SPIR-V Specification

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

Version 1.4, Revision 1, Unified

May 6, 2019

Contents

1 Introduction **10**

1.1 Goals . . . 10
1.2 Execution Environment and Client API . . . 10
1.3 About this document . . . 11
1.4 Extendability . . . 11
1.5 Debuggability . . . 11
1.6 Design Principles . . . 11
1.7 Static Single Assignment (SSA) . . . 12
1.8 Built-In Variables . . . 12
1.9 Specialization . . . 12
1.10 Example . . . 13

2 Specification **16**

2.1 Language Capabilities . . . 16
2.2 Terms . . . 16
2.2.1 Instructions . . . 16
2.2.2 Types . . . 17
2.2.3 Computation . . . 18
2.2.4 Module . . . 18
2.2.5 Control Flow . . . 19
2.3 Physical Layout of a SPIR-V Module and Instruction . . . 21
2.4 Logical Layout of a Module . . . 22
2.5 Instructions . . . 23
2.5.1 SSA Form . . . 23
2.6 Entry Point and Execution Model . . . 24
2.7 Execution Modes . . . 24
2.8 Types and Variables . . . 24
2.8.1 Unsigned Versus Signed Integers . . . 25
2.9 Function Calling . . . 25
2.10 Extended Instruction Sets . . . 25
2.11 Structured Control Flow . . . 26
2.12 Specialization . . . 27
2.13 Linkage . . . 29
2.14 Relaxed Precision . . . 29
2.15 Debug Information . . . 30

2.15.1 Function-Name Mangling 30

2.16 Validation Rules 30

2.16.1 Universal Validation Rules 30

2.16.2 Validation Rules for Shader Capabilities 34

2.16.3 Validation Rules for Kernel Capabilities 35

2.17 Universal Limits 36

2.18 Memory Model 36

2.18.1 Memory Layout 37

2.18.2 Aliasing 37

2.18.3 Null pointers 38

2.19 Derivatives 38

2.20 Code Motion 38

2.21 Deprecation 38

2.22 Unified Specification 39

3 Binary Form **40**

3.1 Magic Number 40

3.2 Source Language 40

3.3 Execution Model 40

3.4 Addressing Model 41

3.5 Memory Model 41

3.6 Execution Mode 41

3.7 Storage Class 49

3.8 Dim 51

3.9 Sampler Addressing Mode 51

3.10 Sampler Filter Mode 52

3.11 Image Format 52

3.12 Image Channel Order 53

3.13 Image Channel Data Type 53

3.14 Image Operands 54

3.15 FP Fast Math Mode 57

3.16 FP Rounding Mode 57

3.17 Linkage Type 57

3.18 Access Qualifier 58

3.19 Function Parameter Attribute 58

3.20 Decoration 59

3.21 BuiltIn 68

3.22 Selection Control 78

3.23 Loop Control . . . 78
3.24 Function Control . . . 79
3.25 Memory Semantics <id> . . . 80
3.26 Memory Operands . . . 82
3.27 Scope <id> . . . 82
3.28 Group Operation . . . 85
3.29 Kernel Enqueue Flags . . . 86
3.30 Kernel Profiling Info . . . 87
3.31 Capability . . . 87
3.32 Instructions . . . 100
- 3.32.1 Miscellaneous Instructions . . . 100
- 3.32.2 Debug Instructions . . . 103
- 3.32.3 Annotation Instructions . . . 106
- 3.32.4 Extension Instructions . . . 109
- 3.32.5 Mode-Setting Instructions . . . 110
- 3.32.6 Type-Declaration Instructions . . . 112
- 3.32.7 Constant-Creation Instructions . . . 119
- 3.32.8 Memory Instructions . . . 124
- 3.32.9 Function Instructions . . . 130
- 3.32.10 Image Instructions . . . 132
- 3.32.11 Conversion Instructions . . . 152
- 3.32.12 Composite Instructions . . . 157
- 3.32.13 Arithmetic Instructions . . . 161
- 3.32.14 Bit Instructions . . . 170
- 3.32.15 Relational and Logical Instructions . . . 175
- 3.32.16 Derivative Instructions . . . 186
- 3.32.17 Control-Flow Instructions . . . 189
- 3.32.18 Atomic Instructions . . . 194
- 3.32.19 Primitive Instructions . . . 203
- 3.32.20 Barrier Instructions . . . 204
- 3.32.21 Group Instructions . . . 206
- 3.32.22 Device-Side Enqueue Instructions . . . 217
- 3.32.23 Pipe Instructions . . . 226
- 3.32.24 Non-Uniform Instructions . . . 241

A Changes **257**

A.1 Changes from Version 0.99, Revision 31 . . . 257
A.2 Changes from Version 0.99, Revision 32 . . . 258
A.3 Changes from Version 1.00, Revision 1 . . . 258
A.4 Changes from Version 1.00, Revision 2 . . . 259
A.5 Changes from Version 1.00, Revision 3 . . . 260
A.6 Changes from Version 1.00, Revision 4 . . . 260
A.7 Changes from Version 1.00, Revision 5 . . . 260
A.8 Changes from Version 1.00, Revision 6 . . . 261
A.9 Changes from Version 1.00, Revision 7 . . . 261
A.10 Changes from Version 1.00, Revision 8 . . . 261
A.11 Changes from Version 1.00, Revision 9 . . . 261
A.12 Changes from Version 1.00, Revision 10 . . . 262
A.13 Changes from Version 1.00, Revision 11 . . . 262
A.14 Changes from Version 1.00 . . . 263
A.15 Changes from Version 1.1, Revision 1 . . . 263
A.16 Changes from Version 1.1, Revision 2 . . . 263
A.17 Changes from Version 1.1, Revision 3 . . . 263
A.18 Changes from Version 1.1, Revision 4 . . . 263
A.19 Changes from Version 1.1, Revision 5 . . . 263
A.20 Changes from Version 1.1, Revision 6 . . . 264
A.21 Changes from Version 1.1, Revision 7 . . . 264
A.22 Changes from Version 1.1 . . . 264
A.23 Changes from Version 1.2, Revision 1 . . . 264
A.24 Changes from Version 1.2, Revision 2 . . . 264
A.25 Changes from Version 1.2, Revision 3 . . . 264
A.26 Changes from Version 1.2 . . . 265
A.27 Changes from Version 1.3, Revision 1 . . . 265
A.28 Changes from Version 1.3, Revision 2 . . . 266
A.29 Changes from Version 1.3, Revision 3 . . . 266
A.30 Changes from Version 1.3, Revision 4 . . . 267
A.31 Changes from Version 1.3, Revision 5 . . . 267
A.32 Changes from Version 1.3, Revision 6 . . . 268
A.33 Changes from Version 1.3, Revision 7 . . . 268
A.34 Changes from Version 1.3 . . . 269

List of Tables

1 First Words of Physical Layout . . . 21
2 Instruction Physical Layout . . . 21
3 Limits . . . 36

Contributors and Acknowledgments

Connor Abbott, Intel

Ben Ashbaugh, Intel

Alexey Bader, Intel

Alan Baker, Google

Dan Baker, Oxide Games

Kenneth Benzie, Codeplay

Gordon Brown, Codeplay

Pat Brown, NVIDIA

Diana Po-Yu Chen, MediaTek

Stephen Clarke, Imagination

Patrick Doane, Blizzard Entertainment

Stefanus Du Toit, Google

Tim Foley, Intel

Ben Gaster, Qualcomm

Alexander Galazin, ARM

Christopher Gautier, ARM

Neil Henning, AMD

Kerch Holt, NVIDIA

Lee Howes, Qualcomm

Roy Ju, MediaTek

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

Christophe Riccio, Unity

Andrew Richards, Codeplay

Ian Romanick, Intel

Graham Sellers, AMD

Robert Simpson, Qualcomm

Bartosz Sochacki, Intel

Nikos Stavropoulos, Think Silicon

Brian Sumner, AMD

Andrew Woloszyn, Google

Ruihao Zhang, Qualcomm

Weifeng Zhang, Qualcomm

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

1 Introduction

Abstract

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

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

- Include everything needed to fully understand, create, and consume SPIR-V. However:
  - Extended instruction sets can be imported and come with their own specifications.
  - Client API-specific rules are documented in client API specifications.
- Separate expository and specification language. The specification-proper is in Specification and Binary Form.

1.4 Extendability

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

- Using the OpExtension instruction to require new semantics that must be supported. Such new semantics would come from an extension 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 will follow the rule: "same spelling, same semantics", and does not have mode bits that modify semantics. If a change to SPIR-V modifies semantics, it should use a different spelling. This makes consumers of SPIR-V much more robust. There are execution modes declared, but these 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 will be 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 will be done through load/store, and initial load and final store can never be eliminated. Other storage classes are shader local and can have their load/store eliminated. It can be considered an optimization to largely eliminate such loads/stores by moving them into intermediate results in SSA form.

1.8 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 must otherwise be 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
```

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 must be 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.

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

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

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

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

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

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

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

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

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

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

Memory Object Declaration: An OpVariable, or an OpFunctionParameter of pointer type.

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

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

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

2.2.2 Types

Boolean type: The type returned by OpTypeBool.

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

Floating-point type: Any width type from OpTypeFloat.

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

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

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

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

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

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

Aggregate: A structure or an array.

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

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

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

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

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

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

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

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

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

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

- OpSelectionMerge
- OpLoopMerge

Header Block: A block containing a merge instruction.

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

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

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

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

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

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

Subgroup: 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 a single point, line, or triangle, including any helper invocations.

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

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

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

2.3 Physical Layout of a SPIR-V Module and Instruction

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

Table 1: First Words of Physical Layout

Word Number	Contents
0	Magic Number.
1	Version number. The bytes are, high-order to low-order: *0 | Major Number | Minor Number | 0* Hence, version 1.3 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://www.khronos.org/registry/spir-v/api/spir-v.xml.
3	*Bound*; where all <id>s in this module are guaranteed to satisfy *0 < id < Bound* *Bound* should be small, smaller is better, with all <id> in a module being densely packed and near 0.
4	0 (Reserved for instruction schema, if needed.)
5	First word of instruction stream, see below.

All remaining words are a linear sequence of instructions.

Each instruction is a stream of words:

Table 2: Instruction Physical Layout

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

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

2.4 Logical Layout of a Module

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

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

Within a function definition:

- A block always starts with an OpLabel instruction. This may be immediately preceded by an OpLine instruction, but the **OpLabel** is considered as the beginning of the block.
- A block always ends with a 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 immediately preceding OpLine instructions, must be the first instructions in that block. (Note the validation rules prevent OpPhi instructions in the first block of a function.)

- A function definition (starts with OpFunction) can be immediately preceded by an OpLine instruction.

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

- Operands that are an OpFunction. This allows for recursion and early declaration of entry points.
- Annotation-instruction operands. This is required to fully know everything about a type or variable once it is declared.
- Labels.
- OpPhi can contain forward references.
- An OpTypeForwardPointer has a forward reference to an OpTypePointer.
- An OpTypeStruct operand that's a forward reference to the *Pointer Type* operand to an OpTypeForwardPointer.
- The list of *<id>* provided in the OpEntryPoint instruction.
- 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 that must be satisfied.

2.5 Instructions

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

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

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

2.5.1 SSA Form

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

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

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

2.6 Entry Point and Execution Model

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

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

2.7 Execution Modes

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

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

For a complete list, see Execution Mode.

2.8 Types and Variables

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

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

Higher-level types are then constructed using opcodes like OpTypeVector, OpTypeMatrix, OpTypeImage, OpTypeArray, OpTypeRuntimeArray, OpTypeStruct, and OpTypePointer. These are parameterized by number of components, array size, member lists, etc. The image types are parameterized by the return type, dimensionality, arrayness, etc. To do sampling or filtering operations, a type from OpTypeSampledImage is used that contains both an image and a sampler. Such a sampled image can be set directly by the 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 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**). When new pointers are formed, their types must be decorated as needed, so the consumer knows how to generate an access through the pointer. Non-aggregate non-pointer types are different: It is invalid to declare multiple type <*id*>*s* for the same scalar, vector, or matrix type. That is, non-aggregate non-pointer type declarations must all have different opcodes or operands. (Note that non-aggregate non-pointer types cannot be decorated in ways that affect their type.)

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

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

Built-in variables 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 typical ways to think about signedness in SPIR-V, both equally valid:

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 when 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 when signedness matters to external interface, or when 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.

2.9 Function Calling

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

2.10 Extended Instruction Sets

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

- trigonometric functions: sin(), cos(), . . .
- exponentiation functions: exp(), pow(), . . .
- geometry functions: reflect(), smoothstep(), . . .
- functions having rich performance/accuracy trade-offs
- etc.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.11 Structured Control Flow

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

Structured control-flow declarations must satisfy the following rules:

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

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

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

2.12 Specialization

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

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

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

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

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

- OpSpecConstantTrue
- OpSpecConstantFalse
- OpSpecConstant
- OpSpecConstantComposite
- OpSpecConstantOp

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

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

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

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

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

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

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

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

2.13 Linkage

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

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

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

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

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

2.14 Relaxed Precision

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

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

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

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

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

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

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

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

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

- A structure-type member (through OpMemberDecorate).

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

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

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

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 when 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.

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

2.15.1 Function-Name Mangling

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

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

2.16 Validation Rules

2.16.1 Universal Validation Rules

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

- The stream of instructions must be ordered as described in the Logical Layout section.
- Any use of a feature described by a capability in the capability section requires that capability to be declared, either directly, or as an "implicitly declares" capability on a capability that is declared.
- Non-structure types (scalars, vectors, arrays, etc.) with the same operand parameterization cannot be type aliases. For non-structures, two type *<id>s* match if-and-only-if the types match.
- If the **Logical** addressing model is selected and the **VariablePointers** capability is not declared:
  - OpVariable cannot allocate an object whose type is a pointer type (that is, it cannot create an object in memory that is itself a pointer and whose result would thus be a pointer to a pointer)
  - A pointer can only be an operand to the following instructions:

    * OpLoad
    * OpStore
    * OpAccessChain
    * OpInBoundsAccessChain
    * OpFunctionCall
    * OpImageTexelPointer
    * OpCopyMemory
    * OpCopyObject
    * all OpAtomic instructions
    * extended instruction-set instructions that are explicitly identified as taking pointer operands

  - A pointer can be the Result <id> of only the following instructions:
    * OpVariable
    * OpAccessChain
    * OpInBoundsAccessChain
    * OpFunctionParameter
    * OpImageTexelPointer
    * OpCopyObject

  - All indexes in OpAccessChain and OpInBoundsAccessChain that are OpConstant with type of OpTypeInt with a *signedness* of 1 must not have their sign bit set.
  - 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 cannot be used.

- If the **Logical** addressing model is selected and the **VariablePointers** or **VariablePointersStorageBuffer** capability is declared (in addition to what is allowed above by the **Logical** addressing model):
  - **OpVariable** can allocate an object whose type is a pointer type, if the *Storage Class* of the **OpVariable** is one of the following:
    * **Function**
    * **Private**

  - A pointer can be the *Object* operand of **OpStore** or result of **OpLoad**, if the storage class the pointer is stored to or loaded from is 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 or an operand to one of:
    * 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 when the Storage Class of the operands' OpTypePointer declaration is
    * **StorageBuffer** when the **VariablePointersStorageBuffer** capability is explicitly or implicitly declared, or
    * **Workgroup**, which can be used only if the **VariablePointers** capability was declared.

- A variable pointer with the **Logical** addressing model cannot
  - 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**

- SSA
  - Each <id> must appear exactly once as the Result <id> of an instruction.
  - The definition of an SSA *<id>* should dominate all uses of it, with the following exceptions:
    * Function calls may call functions not yet defined. However, note that the function's argument and return types will already be known at the call site.
    * 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.
  - No function can be targeted by both an OpEntryPoint instruction and an OpFunctionCall instruction.
  - Each OpEntryPoint can have set at most one of the **DenormFlushToZero** or **DenormPreserve** execution modes for any given *Target Width*.
  - Each OpEntryPoint can have set at most one of the **RoundingModeRTE** or **RoundingModeRTZ** execution modes for any given *Target Width*.

- Functions
  - A function declaration (an OpFunction with no basic blocks), must have a **Linkage Attributes** Decoration with the **Import** Linkage Type.
  - A function definition (an OpFunction with basic blocks) cannot be decorated with the **Import** Linkage Type.
  - A function cannot have both a declaration and a definition (no forward declarations).

- Global (Module Scope) Variables
  - It is illegal to initialize an imported variable. This means that a module-scope OpVariable with initialization value cannot be marked with the **Import** Linkage Type.

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

- A Result <id> resulting from an instruction within a function can only be used in that function.
- A function call must have the same number of arguments as the function definition (or declaration) has parameters, and their respective types must match.
- An instruction requiring a specific number of operands must have that many operands. The word count must agree.
- Each opcode specifies its own requirements for number and type of operands, and these must be followed.
- Atomic access rules
  - The pointers taken by atomic operation instructions must be a pointer into one of the following Storage Classes:
    * **Uniform** when used with the **BufferBlock** Decoration
    * **StorageBuffer**
    * **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*.

2.16.2 Validation Rules for Shader Capabilities

- CFG:
  - Loops must be structured, having an OpLoopMerge instruction in their header.
  - Selections must be structured, having an OpSelectionMerge instruction in their header.
- Entry point and execution model
  - Each entry point in a module, along with its corresponding static call tree within that module, forms a complete pipeline stage.
  - Each OpEntryPoint with the **Fragment** Execution Model must have an OpExecutionMode for either the **OriginLowerLeft** or the **OriginUpperLeft** Execution Mode. (Exactly one of these is required.)
  - An OpEntryPoint with the **Fragment** Execution Model can set at most one of the **DepthGreater**, **DepthLess**, or **DepthUnchanged** Execution Modes.
  - An OpEntryPoint with one of the **Tessellation** Execution Models can set at most one of the **SpacingEqual**, **SpacingFractionalEven**, or **SpacingFractionalOdd** Execution Modes.
  - An OpEntryPoint with one of the **Tessellation** Execution Models can set at most one of the **Triangles**, **Quads**, or **Isolines** Execution Modes.
  - An OpEntryPoint with one of the **Tessellation** Execution Models can set at most one of the **VertexOrderCw** or **VertexOrderCcw** Execution Modes.
  - An OpEntryPoint with the **Geometry** Execution Model must set exactly one of the **InputPoints**, **InputLines**, **InputLinesAdjacency**, **Triangles**, or **TrianglesAdjacency** Execution Modes.
  - An OpEntryPoint with the **Geometry** Execution Model must set exactly one of the **OutputPoints**, **OutputLineStrip**, or **OutputTriangleStrip** Execution Modes.
- Composite objects in the **StorageBuffer**, **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 have be decorated with
    * a **MatrixStride** Decoration, and
    * one of the **RowMajor** or **ColMajor** decorations.
  - The **ArrayStride**, **MatrixStride**, and **Offset** decorations must be large enough to hold the size of the objects they affect (that is, specifying overlap is invalid). Each **ArrayStride** and **MatrixStride** must be greater than zero, and no two members of a given structure can be assigned to the same **Offset**.
  - Each **OpPtrAccessChain** must have a *Base* whose type is decorated with **ArrayStride**.
  - When 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 cannot be decorated with **ArrayStride**.

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

- Data Rules
  - The type for any intermediate object that is an opaque type is restricted to those types that are valid for declaring a global OpVariable.

- Decorations
  - At most one of **Noperspective** or **Flat** decorations can be applied to the same object or member.
  - At most one of **Patch**, **Centroid**, or **Sample** decorations can be applied to the same object or member.
  - At most one of **RowMajor** and **ColMajor** decorations can be applied to a structure type.
  - At most one of **Block** and **BufferBlock** decorations can be applied to a structure type.
  - **Block** and **BufferBlock** decorations cannot decorate a structure type that is nested at any level inside another structure type decorated with **Block** or **BufferBlock**.
  - The **FPRoundingMode** decoration can 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**, **Uniform**, or **Output** Storage Classes.

- All <*id*> used for Scope and Memory Semantics 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 must either

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

Table 3: Limits

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

2.18 Memory Model

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

The **Logical** addressing model means pointers are abstract, having no physical size or numeric value. In this mode, pointers can only be created from existing objects, and they cannot be stored into an object, 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 cannot be used to access objects in another Storage Class. However, they can be converted with conversion opcodes. Any particular addressing model must describe the bit width of pointers for each of the storage classes.

2.18.1 Memory Layout

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

- a vector consumes contiguous memory with lower-numbered components appearing in smaller offsets than higher-numbered components, and with component 0 starting at the vector's **Offset** Decoration, if present
- in an array, lower-numbered elements appear at smaller offsets than higher-numbered elements, with element 0 starting at the **Offset** Decoration for the array, if present
- 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, then those accesses must be ordered according to the memory consistency model specified by the client API.

Alias management depends on the memory model:

- The **Simple** and **GLSL** 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*>. Applying **Restrict** is allowed, but has no effect. Only those memory object declarations decorated with **Aliased** may alias each other.
- The **OpenCL** memory model must, unless otherwise proven, assume 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**, **Uniform**, **UniformConstant**
Output	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Output**
Private	**OpFunctionParameter**	**OpFunctionParameter**, **OpVariable**	**Private**
StorageBuffer	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **Uniform**, **UniformConstant**
Uniform	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **Uniform**, **UniformConstant**
UniformConstant	**OpFunctionParameter**, **OpVariable**	**OpFunctionParameter**, **OpVariable**	**Image**, **StorageBuffer**, **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 cannot make any assumptions about whether or not memory regions of non memory object declarations overlap. As such, a consumer must perform dependency analysis on non memory object declarations if it wishes to reorder instructions affecting memory. Behavior is undefined when 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.

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 when loading or storing through an **OpConstantNull** value.

2.19 Derivatives

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

2.20 Code Motion

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

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**.

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**

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**
3	**Geometry** Geometry shading stage.	**Geometry**
4	**Fragment** Fragment shading stage.	**Shader**
5	**GLCompute** Graphical compute shading stage.	**Shader**

Execution Model		Enabling Capabilities
6	**Kernel** Compute kernel.	**Kernel**
5267	**TaskNV**	**MeshShadingNV**
5268	**MeshNV**	**MeshShadingNV**
5313	**RayGenerationNV**	**RayTracingNV**
5314	**IntersectionNV**	**RayTracingNV**
5315	**AnyHitNV**	**RayTracingNV**
5316	**ClosestHitNV**	**RayTracingNV**
5317	**MissNV**	**RayTracingNV**
5318	**CallableNV**	**RayTracingNV**

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	**PhysicalStorageBuffer64EXT**	**PhysicalStorageBufferAddressesEXT** 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**
1	**GLSL450** Memory model needed by later versions of GLSL and ESSL. Works across multiple versions.	**Shader**
2	**OpenCL** OpenCL memory model.	**Kernel**
3	**VulkanKHR**	**VulkanMemoryModelKHR**

3.6 Execution Mode

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

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

Execution Mode		Extra Operands	Enabling Capabilities
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**
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 will run in transform feedback-capturing mode and this module is responsible for describing the transform-feedback setup. See the **XfbBuffer**, **Offset**, and **XfbStride** Decorations.		**TransformFeedback**
12	**DepthReplacing** This mode must be declared if and only if this entry point dynamically writes the **FragDepth**-decorated variable. 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 will be 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 will be less than 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 will be 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. Only valid with the **GLCompute** or **Kernel** Execution Models.	Literal Number *x size*	Literal Number *y size*	Literal Number *z size*	
18	**LocalSizeHint** A hint to the compiler, which indicates the most likely to be used work-group size in the *x*, *y*, and *z* dimensions. Only valid with the **Kernel** Execution Model.	Literal Number *x size*	Literal Number *y size*	Literal Number *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**
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**

Execution Mode		Extra Operands	Enabling Capabilities
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** For a geometry stage, the maximum number of vertices the shader will ever emit in a single invocation. For a tessellation-control stage, the number of vertices in the output patch produced by the tessellation control shader, which also specifies the number of times the tessellation control shader is invoked. Only valid with the **Geometry** or one of the tessellation Execution Models.	Literal Number *Vertex count*	**Geometry**, **Tessellation**, **MeshShadingNV**
27	**OutputPoints** Stage output primitive is *points*. Only valid with the **Geometry** Execution Model.		**Geometry**, **MeshShadingNV**
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**

Execution Mode		Extra Operands			Enabling Capabilities
30	**VecTypeHint** A hint to the compiler, which indicates that most operations used in the entry point are explicitly vectorized using a particular vector type. The 16 high-order bits of *Vector Type* operand specify the *number of components* of the vector. The 16 low-order bits of *Vector Type* operand specify the *data type* of the vector. These are the legal *data type* values: *0* represents an 8-bit integer value. *1* represents a 16-bit integer value. *2* represents a 32-bit integer value. *3* represents a 64-bit integer value. *4* represents a 16-bit float value. *5* represents a 32-bit float value. *6* represents a 64-bit float value. Only valid with the **Kernel** Execution Model.	Literal Number *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**.
35	**SubgroupSize** Indicates that this entry point requires the specified *Subgroup Size*.	Literal Number *Subgroup Size*			**SubgroupDispatch** Missing before **version 1.1**.
36	**SubgroupsPerWorkgroup** Indicates that this entry point requires the specified number of *Subgroups Per Workgroup*.	Literal Number *Subgroups Per Workgroup*			**SubgroupDispatch** Missing before **version 1.1**.
37	**SubgroupsPerWorkgroupId** Indicates that this entry point requires the specified number of *Subgroups Per Workgroup*. Specified as an Id.	*<id>* *Subgroups Per Workgroup*			**SubgroupDispatch** Missing before **version 1.2**.
38	**LocalSizeId** Indicates the work-group size in the *x*, *y*, and *z* dimensions. Only valid with the **GLCompute** or **Kernel** Execution Models. Specified as Ids.	*<id>* *x size*	*<id>* *y size*	*<id>* *z size*	Missing before **version 1.2**.

Execution Mode		Extra Operands	Enabling Capabilities
39	**LocalSizeHintId** A hint to the compiler, which indicates the most likely to be used work-group size in the *x*, *y*, and *z* dimensions. Only valid with the **Kernel** Execution Model. Specified as an Id.	<*id*> *Local Size Hint*	**Kernel** Missing before **version 1.2**.
4446	**PostDepthCoverage**		**SampleMaskPostDepthCoverage** Reserved. Also see extension: **SPV_KHR_post_depth_coverage**
4459	**DenormPreserve** Any denormalized value input into a shader or potentially generated by any instruction in a shader must be 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 must be preserved. Only affects instructions operating on a floating-point type whose component width is *Target Width*.	Literal Number *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 must be 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 must be flushed to zero. Only affects instructions operating on a floating-point type whose component width is *Target Width*.	Literal Number *Target Width*	**DenormFlushToZero** Missing before **version 1.4**. Also see extension: **SPV_KHR_float_controls**
4461	**SignedZeroInfNanPreserve** The implementation must 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*.	Literal Number *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 must be 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*.	Literal Number *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 must be 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*.	Literal Number *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 Number *Primitive count*	**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**
5289	**DerivativeGroupQuadsNV**		**ComputeDerivativeGroupQuadsNV** Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**

Execution Mode		Extra Operands	Enabling Capabilities
5290	**DerivativeGroupLinearNV**		**ComputeDerivativeGroupLinearNV** Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5298	**OutputTrianglesNV**		**MeshShadingNV** Reserved. Also see extension: **SPV_NV_mesh_shader**

3.7 Storage Class

Class of storage for declared variables (does not include intermediate values). 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 cannot have initializers.	
2	**Uniform** Shared externally, visible across all functions in all invocations in all work groups. Graphics uniform blocks and buffer blocks.	**Shader**
3	**Output** Output to pipeline. Visible across all functions in the current invocation.	**Shader**
4	**Workgroup** Shared across all invocations within a work group. Visible across all functions. The OpenGL "shared" storage qualifier. OpenCL local memory.	
5	**CrossWorkgroup** Visible across all functions of all invocations of all work groups. OpenCL global memory.	

Storage Class		Enabling Capabilities
6	**Private** Visible to all functions in the current invocation. Regular global memory.	**Shader**
7	**Function** Visible only within the declaring function of the current invocation. Regular function memory.	
8	**Generic** For generic pointers, which overload the **Function**, **Workgroup**, and **CrossWorkgroup** Storage Classes.	**GenericPointer**
9	**PushConstant** For holding push-constant memory, visible across all functions in all invocations in all work groups. Intended to contain a small bank of values pushed from the client API. Variables declared with this storage class are read-only, and cannot 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**
5328	**CallableDataNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5329	**IncomingCallableDataNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5338	**RayPayloadNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5339	**HitAttributeNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5342	**IncomingRayPayloadNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**

Storage Class		Enabling Capabilities
5343	**ShaderRecordBufferNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5349	**PhysicalStorageBufferEXT**	**PhysicalStorageBufferAddressesEXT** Also see extension: **SPV_EXT_physical_storage_buffer**

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**
2	**Clamp** Out-of-range image coordinates will return a border color.	**Kernel**
3	**Repeat** Out-of-range image coordinates are wrapped to the valid range. Can only be used with normalized coordinates.	**Kernel**
4	**RepeatMirrored** Flip the image coordinate at every integer junction. Can only be used with normalized coordinates.	**Kernel**

3.10 Sampler Filter Mode

Filter mode for creating constant samplers. Used by OpConstantSampler.

Sampler Filter Mode		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**
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**

Image Format		Enabling Capabilities
36	**Rg16ui**	**StorageImageExtendedFormats**
37	**Rg8ui**	**StorageImageExtendedFormats**
38	**R16ui**	**StorageImageExtendedFormats**
39	**R8ui**	**StorageImageExtendedFormats**

3.12 Image Channel Order

Image channel order returned by OpImageQueryOrder.

Image Channel Order		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 type returned by OpImageQueryFormat.

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

Image Channel Data Type		Enabling Capabilities
15	**UnormInt24**	**Kernel**
16	**UnormInt101010_2**	**Kernel**

3.14 Image Operands

Additional operands to sampling, or getting texels from, an image. Bits that are set can indicate 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).

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

Used by:

- OpImageSampleImplicitLod
- OpImageSampleExplicitLod
- OpImageSampleDrefImplicitLod
- OpImageSampleDrefExplicitLod
- OpImageSampleProjImplicitLod
- OpImageSampleProjExplicitLod
- OpImageSampleProjDrefImplicitLod
- OpImageSampleProjDrefExplicitLod
- OpImageFetch
- OpImageGather
- OpImageDrefGather
- OpImageRead
- OpImageWrite
- OpImageSparseSampleImplicitLod
- OpImageSparseSampleExplicitLod
- OpImageSparseSampleDrefImplicitLod
- OpImageSparseSampleDrefExplicitLod
- OpImageSparseSampleProjImplicitLod
- OpImageSparseSampleProjExplicitLod
- OpImageSparseSampleProjDrefImplicitLod
- OpImageSparseSampleProjDrefExplicitLod
- OpImageSparseFetch
- OpImageSparseGather
- OpImageSparseDrefGather
- OpImageSparseRead
- OpImageSampleFootprintNV

Image Operands		Enabling Capabilities
0x0	**None**	

Image Operands		Enabling Capabilities
0x1	**Bias** A following operand is the bias added to the implicit level of detail. Only valid with implicit-lod instructions. It must be a floating-point type scalar. This can only be used with an OpTypeImage that has a Dim operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	**Shader**
0x2	**Lod** A following operand is the explicit level-of-detail to use. Only valid with explicit-lod instructions. For sampling operations, it must be a floating-point type scalar. For fetch operations, it must be an integer type scalar. This can only be used with an OpTypeImage that has a Dim operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	
0x4	**Grad** Two following operands are *dx* followed by *dy*. These are explicit derivatives in the *x* and *y* direction to use in computing level of detail. Each is a scalar or vector containing (*du/dx*[, *dv/dx*] [, *dw/dx*]) and (*du/dy*[, *dv/dy*] [, *dw/dy*]). The number of components of each must equal the number of components in *Coordinate*, minus the *array layer* component, if present. Only valid with explicit-lod instructions. They must be a scalar or vector of floating-point type. This can only be used with an OpTypeImage that has an *MS* operand of 0. It is invalid to set both the **Lod** and **Grad** bits.	
0x8	**ConstOffset** A following operand is added to (*u*, *v*, *w*) before texel lookup. It must be an <*id*> of an integer-based constant instruction of scalar or vector type. It is 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.	

Image Operands		Enabling Capabilities
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.	**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.	**ImageGatherExtended**
0x40	**Sample** A following operand is the sample number of the sample to use. Only valid with OpImageFetch, OpImageRead, OpImageWrite, OpImageSparseFetch, and OpImageSparseRead. It is invalid to have a **Sample** operand if the underlying OpTypeImage has *MS* of 0. It must be an integer type scalar.	
0x80	**MinLod** A following operand is the minimum level-of-detail to use when accessing the image. Only valid with **Implicit** instructions and **Grad** instructions. It must be a floating-point type scalar. This can only be used with an OpTypeImage that has a Dim operand of **1D**, **2D**, **3D**, or **Cube**, and the *MS* operand must be 0.	**MinLod**
0x100	**MakeTexelAvailableKHR**	**VulkanMemoryModelKHR**
0x200	**MakeTexelVisibleKHR**	**VulkanMemoryModelKHR**
0x400	**NonPrivateTexelKHR**	**VulkanMemoryModelKHR**
0x800	**VolatileTexelKHR**	**VulkanMemoryModelKHR**
0x1000	**SignExtend** The texel value is converted to the target value via sign extension. Only valid when the texel type is a scalar or vector of integer type.	Missing before **version 1.4**.

Image Operands		Enabling Capabilities
0x2000	**ZeroExtend** The texel value is converted to the target value via zero extension. Only valid when the texel type is a scalar or vector of integer type.	Missing before **version 1.4**.

3.15 FP Fast Math Mode

Enables fast math operations which are otherwise unsafe.

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

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

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

3.16 FP Rounding Mode

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

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**

3.18 Access Qualifier

Defines the access permissions.

Used by OpTypeImage and OpTypePipe.

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

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

3.20 Decoration

Used by:

- OpDecorate
- 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 to a scalar specialization constant. Forms the external linkage for setting a specialized value. See specialization.	Literal Number *Specialization Constant ID*	**Shader**, **Kernel**
2	**Block** Apply to a structure type to establish it is a non-SSBO-like shader-interface block.		**Shader**
3	**BufferBlock** Deprecated (use **Block**-decorated **StorageBuffer** Storage Class objects). Apply to a structure type to establish it is an SSBO-like shader-interface block.		**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.		**Matrix**
5	**ColMajor** Applies only to a member of a structure type. Only valid on a matrix or array whose most basic element is a matrix. Indicates that components within a column are contiguous in memory.		**Matrix**

Decoration		Extra Operands	Enabling Capabilities
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, to specify the stride of the array that the element resides in. Must not be applied to any other type.	Literal Number *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. Specifies the stride of rows in a **RowMajor**-decorated matrix, or columns in a **ColMajor**-decorated matrix.	Literal Number *Matrix Stride*	**Matrix**
8	**GLSLShared** Apply to a structure type to get GLSL **shared** memory layout.		**Shader**
9	**GLSLPacked** Apply to a structure type to get GLSL **packed** memory layout.		**Shader**
10	**CPacked** Apply to a structure type, to marks it as "packed", indicating that the alignment of the structure is one and that there is no padding between structure members.		**Kernel**
11	**BuiltIn** 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. Indicates that linear, non-perspective correct, interpolation must be used. 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 will be done. The non-interpolated value will come 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. When used with multi-sampling rasterization, allows a single interpolation location for an entire pixel. The interpolation location must lie in both the pixel and in the primitive being rasterized. Only valid for the **Input** and **Output** Storage Classes.		**Shader**
17	**Sample** Must only be used on a memory object declaration or a member of a structure type. When used with multi-sampling rasterization, requires per-sample interpolation. The interpolation locations must be the locations of the samples lying in both the pixel and in the primitive being rasterized. Only valid for the **Input** and **Output** Storage Classes.		**SampleRateShading**
18	**Invariant** Apply to a variable, to indicate expressions computing its value be done invariant with respect to other modules computing the same expressions.		**Shader**
19	**Restrict** Apply to a 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 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.		
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.		

Decoration		Extra Operands	Enabling Capabilities
22	**Constant** Indicates that a global variable is constant and will **never** be modified. Only allowed on global variables.		**Kernel**
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.		
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 decoration indicates the memory holding the variable is not writable, and that this module does not write to it. It does not prevent the use of initializers on a declaration.		
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 the memory holding the variable is not readable, and that this module does not read from it.		

Decoration		Extra Operands	Enabling Capabilities
26	**Uniform** Apply 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 will compute the same result value.		**Shader**
27	**UniformId** Apply 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** Missing before **version 1.4**.
28	**SaturatedConversion** Indicates that a conversion to an integer type which is outside the representable range of *Result Type* will be clamped to the nearest representable value of *Result Type*. *NaN* will be converted to *0*. This decoration can only be applied to conversion instructions to integer types, not including the OpSatConvertUToS and OpSatConvertSToU instructions.		**Kernel**
29	**Stream** Must only be used on a memory object declaration or a member of a structure type. Indicates the stream number to put an output on. Only valid for the **Output** Storage Class and the **Geometry** Execution Model.	Literal Number *Stream Number*	**GeometryStreams**
30	**Location** Apply to a variable or a structure-type member. 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. Also can 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 Number *Location*	**Shader**

Decoration		Extra Operands	Enabling Capabilities
31	**Component** Must only be used on a memory object declaration or a member of a structure type. Indicates which component within a **Location** will be taken by the decorated entity. Only valid for the **Input** and **Output** Storage Classes.	Literal Number *Component*	**Shader**
32	**Index** Apply to a variable to identify 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 Number *Index*	**Shader**
33	**Binding** Apply to a variable. Part of the main linkage between the client API and SPIR-V modules for memory buffers, images, etc. See the client API specification for more detail.	Literal Number *Binding Point*	**Shader**
34	**DescriptorSet** Apply to a variable. Part of the main linkage between the client API and SPIR-V modules for memory buffers, images, etc. See the client API specification for more detail.	Literal Number *Descriptor Set*	**Shader**
35	**Offset** Apply to a structure-type member. This gives the byte offset of the member relative to the beginning of the structure. Can be used, for example, by both uniform and transform-feedback buffers. It must not cause any overlap of the structure's members, or overflow of a transform-feedback buffer's **XfbStride**.	Literal Number *Byte Offset*	**Shader**
36	**XfbBuffer** Must only be used on a memory object declaration or a member of a structure type. Indicates which transform-feedback buffer an output is written to. Only valid for the **Output** Storage Classes of vertex processing Execution Models.	Literal Number *XFB Buffer Number*	**TransformFeedback**
37	**XfbStride** Apply to anything **XfbBuffer** is applied to. Specifies 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 Number *XFB Stride*	**TransformFeedback**

Decoration		Extra Operands		Enabling Capabilities
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*		
40	**FPFastMathMode** Indicates a floating-point fast math flag.	FP Fast Math Mode *Fast-Math Mode*		**Kernel**
41	**LinkageAttributes** Associate linkage attributes to values. Only valid on OpFunction or global (module scope) OpVariable. See linkage.	Literal String *Name*	Linkage Type *Linkage Type*	**Linkage**
42	**NoContraction** Apply to an arithmetic instruction to indicate the operation cannot be combined with another instruction to form a single operation. For example, if applied to an OpFMul, that multiply can't be combined with an addition to yield a fused multiply-add operation. Furthermore, such operations are not allowed to reassociate; e.g., add(a + add(b+c)) cannot be transformed to add(add(a+b) + c).			**Shader**
43	**InputAttachmentIndex** Apply to a variable to provide an input-target index (as 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 Number *Attachment Index*		**InputAttachment**
44	**Alignment** Apply to a pointer. This declares a known minimum alignment the pointer has.	Literal Number *Alignment*		**Kernel**
45	**MaxByteOffset** Apply to a pointer. This declares a known maximum byte offset this pointer will be incremented by from the point of the decoration. This is a guaranteed upper bound when applied to OpFunctionParameter.	Literal Number *Max Byte Offset*		**Addresses** Missing before **version 1.1**.
46	**AlignmentId** Apply to a pointer. This declares a known minimum alignment the pointer has. Specified as an Id.	*<id>* *Alignment*		**Kernel** Missing before **version 1.2**.

Decoration		Extra Operands	Enabling Capabilities
47	**MaxByteOffsetId** Apply to a pointer. This declares a known maximum byte offset this pointer will be incremented by from the point of the decoration. This is a guaranteed upper bound when applied to OpFunctionParameter. Specified as an Id.	*<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 can 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, the behavior is undefined.		Missing before **version 1.4**. Also see extension: **SPV_KHR_no_integer_wrap_decoration**
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 can 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, the behavior is undefined.		Missing before **version 1.4**. Also see extension: **SPV_KHR_no_integer_wrap_decoration**

Decoration		Extra Operands	Enabling Capabilities
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 Number *Offset*	**ShaderStereoViewNV** Reserved. Also see extension: **SPV_NV_stereo_view_rendering**
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	**PerVertexNV**		**FragmentBarycentricNV** Reserved. Also see extension: **SPV_NV_fragment_shader_barycentric**
5300	**NonUniformEXT**		**ShaderNonUniformEXT**

Decoration		Extra Operands	Enabling Capabilities
5634	**CounterBuffer** The <*id*> of a counter buffer associated with the decorated buffer. It can 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**
5635	**UserSemantic** A string describing a user-defined semantic intent of what it decorates. *Semantic* is case insensitive. It can 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 String *Semantic*	Missing before **version 1.4**.
5635	**HlslSemanticGOOGLE**	Literal String *Semantic*	Reserved. Also see extension: **SPV_GOOGLE_hlsl_functionality1**
5355	**RestrictPointerEXT**		**PhysicalStorageBufferAddressesEXT** Reserved. Also see extension: **SPV_EXT_physical_storage_buffer**
5356	**AliasedPointerEXT**		**PhysicalStorageBufferAddressesEXT** Reserved. Also see extension: **SPV_EXT_physical_storage_buffer**

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.

BuiltIn		Enabling Capabilities
0	**Position** Output vertex position from a vertex processing Execution Model. See the client API specification for more detail.	**Shader**

BuiltIn		Enabling Capabilities
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**
8	**InvocationId** Invocation ID, input to **Geometry** and **TessellationControl** Execution Model. See the client API specification for more detail.	**Geometry**, **Tessellation**
9	**Layer** Layer output by a **Geometry** Execution Model, input to a **Fragment** Execution Model, for multi-layer framebuffer. See the client API specification for more detail.	**Geometry**
10	**ViewportIndex** Viewport Index output by a **Geometry** stage, input to a **Fragment** Execution Model. See the client API specification for more detail.	**MultiViewport**
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**

BuiltIn		Enabling Capabilities
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**
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** 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**
42	**VertexIndex** Vertex index. See the client API specification for more detail.	**Shader**

BuiltIn		Enabling Capabilities
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**.
4417	**SubgroupGeMask** Subgroup invocations bitmask where bit index >= **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4418	**SubgroupGtMask** Subgroup invocations bitmask where bit index > **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4419	**SubgroupLeMask** Subgroup invocations bitmask where bit index <= **SubgroupLocalInvocationId**. See the client API specification for more detail.	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**.
4420	**SubgroupLtMask** 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	**SubgroupGeMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
4418	**SubgroupGtMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**

BuiltIn		Enabling Capabilities
4419	**SubgroupLeMaskKHR**	**SubgroupBallotKHR**, **GroupNonUniformBallot** Missing before **version 1.3**. Also see extension: **SPV_KHR_shader_ballot**
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**
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**
4992	**BaryCoordNoPerspAMD**	Reserved. Also see extension: **SPV_AMD_shader_explicit_vertex_parameter**

BuiltIn		Enabling Capabilities
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**
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**

BuiltIn		Enabling Capabilities
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**
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**

BuiltIn		Enabling Capabilities
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	**BaryCoordNV**	**FragmentBarycentricNV** Reserved. Also see extension: **SPV_NV_fragment_shader_barycentric**
5287	**BaryCoordNoPerspNV**	**FragmentBarycentricNV** Reserved. Also see extension: **SPV_NV_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**

BuiltIn		Enabling Capabilities
5293	**FragInvocationCountEXT**	**FragmentDensityEXT**, **ShadingRateNV** Reserved. Also see extensions: **SPV_EXT_fragment_invocation_density**, **SPV_NV_shading_rate**
5293	**InvocationsPerPixelNV**	**ShadingRateNV**, **FragmentDensityEXT** Reserved. Also see extensions: **SPV_NV_shading_rate**, **SPV_EXT_fragment_invocation_density**
5319	**LaunchIdNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5320	**LaunchSizeNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5321	**WorldRayOriginNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5322	**WorldRayDirectionNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5323	**ObjectRayOriginNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5324	**ObjectRayDirectionNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5325	**RayTminNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5326	**RayTmaxNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5327	**InstanceCustomIndexNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**

BuiltIn		Enabling Capabilities
5330	**ObjectToWorldNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5331	**WorldToObjectNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5332	**HitTNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5333	**HitKindNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**
5351	**IncomingRayFlagsNV**	**RayTracingNV** Also see extension: **SPV_NV_ray_tracing**

3.22 Selection Control

This value 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

Loop controls. Bits that are set can 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.

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

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, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned.	Missing before **version 1.1**.
0x10	**MinIterations** Unchecked assertion that the loop will execute at least a given number of iterations, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned.	Missing before **version 1.4**.
0x20	**MaxIterations** Unchecked assertion that the loop will execute at most a given number of iterations, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned.	Missing before **version 1.4**.
0x40	**IterationMultiple** Unchecked assertion that the loop will execute a multiple of a given number of iterations, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned. It must also be greater than 0.	Missing before **version 1.4**.
0x80	**PeelCount** Request that the loop be peeled by a given number of loop iterations, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned. 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, specified as a subsequent literal-number operand to the instruction. The literal number is treated as unsigned. This must not be used with the **DontUnroll** bit.	Missing before **version 1.4**.

3.24 Function Control

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

Used by OpFunction.

Function Control	
0x0	**None**
0x1	**Inline** Strong request, to the extent possible, to inline the function.
0x2	**DontInline** Strong request, to the extent possible, to not inline the function.
0x4	**Pure** Compiler can assume this function has no side effect, but might read global memory or read through dereferenced function parameters. Always computes the same result for the same argument values.
0x8	**Const** Compiler can assume this function has no side effects, and will not access global memory or dereference function parameters. Always computes the same result for the same argument values.

3.25 Memory Semantics <id>

Must be an <*id*> of a 32-bit integer scalar.

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

Despite being a mask and allowing multiple bits to be combined, it is invalid for more than one of these four bits to be set: **Acquire**, **Release**, **AcquireRelease**, or **SequentiallyConsistent**. Requesting both **Acquire** and **Release** semantics is done by setting the **AcquireRelease** bit, not by setting two bits.

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

Used by:

- OpControlBarrier
- OpMemoryBarrier
- OpAtomicLoad
- OpAtomicStore
- OpAtomicExchange
- OpAtomicCompareExchange
- OpAtomicCompareExchangeWeak
- OpAtomicIIncrement
- OpAtomicIDecrement
- OpAtomicIAdd
- OpAtomicISub
- OpAtomicSMin
- OpAtomicUMin

- OpAtomicSMax
- OpAtomicUMax
- OpAtomicAnd
- OpAtomicOr
- OpAtomicXor
- OpAtomicFlagTestAndSet
- OpAtomicFlagClear
- OpMemoryNamedBarrier

Memory Semantics		Enabling Capabilities
0x0	**None (Relaxed)**	
0x2	**Acquire** All memory operations provided in program order after this memory operation will execute after this memory operation.	
0x4	**Release** All memory operations provided in program order before this memory operation will execute before this memory operation.	
0x8	**AcquireRelease** Has the properties of both Acquire and Release semantics. It is used for read-modify-write operations.	
0x10	**SequentiallyConsistent** All observers will see this memory access in the same order with respect to other sequentially-consistent memory accesses from this invocation.	
0x40	**UniformMemory** Apply the memory-ordering constraints to **StorageBuffer** 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	**OutputMemoryKHR**	**VulkanMemoryModelKHR**

Memory Semantics		Enabling Capabilities
0x2000	**MakeAvailableKHR**	**VulkanMemoryModelKHR**
0x4000	**MakeVisibleKHR**	**VulkanMemoryModelKHR**

3.26 Memory Operands

Additional operands to the listed memory instructions. Bits that are set can 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.

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

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.	
0x2	**Aligned** This access has a known alignment, provided as a literal in the next operand.	
0x4	**Nontemporal** Hints that the accessed address is not likely to be accessed again in the near future.	
0x8	**MakePointerAvailableKHR**	**VulkanMemoryModelKHR**
0x10	**MakePointerVisibleKHR**	**VulkanMemoryModelKHR**
0x20	**NonPrivatePointerKHR**	**VulkanMemoryModelKHR**

3.27 Scope <id>

Must be an <*id*> of a 32-bit integer scalar. Its value must be one of the values in the table below.

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

- OpControlBarrier
- OpMemoryBarrier
- OpAtomicLoad
- OpAtomicStore

- OpAtomicExchange
- OpAtomicCompareExchange
- OpAtomicCompareExchangeWeak
- OpAtomicIIncrement
- OpAtomicIDecrement
- OpAtomicIAdd
- OpAtomicISub
- OpAtomicSMin
- OpAtomicUMin
- OpAtomicSMax
- OpAtomicUMax
- OpAtomicAnd
- OpAtomicOr
- OpAtomicXor
- OpGroupAsyncCopy
- OpGroupWaitEvents
- OpGroupAll
- OpGroupAny
- OpGroupBroadcast
- OpGroupIAdd
- OpGroupFAdd
- OpGroupFMin
- OpGroupUMin
- OpGroupSMin
- OpGroupFMax
- OpGroupUMax
- OpGroupSMax
- OpGroupReserveReadPipePackets
- OpGroupReserveWritePipePackets
- OpGroupCommitReadPipe
- OpGroupCommitWritePipe
- OpAtomicFlagTestAndSet
- OpAtomicFlagClear
- OpMemoryNamedBarrier
- 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
- OpTypeCooperativeMatrixNV

Scope		Enabling Capabilities
0	**CrossDevice** Scope crosses multiple devices.	
1	**Device** Scope is the current device.	

	Scope	Enabling Capabilities
2	**Workgroup** Scope is the current workgroup.	
3	**Subgroup** Scope is the current subgroup.	
4	**Invocation** Scope is the current Invocation.	
5	**QueueFamilyKHR**	**VulkanMemoryModelKHR**

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

3.30 Kernel Profiling Info

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

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

Kernel Profiling Info		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 must be declared, either explicitly with OpCapability or implicitly through the **Implicitly Declares** column. 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**

Capability		Implicitly Declares
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 the sole purpose of creating an OpTypePointer to a 16-bit float. Pointers to a 16-bit float cannot be dereferenced directly, they must only be dereferenced via an extended 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.	**ImageBasic**
15	**ImageMipmap** Uses non-zero **Lod** Image Operands.	**ImageBasic**
17	**Pipes** Uses OpTypePipe, OpTypeReserveId or pipe instructions.	**Kernel**
18	**Groups** Uses group instructions.	
19	**DeviceEnqueue** Uses OpTypeQueue, OpTypeDeviceEvent, and device side enqueue instructions.	**Kernel**
20	**LiteralSampler** Samplers are made from literals within the module. See OpConstantSampler.	**Kernel**

	Capability	Implicitly Declares
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** Uses multi-sample images for non-sampled images.	**Shader**
28	**UniformBufferArrayDynamicIndexing** **Block**-decorated arrays in uniform storage classes use dynamically uniform indexing.	**Shader**
29	**SampledImageArrayDynamicIndexing** Arrays of sampled images use dynamically uniform indexing.	**Shader**
30	**StorageBufferArrayDynamicIndexing** Arrays in the **StorageBuffer** Storage Class, or **BufferBlock**-decorated arrays, use dynamically uniform indexing.	**Shader**
31	**StorageImageArrayDynamicIndexing** Arrays of non-sampled images are accessed with dynamically uniform indexing.	**Shader**
32	**ClipDistance** Uses the **ClipDistance** BuiltIn.	**Shader**
33	**CullDistance** Uses the **CullDistance** BuiltIn.	**Shader**
34	**ImageCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in OpTypeImage, without a sampler.	**SampledCubeArray**
35	**SampleRateShading** Uses per-sample rate shading.	**Shader**
36	**ImageRect** Uses the **Rect** Dim without a sampler.	**SampledRect**
37	**SampledRect** Uses the **Rect** Dim with a sampler.	**Shader**
38	**GenericPointer** Uses the **Generic** Storage Class.	**Addresses**
39	**Int8** Uses OpTypeInt to declare 8-bit integer types.	
40	**InputAttachment** Uses the **SubpassData** Dim.	**Shader**
41	**SparseResidency** Uses **OpImageSparse…** instructions.	**Shader**

Capability		Implicitly Declares
42	**MinLod** Uses the **MinLod** Image Operand.	**Shader**
43	**Sampled1D** Uses the **1D** Dim with a sampler.	
44	**Image1D** Uses the **1D** Dim without a sampler.	**Sampled1D**
45	**SampledCubeArray** Uses the **Cube** Dim with the *Arrayed* operand in OpTypeImage, with a sampler.	**Shader**
46	**SampledBuffer** Uses the **Buffer** Dim with a sampler.	
47	**ImageBuffer** Uses the **Buffer** Dim without a sampler.	**SampledBuffer**
48	**ImageMSArray** An *MS* operand in OpTypeImage indicates multisampled, used without a sampler.	**Shader**
49	**StorageImageExtendedFormats** One of a large set of more advanced image formats are used, namely one of those in the Image Format table listed as requiring this capability.	**Shader**
50	**ImageQuery** The sizes, number of samples, or lod, etc. are queried.	**Shader**
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**.

Capability		Implicitly Declares
60	**PipeStorage** Uses OpTypePipeStorage.	**Pipes** Missing before **version 1.1**.
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**.
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**
4431	**SubgroupVoteKHR**	Reserved. Also see extension: **SPV_KHR_subgroup_vote**

	Capability	Implicitly Declares
4433	**StorageBuffer16BitAccess** Allows 16-bit OpTypeFloat and OpTypeInt for the sole purpose of creating an OpTypePointer to a 16-bit floating-point or 16-bit integer member of an object. The object must be in the **StorageBuffer** Storage Class, or be in the **Uniform** storage class and have the **BufferBlock** decoration. An object of a 16-bit type produced by dereferencing such a pointer may be the result of a width-only conversion instruction (OpFConvert, OpSConvert, or OpUConvert) from a 32-bit type or of an OpLoad, and may be used as an operand to a width-only conversion instruction to a 32-bit type or as the object operand of an OpStore. Other uses of 16-bit types are not enabled by this capability.	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** Allows 16-bit OpTypeFloat and OpTypeInt for the sole purpose of creating an OpTypePointer to a 16-bit floating-point or 16-bit integer member of an object. The object must be in the **StorageBuffer** or **Uniform** Storage Classes. An object of a 16-bit type produced by dereferencing such a pointer may be the result of a width-only conversion instruction from a 32-bit type or of an OpLoad, and may be used as an operand to a width-only conversion instruction to a 32-bit type or as the object operand of an OpStore. Other uses of 16-bit types are not enabled by this capability.	**StorageBuffer16BitAccess**, **StorageUniformBufferBlock16** Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4434	**StorageUniform16**	**StorageBuffer16BitAccess**, **StorageUniformBufferBlock16** Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**

	Capability	Implicitly Declares
4435	**StoragePushConstant16** Allows 16-bit OpTypeFloat and OpTypeInt for the sole purpose of creating an OpTypePointer to a 16-bit floating-point or 16-bit integer object in the **PushConstant** Storage Class. An object of a 16-bit type produced by dereferencing such a pointer may only be the result of a width-only conversion instruction from a 32-bit type or of an OpLoad. Other uses of 16-bit types are not enabled by this capability.	Missing before **version 1.3**. Also see extension: **SPV_KHR_16bit_storage**
4436	**StorageInputOutput16** Allows 16-bit OpTypeFloat and OpTypeInt for the sole purpose of creating an OpTypePointer to a 16-bit floating-point or 16-bit integer object in the **Input** or **Output** Storage Classes. An object of a 16-bit type produced by dereferencing such a pointer may only be the result of a width-only conversion instruction from a 32-bit type or of an OpLoad, and may be used as an operand to a width-only conversion instruction to a 32-bit type or as the object operand of an OpStore. Other uses of 16-bit types are not enabled by this capability.	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**

Capability		Implicitly Declares
4445	**AtomicStorageOps**	Reserved. Also see extension: **SPV_KHR_shader_atomic_counter_ops**
4447	**SampleMaskPostDepthCoverage**	Reserved. Also see extension: **SPV_KHR_post_depth_coverage**
4448	**StorageBuffer8BitAccess**	Reserved. Also see extension: **SPV_KHR_8bit_storage**
4449	**UniformAndStorageBuffer8BitAccess**	**StorageBuffer8BitAccess** Reserved. Also see extension: **SPV_KHR_8bit_storage**
4450	**StoragePushConstant8**	Reserved. 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**
5008	**Float16ImageAMD**	**Shader** Reserved. Also see extension: **SPV_AMD_gpu_shader_half_float_fetch**

Capability		Implicitly Declares
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**
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**

Capability		Implicitly Declares
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**
5265	**FragmentFullyCoveredEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_fragment_fully_covered**
5266	**MeshShadingNV**	**Shader** Reserved. Also see extension: **SPV_NV_mesh_shader**
5301	**ShaderNonUniformEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5302	**RuntimeDescriptorArrayEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5303	**InputAttachmentArrayDynamicIndexingEXT**	**InputAttachment** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**

Capability		Implicitly Declares
5304	**UniformTexelBufferArrayDynamicIndexingEXT**	**SampledBuffer** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5305	**StorageTexelBufferArrayDynamicIndexingEXT**	**ImageBuffer** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5306	**UniformBufferArrayNonUniformIndexingEXT**	**ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5307	**SampledImageArrayNonUniformIndexingEXT**	**ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5308	**StorageBufferArrayNonUniformIndexingEXT**	**ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5309	**StorageImageArrayNonUniformIndexingEXT**	**ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5310	**InputAttachmentArrayNonUniformIndexingEXT**	**InputAttachment**, **ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5311	**UniformTexelBufferArrayNonUniformIndexingEXT**	**SampledBuffer**, **ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**

Capability		Implicitly Declares
5312	**StorageTexelBufferArrayNonUniformIndexingEXT**	**ImageBuffer**, **ShaderNonUniformEXT** Reserved. Also see extension: **SPV_EXT_descriptor_indexing**
5340	**RayTracingNV**	**Shader** Reserved. Also see extension: **SPV_NV_ray_tracing**
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**
5696	**SubgroupAvcMotionEstimationINTEL**	Reserved. Also see extension: **SPV_INTEL_device_side_avc_motion_estimation**
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**
5297	**GroupNonUniformPartitionedNV**	Reserved. Also see extension: **SPV_NV_shader_subgroup_partitioned**
5345	**VulkanMemoryModelKHR**	Reserved. Also see extension: **SPV_KHR_vulkan_memory_model**
5346	**VulkanMemoryModelDeviceScopeKHR**	Reserved. Also see extension: **SPV_KHR_vulkan_memory_model**

Capability		Implicitly Declares
5282	**ImageFootprintNV**	Reserved. Also see extension: **SPV_NV_shader_image_footprint**
5284	**FragmentBarycentricNV**	Reserved. Also see extension: **SPV_NV_fragment_shader_barycentric**
5288	**ComputeDerivativeGroupQuadsNV**	Reserved. Also see extension: **SPV_NV_compute_shader_derivatives**
5350	**ComputeDerivativeGroupLinearNV**	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**
5347	**PhysicalStorageBufferAddressesEXT**	**Shader** Reserved. Also see extension: **SPV_EXT_physical_storage_buffer**
5357	**CooperativeMatrixNV**	**Shader** Reserved. Also see extension: **SPV_NV_cooperative_matrix**

3.32 Instructions

Form for each instruction:

<table>
<tr><td colspan="3">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 will also say "+ variable", after stating the minimum size of the instruction.

Opcode is the low-order 16 bits of word 0 of the instruction, holding its opcode enumerant.

Results, when present, are any Result <id> or Result Type created by the instruction. Each one is always 32 bits.

Operands, when present, are any literals, other instruction’s Result <id>, etc., consumed by the instruction. Each one is always 32 bits.</td><td>Capability
Enabling Capabilities
(when needed)</td></tr>
<tr><td>Word Count</td><td>Opcode</td><td>Results</td><td>Operands</td></tr>
</table>

3.32.1 Miscellaneous Instructions

<table>
<tr><td colspan="2">OpNop

This has no semantic impact and can safely be removed from a module.</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>

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

Make an intermediate object whose value is undefined.

Result Type is the type of object to make.

Each consumption of Result <id> yields an arbitrary, possibly different bit pattern or abstract value resulting in possibly different concrete, abstract, or opaque values.</td></tr>
<tr><td>3</td><td>1</td><td><id>
Result Type</td><td>Result <id></td></tr>
</table>

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*

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*

OpWritePackedPrimitiveIndices4x8NV TBD			Capability: **MeshShadingNV** Reserved.
3	5299	<id> *Index Offset*	<id> *Packed Indices*

OpReportIntersectionNV TBD					Capability: **RayTracingNV**
5	5334	<id> *Result Type*	Result <id>	<id> *Hit*	<id> *HitKind*

OpIgnoreIntersectionNV TBD	Capability: **RayTracingNV**
1	5335

OpTerminateRayNV TBD	Capability: **RayTracingNV**
1	5336

OpTraceNV TBD										Capability: **RayTracingNV**		
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*

OpExecuteCallableNV TBD			Capability: **RayTracingNV**
3	5344	<id> *SBT Index*	<id> *Callable DataId*

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*

3.32.2 Debug Instructions

OpSourceContinued

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

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

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

2 + variable	2	Literal String *Continued Source*

OpSource

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

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

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

Source is the text of the source-level file.

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

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

OpSourceExtension

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

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

2 + variable	4	Literal String *Extension*

OpName

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

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

Name is the string to assign.

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

OpMemberName

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

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

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

Name is the string to assign to the member.

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

OpString

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

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

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

OpLine				
Add source-level location information. This has no semantic impact and can safely be removed from a module. This location information applies to the instructions physically following this instruction, up to the first occurrence of any of the following: the next end of block, the next **OpLine** instruction, or the next OpNoLine instruction. *File* must be an OpString instruction and is the source-level file name. *Line* is the source-level line number. This literal operand is limited to a single word. *Column* is the source-level column number. This literal operand is limited to a single word. **OpLine** can generally immediately precede other instructions, with the following exceptions: - it may not be used until after the annotation instructions, (see the Logical Layout section) - cannot be the last instruction in a block, which is defined to end with a 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 Number *Line*	Literal Number *Column*

OpNoLine	
Discontinue any source-level location information that might be active from a previous OpLine instruction. This has no semantic impact and can safely be removed from a module. This instruction can only appear after the annotation instructions (see the Logical Layout section). It cannot be the last instruction in a block, or the second-to-last instruction if the block has a merge instruction. There is not a requirement that there is a preceding **OpLine** instruction.	
1	317

OpModuleProcessed Document a process that was applied to a module. This has no semantic impact and can safely be removed from a module. *Process* is a string describing a process and/or tool (processor) that did the processing. Its form is dependent on the processor.		Missing before **version 1.1**.
2 + variable	330	Literal String *Process*

3.32.3 Annotation Instructions

OpDecorate

Add a Decoration to another <*id*>.

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

This instruction is only valid when 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 Number *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 when 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 String* operand, and has only *Literal String* operands.					Missing before **version 1.4**.
4 + variable	5632	*<id>* *Target*	Decoration	Literal String See Decoration.	*Optional Literal Strings* See Decoration.

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. The first member is member 0, the next is member 1, . . . *Decoration* is a decoration that takes at least one *Literal String* operand, and has only *Literal String* operands.					Missing before **version 1.4**.	
5 + variable	5633	<*id*> *Struct Type*	Literal Number *Member*	Decoration	Literal String See Decoration.	*Optional Literal Strings* See Decoration.

3.32.4 Extension Instructions

OpExtension Declare use of an extension to SPIR-V. This allows validation of additional instructions, tokens, semantics, etc. *Name* is the extension's name string.		
2 + variable	10	Literal String *Name*

OpExtInstImport Import an extended set of instructions. It can be later referenced by the *Result <id>*. *Name* is the extended instruction-set's name string. There must be an external specification defining the semantics for this extended instruction set. See Extended Instruction Sets for more information.			
3 + variable	11	Result <id>	Literal String *Name*

OpExtInst Execute an instruction in an imported set of extended instructions. *Result Type* is as defined, per *Instruction*, in the external specification for *Set*. *Set* is the result of an OpExtInstImport instruction. *Instruction* is the enumerant of the instruction to execute within *Set*. This literal operand is limited to a single word. The semantics of the instruction must be defined in the external specification for *Set*. *Operand 1, …* are the operands to the extended instruction.						
5 + variable	12	*<id>* *Result Type*	Result <id>	*<id>* *Set*	Literal Number *Instruction*	*<id>, <id>, …* *Operand 1, Operand 2, …*

3.32.5 Mode-Setting Instructions

OpMemoryModel Set addressing model and memory model for the entire module. *Addressing Model* selects the module's Addressing Model. *Memory Model* selects the module's memory model, see Memory Model.			
3	14	Addressing Model	Memory Model

OpEntryPoint Declare an entry point, 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 cannot have two **OpEntryPoint** instructions with the same Execution Model and the same *Name* string. *Interface* is a list of <*id*> of global OpVariable instructions. 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 String *Name*	*<id>, <id>, …* *Interface*

OpExecutionMode Declare an execution mode for an entry point. *Entry Point* must be the *Entry Point* <*id*> operand of an OpEntryPoint instruction. *Mode* is the execution mode. See Execution Mode. This instruction is only valid when 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 <<i>id</i>><i>s</i> as Extra Operands.

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

<i>Mode</i> is the execution mode. See Execution Mode.

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

3.32.6 Type-Declaration Instructions

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

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

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

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

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

<table>
<tr><td colspan="4">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. 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).</td><td>Capability:
Matrix</td></tr>
<tr><td>4</td><td>24</td><td>Result <id></td><td><id>
Column Type</td><td>Literal Number
Column Count</td></tr>
</table>

OpTypeImage

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

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

Dim is the image dimensionality (Dim).

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

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

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

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

Image Format is the Image Format, which can be **Unknown**, 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 Number *Depth*	Literal Number *Arrayed*	Literal Number *MS*	Literal Number *Sampled*	Image Format	Optional Access Qualifier

OpTypeSampler

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

2	26	Result <id>

OpTypeSampledImage Declare a sampled image type, the *Result Type* of OpSampledImage, or an externally combined sampler and image. This type is opaque: values of this type have no defined physical size or bit pattern. *Image Type* must be an OpTypeImage. It is the type of the image in the combined sampler and image type.			
3	27	Result <id>	*<id>* *Image Type*

OpTypeArray Declare a new array type: a dynamically-indexable ordered aggregate of elements all having the same type. *Element Type* is the type of each element in the array. *Length* is the number of elements in the array. It must be at least 1. *Length* must come from a constant instruction of an integer-type scalar whose value is at least 1. Array elements are number consecutively, starting with 0.				
4	28	Result <id>	*<id>* *Element Type*	*<id>* *Length*

OpTypeRuntimeArray Declare a new run-time array type. Its length is not known at compile time. *Element Type* is the type of each element in the array. It must be a concrete type. See OpArrayLength for getting the *Length* of an array of this type.			Capability: **Shader**
3	29	Result <id>	*<id>* *Element Type*

OpTypeStruct Declare a new structure type: an aggregate of zero or more potentially heterogeneous members. *Member N type* is the type of member *N* of the structure. The first member is member 0, the next is member 1, . . . If an operand is not yet defined, it must be defined by an OpTypePointer, where the type pointed to is an **OpTypeStruct**.			
2 + variable	30	Result <id>	*<id>, <id>, . . .* *Member 0 type*, *member 1 type*, *. . .*

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

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

OpTypeFunction Declare a new function type. OpFunction will use this to declare the return type and parameter types of a function. **OpFunction** is the only valid use of **OpTypeFunction**. *Return Type* is the type of the return value of functions of this type. It must be a concrete or abstract type, or a pointer to such a type. If the function has no return value, *Return Type* must be OpTypeVoid. *Parameter N Type* is the type *<id>* of the type of parameter *N*. 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. The type of object the pointer points to is declared by the **OpTypePointer** instruction, not this instruction. Subsequent OpTypeStruct instructions can use *Pointer Type* as an operand. *Storage Class* is the Storage Class of the memory holding the object pointed to.			Capability: **Addresses, PhysicalStorageBufferAddressesEXT**
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>

OpTypeAccelerationStructureNV TBD		Capability: **RayTracingNV**
2	5341	Result <id>

OpTypeCooperativeMatrixNV TBD					Capability: **CooperativeMatrixNV** Reserved.	
6	5358	Result <id>	*<id>* *Component Type*	Scope <id> *Execution*	*<id>* *Rows*	*<id>* *Columns*

3.32.7 Constant-Creation Instructions

OpConstantTrue

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

Result Type must be the scalar Boolean type.

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

OpConstantFalse

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

Result Type must be the scalar Boolean type.

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

OpConstant

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

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

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

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

OpConstantComposite

Declare a new composite constant.

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

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

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

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

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

OpSpecConstantTrue Declare a Boolean-type scalar specialization constant with a default value of **true**. This instruction can be specialized to become either an OpConstantTrue or OpConstantFalse instruction. *Result Type* must be the scalar Boolean type. See Specialization.			
3	48	*<id>* *Result Type*	Result <id>

OpSpecConstantFalse Declare a Boolean-type scalar specialization constant with a default value of **false**. This instruction can be specialized to become either an OpConstantTrue or OpConstantFalse instruction. *Result Type* must be the scalar Boolean type. See Specialization.			
3	49	*<id>* *Result Type*	Result <id>

OpSpecConstant Declare a new integer-type or floating-point-type scalar specialization constant. *Result Type* must be a scalar integer type or floating-point type. *Value* is the bit pattern for the default value of the constant. Types 32 bits wide or smaller take one word. Larger types take multiple words, with low-order words appearing first. This instruction can be specialized to become an OpConstant instruction. See Specialization.				
3 + variable	50	*<id>* *Result Type*	Result <id>	*Literal, Literal, ...* *Value*

OpSpecConstantComposite Declare a new composite specialization constant. *Result Type* must be a composite type, whose top-level members/elements/components/columns have the same type as the types of the *Constituents*. The ordering must be the same between the top-level types in *Result Type* and the *Constituents*. *Constituents* will become members of a structure, or elements of an array, or components of a vector, or columns of a matrix. There must be exactly one *Constituent* for each top-level member/element/component/column of the result. The *Constituents* must appear in the order needed by the definition of the type of the result. The *Constituents* must be the <*id*> of other specialization constant or constant declarations. This instruction will be specialized to an OpConstantComposite instruction. See Specialization.				
3 + variable	51	*<id>* *Result Type*	Result <id>	*<id>, <id>, …* *Constituents*

OpSpecConstantOp

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

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

Opcode must be one of the following opcodes. This literal operand is limited to a single word.
OpSConvert, **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, the following opcode is also valid:
OpQuantizeToF16

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

Operands are the operands required by *opcode*, and satisfy the semantics of *opcode*. In addition, all *Operands* must be 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 Number *Opcode*	*<id>, <id>, ...* *Operands*

3.32.8 Memory Instructions

OpVariable

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

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

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

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

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

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

If *Arrayed* is 1:
1D: 2 components
2D: 3 components
Cube: 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. It must be a valid <*id*> for the value 0 if the OpTypeImage has *MS* of 0.

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

OpLoad Load through a pointer. *Result Type* is the type of the loaded object. It must be a type with fixed size; i.e., it cannot 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 cannot 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 the second applies to *Source*.					
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 cannot be 0. It is invalid for both the constant's type to have *Signedness* of 1 and to have the sign bit set. Otherwise, as a run-time value, *Size* is treated as unsigned, and if its value is 0, no memory access will be made.

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 the second applies to *Source*.

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 that can be used with OpLoad and OpStore.

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

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

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

Each index in *Indexes*
- must be a scalar integer type,
- is treated as a signed count, and
- must be an OpConstant when indexing into a structure.

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

OpInBoundsAccessChain

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

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

<table>
<tr><td colspan="5">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 when the Base type is decorated with ArrayStride. For all other objects, the implementation will calculate 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 that the result is still well defined 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 will select the array element.</td><td colspan="2">Capability:
Addresses, VariablePointers, VariablePointersStorageBuffer, PhysicalStorageBufferAddress-esEXT</td></tr>
<tr><td>5 + variable</td><td>67</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Base</td><td><id>
Element</td><td><id>, <id>, ...
Indexes</td></tr>
</table>

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

Length of a run-time array.

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

Structure must be a pointer to an OpTypeStruct whose last member is a run-time array.

Array member is the index of the last member of the structure that Structure points to. That member's type must be from OpTypeRuntimeArray.</td><td>Capability:
Shader</td></tr>
<tr><td>5</td><td>68</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Structure</td><td>Literal Number
Array member</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>69</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td></tr>
</table>

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

Has the same semantics as OpPtrAccessChain, with the addition that the resulting pointer is known to point within the base object.</td><td colspan="2">Capability:
Addresses</td></tr>
<tr><td>5 + variable</td><td>70</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Base</td><td><id>
Element</td><td><id>, <id>, …
Indexes</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td>Missing before version 1.4.</td></tr>
<tr><td>5</td><td>401</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Operand 1</td><td><id>
Operand 2</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td>Missing before version 1.4.</td></tr>
<tr><td>5</td><td>402</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Operand 1</td><td><id>
Operand 2</td></tr>
</table>

<table>
<tr><td colspan="5">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 will be computed as a signed value, as negative differences are allowed, independently of the signed bit in the type. The result will equal 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.</td><td>Capability:
Addresses,
VariablePointers,
VariablePointer-sStorageBuffer

Missing before version 1.4.</td></tr>
<tr><td>5</td><td>403</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Operand 1</td><td><id>
Operand 2</td></tr>
</table>

3.32.9 Function Instructions

OpFunction

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

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

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

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

OpFunctionParameter

Declare a formal parameter of the current function.

Result Type is the type of the parameter.

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

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

OpFunctionEnd

Last instruction of a function.

1	56

OpFunctionCall Call a function. *Result Type* is the type of the return value of the function. It must be the same as the *Return Type* operand of the *Function Type* operand of the *Function* operand. *Function* is an OpFunction instruction. This could be a forward reference. *Argument N* is the object to copy to parameter *N* of *Function*. **Note:** A forward call is possible because there is no missing type information: *Result Type* must match the *Return Type* of the function, and the calling argument types must match the formal parameter types.					
4 + variable	57	*<id>* *Result Type*	Result <id>	*<id>* *Function*	*<id>, <id>, …* *Argument 0*, *Argument 1*, …

3.32.10 Image Instructions

OpSampledImage Create a sampled image, containing both a sampler and an image. *Result Type* must be the OpTypeSampledImage type 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**. *Sampler* must be an object whose type is OpTypeSampler.					
5	86	*<id>* *Result Type*	Result <id>	*<id>* *Image*	*<id>* *Sampler*

OpImageSampleImplicitLod Sample an image with an implicit level of detail. *Result Type* must be a vector of four components of floating-point type or integer type. Its components must be the same as *Sampled Type* of the underlying OpTypeImage (unless that underlying *Sampled Type* is **OpTypeVoid**). *Sampled Image* must be an object whose type is OpTypeSampledImage. *Coordinate* must be a scalar or vector of floating-point type. It contains (*u*[, *v*] ... [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components. *Image Operands* encodes what operands follow, as per Image Operands. This instruction is only valid in the **Fragment** Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.						Capability: **Shader**	
5 + variable	87	*<id>* *Result Type*	Result <id>	*<id>* *Sampled Image*	*<id>* *Coordinate*	Optional Image Operands	Optional *<id>, <id>*, ...

OpImageSampleExplicitLod

Sample an image using an explicit level of detail.

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

Sampled Image must be an object whose type is OpTypeSampledImage.

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

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

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

OpImageSampleDrefImplicitLod

Capability:
Shader

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

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

Sampled Image must be an object whose type is OpTypeSampledImage.

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

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

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

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

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

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

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

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

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.</td><td colspan="2">Capability:
Shader</td></tr>
<tr><td>8 + variable</td><td>90</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td><<i>id</i>>
<i>Sampled Image</i></td><td><<i>id</i>>
<i>Coordinate</i></td><td><<i>id</i>>
D_{ref}</td><td>Image Operands</td><td><<i>id</i>></td><td>Optional <<i>id</i>>, <<i>id</i>>, …</td></tr>
</table>

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

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

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

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

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

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

This instruction is only valid in the Fragment Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.</td>
<td colspan="2">Capability:
Shader</td>
</tr>
<tr>
<td>5 + variable</td>
<td>91</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Sampled Image</td>
<td><id>
Coordinate</td>
<td>Optional
Image Operands</td>
<td>Optional
<id>, <id>, ...</td>
</tr>
</table>

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

Sample an image with a project coordinate using an explicit level of detail.

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

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

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

Image Operands encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.</td>
<td colspan="2">Capability:
Shader</td>
</tr>
<tr>
<td>7 + variable</td>
<td>92</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Sampled Image</td>
<td><id>
Coordinate</td>
<td>Image Operands</td>
<td><id></td>
<td>Optional <id>, <id>, …</td>
</tr>
</table>

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

OpImageSampleProjDrefImplicitLod

Sample an image with a project coordinate, doing depth-comparison, with an implicit level of detail.

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

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

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

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

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

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

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

Capability:
Shader

</td>
</tr>
<tr>
<td>6 +
variable</td>
<td>93</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>

OpImageSampleProjDrefExplicitLod

Capability:
Shader

Sample an image with a project coordinate, doing depth-comparison, using an explicit level of detail.

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

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

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

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

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

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 cannot be **Cube**, and its *Sampled* operand must be 1.

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

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

5 + variable	95	<*id*> *Result Type*	Result <id>	<*id*> *Image*	<*id*> *Coordinate*	Optional Image Operands	Optional <*id*>, <*id*>, ...

OpImageGather

Gathers the requested component from four texels.

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

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

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

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

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

Capability:
Shader

6 + variable	96	<*id*> *Result Type*	Result <id>	<*id*> *Sampled Image*	<*id*> *Coordinate*	<*id*> *Component*	Optional Image Operands	Optional <*id*>, <*id*>, …

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**.

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

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

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

Capability:
Shader

6 + variable	97	<*id*> *Result Type*	Result <id>	<*id*> *Sampled Image*	<*id*> *Coordinate*	<*id*> D_{ref}	Optional Image Operands	Optional <*id*>, <*id*>, …

OpImageRead

Read a texel from an image without a sampler.

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

Image must be an object whose type is OpTypeImage with a *Sampled* operand of 0 or 2. If the *Sampled* operand is 2, then some dimensions require a capability; e.g., **Image1D**, **ImageRect**, or **ImageBuffer**. 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*. If the coordinates are outside the image, the memory location that is accessed is undefined.

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

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

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

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

OpImageWrite

Write a texel to an image without a sampler.

Image must be an object whose type is OpTypeImage with a *Sampled* operand of 0 or 2. If the *Sampled* operand is 2, then some dimensions require a capability; e.g., **Image1D**, **ImageRect**, or **ImageBuffer**. If the *Arrayed* operand is 1, then additional capabilities may be required; e.g., **ImageCubeArray**, or **ImageMSArray**. Its Dim operand cannot be **SubpassData**.

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

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

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

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

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

OpImage Extract the image from a sampled image. *Result Type* must be OpTypeImage. *Sampled Image* must have type OpTypeSampledImage whose *Image Type* is the same as *Result Type*.				
4	100	<id> *Result Type*	Result <id>	<id> *Sampled Image*

OpImageQueryFormat Query the image format of an image created with an **Unknown** Image Format. *Result Type* must be a scalar integer type. The resulting value is an enumerant from Image Channel Data Type. *Image* must be an object whose type is OpTypeImage.				Capability: **Kernel**
4	101	<id> *Result Type*	Result <id>	<id> *Image*

OpImageQueryOrder Query the channel order of an image created with an **Unknown** Image Format. *Result Type* must be a scalar integer type. The resulting value is an enumerant from Image Channel Order. *Image* must be an object whose type is OpTypeImage.				Capability: **Kernel**
4	102	<id> *Result Type*	Result <id>	<id> *Image*

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

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

Result Type must be an integer type scalar or vector. The number of components must be
1 for 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. This operation is allowed on an image decorated as NonReadable. 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>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. This operation is allowed on an image decorated as NonReadable. 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="5">OpImageQueryLod

Query the mipmap level and the level of detail for a hypothetical sampling of Image at Coordinate using an implicit level of detail.

Result Type must be a two-component floating-point type vector.
The first component of the result will contain the mipmap array layer.
The second component of the result will contain the implicit level of detail relative to the base level.

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.

If called on an incomplete image, the results are undefined.

This instruction is only valid in the Fragment Execution Model. In addition, it consumes an implicit derivative that can be affected by code motion.</td><td>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 will hold a Residency Code that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as Sampled Type of the underlying OpTypeImage (unless that underlying Sampled Type is OpTypeVoid).

Sampled Image must be an object whose type is OpTypeSampledImage.

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

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

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

OpImageSparseSampleExplicitLod Sample a sparse image using an explicit level of detail. *Result Type* must be an OpTypeStruct with two members. The first member's type must be an integer type scalar. It will hold a *Residency Code* that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as *Sampled Type* of the underlying OpTypeImage (unless that underlying *Sampled Type* is **OpTypeVoid**). *Sampled Image* must be an object whose type is OpTypeSampledImage. *Coordinate* must be a scalar or vector of floating-point type or integer type. It contains (*u*[, *v*] ... [, *array layer*]) as needed by the definition of *Sampled Image*. Unless the **Kernel** capability is being used, it must be floating point. It may be a vector larger than needed, but all unused components will appear after all used components. *Image Operands* encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.							Capability: **SparseResidency**	
7 + variable	306	*<id>* *Result Type*	Result <id>	*<id>* *Sampled Image*	*<id>* *Coordinate*	Image Operands	*<id>*	Optional *<id>*, *<id>*, ...

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

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

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

Sampled Image must be an object whose type is OpTypeSampledImage.

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

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

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

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

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 will hold a *Residency Code* that can be passed to OpImageSparseTexelsResident. The second member must be a scalar of integer type or floating-point type. It must be the same as *Sampled Type* of the underlying OpTypeImage. *Sampled Image* must be an object whose type is OpTypeSampledImage. *Coordinate* must be a scalar or vector of floating-point type. It contains (*u*[, *v*] ... [, *array layer*]) as needed by the definition of *Sampled Image*. It may be a vector larger than needed, but all unused components will appear after all used components. D_{ref} is the depth-comparison reference value. *Image Operands* encodes what operands follow, as per Image Operands. At least one operand setting the level of detail must be present.								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 will hold a *Residency Code* that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as *Sampled Type* of the underlying OpTypeImage (unless that underlying *Sampled Type* is **OpTypeVoid**). *Image* must be an object whose type is OpTypeImage. Its Dim operand cannot be **Cube**. *Coordinate* is an integer scalar or vector containing (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. *Image Operands* encodes what operands follow, as per Image Operands.	Capability: **SparseResidency**

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

OpImageSparseGather Gathers the requested component from four texels of a sparse image. *Result Type* must be an OpTypeStruct with two members. The first member's type must be an integer type scalar. It will hold a *Residency Code* that can be passed to OpImageSparseTexelsResident. The second member must be a vector of four components of floating-point type or integer type. Its components must be the same as *Sampled Type* of the underlying OpTypeImage (unless that underlying *Sampled Type* is **OpTypeVoid**). It has one component per gathered texel. *Sampled Image* must be an object whose type is OpTypeSampledImage. Its OpTypeImage must have a Dim of **2D**, **Cube**, or **Rect**. *Coordinate* must be a scalar or vector of floating-point type. It contains (*u*[, *v*] … [, *array layer*]) as needed by the definition of *Sampled Image*. *Component* is the component number that will be gathered from all four texels. It must be 0, 1, 2 or 3. *Image Operands* encodes what operands follow, as per Image Operands.	Capability: **SparseResidency**

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

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

Gathers the requested depth-comparison from four texels of a sparse image.

Result Type must be an OpTypeStruct with two members. The first member's type must be an integer type scalar. It will hold a Residency Code that can be passed to OpImageSparseTexelsResident. The second member must be a 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.

Image Operands encodes what operands follow, as per Image Operands.</td><td colspan="2">Capability:
SparseResidency</td></tr>
<tr><td>6 + variable</td><td>315</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td><<i>id</i>>
<i>Sampled Image</i></td><td><<i>id</i>>
<i>Coordinate</i></td><td><<i>id</i>>
D_{ref}</td><td></td><td>Optional Image Operands</td><td>Optional <<i>id</i>>, <<i>id</i>>, …</td></tr>
</table>

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

Translates a Resident Code into a Boolean. Result is false if any of the texels were in uncommitted texture memory, and true otherwise.

Result Type must be a Boolean type scalar.

Resident Code is a value from an OpImageSparse… instruction that returns a resident code.</td><td>Capability:
SparseResidency</td></tr>
<tr><td>4</td><td>316</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td><<i>id</i>>
<i>Resident Code</i></td></tr>
</table>

OpImageSparseRead Read a texel from a sparse image without a sampler. *Result Type* must be an OpTypeStruct with two members. The first member's type must be an integer type scalar. It will hold a *Residency Code* that can be passed to OpImageSparseTexelsResident. The second member must be a scalar or vector of floating-point type or integer type. Its component type must be the same as *Sampled Type* of the OpTypeImage (unless that *Sampled Type* is **OpTypeVoid**). *Image* must be an object whose type is OpTypeImage with a *Sampled* operand of 2. *Coordinate* is an integer scalar or vector containing non-normalized texel coordinates (*u*[, *v*] ... [, *array layer*]) as needed by the definition of *Image*. If the coordinates are outside the image, the memory location that is accessed is undefined. The *Image* 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*>, ...

3.32.11 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. *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. *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 cannot equal the component width in *Result Type*.

Results are computed per component.

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

OpSConvert

Convert signed width. This is either a truncate or a sign extend.

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

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

Results are computed per component.

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

OpFConvert

Convert value 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 cannot equal the component width in *Result Type*.

Results are computed per component.

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

<table>
<tr><td colspan="4">OpQuantizeToF16

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.</td><td>Capability:
Shader</td></tr>
<tr><td>4</td><td>116</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Value</td></tr>
</table>

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

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 an OpTypePointer. If the bit width of Pointer is smaller than that of Result Type, the conversion will zero extend Pointer. If the bit width of Pointer is larger than that of Result Type, the conversion will truncate Pointer. For same bit width Pointer and Result Type, this is the same as OpBitcast.</td><td>Capability:
Addresses, PhysicalStorageBufferAddressesEXT</td></tr>
<tr><td>4</td><td>117</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td></tr>
</table>

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

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

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

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

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

OpSatConvertUToS 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.				Capability: **Kernel**
4	119	*<id>* *Result Type*	Result <id>	*<id>* *Unsigned Value*

OpConvertUToPtr Bit pattern-preserving conversion of an unsigned scalar integer to a pointer. *Result Type* must be an OpTypePointer. *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 will zero extend *Integer Value*. If the bit width of *Integer Value* is larger than that of *Result Type*, the conversion will truncate *Integer Value*. For same-width *Integer Value* and *Result Type*, this is the same as OpBitcast.				Capability: **Addresses**, **PhysicalStorageBufferAddressesEXT**
4	120	*<id>* *Result Type*	Result <id>	*<id>* *Integer Value*

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*

<table>
<tr><td colspan="4">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>122</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>5</td><td>123</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td><td>Storage Class
Storage</td></tr>
</table>

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

Bit pattern-preserving type conversion.

Result Type must be an OpTypePointer, or a scalar or vector of numerical-type.

Operand must have a type of OpTypePointer, or a scalar or vector of numerical-type. It must be a different type than Result Type.

If Result Type is a pointer, Operand must be a pointer or integer scalar. If Operand is a pointer, Result Type must be a pointer or integer scalar.

If Result Type has the same number of components as Operand, they must also have the same component width, and results are computed per component.

If Result Type has a different number of components than Operand, the total number of bits in Result Type must equal the total number of bits in Operand. Let L be the type, either Result Type or Operand’s type, that has the larger number of components. Let S be the other type, with the smaller number of components. The number of components in L must be an integer multiple of the number of components in S. The first component (that is, the only or lowest-numbered component) of S maps to the first components of L, and so on, up to the last component of S mapping to the last components of L. Within this mapping, any single component of S (mapping to multiple components of L) maps its lower-ordered bits to the lower-numbered components of L.</td></tr>
<tr><td>4</td><td>124</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Operand</td></tr>
</table>

3.32.12 Composite Instructions

OpVectorExtractDynamic

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

Result Type must be a scalar type.

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

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

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

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

OpVectorInsertDynamic

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

Result Type must be an OpTypeVector.

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

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

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

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

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

OpVectorShuffle

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

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

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

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

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

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

OpCompositeConstruct

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

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

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

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

OpCompositeExtract

Extract a part of a composite object.

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

Composite is the composite to extract from.

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

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

OpCompositeInsert

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

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

Object is the object to use as the modified part.

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

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

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

OpCopyObject

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

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

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>

3.32.13 Arithmetic Instructions

OpSNegate Signed-integer subtract of *Operand* from zero. *Result Type* must be a scalar or vector of integer type. *Operand's* type must be a scalar or vector of integer type. It must have the same number of components as *Result Type*. The component width must equal the component width in *Result Type*. Results are computed per component.				
4	126	<*id*> *Result Type*	Result <id>	<*id*> *Operand*

OpFNegate Floating-point subtract of *Operand* from zero. *Result Type* must be a scalar or vector of floating-point type. The type of *Operand* must be the same as *Result Type*. Results are computed per component.				
4	127	<*id*> *Result Type*	Result <id>	<*id*> *Operand*

OpIAdd Integer addition of *Operand 1* and *Operand 2*. *Result Type* must be a scalar or vector of integer type. The type of *Operand 1* and *Operand 2* must be a scalar or vector of integer type. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*. The resulting value will equal 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 will equal 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 will equal 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. The resulting value is undefined if *Operand 2* is 0.

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

OpSDiv Signed-integer division of *Operand 1* divided by *Operand 2*. *Result Type* must be a scalar or vector of integer type. The type of *Operand 1* and *Operand 2* must be a scalar or vector of integer type. They must have the same number of components as *Result Type*. They must have the same component width as *Result Type*. Results are computed per component. The resulting value is undefined if *Operand 2* is 0.					
5	135	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

OpFDiv Floating-point division of *Operand 1* divided by *Operand 2*. *Result Type* must be a scalar or vector of floating-point type. The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*. Results are computed per component. The resulting value is undefined if *Operand 2* is 0.					
5	136	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUMod Unsigned modulo operation of *Operand 1* modulo *Operand 2*. *Result Type* must be a scalar or vector of integer type, whose *Signedness* operand is 0. The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*. Results are computed per component. The resulting value is undefined if *Operand 2* is 0.					
5	137	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSRem Signed remainder operation 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. 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 $r \neq 0$, 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. 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 $r \neq 0$, 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 $r \neq 0$, 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 $r \neq 0$, 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*

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

Scale a floating-point matrix.

Result Type must be an OpTypeMatrix whose Column Type is a vector of floating-point type.

The type of Matrix must be the same as Result Type. Each component in each column in Matrix is multiplied by Scalar.

Scalar must have the same type as the Component Type in Result Type.</td><td>Capability:
Matrix</td></tr>
<tr><td>5</td><td>143</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Matrix</td><td><id>
Scalar</td></tr>
</table>

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

<table>
<tr><td colspan="5">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.</td><td>Capability:
Matrix</td></tr>
<tr><td>5</td><td>147</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Vector 1</td><td><id>
Vector 2</td></tr>
</table>

<table>
<tr><td colspan="6">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.</td></tr>
<tr><td>5</td><td>148</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Vector 1</td><td><id>
Vector 2</td></tr>
</table>

OpISubBorrow					

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* $\geq$ *Operand 2*, and gets 1 otherwise.

5	150	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

OpUMulExtended Result is the full value of the unsigned integer multiplication of *Operand 1* and *Operand 2*. *Result Type* must be from OpTypeStruct. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of integer type, whose *Signedness* operand is 0. *Operand 1* and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as unsigned integers. Results are computed per component. Member 0 of the result gets the low-order bits of the multiplication. Member 1 of the result gets the high-order bits of the multiplication.					
5	151	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

OpSMulExtended Result is the full value of the signed integer multiplication of *Operand 1* and *Operand 2*. *Result Type* must be from OpTypeStruct. The struct must have two members, and the two members must be the same type. The member type must be a scalar or vector of integer type. *Operand 1* and *Operand 2* must have the same type as the members of *Result Type*. These are consumed as signed integers. Results are computed per component. Member 0 of the result gets the low-order bits of the multiplication. Member 1 of the result gets the high-order bits of the multiplication.					
5	152	*<id>* *Result Type*	Result <id>	*<id>* *Operand 1*	*<id>* *Operand 2*

3.32.14 Bit Instructions

OpShiftRightLogical Shift the bits in *Base* right by the number of bits specified in *Shift*. The most-significant bits will be zero filled. *Result Type* must be a scalar or vector of integer type. The type of each *Base* and *Shift* must be a scalar or vector of integer type. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*. *Shift* is consumed as an unsigned integer. The result is undefined if *Shift* is greater than 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 will be filled with the sign bit from *Base*. *Result Type* must be a scalar or vector of integer type. The type of each *Base* and *Shift* must be a scalar or vector of integer type. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*. *Shift* is treated as unsigned. The result is undefined if *Shift* is greater than 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 will be zero filled. *Result Type* must be a scalar or vector of integer type. The type of each *Base* and *Shift* must be a scalar or vector of integer type. *Base* and *Shift* must have the same number of components. The number of components and bit width of the type of *Base* must be the same as in *Result Type*. *Shift* is treated as unsigned. The result is undefined if *Shift* is greater than 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) will come from the corresponding bits in Base.

Any result bits numbered in [Offset, Offset + Count - 1] come, in order, from the bits numbered [0, Count - 1] of Insert.

Count must be an integer type scalar. Count is the number of bits taken from Insert. It will be consumed as an unsigned value. Count can be 0, in which case the result will be Base.

Offset must be an integer type scalar. Offset is the lowest-order bit of the bit field. It will be consumed as an unsigned value.

The resulting value is undefined if Count or Offset or their sum is greater than the number of bits in the result.</td>
<td colspan="2">Capability:
Shader</td>
</tr>
<tr>
<td>7</td>
<td>201</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Base</td>
<td><id>
Insert</td>
<td><id>
Offset</td>
<td><id>
Count</td>
</tr>
</table>

OpBitFieldSExtract Extract a bit field from an object, with sign extension. Results are computed per component. *Result Type* must be a scalar or vector of integer type. The type of *Base* must be the same as *Result Type*. If *Count* is greater than 0: The bits of *Base* numbered in [*Offset*, *Offset* + *Count* - 1] (inclusive) become the bits numbered [0, *Count* - 1] of the result. The remaining bits of the result will all be the same as bit *Offset* + *Count* - *1* of *Base*. *Count* must be an integer type scalar. *Count* is the number of bits extracted from *Base*. It will be consumed as an unsigned value. *Count* can be 0, in which case the result will be 0. *Offset* must be an integer type scalar. *Offset* is the lowest-order bit of the bit field to extract from *Base*. It will be consumed as an unsigned value. The resulting value is undefined if *Count* or *Offset* or their sum is greater than the number of bits in the result.					Capability: **Shader**	
6	202	*<id>* *Result Type*	Result <id>	*<id>* *Base*	*<id>* *Offset*	*<id>* *Count*

OpBitFieldUExtract Extract a bit field from an object, without sign extension. The semantics are the same as with OpBitFieldSExtract with the exception that there is no sign extension. The remaining bits of the result will all be 0.					Capability: **Shader**	
6	203	*<id>* *Result Type*	Result <id>	*<id>* *Base*	*<id>* *Offset*	*<id>* *Count*

OpBitReverse Reverse the bits in an object. Results are computed per component. *Result Type* must be a scalar or vector of integer type. The type of *Base* must be the same as *Result Type*. The bit-number *n* of the result will be taken from bit-number *Width* - *1* - *n* of *Base*, where *Width* is the OpTypeInt operand of the *Result Type*.				Capability: **Shader**
4	204	*<id>* *Result Type*	Result <id>	*<id>* *Base*

OpBitCount Count the number of set bits in an object. Results are computed per component. *Result Type* must be a scalar or vector of integer type. The components must be wide enough to hold the unsigned *Width* of *Base* as an unsigned value. That is, no sign bit is needed or counted when checking for a wide enough result width. *Base* must be a scalar or vector of integer type. It must have the same number of components as *Result Type*. The result is the unsigned value that is the number of bits in *Base* that are 1.				
4	205	*<id>* *Result Type*	Result <id>	*<id>* *Base*

3.32.15 Relational and Logical Instructions

OpAny Result is **true** if any component of *Vector* is **true**, otherwise result is **false**. *Result Type* must be a Boolean type scalar. *Vector* must be a vector of Boolean type.				
4	154	*<id>* *Result Type*	Result <id>	*<id>* *Vector*

OpAll Result is **true** if all components of *Vector* are **true**, otherwise result is **false**. *Result Type* must be a Boolean type scalar. *Vector* must be a vector of Boolean type.				
4	155	*<id>* *Result Type*	Result <id>	*<id>* *Vector*

OpIsNan Result is **true** if *x* is an IEEE NaN, otherwise result is **false**. *Result Type* must be a scalar or vector of Boolean type. *x* must be a scalar or vector of floating-point type. It must have the same number of components as *Result Type*. Results are computed per component.				
4	156	*<id>* *Result Type*	Result <id>	*<id>* *x*

OpIsInf Result is **true** if *x* is an IEEE Inf, otherwise result is **false** *Result Type* must be a scalar or vector of Boolean type. *x* must be a scalar or vector of floating-point type. It must have the same number of components as *Result Type*. Results are computed per component.				
4	157	*<id>* *Result Type*	Result <id>	*<id>* *x*

<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>

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

Result is true if $x < y$ or $x > y$, where IEEE comparisons are used, otherwise result is false.

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

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

y must have the same type as x.

Results are computed per component.</td><td>Capability:
Kernel</td></tr>
<tr><td>5</td><td>161</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td><td><id>
y</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>5</td><td>162</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td><td><id>
y</td></tr>
</table>

<table>
<tr><td colspan="5">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.</td><td>Capability:
Kernel</td></tr>
<tr><td>5</td><td>163</td><td><id>
Result Type</td><td>Result <id></td><td><id>
x</td><td><id>
y</td></tr>
</table>

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*: When 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*

3.32.16 Derivative Instructions

<table>
<tr><td colspan="4">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.</td><td>Capability:
Shader</td></tr>
<tr><td>4</td><td>207</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Shader</td></tr>
<tr><td>4</td><td>208</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
Shader</td></tr>
<tr><td>4</td><td>209</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

OpDPdxFine Result is the partial derivative of *P* with respect to the window *x* coordinate.Will use local differencing based on the value of *P* for the current fragment and its immediate neighbor(s). *Result Type* must be a scalar or vector of floating-point type. The 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	210	*<id>* *Result Type*	Result <id>	*<id>* *P*

OpDPdyFine Result is the partial derivative of *P* with respect to the window *y* coordinate.Will use local differencing based on the value of *P* for the current fragment and its immediate neighbor(s). *Result Type* must be a scalar or vector of floating-point type. The 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	211	*<id>* *Result Type*	Result <id>	*<id>* *P*

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.				Capability: **DerivativeControl**
4	212	*<id>* *Result Type*	Result <id>	*<id>* *P*

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

Result is the partial derivative of P with respect to the window x coordinate. Will use local differencing based on the value of P for the current fragment's neighbors, and will possibly, but not necessarily, include the value of P for the current fragment. That is, over a given area, the implementation can compute x derivatives in fewer unique locations than would be allowed for OpDPdxFine.

Result Type must be a scalar or vector of floating-point type. The 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>213</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

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

Result is the partial derivative of P with respect to the window y coordinate. Will use local differencing based on the value of P for the current fragment's neighbors, and will possibly, but not necessarily, include the value of P for the current fragment. That is, over a given area, the implementation can compute y derivatives in fewer unique locations than would be allowed for OpDPdyFine.

Result Type must be a scalar or vector of floating-point type. The 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>214</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
DerivativeControl</td></tr>
<tr><td>4</td><td>215</td><td><id>
Result Type</td><td>Result <id></td><td><id>
P</td></tr>
</table>

3.32.17 Control-Flow Instructions

OpPhi

The SSA phi function.

The result is selected based on control flow: If control reached the current block from *Parent i*, *Result Id* gets the value that *Variable i* had at the end of *Parent i*.

Result Type can be any type.

Operands are a sequence of pairs: (*Variable 1*, *Parent 1* block), (*Variable 2*, *Parent 2* block), ... Each *Parent i* block is the label of an immediate predecessor in the CFG of the current block. 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, which can be mixed with **OpPhi**).

3 + variable	245	*<id>* *Result Type*	Result <id>	*<id>, <id>, ...* *Variable, Parent, ...*

OpLoopMerge

Declare a structured loop.

This instruction must immediately precede either an OpBranch or OpBranchConditional instruction. That is, it must be the second-to-last instruction in its block.

Merge Block is the label of the merge block for this structured loop.

Continue Target is the label of a block targeted for processing a loop "continue".

Loop Control Parameters appear in Loop Control-table order for any *Loop Control* setting that requires such a parameter.

See Structured Control Flow for more detail.

4 + variable	246	*<id>* *Merge Block*	*<id>* *Continue Target*	Loop Control	*Literal, Literal, ...* *Loop Control Parameters*

OpSelectionMerge Declare a structured selection. This instruction must immediately precede either an OpBranchConditional or OpSwitch instruction. That is, it must be the second-to-last instruction in its block. *Merge Block* is the label of the merge block for this structured selection. See Structured Control Flow for more detail.			
3	247	<*id*> *Merge Block*	Selection Control

OpLabel The block label instruction: Any reference to a block is through the *Result <id>* of its label. Must be the first instruction of any block, and appears only as the first instruction of a block.		
2	248	Result <id>

OpBranch Unconditional branch to *Target Label*. *Target Label* must be the *Result <id>* of an OpLabel instruction in the current function. This instruction must be the last instruction in a block.		
2	249	<*id*> *Target Label*

OpBranchConditional If *Condition* is **true**, branch to *True Label*, otherwise branch to *False Label*. *Condition* must be a Boolean type scalar. *True Label* must be an OpLabel in the current function. *False Label* must be an OpLabel in the current function. *Branch weights* are unsigned 32-bit integer literals. There must be either no *Branch Weights* or exactly two branch weights. If present, the first is the weight for branching to *True Label*, and the second is the weight for branching to *False Label*. The implied probability that a branch is taken is its weight divided by the sum of the two *Branch weights*. 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 two weights must not overflow a 32-bit unsigned integer when added together. This instruction must be the last instruction in a block.					
4 + variable	250	*<id>* *Condition*	*<id>* *True Label*	*<id>* *False Label*	*Literal, Literal, . . .* *Branch weights*

OpSwitch Multi-way branch to one of the operand label *<id>*. *Selector* must have a type of OpTypeInt. *Selector* will be compared for equality to the *Target* literals. *Default* must be the *<id>* of a label. If *Selector* does not equal any of the *Target* literals, control flow will branch to the *Default* label *<id>*. *Target* must be alternating scalar integer *literals* and the *<id>* of a label. If *Selector* equals a *literal*, control flow will branch to the following *label <id>*. It is invalid for any two *literal* to be equal to each other. If *Selector* does not equal any *literal*, control flow will branch to the *Default* label *<id>*. Each *literal* is interpreted with the type of *Selector*: The bit width of *Selector's* type will be the width of each *literal's* type. If this width is not a multiple of 32-bits, the literals must be sign extended when the OpTypeInt *Signedness* is set to 1. (See Literal Number.) This instruction must be the last instruction in a block.				
3 + variable	251	*<id>* *Selector*	*<id>* *Default*	*literal, label <id>,* *literal, label <id>,* *. . .* *Target*

<table>
<tr><td>OpKill

Fragment-shader discard.

Ceases all further processing in any invocation that executes it: Only instructions these invocations executed before OpKill will have observable side effects. If this instruction is executed in non-uniform control flow, all subsequent control flow is non-uniform (for invocations that continue to execute).

This instruction must be the last instruction in a block.

This instruction is only valid in the Fragment Execution Model.</td><td>Capability:
Shader</td></tr>
<tr><td>1</td><td>252</td></tr>
</table>

<table>
<tr><td colspan="2">OpReturn

Return with no value from a function with void return type.

This instruction must be the last instruction in a block.</td></tr>
<tr><td>1</td><td>253</td></tr>
</table>

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

Return a value from a function.

Value is the value returned, by copy, and must match the Return Type operand of the OpTypeFunction type of the OpFunction body this return instruction is in.

This instruction must be the last instruction in a block.</td></tr>
<tr><td>2</td><td>254</td><td><id>
Value</td></tr>
</table>

<table>
<tr><td colspan="2">OpUnreachable

Declares that this block is not reachable in the CFG.

This instruction must be the last instruction in a block.</td></tr>
<tr><td>1</td><td>255</td></tr>
</table>

OpLifetimeStart Declare that an object was not defined before this instruction. *Pointer* is a pointer to the object whose lifetime is starting. Its type must be an OpTypePointer with Storage Class **Function**. *Size* must be 0 if *Pointer* is a pointer to a non-void type or the **Addresses** capability is not being used. If *Size* is non-zero, it is the number of bytes of memory whose lifetime is starting. Its type must be an integer type scalar. It is treated as unsigned; if its type has *Signedness* of 1, its sign bit cannot be set.			Capability: **Kernel**
3	256	*<id>* *Pointer*	Literal Number *Size*

OpLifetimeStop Declare that an object is dead after this instruction. *Pointer* is a pointer to the object whose lifetime is ending. Its type must be an OpTypePointer with Storage Class **Function**. *Size* must be 0 if *Pointer* is a pointer to a non-void type or the **Addresses** capability is not being used. If *Size* is non-zero, it is the number of bytes of memory whose lifetime is ending. Its type must be an integer type scalar. It is treated as unsigned; if its type has *Signedness* of 1, its sign bit cannot be set.			Capability: **Kernel**
3	257	*<id>* *Pointer*	Literal Number *Size*

3.32.18 Atomic Instructions

OpAtomicLoad

Atomically load through *Pointer* using the given *Semantics*. All subparts of the value that is loaded will be read atomically with respect to all other atomic accesses to it within *Scope*.

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

Pointer is the pointer to the memory to read. The type of the value pointed to by *Pointer* must be the same as *Result Type*.

Memory must be a valid 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* will be written atomically with respect to all other atomic accesses to it within *Scope*.

Pointer is the pointer to the memory to write. The type it points to must be a scalar of integer type or floating-point type.

Value is the value to write. The type of *Value* and the type pointed to by *Pointer* must be the same type.

Memory must be a valid 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 must be a valid 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* cannot be set to **Release** or **Acquire and Release**. In addition, *Unequal* cannot be set to a stronger memory-order then *Equal*.

The type of *Value* must be the same as *Result Type*. The type of the value pointed to by *Pointer* must be the same as *Result Type*. This type must also match the type of *Comparator*.

Memory must be a valid 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 must be a valid 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* must be a valid 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* must be a valid 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* must be a valid 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* must be a valid 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* must be a valid 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* must be a valid 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 must be a valid 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 must be a valid 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* must be a valid 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* must be a valid 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* must be a valid memory Scope.							
7	242	<*id*> *Result Type*	Result <id>	<*id*> *Pointer*	Scope <id> *Memory*	Memory Semantics <id> *Semantics*	<*id*> *Value*

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

Atomically sets the flag value pointed to by Pointer to the set state.

Pointer must be a pointer to a 32-bit integer type representing an atomic flag.

The instruction’s result is true if the flag was in the set state or false if the flag was in the clear state immediately before the operation.

Result Type must be a Boolean type.

Results are undefined if an atomic flag is modified by an instruction other than OpAtomicFlagTestAndSet or OpAtomicFlagClear

Memory must be a valid memory Scope.</td><td colspan="2">Capability:
Kernel</td></tr>
<tr><td>6</td><td>318</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pointer</td><td>Scope <id>
Memory</td><td>Memory Semantics <id>
Semantics</td></tr>
</table>

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

Atomically sets the flag value pointed to by Pointer to the clear state.

Pointer must be a pointer to a 32-bit integer type representing an atomic flag.

Memory Semantics cannot be Acquire or AcquireRelease

Results are undefined if an atomic flag is modified by an instruction other than OpAtomicFlagTestAndSet or OpAtomicFlagClear

Memory must be a valid memory Scope.</td><td>Capability:
Kernel</td></tr>
<tr><td>4</td><td>319</td><td><id>
Pointer</td><td>Scope <id>
Memory</td><td>Memory Semantics <id>
Semantics</td></tr>
</table>

3.32.19 Primitive Instructions

OpEmitVertex Emits the current values of all output variables to the current output primitive. After execution, the values of all output variables are undefined. This instruction can only be used when only one stream is present.	Capability: **Geometry**
1	218

OpEndPrimitive Finish the current primitive and start a new one. No vertex is emitted. This instruction can only be used when only one stream is present.	Capability: **Geometry**
1	219

OpEmitStreamVertex Emits the current values of all output variables to the current output primitive. After execution, the values of all output variables are undefined. *Stream* must be an <*id*> of a constant instruction with a scalar integer type. That constant is the output-primitive stream number. This instruction can only be used when multiple streams are present.		Capability: **GeometryStreams**
2	220	<*id*> *Stream*

OpEndStreamPrimitive Finish the current primitive and start a new one. No vertex is emitted. *Stream* must be an <*id*> of a constant instruction with a scalar integer type. That constant is the output-primitive stream number. This instruction can only be used when multiple streams are present.		Capability: **GeometryStreams**
2	221	<*id*> *Stream*

3.32.20 Barrier Instructions

OpControlBarrier

Wait for other invocations of this module to reach the current point of execution.

All invocations of this module within *Execution* scope must reach this point of execution before any invocation will proceed beyond it.

When *Execution* is **Workgroup** or larger, behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. 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 must also perform and adhere to the description and semantics of an **OpMemoryBarrier** instruction with the same *Memory* and *Semantics* operands. This allows atomically specifying both a control barrier and a memory barrier (that is, without needing two instructions). If *Semantics* is **None**, *Memory* is ignored.

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**.

When used with the **TessellationControl** execution model, it also implicitly synchronizes the **Output** Storage Class: Writes to **Output** variables performed by any invocation executed prior to a **OpControlBarrier** will be visible to any other invocation after return from that **OpControlBarrier**.

4	224	Scope <id> *Execution*	Scope <id> *Memory*	Memory Semantics <id> *Semantics*

OpMemoryBarrier

Control the order that memory accesses are observed.

Ensures that memory accesses issued before this instruction will be observed before memory accesses issued after this instruction. This control is ensured only for memory accesses issued by this invocation and observed by another invocation executing within *Memory* scope.

Semantics declares what kind of memory is being controlled and what kind of control to apply.

To execute both a memory barrier and a control barrier, see OpControlBarrier.

3	225	Scope <id> *Memory*	Memory Semantics <id> *Semantics*

OpNamedBarrierInitialize

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*

<table>
<tr><td colspan="4">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 must also perform and adhere to the description and semantics of an OpMemoryBarrier instruction with the same Memory and Semantics operands. This allows atomically specifying both a control barrier and a memory barrier (that is, without needing two instructions). If Semantics None, Memory is ignored.</td><td>Capability:
NamedBarrier

Missing before version 1.1.</td></tr>
<tr><td>4</td><td>329</td><td><<i>id</i>>
<i>Named Barrier</i></td><td>Scope <id>
<i>Memory</i></td><td>Memory Semantics <id>
<i>Semantics</i></td></tr>
</table>

3.32.21 Group Instructions

<table>
<tr><td colspan="9">

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

All invocations of this module within *Execution* must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*.

Result Type must be an OpTypeEvent object.

Destination must be a pointer to a scalar or vector of floating-point type or integer type.

Destination pointer Storage Class must be **Workgroup** or **CrossWorkgroup**.

The type of *Source* must be the same as *Destination*.

When *Destination* pointer Storage Class is **Workgroup**, the *Source* pointer Storage Class must be **CrossWorkgroup**. In this case *Stride* defines the stride in elements when reading from *Source* pointer.

When *Destination* pointer Storage Class is **CrossWorkgroup**, the *Source* pointer Storage Class must be **Workgroup**. In this case *Stride* defines the stride in elements when writing each element to *Destination* pointer.

Stride and *NumElements* must be a 32-bit integer type scalar when the addressing model is *Physical32* and 64 bit integer type scalar when the *Addressing Model* is *Physical64*.

Event must have a type of OpTypeEvent.

Event can be used to associate the copy with a previous copy allowing an event to be shared by multiple copies. Otherwise *Event* should be an OpConstantNull.

If *Event* argument is not OpConstantNull, the event object supplied in event argument will be returned.

</td><td colspan="2">Capability:
Kernel</td></tr>
<tr><td>9</td><td>259</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Destination</td><td><id>
Source</td><td><id>
Num Elements</td><td><id>
Stride</td><td><id>
Event</td></tr>
</table>

OpGroupWaitEvents Wait for events generated by OpGroupAsyncCopy operations to complete. *Events List* points to *Num Events* event objects, which will be released after the wait is performed. All invocations of this module within *Execution* must reach this point of execution. Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. *Execution* must be **Workgroup** or **Subgroup** Scope. *Num Events* must be a 32-bit integer type scalar. *Events List* must be a pointer to OpTypeEvent.				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**. All invocations of this module within *Execution* must reach this point of execution. Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. *Result Type* must be a Boolean type. *Execution* must be **Workgroup** or **Subgroup** Scope. *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**. All invocations of this module within *Execution* must reach this point of execution. Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. *Result Type* must be a Boolean type. *Execution* must be **Workgroup** or **Subgroup** Scope. *Predicate* must be a Boolean type.					Capability: **Groups**
5	262	*<id>* *Result Type*	Result <id>	Scope <id> *Execution*	*<id>* *Predicate*

OpGroupBroadcast Return the *Value* of the invocation identified by the local id *LocalId* to all invocations in the group. All invocations of this module within *Execution* must reach this point of execution. Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. *Result Type* must be a 32-bit or 64-bit integer type or a 16, 32 or 64 float type scalar. *Execution* must be **Workgroup** or **Subgroup** Scope. The type of *Value* must be the same as *Result Type*. *LocalId* must be an integer datatype. It can be a scalar, or a vector with 2 components or a vector with 3 components. *LocalId* must be the same for all invocations in the group.					Capability: **Groups**	
6	263	*<id>* *Result Type*	Result <id>	Scope <id> *Execution*	*<id>* *Value*	*<id>* *LocalId*

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

An integer add group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 32-bit or 64-bit integer type scalar.

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is 0.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>264</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

A floating-point add group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 16-bit, 32-bit, or 64-bit floating-point type scalar.

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is 0.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>265</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

A floating-point minimum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 16-bit, 32-bit, or 64-bit floating-point type scalar.

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is +INF.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>266</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

An unsigned integer minimum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 32-bit or 64-bit integer type scalar.

Execution must be Workgroup or Subgroup Scope.

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.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>267</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

A signed integer minimum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 32-bit or 64-bit integer type scalar.

Execution must be Workgroup or Subgroup Scope.

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.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>268</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

A floating-point maximum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 16-bit, 32-bit, or 64-bit floating-point type scalar.

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is -INF.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>269</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td>Group Operation
Operation</td><td><id>
X</td></tr>
</table>

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

An unsigned integer maximum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Result Type must be a 32-bit or 64-bit integer type scalar.

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is 0.

The type of X must be the same as Result Type.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>270</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td>Scope <id>
<i>Execution</i></td><td>Group Operation
<i>Operation</i></td><td><<i>id</i>>
<i>X</i></td></tr>
</table>

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

A signed integer maximum group operation specified for all values of X specified by invocations in the group.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

X and Result Type must be a 32-bit or 64-bit OpTypeInt data type.

Execution must be Workgroup or Subgroup Scope.

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.</td><td colspan="2">Capability:
Groups</td></tr>
<tr><td>6</td><td>271</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td>Scope <id>
<i>Execution</i></td><td>Group Operation
<i>Operation</i></td><td><<i>id</i>>
<i>X</i></td></tr>
</table>

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

See extension SPV_KHR_shader_ballot</td><td>Capability:
SubgroupBallotKHR

Reserved.</td></tr>
<tr><td>4</td><td>4421</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td><<i>id</i>>
<i>Predicate</i></td></tr>
</table>

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*

3.32.22 Device-Side Enqueue Instructions

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

OpEnqueueMarker

Enqueue a marker command to the queue object specified by *Queue*. The marker command waits for a list of events to complete, or if the list is empty it waits for all previously enqueued commands in *Queue* to complete before the marker completes.

Result Type must be a 32-bit integer type scalar. A successful enqueue results in the value 0. A failed enqueue results in a non-0 value.

Queue must be of the type OpTypeQueue.

Num Events specifies the number of event objects in the wait list pointed to by *Wait Events* and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Wait Events specifies the list of wait event objects and must be a pointer to OpTypeDeviceEvent.

Ret Event is a pointer to a device event which gets implicitly retained by this instruction. It must have a type of OpTypePointer to OpTypeDeviceEvent. If *Ret Event* is set to null this instruction becomes a no-op.

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

Capability:
DeviceEnqueue

</td>
</tr>
<tr>
<td>7</td>
<td>291</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Queue</td>
<td><id>
Num Events</td>
<td><id>
Wait Events</td>
<td><id>
Ret Event</td>
</tr>
</table>

OpEnqueueKernel

Capability:
DeviceEnqueue

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.

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="7">OpGetKernelNDrangeSubGroupCount

Returns the number of subgroups in each workgroup of the dispatch (except for the last in cases where the global size does not divide cleanly into work-groups) given the combination of the passed NDRange descriptor specified by ND Range and the function specified by Invoke.

Result Type must be a 32-bit integer type scalar.

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>

OpGetKernelNDrangeMaxSubGroupSize Returns the maximum sub-group size for the function specified by *Invoke* and the NDRange specified by *ND Range*. *Result Type* must be a 32-bit integer type scalar. 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.							Capability: **DeviceEnqueue**	
8	294	*<id>* *Result Type*	Result <id>	*<id>* *ND Range*	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

OpGetKernelWorkGroupSize Returns the maximum work-group size that can be used to execute the function specified by *Invoke* on the device. *Result Type* must be a 32-bit integer type scalar. *Invoke* must be an OpFunction whose OpTypeFunction operand has: - *Result Type* must be OpTypeVoid. - The first parameter must have a type of OpTypePointer to an 8-bit OpTypeInt. - An optional list of parameters, each of which must have a type of OpTypePointer to the **Workgroup** Storage Class. *Param* is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit integer type scalar. *Param Size* is the size in bytes of the memory pointed to by *Param* and must be a 32-bit integer type scalar, which is treated as an unsigned integer. *Param Align* is the alignment of *Param* and must be a 32-bit integer type scalar, which is treated as an unsigned integer.						Capability: **DeviceEnqueue**	
7	295	*<id>* *Result Type*	Result <id>	*<id>* *Invoke*	*<id>* *Param*	*<id>* *Param Size*	*<id>* *Param Align*

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

Returns the preferred multiple of work-group size for the function specified by Invoke. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size will not fail to enqueue Invoke for execution unless the work-group size specified is larger than the device maximum.

Result Type must be a 32-bit integer type scalar.

Invoke must be an OpFunction whose OpTypeFunction operand has:
- Result Type must be OpTypeVoid.
- The first parameter must have a type of OpTypePointer to an 8-bit OpTypeInt.
- An optional list of parameters, each of which must have a type of OpTypePointer to the Workgroup Storage Class.

Param is the first parameter of the function specified by Invoke and must be a pointer to an 8-bit integer type scalar.

Param Size is the size in bytes of the memory pointed to by Param and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Param Align is the alignment of Param and must be a 32-bit integer type scalar, which is treated as an unsigned integer.</td><td colspan="2">Capability:
DeviceEnqueue</td></tr>
<tr><td>7</td><td>296</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Invoke</td><td><id>
Param</td><td><id>
Param Size</td><td><id>
Param Align</td></tr>
</table>

<table>
<tr><td colspan="2">OpRetainEvent

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

Event must be an event that was produced by OpEnqueueKernel, OpEnqueueMarker or OpCreateUserEvent.</td><td>Capability:
DeviceEnqueue</td></tr>
<tr><td>2</td><td>297</td><td><id>
Event</td></tr>
</table>

OpReleaseEvent Decrements the reference count of the event object specified by *Event*. The event object is deleted once the event reference count is zero, the specific command identified by this event has completed (or terminated) and there are no commands in any device command queue that require a wait for this event to complete. *Event* must be an event that was produced by OpEnqueueKernel, OpEnqueueMarker or OpCreateUserEvent.		Capability: **DeviceEnqueue**
2	298	*<id>* *Event*

OpCreateUserEvent Create a user event. The execution status of the created event is set to a value of 2 (CL_SUBMITTED). *Result Type* must be OpTypeDeviceEvent.			Capability: **DeviceEnqueue**
3	299	*<id>* *Result Type*	Result <id>

OpIsValidEvent Returns **true** if the event specified by *Event* is a valid event, otherwise result is **false**. *Result Type* must be a Boolean type. *Event* must have a type of OpTypeDeviceEvent				Capability: **DeviceEnqueue**
4	300	*<id>* *Result Type*	Result <id>	*<id>* *Event*

OpSetUserEventStatus Sets the execution status of a user event specified by *Event.Status* can be either 0 (CL_COMPLETE) to indicate that this kernel and all its child kernels finished execution successfully, or a negative integer value indicating an error. *Event* must have a type of OpTypeDeviceEvent that was produced by OpCreateUserEvent. *Status* must have a type of 32-bit OpTypeInt treated as a signed integer.			Capability: **DeviceEnqueue**
3	301	*<id>* *Event*	*<id>* *Status*

<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 will be available in the memory pointed to by Value once the command identified by Event has completed.

Event must have a type of OpTypeDeviceEvent that was produced by OpEnqueueKernel or OpEnqueueMarker.

Profiling Info must be an integer type scalar. The content of Profiling Info is interpreted as Kernel Profiling Info mask.

Value must be a pointer to a scalar 8-bit integer type in the CrossWorkgroup Storage Class.

When Profiling Info is CmdExecTime, Value must point to 128-bit memory range.
The first 64 bits contain the elapsed time CL_PROFILING_COMMAND_END - CL_PROFILING_COMMAND_START for the command identified by Event in nanoseconds.
The second 64 bits contain the elapsed time CL_PROFILING_COMMAND_COMPLETE - CL_PROFILING_COMMAND_START for the command identified by Event in nanoseconds.

Note: The behavior of this instruction is undefined when called multiple times for the same event.</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

Returns the default device queue. If a default device queue has not been created, a null queue object is returned.

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

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

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

Result Type must be an OpTypeStruct with the following ordered list of members, starting from the first to last:

1) 32-bit integer type scalar, that specifies the number of dimensions used to specify the global work-items and work-items in the work-group.

2) OpTypeArray with 3 elements, where each element is 32-bit integer type scalar when the addressing model is Physical32 and 64-bit integer type scalar when the addressing model is Physical64. This member is an array of per-dimension unsigned values that describe the offset used to calculate the global ID of a work-item.

3) OpTypeArray with 3 elements, where each element is 32-bit integer type scalar when the addressing model is Physical32 and 64-bit integer type scalar when the addressing model is Physical64. This member is an array of per-dimension unsigned values that describe the number of global work-items in the dimensions that will execute the kernel function.

4) OpTypeArray with 3 elements, where each element is 32-bit integer type scalar when the addressing model is Physical32 and 64-bit integer type scalar when the addressing model is Physical64. This member is an array of per-dimension unsigned values that describe the number of work-items that make up a work-group.

GlobalWorkSize must be a scalar or an array with 2 or 3 components. Where the type of each element in the array is 32-bit integer type scalar when the addressing model is Physical32 or 64-bit integer type scalar when the addressing model is Physical64.

The type of LocalWorkSize must be the same as GlobalWorkSize.

The type of GlobalWorkOffset must be the same as GlobalWorkSize.</td>
<td colspan="2">Capability:
DeviceEnqueue</td>
</tr>
<tr>
<td>6</td>
<td>304</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
GlobalWorkSize</td>
<td><id>
LocalWorkSize</td>
<td><id>
GlobalWorkOffset</td>
</tr>
</table>

OpGetKernelLocalSizeForSubgroupCount

Returns the 1D local size to enqueue *Invoke* with *Subgroup Count* subgroups per workgroup.

Result Type must be a 32-bit integer type scalar.

Subgroup Count must be a 32-bit integer type scalar.

Invoke must be an OpFunction whose OpTypeFunction operand has:
- *Result Type* must be OpTypeVoid.
- The first parameter must have a type of OpTypePointer to an 8-bit OpTypeInt.
- 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*

OpGetKernelMaxNumSubgroups

Returns the maximum number of subgroups that can be used to execute *Invoke* on the devce.

Result Type must be a 32-bit integer type scalar.

Invoke must be an OpFunction whose OpTypeFunction operand has:
- *Result Type* must be OpTypeVoid.
- The first parameter must have a type of OpTypePointer to an 8-bit OpTypeInt.
- An optional list of parameters, each of which must have a type of OpTypePointer to the **Workgroup** Storage Class.

Param is the first parameter of the function specified by *Invoke* and must be a pointer to an 8-bit integer type scalar.

Param Size is the size in bytes of the memory pointed to by *Param* and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Param Align is the alignment of *Param* and must be a 32-bit integer type scalar, which is treated as an unsigned integer.

Capability:
SubgroupDispatch

Missing before **version 1.1**.

7	326	<id> *Result Type*	Result <id>	<id> *Invoke*	<id> *Param*	<id> *Param Size*	<id> *Param Align*

3.32.23 Pipe Instructions

OpReadPipe

Read a packet from the pipe object specified by *Pipe* into *Pointer*. Result is 0 if the operation is successful and a negative value if the pipe is empty.

Result Type must be a 32-bit integer type scalar.

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

Pointer must have a type of OpTypePointer with the same data type as *Pipe* and a **Generic** Storage Class.

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and *Packet Alignment* must satisfy the following:
- 1 <= *Packet Alignment* <= *Packet Size*.
- *Packet Alignment* must evenly divide *Packet Size*

For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.

Capability:
Pipes

7	274	*<id>* *Result Type*	Result <id>	*<id>* *Pipe*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

OpWritePipe Write a packet from *Pointer* to the pipe object specified by *Pipe*. Result is 0 if the operation is successful and a negative value if the pipe is full. *Result Type* must be a 32-bit integer type scalar. *Pipe* must have a type of OpTypePipe with **WriteOnly** access qualifier. *Pointer* must have a type of OpTypePointer with the same data type as *Pipe* and a **Generic** Storage Class. *Packet Size* 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 presents the alignment in bytes of each packet in the pipe. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.						Capability: **Pipes**	
7	275	*<id>* *Result Type*	Result <id>	*<id>* *Pipe*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

<table>
<tr><td colspan="8">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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>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 presents the alignment in bytes of each packet in the pipe. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.								Capability: **Pipes**	
9	277	*<id>* *Result Type*	Result <id>	*<id>* *Pipe*	*<id>* *Reserve Id*	*<id>* *Index*	*<id>* *Pointer*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

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

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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td>
<td colspan="2">Capability:
Pipes</td>
</tr>
<tr>
<td>7</td>
<td>278</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Pipe</td>
<td><id>
Num Packets</td>
<td><id>
Packet Size</td>
<td><id>
Packet Alignment</td>
</tr>
</table>

OpReserveWritePipePackets Reserve *num_packets* entries for writing to the pipe object specified by *Pipe*. Result is a valid reservation ID if the reservation is successful. *Pipe* must have a type of OpTypePipe with **WriteOnly** access qualifier. *Num Packets* must be a 32-bit OpTypeInt which is treated as an unsigned value. *Result Type* must be an OpTypeReserveId. *Packet Size* 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 presents the alignment in bytes of each packet in the pipe. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.						Capability: **Pipes**	
7	279	<id> *Result Type*	Result <id>	<id> *Pipe*	<id> *Num Packets*	<id> *Packet Size*	<id> *Packet Alignment*

<table>
<tr><td colspan="5">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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td>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="5">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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td>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 colspan="4">OpIsValidReserveId

Return true if Reserve Id is a valid reservation id and false otherwise.

Result Type must be a Boolean type.

Reserve Id must have a type of OpTypeReserveId.</td><td>Capability:
Pipes</td></tr>
<tr><td>4</td><td>282</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Reserve Id</td></tr>
</table>

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

Result is the number of available entries in the pipe object specified by Pipe. The number of available entries in a pipe is a dynamic value. The value returned should be considered immediately stale.

Result Type must be a 32-bit integer type scalar, which should be treated as an unsigned value.

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

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td>
<td colspan="2">Capability:
Pipes</td>
</tr>
<tr>
<td>6</td>
<td>283</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Pipe</td>
<td><id>
Packet Size</td>
<td><id>
Packet Alignment</td>
</tr>
</table>

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

Result is the maximum number of packets specified when the pipe object specified by Pipe was created.

Result Type must be a 32-bit integer type scalar, which should be treated as an unsigned value.

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

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td>
<td colspan="2">Capability:
Pipes</td>
</tr>
<tr>
<td>6</td>
<td>284</td>
<td><id>
Result Type</td>
<td>Result <id></td>
<td><id>
Pipe</td>
<td><id>
Packet Size</td>
<td><id>
Packet Alignment</td>
</tr>
</table>

OpGroupReserveReadPipePackets Reserve *Num Packets* entries for reading from the pipe object specified by *Pipe* at group level. Result is a valid reservation id if the reservation is successful. The reserved pipe entries are referred to by indices that go from 0 . . . *Num Packets* - 1. All invocations of this module within *Execution* must reach this point of execution. Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*. *Result Type* must be an OpTypeReserveId. *Execution* must be **Workgroup** or **Subgroup** Scope. *Pipe* must have a type of OpTypePipe with **ReadOnly** access qualifier. *Num Packets* must be a 32-bit integer type scalar, which is treated as an unsigned value. *Packet Size* 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 presents the alignment in bytes of each packet in the pipe. *Packet Size* and *Packet Alignment* must satisfy the following: - 1 <= *Packet Alignment* <= *Packet Size*. - *Packet Alignment* must evenly divide *Packet Size* For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.							Capability: **Pipes**	
8	285	*<id>* *Result Type*	Result <id>	Scope <id> *Execution*	*<id>* *Pipe*	*<id>* *Num Packets*	*<id>* *Packet Size*	*<id>* *Packet Alignment*

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

OpGroupReserveWritePipePackets

Reserve *Num Packets* entries for writing to the pipe object specified by *Pipe* at group level. Result is a valid reservation ID if the reservation is successful.

The reserved pipe entries are referred to by indices that go from 0 . . . *Num Packets* - 1.

All invocations of this module within *Execution* must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within *Execution*.

Result Type must be an OpTypeReserveId.

Execution must be **Workgroup** or **Subgroup** Scope.

Pipe must have a type of OpTypePipe with **WriteOnly** access qualifier.

Num Packets must be a 32-bit integer type scalar, which is treated as an unsigned value.

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and *Packet Alignment* must satisfy the following:
- 1 <= *Packet Alignment* <= *Packet Size*.
- *Packet Alignment* must evenly divide *Packet Size*

For concrete types, *Packet Alignment* should equal *Packet Size*. For aggregate types, *Packet Alignment* should be the size of the largest primitive type in the hierarchy of types.

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

Capability:
Pipes

</td>
</tr>
<tr>
<td>8</td>
<td>286</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>

<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.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Execution must be Workgroup or Subgroup Scope.

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

Reserve Id must have a type of OpTypeReserveId.

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td><td colspan="2">Capability:
Pipes</td></tr>
<tr><td>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>

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

A group level indication that all writes to Num Packets associated with the reservation specified by Reserve Id to the pipe object specified by Pipe are completed.

All invocations of this module within Execution must reach this point of execution.

Behavior is undefined if this instruction is used in control flow that is non-uniform within Execution.

Execution must be Workgroup or Subgroup Scope.

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

Reserve Id must have a type of OpTypeReserveId.

Packet Size 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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.</td>
<td colspan="2">Capability:
Pipes</td>
</tr>
<tr>
<td>6</td>
<td>288</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>

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

Creates a pipe-storage object.

Result Type must be OpTypePipeStorage.

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 presents the alignment in bytes of each packet in the pipe.

Packet Size and Packet Alignment must satisfy the following:
- 1 <= Packet Alignment <= Packet Size.
- Packet Alignment must evenly divide Packet Size

For concrete types, Packet Alignment should equal Packet Size. For aggregate types, Packet Alignment should be the size of the largest primitive type in the hierarchy of types.

Capacity is the minimum number of Packet Size blocks the resulting OpTypePipeStorage can hold.</td><td colspan="2">Capability:
PipeStorage

Missing before version 1.1.</td></tr>
<tr><td>6</td><td>323</td><td><id>
Result Type</td><td>Result <id></td><td>Literal Number
Packet Size</td><td>Literal Number
Packet Alignment</td><td>Literal Number
Capacity</td></tr>
</table>

<table>
<tr><td colspan="4">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.</td><td>Capability:
PipeStorage

Missing before version 1.1.</td></tr>
<tr><td>4</td><td>324</td><td><id>
Result Type</td><td>Result <id></td><td><id>
Pipe Storage</td></tr>
</table>

3.32.24 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* must be **Workgroup** or **Subgroup** Scope.				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* must be **Workgroup** or **Subgroup** Scope. *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* must be **Workgroup** or **Subgroup** Scope. *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*

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* must be **Workgroup** or **Subgroup** Scope. *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 when it is a floating-point type, an ordered-and-equal compare is used.	Capability: **GroupNonUniformVote** Missing before **version 1.3**.

5	336	<*id*> *Result Type*	Result <id>	Scope <id> *Execution*	<*id*> *Value*

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

Return 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 must be Workgroup or Subgroup Scope.

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

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

Id must come from a constant instruction.

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><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td>Scope <id>
<i>Execution</i></td><td><<i>id</i>>
<i>Value</i></td><td><<i>id</i>>
<i>Id</i></td></tr>
</table>

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

Return 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 must be Workgroup or Subgroup Scope.

The type of Value must be the same as Result Type.</td><td>Capability:
GroupNonUniformBallot

Missing before version 1.3.</td></tr>
<tr><td>5</td><td>338</td><td><<i>id</i>>
<i>Result Type</i></td><td>Result <id></td><td>Scope <id>
<i>Execution</i></td><td><<i>id</i>>
<i>Value</i></td></tr>
</table>

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

Returns 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 must be Workgroup or Subgroup Scope.

Predicate must be a Boolean type.</td><td>Capability:
GroupNonUniformBallot

Missing before version 1.3.</td></tr>
<tr><td>5</td><td>339</td><td><id>
Result Type</td><td>Result <id></td><td>Scope <id>
Execution</td><td><id>
Predicate</td></tr>
</table>

<table>
<tr><td colspan="5">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 must be Workgroup or Subgroup Scope.

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

Value must be 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.</td><td>Capability:
GroupNonUniformBallot

Missing before version 1.3.</td></tr>
<tr><td>5</td><td>340</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">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 must be Workgroup or Subgroup Scope.

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.</td><td colspan="2">Capability:
GroupNonUniformBallot

Missing before version 1.3.</td></tr>
<tr><td>6</td><td>341</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>

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

A group operation that returns the number of bits that are set to 1 in Value, only considering 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 must be Workgroup or Subgroup Scope.

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="5">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 result is undefined.

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

Execution must be Workgroup or Subgroup Scope.

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>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="5">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 result is undefined.

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

Execution must be Workgroup or Subgroup Scope.

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>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

Return 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 must be Workgroup or Subgroup Scope.

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

Return 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 must be Workgroup or Subgroup Scope.

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

Return 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 must be Workgroup or Subgroup Scope.

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>

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

Return 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 must be Workgroup or Subgroup Scope.

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.</td><td colspan="2">Capability:
GroupNonUniformShuffleRelative

Missing before version 1.3.</td></tr>
<tr><td>6</td><td>348</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>

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

An integer add group operation of all Value operands contributed active by invocations in the group.

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

Execution must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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>349</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">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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 1. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 1. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is INT_MAX. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is UINT_MAX. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be **Workgroup** or **Subgroup** Scope.

The identity *I* for *Operation* is +INF. If *Operation* is **ClusteredReduce**, *ClusterSize* must be specified.

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. *ClusterSize* must be at least 1, and must be 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 must be **Workgroup** or **Subgroup** Scope.

The identity *I* for *Operation* is INT_MIN. If *Operation* is **ClusteredReduce**, *ClusterSize* must be specified.

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. *ClusterSize* must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is -INF. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is ~0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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>

<table>
<tr><td colspan="6">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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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>360</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>

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* must be **Workgroup** or **Subgroup** Scope. The identity *I* for *Operation* is 0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be specified. 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. *ClusterSize* must be at least 1, and must be 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*

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* must be **Workgroup** or **Subgroup** Scope. The identity *I* for *Operation* is ~0. If *Operation* is **ClusteredReduce**, *ClusterSize* must be specified. 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. *ClusterSize* must be at least 1, and must be 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	362	<*id*> *Result Type*	Result <id>	Scope <id> *Execution*	Group Operation *Operation*	<*id*> *Value*	Optional <*id*> *ClusterSize*

<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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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 must be Workgroup or Subgroup Scope.

The identity I for Operation is 0. If Operation is ClusteredReduce, ClusterSize must be specified.

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. ClusterSize must be at least 1, and must be 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>

OpGroupNonUniformQuadBroadcast

Return the *Value* of the invocation within the quad whose **SubgroupLocalInvocationId** % 4 is equal to *Index*.

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

Execution must be **Workgroup** or **Subgroup** Scope.

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

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

Index must come from a constant instruction.

If the value of *Index* is greater or equal to 4, an undefined result is returned.

Capability:
GroupNonUniformQuad

Missing before **version 1.3**.

6	365	<id> *Result Type*	Result <id>	Scope <id> *Execution*	<id> *Value*	<id> *Index*

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 must be **Workgroup** or **Subgroup** Scope.

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 of *Direction* is evaluated such that:
0 indicates a horizontal swap within the quad.
1 indicates a vertical swap within the quad.
2 indicates a diagonal swap within the quad.

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*

A Changes

A.1 Changes from Version 0.99, Revision 31

- Added the **PushConstant** Storage Class.
- Added OpIAddCarry, OpISubBorrow, OpUMulExtended, and OpSMulExtended.
- Added OpInBoundsPtrAccessChain.
- Added the Decoration **NoContraction** to prevent combining multiple operations into a single operation (bug 14396).
- Added sparse texturing (14486):
  - Added **OpImageSparse…** for accessing images that might not be resident.
  - Added **MinLod** functionality for accessing images with a minimum level of detail.
- Added back the **Alignment** Decoration, for the **Kernel** capability (14505).
- Added a **NonTemporal** Memory Access (14566).
- Structured control flow changes:
  - Changed structured loops to have a structured continue *Continue Target* in OpLoopMerge (14422).
  - Added rules for how "fall through" works with **OpSwitch** (13579).
  - Added definitions for what is "inside" a structured control-flow construct (14422).
- Added **SubpassData** Dim to support input targets written by a previous subpass as an output target (14304). This is also a Decoration and a Capability, and can be used by some image ops to read the input target.
- Added OpTypeForwardPointer to establish the Storage Class of a forward reference to a pointer type (13822).
- Improved Debuggability
  - Changed OpLine to not have a target <*id*>, but instead be placed immediately preceding the instruction(s) it is annotating (13905).
  - Added OpNoLine to terminate the affect of **OpLine** (13905).
  - Changed OpSource to include the source code:
    * Allow multiple occurrences.
    * Be mixed in with the OpString instructions.
    * Optionally consume an OpString result to say which file it is annotating.
    * Optionally include the source text corresponding to that OpString.
    * Included adding OpSourceContinued for source text that is too long for a single instruction.
- Added a large number of Capabilities for subsetting functionality (14520, 14453), including 8-bit integer support for OpenCL kernels.
- Added **VertexIndex** and **InstanceIndex** BuiltIn Decorations (14255).
- Added **GenericPointer** capability that allows the ability to use the **Generic** Storage Class (14287).
- Added **IndependentForwardProgress** Execution Mode (14271).
- Added OpAtomicFlagClear and OpAtomicFlagTestAndSet instructions (14315).
- Changed OpEntryPoint to take a list of **Input** and **Output** <*id*> for declaring the entry point's interface.
- Fixed internal bugs
  - 14411 Added missing documentation for mad_sat OpenCL extended instructions (enums existed, just the documentation was missing)
  - 14241 Removed shader capability requirement from **OpImageQueryLevels** and **OpImageQuerySamples**.
  - 14241 Removed unneeded OpImageQueryDim instruction.

– 14241 Filled in *TBD* section for OpAtomicCompareExchangeWeek
– 14366 All OpSampledImage must appear before uses of sampled images (and still in the first block of the entry point).
– 14450 DeviceEnqueue capability is required for OpTypeQueue and OpTypeDeviceEvent
– 14363 OpTypePipe is opaque - moved packet size and alignment to opcodes
– 14367 Float16Buffer capability clarified
– 14241 Clarified how OpSampledImage can be used
– 14402 Clarified OpTypeImage encodings for OpenCL extended instructions
– 14569 Removed mention of non-existent OpFunctionDecl
– 14372 Clarified usage of OpGenericPtrMemSemantics
– 13801 Clarified the **SpecId** Decoration is just for constants
– 14447 Changed literal values of Memory Semantic enums to match OpenCL/C++11 atomics, and made the Memory Semantic **None** and **Relaxed** be aliases
– 14637 Removed subgroup scope from OpGroupAsyncCopy and OpGroupWaitEvents

A.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.

A.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**

A.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".

A.5 Changes from Version 1.00, Revision 3

- Added definition of derivative group, and use it to say when derivatives are well defined.

A.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

A.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.

A.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.

A.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.

A.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*.

A.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

A.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.

A.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

A.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

A.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).

A.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).

A.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).

A.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).

A.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).

A.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).

A.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.

A.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**.

A.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).

A.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.

A.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**.

A.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.

A.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.

A.28 Changes from Version 1.3, Revision 2

- New enumarents:
  - 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 refering to them from OpImageRead and OpImageWrite.
  - Khronos SPIR-V Issue #333: Be clear about overflow behavior for OpIAdd, OpISub, and OpIMul.

A.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.

A.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.

A.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 derocations 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.

A.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).

A.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.

A.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.