPrimitiveCullingKHR** Reserved.

3.33. Reserved Ray Query Intersection

Reserved Ray Query Intersection		Enabling Capabilities
0	**RayQueryCandidateIntersectionKHR**	**RayQueryKHR** Reserved.
1	**RayQueryCommittedIntersectionKHR**	**RayQueryKHR** Reserved.

3.34. Reserved Ray Query Committed Type

Reserved Ray Query Committed Type		Enabling Capabilities
0	**RayQueryCommittedIntersectionNoneKHR**	**RayQueryKHR** Reserved.
1	**RayQueryCommittedIntersectionTriangleKHR**	**RayQueryKHR** Reserved.
2	**RayQueryCommittedIntersectionGeneratedKHR**	**RayQueryKHR** Reserved.

3.35. Reserved Ray Query Candidate Type

Reserved Ray Query Candidate Type		Enabling Capabilities
0	**RayQueryCandidateIntersectionTriangleKHR**	**RayQueryKHR** Reserved.
1	**RayQueryCandidateIntersectionAABBKHR**	**RayQueryKHR** Reserved.

3.36. Reserved Fragment Shading Rate

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

Reserved Fragment Shading Rate		Enabling Capabilities
0x0	**None**	
0x1	**Vertical2Pixels**	**FragmentShadingRateKHR** Reserved.
0x2	**Vertical4Pixels**	**FragmentShadingRateKHR** Reserved.
0x4	**Horizontal2Pixels**	**FragmentShadingRateKHR** Reserved.
0x8	**Horizontal4Pixels**	**FragmentShadingRateKHR** Reserved.

3.37. Reserved FP Denorm Mode

Floating point denormalized handling mode.

Reserved FP Denorm Mode		Enabling Capabilities
0	**Preserve**	**FunctionFloatControlINTEL** Reserved.
1	**FlushToZero**	**FunctionFloatControlINTEL** Reserved.

3.38. Reserved FP Operation Mode

Floating point operation mode.

Reserved FP Operation Mode		Enabling Capabilities
0	**IEEE**	**FunctionFloatControlINTEL** Reserved.
1	**ALT**	**FunctionFloatControlINTEL** Reserved.

3.39. Quantization Mode

Quantization Mode		Enabling Capabilities
0	**TRN**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
1	**TRN_ZERO**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
2	**RND**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
3	**RND_ZERO**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
4	**RND_INF**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
5	**RND_MIN_INF**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
6	**RND_CONV**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
7	**RND_CONV_ODD**	**ArbitraryPrecisionFixedPointINTEL** Reserved.

3.40. Overflow Mode

Overflow Mode		Enabling Capabilities
0	**WRAP**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
1	**SAT**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
2	**SAT_ZERO**	**ArbitraryPrecisionFixedPointINTEL** Reserved.
3	**SAT_SYM**	**ArbitraryPrecisionFixedPointINTEL** Reserved.

3.41. Packed Vector Format

Used by:

- **OpSDot**
- **OpSDotKHR**
- **OpUDot**
- **OpUDotKHR**
- **OpSUDot**
- **OpSUDotKHR**
- **OpSDotAccSat**
- **OpSDotAccSatKHR**
- **OpUDotAccSat**
- **OpUDotAccSatKHR**
- **OpSUDotAccSat**
- **OpSUDotAccSatKHR**

Packed Vector Format		Enabling Capabilities
0	**PackedVectorFormat4x8Bit** Interpret 32-bit scalar integer operands as vectors of four 8-bit components. Vector components follow byte significance order with the lowest-numbered component stored in the least significant byte.	Missing before **version 1.6**.
0	**PackedVectorFormat4x8BitKHR**	Missing before **version 1.6**. Also see extension: **SPV_KHR_integer_dot_product**

3.42. Instructions

Form for each instruction:

Opcode Name (name-alias, name-alias, …) 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* also says "+ 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 *Result <id>* is always 32 bits. *Operands*, when present, are any literals, other instruction's *Result <id>*, etc., consumed by the instruction. Each operand is always 32 bits.			Capability **Enabling Capabilities** (when needed)
Word Count	*Opcode*	*Results*	*Operands*

3.42.1. 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. *Result Type* can be any type except **OpTypeVoid**. Each consumption of *Result <id>* yields an arbitrary, possibly different bit pattern or abstract value resulting in possibly different concrete, abstract, or opaque values.			
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** Missing before **version 1.1**.
4	321	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpAssumeTrueKHR TBD		Capability: **ExpectAssumeKHR** Reserved.
2	5630	*<id>* *Condition*

OpExpectKHR TBD				Capability: **ExpectAssumeKHR** Reserved.	
5	5631	*<id>* *Result Type*	*Result <id>*	*<id>* *Value*	*<id>* *ExpectedValue*

3.42.2. 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 nul 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* *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. It is an unsigned 32-bit integer. *File* is an **OpString** instruction and is the source-level file name. *Source* is the text of the source-level file. Each client API specifies what form the *Version* operand takes, per source language.					
3 + variable	3	*Source Language*	*Literal* *Version*	Optional *<id>* *File*	Optional *Literal* *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* *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* *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, … *Member* is an unsigned 32-bit integer.

Name is the string to assign to the member.

4 + variable	6	*<id>* *Type*	*Literal* *Member*	*Literal* *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 string being assigned a *Result <id>*.

3 + variable	7	*Result <id>*	*Literal* *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. *Line* is an unsigned 32-bit integer. *Column* is the source-level column number. *Column* is an unsigned 32-bit integer. **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) - must not be the last instruction in a block, which is defined to end with a termination 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* *Line*	*Literal* *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 must only appear after the annotation instructions (see the Logical Layout section). It must not 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.		Missing before **version 1.1**.
2 + variable	330	*Literal* *Process*

3.42.3. 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 decoration instructions targeting the same **OpDecorationGroup** instruction.

This instruction is only valid if the *Decoration* operand is a decoration that takes no **Extra Operands**, or takes **Extra Operands** that are not *<id>* operands.

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, …

Note: See **OpDecorate** for creating groups of decorations for consumption by **OpGroupMemberDecorate**

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

OpDecorationGroup

Deprecated (directly use non-group decoration instructions instead).

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

2	73	*Result <id>*

OpGroupDecorate

Deprecated (directly use non-group decoration instructions instead).

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. The *Targets* list must not include the *<id>* of any **OpDecorationGroup** instruction.

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

OpGroupMemberDecorate Deprecated (directly use non-group decoration instructions instead). 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*

OpDecorateId Add a Decoration to another <*id*>, using <*id*>*s* as **Extra Operands**. *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 decoration instructions targeting the same **OpDecorationGroup** instruction. This instruction is only valid if the *Decoration* operand is a decoration that takes **Extra Operands** that are <*id*> operands. All such <*id*> **Extra Operands** must be constant instructions or **OpVariable** instructions.				Missing before **version 1.2**.
3 + variable	332	*<id>* *Target*	*Decoration*	*<id>, <id>, …* See *Decoration*.

OpDecorateString (OpDecorateStringGOOGLE) Add a string Decoration to another <*id*>. *Target* is the <*id*> to decorate. It can potentially be any <*id*> that is a forward reference, except it must not be the <*id*> of an **OpDecorationGroup**. *Decoration* is a decoration that takes at least one *Literal* operand, and has only *Literal* string operands.				Missing before **version 1.4**.	
4 + variable	5632	*<id>* *Target*	*Decoration*	*Literal* See *Decoration*.	*Optional Literals* See *Decoration*.

<table>
<tr><td colspan="5">OpMemberDecorateString (OpMemberDecorateStringGOOGLE)

Add a string Decoration to a member of a structure type.

Structure Type is the <id> of an OpTypeStruct.

Member is the number of the member to decorate in the type. Member is an unsigned 32-bit integer. The first member is member 0, the next is member 1, …

Decoration is a decoration that takes at least one Literal operand, and has only Literal string operands.</td><td colspan="2">Missing before version 1.4.</td></tr>
<tr><td>5 + variable</td><td>5633</td><td><id>
Struct Type</td><td>Literal
Member</td><td>Decoration</td><td>Literal
See Decoration.</td><td>Optional Literals
See Decoration.</td></tr>
</table>

3.42.4. 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* *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. Before version 1.6, there must be an external specification defining the semantics for this extended instruction set. Starting with version 1.6, if *Name* starts with "NonSemantic.", including the period that separates the namespace "NonSemantic" from the rest of the name, it is encouraged for a specification to exist on the SPIR-V Registry, but it is not required. Starting with version 1.6, an extended instruction-set name which is prefixed with "NonSemantic." is guaranteed to contain only non-semantic instructions, and all **OpExtInst** instructions referencing this set can be ignored. All instructions within such a set must have only *<id>* operands; no literals. When literals are needed, then the *Result <id>* from an **OpConstant** or **OpString** instruction is referenced as appropriate. *Result <id>s* from these non-semantic instruction-set instructions must be used only in other non-semantic instructions. See Extended Instruction Sets for more information.			
3 + variable	11	*Result <id>*	*Literal* *Name*

OpExtInst Execute an instruction in an imported set of extended instructions. *Result Type* is 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*. It is an unsigned 32-bit integer. The semantics of the instruction are 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* *Instruction*	*<id>, <id>, …* *Operand 1*, *Operand 2*, …

3.42.5. 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, its execution model, and its interface.

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 must not have two **OpEntryPoint** instructions with the same Execution Model and the same *Name* string.

Interface is a list of *<id>* of global **OpVariable** instructions. These declare the set of global variables from a module that form the interface of this entry point. The set of *Interface <id>* must be equal to or a superset of the global **OpVariable** *Result <id>* referenced by the entry point’s static call tree, within the interface’s storage classes. Before **version 1.4**, the interface’s storage classes are limited to the **Input** and **Output** storage classes. Starting with **version 1.4**, the interface’s storage classes are all storage classes used in declaring all global variables referenced by the entry point’s call tree.

Interface <id> are forward references. Before **version 1.4**, duplication of these *<id>* is tolerated. Starting with **version 1.4**, an *<id>* must not appear more than once.

4 + variable	15	*Execution Model*	*<id>* *Entry Point*	*Literal* *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.

This instruction is only valid if the *Mode* operand is an execution mode that takes no **Extra Operands**, or takes **Extra Operands** that are not *<id>* operands.

3 + variable	16	*<id>* *Entry Point*	*Execution Mode* *Mode*	*Literal, Literal, …* See Execution Mode

<table>
<tr><td colspan="3">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.</td></tr>
<tr><td>2</td><td>17</td><td>Capability
Capability</td></tr>
</table>

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

Declare an execution mode for an entry point, using <id>s as Extra Operands.

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

Mode is the execution mode. See Execution Mode.

This instruction is only valid if the Mode operand is an execution mode that takes Extra Operands that are <id> operands. All such <id> Extra Operands must be constant instructions.</td><td>Missing before version 1.2.</td></tr>
<tr><td>3 + variable</td><td>331</td><td><id>
Entry Point</td><td>Execution Mode
Mode</td><td><id>, <id>, …
See Execution Mode</td></tr>
</table>

3.42.6. 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 must only be used with logical addressing operations, not physical, and only with non-externally visible shader Storage Classes: **Workgroup**, **CrossWorkgroup**, **Private**, **Function**, **Input**, and **Output**.		
2	20	*Result <id>*

OpTypeInt Declare a new *integer type*. *Width* specifies how many bits wide the type is. *Width* is an unsigned 32-bit integer. 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* *Width*	*Literal* *Signedness*

OpTypeFloat Declare a new *floating-point type*. *Width* specifies how many bits wide the type is. *Width* is an unsigned 32-bit integer. The bit pattern of a floating-point value is as described by the IEEE 754 standard.			
3	22	*Result <id>*	*Literal* *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. *Component Count* is an unsigned 32-bit integer. It must be at least 2. Components are numbered consecutively, starting with 0.				
4	23	*Result <id>*	*<id>* *Component Type*	*Literal* *Component Count*

OpTypeMatrix 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. *Column Count* is an unsigned 32-bit integer. 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**).				Capability: **Matrix**
4	24	*Result <id>*	*<id>* *Column Type*	*Literal* *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).

All the following literals are integers taking one operand each.

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 is 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 an image compatible with sampling operations
2 indicates an image compatible with read/write operations (a storage or subpass data image).

Image Format is the Image Format, which can be **Unknown**, as specified by 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 Depth*	*Literal Arrayed*	*Literal MS*	*Literal 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. It must not have a *Dim* of **SubpassData**. Additionally, starting with **version 1.6**, it must not have a *Dim* of **Buffer**.

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

OpTypeArray

Declare a new array 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 numbered 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.

See **OpArrayLength** for getting the *Length* of an array of this type.

Capability:
Shader

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

OpTypeStruct

Declare a new structure type.

Member N type is the type of member *N* of the structure. The first member is member 0, the next is member 1, … It is valid for the structure to have no members.

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* 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** uses this to declare the return type and parameter types of a function. *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*. It must not be **OpTypeVoid**				
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**. That **OpTypePointer** instruction must declare *Pointer Type* to be a pointer to an **OpTypeStruct**. Any consumption of *Pointer Type* before its **OpTypePointer** declaration must be a type-declaration instruction. *Storage Class* is the Storage Class of the memory holding the object pointed to.			Capability: **Addresses**, **PhysicalStorageBufferAddresses**
3	39	*<id>* *Pointer Type*	*Storage Class*

OpTypePipeStorage Declare the OpenCL pipe-storage type.		Capability: **PipeStorage** Missing before **version 1.1**.
2	322	*Result <id>*

OpTypeNamedBarrier Declare the named-barrier type.		Capability: **NamedBarrier** Missing before **version 1.1**.
2	327	*Result <id>*

OpTypeBufferSurfaceINTEL TBD			Capability: **VectorComputeINTEL** Reserved.
3	6086	*Result <id>*	*Access Qualifier* *AccessQualifier*

OpTypeStructContinuedINTEL TBD		Capability: **LongConstantCompositeINTEL** Reserved.
1 + variable	6090	*<id>, <id>, …* *Member 0 type*, *member 1 type*, …

3.42.7. 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.

4 + variable	43	*<id>* *Result Type*	*Result <id>*	*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 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 non-specialization constant-instruction declarations or an **OpUndef**.

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

<table>
<tr><td colspan="5">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 a 32-bit integer and is one of:
0: Non Normalized
1: Normalized

Sampler Filter Mode is the filter mode; a literal from Sampler Filter Mode.</td><td colspan="2">Capability:
LiteralSampler</td></tr>
<tr><td>6</td><td>45</td><td><id>
Result Type</td><td>Result <id></td><td>Sampler Addressing Mode</td><td>Literal Param</td><td>Sampler Filter Mode</td></tr>
</table>

<table>
<tr><td colspan="4">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</td></tr>
<tr><td>3</td><td>46</td><td><id>
Result Type</td><td>Result <id></td></tr>
</table>

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.

4 + variable	50	*<id>* *Result Type*	*Result <id>*	*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 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 constants, constant declarations, or an **OpUndef**.

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 is an unsigned 32-bit integer. It must equal one of the following opcodes.
OpSConvert, **OpUConvert** (missing before **version 1.4**), **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, **OpQuantizeToF16** is also valid.

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* that are *<id>s* must be either:
- the *<id>s* of other constant instructions, or
- **OpUndef**, when allowed by *opcode*, 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* *Opcode*	*<id>, <id>, …* *Operands*

OpConstantCompositeContinuedINTEL TBD		Capability: **LongConstantCompositeINTEL** Reserved.
1 + variable	6091	*<id>, <id>, …* *Constituents*

OpSpecConstantCompositeContinuedINTEL TBD		Capability: **LongConstantCompositeINTEL** Reserved.
1 + variable	6092	*<id>, <id>, …* *Constituents*

3.42.8. 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 must not be **Generic**. It must be the same as the *Storage Class* operand of the *Result Type*. *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* must not 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**: 3 components; the face and layer combine into the 3rd component, *layer_face*, such that face is *layer_face* % 6 and layer is floor(*layer_face* / 6) *Sample* must be an *integer type* scalar. It specifies which sample to select at the given coordinate. Behavior is undefined unless it is a valid *<id>* for the value 0 when 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. It must be a type with fixed size; i.e., it must not be, nor include, any **OpTypeRuntimeArray** types.

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

If present, any *Memory Operands* must begin with a memory operand literal. If not present, it is the same as specifying the memory operand **None**.

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

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.

If present, any *Memory Operands* must begin with a memory operand literal. If not present, it is the same as specifying the memory operand **None**.

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

OpCopyMemory

Copy from the memory pointed to by *Source* to the memory pointed to by *Target*. Both operands must be non-void pointers and having the same *<id> Type* operand in their **OpTypePointer** type declaration. Matching Storage Class is not required. The amount of memory copied is the size of the type pointed to. The copied type must have a fixed size; i.e., it must not be, nor include, any **OpTypeRuntimeArray** types.

If present, any *Memory Operands* must begin with a memory operand literal. If not present, it is the same as specifying the memory operand **None**. Before **version 1.4**, at most one memory operands mask can be provided. Starting with **version 1.4** two masks can be provided, as described in **Memory Operands**. If no masks or only one mask is present, it applies to both *Source* and *Target*. If two masks are present, the first applies to *Target* and must not include **MakePointerVisible**, and the second applies to *Source* and must not include **MakePointerAvailable**.

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

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 must not 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 is made.

If present, any *Memory Operands* must begin with a memory operand literal. If not present, it is the same as specifying the memory operand **None**. Before **version 1.4**, at most one memory operands mask can be provided. Starting with **version 1.4** two masks can be provided, as described in **Memory Operands**. If no masks or only one mask is present, it applies to both *Source* and *Target*. If two masks are present, the first applies to *Target* and must not include **MakePointerVisible**, and the second applies to *Source* and must not include **MakePointerAvailable**.

Capability:
Addresses

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

OpAccessChain

Create a pointer into a *composite* object.

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* selects 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 applies similarly to that result, and so on. Once any non-composite type is reached, there must be no remaining (unused) indexes.

Each index in *Indexes*
- must have a scalar integer type
- is treated as signed
- if indexing into a structure, must be an **OpConstant** whose value is in bounds for selecting a member
- if indexing into a vector, array, or matrix, with the result type being a logical pointer type, causes undefined behavior if not in bounds.

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 an initial dereference of *Base*: *Base* is treated as the address of an element in an array, and a new element address is computed from *Base* and *Element* to become the **OpAccessChain** *Base* to dereference as per **OpAccessChain**. This computed *Base* has the same type as the originating *Base*. To compute the new element address, *Element* is treated as a signed count of elements *E*, relative to the original *Base* element *B*, and the address of element *B* + *E* is computed using enough precision to avoid overflow and underflow. For objects in the **Uniform**, **StorageBuffer**, or **PushConstant** storage classes, the element's address or location is calculated using a stride, which will be the *Base*-type's *Array Stride* if the *Base* type is decorated with **ArrayStride**. For all other objects, the implementation calculates the element's address or location. With one exception, undefined behavior results when *B* + *E* is not an element in the same array (same innermost array, if array types are nested) as *B*. The exception being when *B* + *E* = *L*, where *L* is the length of the array: the address computation for element *L* is done with the same stride as any other *B* + *E* computation that stays within the array. Note: If *Base* is typed to be a pointer to an array and the desired operation is to select an element of that array, **OpAccessChain** should be directly used, as its first *Index* selects the array element.					Capability: **Addresses**, **VariablePointers**, **VariablePointersStorageBuffer**, **PhysicalStorageBufferAddresses**	
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 be a logical pointer to an **OpTypeStruct** whose last member is a run-time array. *Array member* is an unsigned 32-bit integer index of the last member of the structure that *Structure* points to. That member's type must be from **OpTypeRuntimeArray**.				Capability: **Shader**	
5	68	*<id>* *Result Type*	*Result <id>*	*<id>* *Structure*	*Literal* *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*

OpPtrEqual 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 *Boolean type* scalar. The types of *Operand 1* and *Operand 2* must be **OpTypePointer** of the same type.				Missing before **version 1.4**.	
5	401	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpPtrNotEqual 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 *Boolean type* scalar. The types of *Operand 1* and *Operand 2* must be **OpTypePointer** of the same type.				Missing before **version 1.4**.	
5	402	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpPtrDiff Element-number subtraction: The number of elements to add to *Operand 2* to get to *Operand 1*. *Result Type* must be an *integer type* scalar. It is computed as a signed value, as negative differences are allowed, independently of the signed bit in the type. The result equals the low-order *N* bits of the correct result *R*, where *R* is computed with enough precision to avoid overflow and underflow and *Result Type* has a bitwidth of *N* bits. The units of *Result Type* are a count of elements. I.e., the same value you would use as the *Element* operand to **OpPtrAccessChain**. The types of *Operand 1* and *Operand 2* must be **OpTypePointer** of exactly the same type, and point to a type that can be aggregated into an array. For an array of length *L*, *Operand 1* and *Operand 2* can point to any element in the range *[0, L]*, where element *L* is outside the array but has a representative address computed with the same stride as elements in the array. Additionally, *Operand 1* must be a valid *Base* operand of **OpPtrAccessChain**. Behavior is undefined if *Operand 1* and *Operand 2* are not pointers to element numbers in *[0, L]* in the same array.				Capability: **Addresses**, **VariablePointers**, **VariablePointersStorageBuffer** Missing before **version 1.4**.	
5	403	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

3.42.9. 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 terminates with the next **OpFunctionEnd** instruction. *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 are 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*, …

3.42.10. Image Instructions

OpSampledImage

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

Result Type must be the **OpTypeSampledImage** type whose *Image Type* operand is the type of *Image*.

Image is an object whose type is an **OpTypeImage**, whose *Sampled* operand is 0 or 1, and whose *Dim* operand is not **SubpassData**. Additionally, starting with **version 1.6**, the *Dim* operand must not be **Buffer**.

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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

Image Operands encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands 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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

D_{ref} is the depth-comparison reference value. It must be a 32-bit *floating-point type* scalar.

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

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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

D_{ref} is the depth-comparison reference value. It must be a 32-bit *floating-point type* scalar.

Image Operands encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands must be present.

Capability:
Shader

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 is (*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 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 is (*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 appear after all used components. *Image Operands* encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands 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 is (*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 appear after all used components. D_{ref} /*q* is the depth-comparison reference value. D_{ref} must be a 32-bit *floating-point type* scalar. *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 is (*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 appear after all used components.

D_{ref} */q* is the depth-comparison reference value. D_{ref} must be a 32-bit *floating-point type* scalar.

Image Operands encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands 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 an image whose *Sampled* operand is 1.

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 must not 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. The MS operand of the underlying OpTypeImage must be 0.

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 gathered from all four texels. It must be a 32-bit integer type scalar. Behavior is undefined if its value is not 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 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. The MS operand of the underlying OpTypeImage must be 0.

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. It must be a 32-bit floating-point type scalar.

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*. It must be a scalar or vector with component type 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 *Arrayed* operand is 1, then additional capabilities may be required; e.g., **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*. See the client API specification for handling of coordinates outside the image.

If the *Image Dim* operand is **SubpassData**, *Coordinate* is relative to the current fragment location. See the client API specification for more detail on how these coordinates are applied.

If 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 *Arrayed* operand is 1, then additional capabilities may be required; e.g., **ImageCubeArray**, or **ImageMSArray**. Its *Dim* operand must not 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*. See the client API specification for handling of coordinates outside the image.

Texel is the data to write. It must be a scalar or vector with component type 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="4">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 the 1D dimensionality,
2 for the 2D and Cube dimensionalities,
3 for the 3D dimensionality,
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. See the client API specification for additional image type restrictions.

Level of Detail is used to compute which mipmap level to query, as specified by the client API.</td><td colspan="2">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 the 1D and Buffer dimensionalities,
2 for the 2D, Cube, and Rect dimensionalities,
3 for the 3D dimensionality,
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 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 those listed under Result Type, above. Additionally, if its Dim is 1D, 2D, 3D, or Cube, it must also have either an MS of 1 or a Sampled of 0 or 2. There is no implicit level-of-detail consumed by this instruction. See OpImageQuerySizeLod for querying images having level of detail. See the client API specification for additional image type restrictions.</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>

<table>
<tr><td colspan="4">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 contains the mipmap array layer.
The second component of the result contains the implicit level of detail relative to the base level.

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

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:
ImageQuery</td></tr>
<tr><td>5</td><td>105</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Sampled Image</td><td><id>
Coordinate</td></tr>
</table>

<table>
<tr><td colspan="4">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 specified by the client API.

Image must be an object whose type is OpTypeImage. Its Dim operand must be one of 1D, 2D, 3D, or Cube. See the client API specification for additional image type restrictions.</td><td>Capability:
Kernel, ImageQuery</td></tr>
<tr><td>4</td><td>106</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Image</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel, ImageQuery</td></tr>
<tr><td>4</td><td>107</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Image</td></tr>
</table>

<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 holds 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. Its OpTypeImage must not have a Dim of Buffer. The MS operand of the underlying OpTypeImage must be 0.

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 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>

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 holds 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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

Image Operands encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands must be present.

Capability:
SparseResidency

7 + variable	306	<id> *Result Type*	*Result <id>*	<id> *Sampled Image*	<id> *Coordinate*	*Image Operands*	<id>	Optional <id>, <id>, …

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 holds 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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

D_{ref} is the depth-comparison reference value. It must be a 32-bit *floating-point type* scalar.

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:
SparseResidency

6 + variable	307	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseSampleDrefExplicitLod

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 holds 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**. Its **OpTypeImage** must not have a *Dim* of **Buffer**. The *MS* operand of the underlying **OpTypeImage** must be 0.

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 appear after all used components.

D_{ref} is the depth-comparison reference value. It must be a 32-bit *floating-point type* scalar.

Image Operands encodes what operands follow, as per Image Operands. Either **Lod** or **Grad** image operands must be present.

Capability: **SparseResidency**

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

OpImageSparseSampleProjImplicitLod

~~Sample a sparse image with a projective coordinate and an implicit level of detail.~~

Capability: **SparseResidency**

Reserved.

5 + variable	309	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseSampleProjExplicitLod

~~Sample a sparse image with a projective coordinate using an explicit level of detail.~~

Capability: **SparseResidency**

Reserved.

7 + variable	310	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*Image Operands*	*<id>*	Optional *<id>, <id>, …*

OpImageSparseSampleProjDrefImplicitLod ~~Sample a sparse image with a projective coordinate, doing depth-comparison, with an implicit level of detail.~~							Capability: **SparseResidency** Reserved.	
6 + variable	311	*<id> Result Type*	*Result <id>*	*<id> Sampled Image*	*<id> Coordinate*	*<id>* D_{ref}	Optional *Image Operands*	Optional *<id>, <id>, …*

OpImageSparseSampleProjDrefExplicitLod ~~Sample a sparse image with a projective coordinate, doing depth-comparison, using an explicit level of detail.~~							Capability: **SparseResidency** Reserved.		
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 sampled sparse image. *Result Type* must be an **OpTypeStruct** with two members. The first member's type must be an *integer type* scalar. It holds 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 must not 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

Capability:
SparseResidency

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 holds 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 gathered from all four texels. It must be a 32-bit *integer type* scalar. Behavior is undefined if its value is not 0, 1, 2 or 3.

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

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

OpImageSparseDrefGather

Capability:
SparseResidency

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 holds 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*.

D_{ref} is the depth-comparison reference value. It must be a 32-bit *floating-point type* scalar.

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

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 results in 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 holds 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*. It must be a scalar or vector with component type 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*. See the client API specification for handling of coordinates outside the image.

The *Image Dim* operand must not be **SubpassData**. 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>, …*

OpImageSampleFootprintNV

TBD

Capability:
ImageFootprintNV

Reserved.

7 + variable	5283	<id> *Result Type*	*Result <id>*	<id> *Sampled Image*	<id> *Coordinate*	<id> *Granularity*	<id> *Coarse*	Optional *Image Operands*	Optional <id>, <id>, …

3.42.11. Conversion Instructions

OpConvertFToU Convert value numerically 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. Behavior is undefined if *Result Type* is not wide enough to hold the converted value. *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 numerically from floating point to signed integer, with round toward 0.0. *Result Type* must be a scalar or vector of *integer type*. Behavior is undefined if *Result Type* is not wide enough to hold the converted value. *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 numerically 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 numerically 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 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 must not equal the component width in *Result Type*.

Results are computed per component.

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

OpSConvert

Convert 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 must not 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 numerically from one floating-point width to another 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 must not equal the component width in *Result Type*.

Results are computed per component.

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

OpQuantizeToF16

Capability:
Shader

Quantize a floating-point value to what is expressible by a 16-bit 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 may be either +0 or -0.

The **RelaxedPrecision** Decoration has no effect on this instruction.

Results are computed per component.

4	116	<id> *Result Type*	*Result <id>*	<id> *Value*

OpConvertPtrToU

Capability:
Addresses, **PhysicalStorageBufferAddresses**

Bit pattern-preserving conversion of a pointer to an unsigned scalar integer of possibly different bit width.

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

Pointer must be a physical pointer type. If the bit width of *Pointer* is smaller than that of *Result Type*, the conversion zero extends *Pointer*. If the bit width of *Pointer* is larger than that of *Result Type*, the conversion truncates *Pointer*. For same bit width *Pointer* and *Result Type*, this is the same as **OpBitcast**.

4	117	<id> *Result Type*	*Result <id>*	<id> *Pointer*

<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>

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

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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>119</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Unsigned Value</td></tr>
</table>

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

Bit pattern-preserving conversion of an unsigned scalar integer to a pointer.

Result Type must be a physical pointer type.

Integer Value must be a scalar of integer type, whose Signedness operand is 0. If the bit width of Integer Value is smaller than that of Result Type, the conversion zero extends Integer Value. If the bit width of Integer Value is larger than that of Result Type, the conversion truncates Integer Value. For same-width Integer Value and Result Type, this is the same as OpBitcast.</td><td>Capability:
Addresses, PhysicalStorageBuffer Addresses</td></tr>
<tr><td>4</td><td>120</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Integer Value</td></tr>
</table>

OpPtrCastToGeneric 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.				Capability: **Kernel**
4	121	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpGenericCastToPtr 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.				Capability: **Kernel**
4	122	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*

OpGenericCastToPtrExplicit 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**.				Capability: **Kernel**	
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*.

Before **version 1.5**: If either *Result Type* or *Operand* is a pointer, the other must be a pointer or an integer scalar.
Starting with **version 1.5**: If either *Result Type* or *Operand* is a pointer, the other must be a pointer, an integer scalar, or an integer vector.

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*

3.42.12. 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. It is interpreted as a 0-based index of which component of *Vector* to extract.

Behavior 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 are copied from.

Component is the value 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. It is interpreted as a 0-based index of which component to modify.

Behavior 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. Each component is an unsigned 32-bit integer. 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.

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. If constructing a vector, the total number of components in all the operands must equal the number of components in *Result Type*.

Constituents 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. If 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. Each index is an unsigned 32-bit integer.

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*. Each index is an unsigned 32-bit integer.

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

OpCopyObject

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

Result Type must equal *Operand* type. *Result Type* can be any type except **OpTypeVoid**.

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

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

Transpose a matrix.

Result Type must be an OpTypeMatrix.

Matrix must be an object of type OpTypeMatrix. The number of columns and the column size of Matrix must be the reverse of those in Result Type. The types of the scalar components in Matrix and Result Type must be the same.

Matrix must have of type of OpTypeMatrix.</td><td>Capability:
Matrix</td></tr>
<tr><td>4</td><td>84</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Matrix</td></tr>
</table>

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

Make a logical copy of Operand. There are no pointer dereferences involved.

Result Type must not equal the type of Operand (see OpCopyObject), but Result Type must logically match the Operand type.

Logically match is recursively defined by these three rules:
1. They must be either both be OpTypeArray or both be OpTypeStruct
2. If they are OpTypeArray:
- they must have the same Length operand, and
- their Element Type operands must be either the same or must logically match.
3. If they are OpTypeStruct:
- they must have the same number of Member type, and
- Member N type for the same N in the two types must be either the same or must logically match.</td><td>Missing before version 1.4.</td></tr>
<tr><td>4</td><td>400</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Operand</td></tr>
</table>

3.42.13. 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

Inverts the sign bit of *Operand*. (Note, however, that **OpFNegate** is still considered a floating-point instruction, and so is subject to the general floating-point rules regarding, for example, subnormals and NaN propagation).

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*.

The resulting value equals the low-order *N* bits of the correct result *R*, where *N* is the component width and *R* is computed with enough precision to avoid overflow and underflow.

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*. The resulting value equals the low-order *N* bits of the correct result *R*, where *N* is the component width and *R* is computed with enough precision to avoid overflow and underflow. 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*.

The resulting value equals the low-order *N* bits of the correct result *R*, where *N* is the component width and *R* is computed with enough precision to avoid overflow and underflow.

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. Behavior 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. Behavior is undefined if *Operand 2* is 0. Behavior is undefined if *Operand 2* is -1 and *Operand 1* is the minimum representable value for the operands' type, causing signed overflow.

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.

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. Behavior is undefined if *Operand 2* is 0.

5	137	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSRem

Signed remainder operation for the remainder whose sign matches the sign of *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. Behavior is undefined if *Operand 2* is 0. Behavior is undefined if *Operand 2* is -1 and *Operand 1* is the minimum representable value for the operands' type, causing signed overflow. Otherwise, the result is the *remainder* *r* of *Operand 1* divided by *Operand 2* where if *r* ¬ 0, the sign of *r* is the same as the sign of *Operand 1*.

5	138	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSMod

Signed remainder operation for the remainder whose sign matches the sign of *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. Behavior is undefined if *Operand 2* is 0. Behavior is undefined if *Operand 2* is -1 and *Operand 1* is the minimum representable value for the operands' type, causing signed overflow. Otherwise, the result is the *remainder* *r* of *Operand 1* divided by *Operand 2* where if *r* ¬ 0, the sign of *r* is the same as the sign of *Operand 2*.

5	139	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFRem

The floating-point *remainder* whose sign matches the sign of *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. Otherwise, the result is the *remainder* *r* of *Operand 1* divided by *Operand 2* where if *r* ¬ 0, the sign of *r* is the same as the sign of *Operand 1*.

5	140	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFMod

The floating-point *remainder* whose sign matches the sign of *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. Otherwise, the result is the *remainder* *r* of *Operand 1* divided by *Operand 2* where if *r* ¬ 0, the sign of *r* is the same as the sign of *Operand 2*.

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

Capability:
Matrix

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*.

5	143	*<id>* *Result Type*	*Result <id>*	*<id>* *Matrix*	*<id>* *Scalar*

<table>
<tr><td colspan="4">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.</td><td colspan="2">Capability:
Matrix</td></tr>
<tr><td>5</td><td>144</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Vector</td><td><id>
Matrix</td></tr>
</table>

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

Linear-algebraic Matrix X Vector.

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.</td><td colspan="2">Capability:
Matrix</td></tr>
<tr><td>5</td><td>145</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Matrix</td><td><id>
Vector</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td colspan="2">Capability:
Matrix</td></tr>
<tr><td>5</td><td>146</td><td><id>
Result Type</td><td>Result <id></td><td><id>
LeftMatrix</td><td><id>
RightMatrix</td></tr>
</table>

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 be vectors of 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*

OpSDot (OpSDotKHR)

Signed integer dot product of *Vector 1* and *Vector 2*.

Result Type must be an integer type whose *Width* must be greater than or equal to that of the components of *Vector 1* and *Vector 2*.

Vector 1 and *Vector 2* must have the same type.

Vector 1 and *Vector 2* must be either 32-bit integers (enabled by the **DotProductInput4x8BitPacked** capability) or vectors of integer type (enabled by the **DotProductInput4x8Bit** or **DotProductInputAll** capability).

When *Vector 1* and *Vector 2* are scalar integer types, *Packed Vector Format* must be specified to select how the integers are to be interpreted as vectors.

All components of the input vectors are sign-extended to the bit width of the result's type. The sign-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. The resulting value will equal the low-order N bits of the correct result R, where N is the result width and R is computed with enough precision to avoid overflow and underflow.

Capability:
DotProduct

Missing before **version 1.6**.

5 + variable	4450	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector 1*	*<id>* *Vector 2*	Optional *Packed Vector Format* *Packed Vector Format*

<table>
<tr>
<td colspan="5">OpUDot (OpUDotKHR)

Unsigned integer dot product of Vector 1 and Vector 2.

Result Type must be an integer type with Signedness of 0 whose Width must be greater than or equal to that of the components of Vector 1 and Vector 2.

Vector 1 and Vector 2 must have the same type.

Vector 1 and Vector 2 must be either 32-bit integers (enabled by the DotProductInput4x8BitPacked capability) or vectors of integer type with Signedness of 0 (enabled by the DotProductInput4x8Bit or DotProductInputAll capability).

When Vector 1 and Vector 2 are scalar integer types, Packed Vector Format must be specified to select how the integers are to be interpreted as vectors.

All components of the input vectors are zero-extended to the bit width of the result's type. The zero-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. The resulting value will equal the low-order N bits of the correct result R, where N is the result width and R is computed with enough precision to avoid overflow and underflow.</td>
<td colspan="2">Capability:
DotProduct

Missing before version 1.6.</td>
</tr>
<tr>
<td>5 + variable</td>
<td>4451</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Vector 1</td>
<td><id>
Vector 2</td>
<td>Optional
Packed Vector Format
Packed Vector Format</td>
</tr>
</table>

OpSUDot (OpSUDotKHR)

Mixed-signedness integer dot product of *Vector 1* and *Vector 2*. Components of *Vector 1* are treated as signed, components of *Vector 2* are treated as unsigned.

Result Type must be an integer type whose *Width* must be greater than or equal to that of the components of *Vector 1* and *Vector 2*.

Vector 1 and *Vector 2* must be either 32-bit integers (enabled by the **DotProductInput4x8BitPacked** capability) or vectors of integer type with the same number of components and same component *Width* (enabled by the **DotProductInput4x8Bit** or **DotProductInputAll** capability). When *Vector 1* and *Vector 2* are vectors, the components of *Vector 2* must have a *Signedness* of 0.

When *Vector 1* and *Vector 2* are scalar integer types, *Packed Vector Format* must be specified to select how the integers are to be interpreted as vectors.

All components of *Vector 1* are sign-extended to the bit width of the result's type. All components of *Vector 2* are zero-extended to the bit width of the result's type. The sign- or zero-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. The resulting value will equal the low-order N bits of the correct result R, where N is the result width and R is computed with enough precision to avoid overflow and underflow.

Capability:
DotProduct

Missing before **version 1.6**.

5 + variable	4452	*<id>* *Result Type*	*Result <id>*	*<id>* *Vector 1*	*<id>* *Vector 2*	Optional *Packed Vector Format* *Packed Vector Format*

<table>
<tr>
<td colspan="6">OpSDotAccSat (OpSDotAccSatKHR)

Signed integer dot product of Vector 1 and Vector 2 and signed saturating addition of the result with Accumulator.

Result Type must be an integer type whose Width must be greater than or equal to that of the components of Vector 1 and Vector 2.

Vector 1 and Vector 2 must have the same type.

Vector 1 and Vector 2 must be either 32-bit integers (enabled by the DotProductInput4x8BitPacked capability) or vectors of integer type (enabled by the DotProductInput4x8Bit or DotProductInputAll capability).

The type of Accumulator must be the same as Result Type.

When Vector 1 and Vector 2 are scalar integer types, Packed Vector Format must be specified to select how the integers are to be interpreted as vectors.

All components of the input vectors are sign-extended to the bit width of the result's type. The sign-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. Finally, the resulting sum is added to the input accumulator. This final addition is saturating.

If any of the multiplications or additions, with the exception of the final accumulation, overflow or underflow, the result of the instruction is undefined.</td>
<td colspan="2">Capability:
DotProduct

Missing before version 1.6.</td>
</tr>
<tr>
<td>6 + variable</td>
<td>4453</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Vector 1</td>
<td><id>
Vector 2</td>
<td><id>
Accumulator</td>
<td>Optional
Packed Vector Format
Packed Vector Format</td>
</tr>
</table>

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

OpUDotAccSat (OpUDotAccSatKHR)

Unsigned integer dot product of *Vector 1* and *Vector 2* and unsigned saturating addition of the result with *Accumulator*.

Result Type must be an integer type with *Signedness* of 0 whose *Width* must be greater than or equal to that of the components of *Vector 1* and *Vector 2*.

Vector 1 and *Vector 2* must have the same type.

Vector 1 and *Vector 2* must be either 32-bit integers (enabled by the **DotProductInput4x8BitPacked** capability) or vectors of integer type with *Signedness* of 0 (enabled by the **DotProductInput4x8Bit** or **DotProductInputAll** capability).

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

When *Vector 1* and *Vector 2* are scalar integer types, *Packed Vector Format* must be specified to select how the integers are to be interpreted as vectors.

All components of the input vectors are zero-extended to the bit width of the result's type. The zero-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. Finally, the resulting sum is added to the input accumulator. This final addition is saturating.

If any of the multiplications or additions, with the exception of the final accumulation, overflow or underflow, the result of the instruction is undefined.

</td>
<td colspan="2">

Capability:
DotProduct

Missing before **version 1.6**.

</td>
</tr>
<tr>
<td>6 + variable</td>
<td>4454</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Vector 1</td>
<td><id>
Vector 2</td>
<td><id>
Accumulator</td>
<td>Optional
Packed Vector Format
Packed Vector Format</td>
</tr>
</table>

<table>
<tr><td colspan="6">OpSUDotAccSat (OpSUDotAccSatKHR)

Mixed-signedness integer dot product of Vector 1 and Vector 2 and signed saturating addition of the result with Accumulator. Components of Vector 1 are treated as signed, components of Vector 2 are treated as unsigned.

Result Type must be an integer type whose Width must be greater than or equal to that of the components of Vector 1 and Vector 2.

Vector 1 and Vector 2 must be either 32-bit integers (enabled by the DotProductInput4x8BitPacked capability) or vectors of integer type with the same number of components and same component Width (enabled by the DotProductInput4x8Bit or DotProductInputAll capability). When Vector 1 and Vector 2 are vectors, the components of Vector 2 must have a Signedness of 0.

The type of Accumulator must be the same as Result Type.

When Vector 1 and Vector 2 are scalar integer types, Packed Vector Format must be specified to select how the integers are to be interpreted as vectors.

All components of Vector 1 are sign-extended to the bit width of the result’s type. All components of Vector 2 are zero-extended to the bit width of the result’s type. The sign- or zero-extended input vectors are then multiplied component-wise and all components of the vector resulting from the component-wise multiplication are added together. Finally, the resulting sum is added to the input accumulator. This final addition is saturating.

If any of the multiplications or additions, with the exception of the final accumulation, overflow or underflow, the result of the instruction is undefined.</td><td colspan="2">Capability:
DotProduct

Missing before version 1.6.</td></tr>
<tr><td>6 + variable</td><td>4455</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Vector 1</td><td><id>
Vector 2</td><td><id>
Accumulator</td><td>Optional
Packed Vector Format
Packed Vector Format</td></tr>
</table>

3.42.14. Bit Instructions

OpShiftRightLogical

Shift the bits in *Base* right by the number of bits specified in *Shift*. The most-significant bits are 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 resulting value is undefined if *Shift* is greater than or equal to 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 are 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 resulting value is undefined if *Shift* is greater than or equal to 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 are 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 resulting value is undefined if *Shift* is greater than or equal to 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*

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

Make a copy of an object, with a modified bit field that comes from another object.

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) 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 is consumed as an unsigned value. Count can be 0, in which case the result is Base.

Offset must be an integer type scalar. Offset is the lowest-order bit of the bit field. It is 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">Capability:
Shader, BitInstructions</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>

<table>
<tr><td colspan="5">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 is consumed as an unsigned value. Count can be 0, in which case the result is 0.

Offset must be an integer type scalar. Offset is the lowest-order bit of the bit field to extract from Base. It is 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">Capability:
Shader, BitInstructions</td></tr>
<tr><td>6</td><td>202</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Base</td><td><id>
Offset</td><td><id>
Count</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td colspan="2">Capability:
Shader, BitInstructions</td></tr>
<tr><td>6</td><td>203</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Base</td><td><id>
Offset</td><td><id>
Count</td></tr>
</table>

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 is taken from bit-number *Width - 1 - n* of *Base*, where *Width* is the **OpTypeInt** operand of the *Result Type*.				Capability: **Shader**, **BitInstructions**
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*

3.42.15. 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*

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>158</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>159</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>160</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td></tr>
</table>

OpLessOrGreater Deprecated (use **OpFOrdNotEqual**). Has the same semantics as **OpFOrdNotEqual**. *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** Missing after **version 1.5**.	
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. Before **version 1.4**, results are only computed per component.

Before **version 1.4**, *Result Type* must be a pointer, scalar, or vector. Starting with **version 1.4**, *Result Type* can additionally be a composite type other than a vector.

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

Condition must be a scalar or vector of *Boolean type*.

If *Condition* is a scalar and **true**, the result is *Object 1*. If *Condition* is a scalar and **false**, the result is *Object 2*.

If *Condition* is a vector, *Result Type* must be a vector with the same number of components as *Condition* and the result is a mix of *Object 1* and *Object 2*: If a component of *Condition* is **true**, the corresponding component in the result is taken from *Object 1*, otherwise it is taken from *Object 2*.

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*

3.42.16. 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 component width must be 32 bits. 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 component width must be 32 bits. 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 component width must be 32 bits. 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.Uses 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 component width must be 32 bits.

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.Uses 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 component width must be 32 bits.

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 component width must be 32 bits.

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. Uses local differencing based on the value of *P* for the current fragment's neighbors, and possibly, but not necessarily, includes 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 component width must be 32 bits. 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. Uses local differencing based on the value of *P* for the current fragment's neighbors, and possibly, but not necessarily, includes 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 component width must be 32 bits. 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 component width must be 32 bits. 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*

3.42.17. 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 except **OpTypeVoid**.

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. There must be exactly one *Parent i* for each parent block of the current block in the CFG. If *Parent i* is reachable in the CFG and *Variable i* is defined in a block, that defining block must dominate *Parent i*. All *Variables* must have a type matching *Result Type*.

Within a block, this instruction must appear before all non-**OpPhi** instructions (except for **OpLine** and **OpNoLine**, 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 label instruction of a block.

References to a block are through the *Result <id>* of its label.

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.

Starting with **version 1.6**, *True Label* and *False Label* **must not** be the same *<id>*.

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*. At least one weight must be non-zero. A weight of zero does not imply a branch is dead or permit its removal; branch weights are only hints. The sum of the two weights must not overflow a 32-bit unsigned integer.

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* is 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 branches 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 branches 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 branches to the *Default* label *<id>*. Each *literal* is interpreted with the type of *Selector*: The bit width of *Selector's* type is the width of each *literal's* type. If this width is not a multiple of 32-bits and the **OpTypeInt** *Signedness* is set to 1, the literal values are interpreted as being sign extended.

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 Deprecated (use **OpTerminateInvocation** or **OpDemoteToHelperInvocation**). Fragment-shader discard. Ceases all further processing in any invocation that executes it: Only instructions these invocations executed before **OpKill** 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.	Capability: **Shader**
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. *Value* must not have type **OpTypeVoid**. This instruction must be the last instruction in a block.		
2	254	*<id>* *Value*

OpUnreachable Behavior is undefined if this instruction is executed. 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* is an unsigned 32-bit integer. *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.			Capability: **Kernel**
3	256	*<id>* *Pointer*	*Literal* *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* is an unsigned 32-bit integer. *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.			Capability: **Kernel**
3	257	*<id>* *Pointer*	*Literal* *Size*

OpTerminateInvocation Fragment-shader terminate. Ceases all further processing in any invocation that executes it: Only instructions these invocations executed before **OpTerminateInvocation** 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.	Capability: **Shader** Missing before **version 1.6**.
1	4416

OpDemoteToHelperInvocation (OpDemoteToHelperInvocationEXT) Demote this fragment shader invocation to a helper invocation. Any stores to memory after this instruction are suppressed and the fragment does not write outputs to the framebuffer. Unlike the **OpTerminateInvocation** instruction, this does not necessarily terminate the invocation which might be needed for derivative calculations. It is not considered a flow control instruction (flow control does not become non-uniform) and does not terminate the block. The implementation may terminate helper invocations before the end of the shader as an optimization, but doing so must not affect derivative calculations and does not make control flow non-uniform. After an invocation executes this instruction, any subsequent load of **HelperInvocation** within that invocation will load an undefined value unless the **HelperInvocation** built-in variable is decorated with **Volatile** or the load included **Volatile** in its **Memory Operands** This instruction is only valid in the **Fragment** Execution Model.	Capability: **DemoteToHelperInvocation** Missing before **version 1.6**.
1	5380

3.42.18. Atomic Instructions

OpAtomicLoad

Atomically load through *Pointer* using the given *Semantics*. All subparts of the value that is loaded are 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*.

Memory is a memory *Scope*.

6	227	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

OpAtomicStore

Atomically store through *Pointer* using the given *Semantics*. All subparts of *Value* are 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.

Memory is a memory *Scope*.

5	228	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	229	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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* from *Value* only if *Original Value* equals *Comparator*, and
3) store the *New Value* back through *Pointer* only if *Original Value* equaled *Comparator*.

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* must not 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*.

Memory is a memory *Scope*.

9	230	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Equal*	*Memory Semantics <id>* *Unequal*	<id> *Value*	<id> *Comparator*

OpAtomicCompareExchangeWeak

Deprecated (use **OpAtomicCompareExchange**).

Has the same semantics as **OpAtomicCompareExchange**.

Memory is a memory *Scope*.

Capability:
Kernel

Missing after **version 1.3**.

9	231	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Equal*	*Memory Semantics <id>* *Unequal*	<id> *Value*	<id> *Comparator*

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*.

Memory is a memory *Scope*.

6	232	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

6	233	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	234	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	235	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	236	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	237	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	238	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	239	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	240	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	241	<id> *Result Type*	*Result <id>*	<id> *Pointer*	*Scope <id>* *Memory*	*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*.

Memory is a memory *Scope*.

7	242	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicFlagTestAndSet

Capability:
Kernel

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*.

The resulting values are undefined if an atomic flag is modified by an instruction other than **OpAtomicFlagTestAndSet** or **OpAtomicFlagClear**.

Memory is a memory *Scope*.

6	318	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

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 must not be **Acquire** or **AcquireRelease** The resulting values are undefined if an atomic flag is modified by an instruction other than **OpAtomicFlagTestAndSet** or **OpAtomicFlagClear**. *Memory* is a memory *Scope*.				Capability: **Kernel**
4	319	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

OpAtomicFMinEXT TBD						Capability: **AtomicFloat16MinMaxEXT**, **AtomicFloat32MinMaxEXT**, **AtomicFloat64MinMaxEXT** Reserved.	
7	5614	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicFMaxEXT TBD						Capability: **AtomicFloat16MinMaxEXT**, **AtomicFloat32MinMaxEXT**, **AtomicFloat64MinMaxEXT** Reserved.	
7	5615	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

OpAtomicFAddEXT TBD						Capability: **AtomicFloat16AddEXT**, **AtomicFloat32AddEXT**, **AtomicFloat64AddEXT** Reserved.	
7	6035	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*	*<id>* *Value*

3.42.19. 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 must 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 must 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 must 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 must only be used when multiple streams are present.		Capability: **GeometryStreams**
2	221	*<id>* *Stream*

3.42.20. 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 reach this point of execution before any invocation proceeds beyond it.

When *Execution* is **Workgroup** or larger, behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. When *Execution* is **Subgroup** or **Invocation**, the behavior of this instruction in non-uniform control flow is defined by the client API.

If *Semantics* is not **None**, this instruction also serves as an **OpMemoryBarrier** instruction, and also performs and adheres 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.

Before **version 1.3**, it is only valid to use this instruction with **TessellationControl**, **GLCompute**, or **Kernel** execution models. There is no such restriction starting with **version 1.3**.

If 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** are visible to any other invocation proceeding beyond 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 are 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. If the **Vulkan** memory model is declared, this ordering only applies to memory accesses that use the **NonPrivatePointer** memory operand or **NonPrivateTexel** image operand.

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 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.				Capability: **NamedBarrier** Missing before **version 1.1**.
4	328	<id> *Result Type*	*Result <id>*	<id> *Subgroup Count*

OpMemoryNamedBarrier 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 also performs and adheres 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.				Capability: **NamedBarrier** Missing before **version 1.1**.
4	329	<id> *Named Barrier*	*Scope <id>* *Memory*	*Memory Semantics <id>* *Semantics*

3.42.21. Group and Subgroup Instructions

OpGroupAsyncCopy

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 results in an event object that can be used by **OpGroupWaitEvents** to wait for the async copy to finish.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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*.

If *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.

If *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 if the addressing model is *Physical32* and 64 bit *integer type* scalar if 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* is not **OpConstantNull**, the result is the event object supplied by the *Event* operand.

Capability:
Kernel

9	259	<id> *Result Type*	*Result <id>*	*Scope <id> Execution*	<id> *Destination*	<id> *Source*	<id> *Num Elements*	<id> *Stride*	<id> *Event*

OpGroupWaitEvents Wait for events generated by **OpGroupAsyncCopy** operations to complete. *Events List* points to *Num Events* event objects, which is released after the wait is performed. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Num Events* must be a 32-bit *integer type* scalar. *Events List* must be a pointer to **OpTypeEvent**.				Capability: **Kernel**
4	260	*Scope <id>* *Execution*	<id> *Num Events*	<id> *Events List*

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**. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Predicate* must be a *Boolean type*.				Capability: **Groups**	
5	261	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	<id> *Predicate*

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**.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

Result Type must be a *Boolean type*.

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

Predicate must be a *Boolean type*.

Capability:
Groups

5	262	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Predicate*

OpGroupBroadcast

Broadcast the *Value* of the invocation identified by the local id *LocalId* to the result of all invocations in the group.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

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

LocalId must be an integer datatype. It must be a scalar, a vector with 2 components, or a vector with 3 components. Behavior is undefined unless *LocalId* is 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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is 0.

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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is 0.

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*

OpGroupFMin A floating-point minimum group operation specified for all values of *X* specified by invocations in the group. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Result Type* must be a scalar or vector of *floating-point type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is +INF. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	266	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	<id> *X*

OpGroupUMin An unsigned integer minimum group operation specified for all values of *X* specified by invocations in the group. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Result Type* must be a scalar or vector of *integer type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is UINT_MAX when *X* is 32 bits wide and ULONG_MAX when *X* is 64 bits wide. The type of *X* must be the same as *Result Type*.					Capability: **Groups**	
6	267	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	<id> *X*

OpGroupSMin A signed integer minimum group operation specified for all values of *X* specified by invocations in the group. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Result Type* must be a scalar or vector of *integer type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is INT_MAX when *X* is 32 bits wide and LONG_MAX when *X* is 64 bits wide. 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. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Result Type* must be a scalar or vector of *floating-point type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is -INF. 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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is 0.

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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is INT_MIN when *X* is 32 bits wide and LONG_MIN when *X* is 64 bits wide.

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*

OpSubgroupBallotKHR

See extension SPV_KHR_shader_ballot

Capability:
SubgroupBallotKHR

Reserved.

4	4421	<id> *Result Type*	*Result <id>*	<id> *Predicate*

OpSubgroupFirstInvocationKHR See extension SPV_KHR_shader_ballot				Capability: **SubgroupBallotKHR** Reserved.
4	4422	*<id>* *Result Type*	*Result <id>*	*<id>* *Value*

OpSubgroupAllKHR TBD				Capability: **SubgroupVoteKHR** Reserved.
4	4428	*<id>* *Result Type*	*Result <id>*	*<id>* *Predicate*

OpSubgroupAnyKHR TBD				Capability: **SubgroupVoteKHR** Reserved.
4	4429	*<id>* *Result Type*	*Result <id>*	*<id>* *Predicate*

OpSubgroupAllEqualKHR TBD				Capability: **SubgroupVoteKHR** Reserved.
4	4430	*<id>* *Result Type*	*Result <id>*	*<id>* *Predicate*

OpSubgroupReadInvocationKHR See extension SPV_KHR_shader_ballot				Capability: **SubgroupBallotKHR** Reserved.	
5	4432	*<id>* *Result Type*	*Result <id>*	*<id>* *Value*	*<id>* *Index*

OpGroupIAddNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5000	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupFAddNonUniformAMD TBD	Capability: **Groups** Reserved.

6	5001	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupFMinNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5002	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupUMinNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5003	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupSMinNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5004	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupFMaxNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5005	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupUMaxNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5006	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpGroupSMaxNonUniformAMD TBD					Capability: **Groups** Reserved.	
6	5007	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *X*

OpSubgroupShuffleINTEL TBD				Capability: **SubgroupShuffleINTEL** Reserved.	
5	5571	*<id>* *Result Type*	*Result <id>*	*<id>* *Data*	*<id>* *InvocationId*

OpSubgroupShuffleDownINTEL TBD					Capability: **SubgroupShuffleINTEL** Reserved.	
6	5572	*<id>* *Result Type*	*Result <id>*	*<id>* *Current*	*<id>* *Next*	*<id>* *Delta*

OpSubgroupShuffleUpINTEL TBD					Capability: **SubgroupShuffleINTEL** Reserved.	
6	5573	*<id>* *Result Type*	*Result <id>*	*<id>* *Previous*	*<id>* *Current*	*<id>* *Delta*

OpSubgroupShuffleXorINTEL TBD				Capability: **SubgroupShuffleINTEL** Reserved.	
5	5574	*<id>* *Result Type*	*Result <id>*	*<id>* *Data*	*<id>* *Value*

OpSubgroupBlockReadINTEL TBD				Capability: **SubgroupBufferBlockIOINTEL** Reserved.
4	5575	*<id>* *Result Type*	*Result <id>*	*<id>* *Ptr*

OpSubgroupBlockWriteINTEL TBD			Capability: **SubgroupBufferBlockIOINTEL** Reserved.
3	5576	*<id>* *Ptr*	*<id>* *Data*

OpSubgroupImageBlockReadINTEL TBD	Capability: **SubgroupImageBlockIOINTEL** Reserved.

5	5577	<id> *Result Type*	*Result <id>*	<id> *Image*	<id> *Coordinate*

OpSubgroupImageBlockWriteINTEL TBD				Capability: **SubgroupImageBlockIOINTEL** Reserved.
4	5578	<id> *Image*	<id> *Coordinate*	<id> *Data*

OpSubgroupImageMediaBlockReadINTEL TBD						Capability: **SubgroupImageMediaBlockIOINTEL** Reserved.	
7	5580	<id> *Result Type*	*Result <id>*	<id> *Image*	<id> *Coordinate*	<id> *Width*	<id> *Height*

OpSubgroupImageMediaBlockWriteINTEL TBD					Capability: **SubgroupImageMediaBlockIOINTEL** Reserved.	
6	5581	<id> *Image*	<id> *Coordinate*	<id> *Width*	<id> *Height*	<id> *Data*

3.42.22. Device-Side Enqueue Instructions

<table>
<tr><td colspan="7">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 colspan="2">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.

The type of *ND Range* must be an **OpTypeStruct** whose members are as described by the *Result Type* of **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*

<table>
<tr><td colspan="8">OpGetKernelNDrangeSubGroupCount

Result is 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.

The type of ND Range must be an OpTypeStruct whose members are as described by the Result Type of 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.</td><td colspan="2">Capability:
DeviceEnqueue</td></tr>
<tr><td>8</td><td>293</td><td><id>
Result Type</td><td>Result <id></td><td><id>
ND Range</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="7">OpGetKernelNDrangeMaxSubGroupSize

Result is 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.

The type of ND Range must be an OpTypeStruct whose members are as described by the Result Type of 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.</td>
<td colspan="2">Capability:
DeviceEnqueue</td>
</tr>
<tr>
<td>8</td>
<td>294</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
ND Range</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="6">OpGetKernelWorkGroupSize

Result is 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.</td><td colspan="2">Capability:
DeviceEnqueue</td></tr>
<tr><td>7</td><td>295</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>

OpGetKernelPreferredWorkGroupSizeMultiple

Result is 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 this result as the value of the local work size does 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.

Capability:
DeviceEnqueue

7	296	<id> *Result Type*	*Result <id>*	<id> *Invoke*	<id> *Param*	<id> *Param Size*	<id> *Param Align*

OpRetainEvent

Increments the reference count of the event object specified by *Event*.

Behavior is undefined if *Event* is not a valid event.

Capability:
DeviceEnqueue

2	297	<id> *Event*

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.

Behavior is undefined if *Event* is not a valid event.

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 Result is **true** if the event specified by *Event* is a valid event, otherwise **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*

<table>
<tr><td colspan="4">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 is available in the memory pointed to by Value after 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.

If Profiling Info is CmdExecTime, Value behavior is defined only if it points 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: What is captured is undefined if this instruction is called multiple times for the same event.</td><td>Capability:
DeviceEnqueue</td></tr>
<tr><td>4</td><td>302</td><td><id>
Event</td><td><id>
Profiling Info</td><td><id>
Value</td></tr>
</table>

<table>
<tr><td colspan="3">OpGetDefaultQueue

The result is the default device queue, or if a default device queue has not been created, a null queue object.

Result Type must be an OpTypeQueue.</td><td>Capability:
DeviceEnqueue</td></tr>
<tr><td>3</td><td>303</td><td><id>
Result Type</td><td>Result <id></td></tr>
</table>

OpBuildNDRange

Given the global work size specified by *GlobalWorkSize*, local work size specified by *LocalWorkSize* and global work offset specified by *GlobalWorkOffset*, builds the result as a 1D, 2D, or 3D ND-range descriptor structure.

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 if the *addressing model* is **Physical32** and 64-bit *integer type* scalar if 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 if the *addressing model* is **Physical32** and 64-bit *integer type* scalar if 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 execute the kernel function.

4) **OpTypeArray** with 3 elements, where each element is 32-bit *integer type* scalar if the *addressing model* is **Physical32** and 64-bit *integer type* scalar if 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 if the *addressing model* is **Physical32** or 64-bit *integer type* scalar if 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

Result is 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**.
- 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:
SubgroupDispatch

Missing before **version 1.1**.

8	325	<id> *Result Type*	*Result <id>*	<id> *Subgroup Count*	<id> *Invoke*	<id> *Param*	<id> *Param Size*	<id> *Param Align*

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

Result is the maximum number of subgroups that can be used to execute 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.</td>
<td colspan="2">Capability:
SubgroupDispatch

Missing before version 1.1.</td>
</tr>
<tr>
<td>7</td>
<td>326</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>

3.42.23. 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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

Capability:
Pipes

7	275	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

<table>
<tr>
<td colspan="7">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 must be a 32-bit integer type scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit integer type scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless Packet Alignment > 0 and evenly divides Packet Size.</td>
<td colspan="3">Capability:
Pipes</td>
</tr>
<tr>
<td>9</td>
<td>276</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Pipe</td>
<td><id>
Reserve Id</td>
<td><id>
Index</td>
<td><id>
Pointer</td>
<td><id>
Packet Size</td>
<td><id>
Packet Alignment</td>
</tr>
</table>

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

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:
Pipes

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

7	278	<id> *Result Type*	*Result <id>*	<id> *Pipe*	<id> *Num Packets*	<id> *Packet Size*	<id> *Packet Alignment*

OpReserveWritePipePackets

Capability:
Pipes

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

7	279	<id> *Result Type*	*Result <id>*	<id> *Pipe*	<id> *Num Packets*	<id> *Packet Size*	<id> *Packet Alignment*

<table>
<tr><td colspan="4">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 must be a 32-bit integer type scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit integer type scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless Packet Alignment > 0 and evenly divides Packet Size.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>5</td><td>280</td><td><id>
Pipe</td><td><id>
Reserve Id</td><td><id>
Packet Size</td><td><id>
Packet Alignment</td></tr>
</table>

<table>
<tr><td colspan="4">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 must be a 32-bit integer type scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit integer type scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless Packet Alignment > 0 and evenly divides Packet Size.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>5</td><td>281</td><td><id>
Pipe</td><td><id>
Reserve Id</td><td><id>
Packet Size</td><td><id>
Packet Alignment</td></tr>
</table>

<table>
<tr><td>OpIsValidReserveId

Result is 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.</td><td>Capability:
Pipes</td></tr>
</table>

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 result is 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* must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe. *Packet Alignment* must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe. Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.					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 by the creation of *Pipe*. *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* must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe. *Packet Alignment* must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe. Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.					Capability: **Pipes**	
6	284	<id> *Result Type*	*Result <id>*	<id> *Pipe*	<id> *Packet Size*	<id> *Packet Alignment*

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

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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

Result Type must be an **OpTypeReserveId**.

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

</td>
<td colspan="2">

Capability:
Pipes

</td>
</tr>
<tr>
<td>8</td>
<td>285</td>
<td>*<id>*
Result Type</td>
<td>*Result <id>*</td>
<td>*Scope <id>*
Execution</td>
<td>*<id>*
Pipe</td>
<td>*<id>*
Num Packets</td>
<td>*<id>*
Packet Size</td>
<td>*<id>*
Packet Alignment</td>
</tr>
</table>

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.

Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution.

Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction.

Result Type must be an **OpTypeReserveId**.

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

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 must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.

Capability:
Pipes

8	286	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	<id> *Pipe*	<id> *Num Packets*	<id> *Packet Size*	<id> *Packet Alignment*

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

Behavior is undefined if not all invocations of this module within Execution reach this point of execution.

Behavior is undefined unless all invocations within Execution execute the same dynamic instance of this instruction.

Execution is a Scope. It must be either Workgroup or Subgroup.

Pipe must have a type of OpTypePipe with ReadOnly access qualifier.

Reserve Id must have a type of OpTypeReserveId.

Packet Size must be a 32-bit integer type scalar that represents the size in bytes of each packet in the pipe.

Packet Alignment must be a 32-bit integer type scalar that represents the alignment in bytes of each packet in the pipe.

Behavior is undefined unless Packet Alignment > 0 and evenly divides Packet Size.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>6</td><td>287</td><td>Scope <id>
Execution</td><td><id>
Pipe</td><td><id>
Reserve Id</td><td><id>
Packet Size</td><td><id>
Packet Alignment</td></tr>
</table>

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. Behavior is undefined if not all invocations of this module within *Execution* reach this point of execution. Behavior is undefined unless all invocations within *Execution* execute the same dynamic instance of this instruction. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Pipe* must have a type of **OpTypePipe** with **WriteOnly** *access qualifier*. *Reserve Id* must have a type of **OpTypeReserveId**. *Packet Size* must be a 32-bit *integer type* scalar that represents the size in bytes of each packet in the pipe. *Packet Alignment* must be a 32-bit *integer type* scalar that represents the alignment in bytes of each packet in the pipe. Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*.					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* is an unsigned 32-bit integer. It represents the size in bytes of each packet in the pipe. *Packet Alignment* is an unsigned 32-bit integer. It represents the alignment in bytes of each packet in the pipe. Behavior is undefined unless *Packet Alignment* > 0 and evenly divides *Packet Size*. *Capacity* is an unsigned 32-bit integer. It is the minimum number of *Packet Size* blocks the resulting **OpTypePipeStorage** can hold.					Capability: **PipeStorage** Missing before **version 1.1**.	
6	323	<id> *Result Type*	*Result <id>*	*Literal* *Packet Size*	*Literal* *Packet Alignment*	*Literal* *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**. *Qualifier* is the pipe access qualifier.				Capability: **PipeStorage** Missing before **version 1.1**.
4	324	*<id>* *Result Type*	*Result <id>*	*<id>* *Pipe Storage*

OpReadPipeBlockingINTEL TBD				Capability: **BlockingPipesINTEL** Reserved.	
5	5946	*<id>* *Result Type*	*Result <id>*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpWritePipeBlockingINTEL TBD				Capability: **BlockingPipesINTEL** Reserved.	
5	5947	*<id>* *Result Type*	*Result <id>*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

3.42.24. Non-Uniform Instructions

OpGroupNonUniformElect Result is **true** only in the active invocation with the lowest id in the group, otherwise result is false. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**.				Capability: **GroupNonUniform** Missing before **version 1.3**.
4	333	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*

OpGroupNonUniformAll Evaluates a predicate for all active invocations in the group, resulting in **true** if predicate evaluates to **true** for all active invocations in the group, otherwise the result is **false**. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Predicate* must be a *Boolean type*.				Capability: **GroupNonUniformVote** Missing before **version 1.3**.	
5	334	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Predicate*

OpGroupNonUniformAny Evaluates a predicate for all active invocations in the group, resulting in **true** if predicate evaluates to **true** for any active invocation in the group, otherwise the result is **false**. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Predicate* must be a *Boolean type*.				Capability: **GroupNonUniformVote** Missing before **version 1.3**.	
5	335	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Predicate*

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

Evaluates a value for all active invocations in the group. The result is true if Value is equal for all active invocations in the group. Otherwise, the result is false.

Result Type must be a Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

Value must be a scalar or vector of floating-point type, integer type, or Boolean type. The compare operation is based on this type, and if it is a floating-point type, an ordered-and-equal compare is used.</td>
<td colspan="2">Capability:
GroupNonUniformVote

Missing before version 1.3.</td>
</tr>
<tr>
<td>5</td>
<td>336</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td>Scope <id>
Execution</td>
<td><id>
Value</td>
</tr>
</table>

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

Result is the Value of the invocation identified by the id Id to all active invocations in the group.

Result Type must be a scalar or vector of floating-point type, integer type, or Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

The type of Value must be the same as Result Type.

Id must be a scalar of integer type, whose Signedness operand is 0.

Before version 1.5, Id must come from a constant instruction. Starting with version 1.5, this restriction is lifted. However, behavior is undefined when Id is not dynamically uniform.

The resulting value is undefined if Id is an inactive invocation, or is greater than or equal to the size of the group.</td>
<td colspan="2">Capability:
GroupNonUniformBallot

Missing before version 1.3.</td>
</tr>
<tr>
<td>6</td>
<td>337</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td>Scope <id>
Execution</td>
<td><id>
Value</td>
<td><id>
Id</td>
</tr>
</table>

OpGroupNonUniformBroadcastFirst Result is the *Value* of the invocation from the active invocation with the lowest id in the group to all active invocations in the group. *Result Type* must be a scalar or vector of *floating-point type*, *integer type*, or *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The type of *Value* must be the same as *Result Type*.				Capability: **GroupNonUniformBallot** Missing before **version 1.3**.	
5	338	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Value*

OpGroupNonUniformBallot Result is a bitfield value combining the *Predicate* value from all invocations in the group that execute the same dynamic instance of this instruction. The bit is set to one if the corresponding invocation is active and the *Predicate* for that invocation evaluated to true; otherwise, it is set to zero. *Result Type* must be a vector of four components of *integer type* scalar, whose *Signedness* operand is 0. *Result* is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Predicate* must be a *Boolean type*.				Capability: **GroupNonUniformBallot** Missing before **version 1.3**.	
5	339	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Predicate*

OpGroupNonUniformInverseBallot Evaluates a value for all active invocations in the group, resulting in **true** if the bit in *Value* for the corresponding invocation is set to one, otherwise the result is **false**. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Value* must be a vector of four components of *integer type* scalar, whose *Signedness* operand is 0. Behavior is undefined unless *Value* is the same for all invocations that execute the same dynamic instance of this instruction. *Value* is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations.				Capability: **GroupNonUniformBallot** Missing before **version 1.3**.	
5	340	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Value*

OpGroupNonUniformBallotBitExtract Evaluates a value for all active invocations in the group, resulting in **true** if the bit in *Value* that corresponds to *Index* is set to one, otherwise the result is **false**. *Result Type* must be a *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. *Value* must be a vector of four components of *integer type* scalar, whose *Signedness* operand is 0. *Value* is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations. *Index* must be a scalar of *integer type*, whose *Signedness* operand is 0. The resulting value is undefined if *Index* is greater than or equal to the size of the group.					Capability: **GroupNonUniformBallot** Missing before **version 1.3**.	
6	341	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*<id>* *Value*	*<id>* *Index*

<table>
<tr><td colspan="5">**OpGroupNonUniformBallotBitCount**

Result is the number of bits that are set to 1 in *Value*, considering only the bits in *Value* required to represent all bits of the group's invocations.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is 0.

Value must be a vector of four components of *integer type* scalar, whose *Signedness* operand is 0.

Value is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations.</td><td colspan="2">Capability:
GroupNonUniformBallot

Missing before **version 1.3**.</td></tr>
<tr><td>6</td><td>342</td><td>*<id>*
Result Type</td><td>*Result <id>*</td><td>*Scope <id>*
Execution</td><td>*Group Operation*
Operation</td><td>*<id>*
Value</td></tr>
</table>

<table>
<tr><td colspan="4">**OpGroupNonUniformBallotFindLSB**

Find the least significant bit set to 1 in *Value*, considering only the bits in *Value* required to represent all bits of the group's invocations. If none of the considered bits is set to 1, the resulting value is undefined.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

Value must be a vector of four components of *integer type* scalar, whose *Signedness* operand is 0.

Value is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations.</td><td colspan="2">Capability:
GroupNonUniformBallot

Missing before **version 1.3**.</td></tr>
<tr><td>5</td><td>343</td><td>*<id>*
Result Type</td><td>*Result <id>*</td><td>*Scope <id>*
Execution</td><td>*<id>*
Value</td></tr>
</table>

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

Find the most significant bit set to 1 in Value, considering only the bits in Value required to represent all bits of the group's invocations. If none of the considered bits is set to 1, the resulting value is undefined.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

Value must be a vector of four components of integer type scalar, whose Signedness operand is 0.

Value is a set of bitfields where the first invocation is represented in the lowest bit of the first vector component and the last (up to the size of the group) is the higher bit number of the last bitmask needed to represent all bits of the group invocations.</td><td colspan="2">Capability:
GroupNonUniformBallot

Missing before version 1.3.</td></tr>
<tr><td>5</td><td>344</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Value</td></tr>
</table>

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

Result is the Value of the invocation identified by the id Id.

Result Type must be a scalar or vector of floating-point type, integer type, or Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

The type of Value must be the same as Result Type.

Id must be a scalar of integer type, whose Signedness operand is 0.

The resulting value is undefined if Id is an inactive invocation, or is greater than or equal to the size of the group.</td><td colspan="2">Capability:
GroupNonUniformShuffle

Missing before version 1.3.</td></tr>
<tr><td>6</td><td>345</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Value</td><td><id>
Id</td></tr>
</table>

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

Result is the Value of the invocation identified by the current invocation’s id within the group xor’ed with Mask.

Result Type must be a scalar or vector of floating-point type, integer type, or Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

The type of Value must be the same as Result Type.

Mask must be a scalar of integer type, whose Signedness operand is 0.

The resulting value is undefined if current invocation’s id within the group xor’ed with Mask is an inactive invocation, or is greater than or equal to the size of the group.</td>
<td colspan="2">Capability:
GroupNonUniformShuffle

Missing before version 1.3.</td>
</tr>
<tr>
<td>6</td>
<td>346</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td>Scope <id>
Execution</td>
<td><id>
Value</td>
<td><id>
Mask</td>
</tr>
</table>

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

Result is the Value of the invocation identified by the current invocation’s id within the group - Delta.

Result Type must be a scalar or vector of floating-point type, integer type, or Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

The type of Value must be the same as Result Type.

Delta must be a scalar of integer type, whose Signedness operand is 0.

Delta is treated as unsigned and the resulting value is undefined if Delta is greater than the current invocation’s id within the group or if the selected lane is inactive.</td>
<td colspan="2">Capability:
GroupNonUniformShuffleRelative

Missing before version 1.3.</td>
</tr>
<tr>
<td>6</td>
<td>347</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td>Scope <id>
Execution</td>
<td><id>
Value</td>
<td><id>
Delta</td>
</tr>
</table>

OpGroupNonUniformShuffleDown Result is the *Value* of the invocation identified by the current invocation’s id within the group + *Delta*. *Result Type* must be a scalar or vector of *floating-point type*, *integer type*, or *Boolean type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The type of *Value* must be the same as *Result Type*. *Delta* must be a scalar of *integer type*, whose *Signedness* operand is 0. *Delta* is treated as unsigned and the resulting value is undefined if *Delta* is greater than or equal to the size of the group, or if the current invocation’s id within the group + *Delta* is either an inactive invocation or greater than or equal to the size of the group.					Capability: **GroupNonUniformShuffleRelative** Missing before **version 1.3**.	
6	348	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	<id> *Value*	<id> *Delta*

OpGroupNonUniformIAdd An integer add group operation of all *Value* operands contributed by active invocations in the group. *Result Type* must be a scalar or vector of *integer type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is 0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present. The type of *Value* must be the same as *Result Type*. *ClusterSize* is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.						Capability: **GroupNonUniformArithmetic**, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV** Missing before **version 1.3**.	
6 + variable	349	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	<id> *Value*	Optional <id> *ClusterSize*

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

A floating point add group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type. The method used to perform the group operation on the contributed Value(s) from active invocations is implementation defined.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>350</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

An integer multiply group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is 1. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>351</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

A floating point multiply group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is 1. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type. The method used to perform the group operation on the contributed Value(s) from active invocations is implementation defined.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>352</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

A signed integer minimum group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is INT_MAX. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>353</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

<table>
<tr><td colspan="6">**OpGroupNonUniformUMin**

An unsigned integer minimum group operation of all *Value* operands contributed by active invocations in the group.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is UINT_MAX. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present.

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

ClusterSize is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV**

Missing before **version 1.3**.</td></tr>
<tr><td>6 + variable</td><td>354</td><td><id>
Result Type</td><td>*Result <id>*</td><td>*Scope <id>*
Execution</td><td>*Group Operation*
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

<table>
<tr><td colspan="6">**OpGroupNonUniformFMin**

A floating point minimum group operation of all *Value* operands contributed by active invocations in the group.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is +INF. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present.

The type of *Value* must be the same as *Result Type*. The method used to perform the group operation on the contributed *Value*(s) from active invocations is implementation defined. From the set of *Value*(s) provided by active invocations within a subgroup, if for any two *Value*s one of them is a NaN, the other is chosen. If all *Value*(s) that are used by the current invocation are NaN, then the result is an undefined value.

ClusterSize is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV**

Missing before **version 1.3**.</td></tr>
<tr><td>6 + variable</td><td>355</td><td><id>
Result Type</td><td>*Result <id>*</td><td>*Scope <id>*
Execution</td><td>*Group Operation*
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

A signed integer maximum group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is INT_MIN. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>356</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

An unsigned integer maximum group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>357</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

A floating point maximum group operation of all Value operands contributed by active invocations in by group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is -INF. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type. The method used to perform the group operation on the contributed Value(s) from active invocations is implementation defined. From the set of Value(s) provided by active invocations within a subgroup, if for any two Values one of them is a NaN, the other is chosen. If all Value(s) that are used by the current invocation are NaN, then the result is an undefined value.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>358</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

A bitwise and group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is ~0. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>359</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

OpGroupNonUniformBitwiseOr A bitwise or group operation of all *Value* operands contributed by active invocations in the group. *Result Type* must be a scalar or vector of *integer type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is 0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present. The type of *Value* must be the same as *Result Type*. *ClusterSize* is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.						Capability: **GroupNonUniformArithmetic**, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV** Missing before **version 1.3**.	
6 + variable	360	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *Value*	Optional *<id>* *ClusterSize*

OpGroupNonUniformBitwiseXor A bitwise xor group operation of all *Value* operands contributed by active invocations in the group. *Result Type* must be a scalar or vector of *integer type*. *Execution* is a *Scope*. It must be either **Workgroup** or **Subgroup**. The identity *I* for *Operation* is 0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present. The type of *Value* must be the same as *Result Type*. *ClusterSize* is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.						Capability: **GroupNonUniformArithmetic**, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV** Missing before **version 1.3**.	
6 + variable	361	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *Value*	Optional *<id>* *ClusterSize*

<table>
<tr>
<td colspan="6">**OpGroupNonUniformLogicalAnd**

A logical and group operation of all *Value* operands contributed by active invocations in the group.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is ~0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present.

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

ClusterSize is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.</td>
<td colspan="2">Capability:
GroupNonUniformArithmetic, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV**

Missing before **version 1.3**.</td>
</tr>
<tr>
<td>6 + variable</td>
<td>362</td>
<td>*<id>*
Result Type</td>
<td>*Result <id>*</td>
<td>*Scope <id>*
Execution</td>
<td>*Group Operation*
Operation</td>
<td>*<id>*
Value</td>
<td>Optional
<id>
ClusterSize</td>
</tr>
</table>

<table>
<tr>
<td colspan="6">**OpGroupNonUniformLogicalOr**

A logical or group operation of all *Value* operands contributed by active invocations in the group.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is 0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present.

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

ClusterSize is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.</td>
<td colspan="2">Capability:
GroupNonUniformArithmetic, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV**

Missing before **version 1.3**.</td>
</tr>
<tr>
<td>6 + variable</td>
<td>363</td>
<td>*<id>*
Result Type</td>
<td>*Result <id>*</td>
<td>*Scope <id>*
Execution</td>
<td>*Group Operation*
Operation</td>
<td>*<id>*
Value</td>
<td>Optional
<id>
ClusterSize</td>
</tr>
</table>

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

A logical xor group operation of all Value operands contributed by active invocations in the group.

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

Execution is a Scope. It must be either Workgroup or Subgroup.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be present.

The type of Value must be the same as Result Type.

ClusterSize is the size of cluster to use. ClusterSize must be a scalar of integer type, whose Signedness operand is 0. ClusterSize must come from a constant instruction. Behavior is undefined unless ClusterSize is at least 1 and a power of 2. If ClusterSize is greater than the declared SubGroupSize, executing this instruction results in undefined behavior.</td><td colspan="2">Capability:
GroupNonUniformArithmetic, GroupNonUniformClustered, GroupNonUniformPartitionedNV

Missing before version 1.3.</td></tr>
<tr><td>6 + variable</td><td>364</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
Value</td><td>Optional
<id>
ClusterSize</td></tr>
</table>

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

Result is the Value of the invocation within the quad with a quad index equal to Index.

Result Type must be a scalar or vector of floating-point type, integer type, or Boolean type.

Execution is a Scope. It must be either Workgroup or Subgroup.

The type of Value must be the same as Result Type.

Index must be a scalar of integer type, whose Signedness operand is 0.

Before version 1.5, Index must come from a constant instruction. Starting with version 1.5, Index must be dynamically uniform.

If the value of Index is greater than or equal to 4, or refers to an inactive invocation, the resulting value is undefined.</td><td colspan="2">Capability:
GroupNonUniformQuad

Missing before version 1.3.</td></tr>
<tr><td>6</td><td>365</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Value</td><td><id>
Index</td></tr>
</table>

OpGroupNonUniformQuadSwap

Swap the *Value* of the invocation within the quad with another invocation in the quad using *Direction*.

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

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

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

Direction is the kind of swap to perform.

Direction must be a scalar of *integer type*, whose *Signedness* operand is 0.

Direction must come from a *constant instruction*.

The value returned in *Result* is the value provided to *Value* by another invocation in the same quad scope instance. The invocation providing this value is determined according to *Direction*.

A *Direction* of 0 indicates a horizontal swap;
- Invocations with quad indices of 0 and 1 swap values
- Invocations with quad indices of 2 and 3 swap values
A *Direction* of 1 indicates a vertical swap;
- Invocations with quad indices of 0 and 2 swap values
- Invocations with quad indices of 1 and 3 swap values
A *Direction* of 2 indicates a diagonal swap;
- Invocations with quad indices of 0 and 3 swap values
- Invocations with quad indices of 1 and 2 swap values

If an active invocation reads *Value* from an inactive invocation, the resulting value is undefined.

Capability:
GroupNonUniformQuad

Missing before **version 1.3**.

6	366	<id> *Result Type*	*Result <id>*	*Scope <id>* *Execution*	<id> *Value*	<id> *Direction*

OpGroupNonUniformPartitionNV

TBD

Capability:
GroupNonUniformPartitionedNV

Reserved.

4	5296	<id> *Result Type*	*Result <id>*	<id> *Value*

3.42.25. Reserved Instructions

OpTraceRayKHR TBD										Capability: **RayTracingKHR** Reserved.		
12	4445	*<id>* *Accel*	*<id>* *Ray Flags*	*<id>* *Cull Mask*	*<id>* *SBT Offset*	*<id>* *SBT Stride*	*<id>* *Miss Index*	*<id>* *Ray Origin*	*<id>* *Ray Tmin*	*<id>* *Ray Direction*	*<id>* *Ray Tmax*	*<id>* *Payload*

OpExecuteCallableKHR TBD			Capability: **RayTracingKHR** Reserved.
3	4446	*<id>* *SBT Index*	*<id>* *Callable Data*

OpConvertUToAccelerationStructureKHR TBD				Capability: **RayTracingKHR**, **RayQueryKHR** Reserved.
4	4447	*<id>* *Result Type*	*Result <id>*	*<id>* *Accel*

OpIgnoreIntersectionKHR TBD	Capability: **RayTracingKHR** Reserved.
1	4448

OpTerminateRayKHR TBD	Capability: **RayTracingKHR** Reserved.
1	4449

OpTypeRayQueryKHR TBD		Capability: **RayQueryKHR** Reserved.
2	4472	*Result <id>*

OpRayQueryInitializeKHR TBD	Capability: **RayQueryKHR** Reserved.

9	4473	<id> *RayQuery*	<id> *Accel*	<id> *RayFlags*	<id> *CullMask*	<id> *RayOrigin*	<id> *RayTMin*	<id> *RayDirection*	<id> *RayTMax*

OpRayQueryTerminateKHR TBD		Capability: **RayQueryKHR** Reserved.
2	4474	<id> *RayQuery*

OpRayQueryGenerateIntersectionKHR TBD			Capability: **RayQueryKHR** Reserved.
3	4475	<id> *RayQuery*	<id> *HitT*

OpRayQueryConfirmIntersectionKHR TBD		Capability: **RayQueryKHR** Reserved.
2	4476	<id> *RayQuery*

OpRayQueryProceedKHR TBD				Capability: **RayQueryKHR** Reserved.
4	4477	<id> *Result Type*	*Result <id>*	<id> *RayQuery*

OpRayQueryGetIntersectionTypeKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	4479	<id> *Result Type*	*Result <id>*	<id> *RayQuery*	<id> *Intersection*

OpFragmentMaskFetchAMD TBD				Capability: **FragmentMaskAMD** Reserved.	
5	5011	<id> *Result Type*	*Result <id>*	<id> *Image*	<id> *Coordinate*

OpFragmentFetchAMD

TBD

Capability: **FragmentMaskAMD**

Reserved.

6	5012	*<id>* *Result Type*	*Result <id>*	*<id>* *Image*	*<id>* *Coordinate*	*<id>* *Fragment Index*

OpReadClockKHR

TBD

Capability: **ShaderClockKHR**

Reserved.

4	5056	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Scope*

OpWritePackedPrimitiveIndices4x8NV

TBD

Capability: **MeshShadingNV**

Reserved.

3	5299	*<id>* *Index Offset*	*<id>* *Packed Indices*

OpReportIntersectionNV (OpReportIntersectionKHR)

TBD

Capability: **RayTracingNV**, **RayTracingKHR**

Reserved.

5	5334	*<id>* *Result Type*	*Result <id>*	*<id>* *Hit*	*<id>* *HitKind*

OpIgnoreIntersectionNV

TBD

Capability: **RayTracingNV**

Reserved.

1	5335

OpTerminateRayNV

TBD

Capability: **RayTracingNV**

Reserved.

1	5336

OpTraceNV

TBD

Capability: **RayTracingNV**

Reserved.

12	5337	*<id> Accel*	*<id> Ray Flags*	*<id> Cull Mask*	*<id> SBT Offset*	*<id> SBT Stride*	*<id> Miss Index*	*<id> Ray Origin*	*<id> Ray Tmin*	*<id> Ray Direction*	*<id> Ray Tmax*	*<id> PayloadId*

OpTraceMotionNV TBD											Capability: **RayTracingMotionBlurNV** Reserved.	
13	5338	*<id> Accel*	*<id> Ray Flags*	*<id> Cull Mask*	*<id> SBT Offset*	*<id> SBT Stride*	*<id> Miss Index*	*<id> Ray Origin*	*<id> Ray Tmin*	*<id> Ray Direction*	*<id> Ray Tmax*	*<id> Time*	*<id> PayloadId*

OpTraceRayMotionNV TBD											Capability: **RayTracingMotionBlurNV** Reserved.	
13	5339	*<id> Accel*	*<id> Ray Flags*	*<id> Cull Mask*	*<id> SBT Offset*	*<id> SBT Stride*	*<id> Miss Index*	*<id> Ray Origin*	*<id> Ray Tmin*	*<id> Ray Direction*	*<id> Ray Tmax*	*<id> Time*	*<id> Payload*

OpTypeAccelerationStructureNV (OpTypeAccelerationStructureKHR) TBD		Capability: **RayTracingNV**, **RayTracingKHR**, **RayQueryKHR** Reserved.
2	5341	*Result <id>*

OpExecuteCallableNV TBD			Capability: **RayTracingNV** Reserved.
3	5344	*<id> SBT Index*	*<id> Callable DataId*

OpTypeCooperativeMatrixNV TBD					Capability: **CooperativeMatrixNV** Reserved.	
6	5358	*Result <id>*	*<id> Component Type*	*Scope <id> Execution*	*<id> Rows*	*<id> Columns*

OpCooperativeMatrixLoadNV TBD	Capability: **CooperativeMatrixNV** Reserved.

6 + variable	5359	*<id>* *Result Type*	*Result <id>*	*<id>* *Pointer*	*<id>* *Stride*	*<id>* *Column Major*	Optional *Memory Operands*

OpCooperativeMatrixStoreNV TBD					Capability: **CooperativeMatrixNV** Reserved.	
5 + variable	5360	*<id>* *Pointer*	*<id>* *Object*	*<id>* *Stride*	*<id>* *Column Major*	Optional *Memory Operands*

OpCooperativeMatrixMulAddNV TBD					Capability: **CooperativeMatrixNV** Reserved.	
6	5361	*<id>* *Result Type*	*Result <id>*	*<id>* *A*	*<id>* *B*	*<id>* *C*

OpCooperativeMatrixLengthNV TBD				Capability: **CooperativeMatrixNV** Reserved.
4	5362	*<id>* *Result Type*	*Result <id>*	*<id>* *Type*

OpBeginInvocationInterlockEXT TBD	Capability: **FragmentShaderSampleInterlockEXT**, **FragmentShaderPixelInterlockEXT**, **FragmentShaderShadingRateInterlockEXT** Reserved.
1	5364

OpEndInvocationInterlockEXT TBD	Capability: **FragmentShaderSampleInterlockEXT**, **FragmentShaderPixelInterlockEXT**, **FragmentShaderShadingRateInterlockEXT** Reserved.
1	5365

OpIsHelperInvocationEXT TBD			Capability: **DemoteToHelperInvocationEXT** Reserved.
3	5381	*<id>* *Result Type*	*Result <id>*

OpConvertUToImageNV TBD				Capability: **BindlessTextureNV** Reserved.
4	5391	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpConvertUToSamplerNV TBD				Capability: **BindlessTextureNV** Reserved.
4	5392	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpConvertImageToUNV TBD				Capability: **BindlessTextureNV** Reserved.
4	5393	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpConvertSamplerToUNV TBD				Capability: **BindlessTextureNV** Reserved.
4	5394	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpConvertUToSampledImageNV TBD				Capability: **BindlessTextureNV** Reserved.
4	5395	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpConvertSampledImageToUNV TBD	Capability: **BindlessTextureNV** Reserved.

4	5396	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpSamplerImageAddressingModeNV TBD		Capability: **BindlessTextureNV** Reserved.
2	5397	*Literal* *Bit Width*

OpUCountLeadingZerosINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.
4	5585	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpUCountTrailingZerosINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.
4	5586	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand*

OpAbsISubINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5587	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpAbsUSubINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5588	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpIAddSatINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5589	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUAddSatINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5590	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpIAverageINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5591	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUAverageINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5592	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpIAverageRoundedINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5593	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUAverageRoundedINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5594	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpISubSatINTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5595	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUSubSatINTEL TBD	Capability: **IntegerFunctions2INTEL** Reserved.

5	5596	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpIMul32x16INTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5597	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUMul32x16INTEL TBD				Capability: **IntegerFunctions2INTEL** Reserved.	
5	5598	*<id>* *Result Type*	*Result <id>*	*<id>* *Operand 1*	*<id>* *Operand 2*

OpLoopControlINTEL TBD		Capability: **UnstructuredLoopControlsINTEL** Reserved.
1 + variable	5887	*Literal, Literal, …* *Loop Control Parameters*

OpFPGARegINTEL TBD				Capability: **FPGARegINTEL** Reserved.	
5	5949	*<id>* *Result Type*	*Result <id>*	*<id>* *Result*	*<id>* *Input*

OpRayQueryGetRayTMinKHR TBD				Capability: **RayQueryKHR** Reserved.
4	6016	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*

OpRayQueryGetRayFlagsKHR TBD				Capability: **RayQueryKHR** Reserved.
4	6017	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*

OpRayQueryGetIntersectionTKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6018	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionInstanceCustomIndexKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6019	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionInstanceIdKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6020	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionInstanceShaderBindingTableRecordOffsetKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6021	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionGeometryIndexKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6022	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionPrimitiveIndexKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6023	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionBarycentricsKHR TBD	Capability: **RayQueryKHR** Reserved.

5	6024	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionFrontFaceKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6025	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionCandidateAABBOpaqueKHR TBD				Capability: **RayQueryKHR** Reserved.
4	6026	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*

OpRayQueryGetIntersectionObjectRayDirectionKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6027	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionObjectRayOriginKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6028	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetWorldRayDirectionKHR TBD				Capability: **RayQueryKHR** Reserved.
4	6029	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*

OpRayQueryGetWorldRayOriginKHR TBD				Capability: **RayQueryKHR** Reserved.
4	6030	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*

OpRayQueryGetIntersectionObjectToWorldKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6031	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

OpRayQueryGetIntersectionWorldToObjectKHR TBD				Capability: **RayQueryKHR** Reserved.	
5	6032	*<id>* *Result Type*	*Result <id>*	*<id>* *RayQuery*	*<id>* *Intersection*

Chapter 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 Operand (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.
  - 14862 Removed the **IndependentForwardProgress** Execution Mode.
- 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, Revision 7

- Khronos SPIR-V issue #69: **OpImageSparseFetch** editorial change in summary: include that it is sampled image.
- Khronos SPIR-V issue #74: **OpImageQueryLod** requires a sampler.
- Khronos SPIR-V issue #82: Clarification to the **Float16Buffer** Capability.
- Khronos SPIR-V issue #89: Editorial improvements to **OpMemberDecorate** and **OpDecorationGroup**.

4.10. Changes from Version 1.00, Revision 8

- Add SPV_KHR_subgroup_vote tokens.
- Typo: Change "without a sampler" to "with a sampler" for the description of the **SampledBuffer** Capability.
- Khronos SPIR-V issue #61: Clarification of packet size and alignment on all instructions that use the **Pipes** Capability.
- Khronos SPIR-V issue #99: Use "invalid" language to replace any "compile-time error" language.
- Khronos SPIR-V issue #55: Distinguish between branch instructions and termination instructions.
- Khronos SPIR-V issue #94: Add missing **OpSubgroupReadInvocationKHR** enumerant.
- Khronos SPIR-V issue #114: Header blocks strictly dominate their merge blocks.
- Khronos SPIR-V issue #119: **OpSpecConstantOp** allows **OpUndef** where allowed by its *opcode*.

4.11. Changes from Version 1.00, Revision 9

- Khronos Vulkan issue #652: Remove statements about matrix offsets and padding. These are described correctly in the Vulkan API specifications.
- Khronos SPIR-V issue #113: Remove the "By Default" statements in FP Rounding Mode. These should be properly specified by the client API.
- Add extension enumerants for
  - SPV_KHR_16bit_storage
  - SPV_KHR_device_group
  - SPV_KHR_multiview
  - SPV_NV_sample_mask_override_coverage
  - SPV_NV_geometry_shader_passthrough
  - SPV_NV_viewport_array2
  - SPV_NV_stereo_view_rendering
  - SPV_NVX_multiview_per_view_attributes

4.12. Changes from Version 1.00, Revision 10

- Add **HLSL** **source language**.

- Add **StorageBuffer** **storage class**.
- Add **StorageBuffer16BitAccess**, **UniformAndStorageBuffer16BitAccess**, **VariablePointersStorageBuffer**, and **VariablePointers** capabilities.
- Khronos SPIR-V issue #163: Be more clear that **OpTypeStruct** allows zero members. Also affects **ArrayStride** and **Offset** decoration validation rules.
- Khronos SPIR-V issue #159: List allowed **AtomicCounter** instructions with the **AtomicStorage** capability rather than the validation rules.
- Khronos SPIR-V issue #36: Describe more clearly the type of *ND Range* in **OpGetKernelNDrangeSubGroupCount**, **OpGetKernelNDrangeMaxSubGroupSize**, and **OpEnqueueKernel**.
- Khronos SPIR-V issue #128: Be clear the **OpDot** operates only on vectors.
- Khronos SPIR-V issue #80: Loop headers must dominate their continue target. See Structured Control Flow.
- Khronos SPIR-V issue #150 allow **UniformConstant** storage-class variables to have initializers, depending on the client API.

4.13. Changes from Version 1.00, Revision 11

- Public issue #2: Disallow the **Cube** dimension from use with the **Offset**, **ConstOffset**, and **ConstOffset** image operands.
- Public issue #48: **OpConvertPtrToU** only returns a scalar, not a vector.
- Khronos SPIR-V issue #130: Be more clear which masks are literal and which are not.
- Khronos SPIR-V issue #154: Clarify only one of the listed **Capabilities** needs to be declared to use a feature that lists multiple capabilities. The non-declared capabilities need not be supported by the underlying implementation.
- Khronos SPIR-V issue #174: **OpImageDrefGather** and **OpImageSparseDrefGather** return vectors, not scalars.
- Khronos SPIR-V issue #182: The **SampleMask** built in does not depend on **SampleRateShading**, only **Shader**.
- Khronos SPIR-V issue #183: **OpQuantizeToF16** with too-small magnitude can result in either +0 or -0.
- Khronos SPIR-V issue #203: **OpImageTexelPointer** has 3 components for cube arrays, not 4.
- Khronos SPIR-V issue #217: Clearer language for **OpArrayLength**.
- Khronos SPIR-V issue #213: Image Operand **LoD** is not used by query operations.
- Khronos SPIR-V issue #223: **OpPhi** has exactly one parent operand per parent block.
- Khronos SPIR-V issue #212: In the Validation Rules, make clear a pointer can be an operand in an extended instruction set.
- Add extension enumerants for
  - SPV_AMD_shader_ballot
  - SPV_KHR_post_depth_coverage
  - SPV_AMD_shader_explicit_vertex_parameter
  - SPV_EXT_shader_stencil_export
  - SPV_INTEL_subgroups

4.14. 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

4.15. 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).

4.16. 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).

4.17. Changes from Version 1.1, Revision 3

- Incorporated bug fixes from Revision 8 of Version 1.00 (see section 4.9. Changes from Version 1.00, Revision 7).

4.18. Changes from Version 1.1, Revision 4

- Incorporated bug fixes from Revision 9 of Version 1.00 (see section 4.10. Changes from Version 1.00, Revision 8).

4.19. Changes from Version 1.1, Revision 5

- Incorporated changes from Revision 10 of Version 1.00 (see section 4.11. Changes from Version 1.00, Revision 9).

4.20. Changes from Version 1.1, Revision 6

- Incorporated changes from Revision 11 of Version 1.00 (see section 4.12. Changes from Version 1.00, Revision 10).

4.21. Changes from Version 1.1, Revision 7

- Incorporated changes from Revision 12 of Version 1.00 (see section 4.13. Changes from Version 1.00, Revision 11).
- State where all **OpModuleProcessed** belong, in the logical layout.

4.22. Changes from Version 1.1

- Moved version number to SPIR-V 1.2
- New functionality:
  - Added **OpExecutionModeId** to allow using an *<id>* to set the execution modes **SubgroupsPerWorkgroupId**, **LocalSizeId**, and **LocalSizeHintId**.
  - Added **OpDecorateId** to allow using an *<id>* to set the decorations **AlignmentId** and **MaxByteOffsetId**.

4.23. Changes from Version 1.2, Revision 1

- Incorporated changes from Revision 12 of Version 1.00 (see section 4.13. Changes from Version 1.00, Revision 11).
- Incorporated changes from Revision 8 of Version 1.1 (see section 4.21. Changes from Version 1.1, Revision 7).

4.24. Changes from Version 1.2, Revision 2

- Combine the 1.0, 1.1, and 1.2 specifications, making a unified specification. The previous 1.0, 1.1, and 1.2 specifications are replaced with this one unified specification.

4.25. Changes from Version 1.2, Revision 3

Fixed Khronos-internal issues:

- #249: Improve description of **OpTranspose**.
- #251: Undefined values in **OpUndef** include abstract and opaque values.

- #258: Deprecate **OpAtomicCompareExchangeWeak** in favor of **OpAtomicCompareExchange**.
- #241: Use "invalid" instead of "compile-time" error for **ConstOffsets**.
- #248: **OpImageSparseRead** is not for **SubpassData**.
- #257: Allow **OpImageSparseFetch** and **OpImageSparseRead** with the **Sample** image operands.
- #229: Some sensible constraints on branch hints for **OpBranchConditional**.
- #236: **OpVariable**'s storage class must match storage class of the pointer type.
- #216: Can decorate pointer types with **Coherent** and **Volatile**.
- #247: Don’t say Scope *<id>* is a mask; it is not.
- #254: Remove validation rules about the types atomic instructions can operate on. These rules belong instead to the client API.
- #265: **OpGroupDecorate** cannot target an **OpDecorationGroup**.

4.26. Changes from Version 1.2

- Moved version number to SPIR-V 1.3
- New functionality:
  - Added subgroup operations:
    - the **OpGroupNonUniform** instructions and capabilities.
    - **Subgroup**-mask built-in decorations.
  - Khronos SPIR-V issue #125, #138, #196: Removed capabilities from the rounding modes.
  - Khronos SPIR-V issue #110: Removed the execution-model restrictions from **OpControlBarrier**.
- Incorporated the following extensions:
  - SPV_KHR_shader_draw_parameters
  - SPV_KHR_16bit_storage
  - SPV_KHR_device_group
  - SPV_KHR_multiview
  - SPV_KHR_storage_buffer_storage_class
  - SPV_KHR_variable_pointers
- Reserved symbols for
  - SPV_GOOGLE_decorate_string
  - SPV_GOOGLE_hlsl_functionality1
  - SPV_AMD_gpu_shader_half_float_fetch
- Added deprecation model.

4.27. Changes from Version 1.3, Revision 1

- Fixed Issues:
  - Public SPIRV-Headers PR #73: Add missing fields for some NVIDIA-specific tokens.
  - Khronos SPIR-V Issue #202: Shader Validation: Be clear that arrays of blocks set by the client API cannot have an **ArrayStride**.

- Khronos SPIR-V Issue #210: Clarify the *Result Type* of **OpSampledImage**.
- Khronos SPIR-V Issue #211: State that Derivative instructions only work on 32-bit width components.
- Khronos SPIR-V Issue #239: Clarify **OpImageFetch** is for an image whose *Sampled* operand is 1.
- Khronos SPIR-V Issue #256: **OpAtomicCompareExchange** does not store if comparison fails.
- Khronos SPIR-V Issue #269: Be more clear which bits are mutually exclusive for memory semantics.
- Khronos SPIR-V Issue #278: Delete **OpTypeRuntimeArray** restriction on storage classes, as this is already covered by the client API.
- Khronos SPIR-V Issue #279:
  - Add section expository section 2.8.1 "Unsigned Versus Signed Integers".
  - As expected, **OpUConvert** can have vector *Result Type*.
- Khronos SPIR-V Issue #280: **OpImageQuerySizeLod** and **OpImageQueryLevels** can be limited by the client API.
- Khronos SPIR-V Issue #285: Remove **Kernel** as a capability implicitly declared by **Int8**.
- Khronos SPIR-V Issue #290: Clarify implicit declaration of capabilities, in part by changing the column heading to *Implicitly Declares".
- Khronos SPIR-V Issues #295: Explicitly say blocks cannot be nested in blocks, in the validation section. (This was already indirectly required.)
- Khronos SPIR-V Issue #299: Add the **ImageGatherExtended** capability to **ConstOffsets** in the image operands section.
- Khronos SPIR-V Issues #303 and #304: **OpGroupNonUniformBallotBitExtract** documentation: add **Result Type** and fix **Index** parameter.
- Khronos SPIR-V Issue #310: Remove instruction word count from the Limits table, as it is already intrinsically limited.
- Khronos SPIR-V Issue #313: Move the **FPRoundingMode**-decoration validation rule to the shader validation section (not a universal rule). Also, include the **StorageBuffer** storage class in this rule.

4.28. Changes from Version 1.3, Revision 2

- New enumerants:
  - For SPV_KHR_8bit_storage
- Fixed Issues:
  - Add definition of Memory Object Declaration.
  - Khronos SPIR-V Issue #275: Clarify the meaning of **Aliased** and **Restrict** in the Aliasing section.
  - Khronos SPIR-V Issue #315: Be more specific about where many decorations are allowed, particularly for **OpFunctionParameter**. Includes being clear that the **BuiltIn** decoration does not apply to **OpFunctionParamater**.
  - Khronos SPIR-V Issue #348: Clarify *remainder* descriptions in **OpFRem**, **OpFMod**, **OpSRem**, and **OpSMod**.
  - Khronos SPIR-V Issue #342: State the **DepthReplacing** execution-mode behavior more specifically.
  - Khronos SPIR-V Issue #341: More specific wording for depth-hint execution modes **DepthGreater**, **DepthLess**, and **DepthUnchanged**.

- Khronos SPIR-V Issues #276 and #311: Take more care with unreachable blocks in structured control flow and how to branch into a construct.
- Khronos SPIR-V Issue #320: Include **OpExecutionModeId** in the logical layout.
- Khronos SPIR-V Issue #238: Fix description of **OpImageQuerySize** to correct *Sampled Type* ¬ *Sampled* and list the correct set of dimensions.
- Khronos SPIR-V Issue #346: Remove ordered rule for structures in the memory layout: Vulkan allows out-of-order **Offset** layouts.
- Khronos SPIR-V Issue #322: Allow **OpImageQuerySize** to query the size of a **NonReadable** image.
- Khronos SPIR-V Issue #244: Be more clear about the connections between dimensionalities and capabilities, and in referring to them from **OpImageRead** and **OpImageWrite**.
- Khronos SPIR-V Issue #333: Be clear about overflow behavior for **OpIAdd**, **OpISub**, and **OpIMul**.

4.29. Changes from Version 1.3, Revision 3

- Add enumerants for
  - SPV_KHR_vulkan_memory_model
- Fixed Issues:
  - Typo: say **OpMatrixTimesVector** is **Matrix** X **Vector**.
  - Update on Khronos SPIR-V issue #244: Added **Shader** and **Kernel** capabilities to the **2D** dimensionality.
  - Khronos SPIR-V Issue #317: Clarify that the **Uniform** decoration should apply only to objects, and that the dynamic instance of the object is the same, rather than at the consumer usage.
  - Khronos SPIR-V Issue #335: Clarify and correct when it is valid for pointers to be operands to **OpFunctionCall**. Corrections are believed to be consistent with existing front-end and back-end support.
  - Khronos SPIR-V Issue #344: don’t include inactive invocations in what makes the result of **OpGroupNonUniformBallotBitExtract** undefined.

4.30. Changes from Version 1.3, Revision 4

- Add enumerants for
  - SPV_NV_fragment_shader_barycentric
  - SPV_NV_compute_shader_derivatives
  - SPV_NV_shader_image_footprint
  - SPV_NV_shading_rate
  - SPV_NV_mesh_shader
  - SPV_NVX_Raytracing
- Formatting: Removed **Enabling Extensions** column and instead list the extensions in the **Enabling Capabilities** column.

4.31. Changes from Version 1.3, Revision 5

- Reserve Tokens for:

- SPV_KHR_no_integer_wrap_decoration
- SPV_KHR_float_controls

• Fixed Issues:
  - Khronos SPIR-V Issue #352: Remove from **OpFunction** the statement limiting the use its result. This does not result in any change in intent; it only avoids any past and potential future contradictions.
  - Khronos SPIR-V Issue #308: Don't allow runtime-sized arrays to be loaded or copied by **OpLoad** or **OpCopyMemory**.
  - Include back-edge blocks in the list of blocks that can branch outside their own construct in the structured control-flow rules.
  - Khronos OpenGL API issue #77: Clarify the **OriginUpperLeft** and **OriginLowerLeft** execution modes apply only to **FragCoord**.
  - State the **XfbStride** and **Stream** restrictions in the Universal Validation Rules.
  - Khronos SPIR-V Issue #357: The *Memory Operands* of **OpCopyMemory** and **OpCopyMemorySized** applies to both *Source* and *Target*.
  - Khronos SPIR-V Issue #385: Be more clear what type *<id>* must be the same in **OpCopyMemory**.
  - Khronos SPIR-V Issue #359: **OpAccessChain** and **OpPtrAccessChain** do indexing with signed indexes, and **OpPtrAccessChain** is allowed to compute addresses of elements one past the end of an array.
  - Khronos SPIR-V Issue #367: General validation rules allow the **Function** storage class for atomic access, while the shader-specific validation rules do not.
  - Khronos SPIR-V Issue #382: In **OpTypeFunction**, disallow parameter types from being **OpTypeVoid**.
  - Khronos SPIR-V Issue #374: Built-in decorations can also apply to a constant instruction.

• Editorial:
  - Make it more clear in **OpVariable** what *Storage Classes* must be the same.
  - Remove references to specific APIs, and instead generally refer only to "client API"s. Note that the previous lists of APIs was nonnormative.
  - State the **FPRoundingMode** decoration rule more clearly in the section listing Validation Rules for Shader Capabilities.
  - Don't say "value preserving" in the Conversion instructions. These now convert the "value numerically".
  - State variable-pointer validation rules more clearly.

4.32. Changes from Version 1.3, Revision 6

• Reserve Tokens for:
  - SPV_INTEL_media_block_io
  - SPV_NV_cooperative_matrix
  - SPV_INTEL_device_side_avc_motion_estimation, partially. See the SPV_INTEL_device_side_avc_motion_estimation extension specification for a full listing of tokens.

• Fixed Issues:
  - Khronos SPIR-V Issue #406: Scope values must come from the table of scope values.

- Khronos SPIR-V Issue #419: Validation rules include **AtomicCounter** in the list of storage classes allowed for pointer operands to an **OpFunctionCall**.
- Khronos SPIR-V Issue #325: **OpPhi** clarifications regarding parent dominance, in the instruction and the validation rules, and forward references in the Logical Layout section.
- Khronos SPIR-V Issue #415: Remove the non-writable storage classes **PushConstant** and **Input** from the **FPRoundingMode** decoration shader validation rule.
- Khronos SPIR-V Issue #404: Clarify when **OpGroupNonUniformShuffleXor**, **OpGroupNonUniformShuffleUp**, and **OpGroupNonUniformShuffleDown** are valid or result in undefined values.
- Khronos SPIR-V Issue #393: Be more clear that **OpConvertUToPtr** and **OpConvertPtrToU** operate only on unsigned scalar integers.
- Khronos SPIR-V Issue #416: Result are undefined for all Shift instructions for shifts amounts equal to the bit width of the operand.
- Khronos SPIR-V Issue #399: Refine the definition of a variable pointer, particularly for function parameters receiving a variable pointer.
- Khronos SPIR-V Issue #441: Clarify that atomic instruction's *Scope <id>* must be a valid memory scope. More generally, all *Scope <id>* operands are now either *Memory* or *Execution*.
- Khronos SPIR-V Issue #426: Be more direct about undefined behavior for non-uniform control flow in **OpControlBarrier** and the **OpGroup…** instructions that discuss this.

• Deprecate

- Khronos SPIR-V Issue #429: Deprecate **OpDecorationGroup**, **OpGroupDecorate**, and **OpGroupMemberDecorate**

• Editorial

- Add more clarity that the full client API describes the execution environment (there is not a separate specification from the client API specification).

4.33. Changes from Version 1.3, Revision 7

• Fixed Issues:

- Khronos SPIR-V Issue #371: Restrict *intermediate object* types to variable types allowed at global scope. See shader validation data rules.
- Khronos SPIR-V Issue #408: (Re)allow the decorations **Volatile**, **Coherent**, **NonWritable**, and **NonReadable** on members of blocks. (Temporarily dropping this functionality was accidental/clerical; intent is that it has always been present.)
- Khronos SPIR-V Issue #418: Add statements about undefinedness and how NaNs are mixed to **OpGroupNonUniformFAdd**, **OpGroupNonUniformFMul**, **OpGroupNonUniformFMin**, and **OpGroupNonUniformFMax**.
- Khronos SPIR-V Issue #435: Expand the universal validation rule for variable pointers and matrices to also disallow pointing within a matrix.
- Khronos SPIR-V Issue #447: Remove implication that **OpPtrAccessChain** obeys an **ArrayStride** decoration in storage classes laid out by the implementation.
- Khronos SPIR-V Issue #450: Allow pointers to **OpFunctionCall** to be pointers to an element of an array of samplers or images. See the universal validation rules under the **Logical** addressing model without variable pointers.
- Khronos SPIR-V Issue #452: **OpGroupNonUniformAllEqual** uses ordered compares for floating-point values.

- Khronos SPIR-V Issue #454: Add **OpExecutionModeId** to the list of allowed forward references in the Logical Layout of a Module.

4.34. Changes from Version 1.3

- New Functionality:
  - Public issue #35: **OpEntryPoint** must list all global variables in the interface. Additionally, duplication in the list is not allowed.
  - Khronos SPIR-V Issue #140: Generalize **OpSelect** to select between two objects.
  - Khronos SPIR-V Issue #156: Add **OpUConvert** to the list of required opcodes in **OpSpecConstantOp**.
  - Khronos SPIR-V Issue #345: Generalize the **NonWritable** decoration to include **Private** and **Function** storage classes. This helps identify lookup tables.
  - Khronos SPIR-V Issue #84: Add **OpCopyLogical** to copy similar but unequal types.
  - Khronos SPIR-V Issue #170: Add **OpPtrEqual** and **OpPtrNotEqual** to compare pointers.
  - Khronos SPIR-V Issue #362: Add **OpPtrDiff** to count the number of elements between two element pointers.
  - Khronos SPIR-V Issue #332: Add **SignExtend** and **ZeroExtend** image operands.
  - Khronos SPIR-V Issue #340: Add the **UniformId** decoration, which takes a *Scope* operand.
  - Khronos SPIR-V Issue #112: Add iteration-control loop controls.
  - Khronos SPIR-V Issue #366: Change *Memory Access* operands and the **Memory Access** section to now be *Memory Operands* and the **Memory Operands** section.
  - Khronos SPIR-V Issue #357: Allow **OpCopyMemory** and **OpCopyMemorySized** to have *Memory Operands* for both their *Source* and *Target*.
- New Extensions Incorporated into SPIR-V 1.4:
  - SPV_KHR_no_integer_wrap_decoration. See **NoSignedWrap** and **NoUnsignedWrap** decorations and universal validation decoration rules.
  - SPV_GOOGLE_decorate_string. See **OpDecorateString** and **OpMemberDecorateString**.
  - SPV_GOOGLE_hlsl_functionality1. See **CounterBuffer** and **UserSemantic** decorations.
  - SPV_KHR_float_controls. See **DenormPreserve**, **DenormFlushToZero**, **SignedZeroInfNanPreserve**, **RoundingModeRTE**, and **RoundingModeRTZ** execution modes and capabilities.
- Removed:
  - Khronos SPIR-V Issue #437: Removed **OpAtomicCompareExchangeWeak**, and the **BufferBlock** decoration.

4.35. Changes from Version 1.4, Revision 1

- GitHub SPIRV-Registry Issue #25: Remove validation rule for simultaneous use of **RowMajor** and **ColMajor**, instead stating this in the decoration cells themselves.
- Khronos Issue #319: Bring in fixes to the SPV_KHR_16bit_storage extension. See the **StorageBuffer16BitAccess** and the related 16-bit capabilities.
- Khronos Issue #363: **OpTypeBool** can be used in the Input and Output storage classes, but the client APIs still only allow built-in Boolean variables (e.g. FrontFacing), not user variables.

- Khronos Issue #432: Remove the untrue expository statement "**OpFunction** is the only valid use of **OpTypeFunction**."
- Khronos Issue #465: Distinguish between the **Groups** capability and the Group and Subgroup instructions.
- Khronos Issue #484: Have **OpTypeArray** and **OpTypeStruct** point to their definitions.
- Khronos Issue #477: Include 0.0 in the range of required values for **RelaxedPrecision** and other minor clarifications in the relaxed-precision section regarding floating-point precision.
- Khronos Issue #226: Be more clear about explicit level-of-detail being either **Lod** or **Grad** throughout the sampling instructions, and that **ConstOffset**, **Offset**, and **ConstOffsets** are mutually exclusive in the image operand's descriptions.
- Khronos Issue #390: The **Volatile** decoration does not guarantee each invocation performs the access.
- Reserved New Tokens for:
  - SPV_EXT_fragment_shader_interlock
  - SPV_NV_shader_sm_builtins
  - SPV_INTEL_shader_integer_functions2
  - SPV_EXT_demote_to_helper_invocation
  - SPV_KHR_shader_clock
  - SPV_GOOGLE_user_type
  - **Volatile**, for SPV_KHR_vulkan_memory_model

4.36. Changes from Version 1.4

- Extensions Incorporated into SPIR-V 1.5:
  - SPV_KHR_8bit_storage
  - SPV_EXT_descriptor_indexing
  - SPV_EXT_shader_viewport_index_layer, with changes: Replaced the single **ShaderViewportIndexLayerEXT** capability with the two new capabilities **ShaderViewportIndex** and **ShaderLayer**. Declaring both is equivalent to declaring **ShaderViewportIndexLayerEXT**.
  - SPV_EXT_physical_storage_buffer and SPV_KHR_physical_storage_buffer
  - SPV_KHR_vulkan_memory_model
- Khronos Issue #402: Relax **OpGroupNonUniformBroadcast** *Id* from constant to dynamically uniform, starting with version 1.5.
- Khronos Issue #493: Relax **OpGroupNonUniformQuadBroadcast** *Id* from constant to dynamically uniform, starting with version 1.5.
- Khronos Issue #494: Update the *Dynamically Uniform* definition to say that the invocation group is the set of invocations, *unless otherwise stated*.
- Khronos Issue #485: When RelaxedPrecision is applied to a numerical instruction, the operands may be truncated.

4.37. Changes from Version 1.5, Revision 1

- Khronos Issue #511: Allow non-execution non-memory scopes in the introduction to the Scope *<id>* section .

- Khronos MR !147: Fix **OpFNegate** so it handles 0.0f properly
- Khronos Issue #502: **OpAccessChain** array indexes must be an in-bounds for logical pointer types.
- Khronos Issue #518: Include both **VariablePointers** and **VariablePointersStorageBuffer** capabilities in the validation rules when discussing variable pointer rules.
- Khronos Issue #496: Allow **Invariant** to decorate a block member.
- Khronos Issue #469: Disallow **OpConstantNull** result and **OpPtrEqual**, **OpPtrNotEqual**, and **OpPtrDiff** operands from being pointers into the **PhysicalStorageBuffer** storage class. See the **PhysicalStorageBuffer** validation rules.
- Khronos Issue #425: Clarify what variables can allocate pointers, in the validation rules, based on the declarations of the **VariablePointers** or **VariablePointersStorageBuffer** capabilities.
- Khronos Issue #442: Add a note pointing out where signedness has some semantic meaning.
- Khronos Issue #498: Relaxed the set of allowed types for some Group and Subgroup instructions.
- Khronos Issue #500: Deprecate **OpLessOrGreater** in favor of **OpFOrdNotEqual**.
- Khronos Issue #354: Rationalize literals throughout the specification. Remove "immediate" as a separate definition. Be more rigid about a single literal mapping to one or more operands, and that the instruction description defines the type of the literal.
- Khronos Issue #479: Disallow intermediate aggregate types that could not be used to declare global variables, and disallow all types that can't be used for declaring variables. See the shader validation "Type Rules". Also, more strongly state that intermediate values don't form a storage class, in the introduction to storage classes.
- Khronos Issue #78: Use a more correct definition of *back edge*.
- Khronos Issue #492: Overflow with **OpSDiv**, **OpSRem**, and **OpSMod** results in undefined behavior.

4.38. Changes from Version 1.5, Revision 2

- Reserve enumerants for SPV_KHR_ray_query and SPV_KHR_ray_tracing.
- Khronos MR #164: Subtract all exits from what a construct contains, not just the construct's merge block. See the Structured Control Flow section.
- Khronos Issues #394 and #473: More clearly state that the <*id*> declared by an **OpTypeForwardPointer** can be consumed by any type-declaration instruction that can legally consume the type of <*id*>. Also consolidated the rules for this within the instruction itself.
- Khronos Vulkan Issue #1951: Clarify that the **SampledImageArrayDynamicIndexing** capability applies to dynamic indexing of image, sampler and sampled image objects.
- Khronos Issue #523: Label as memory Scope the additional operand for each of
  - **MakeTexelAvailable** and **MakeTexelVisible** image operands, and
  - **MakePointerAvailable** and **MakePointerVisible** memory operands.
- Khronos Issue #529: Allow the scope of uniform control flow to be defined by the client API.
- Khronos Issue #530: Allow the definition of derivative group to be set by the client API.
- Khronos Issue #293: Editorial simplification and clarification of different types under Types and Variables.
- Khronos Issue #506: Add to the definition of **Pure** under Function Control that assuming it computes the same results also requires the same global state.
- Khronos Issue #539: Clarify out-of-bounds indexes for **OpAccessChain**.
- Khronos Issue #550: Include **OpUndef** in the allowed constituents for **OpSpecConstantComposite**.

- Khronos Issue #389: Be more clear which instructions can be updated with a specialization constant in the specialization section.
- Khronos Issue #544: Be more concise with **OpLabel** language.
- Khronos Issue #245: State that D_{ref} operands must be 32-bit scalar floats in the image instructions.
- Khronos Issue #457: Change rule for **OpUnreachable** to being that behavior is undefined if it is executed.
- Khronos Issue #231: Explicitly state that the component numbers 0, 1, 2, and 3 are 32-bit scalar integers for **OpImageGather** and **OpImageSparseGather**.
- Khronos Issue #534: State where **OpNoLine** can be in the logical layout and with **OpPhi**.
- Khronos MR #168: Add definitions of quad and quad index, used by **OpGroupNonUniformQuadBroadcast** and **OpGroupNonUniformQuadSwap**.

4.39. Changes from Version 1.5, Revision 3

- Reserve enumerants for the extensions
  - SPV_INTEL_fpga_loop_controls
  - SPV_INTEL_blocking_pipes
  - SPV_INTEL_unstructured_loop_controls
  - SPV_INTEL_fpga_reg
  - SPV_INTEL_fpga_memory_attributes
  - SPV_INTEL_kernel_attributes
  - SPV_INTEL_function_pointers
  - SPV_EXT_shader_image_int64
  - SPV_KHR_fragment_shading_rate
  - SPV_EXT_shader_atomic_float_add
- Establish formal meanings for validity (being statically expressed) and behavior (regarding dynamic execution), in Validity and Defined Behavior. This also changed a number of uses of these terms throughout the specifications to be consistent with these definitions.
  - Main issue for this: Khronos issue #540.
  - Addresses Khronos issues #542, #540, #545, #546, #547, and #548.
  - Khronos issue #491: For **OpConvertFToU** and **OpConvertFToS**, behavior is undefined if *Result Type* is not wide enough to hold the converted value.
  - Khronos issue #591: Module validity does not depend on the default values of specialization constants.
- Fix Khronos issues:
  - #214: LoD and gather Image Instructions need non-multisampled images (*MS* of 0), while others that provide a *Sample* Image Operand need a multisampled image (*MS* of 1).
  - #324: For several Capabilities, explicitly list the values **OpTypeImage** has for *Sampled*, instead of saying sampled or unsampled.
  - #361: Stop requiring **OpTypeRuntimeArray** to be concrete, in the description of **OpTypeRuntimeArray**. (This may still be restricted elsewhere though.)
  - #553: Add definition of a tangled instruction and update the definitions of dynamic instance and uniform control flow.

- #517: Expand the About This Document section to also discuss versioning.
- #564: Depth hint for the **DepthLess** execution mode means less-than-or-equal to.
- #558: Explicitly say (rather than imply) that **ImageMipmap** and **ImageReadWrite** capabilities apply to kernels.
- #563: Delete unnecessary statement about incomplete images in **OpImageQueryLod**.
- #570: Update the definitions of the **Acquire** and **Release** memory semantics.
- #560: It is not valid to make duplicate **BuiltIn** variables.
- #566: The Client API specificies what happens with image coordinates outside the image for **OpImageRead**, **OpImageWrite**, and **OpImageSparseRead**.
- #573: Clarify the type read/written is scalar or vector in **OpImageRead**, **OpImageWrite**, and **OpImageSparseRead**.
- #595: Remove the parenthetical partial list of annotation instructions in the logical layout section.
- #574: Constituents of **OpConstantComposite** must not be specialization constants.
- #444: Use more restrictive "only" language for what decorations may apply to.
- MR !182: See the client API for how **SubpassData** coordinates are applied in **OpImageRead**.

4.40. Changes from Version 1.5, Revision 4

- Update to January 7, 2021 public headers.

4.41. Changes from Version 1.5, Revision 5

- Ported the specification itself to use asciidoctor instead of asciidoc.
- Reserve enumerants for the extensions:
  - SPV_INTEL_float_controls2
  - SPV_INTEL_vector_compute
  - SPV_INTEL_arbitrary_precision_floating_point
  - SPV_INTEL_usm_storage_classes
  - SPV_INTEL_unstructured_loop_controls
  - SPV_KHR_subgroup_uniform_control_flow
  - SPV_KHR_linkonce_odr
  - SPV_KHR_expect_assume
  - SPV_EXT_shader_atomic_float_min_max
  - SPV_KHR_integer_dot_product
  - SPV_KHR_bit_instructions
  - SPV_NV_ray_tracing_motion_blur
  - SPV_INTEL_optnone
  - SPV_NV_bindless_texture
- Add **CPP_for_OpenCL** **source language**.
- Clarify that **OpFDiv** has a defined result when the divisor is 0. (MR !195.)
- Fix execution-mode table to show all 3 operands for **LocalSizeHintId**.

- Fix GitHub SPIRV-Registry issues:
  - #79: Clarify the definitions of **StorageImageMultisample** and **ImageMSArray** capabilities.
- Fix Khronos issues:
  - #351: **OpUDiv** and **OpUMod** have undefined behavior if the divisor is 0.
  - #621: Clarify the definition of the *Sampled* operand for **OpTypeImage**.
  - #611: Clarifying string literals are case sensitive for comparisons.
  - #615: Clarify **Block** and **BufferBlock** decorations.
  - #654: Clarify that the **ZeroExtend** image operand is not valid with signed types.
  - #623: Clarify **OpAccessChain** doesn't create any extra restrictions.
  - #647: Clarify **NoWrite** and **NoReadWrite** function parameter attributes apply to the pointer, not to the underlying memory.
  - #585: Clarify that **OpCopyObject** cannot have result type **OpTypeVoid**.
  - #614: Clarify that **OpUndef**, **OpPhi**, and **OpReturnValue** cannot have result type **OpTypeVoid**.
  - #115: Clarify the Shader validation rules for when **OpSelectionMerge** and **OpLoopMerge** instructions are necessary.
  - #656: Clarify the <*id*>-based rules for operands apply only to operands that are <*id*>*s*, in the **OpSpecConstantOp** instruction.
  - #627: Clarify the places that the **RelaxedPrecision** decoration must apply to.
  - #549: Clarify the **VariablePointers** and **VariablePointersStorageBuffer** capabilities enable additional features for logical pointers, but keep other prohibitions. Also that the **VariablePointers** and **VariablePointersStorageBuffer** capabilities allow a pointer to be an operand to **OpReturnValue**.
  - #640: Add parenthetical note in structured control flow about reconverging before reaching a merge block.
  - #656: Clarify the <*id*>-based rules for **OpSpecConstantOp** operands apply only to operands that are <*id*>*s*.
  - #651: Add a validation rule that the workgroup size cannot have a dimension with the value zero statically.
  - #580: Clarify that **SubpassInput** is not valid as the *Dim* operand of **OpTypeSampledImage**, and that sampled images with a *Dim* of **Buffer** are not valid in image sampling instructions.
  - #619: Add a validation rule that **LocalSize**, **LocalSizeId**, **LocalSizeHint**, and **LocalSizeHintId** can't be used at the same time.
  - #663: Restrict **OpSwitch** from being used to directly break or continue in a structured loop.
  - #678: Allow the **AliasedPointer** and **RestrictPointer** decorations to apply to memory object declarations.
  - #682: Clarify that the **VariablePointersStorageBuffer** capability is sufficient to compare pointers that point into different storage buffers using **OpPtrEqual** and **OpPtrNotEqual**.
- Changes from public headers
  - PR #240: Remove the **Kernel** capability from fast-math flags.
  - PR #257: Remove the **Shader** implicit declaration from SPV_EXT_shader_atomic_float_add capabilities.

4.42. Changes from Version 1.5

- New Functionality:
  - Khronos SPIR-V issue #515: The **FPFastMathMode** decoration may now be used with **OpFNegate**, with the binary floating-point comparison instructions (including **OpOrdered** and **OpUnordered**), and with **OpExtInst** where expressly permitted by the extended instruction set.
  - #661: Added a **Nontemporal** Image Operand.
- Extensions Incorporated into SPIR-V 1.6:
  - SPV_KHR_non_semantic_info, see **OpExtInstImport**.
  - SPV_KHR_integer_dot_product
  - SPV_KHR_terminate_invocation
  - SPV_EXT_demote_to_helper_invocation, with changes: Only **OpDemoteToHelperInvocationEXT** was incorporated. Instead of using **OpIsHelperInvocationEXT**, modules should use **Volatile** loads of the **HelperInvocation** built-in variable.
- Deprecations and Removals, from Khronos SPIR-V issues:
  - Removed **OpLessOrGreater**. Use **OpFOrdNotEqual** instead.
  - #620: The **WorkgroupSize** built-in is deprecated starting with version 1.6.
  - #645: The *True Label* and *False Label* of an **OpBranchConditional** must not be the same, starting with version 1.6.
  - #584: Disallow *Dim* **Buffer** in **OpTypeSampledImage** and **OpSampledImage** starting with version 1.6.
  - Deprecated **OpKill**, in favor of **OpTerminateInvocation**, or **OpDemoteToHelperInvocation**.
- Reserve enumerants for the SPV_KHR_fragment_shader_barycentric extension.