HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)

Publication #: 49828 • Rev: Version 0.95 • Issue Date: 7 March 2013

Acknowledgements

The HSAIL specification is the result of the contributions of many people. Here is a partial list of the contributors, including the company that they represented at the time of their contribution:

- Paul Blinzer AMD
- Mark Fowler AMD
- Mike Houston AMD
- Lee Howes AMD
- Roy Ju AMD
- Bill Licea-Kane AMD
- Leonid Lobachev AMD
- Mike Mantor AMD
- Vicki Meagher AMD
- Dmitry Preobrazhensky AMD
- Phil Rogers AMD
- Norm Rubin AMD
- Benjamin Sander AMD
- Elizabeth Sanville AMD
- Oleg Semenov AMD
- Brian Sumner AMD
- Yaki Tebeka AMD
- Tony Tye AMD
- Micah Villmow AMD
- Jem Davies ARM
- Ian Devereux ARM
- Robert Elliott ARM
- Alexander Galazin ARM
- Rune Holm ARM
- Kurt Shuler Arteris
- Greg Stoner HSA Foundation
- Theo Drane Imagination Technologies
- Yoong-Chert Foo Imagination Technologies

- John Howson Imagination Technologies
- Georg Kolling Imagination Technologies
- James McCarthy Imagination Technologies
- Jason Meridith Imagination Technologies
- Mark Rankilor Imagination Technologies
- Chien-Ping Lu MediaTek Inc.
- Thomas Jablin MulticoreWare Inc.
- Chuang Na MulticoreWare Inc.
- Greg Bellows Qualcomm
- P.J. Bostley Qualcomm
- Alex Bourd Qualcomm
- Ken Dockser Qualcomm
- Jamie Esliger Qualcomm
- Ben Gaster Qualcomm
- Andrew Gruber Qualcomm
- Wilson Kwan Qualcomm
- Bob Rychlik Qualcomm
- Ignacio Llamas Samsung Electronics Co, Ltd
- Soojung Ryu Samsung Electronics Co, Ltd
- Matthew Locke Texas Instruments
- Chelsi Odegaard VTM Group

About the HSA Programmer's Reference Manual

This document describes the Heterogeneous System Architecture Intermediate Language (HSAIL), which is a virtual machine and an intermediate language.

This document serves as the specification for the HSAIL language for HSA implementers. Note that there are a wide variety of methods for implementing these requirements.

Audience

This document is written for developers involved in developing an HSA implementation.

Terminology

This document shows new terms in italics. See Appendix C Glossary of HSAIL Terms (p. 343) for their definitions.

HSA Information Sources

- *HSA Programmer's Reference Manual* - publication # 49828
- *HSA Software System Architecture Specification* - publication # 51958
- *HSA Hardware System Architecture Specification* - publication # 50830
- The OpenCL™ Specification: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Contents

Figures

Chapter 2 HSAIL Programming Model

Chapter 4 HSAIL Syntax and Semantics

Chapter 5 Arithmetic Operations

Chapter 6 Memory Operations

Tables

Chapter 2 HSAIL Programming Model

Chapter 4 HSAIL Syntax and Semantics

Chapter 5 Arithmetic Operations

Chapter 6 Memory Operations

Chapter 7 Image Operations

Chapter 8 Branch Operations

Chapter 9 Parallel Synchronization and Communication Operations

Chapter 11 Operations Related to Functions

Chapter 12 Special Operations

Chapter 14 Directives

Chapter 19 BRIG: HSAIL Binary Format

Chapter 1

Overview

This chapter provides an overview of Heterogeneous System Architecture Intermediate Language (HSAIL).

1.1 What Is HSAIL?

The Heterogeneous System Architecture (HSA) is designed to efficiently support a wide assortment of data-parallel and task-parallel programing models. A single HSA system can support multiple instruction sets based on CPU(s), GPU(s), and specialized processor(s).

HSA supports two machine models: large mode (64-bit address space) and small mode (32-bit address space).

Programmers normally build code for HSA in a virtual machine and intermediate language called HSAIL (Heterogeneous System Architecture Intermediate Language). Using HSAIL allows a single program to execute on a wide range of platforms, because the native instruction set has been abstracted away.

HSAIL is required for parallel computing on an HSA platform.

This manual describes the HSAIL virtual machine and the HSAIL intermediate language.

An *HSA implementation* consists of:

- Hardware components that execute one or more machine instruction set architectures (ISAs). Supporting multiple ISAs is a key component of HSA.
- A compiler, linker, and loader.
- A *finalizer* that translates HSAIL code into the appropriate native ISA if the hardware components cannot support HSAIL natively.
- A runtime system.

Each implementation is able to execute the same HSAIL virtual machine and language, though different implementations might run at different speeds.

A device that participates in the HSA memory model is called an *agent*.

An HSAIL virtual machine consists of multiple agents including at least one host CPU and one HSA component:

- A *host CPU* is an agent that also supports the native CPU instruction set and runs the host operating system and the HSA runtime. As an agent, the host CPU can dispatch commands to an HSA component using memory operations to construct and enqueue *AQL* packets. In some systems, a host CPU can also act as an HSA component (with appropriate HSAIL finalizer and AQL mechanisms).
- An *HSA component* is an agent that supports the HSAIL instruction set and the AQL packet format. As an agent, an HSA component can dispatch commands to any HSA component (including itself) using memory operations to construct and enqueue AQL packets.

Neither kind of compute unit needs to execute HSAIL code directly.

Different implementations can choose to invoke the finalizer at various times: statically at the same time the application is built, when the application is installed, when it is loaded, or even during execution.

An HSA-enabled application is an amalgam of both of the following:

- Code that can execute only on host CPUs
- HSAIL code, which can execute only on HSA components

Certain sections of code, called *kernels*, are executed in a data-parallel way by HSA components. Kernels are written in HSAIL and then separately translated (statically, at install time, at load time, or dynamically) by a finalizer to the target instruction set.

A kernel does not return a value.

HSAIL supports two machine models:

- Large mode (addresses are 64 bits)
- Small mode (addresses are 32 bits)

For more information, see 2.10 Small and Large Machine Models (p. 20).

1.2 HSAIL Virtual Language

HSAIL is a virtual instruction set designed for parallel processing which can be translated on-the-fly into many native instruction sets. Internally, each implementation of HSA might be quite different, yet all implementations will run any program written in HSAIL, provided it supports the profile used (see Chapter 17 Profiles (p. 249)). HSAIL has no explicit parallel constructs; instead, each kernel contains operations for a single work-item.

When the kernel starts, a multidimensional cube-shaped *grid* is defined and one *work-item* is launched for each point in the grid. A typical grid will be large, so a single kernel might launch thousands of work-items. Each launched work-item executes the same kernel code, but might take different control flow paths. Execution of the kernel is complete when all work-items of the grid have been launched and have completed their execution.

Work-items are extremely lightweight, meaning that the overhead of context switching among work-items is not a costly operation.

An HSAIL program looks like a simple assembly language program for a RISC machine, with text written as a sequence of characters.

See Chapter 3 Examples of HSAIL Programs (p. 25).

Most lines of source text contain operations made up of an opcode with a set of suffixes specifying data type, length, and other attributes. Operations in HSAIL are simple three-operand, RISC-like constructs. There are also assorted pseudo-operations used to declare variables.

All mathematical operations are register-to-register only. For example, to multiply two numbers, the values are loaded into registers and one of the multiply operations (`mul_s32`, `mul_u32`, `mul_s64`, `mul_u64`, `mul_f32`, or `mul_f64`) is used.

Each HSAIL program has its own set of resources. For example, each work-item has a private set of registers.

HSA has a unified memory model, where all HSAIL work-items and agents can use the same pointers, and a pointer can address any kind of HSA memory. Programmers are relieved of much of the burden of memory management. The HSA system determines if a load or store address should be visible to all agents in the system (global memory), visible only to work-items in a group (group memory), or private to a work-item (private memory). The same pointer can be used by all agents in the system including all host CPUs and all HSA components. Global memory (but not group memory or private memory) is coherent between all agents.

Chapter 2

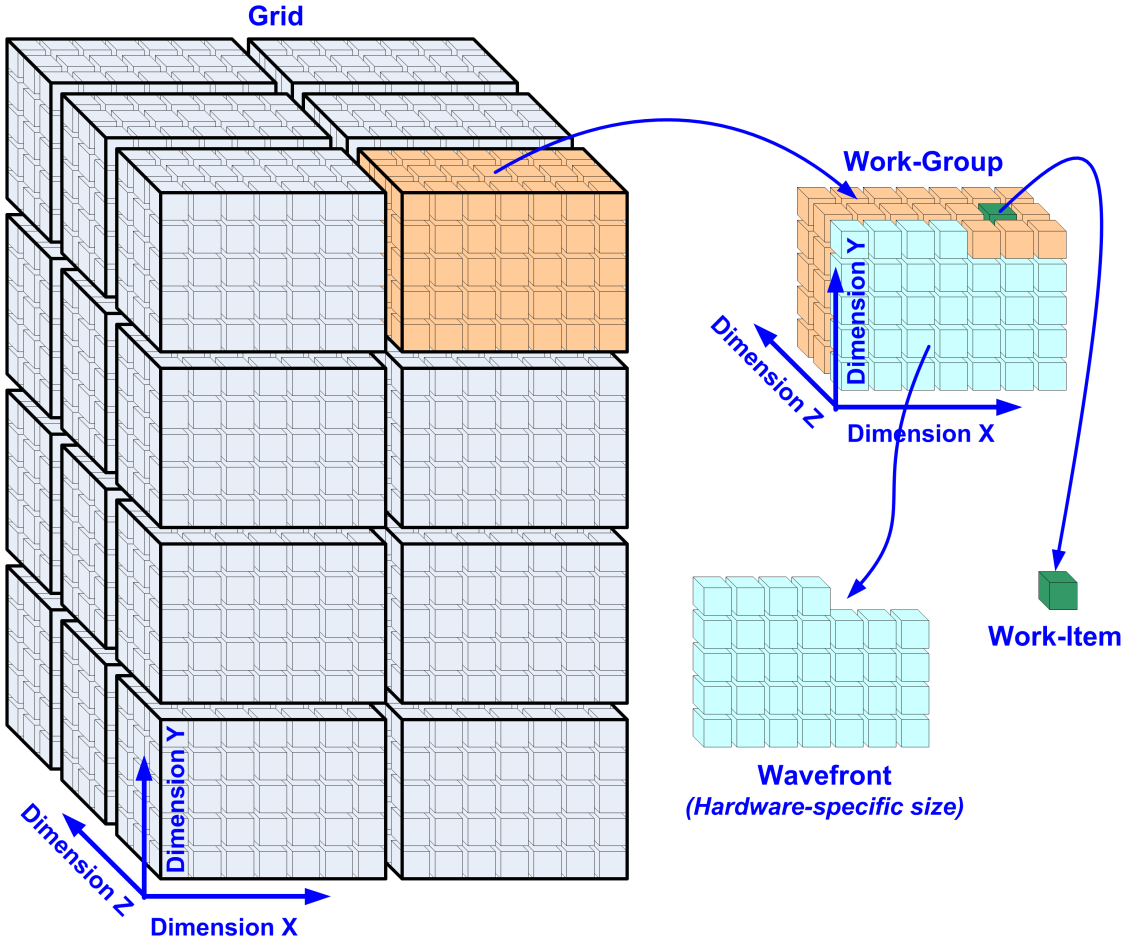
HSAIL Programming Model

This chapter describes the HSAIL programming model.

2.1 Overview of Grids, Work-Groups, and Work-Items

The figure below shows a graphical view of the concepts that affect an HSAIL implementation.

Figure 2–1 A Grid and Its Work-Groups and Work-Items



Programmers, compilers, and tools identify a portion of an application that is executed many times, but independently on different data. They can structure that code into a kernel that will be executed by many different work-items.

The kernel language runtime can be used to invoke the kernel language compiler that will produce HSAIL. The HSA runtime can then be used by the language runtime to execute the finalizer for the HSA component that will execute the kernel. The finalizer takes the HSAIL and produces kernel ISA that will execute on that HSA component. If the HSAIL requires more resources than are available on the device, it will return a failure result. For example, the kernel might require more group memory, or more fbarriers than are available on the device.

An HSA component can have multiple AQL queues associated with it. Each queue has a *queue ID*, which is unique across all the queues currently created by the process executing the program.

A request to execute a kernel is made by appending an AQL dispatch packet on a queue associated with an HSA component. Each dispatch packet is assigned a *dispatch ID* that is unique for each queue.

Each HSA component services all the queues associated with it, and dispatches the kernel ISA associated with the queued dispatch packets, which causes the kernel to be executed. If the HSA component has insufficient resources to execute at least one work-group, then the dispatch fails, no kernel execution occurs, and the dispatch completion object indicates a failure. For example, the dispatch might request more dynamic group memory than is available. A dispatch may, but is not required to, fail if the dispatch arguments are not compatible with any control directives specified when the kernel was finalized. For example, the dispatch work-group size might not match the values specified by a `requiredworkgroupsize` control directive.

The combination of the dispatch ID and the queue ID is globally unique. Operations in a kernel can access these IDs by means of the `dispatchid` and `qid` special operations. (See Chapter 12 Special Operations (p. 217).)

The dispatch forms a *grid*. The grid can be composed of one, two, or three dimensions. The dimension components are referred to as X, Y, and Z. If the grid has one dimension, then it has only an X component, if it has two dimensions, then it has X and Y components, and if it has three dimensions, it has X, Y, and Z components.

A grid is a collection of *work-items*. (See 2.3 Work-Items (p. 8).

The work-items in the grid are partitioned into *work-groups* that have the same number of dimensions as the grid. (See 2.2 Work-Groups (p. 7).)

A work-group is an instance of execution on the HSA component. Execution is performed by a compute unit. An HSA component can have one or more compute units.

When a kernel is dispatched, the number of dimensions of the grid (which is also the number of dimensions of the work-group), the size of each grid dimension, the size of each work-group dimension, and the kernel argument values must be specified. If the number of dimensions specified for a kernel dispatch is 1, then the Y and Z components for the grid and work-group size must be specified as 1; if the number of dimensions specified for a kernel dispatch is 2, then the Z component for the grid and work-group size must be specified as 1.

As execution proceeds, the work-groups in the grid are distributed to compute units. All work-items of a work-group are executed on the same compute unit at the same time, each work-item running the kernel. Execution can either be concurrent, or through some form of scheduling. (See 2.6 Wavefronts, Lanes, and Wavefront Sizes (p. 11).)

The grid size is not required to be an integral multiple of the work-group size, so the grid might contain partial work-groups. In a partial work-group, only some of the

work-items are valid. The compute unit will only execute the valid work-items in a partial work-group.

A compute unit may execute multiple work-groups at the same time. The resources used by a work-group (such as group memory, barrier and fbarrier resources, and number of wavefronts that can be scheduled) and work-items within the work-group (such as registers) may limit the number of work-groups that a compute unit can execute at the same time. However, a compute unit must be able to execute at least one work-group. If an HSA component has more than one compute unit, different work-groups may execute on different compute units.

In the figure, the grid is composed of 24 work-groups. (Dimension X = 2, dimension Y = 4, and dimension Z = 3.)

In the figure, each work-group is a three-dimensional work-group, and each work-group is composed of 105 work-items. (Dimension X = 7, dimension Y = 5, and dimension Z = 3.)

For information about wavefronts, see 2.6 Wavefronts, Lanes, and Wavefront Sizes (p. 11).

2.2 Work-Groups

A work-group is an instance of execution in a compute unit. A compute unit must have enough resources to execute at least one work-group at a time. Thus, it is not possible for a compute unit to be too small.

Assorted synchronization operations can be used to control communication within a work-group. For example, it is possible to mark barrier synchronization points where work-items wait until other work-items in the work-group have arrived.

All implementations can execute at least the number of work-items in a work-group such that they are all guaranteed to make forward progress in the presence of work-group barriers.

Implementations that provide multiple compute units or more capable compute units can execute multiple work-groups simultaneously.

2.2.1 Work-Group ID

Every work-group has a multidimensional identifier containing up to three integer values (for the three dimensions) called the *work-group ID*. The work-group ID is calculated by dividing each component of the work-item absolute ID by the corresponding work-group size component and ignoring the remainder. (See 2.3.3 Work-Item Absolute ID (p. 9).)

Work-group size is the product of the three dimensions:

$$\text{work-group size} = \text{workgroupsize}_0 * \text{workgroupsize}_1 * \text{workgroupsize}_2$$

Each work-group can access assorted predefined read-only values such as work-group ID, work-group size, and so forth through the use of special operations. See Chapter 12 Special Operations (p. 217).)

The value of the work-group ID is returned by the `workgroupid` operation.

The size of the work-group specified when the kernel was dispatched is returned by the `workgroupsize` operation.

Because the grid is not required to be an integral multiple of the work-group size, there can be partial work-groups. The `currentworkgroupsize` operation returns the work-group size that the current work-item belongs to. The value returned by this operation will only be different from that returned by `workgroupsize` operation if the current work-item belongs to a partial work-group.

2.2.2 Work-Group Flattened ID

Each work-group has a *work-group flattened ID*.

The work-group flattened ID is defined as:

```
work-group flattened ID = work-item flattened absolute ID / work-group size
```

HSAIL implementations need to ensure forward progress. That is, any program can count on one-way communication and later work-groups (in work-group flattened ID order) can wait for values written by earlier work-groups without deadlock.

2.3 Work-Items

Each work-item has its own set of registers, has private memory, and can access assorted predefined read-only values such as work-item ID, work-group ID, and so forth through the use of special operations. See Chapter 12 Special Operations (p. 217).)

To access private memory, work-items use regular loads and stores, and the HSA hardware will examine addresses and detect the ranges that are private to the work-item. One of the system-generated values tells the work-item the address range for private data.

Work-items are able to share data with other work-items in the same work-group through a memory segment called the *group segment*. Memory in a group segment is accessed using loads and stores. This memory is not accessible outside its associated work-group (that is, it is not seen by other work-groups or agents). See 2.8 Segments (p. 13).

2.3.1 Work-Item ID

Each work-item has a multidimensional identifier containing up to three integer values (for the three dimensions) within the work-group called the *work-item ID*.

`max` is the size of the work-group or 1.

For each dimension i, the set of values of ID_i is the dense set `[0, 1, 2, ... max`$_i$ `- 1]`.

The value of max_i can be accessed by means of the special operation `workgroupsize`.

The work-item ID can be accessed by means of the special operation `workitemid`.

2.3.2 Work-Item Flattened ID

The work-item ID can be flattened into one dimension, which is relative to the containing work-group. This is called the *work-item flattened ID*.

The work-item flattened ID is defined as:

$\texttt{work-item flattened ID} = ID_0 + ID_1 * max_0 + ID_2 * max_0 * max_1$

where:

ID_0 = workitemid (dimension 0)
ID_1 = workitemid (dimension 1)
ID_2 = workitemid (dimension 2)
max_0 = workgroupsize (dimension 0)
max_1 = workgroupsize (dimension 1)

The work-item flattened ID can be accessed by means of the special operation `workitemflatid`.

2.3.3 Work-Item Absolute ID

Each work-item has a unique multidimensional identifier containing up to three integer values (for the three dimensions) called the *work-item absolute ID*. The work-item absolute ID is unique within the grid.

Programs can use the work-item absolute IDs to partition data input and work across the work-items.

For each dimension *i*, the set of values of absolute ID_i are the dense set [0, 1, 2, ... max_i – 1].

The value of max_i can be accessed by means of the special operation `gridsize`.

The work-item absolute ID can be accessed by means of the special operation `workitemabsid`.

2.3.4 Work-Item Flattened Absolute ID

The work-item absolute ID can be flattened into one dimension into an identifier called the *work-item flattened absolute ID*. The work-item flattened absolute ID enumerates all the work-items in a grid.

The work-item flattened absolute ID is defined as:

$\texttt{work-item flattened absolute ID} = ID_0 + ID_1 * max_0 + ID_2 * max_0 * max_1$

where:

ID_0 = workitemabsid (dimension 0)
ID_1 = workitemabsid (dimension 1)
ID_2 = workitemabsid (dimension 2)
max_0 = gridsize (dimension 0)
max_1 = gridsize (dimension 1)

The work-item flattened absolute ID can be accessed by means of the special operation `workitemflatabsid`.

2.4 Scalable Data-Parallel Computing

For CPU developers, the idea of work-items and work-groups might seem odd, because one level of threads has traditionally been enough.

Work-items are similar in some ways to traditional CPU threads, because they have local data and a program counter. But they differ in a couple of important ways:

- Work-items can be gang-scheduled while CPU threads are scheduled separately.
- Work-items are extremely lightweight. Thus, a context change between two work-items is not a costly operation.

The number of work-groups that can be processed at once is dependent on the amount of hardware resources. Adding work-groups makes it possible to abstract away this concept so that developers can apply a kernel to a large grid without worrying about fixed resources. If hardware has few resources, it executes the work-groups sequentially. But if it has a large number of compute units, it can process them in parallel.

2.5 Active Work-Groups and Active Work-Items

At any instance of time, the work-groups executing in compute units are called the *active work-groups*. When a work-group finishes execution, it stops being active and another work-group can start. The work-items in the active work-groups are called *active work-items*. Resource limits, including group memory, can constrain the number of active work-groups.

An active work-item at an operation is one that executes the current operation. For example:

```
if (condition) {
    operation;
}
```

The active work-items at this `operation` are the work-items where `condition` was true.

Resource limits might constrain the number of active work-items. However, every HSAIL implementation must be able to support enough active work-items to be able to execute at least one maximum-size work-group. Resources such as private memory and registers are not persistent over work-items, so implementations are allowed to reuse resources. When a work-group finishes, it and all its work-items stop being active and the resources they used (private memory, registers, group memory, hardware resources used to implement barriers, and so forth) might be reassigned.

Work-group ($i+j$) might start after work-group (i) finishes, so it is not valid for a work-group to wait on an operation performed by a later work-group.

When a work-group finishes, the associated resources become free so that another work-group can start.

2.6 Wavefronts, Lanes, and Wavefront Sizes

Work-items within a work-group can be executed in an extended SIMD (single instruction, multiple data) style. That is, work-items are gang-scheduled in chunks called *wavefronts*. Executing work-items in wavefronts can allow implementations to improve computational density.

Work-items are assigned to wavefronts in work-item flattened absolute ID order: X then Y then Z. This can be useful to expert programmers. (See 2.3.4 Work-Item Flattened Absolute ID (p. 9).)

A *lane* is an element of a wavefront. The *wavefront size* is the number of lanes in a wavefront. Wavefront size is an implementation-defined constant, and must be a power of 2 in the range from 1 to 64 inclusive. Thus, a wavefront with a wavefront size of 64 has 64 lanes.

If the work-group size is not a multiple of the wavefront size, the last wavefront will have extra lanes that do not contribute to the computation.

Two work-items in the same work-group will be in the same wavefront if the floor of `work-item flattened absolute ID / wavefront size` is the same.

2.6.1 Example of Contents of a Wavefront

Assume that the work-group size is 13 (X dimension) by 3 (Y dimension) by 11 (Z dimension) and the wavefront size is 64. Thus, a work-group would need 13 * 3 * 11 = 429 work-items. The number of work-items divided by 64 = 6 with a remainder of 45.

Six wavefronts (wavefronts 0, 1, 2, 3, 4, and 5) would hold 384 work-items. The remaining 45 work-items would be in the seventh wavefront (wavefront 6), which would be partially filled.

See the tables below.

Table 2–1 Wavefronts 0 Through 6

Wavefront 0					
Dimensions X, Y, Z	0-12, 0, 0	0-12, 1, 0	0-12, 2, 0	0-12, 0, 1	0-11, 1, 1
Work-Item Absolute Flattened IDs	0-12	13-25	26-38	39-51	52-63
Lane IDs	0-12	13-25	26-38	39-51	52-63

Wavefront 1						
Dimensions X, Y, Z	12, 1, 1	0-12, 2, 1	0-12, 0, 2	0-12, 1, 2	0-12, 2, 2	0-10, 0, 3
Work-Item Absolute Flattened IDs	64	65-77	78-90	91-103	104-116	117-127
Lane IDs	0	1-13	14-26	27-39	40-52	53-63

Wavefront 2						
Dimensions X, Y, Z	11-12, 0, 3	0-12, 1, 3	0-12, 2, 3	0-12, 0, 4	0-12, 1, 4	0-9, 2, 4
Work-Item Absolute Flattened IDs	128-129	130-142	143-155	156-168	169-181	182-191
Lane IDs	0-1	2-14	15-27	28-40	41-53	55-63

Wavefront 3						
Dimensions X, Y, Z	10-12, 2, 4	0-12, 0, 5	0-12, 1, 5	0-12, 2, 5	0-12, 0, 6	0-8, 2, 4
Work-Item Absolute Flattened IDs	192-194	195-207	208-220	221-233	234-246	247-255
Lane IDs	0-2	3-15	16-28	29-41	42-54	53-63

Wavefront 4						
Dimensions X, Y, Z	9-12, 1, 6	0-12, 2, 6	0-12, 0, 7	0-12, 1, 7	0-12, 2, 7	0-7, 0, 8
Work-Item Absolute Flattened IDs	256-259	260-272	273-285	286-298	299-311	312-319
Lane IDs	0-3	4-16	17-29	30-42	43-55	56-63

Wavefront 5						
Dimensions X, Y, Z	8-12, 0, 8	0-12, 1, 8	0-12, 2, 8	0-12, 0, 9	0-12, 1, 9	0-6, 2, 9
Work-Item Absolute Flattened IDs	320-324	325-337	338-350	351-363	364-376	377-383
Lane IDs	0-4	5-17	18-30	31-43	44-56	57-63

Wavefront 6				
Dimensions X, Y, Z	7-12, 2, 9	0-12, 0, 10	0-12, 1, 10	0-12, 2, 10
Work-Item Absolute Flattened IDs	384-389	390-402	403-415	416-428
Lane IDs	0-5	6-18	19-31	32-44

The rest of wavefront 6 is unused.

2.6.2 Wavefront Size

On some implementations, a kernel might be more efficient if it is written with knowledge of the wavefront size. Thus, HSAIL includes a compile-time macro, `WAVESIZE`, which can be used in any operation where an integer or bit immediate value is allowed, and as the argument to the width modifier (see 2.13.1 Width Modifier (p. 22)).

`WAVESIZE` is only available inside the HSAIL code.

In Extended Backus-Naur Form, `WAVESIZE` is called `TOKEN_WAVESIZE`.

Developers need to be careful about wavefront size assumptions, because programs coded for a single wavefront size could generate wrong answers or deadlock if the code is executed on implementations with a different wavefront size.

The grid size does not need to be an integral multiple of the wavefront size.

2.7 Types of Memory

HSAIL memory is organized into three types:

- Flat memory

  Flat memory is a simple interface using byte addresses. Loads and stores can be used to reference any visible location in the flat memory.

  For more information, see 2.8 Segments (p. 13) and 2.9 Flat Memory and Agents (p. 18).

- Registers

  There are four register sizes:

  - 1-bit
  - 32-bit
  - 64-bit
  - 128-bit

  Registers are untyped.

  For more information, see 4.12 Registers (p. 40).

- Image memory

  Image memory is a special kind of memory access that can make use of dedicated hardware often provided for graphics. Only programmers seeking extreme performance need to understand image memory.

  For more information, see Chapter 7 Image Operations (p. 151).

All HSAIL implementations support all three types of memory.

2.8 Segments

Flat memory is divided into segments based on:

- The way data can be shared
- The intended usage

A *segment* is a block of memory. The characteristics of a segment space include its size, addressability, access speed, access rights, and level of sharing between work-items.

The segment determines the part of memory that will hold the object, how long the storage allocation exists, and the properties of the memory. The finalizer uses the

segment to determine the intended usage of the memory. There is no bounds checking over segments.

No access protection between segments is provided. That is, the behavior is undefined when memory operations generate addresses that are outside the bounds of a segment.

2.8.1 Types of Segments

There are seven types of segments:

- *Global*

  Memory that is visible to all work-groups and to all agents.

  This segment supports read/write access for all work-items in all work-groups. All agents including HSA components can read and write global memory.

  All global memory is persistent across the application.

- *Group*

  Memory that is visible to a single work-group.

  This segment can be used to hold variables that are shared by all work-items in a work-group. An address in group memory can be read and written by any work-item in the group.

  If an implementation uses regular memory to implement group memory, it must adjust the group segment addresses used by work-items in one work-group so that accesses by work-items in a different work-group access different memory locations for the same group segment address.

  Group memory is uninitialized when the work-group starts execution.

  It is undefined whether or not group memory is visible to other agents.

  One specific implementation-defined range of flat addresses is reserved for group memory.

- *Private*

  Memory that is visible only to a single work-item.

  Although it is permitted for an implementation to allow other work-items to access the values in a different work-item's private memory, a program is undefined if it does so. For example, an implementation may use global memory to implement private memory and so a work-item could use a global or flat address to access the private memory of any work-item.

  If an implementation uses regular memory to implement private memory, it must adjust the private segment addresses used by each work-item so that different work-items access different memory locations for the same private segment address.

  It is implementation-defined whether or not private memory is visible to other agents.

  Private memory is uninitialized when the work-item starts. The finalizer is free to remove loads and stores to private memory if this does not change the single work-item answer without regard to exceptions. For example, a floating-point divide can be removed if its only effect is to cause a divide by zero exception.

- *Kernarg*

  Read-only memory used to pass arguments into a kernel.

  Implementations are allowed either to provide special hardware or to use kernarg memory.

  Implementations are allowed to treat kernarg and global memory as though they are a single segment.

  For more information, see 4.25 Kernarg Segment (p. 63).

- *Readonly*

  Read-only memory.

  Readonly memory remains constant during the execution of a kernel.

  The result of a store or atomic operation to readonly memory (such as by using a flat address) is undefined, and it is implementation-defined if such accesses will be detected and generate an exception.

  Implementations are allowed to treat readonly and global memory as though they are a single segment.

  The finalizer might place a variable in readonly memory in specialized read-only caches.

- *Spill*

  Memory that is visible only to a single work-item.

  Used to load or store register spills. This segment is used as a hint to the finalizer so it can generate better code.

  The finalizer might remove operations using the spill segment.

  Implementations are allowed to treat spill and private memory as though they are a single segment.

- *Arg*

  Memory that is visible only to a single work-item.

  Used to pass arguments into and out of functions.

  Implementations are allowed to provide special hardware to accelerate arg memory.

  Implementations are allowed to treat arg and private memory as though they are a single segment.

  For more information, see 10.4 Arg Segment (p. 207).

See also 5.15 Segment Checking (segmentp) Operation (p. 110) and 5.16 Segment Conversion Operations (p. 111).

2.8.2 Shared Virtual Memory

Shared virtual memory is a basis of HSA. It means:

- A single work-item sees a flat address space.

  Within that address space, certain address ranges are group memory, other ranges are private, and so on. Implementations use the address to determine the kind of memory. Consequently, compilers need not generate special forms of loads and stores for each type of memory. Pointers to memory can be freely cast to integer and back without problems.

- Non-shared objects are hidden.

  This means that each object is declared to be in one of three sharing levels: shared over all work-items (global), shared over the work-group (group), or never shared (private).

  The private segments for each work-item overlay. Overlaying means that reads and writes to address *X* in work-item 1 access work-item 1's private data, while reads and writes to the same address *X* in work-item 2 access different storage. Thus, if work-item 1 declares a private variable at address *X*, then work-item 2 cannot read or write the variable. (Spill segments behave in the same way.)

  Similarly, every work-group sees only its own group segment, which is shared by the work-items within the work-group, so no work-group can access the group memory of another work-group.

  Every work-item and agent sees the same global memory.

2.8.3 Addressing for Segments

Memory operations can use a flat address or specify the particular segment used.

If they use flat addresses, implementations will recognize when an address is within a particular segment.

If they specify the particular segment used, the address is relative to the start of the segment.

If an address in group memory for work-group A is stored in global memory and then is accessed by a different work-group B, the results are undefined.

When a flat memory operation addresses location P, the address P is translated to an effective address Q as follows:

1. If P is inside the flat address bounds of the private, spill, or arg memory segments, then Q is set to an implementation-defined function of (P – start of the segment) and the work-item absolute ID. The implementation-defined function is intended to enable optimized memory layouts such as interleaving the memory locations accessible by each work-item to improve the memory access pattern of the gang-scheduled wavefronts.
2. If P is inside the flat address bounds of the group memory segment, then Q is set to an implementation-defined function of (P – start of the group segment) and the work-group absolute ID.
3. If P is not inside the flat address bounds of the private, spill, arg, or group memory segments, then Q is set to an implementation-defined function of P. The implementation-defined function is intended to enable optimized memory layouts such as interleaving or tiling.

Implementations can provide special hardware to accelerate this translation.

If two work-items try to reference the same address in private, spill, or arg memory, step 2 above will ensure that the effective addresses are different. This guarantees that private really is private, and allows programs to address private memory without complex addressing.

For example, if the private segment started at address 1000 and ended at 2000, then the private segment for work-group A might be from 1000 to 1255, while work-group B might use 1256 to 1511, and so forth.

If work-item 0 in work-group A used segment-relative address 100, it would address 1100, while if work-item 0 in work-group B used the same relative address 100, it would address 1356.

A memory operation can be marked with a segment. In that case, the address in the operation is treated as segment-relative.

For more information, see 6.1 Memory and Addressing (p. 123).

2.9 Flat Memory and Agents

All HSAIL implementations can map some flat memory into the address space accessible to agents. See Table 2–2 (p. 18).

Table 2–2 Flat Memory and Agents

Segment	HSAIL	Can be initialized?	Persistence	Agent interaction	Combinable?
Global	General global space	Yes	Application	Shared with all agents.	
Readonly	Read-only	Yes	Application	Can be initialized by host CPU.	Can be combined with global segment.
Kernarg	Holds kernel arguments; read-only	No	Kernel	Written by the agent when the kernel dispatch is queued.	Can be combined with global segment.
Group	Read-write	No	Work-group	It is implementation-defined if other agents can see group memory.	
Arg	Holds function arguments; input arguments are read-only; output arguments are write-only	No	Work-item	Input arguments initialized when function is called by kernel or another function. It is implementation-defined if other agents can see arg memory.	Can be combined with private segment.
Private		No	Work-item	It is implementation-defined if other agents can see private memory.	
Spill	Holds spilled register values; read-write	No	Work-item	It is implementation-defined if other agents can see spill memory.	Can be combined with private segment.

The same global and readonly segment address, and its corresponding flat address, can be used in two different work-items (which can be in different kernels and different dispatches of the same kernel) and in an agent to access the same memory location. Standard page protections (read-only/read-write/protected, and so forth) apply to global and readonly segment memory. See the *HSA System Architecture Specification* for information.

An implementation is not required to use read-only protection on the readonly segment variables.

Global and readonly memory may be visible to agents. Runtime libraries can provide a way for applications to indicate that some global memory is not accessible to agents.

An implementation can map group addresses to special scratchpad memory allocated for each HSA component compute unit, to addresses in global memory, or to addresses in other agent memory. It is implementation-defined whether or not other agents can access group memory.

Private, spill, and arg memory can be expanded to multiple addresses in the address space (one for every active work-item) or can be implemented by special hardware. It is implementation-defined whether or not other agents can access private, spill, or arg memory.

2.9.1 Persistence Rules

The persistence of a memory segment specifies how stores in the segment can be seen by other loads.

Each segment has one of the following persistence values:

- Application: stores in one kernel can be seen by loads of another kernel in the same application execution.
- Kernel: stores in one kernel execution can be seen by loads in the same kernel execution.
- Work-group: stores in work-items in one work-group can only be seen by loads in work-items in the same work-group.
- Work-item: stores in one work-item can only be seen by loads in the same work-item.

In addition, the scope of the declaration can further restrict if its value can be accessed. Private and spill variables declared in a function can only be accessed while the function is being executed by the work-item. Arg variables can only be accessed while the containing argument scope is being executed by the work-item. See 4.10 Storage Duration (p. 38).

The persistence also specifies if it is defined whether a segment address can be used in a memory access. It can only be used in the same persistence entity that created it. For example, if the persistence is application, then the address can be used to access the memory value in any work item in any kernel dispatched by the application. If the persistence is work-item, then only the work-item that created the address can access it.

The variable referenced by a segment address is only defined if the value it references is defined. For example, it is not defined if a group segment address created in a work-item of one work-group will access the same named variable in a work-item of another work-group.

If a segment address is converted to a flat address, it is only defined to convert the flat address back to a segment address of the original kind or to a segment that the implementation has combined with that segment. This is consistent with the results returned by the `segmentp` operation. See 5.15 Segment Checking (segmentp) Operation (p. 110).

If a segment address is converted to a flat address, and back to a segment address of a different segment kind that the implementation has combined with the original segment kind, then the address does not need to be the same value as the original address. However, accesses using it must behave the same as accesses using the original address. This allows `segmentp` to be used to determine a valid segment address to which the flat address can be converted. This can then be used to perform segment address accesses, which might perform better on some implementations than flat address accesses.

The persistence rules also apply to flat addresses. A flat address memory access is only defined if the memory access is defined for the original segment address.

It is only defined to convert a flat address to a segment address if the value accessed by the flat address is defined. For example, it is not defined to convert a private segment address into a flat address in one work-item, and then convert the flat address back to a private segment address in another work-item. It is not defined to access the private value in the first work-item, nor is it defined to access the value of the same named variable in the second work-item.

2.10 Small and Large Machine Models

HSAIL supports two machine models. Machine models determine the size of a data pointer and are not compatible. Table 2–3 (p. 20) shows the address sizes used for the two models supported by HSAIL.

The machine model of the HSAIL code executed by an HSA component must match the address space size of the process that owns the queue on which the kernel was dispatched. A process executing with a 32-bit address space size requires the HSAIL code to have the small machine model. A process executing with a 64-bit address space requires the HSAIL code to have the large machine model.

The small model might be appropriate for a legacy CPU 32-bit application that wants to use program data-parallel sections.

The model must be specified using the `version` statement. See 15.1 Syntax of the version Statement (p. 243).

Table 2–3 Machine Model Address Sizes

	Small	Large
Flat address	32-bit	64-bit
Global segment address	32-bit	64-bit
Readonly segment address	32-bit	64-bit
Kernarg segment address	32-bit	64-bit
Group segment address	32-bit	32-bit
Arg segment address	32-bit	32-bit
Private segment address	32-bit	32-bit
Spill segment address	32-bit	32-bit

	Small	Large
Label address	32-bit	64-bit
Function pointer	32-bit	64-bit
Fbarrier address	32-bit	32-bit
Address expression offset	32-bit	64-bit

The small machine model has these constraints:

- Stores may be done 32 bits at a time. A 64-bit store can be done in 32-bit chunks.
- Acquire and release can only be applied to 32-bit (or smaller) loads and stores.
- Pointers are 32 bits.
- 64-bit atomic operations are not supported.
- For register plus offset addressing, the offset must be 32 bits.

2.11 Base and Full Profiles

HSAIL provides two kinds of profiles:

- Base
- Full

HSAIL profiles are provided to guarantee that the implementation supports a required feature set and meets a given set of program limits. The strictly defined set of HSAIL profile requirements provides portability assurance to users that a certain level of support is present.

The profile must be specified using the `version` statement. See 15.1 Syntax of the version Statement (p. 243).

For more information, see Chapter 17 Profiles (p. 249).

2.12 Race Conditions

If multiple work-items access the same addresses in group or global memory and one of the accesses is a store, then it is possible to have a race condition.

In general, programs should add synchronization to avoid race conditions.

2.13 Divergent Control Flow

On HSA components with a wavefront size greater than 1, branches can introduce a performance issue called *divergent control flow*.

When a wavefront executes a conditional or indirect branch, it is possible that the work-items in the wavefront take different paths. This causes the wavefront to enter divergent control flow. Because SIMD implementations cannot execute different

instructions in the same cycle, executing in divergent control flow might be less efficient.

An implementation can improve performance in divergent control flow by reconverging the work-items. For example, given an IF/THEN/ELSE/ENDIF, the wavefront could diverge at the IF and reconverge at the ENDIF.

Implementations must reconverge no later than the immediate post-dominator (as described in 2.13.2 Post-Dominator and Immediate Post-Dominator (p. 23)), but may reconverge earlier.

Because implementations are allowed to execute the work-items in a wavefront in lockstep, it is illegal for a work-item in a wavefront to spin wait for a value written by a second work-item in the same wavefront.

Reliable communication between work-items requires synchronization. If one work-item writes into a location and a different work-item reads back the same location without using synchronization, the result is undefined.

2.13.1 Width Modifier

Because each implementation might have a different wavefront size, HSAIL provides a way to indicate that certain operations have behavior that changes depending on the wavefront size. Some HSAIL operations (specifically `ld`, `brn`, `cbr`, `barrier`, and `call`) take a width modifier. It is specified as `width(`*n*`)`, `width(WAVESIZE)`, or `width(all)`.

The value of *n* must be a power of 2 between 1 and 2^{31} inclusive. `width(WAVESIZE)` specifies the implementation-defined number of work-items in a wavefront (see 2.6 Wavefronts, Lanes, and Wavefront Sizes (p. 11)). `width(all)` specifies that *n* includes all work-items in the work-group. The default for the width modifier if it is omitted depends on the operation, and can either be `width(1)` or `width(all)`.

The width modifier conceptually divides the work-group's work-items into slices of size *n* work-items. Two work-items in the same work-group are in the same slice if the two work-items' flattened ID modulo *n* are the same.

For example, `barrier_width(`*n*`)` is a barrier operation specifying that a subset of all work-items within a work-group is participating in some form of communication at this point. The `barrier_width(`*n*`)` operation can be performed between the *n* work-items in the same slice. There is no requirement for the work-items in other slices to participate in the barrier, and no guarantees are made in this respect. `barrier_width(`*n*`)` indicates that no communication will happen between work-items that are in different slices.

The width modifier is only a performance hint. An implementation is allowed to ignore the width modifier and always synchronize with all work-items of the work-group. An implementation is also allowed to ignore the width modifier in these situations: *n* is greater than the wavefront size and *n* is not an exact multiple of the wavefront size; or *n* is less than or equal to wavefront size and the wavefront size is not an exact multiple of *n*. If *n* is less than or equal to wavefront size and the wavefront size is an exact multiple of *n*, then an implementation may be able to avoid using explicit code to implement barriers, because the gang-scheduling execution of wavefronts will ensure synchronization.

See also 8.3 Using the Width Modifier (p. 182).

2.13.2 Post-Dominator and Immediate Post-Dominator

The post-dominator of a branch operation *b* is defined as a point *p* in the program such that every path from the operation *b* that reaches the end of the function or kernel must go through *p*. No matter which path is taken out of *b*, control will eventually reach *p*. The immediate post-dominator is the unique point that does not post-dominate any other post-dominator of *b*.

For example:

```
cbr $c1, @x; // a conditional branch
// ...
@x:          // all code that leaves the cbr must eventually reach @x
// ...
@y:          // and that code must reach @y
```

In this example, both `@x` and `@y` are post-dominators of the branch, but only `x` is the immediate post-dominator.

2.14 Uniform Operations

An operation over a set of work-items is termed a *uniform operation* if all work-items in the set produce the same result. The set of work-items could be the grid, the work-group, the slice of work-items specified by the width modifier, or the wavefront.

For example, a load operation is uniform if all work-items in the set use the same value for the source address. For another example, a conditional branch operation is uniform if all work-items in the set either take the branch or do not take the branch.

If a finalizer can determine that an operation is uniform over all sets of work-items of a particular kind within the grid, it might be able to generate more efficient code by executing the operation once and broadcasting the result to all work-items in the set.

Chapter 3

Examples of HSAIL Programs

This chapter provides examples of HSAIL programs.

The syntax and semantics and operations are explained in subsequent chapters. These examples are provided early in this manual so you can see what an HSAIL program looks like.

3.1 Vector Add Translated to HSAIL

The "hello world" of data parallel processing is a vector add.

Suppose the high-level compiler has identified a section of code containing a vector add operation, as shown below:

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c,
                const unsigned int n)
{
  // Get our global thread ID
  int id = get_global_id(0);

  // Make sure we do not go out of bounds
  if (id < n)
    c[id] = a[id] + b[id];
}
```

The code below shows one possible translation to HSAIL:

```
version 1:0:$full:$small;

function &get_global_id(arg_u32 %ret_val) (arg_u32 %arg_val0);

function &abort() ();

kernel &__OpenCL_vec_add_kernel(
      kernarg_u32 %arg_val0,
      kernarg_u32 %arg_val1,
      kernarg_u32 %arg_val2,
      kernarg_u32 %arg_val3)
{
@__OpenCL_vec_add_kernel_entry:
// BB#0:                                // %entry
      ld_kernarg_u32    $s0, [%arg_val3];
      workitemabsid_u32 $s1, 0;
      cmp_lt_b1_u32     $c0, $s1, $s0;
      ld_kernarg_u32    $s0, [%arg_val2];
      ld_kernarg_u32    $s2, [%arg_val1];
      ld_kernarg_u32    $s3, [%arg_val0];
      cbr    $c0, @BB0_2;
      brn    @BB0_1;
@BB0_1:                                // %if.end
      ret;
@BB0_2:                                // %if.then
      shl_u32     $s1, $s1, 2;
      add_u32     $s2, $s2, $s1;
      ld_global_f32    $s2, [$s2];
      add_u32     $s3, $s3, $s1;
      ld_global_f32    $s3, [$s3];
      add_f32     $s2, $s3, $s2;
      add_u32     $s0, $s0, $s1;
      st_global_f32    $s2, [$s0];
      brn    @BB0_1;
};
```

3.2 Transpose Translated to HSAIL

The code below shows one way to write a transpose.

```
version 1:0:$full:$small;

function &get_global_id(arg_u32 %ret_val) (arg_u32 %arg_val0);

function &get_local_id(arg_u32 %ret_val) (arg_u32 %arg_val0);

function &barrier() (arg_u32 %arg_val0, arg_u32 %arg_val1);

function &get_group_id(arg_u32 %ret_val) (arg_u32 %arg_val0);

function &abort() ();

kernel &__OpenCL_matrixTranspose_kernel(
      kernarg_u32 %arg_val0,
      kernarg_u32 %arg_val1,
      kernarg_u32 %arg_val2,
      kernarg_u32 %arg_val3,
      kernarg_u32 %arg_val4,
      kernarg_u32 %arg_val5)
{
@__OpenCL_matrixTranspose_kernel_entry:
// BB#0:                                 // %entry
      workitemabsid_u32   $s0, 0;
      workitemabsid_u32   $s1, 1;
      ld_kernarg_u32      $s2, [%arg_val5];
      workitemid_u32      $s3, 0;
      workitemid_u32      $s4, 1;
      mad_u32             $s5, $s4, $s2, $s3;
      shl_u32             $s5, $s5, 2;
      ld_kernarg_u32      $s6, [%arg_val2];
      add_u32             $s5, $s6, $s5;
      ld_kernarg_u32      $s6, [%arg_val3];
      mad_u32             $s0, $s1, $s6, $s0;
      shl_u32             $s0, $s0, 2;
      ld_kernarg_u32      $s1, [%arg_val1];
      add_u32             $s0, $s1, $s0;
      ld_global_f32       $s0, [$s0];
      st_group_f32        $s0, [$s5];
      barrier;
      workgroupid_u32     $s0, 0;
      mad_u32             $s0, $s0, $s2, $s3;
      workgroupid_u32     $s1, 1;
      mad_u32             $s1, $s1, $s2, $s4;
      ld_kernarg_u32      $s2, [%arg_val4];
      mad_u32             $s0, $s0, $s2, $s1;
      shl_u32             $s0, $s0, 2;
      ld_kernarg_u32      $s1, [%arg_val0];
      add_u32             $s0, $s1, $s0;
      ld_group_f32        $s1, [$s5];
      st_global_f32       $s1, [$s0];
      ret;
};
```

Chapter 4

HSAIL Syntax and Semantics

This chapter describes the HSAIL syntax and semantics.

See also Appendix A HSAIL Grammar in Extended Backus-Naur Form (EBNF) (p. 327).

4.1 Two Formats

HSAIL programs can be sent to the finalizer in either of two formats:

- Text format
- Binary format (BRIG)

This chapter describes the text format.

The chapters describing HSAIL operations show syntax for both formats.

For more information about BRIG, see Chapter 19 BRIG: HSAIL Binary Format (p. 257).

4.2 Source Format

Source sequences are ASCII characters.

Lines are separated by the newline character (`'\n'`) or by semicolons.

Tokens are separated by white space; this consists of comments (described later), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both.

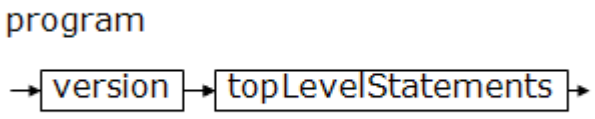
The source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

HSAIL is case-sensitive.

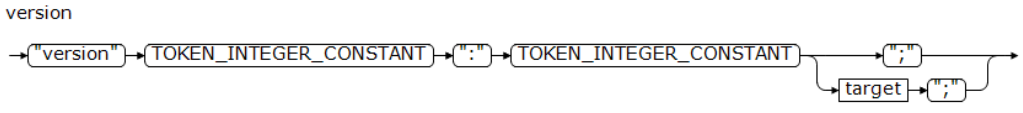
Every HSAIL input consists of a sequence of one or more programs.

Figure 4–1 Program Syntax Diagram



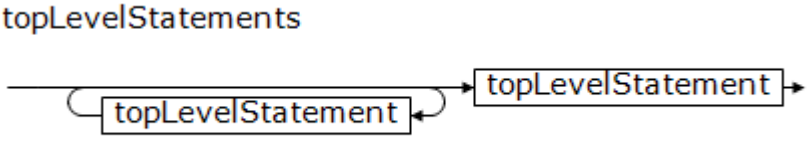
A program begins with a `version` statement specifying the HSAIL language version and the required target architecture.

Figure 4–2 version Statement



The `version` statement is followed by one or more top-level statements.

Figure 4–3 Top-Level Statements



For more information, see:

- Chapter 15 version Statement (p. 243)
- 4.5 Top-Level Statements (p. 31)

4.3 Code Blocks

A code block consists of a left curly brace (`{`) followed by one or more body statements followed by a right curly brace (`}`) and a semicolon (`;`).

Figure 4–4 Code Block

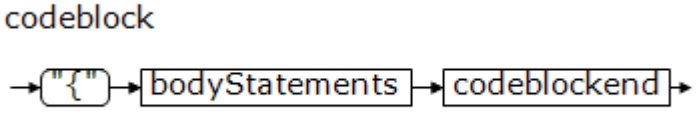
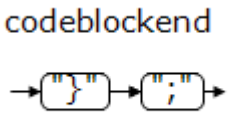


Figure 4–5 Code Block End



For example:

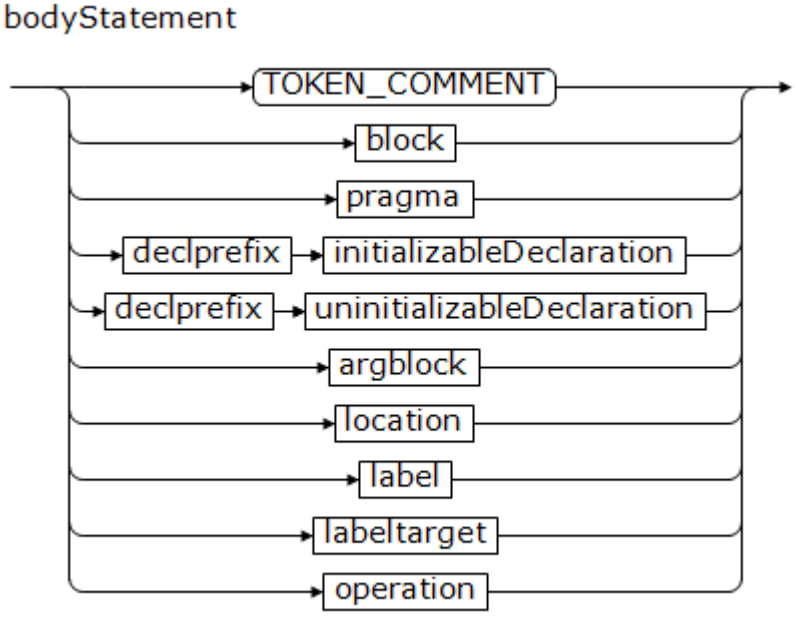
```
{
private_u32 %z;
ret;
};
```

See 4.4 Body Statements (p. 31).

4.4 Body Statements

Body statements contain the bulk of the code in an HSAIL program.

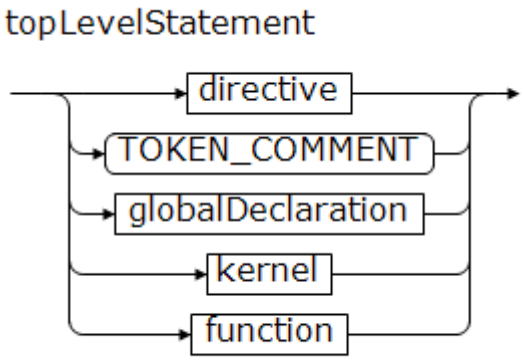
Figure 4–6 Body Statement Syntax Diagram



4.5 Top-Level Statements

A top-level statement can be a directive, comment, global declaration, kernel, or function.

Figure 4–7 Top-Level Statement Syntax Diagram



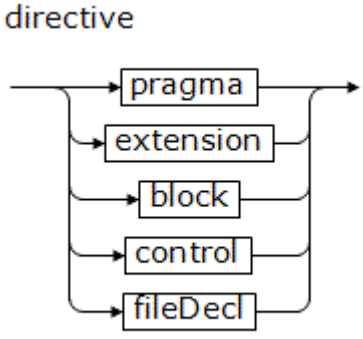
For more information, see:

- 4.5.1 Directive (p. 32)
- 4.5.2 Comment (p. 32)
- 4.5.3 Global Declaration (p. 33)
- 4.5.4 Kernel (p. 33)
- 4.5.5 Function (p. 33)

4.5.1 Directive

A directive is used to control information in HSAIL.

Figure 4–8 Directive Syntax Diagram



For more information, see Chapter 14 Directives (p. 231).

4.5.2 Comment

Comments that can span multiple lines use non-nested `/*` and `*/`. The comment starts at the `/*` and extends to the next `*/`, which might be on a different line.

Comments use `//` to begin a comment that extends to the end of the current line.

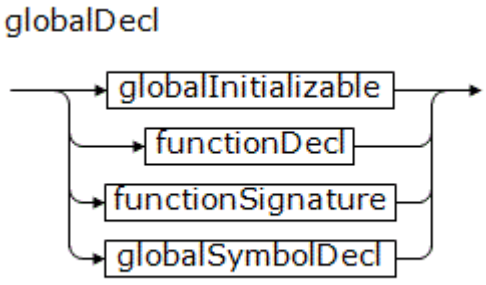
Comments are treated as white-space.

In Extended Backus-Naur Form, `TOKEN_COMMENT` is used for both types of comment.

4.5.3 Global Declaration

Outside of any function, programs can declare functions, function signatures, and two kinds of symbols: those that can be initialized and those that cannot.

Figure 4–9 Global Declaration Syntax Diagram



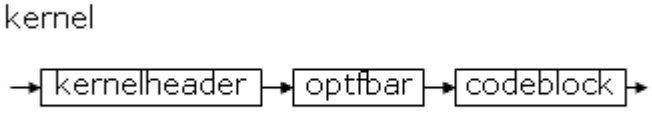
For more information, see

10.3 Function Declarations, Function Definitions, and Function Signatures (p. 205).

4.5.4 Kernel

A kernel is a kernel name followed by a kernel argument list followed by a code block.

Figure 4–10 Kernel Syntax Diagram



A single compilation unit can contain multiple kernels.

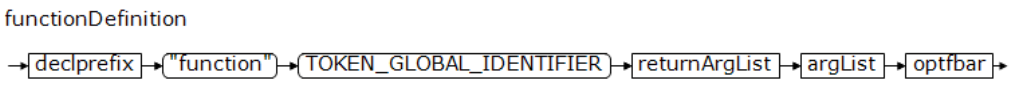
For information about code blocks, see 4.3 Code Blocks (p. 30).

4.5.5 Function

A function is a function definition followed by a code block.

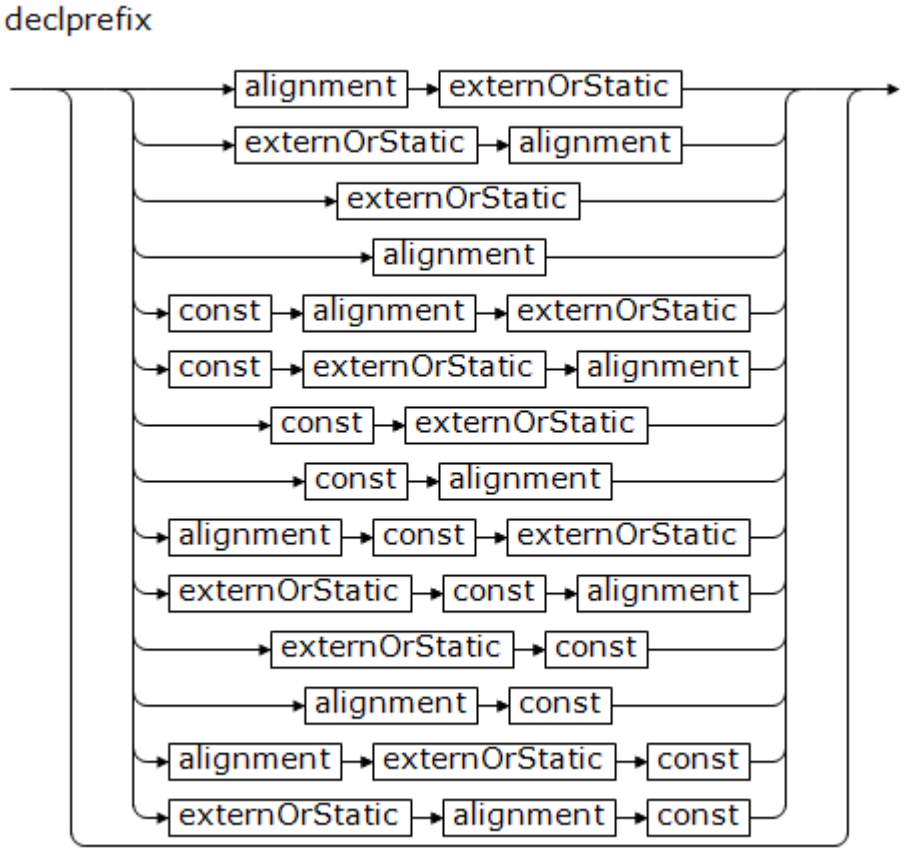
A single compilation unit can contain multiple functions.

Figure 4–11 Function Definition Syntax Diagram



A `declprefix` contains information about a declaration.

Figure 4–12 declprefix Syntax Diagram



See also 4.3 Code Blocks (p. 30).

See also Chapter 10 Functions (p. 203).

4.6 Operations

An operation is an executable HSAIL instruction.

The example below shows four operations:

```
global_f32 &array[256];
@start:  workitemid    $s1, 0;
         shl_u32       $s1, $s1, 2;         // multiply by 2
         ld_global_b32 $s2, [&array][$s1]; // reads array[4* workid]
         add_f32       $s2, $s2, 0.5;       // add 1/2
```

Operations consist of an opcode usually followed by an underscore followed by a type and a length followed by a comma-separated list of zero or more operands and ending with a semicolon.

Operands can be registers, constants, address expressions, or label names. The destination operand is first, followed by source operands.

HSAIL allows finalizers to implement extensions specific to the finalizer. A *finalizer extension* is an operation that is specified in the `extension` directive and accessed like all HSAIL operations. For more information, see 14.1.1 How to Set Up Finalizer Extensions (p. 231).

4.7 Strings

A string is a sequence of characters enclosed in double quotes (such as "abc").

Any character except for double quote (`"`), backslash (`\`) or newline can appear in the sequence.

A backslash in the character string is treated specially. It starts an escape sequence. Escape sequences are used for character patterns that are difficult to type.

There are three kinds of escape sequences:

- A backslash followed by up to three octal numbers (leading 0 not needed). For example, `'\012'` is a newline.
- A backslash followed by an x (or X) and a hexadecimal number.
- A backslash followed by one of the following characters:
  - `\` - backslash character (octal 134)
  - `'` - single quote character (octal 047)
  - `"` - double quote character (octal 042)
  - `?` - question mark character (octal 077)
  - `a` - alarm or bell character (octal 007)
  - `b` - backspace character (octal 010)
  - `f` - formfeed character (octal 006)
  - `n` - newline character (octal 012)
  - `r` - carriage-return character (octal 015)
  - `t` - tab character (octal 011)

  This is a subset of the full C character-string constants, because Unicode forms u,U,L are not supported.

In Extended Backus-Naur Form, a string is called a `TOKEN_STRING`.

4.8 Identifiers

An identifier is a sequence of characters used to identify an HSAIL object.

Figure 4–13 Identifier Syntax Diagram

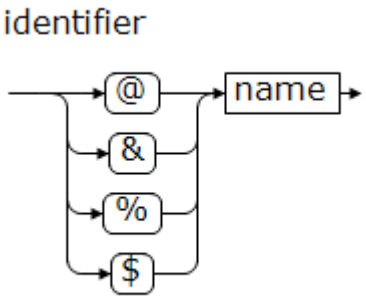
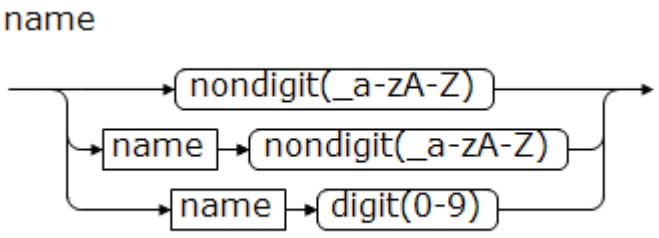


Figure 4–14 Name Syntax Diagram



4.8.1 Syntax

Identifiers that are register names must start with a dollar sign (`$`).

Identifiers that are labels must start with an at sign (`@`).

Identifiers that are not labels cannot contain an at sign (`@`).

Non-label identifiers with function scope start with a percent sign (`%`).

Non-label identifiers with compilation unit scope start with an ampersand (`&`).

Identifiers must not start with the characters `__hsa`.

The Extended Backus-Naur Form syntax is:

- A global identifier is referred to as a `TOKEN_GLOBAL_IDENTIFIER`.
- A local identifier is referred to as a `TOKEN_LOCAL_IDENTIFIER`.
- A label is referred to as a `TOKEN_LABEL`.
- A register is referred to as a `TOKEN_CREGISTER`, `TOKEN_SREGISTER`, or `TOKEN_DREGISTER`.

The second character of an identifier must be a letter (either lowercase a-z or uppercase A-Z) or the special character _.

The remaining characters of an identifier can be either letters, _, or digits.

All characters in the name of an identifier are significant.

Every HSAIL implementation must support identifiers with names whose size ranges from 1 to 1024 characters. Implementations are allowed to support longer names.

The same identifier can denote different things at different points in the program.

See also 4.22 Declaring and Defining Identifiers (p. 56).

4.8.2 Scope

An identifier is visible (that is, can be used) only within a section of program text called a scope. Different objects named by the same identifier must have different scopes.

HSAIL uses a single global name space. Thus, it is not valid to have functions, kernels, or variables declared outside a kernel or function with the same name.

Variables declared inside a kernel or function must be unique within the kernel or function, but are not required to be unique with respect to other kernels or functions that can define distinct objects with the same name.

Arguments defined inside an argument scope must be unique within the argument scope, but can have the same name as the arguments and variables in other argument scopes, or in the enclosing kernel or function's code block (in which case the argument name hides the code block name).

There are four kinds of scopes:

- Compilation unit
- Function
- Signature
- Argument

Labels and formal parameters have function scope.

An argument defined in a function or kernel has argument scope. Arguments are scoped from the point of definition to the end of the argument's scope. See 4.9 Argument Scope (p. 38).

Every other identifier has scope determined by the placement of its declaration:

- If the declaration appears outside of any function or kernel, the identifier has compilation unit scope, which terminates at the end of the source program.
- If the declaration appears inside a function or kernel, the identifier has function scope, which terminates at the end of the function or kernel code block.

If an identifier appears as a kernel argument, its scope terminates at the end of the kernel's code block.

If an identifier appears as a formal parameter in a function definition, its scope terminates at the end of the function's code block.

If an identifier appears as an argument of a function in a function signature, then it has signature scope, which terminates at the end of the signature. (Signature scope is needed for function pointers.)

4.9 Argument Scope

Arguments must start with a percent (%) sign.

An argument scope consists of a code block. See 4.3 Code Blocks (p. 30).

Within an argument scope, there are optional definitions of arguments and exactly one call operation.

Arguments are scoped from the point of definition to the end of the enclosing argument scope.

It is not valid to branch into or out of an argument scope.

An argument declared inside an argument scope hides a declaration with the same name outside the argument scope in the enclosing kernel or function scope.

Identifiers in the arg segment that are defined in an argument scope are visible from the point of definition to the end of the argument scope code block. For more information, see 10.4 Arg Segment (p. 207).

Argument scope does not affect variables that are not arguments. If a non-argument variable is defined inside an argument scope, its lifetime extends to the end of the kernel or function's code block.

Argument scopes cannot be nested.

Argument scopes can include multiple basic blocks.

See also 10.4 Arg Segment (p. 207).

4.10 Storage Duration

Global and readonly declarations can be used to allocate blocks of memory. The memory is allocated when the program begins and lasts until the program ends. This corresponds to the C++ notion of static storage duration. (See the C++ specification ISO/IEC 14882:2011.)

Kernarg declarations that appear in a kernel's formal arguments are allocated when a kernel starts and released when the kernel finishes.

Group declarations that appear inside a kernel, or at file scope, and are used by the kernel or any of the functions it can call are allocated when a work-group starts executing the kernel, and last until the work-group exits the kernel. Group declarations that appear inside any function that can be called by the kernel are allocated the same way. This is because group memory is shared between all work-items in a work-group, and the work-items within the work-group might execute the same function at different times. A consequence of this is that, if a function is called recursively by a work-item, the work-item's multiple activations of the function will be accessing the same group memory.

Private and spill declarations that appear inside a kernel, or private declarations that appear at file scope (spill cannot appear at file scope), and are used by the kernel or

any of the functions it can call are allocated when a work-item starts executing the kernel, and last until the work-item exits the kernel.

Private and spill declarations that appear inside a function are allocated each time the function is entered by a work-item, and last until the work-item exits the function.

Arg declarations inside an argument scope are allocated each time the argument scope is entered by a work-item, and last until the work-item exits the argument scope.

Recursive calls to a function will allocate multiple copies of the variables. This allows full support for recursive functions and corresponds to the C++ notion of automatic storage duration. (See the C++ specification ISO/IEC 14882:2011.) If a finalizer determines there is no recursion, it can choose to allocate these statically and avoid requiring a stack.

4.11 Rounding Modes

IEEE/ANSI Standard 754-2008 rounding modes are used for some floating-point operations:

- `up` specifies that the operation should be rounded to positive infinity.
- `down` specifies that the operation should be rounded to negative infinity.
- `zero` specifies that the operation should be rounded to zero.
- `near` specifies that the operation should be rounded to the nearest representable number and that ties should be broken by selecting the value with an even least significant bit.

See 5.11 Floating-Point Arithmetic Operations (p. 96).

The conversion (`cvt`) operation also uses rounding modes: either integer or floating-point rounding mode.

See 5.18 Conversion (cvt) Operation (p. 116).

4.12 Registers

There are four types of registers:

- Control registers (`c` registers)

  These hold a single bit value.

  Compare operations write into control registers. Conditional branches test control register values.

  Control registers are similar to CPU condition codes.

  Every HSAIL implementation supports eight control registers.

  These registers are named `$c0` to `$c7`.

- 32-bit registers (`s` registers)

  These can hold signed integers, unsigned integers, or floating-point values.

  Every HSAIL implementation supports 128 32-bit registers.

  These registers are named `$s0` to `$s127`.

- 64-bit registers (`d` registers)

  These can hold signed long integers, unsigned long integers, or double float values.

  Every HSAIL implementation supports 64 64-bit registers.

  These registers are named`$d0` to `$d63`.

- 128-bit registers (`q` registers)

  These hold packed data.

  Every HSAIL implementation supports 32 128-bit registers.

  These registers are named`$q0` to `$q31`.

Registers follow these rules:

- Registers are not declared in HSAIL.
- All registers have function scope, so there is no way to pass an argument into a function through a register.
- All registers are preserved at call sites.
- Every work-item has its own set of registers.
- No registers are shared between work-items.
- It is not possible to take the address of a register.

The `s`, `d`, and `q` registers in HSAIL share a single pool of resources. The finalizer will report an error if the value `(s + 2*d + 4*q)` exceeds 128, where `s`, `d`, and `q` are the allocated registers in an HSAIL program. Some architectures have an inverse relationship between register usage and occupancy, and high-level compilers may choose to target fewer than 128 registers to optimize for performance.

Registers are a limited resource in HSAIL, so high-level compilers are expected to manage registers carefully.

4.13 Constants

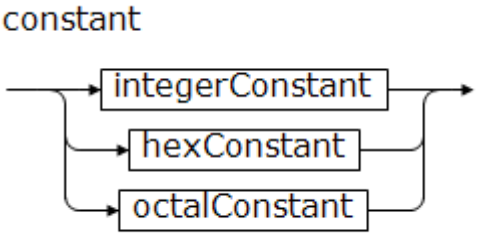
In text format, HSAIL supports three kinds of constants: integer, floating-point, and packed.

In BRIG, the binary format, all constants are stored as bit strings.

HSAIL operations can specify the type and bit size for each input. Types include integer, floating-point, packed, and bit string.

Constants can be used in data initialization directives and as operands to operations.

Figure 4–15 Constant Syntax Diagram



4.13.1 Integer Constants

Integer constants are 64 bits in size.

When used in a directive or in an operation, each integer constant is converted to the appropriate size based on the operation type at its use.

Both data initialization directives and operations accept immediate values. In BRIG, the size of the immediate value must be the number of bits needed by the operation or directive that uses the immediate.

It is possible in text format to write immediate values that are bigger than needed.

For example, in both of the following operations, the 24 and 25 are unsigned 64-bit values, but the operation expects 32-bit signed types:

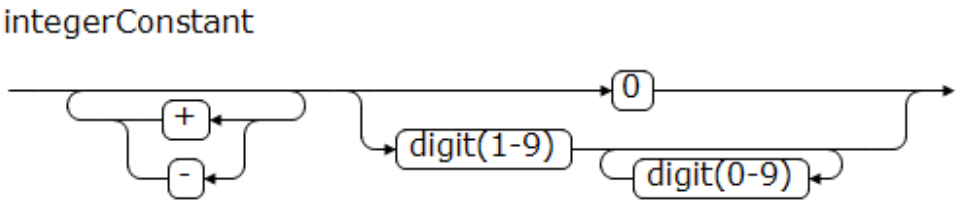
```
global_s32 &someident[] = {24, 25};

add_s32 $s1, 24, 25;
```

It is not valid to use an integer constant in a directive or operation where a floating-point value is expected.

An integer constant can be written in decimal, octal, or hexadecimal form. Both + and - signs are allowed.

Note that the syntax follows C++ syntax.

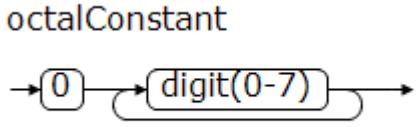
Figure 4–16 Integer Constant Syntax Diagram



Decimal constants are sign-extended to 64 bits.

A decimal constant (optionally signed) starts with a non-zero digit. An octal constant starts with `0`, as shown below:

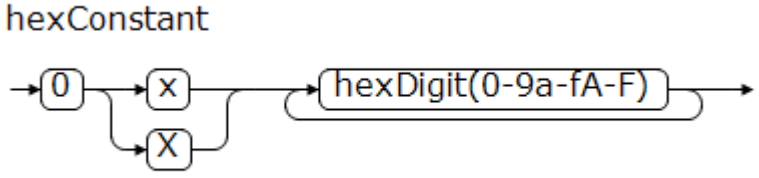
Figure 4–17 Octal Constant Syntax Diagram



Octal and hexadecimal constants are zero-extended to 64 bits.

A hexadecimal constant starts with `0x` or `0X`, as shown below:

Figure 4–18 Hex Constant Syntax Diagram



Integer constants can be used as controls. For control-type data initializers and operation operands, integer constants are interpreted as in C and C++. (That is, zero values are False and non-zero values are True.)

In Extended Backus-Naur Form, `TOKEN_INTEGER_CONSTANT` is used for all three types of constant.

4.13.2 Floating-Point Constants

Floating-point constants are represented as either 32-bit single-precision or 64-bit double-precision values.

It is an error if a floating-point constant is used in an operation or directive that expects a different size.

It is an error to use a floating-point value where an integer or bit string value is expected.

In Extended Backus-Naur Form, `TOKEN_SINGLE_CONSTANT` is used for 32-bit floating-point constants and `TOKEN_DOUBLE_CONSTANT` for 64-bit floating-point constants. Floating-point constants can be written with a decimal point or a signed exponent, double-precision format followed by an optional float size suffix. Either the decimal point, the exponent, or the suffix must be present, because a sequence of digits alone is interpreted as an integer constant.

Floating-point constants can also be specified as hexadecimal constants in two ways.

One way is to specify IEEE/ANSI Standard 754-2008 binary interchange format. To specify IEEE/ANSI Standard 754-2008 double-precision floating-point values, the constant begins with `0d` or `0D` followed by 16 hexadecimal digits. To specify IEEE/ANSI Standard 754-2008 single-precision floating-point values, the constant begins with `0f` or `0F` followed by eight hexadecimal digits.

The second way to specify floating-point constants in hex notation uses the C99 format. A C99 float constant in hex consists of a hexadecimal prefix, a significand part, a binary exponent part, and an optional suffix. The significand part represents a rational number and consists of a sequence of hexadecimal digits (the whole number) followed by an optional fraction part (a period followed by a sequence of hexadecimal digits). The binary exponent part is an optionally signed decimal integer that indicates the power of 2. The significand is raised to that power of 2. The optional suffix indicates the type: f or F indicates 32 bits, l or L indicates 64 bits. If the type is omitted, 64 bits is used. A double like 12.345 can be written as 0d4028b0a3d70a3d71, or 0x1.8b0a3d70a3d71p+3.

Figure 4–19 Floating-Point Single Constant Syntax Diagram

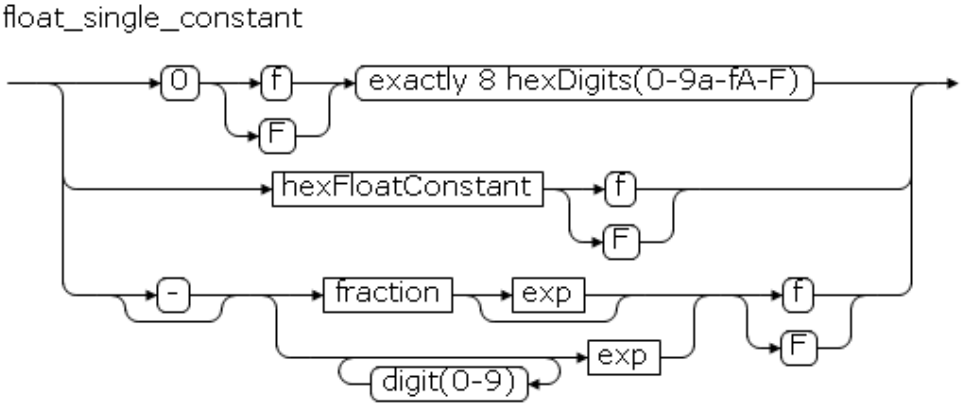


Figure 4–20 Floating-Point Double Constant Syntax Diagram

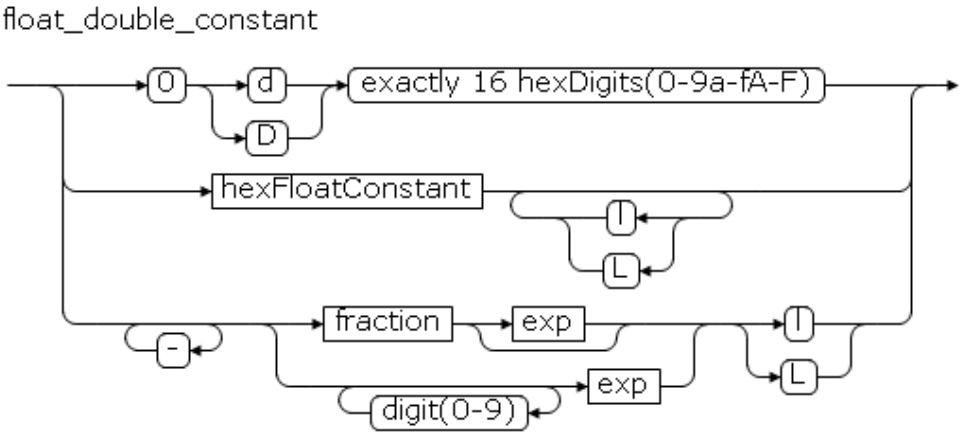


Figure 4–21 hexFloatConstant Syntax Diagram

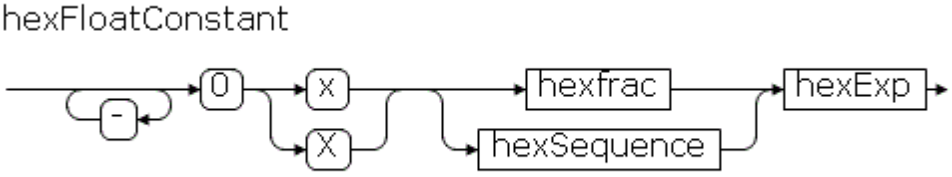


Figure 4–22 hexFrac Syntax Diagram

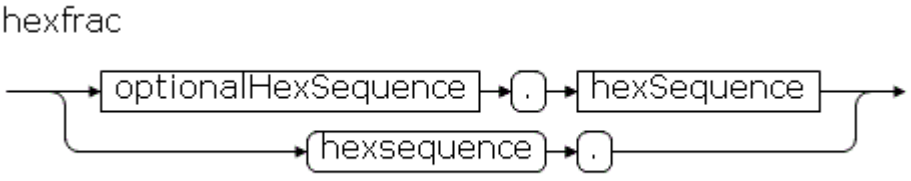


Figure 4–23 hexExp Syntax Diagram

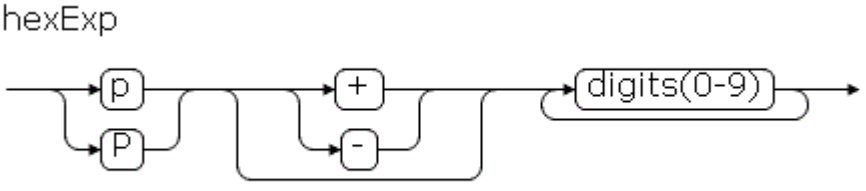
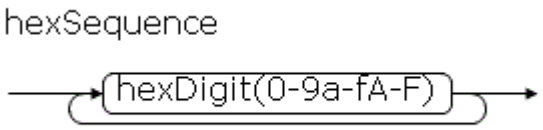


Figure 4–24 hexSequence Syntax Diagram



4.13.3 Packed Constants

For information, see 4.15.2 Packed Constants (p. 49).

4.13.4 How Text Format Constants Are Converted to Bit String Constants

HSAIL assemblers convert from text format to binary format, changing each constant to a bit string. The conversion rules are determined by the type of the constant provided and by the type the operation wants.

See the following table, which describes how text format constants are converted to bit string constants used in BRIG. What happens with the conversion depends on the data type expected by the operation.

Table 4–1 Text Constants and Results of the Conversion

Kind of text format constant provided	Data type of expected value			
	Signed/unsigned	Bit	Floating-point	Packed
Integer constant	Truncate	Truncate	Error	Truncate
Floating-point constant	Error	Length-only rule	Type and length rule	Error
Packed constant	Error	Length-only rule	Error	Type and length rule

Truncation for an integer value in the text is as follows: the value is input as 64 bits, then the length needed is compared to the size the operation needs.

If the operation needs 64 bits or fewer, the 64-bit value is truncated if necessary.

If the operation needs more than 64 bits, it is an error.

For example:

```
add_s32x2 ... 0xfffffffff; // 9 f's
```

The 9 f's is a 64-bit integer constant with 36 non-zero bits. The operation uses a packed type s32x2 (two 32-bit signed integers), so the number of bits match identically in BRIG. This would be stored as b64 0x0000000fffffffff.

It is not possible to provide an integer constant to a 128-bit data type. However, a packed constant can be used for a 128-bit data type.

For example:

```
mov_b128 q1, 2; // illegal because integer constant is evaluated as 64 bits
                // and operation requires 128 bits.
```

These operations are legal:

```
mov_b128 q1, _u32x4(1, 2, 3, 4); // legal to use packed constant of same size
mov_b128 q1, _u64x2(1, 2);       // legal to use packed constant of same size
```

If the operation was `add_s8x4`, four signed 8-bit elements, for a total of 32 bits, the constant would be truncated and stored as b32 0xffffffff.

If the type was `s32x4` (four elements each a signed 32-bit number), there would not be enough data so there would be an error.

The type and length match rule is the following: the number of bits and the type must be the same; otherwise this is an error.

The length-only rule is the following: the bits in the constant are used provided the number of bits is the same. For example, `mov_b32 ... 3.7f` expects a 32-bit value. The operation has a 32-bit floating-point constant 3.7f. Although the types do not match, the lengths do match, so the hexadecimal value 3.7f is used.

4.14 Data Types

4.14.1 Base Data Types

HSAIL has four base data types, each of which supports a number of bit lengths. See Table 4–2 (p. 46).

Table 4–2 Base Data Types

Type	Description	Possible lengths in bits
b	Bit type	1, 8, 16, 32, 64, 128
s	Signed integer type	8, 16, 32, 64
u	Unsigned integer type	8, 16, 32, 64
f	Floating-point type	16, 32, 64

A *compound type* is made up of a base data type and a length.

Most operations specify a single compound type, used for both destinations and sources. However, the conversion operations (`cvt`, `ftos`, `stof`, and `segmentp`) specify an additional compound type for the sources. The order is destination compound type followed by the source compound type.

The finalizer might perform some checking on operand sizes.

4.14.2 Packed Data

Packed data allows multiple small values to be treated as a single object.

Packed data lengths are specified as an element size in bits followed by an x followed by a count. For example, 8x4 indicates that there are four elements, each of size 8 bits.

See Table 4–3 (p. 47).

Table 4–3 Packed Data Types and Possible Lengths

Type	Description	Lengths for 32-bit types	Lengths for 64-bit types	Lengths for 128-bit types
s	Signed integer	8x4, 16x2	8x8, 16x4, 32x2	8x16, 16x8, 32x4, 64x2
u	Unsigned integer	8x4, 16x2	8x8, 16x4, 32x2	8x16, 16x8, 32x4, 64x2
f	Floating-point	16x2	16x4, 32x2	16x8, 32x4, 64x2

32-bit sizes are:

- 8x4 — four bytes; can be used with s and u types
- 16x2 — two shorts or half-floats; can be used with s, u, and f types

64-bit sizes are:

- 8x8 — eight bytes; can be used with s and u types
- 16x4 — four shorts or half-floats; can be used with s, u, and f types
- 32x2 — two integers or floats; can be used with s, u, and f types

128-bit sizes are:

- 8x16 — 16 bytes; can be used with s and u types
- 16x8 — eight shorts or half-floats; can be used with s or u, and f types
- 32x4 — four integers or floats; can be used with s, u, and f types
- 64x2 — two 64-bit integers or two doubles; can be used with s, u, and f types

4.14.3 Opaque Data Types

HSAIL also has the following opaque types (see 7.1.2 How Images Are Described (p. 152)):

Table 4–4 Opaque Data Types

Type	Description	Length in bits
roimg	Read-only image object	64
rwimg	Read-write image object	64
samp	Sampler object	64

An opaque type has a fixed size, but its representation is implementation-defined.

4.15 Packing Controls for Packed Data

Certain HSAIL operations operate on packed data. Packed data allows multiple small values to be treated as a single object. For example, the u8x4 data type uses 32 bits to hold four unsigned 8-bit bytes.

Operations on packed data have both a data type and a packing control. The packing control indicates how the operation selects elements.

See 4.14.2 Packed Data (p. 46).

The packing controls differ depending on whether an operation has one source input or two.

See the tables below.

Table 4–5 Packing Controls for Operations With One Source Input

Control	Description
p	The single source is treated as packed. The operation is applied to each element separately.
p_sat	Same as p, except that each result is saturated. (Cannot be used with floating-point values.)
s	The lower element of the source is used. The result is written into the lower element of the destination. The other bits of the destination are not modified.
s_sat	Same as s, except that the result is saturated. (Cannot be used with floating-point values.)

Table 4–6 Packing Controls for Operations With Two Source Inputs

Control	Description
pp	Both sources are treated as packed. The operation is applied pairwise to corresponding elements independently.
pp_sat	Same as pp, except that each result is saturated. (Cannot be used with floating-point values.)
ps	The first source operand is treated as packed and the lower element of the second source operand is broadcast and used for all its element positions. The operation is applied independently pairwise between the elements of the first packed source operand and the lower element of the second packed operand. The result is stored in the corresponding element of the packed destination operand.
ps_sat	Same as ps, except that each result is saturated. (Cannot be used with floating-point values.)
sp	The lower element of the first source operand is broadcast and used for all its element positions, and the second source operand is treated as packed. The operation is applied independently pairwise between the lower element of the first packed operand and the elements of the second packed operand. The result is stored in the corresponding element of the packed destination operand.
sp_sat	Same as sp, except that each result is saturated. (Cannot be used with floating-point values.)
ss	The lower element of both sources is used. The result is written into the lower element of the destination. The other bits of the destination are not modified.
ss_sat	Same as ss, except that the result is saturated. (Cannot be used with floating-point values.)

4.15.1 Ranges

For all packing controls, the following applies:

- For u8x4 and u8x8, the range of an element is 0 to 255.
- For s8x4 and s8x8, the range of an element is –128 to + 127.
- For u16x2 and u16x4, the range of an element is 0 to 65536.
- For s16x2 and s16x4, the range of an element is –32768 to 32767.

For packing controls with the `_sat` suffix, the following applies:

- If the result value is larger than the range of an element, it is set to the maximum representable value.
- If the result value is less than the range of an element, it is set to the minimum representable value.

4.15.2 Packed Constants

HSAIL provides a simple notation for writing packed constant values: an operand that consists of an underscore (_) followed by a packed type and length followed by a parenthesized list of signed values is converted to a single packed constant.

For `s` and `u` types, the values must be integer. If a value is too large to fit in the format, the lower-order bits are used.

For `f` types, the values must be floating-point. The floating-point constant is read as `f64` and converted to the expected size using convert to nearest.

In the following examples, each pair of lines generates the same constant value:

```
add_pp_s16x2 $s1, $s2, _s16x2(-23,56);
add_pp_s16x2 $s1, $s2, 0xffe90038;

add_pp_u16x2 $s1, $s2, _u16x2(23,56);
add_pp_u16x2 $s1, $s2, 0x170038;

add_pp_s16x4 $d1, $d2, _s16x4(23,56,34,10);
add_pp_s16x4 $d1, $d2, 0x1700380022000a;

add_pp_u16x4 $d1, $d2, _u16x4(1,0,1,0);
add_pp_u16x4 $d1, $d2, 0x1000000010000;

add_pp_s8x4 $s1, $s2, _s8x4(23,56,34,10);
add_pp_s8x4 $s1, $s2, 0x1738220a;

add_pp_u8x4 $s1, $s2, _u8x4(1,0,1,0);
add_pp_u8x4 $s1, $s2, 0x1000100;

add_pp_s8x8 $d1, $d2, _s8x8(23,56,34,10,0,0,0,0);
add_pp_s8x8 $d1, $d2, 1673124687913156608;

add_pp_s8x8 $d1, $d2, _s8x8(23,56,34,10,0,0,0,0);
add_pp_s8x8 $d1, $d2, 0x1738220a00000000;

add_pp_f32x2 $d1, $d2, _f32x2(2.0, 1.0);
add_pp_f32x2 $d1, $d2, 0x3f80000040000000;
```

4.15.3 Examples

The following example does four separate 8-bit signed adds:

```
add_pp_s8x4 $s1, $s2, $s3;
```

`s1` = the logical OR of:

```
s2[0-7] + s3[0-7]
s2[8-15] + s3[8-15]
s2[16-23] + s3[16-23]
s2[24-31] + s3[24-31]
```

The following example does four separate signed adds, adding the lower byte of `$s3` (bits 0-7) to each of the four bytes in `$s2`:

```
add_ps_s8x4 $s1, $s2, $s3;
```

4.16 Subword Sizes

The `b8`, `b16`, `s8`, `s16`, `u8`, and `u16` types are allowed only in loads/stores and conversions.

4.17 Operands

HSAIL is a classic load-store machine, with most ALU operands being either in registers or immediate values. In addition, there are several other kinds of operands.

The operation specifies the valid kind of each operand using these rules:

- A source operand and a destination operand can be a register. The rules for register operands are described below.
- A source operand can be an immediate. Immediate values can be either a constant (see 4.13 Constants (p. 41)) or `WAVESIZE` (see 2.6.2 Wavefront Size (p. 12)).
- Memory, image, segment checking, segment conversion, and `lda` operations take an address expression as a source operand (see 4.18 Address Expressions (p. 52)).
- Memory, image, and some copy (move) operations allow a vector register as source and destination operands. These comprise a list of registers (see 4.19 Vector Operands (p. 53)).
- Branch operations can take a label and list of labels as a source operand (see Chapter 8 Branch Operations (p. 177)).
- Call operations can take a function identifier, list of function identifiers, and signature identifier as a source operand (see Chapter 11 Operations Related to Functions (p. 211)).

The source operands are usually denoted in the operation descriptions by the names *`src0`*, *`src1`*, *`src2`*, and so forth.

The destination operand of an operation must be a register. It is denoted in the operation descriptions by the name *`dest`*. A destination operand can also be a vector register, in which case it is denoted as a list of registers with names *`dest0`*, *`dest1`*, and so forth.

4.17.1 Operand Compound Type

Register, immediate, and address expression operands have an associated compound type (see 4.14 Data Types (p. 46)). This defines the size and representation of the value provided by the source operand or stored in the destination operand.

For most operations, the compound type used is the operation's compound type. However, some operations have two compound types, the first for the destination operand and the second for the source operands. In addition, for some operations, certain operands have a fixed compound type defined by the operation.

For address expressions, the compound type refers to the value in memory, not the compound type of the address, which is always `u32` or `u64` according to the address size (see Table 2–3 (p. 20)).

For vector registers, the compound type applies to each register, and the rules for register operands below apply to each individual register. The individual registers do not need to be different for source operands, but do need to be different for destination operands.

The rules for converting constant values to the source operand compound type are given in 4.13.4 How Text Format Constants Are Converted to Bit String Constants (p. 45).

`WAVESIZE` is allowed only if the source operand is an integer compound type.

4.17.2 Rules for Source Operand Registers

The following rules apply to source operand registers:

- If the source operand compound type is an integer type (`s` and `u`), bit type (`b`), `f32`, `f64`, or a packed type, `roimg`, `rwimg`, or `samp`, the source operand register must match the size of its compound type (with the exception of the `cvt` and `st` operations described below).
- If the source operand of a `cvt` operation has a compound type of `b1` or has an integer type, or the source operand containing the value being stored by a `st` operation has an integer type: the register must be at least the size of its operand compound type. If it is larger, then the size of the compound type dictates the number of least significant bits of the register that are used.
- If the source operand is the compound type of `f16`, then it must be an `s` register. The `s` register representation of `f16` is implementation-defined and might not match the memory representation (see 4.21 Floating-Point Numbers (p. 55)), so care should be taken to use operations that correctly convert to and from register representation if arithmetic is to be performed: `ld`, `st`, `pack`, and `unpack`.

4.17.3 Rules for Destination Operand Registers

The following rules apply to destination operand registers:

- If the destination operand compound type is an integer type (`s` and `u`), bit type (`b`), `f32`, `f64`, or a packed type, `roimg`, `rwimg`, or `samp`, the destination operand register must match the size of its compound type (with the exception of the `cvt` and `ld` operations described below).
- If the destination operand of a `cvt` operation has a compound type of `b1` or has an integer type, or the destination operand of an `ld` operation has an integer type: the register must be at least the size of its operand compound type. If it is larger, then the operation is performed in the size of the operand compound type. The result is then zero-extended for `b` (for `cvt_b1`) and `u` types, and sign-extended for `s` types, to the size of the destination register. For example, when an `ld_u16` operation has a `d` (64-bit) destination register, then a 16-bit value is loaded from memory, zero-extended to 64 bits, and stored in the `d` register.
- If the destination operand is the compound type of `f16`, then it must be an `s` register. The `s` register representation of `f16` is implementation-defined and might not match the memory representation (see 4.21 Floating-Point Numbers (p. 55)). The `ld` and `unpack` operations will convert the source value from the memory representation of `f16` into the `s` register representation. All other operations will operate on the `s` register representation of `f16`. Despite using an `s` register, an `f16` value is not converted to an `f32` value: the `cvt` operation must be used to explicitly perform the conversion. (Note that packed `f16`, such as `f16x2`, is not the same type as `f16`, and the packed components will be operated on using the memory representation of `f16`.)

If it is necessary to transfer an integer value in a `d` register into an `s` register, or vice versa, the `cvt` operation must be used to do the appropriate truncation or zero/sign extension.

4.18 Address Expressions

Most variables have two addresses:

- Flat address
- Segment address

A flat address is a general address that can be used to address any HSAIL memory. Flat addresses are in bytes.

A segment address is an offset within the segment in bytes.

If a segment is used in a load or store, the address is treated as a segment address.

Address expressions consist of one of the following:

- A name (file, kernel, function, or argument scope) in square brackets
- An address in square brackets
- A name in square brackets followed by an address in square brackets

An address is one of the following:

- A register
- A constant
- A register + constant
- A register – constant

For information about how to declare an identifier, see 4.22 Declaring and Defining Identifiers (p. 56).

Addresses are always in bytes. For information about how addresses are formed from an address expression, see 6.1.1 How Addresses Are Formed (p. 123).

Some examples of addresses are:

```
global_f32 %g1[10];            // allocate an array in a global segment
group_f32 %x[10];              // allocate an array in a group segment
ld_global_f32 $s1, [%g1][0];   // the [0] is optional
ld_global_f32 $s2, [%g1][2];
ld_global_u32 $s3, [%g1][$s2]; // read the float bits as an unsigned integer
ld_global_u32 $s4, [%g1][$s2+4];
ld_global_u32 $s5, [100];      // read from absolute flat address
ld_group_f32 $s3, [%x][$s2];   // segment-relative
ld_group_f16 $s5, [100];       // read 16 bits at absolute offset 100
```

See 6.2 Load (ld) Operation (p. 126).

4.19 Vector Operands

Several operations support vector operands.

Both destination and source vector operands are written as a comma-separated list of registers enclosed in parentheses. A v2 operand contains two registers, a v3 operand contains three registers, and a v4 operand contains four registers.

The registers in the list must be the same size: either all c registers, all s registers, or all d registers. q registers cannot appear in a vector operand.

In BRIG, the type of the vector operand is the type of each register (see 4.17 Operands (p. 50)).

In a vector operand used as a destination, it is not valid to repeat a register.

Loads and stores with vector operands can be used to implement loading and storing of contiguous multiple bytes of memory, which can improve memory performance.

It is not valid to omit a register from the list.

Examples:

```
group_u32 %x;
readonly_s32 %tbl[256];

ld_group_u16 $s0, [%x];          // via offset
ld_group_u32 $s0, [%x];
ld_group_f32 $s2, [%x][0];       // treat result as floating-point

ld_readonly_v4_f32 ($s0,$s3,$s1,$s2), [%tbl];
ld_readonly_s32 $s1, [%tbl][12];
ld_width(all)_readonly_v4_f32 ($s0,$s3,$s9,$s1),
     [%tbl][2];                  // broadcast form
ld_v2_f32 ($s9,$s2), [$s1+3];
```

See 6.2 Load (ld) Operation (p. 126).

4.20 Labels

Labels consist of an at sign (`@`) followed by the name of the identifier. Label definitions consist of a label followed by a colon (`:`).

Labels cannot be used in any operations except `brn`, `cbr`, and `ldc`.

Labels cannot appear in an address expression.

Labels can be used in global and readonly variable initializers.

4.21 Floating-Point Numbers

Floating-point data is stored in binary format using the rules of IEEE/ANSI Standard 754-2008:

- An `f16` number is stored in memory as 1 bit of sign, 5 bits of exponent, and 10 bits of mantissa. The exponent is biased with an excess value of 15. The representation of an `f16` value stored in an `s` register is implementation-defined and need not match the memory representation. For example, it is allowed to use all 32 bits of the `s` register.

  The IEEE/ANSI Standard 754-2008 precision requirements for `f16` are a minimum requirement, and register operations may be performed at greater precision and greater range, only converting to the `f16` memory representation when stored. Therefore, `f16` results are not required to be bit-reproducible across different HSA implementations.

  The conversion between memory representation and register representation might lead to unexpected results. For example, if the `fract` operation is performed on an `s` register and the result stored into memory, the memory value might have the value 1.0 due to the rounding from the register representation to the memory representation.
- An `f32` number is stored in memory and in an `s` register as 1 bit of sign, 8 bits of exponent, and 23 bits of mantissa. The exponent is biased with an excess value of 127.
- An `f64` number is stored in memory and in a `d` register as 1 bit of sign, 11 bits of exponent, and 52 bits of mantissa. The exponent is biased with an excess value of 1023.

In all cases, if the exponent is all 1's and the mantissa is not 0, the number is a NaN.

If the exponent is all 1's and the mantissa is 0, then the value is either Infinity (sign == 0) or –Infinity (sign == 1).

There are two representations for 0: positive zero has all bits 0; negative zero has a 1 in the sign bit and all other bits 0.

The first bit of the mantissa is used to distinguish between signaling NaNs (first bit 0) and quiet Nans (first bit 1).

Signaling NaNs are never the result of arithmetic operations.

The remaining bits of the mantissa of a NaN can be used to carry a payload (information about what caused the NaN).

The sign of a NaN has no meaning, but it can be predictable in some circumstances.

HSAIL programs can use hex formats to indicate the exact bit pattern to be used for a floating-point constant.

Conversions to floating-point that are beyond the range of floating-point numbers are represented with the maximum floating-point value (IEEE/ANSI Standard 754-2008 Inf) for `f16`, `f32`, and `f64`.

In the Full profile, subnormal values are only flushed to zero when requested with the `ftz` modifier. In the Base profile, all operations that support the `ftz` modifier must specify it, and will always flush subnormal values to zero. See 17.2 Profile-Specific Requirements (p. 250).

4.22 Declaring and Defining Identifiers

For a description of identifiers, see 4.8 Identifiers (p. 36).

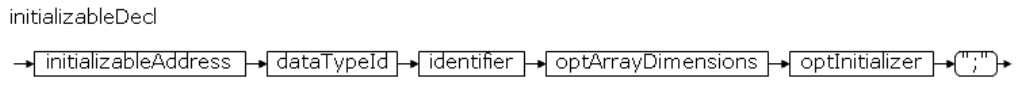
A declaration establishes the name and characteristics of an identifier.

A definition declares the identifier, allocates storage, and for some segments may optionally initialize the storage.

If the same object is both declared and defined, the object must have the same properties. If the object is an array, the size of the array must be specified in the definition but can be omitted in the declaration.

An optional initializer can appear on a definition but not on a declaration.

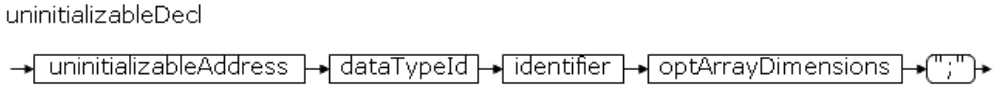
Figure 4–25 Initializable Declaration or Definition Syntax Diagram



An initializable address can be in one of the following segments:

- Global
- Readonly

Figure 4–26 Uninitializable Declaration or Definition Syntax Diagram



An uninitializable address can be in one of the following segments:

- Group
- Private
- Kernarg
- Spill
- Arg

The data type ID can be one of the data types described in 4.14 Data Types (p. 46) (except for `b1`).

A definition determines the following properties of an identifier:

- Scope — The section of program text where the identifier is visible and can be used.
- Segment — The memory segment that will be used to hold the object.

- BaseType — How the bit pattern in storage is interpreted.
- Size — The number of bytes of storage needed to hold the identifier.
- Initializer — The initial value of the storage.

A variable declaration describes both the variable's compound type and its segment. In addition to scalars, there is support for aggregate objects such as arrays.

One or more optional type qualifiers can be specified:

- `const`

  The variable cannot be written to after the kernel starts execution. Only global and readonly segment variable definitions can be marked `const`. (Readonly segment variables are implicitly `const` so they do not need to be marked `const`.)

  Memory for `const` variables remains constant during the execution of a kernel.

  The variable should not be modified by stores or atomic operations. It is an error to use a store or atomic operation with a `const` variable. It is undefined if implementations will detect stores or atomic operations to `const` variables.

  The finalizer might place a `const` variable in specialized read-only caches.

- `align` *n*

  The storage for the variable must be aligned on an address that is an integer multiple of *n*. Valid values of *n* are 1, 2, 4, 8, and 16.

  For arrays, alignment specifies the address alignment of the starting address of the entire array, not the alignment of individual elements.

  Without an `align` qualifier, the variable will be naturally aligned. That is, the address assigned to the variable will be a multiple of the variable's base type length.

  Packed data types are naturally aligned to the size of the entire packed type (not the size of the each element). For example, the `s32x4` packed type (four 32-bit integers) is naturally aligned to a 128-bit boundary.

  If an alignment is specified, it must be equal to or greater than the variable's natural alignment. Thus, `global_f64 x[10]` must be aligned on a 64-bit (8-byte) boundary. For example, `align 8 global_f64 x[10]` is valid, but smaller values of *n* are not valid.

- `extern`

  The variable is being declared and not defined. Its address will be allocated by a definition of the same variable, which can be in this or another compilation unit. See 4.23.1 External Linkage (p. 61).

- `static`

  The variable is defined in the current compilation unit and cannot be used to satisfy an `extern` declaration in another compilation unit. See 4.23.2 Static Linkage (p. 62).

The supported segments are:

- Global and readonly

  The storage for the variable can be accessed by all work-items in the grid.

  Declarations for `global` can appear either inside or outside of a kernel or function. Such variables that appear outside of a kernel or function have scope from the point of declaration until the end of the compilation unit. Those defined inside a kernel or function have scope from the point of declaration to the end of the kernel or function code block (even if declared in a nested argument scope).

- Group

  The storage for the variable can be accessed by all work-items in a work-group, but not by work-items in other work-groups. Each work-group will get an independent copy of any variable assigned to the group segment.

  Declarations for `group` can appear either inside or outside of a kernel or function. Such variables that appear outside of a kernel or function have scope from the point of declaration until the end of the compilation unit. Those defined inside a kernel or function have scope from the point of declaration to the end of the kernel or function code block (even if declared in a nested argument scope).

- Private

  The storage for the variable is accessible only to one work-item and is not accessible to other work-items.

  Declarations for `private` can appear either inside or outside of a kernel or function. Such variables that appear outside of a kernel or function have scope from the point of declaration until the end of the compilation unit. Those defined inside a kernel or function have scope from the point of declaration to the end of the kernel or function code block (even if declared in a nested argument scope).

- Kernarg

  The value of the variable can be accessed by all work-items in the grid. It is a formal argument of the kernel. The finalizer might use the type information to better allocate registers.

  Declarations for `kernarg` must be in a kernel argument list. Such variables have scope from the point of declaration until the end of the kernel code block.

- Spill

  The storage for the variable is accessible only to one work-item and is not accessible to other work-items. Such variables are used to save and restore registers.

  Declarations for `spill` must appear inside a kernel or function. Such variables have scope from the point of declaration until the end of the kernel or function code block (even if declared in a nested argument scope).

- Arg

  The storage for the variable is accessible only to one work-item and is not accessible to other work-items. Such variables are used to pass per work-item arguments to functions.

  Declarations for `arg` must appear inside an argument scope within a kernel, function, or within a function signature. Such variables that appear inside an argument scope have scope from the point of declaration until the end of the argument scope. Those defined inside a function signature of a function declaration have scope from the point of declaration to the end of the function signature. Those defined inside a function signature of a function definition have scope from the point of declaration to the end of the function code block.

See 4.10 Storage Duration (p. 38) for a description of when storage is allocated for variables.

Here is an example:

```
function &fib (arg_s32 %r) (arg_s32 %n)
{
   ld_arg_s32 $s1, [%n];
   cmp_lt_b1_s32 $c1, $s1, 3; // if n < 3 go to return
   cbr $c1, @return;
   private_s32 %p;            // allocate a private variable
                              // to hold the partial result
   {
      arg_s32 %nm2;
      arg_s32 %res;
      sub_s32 $s2, $s1, 2;     // compute fib (n-2)
      st_arg_s32 $s2, [%nm2];
      call &fib (%res)(%nm2);
      ld_arg_s32 $s2, [%res];
   }
   st_private_s32 $s2, [%p];  // save the result in p
   {
      arg_s32 %nm2;
      arg_s32 %res;
      sub_s32 $s2, $s1, 1;     // compute fib (n-1)
      st_arg_s32 $s2, [%nm2];
      call &fib (%res)(%nm2);
      ld_arg_u32 $s2, [%res];
   }
   ld_private_u32 $s3, [%p];  // add in the saved result
   add_u32 $s2, $s2, $s3;
   st_arg_s32 $s2, [%r];
@return: ret;
};
```

4.22.1 Array Declarations

Array declarations are provided to allow the high-level compiler to reserve space. To declare an array, the variable name is followed with dimensional declarations. The size of the dimension is either a constant or is left empty.

Note that the array declaration is similar to C++.

The size of the array specifies how many elements should be reserved. Each element is aligned on the base type length, so no padding is necessary.

Variable definitions in the global and readonly segments can optionally specify an initial value. The variable name is followed by an equals (=) sign and the initial value or values for the variable. A scalar takes a single value, while vectors take a list of values inside of curly braces. For the initialization of image and sampler objects, see 7.1.4 Image Objects (p. 154) and 7.1.7 Sampler Objects (p. 160).

The values are converted to the size needed to fit into the destination.

If an immediate value is not the same type or size as the element, then the rules in Table 4–1 (p. 45) apply. For example, `global_u32 &x = {3.0f}` initializes the identifier `x` to the 32-bit value 0x40400000.

It is only possible to initialize the global and readonly segments.

Label names appearing in initializers represent the address of the next operation following the label. This can be used to initialize a jump table to be used with indirect jumps. Label addresses should be of type `u` or `s` and of size 32.

Variables that hold addresses of variables and code labels should be of type `u` or `s` and of size 32 or 64.

The size of the array can be set by the length of the initialization list:

```
readonly_u32 &days[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Note that this follows the C++ rules.

Only constant values and label names can appear in an initialization list. Values and labels cannot be mixed in the same initialization list.

If there is no initializer, the value of the object is undefined. However, if some of the elements are initialized, then the rest are initialized to zero.

For example:

```
global_u32 &x1[10];                     // no initializer (values start
                                        // as undefined)
global_u32 &y1[10]= {1,2,3,4,5,6,7,8}; // elements 0,1,2,3,4,5,6,7
                                        // are initialized
```

Examples:

```
global_u32 &loc1;
global_f32 &bias[] = {-1.0, 1.0};
global_u8 &bg[4] = {0, 0, 0, 0};
const align 8 global_b8 &willholddouble [8] = {0,0,0,0,0,0,0,0};
```

The last formal argument of a function signature can be an array without a specified size. The size passed is determined by the size specified by the arg definition of the function call. See 10.5 Variadic Functions (p. 208).

4.23 Linkage: External, Static, and None

Linkage determines the rules that permit a name (function, kernel, or variable) to refer to the same object as a name declared in another scope.

HSAIL provides three types of linkage:

- External: The object denoted can also be referenced by the same name from another scope in this or another compilation unit.
- Static: The object denoted can be referenced by the same name in another scope within this compilation unit, but not from another compilation unit.
- None: The object denoted cannot be referenced by the same name in another scope.

4.23.1 External Linkage

Under some conditions, a name of a function or variable in one compilation unit might refer to an object with the same name defined in a different compilation unit. The two names are linked together. Only one compilation unit is allowed to have a definition for the name. In all other compilation units that refer to the same object, the name must be a declaration marked `extern`. Objects that can be linked together in this way are said to have *external linkage*.

A name can be both defined in a compilation unit and also declared as `extern` in the same compilation unit.

For example:

```
extern function &foo ()(); // declaration: says it is coming or
                           // is in anther compilation unit
// ...

function &foo()(){          // definition (contains the body)

// ... the body

};
```

`extern` cannot appear on a definition.

A function declaration marked `extern` cannot have a body, because that would make it a definition.

A variable marked `extern` is not defined, so it cannot have an initializer.

Only file scope global or readonly variables and function declarations can be marked `extern`.

The finalizer does not allocate space for names marked `extern`.

By default, all definitions of names starting with an ampersand (`&`) in a compilation unit are visible to other compilation units and have external linkage.

4.23.2 Static Linkage

A definition or declaration can have *static linkage* (marked `static`).

Only file scope global, readonly, global, and private variables and function declarations and definitions can be marked static.

No definition or declaration can have both static and external linkage. If a definition or declaration is marked `static`, then no other compilation name can use an `extern` to refer to the same object.

The name is not visible to any other compilation unit.

Static declarations must be defined in the current compilation unit.

4.23.3 None Linkage

Kernels have linkage none (neither external nor static and marked `none` in BRIG), because they can only be accessed from a dispatch call.

Labels and variables in the spill, arg, or kernarg segment have linkage none and are not visible to other compilation units.

Private and group variables defined inside a function or kernel have linkage none.

4.24 Dynamic Group Memory Allocation

Some developers like to write code using dynamically sized group memory. For example, in the following code there are four arrays allocated to group memory, two of known size and two of unknown size:

```
group_u32 dynamic[size];       // illegal as size not a constant value
group_u32 more_dynamic[size2]; // illegal as size not a constant value
group_u32 known[2];
group_u32 more_known[4];
kernel &k1 (kernarg_u32 size, kernarg_u32 size2){
    st_group_f32 1.0f, [dynamic][8];
    st_group_f32 2.0f, [more_dynamic];
...
}
```

Internally, group memory might be organized as:

```
start of group memory
offset 0, fixed
offset 8, more_fixed
offset 24, dynamic
offset ?, more_dynamic
end of group memory ?
```

The question marks indicate information that is not available at finalization time.

HSAIL does not support this sort of dynamically sized array because of two problems:

- The finalizer cannot emit code that addresses the array `more_dynamic`.
- The dispatch cannot launch the kernel because it does not know the amount of group space required for a work-group.

In order to provide equivalent functionality, the HSAIL runtime library supports dynamic allocation of group memory. Dynamic allocation of group memory uses these steps:

1. The application declares the HSAIL kernel with additional arguments, which are group segment addresses for the dynamically sized group memory, and uses these to access the dynamically sized group memory.
2. The finalizer calculates the amount of group memory used by the kernel and makes the information available as usual.
3. The application computes the group segment address for each of the additional kernel arguments by starting at the offset provided by the finalizer. The offset can be rounded up to meet any alignment requirements.
4. The application dispatches the kernel using the group segment addresses it computed, and specifies the amount of group memory as the sum of the amount returned by the finalizer plus the amount required for the dynamic group memory.

Using this mechanism, the previous example would be coded as follows:

```
group_u32 &known[2];
group_u32 &more_known[4];
kernel &k1 (kernarg_u32 %dynamic_ptr, kernarg_u32 %more_dynamic_ptr ){
    ld_kernarg_u32 $s1, [%dynamic_ptr];
    ld_kernarg_u32 $s2, [%more_dynamic_ptr];
    st_group_f32 1.0f, [$s1 + 8];
    st_group_f32 2.0f, [$s2];
    //...
    };
```

4.25 Kernarg Segment

The kernarg segment is used to hold kernel arguments.

Arguments to a kernel are always constant, because all work-items get the same values. Arguments to a kernel are read-only.

Kernarg variables can only be declared in the list of kernel formal arguments.

The finalizer might place kernarg variables in a specialized read-only cache.

The kernarg segment is typically written in memory by the agent that launches the kernel, and then read by the ISA when the kernel executes. Some additional rules apply so that the memory format of the kernarg segment is clearly specified:

- The size and layout of all kernel arguments can be determined by examining the kernel signature.
- Arguments follow the natural alignment rules. Stricter alignment can be forced with the `align` qualifier. For information about the `align` qualifier, see 4.22 Declaring and Defining Identifiers (p. 56).
- The base of the kernarg segment is always aligned to 16 bytes. Additionally, implementations can choose to add padding bytes to the end of the kernarg segment allocation to avoid false sharing conflicts.
- Arrays in the kernarg segment are passed by value. See 4.22.1 Array Declarations (p. 60).

- Image objects are always 48 bytes in size and are passed by value.
- Sampler objects are always 32 bytes in size and are passed by value.
- Kernel arguments are stored left-to-right in increasing memory locations. For example, the first kernel argument is stored at the kernarg base, the second is stored at the kernarg base + sizeof(first kernarg) (and is then aligned based on the type of the second argument), and so forth.
- HSA requires that the agent dispatching the command and the target HSA component have the same endian format.

In the following code, the load (`ld`) operation reads the contents of the address `z` into the register `$s1`:

```
kernel &top (kernarg_u32 %z)
{
    ld_kernarg_u32 $s1, [%z]; // read z into $s1

//...

};
```

It is possible to obtain the address of `z` with an `lda` operation:

```
lda_kernarg_u64 $d2, [%z]; // get the 64-bit pointer to z (a segment address)
```

Such addresses must not be used in store operations.

For more information, see 6.2 Load (ld) Operation (p. 126) and 5.8 Copy (Move) Operations (p. 83).

Chapter 5

Arithmetic Operations

This chapter describes the HSAIL arithmetic operations.

5.1 Overview of Arithmetic Operations

Unless stated otherwise, arithmetic operations expect all inputs to be in registers, immediate values, or `WAVESIZE` (see 2.6.2 Wavefront Size (p. 12)), and to produce a single result in a register (see 4.17 Operands (p. 50)).

Consider this operation:

```
max_s32 $s1, $s2, $s3;
```

In this case, the `max` operation is followed by a base type `s` and a length `32`.

Next there is a destination operand `s1`.

Finally, there are zero or more source operands, in this case `s2` and `s3`.

The type expands on the operation. For example, a `max` operation could be signed integer, unsigned integer, or floating-point.

The length determines the size of the register used. In the descriptions of the operations in this manual, a size *n* operation expects all input registers to be of length *n* bits. For more information on the rules concerning operands, see 4.17 Operands (p. 50).

Consider this operation:

```
add_s32 $s5, $s0, $s1;
```

This operation adds two registers `s0` and `s1` and stores the result into the register `s5`. However, it is possible that the result does not fit. An overflow occurs when an arithmetic operation attempts to create a numeric value that is larger than the value that can be represented. Integer overflow stores the least significant bits of the results. Floating-point overflow follows the rules of the IEEE/ANSI Standard 754-2008 for floating-point arithmetic.

5.2 Integer Arithmetic Operations

Integer arithmetic operations treat the data as signed (two's complement) or unsigned data types of 32-bit or 64-bit lengths.

HSAIL supports packed versions of some integer arithmetic operations.

5.2.1 Syntax

Table 5–1 Syntax for Integer Arithmetic Operations

Opcodes and Modifiers	Operands
abs_*sLength*	***dest*, *src0***
add_*TypeLength*	***dest*, *src0*, *src1***
borrow_*TypeLength*	***dest*, *src0*, *src1***
carry_*TypeLength*	***dest*, *src0*, *src1***
div_*TypeLength*	***dest*, *src0*, *src1***
max_*TypeLength*	***dest*, *src0*, *src1***
min_*TypeLength*	***dest*, *src0*, *src1***
mul_*TypeLength*	***dest*, *src0*, *src1***
mulhi_*TypeLength*	***dest*, *src0*, *src1***
neg_*sLength*	***dest*, *src0***
rem_*TypeLength*	***dest*, *src0*, *src1***
sub_*TypeLength*	***dest*, *src0*, *src1***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: s, u
Length: 32, 64

Explanation of Operands
dest: Destination register.
src0, *src1*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
A divide by zero exception or an implementation-defined exception are allowed for div and rem. No other exceptions are allowed.

Table 5–2 Syntax for Packed Versions of Integer Arithmetic Operations

Opcodes and Modifiers	Operand
abs_*Control_sLength*	***dest*, *src0***
add_*Control_TypeLength*	***dest*, *src0*, *src1***
max_*Control_TypeLength*	***dest*, *src0*, *src1***
min_*Control_TypeLength*	***dest*, *src0*, *src1***

Opcodes and Modifiers	Operand
mul_*Control*_*TypeLength*	*dest*, *src0*, *src1*
mulhi_*Control*_*TypeLength*	*dest*, *src0*, *src1*
neg_*Control*_s*Length*	*dest*, *src0*
sub_*Control*_*TypeLength*	*dest*, *src0*, *src1*

Explanation of Modifiers (see 4.15 Packing Controls for Packed Data (p. 47))
Control for abs and neg: p or s. *Control* for add, mul, and sub: pp, pp_sat, ps, ps_sat, sp, sp_sat, ss, or ss_sat. *Control* for max, min, and mulhi: pp, ps, sp, or ss.
Type: s, u.
Length: 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, or 64x2. See 4.14.2 Packed Data (p. 46).

Explanation of Operands
dest: Destination register.
src0, *src1*: Sources. Can be a register or immediate value.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.1 BRIG Syntax for Integer Arithmetic Operations (p. 314).

5.2.2 Description

abs
: The abs operation computes the absolute value of the source *src0* and stores the result into the destination *dest*. There are no unsigned versions of abs, so only abs_s*Length* is valid.

  abs(-2^{31}) returns -2^{31} for 32-bit operands. abs(-2^{63}) returns -2^{63} for 64-bit operands.

add
: The add operation computes the sum of the two sources *src0* and *src1* and stores the result into the destination *dest*. The add operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

`div`

The `div` operation divides source *`src0`* by source *`src1`* and stores the quotient in destination *`dest`*.

The `div` operation follows the c99 model for signed division: the remainder has the same sign as the dividend, and divide always truncates toward zero (–22/7 produces –3). The result of integer divide with a divisor of zero is undefined, and it is implementation-defined if: no exception is generated; a divide by zero exception is generated; or some other implementation-defined exception is generated.

The result of dividing -2^{31} for `s32` types, or -2^{63} for `s64` types, by –1 is undefined, and it is implementation-defined if: no exception is generated; or an implementation-defined exception is generated.

`rem`

The `rem` operation divides source *`src0`* by source *`src1`* and stores the remainder in destination *`dest`*.

The `rem` operation follows the c99 model for signed remainder: the remainder has the same sign as the dividend, and divide always truncates toward zero (–22/7 produces –1). The result of integer remainder with a divisor of zero is undefined, and it is implementation-defined if: no exception is generated; a divide by zero exception is generated; or some other implementation-defined exception is generated.

`rem(`-2^{31}`, -1)` returns 0 for `s32` types. `rem(`-2^{63}`, -1)` returns 0 for `s64` types.

`max`

The `max` operation computes the maximum of source *`src0`* and source *`src1`* and stores the result into the destination *`dest`*.

`min`

The `min` operation computes the minimum of source *`src0`* and source *`src1`* and stores the result into the destination *`dest`*.

`mul`

The `mul` operation produces the lower bits of the product. `mul` supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

It is undefined what value is returned by `mul(`-2^{31}`, -1)` for 32-bit operands, and `mul(`-2^{63}`, -1)` for 64-bit operands.

`mulhi`

`mulhi_s32` produces the upper bits of the 64-bit unsigned product; `mulhi_u32` produces the upper bits of the 64-bit unsigned product.

`mulhi_s64` produces the upper bits of the 128-bit unsigned product; `mulhi_u64` produces the upper bits of the 128-bit unsigned product.

For example: In the operation –1 x 1, the upper 32 bits of the signed integer product are all 1's while the upper 32 bits of the unsigned product are all 0's.

Similarly, for packed operands M x N, the top M bits of each of the N signed or unsigned products is placed in the packed M x N result.

To generate a 128-bit product from 64-bit sources, compilers can generate both 64-bit half results using `mul_u64/mul_s64` and `mulhi_u64/mulhi_s64` and then combine the partial results using a `combine` operation (see 5.8 Copy (Move) Operations (p. 83)).

`neg`

The `neg` operation computes 0 minus source *`src0`* and stores the result into the destination *`dest`*. There are no unsigned versions of `neg`, so only `neg_s`*`Length`* is valid.

`neg`(-2^{31}) returns -2^{31} for 32-bit operands. `neg`(-2^{63}) returns -2^{63} for 64-bit operands.

`sub`

The `sub` operation subtracts source *`src1`* from source *`src0`* and places the result in the destination *`dest`*.

The `sub` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

`carry`

The `carry` operation adds the two sources *`src0`* and *`src1`*. If the addition causes a carry out of the most significant (leftmost) bit, it sets the destination *`dest`* to 1; otherwise it sets the *`dest`* to 0.

The `carry` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

`borrow`

The `borrow` operation subtracts source *`src1`* from source *`src0`*. If the subtraction requires a borrow into the most significant (leftmost) bit, it sets the destination *`dest`* to 1; otherwise it sets the *`dest`* to 0.

The `borrow` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

Examples of Regular (Nonpacked) Operations

```
abs_s32 $s1, $s2;
abs_s64 $d1, $d2;

neg_s32 $s1, 100;
neg_s64 $d1, $d2;

add_s32 $s1, 42, $s2;
add_u32 $s1, $s2, 0x23;
add_s64 $d1, $d2, 23;
add_u64 $d1, 61, 0x233412349456;

div_s32 $s1, 100, 10;
div_u32 $s1, $s2, 0x23;
div_s64 $d1, $d2, 23;
div_u64 $d1, $d3, 0x233412349456;

max_s32 $s1, 100, 10;
max_u32 $s1, $s2, 0x23;
max_s64 $d1, $d2, 23;
max_u64 $d1, $d3, 0x233412349456;

min_s32 $s1, 100, 10;
min_u32 $s1, $s2, 0x23;
min_s64 $d1, $d2, 23;
min_u64 $d1, $d3, 0x233412349456;

mul_s32 $s1, 100, 10;
mul_u32 $s1, $s2, 0x23;
mul_s64 $d1, $d2, 23;
mul_u64 $d1, $d3, 0x233412349456;

mulhi_s32 $s1, $s3, $s3;
mulhi_u32 $s1, $s2, $s9;

rem_s32 $s1, 100, 10;
rem_u32 $s1, $s2, 0x23;
rem_s64 $d1, $d2, 23;
rem_u64 $d1, $d3, 0x233412349456;

sub_s32 $s1, 100, 10;
sub_u32 $s1, $s2, 0x23;
sub_s64 $d1, $d2, 23;
sub_u64 $d1, $d3, 0x233412349456;
```

Examples of Packed Operations

```
abs_p_s8x4 $s1, $s2;
abs_p_f32x2 $d1, $d1;

neg_s_u8x4 $s1, $s2;
neg_s_sat_u8x4 $s1, $s2;

add_pp_sat_u16x2 $s1, $s0, $s3;
add_pp_sat_u16x4 $d1, $d0, $d3;

mul_pp_u16x4 $d1, $d0, $d3;
mulhi_pp_u8x8 $d1, $d3, $d4;

sub_sp_u8x8  $d1, $d0, $d3;

max_pp_u8x4 $s1, $s0, $s3;

min_pp_u8x4 $s1, $s0, $s3;
```

5.3 Integer Optimization Operation

Integer optimizations are intended to improve performance. High-level compilers should attempt to generate these whenever possible.

See also 5.4 24-Bit Integer Optimization Operations (p. 72).

5.3.1 Syntax

Table 5–3 Syntax for Integer Optimization Operation

Opcode and Modifiers	Operands
mad_*TypeLength*	***dest*, *src0*, *src1*, *src2***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: s, u
Length: 32, 64

Explanation of Operands
dest: Destination register.
src0, *src1*, *src2*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.2 BRIG Syntax for Integer Optimization Operation (p. 315).

5.3.2 Description

The integer `mad` (multiply add) operation multiplies source *`src0`* times source *`src1`* and then adds source *`src2`*. The least significant bits of the result are then stored in the destination *`dest`*.

Integer `mad` supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

The math is: `((s0 * s1) + s2) & ((1 << length) – 1)`.

Examples

```
mad_s32 $s1, $s2, $s3, $s5;
mad_s64 $d1, $d2, $d3, $d2;
mad_u32 $s1, $s2, $s3, $s3;
mad_u64 $d1, $d2, $d3, $d1;
```

5.4 24-Bit Integer Optimization Operations

Integer optimizations are intended to improve performance. High-level compilers should attempt to generate these whenever possible. These operations operate on 24-bit integer data held in 32-bit registers.

For `s` types, the 24 least significant bits of the source values are treated as a two's complement signed value. The result is computed as a 48-bit two's complement value, and is undefined if the two's complement 32-bit source values are outside the range of $-2^{23}..2^{23}-1$. This allows an implementation to use equivalent 32-bit signed operations if it does not support native 24-bit signed operations.

For `u` types, the 24 least significant bits of the source values are treated as an unsigned value. The result is computed as a 48-bit unsigned value, and is undefined if the unsigned 32-bit source values are outside the range of $0..2^{24}-1$. This allows an implementation to use equivalent 32-bit unsigned operations if it does not support native 24-bit unsigned operations.

See also 5.3 Integer Optimization Operation (p. 71).

5.4.1 Syntax

Table 5–4 Syntax for 24-Bit Integer Optimization Operations

Opcode and Modifiers	Operands
mad24_*TypeLength*	***dest*, *src0*, *src1*, *src2***
mad24hi_*TypeLength*	***dest*, *src0*, *src1*, *src2***
mul24_*TypeLength*	***dest*, *src0*, *src1***
mul24hi_*TypeLength*	***dest*, *src0*, *src1***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: s, u
Length: 32

Explanation of Operands
dest: Destination register.
src0, *src1*, *src2*: Sources: Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.3 BRIG Syntax for 24-Bit Integer Optimization Operations (p. 315).

5.4.2 Description

mad24
: Computes the 48-bit product of the two 24-bit integer sources *src0* and *src1*. It then adds the 32 bits of *src2* to the result and stores the least significant 32 bits of the result in the destination.

mad24hi
: Computes mul24hi(*src0*, *src1*) + *src2* and stores the least significant 32 bits of the result in the destination.

mul24
: Computes the 48-bit product of the two 24-bit integer sources *src0* and *src1* and stores the least significant 32 bits of the result in the destination.

mul24hi
: Uses the same computation as mul24, but stores the most significant 16 bits of the 48-bit product in the destination. s32 sign-extends the result and u32 zero-extends the result.

Examples

```
mad24_s32 $s1, $s2, -12, 23;
mad24_u32 $s1, $s2, 12, 2;

mad24hi_s32 $s1, $s2, -12, 23;
mad24hi_u32 $s1, $s2, 12, 2;

mul24_s32 $s1, $s2, -12;
mul24_u32 $s1, $s2, 12;

mul24hi_s32 $s1, $s2, -12;
mul24hi_u32 $s1, $s2, 12;
```

5.5 Integer Shift Operations

These operations perform right or left shifts of bits.

These operations have a packed form.

5.5.1 Syntax

Table 5–5 Syntax for Integer Shift Operations

Opcode and Modifiers	Operands
shl_*TypeLength*	***dest*, *src0*, *src1***
shr_*TypeLength*	***dest*, *src0*, *src1***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: s, u
Length: For regular form: 32, 64; for packed form: 8x4, 8x8, 16x2, 16x4, 32x2, 32x4, or 64x2.

Explanation of Operands
dest: Destination register.
src0, *src1*: Sources. Can be a register, immediate value, or WAVESIZE. Regardless of *TypeLength*, *src1* is always u32.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.4 BRIG Syntax for Integer Shift Operations (p. 316).

5.5.2 Description for Standard Form

If the *Length* is 32, then the amount to shift ignores all but the lower five bits of *src1*. For example, shifts of 33 and 1 are treated identically.

If the *Length* is 64, then the amount to shift ignores all but the lower six bits of *src1*.

`shl`
: Shifts source *src0* left by the least significant bits of source *src1* and stores the result into the destination *dest*. This is the left arithmetic shift, adding zeros to the least significant bits. The value in *src1* is treated as unsigned.

  The `shl` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

`shr_s`
: Shifts source *src0* right by the least significant bits of source *src1* and stores the result into the destination *dest*. This is the right arithmetic shift, filling the exposed positions (the most significant bits) with the sign of *src0*. The value in *src1* is treated as unsigned.

`shr_u`
: Shifts source *src0* right by the least significant bits of source *src1* and stores the result into the destination *dest*. This is the right logical shift, filling the exposed positions (the most significant bits) with zeros. The value in *src1* is treated as unsigned.

Both `shr_s` and `shr_u` produce the same result if *src0* is non-negative or if the least significant bits of the shift amount (*src1*) is zero.

5.5.3 Description for Packed Form

Each element in *src0* is shifted by the same amount. The amount is in *src1*.

If the element size is 8 (that is, the *Length* starts with 8x), the shift amount is specified in the least significant 3 bits of *src1*.

If the element size is 16 (that is, the *Length* starts with 16x), the shift amount is specified in the least significant 4 bits of *src1*.

If the element size is 32 (that is, the *Length* starts with 32x), the shift amount is specified in the least significant 5 bits of *src1*.

If the element size is 64 (that is, the *Length* starts with 64x), the shift amount is specified in the least significant 6 bits of *src1*.

Examples

```
shl_u32 $s1, $s2, 2;
shl_u64 $d1, $d2, 2;
shl_s32 $s1, $s2, 2;
shl_s64 $d1, $d2, 2;

shr_u32 $s1, $s2, 2;
shr_u64 $d1, $d2, 2;
shr_s32 $s1, $s2, 2;
shr_s64 $d1, $d2, 2;

shl_u8x8 $d0, $d1, 2;
shl_u8x4 $s1, $s2, 2;
shl_u8x8 $d1, $d2, 1;
shr_u8x4 $s1, $s2, 1;
shr_u8x8 $d1, $d2, 2;
```

5.6 Individual Bit Operations

It is often useful to consider a 32-bit or 64-bit register as 32 or 64 individual bits and to perform operations simultaneously on each of the bits of two sources.

5.6.1 Syntax

Table 5–6 Syntax for Individual Bit Operations

Opcode and Modifiers	Operands
and_*TypeLength*	***dest, src0, src1***
or_*TypeLength*	***dest, src0, src1***
xor_*TypeLength*	***dest, src0, src1***
not_*TypeLength*	***dest, src0***
popcount_u32_*TypeLength*	***dest, src0***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: b
Length: 1, 32, 64; popcount does not support b1.

Explanation of Operands
dest: Destination register.
src0, *src1*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.5 BRIG Syntax for Individual Bit Operations (p. 316).

5.6.2 Description

The `b1` form is used with control (`c`) register sources. It can only be used with the operations `and`, `or`, `xor`, and `not`.

`and`
: Performs the bitwise AND operation on the two sources *`src0`* and *`src1`* and places the result in the destination *`dest`*. The `and` operation can be applied to 1-, 32-, and 64-bit values.

`or`
: Performs the bitwise OR operation on the two sources *`src0`* and *`src1`* and places the result in the destination *`dest`*. The `or` operation can be applied to 1-, 32-, and 64-bit values.

`xor`
: Performs the bitwise XOR operation on the two sources *`src0`* and *`src1`* and places the result in the destination *`dest`*. The `xor` operation can be applied to 1-, 32-, and 64-bit values.

`not`
: Performs the bitwise NOT operation on the source *`src0`* and places the result in the destination *`dest`*. The `not` operation can be applied to 1-, 32-, and 64-bit values.

`popcount`
: Counts the number of 1 bits in *`src0`*. Only `b32` and `b64` inputs are supported. The *`Type`* and *`Length`* fields specify the type and size of *`src0`*. *`dest`* has a fixed compound type of `u32` and must be a 32-bit register.

  See this pseudocode:

```
int popcount(unsigned int a)
{
   int d = 0;
   while (a != 0) {
      if (a & 1) d++;
      a >>= 1;
   }
   return d;
}
```

  See also Table 5–7 (p. 78).

Table 5–7 Inputs and Results for popcount Operation

Input	popcount
00000000	0
00ffffff	24
7fffffff	31
01ffffff	25
ffffffff	32
ffff0f00	20

Examples

```
and_b1 $c0, $c2, $c3;
and_b32 $s0, $s2, $s3;
and_b64 $d0, $d1, $d2;

or_b1 $c0, $c2, $c3;
or_b32 $s0, $s2, $s3;
or_b64 $d0, $d1, $d2;

xor_b1 $c0, $c2, $c3;
xor_b32 $s0, $s2, $s3;
xor_b64 $d0, $d1, $d2;

not_b1 $c1, $c2;
not_b32 $s0, $s2;
not_b64 $d0, $d1;

popcount_u32_b32 $s1, $s2;
popcount_u32_b64 $s1, $d2;
```

5.7 Bit String Operations

A common operation on elements is packing or unpacking a bit string. HSAIL provides bit string operations to access bit and byte strings within elements.

5.7.1 Syntax

Table 5–8 Syntax for Bit String Operations

Opcode and Modifiers	Operands
bitextract_*TypeLength*	***dest*, *src0*, *src1*, *src2***
bitinsert_*TypeLength*	***dest*, *src0*, *src1*, *src2*, *src3***
bitmask_*TypeLength*	***dest*, *src0*, *src1***

Opcode and Modifiers	Operands
bitrev_*TypeLength*	***dest*, *src0***
bitselect_*TypeLength*	***dest*, *src0*, *src1*, *src2***
firstbit_u32_*TypeLength*	***dest*, *src0***
lastbit_u32_*TypeLength*	***dest*, *src0***

Explanation of Modifiers (see Table 4–2 (p. 46))
Type: b for `bitmask`, `bitrev`, and `bitselect`; s and u for `bitextract`, `bitinsert`, `firstbit`, and `lastbit`.
Length: 32, 64.

Explanation of Operands
dest: Destination register. Must match the size of *Length*.
src0, *src1*, *src2*: Sources. Can be a register, immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.6 BRIG Syntax for Bit String Operations (p. 316).

5.7.2 Description

`bitextract`

Extracts a range of bits.

src0 and *dest* are treated as the *TypeLength* of the operation. *src1* and *src2* are treated as u32.

The least significant 5 (for 32-bit) or 6 (for 64-bit) bits of *src1* specify bit offset from bit 0. The least significant 5 (for 32-bit) or 6 (for 64-bit) of *src2* specify a bit-field width. *src0* specifies the replacement bits.

The bits are extracted from *src0* starting at bit position offset and extending for width bits and placed into the destination *dest*.

The result is undefined if the bit offset plus bit-field width is greater than the *dest* operand length.

`bitextract_s` sign-extends the most significant bit of the extracted bit field. `bitextract_u` zero-extends the extracted bit field.

```
offset = src1 & (operation.length == 32 ? 31 : 63);
width = src2 & (operation.length == 32 ? 31 : 63);
if (width == 0) {
  dest = 0;
} else {
  dest = (src0 << (operation.length - width - offset))
              >> (operation.length - width);
  // signed or unsigned >>, depending on operation.type
}
```

`bitinsert`

Replaces a range of bits.

src0, *src1*, and *dest* are treated as the *TypeLength* of the operation. *src2* and *src3* are treated as u32.

The least significant 5 (for 32-bit) or 6 (for 64-bit) bits of *src2* specify bit offset from bit 0. The least significant 5 (for 32-bit) or 6 (for 64-bit) of *src3* specify a bit-field width. *src0* specifies the bits into which the replacement bits specified by *src1* are inserted.

The result is undefined if the bit offset plus bit-field width is greater than the *dest* operand length.

The `bitinsert` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

```
offset = src2 & (operation.length == 32 ? 31 : 63);
width = src3 & (operation.length == 32 ? 31 : 63);
mask = (1 << width) - 1;
dest = (src0 & ~(mask << offset)) | ((src1 & mask) << offset);
```

`bitmask`

Creates a bit mask that can be used with `bitselect`.

`dest` is treated as the *`TypeLength`* of the operation. *`src0`* and *`src1`* are treated as `u32`.

The least significant 5 (for 32-bit) or 6 (for 64-bit) bits of *`src0`* specify bit offset from bit 0. The least significant 5 (for 32-bit) or 6 (for 64-bit) of *`src1`* specify a bit-mask width. *`dest`* is set to a bit mask that contains width consecutive 1 bits starting at offset.

The result is undefined if the bit offset plus bit mask width is greater than the *`dest`* operand length.

```
offset = src0 & (operation.length == 32 ? 31 : 63);
width = src1 & (operation.length == 32 ? 31 : 63);
mask = (1 << width) - 1;
dest = mask << offset;
```

`bitrev`

Reverses the bits in a register. For example, given `0x12345678`, the result would be `0x1e6a2c48`.

`bitselect`

Bit field select. This operation sets the destination *`dest`* to selected bits of *`src1`* and *`src2`*. The source *`src0`* is a mask used to select bits from *`src1`* or *`src2`*, using this formula:

```
dest = (src1 & src0) | (src2 & ~src0)
```

`firstbit_u`

For unsigned inputs, `firstbit` finds the first bit set to 1 in a number starting from the most significant bit. For example:

- `firstbit_u32_u32` of 0xffffff (all 1's) returns 0
- `firstbit_u32_u32` of 0x7fffffff (one 0 followed by 31 1's) returns 1
- `firstbit_u32_u32` of 0x01ffffff (seven 0's followed by 25 1's) returns 7

If no bits or all bits in *`src0`* are set, then *`dest`* is set to –1. The result is always a 32-bit register.

`Length` applies only to the source.

See this pseudocode:

```
int firstbit_u(uint a)
{
   if (a == 0)
   return -1;
   int pos = 0;
   while ((int)a > 0) {
      a <<= 1; pos++;
   }
   return pos;
}
```

See also Table 5–7 (p. 78).

`firstbit_s`

For signed inputs, `firstbit` finds the first bit set in a positive integer starting from the most significant bit, or finds the first bit clear in a negative integer from the most significant bit.

If no bits in *`src0`* are set, then *`dest`* is set to –1. The result is always a 32-bit register.

`Length` applies only to the source.

See this pseudocode:

```
int firstbit_s (int a)
{
   uint u = a >= 0? a: ~a; // complement negative numbers
   return firstbit_u(u);
}
```

See also Table 5–7 (p. 78).

`lastbit`

Finds the first bit set to 1 in a number starting from the least significant bit. For example, `lastbit` of 0x00000001 produces 0. If no bits in *`src0`* are set, then *`dest`* is set to –1. The result is always a 32-bit register.

`Length` applies only to the source.

The `lastbit` operation supports both signed and unsigned forms to aid readers of the code, though both forms compute the same result.

See this pseudocode:

```
int lastbit(uint a)
{
   if (a == 0) return -1;
   int pos = 0;
   while ((a&1) != 1) {
       a >>= 1; pos++;
   }
   return pos;
}
```

See also Table 5–7 (p. 78).

Table 5–9 Inputs and Results for firstbit and lastbit Operations

Input	firstbit	lastbit
00000000	–1	–1
00ffffff	8	0
7fffffff	1	0
01ffffff	7	0
ffffffff	0	0
ffff0f00	0	8

Examples

```
bitrev_b32 $s1, $s2;
bitrev_b64 $d1, 0x234;

bitextract_s32 $s1, $s1, 2, 3;
bitextract_u64 $d1, $d1, $s1, $s2;

bitinsert_s32 $s1, $s1, $s2, 2, 3;
bitinsert_u64 $d1, $d2, $d3, $s1, $s2;

bitmask_b32 $s0, $s1, $s2;

bitselect_b32 $s3, $s0, $s3, $s4;

firstbit_u32_s32 $s0, $s0;
firstbit_u32_u64 $s0, $d6;

lastbit_u32_u32 $s0, $s0;
lastbit_u32_s64 $s0, $d6;
```

5.8 Copy (Move) Operations

These operations perform copy or move operations.

5.8.1 Syntax

Table 5–10 Syntax for Copy (Move) Operations

Opcode and Modifiers	Operands
combine_v2_b64_b32	***dest*, (*src0*,*src1*)**
combine_v4_b128_b32	***dest*, (*src0*,*src1*,*src2*,*src3*)**
combine_v2_b128_b64	***dest*, (*src0*,*src1*)**
expand_v2_b32_b64	**(*dest0*,*dest1*), *src0***
expand_v4_b32_b128	**(*dest0*,*dest1*,*dest2*,*dest3*), *src0***
expand_v2_b64_b128	**(*dest0*,*dest1*), *src0***
lda_*segment*_**u*Length***	***dest*, *address-expression***
ldc_u*Length*	***dest*, *label***
ldc_u*Length*	***dest*, *function***
mov_*moveType*	***dest*, *src0***

Explanation of Modifiers
segment: Optional segment: global, group, private, kernarg, readonly, spill, or arg. If omitted, flat is used. (See 2.8 Segments (p. 13)).
Length: 1, 32, 64 , 128 (see Table 4–2 (p. 46)). For `lda` and `ldc`, must match the address size (see Table 2–3 (p. 20)).
moveType: `b1`, `b32`, `b64`, `b128`, `u32`, `u64`, `s32`, `s64`, `f16`, `f32`, `f64`, `roimg`, `rwimg`, `samp`.

Explanation of Operands
dest, *dest0*, *dest1*, *dest2*, *dest3*: Destination.
src0, *src1*, *src2*, *src3*: Sources. Can be a register, immediate value, or `WAVESIZE`.
address-expression: An address expression (see 4.18 Address Expressions (p. 52)).
label: A label identifier.
function: A function identifier.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.7 BRIG Syntax for Copy (Move) Operations (p. 317).

5.8.2 Description

`combine`
: Combines the values in the multiple source registers *src0*, *src1*, and so forth to form a single result, which is stored in the destination register *dest*. *src0* becomes the least significant bits, *src1* the next least significant bits, and so forth.

  This operation has a vector source made up of two or four registers. The length of each source multiplied by the number of source registers must equal the length of the destination register.

`expand`
: Splits the value in the source operand *src0* into multiple parts and stores them in the multiple destination registers *dest0*, *dest1*, and so forth. The least significant bits of the value are stored in *dest0*, the next least significant bits in *dest1*, and so forth.

  This operation has a vector destination made up of two or four registers. The length of each destination multiplied by the number of destination registers must equal the length of the source operand.

`lda`
This operation sets the destination *`dest`* to the address of the source.

If *`segment`* is present, the address is a segment address of that kind. If *`segment`* is omitted, the address is a flat address.

The address kind must match the source address expression (see 6.1.1 How Addresses Are Formed (p. 123)). The size of *`dest`* must match the address size of the segment (see Table 2–3 (p. 20)).

The address of a function or label cannot be taken. The `ldc` operation can be used instead.

This operation can be followed by an `stof` or `ftos` operation to convert to a flat or segment address if necessary.

You can use this operation to take the byte address of a function's formal parameter arg variable, but it cannot be used to take the address of an arg variable allocated in the current scope. This operation can also be used to take the byte address of a kernel's formal kernarg parameters.

`ldc`
Places the address of a label or function into the destination *`dest`*. The address of code is always in the global segment. The size of *`dest`* must match the address size of a label or function as appropriate (see Table 2–3 (p. 20)).

See also 8.4 Label Targets (labeltargets Statement) (p. 183).

`mov`
Copies the source *`src0`* into the destination *`dest`*.

5.8.3 Additional Information About lda

Assume the following:

- There is a variable `%g` in the group segment with group segment address 20.
- The group segment starts at flat address x.
- Register `$d0` contains the following flat address: `x + 10`.

If the address contains an identifier, then the segment for the identifier must agree with the segment used in the operation. `lda` only computes addresses. It does not convert between segments and flat addressing.

```
lda_u64 $d1, [$d0 + 10];     // sets $d1 to the flat address x + 20
mov_b64 $d1, $d0;            // sets $d1 to the flat address x + 10

lda_group_u32 $s1, [%g];     // loads the segment address of %g into $s1
stof_group_u64_u32 $d1, $s1; // convert $s1 to flat address in large machine
                             // model; result is (x + 20)
```

Examples

```
combine_v2_b64_b32 $d0, ($s0, $s1);
combine_v4_b128_b32 $q0, ($s0, $s1, $s2, $s3);
combine_v2_b128_b64 $q0, ($d0, $d1);

expand_v2_b32_b64 ($s0, $s1), $d0;
expand_v4_b32_b128 ($s0, $s1, $s2, $s3), $q0;
expand_v2_b64_b128 ($d0, $d1), $q0;

lda_private_u32 $s1, [&p];
global_u32 %g[3];
lda_global_u64 $d1, [%g];
stof_global_u64 $d0, $d1;
lda_global_u64 $d1, [$d1 + 8];

ldc_u64 $s1, &some_function;
ldc_u32 $s2, @lab;
// ...
@lab:

mov_b1 $c1, 0;

mov_b32 $s1, 0;
mov_b32 $s1, 0.0f;

mov_b64 $d1, 0;
mov_b64 $d1, 0.0;
```

5.9 Packed Data Operations

These operations perform shuffle, interleave, pack, and unpack operations on packed data. In addition, many of the integer (see 5.2 Integer Arithmetic Operations (p. 65)) and floating-point (see 5.11 Floating-Point Arithmetic Operations (p. 96)) operations support packed data as does the `cmp` operation (see 5.17 Compare (cmp) Operation (p. 112)).

See Table 5–11 (p. 86) and Table 5–12 (p. 87).

5.9.1 Syntax

Table 5–11 Syntax for Shuffle and Interleave Operations

Opcodes and Modifiers	Operands
shuffle_*TypeLength*	***dest*, *src0*, *src1*, *src2***
unpacklo_*TypeLength*	***dest*, *src0*, *src1***
unpackhi_*TypeLength*	***dest*, *src0*, *src1***

Explanation of Modifiers (see 4.14.2 Packed Data (p. 46))
Type: s, u, f.
Length: 8x4, 8x8, 16x2, 16x4, 32x2

Explanation of Operands
dest: Destination. See the Description below.
src0, *src1*: Sources. Must be a packed register or a constant value.
src2: Source. Must be a constant value used to select elements. See Table 5–13 (p. 90).

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.8 BRIG Syntax for Packed Data Operations (p. 317).

Table 5–12 Syntax for Pack and Unpack Operations

Opcodes and Modifiers	Operands
pack_*destTypedestLength_srcTypesrcLength*	***dest*, *src0*, *src1*, *src2***
unpack_*destTypedestLength_srcTypesrcLength*	***dest*, *src0*, *src1***

Explanation of Modifiers
destType: s, u, f.
srcType: s, u, f.
destLength: For pack, can be 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2. For unpack, can be 32, 64, and, if *destType* is f, can be 16.
srcLength: For pack, can be 32, 64, and, if *destType* is f, can be 16. For unpack, can be 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2.
See Table 4–2 (p. 46) and Table 4–3 (p. 47).

Explanation of Operands
dest: Destination register.
src0, *src1*, *src2*, *src3*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.8 BRIG Syntax for Packed Data Operations (p. 317).

5.9.2 Description

`shuffle`
: Selects half of the elements of *`src0`* based on controls in *`src2`* and copies them into the upper half of the *`dest`*. It then selects half of the elements of *`src1`* based on controls in *`src2`* and copies them into the lower half of the *`dest`*. See 5.9.3 Controls in src2 for shuffle Operation (p. 90).

`unpacklo`
: Copies and interleaves the lower elements from each source into the destination.

`unpackhi`
: Copies and interleaves the upper elements from each source into the destination.

`pack`

Assigns the elements of the packed value in *`src0`* to *`dest`*, replacing the element specified by *`src2`* with the value from *`src1`*.

`src0` is the same packed type as *`dest`*.

`src2` has the fixed compound type of `u32`. It specifies the index of the element to pack.

If the element count is 2 (that is, the *`Length`* ends with x2), the index is specified in the least significant bit of *`src2`*.

If the element count is 4 (that is, the *`Length`* ends with x4), the index is specified in the least significant 2 bits of *`src2`*.

If the element count is 8 (that is, the *`Length`* ends with x8), the index is specified in the least significant 3 bits of *`src2`*.

If the element count is 16 (that is, the *`Length`* ends with x16), the index is specified in the least significant 4 bits of *`src2`*.

The index 0 corresponds to the least significant bits, with higher values corresponding to elements with serially higher significant bits.

`src1` has the compound type *`srcTypesrcLength`*.

See 4.17 Operands (p. 50). The normal rules for source and destination operands apply but using the destination packed type's element compound type:

- The source and destination type (`s`, `u`, `f`) must match.
- For integer types, the source compound type size must be at least the size of the packed destination type's element size. If the source is a register, the register must be the size of the source compound type. If the source size is bigger than the destination type's element size, then the value will be truncated and the least significant bits used.
- For `f32` and `f64` types, if the source is a register, its size must match the destination type's element size.
- For `f16` type, if the source is a register, it must be an `s` register. Its value will be converted from the register representation to the memory representation to create a 16-bit value that is then used. See 4.21 Floating-Point Numbers (p. 55).

`unpack`

Assigns the element specified by *`src1`* from the packed value in *`src0`* to *`dest`*.

`src1` has the fixed compound type of `u32`. Its value must be in the range 0 to number of elements in the packed source type – 1. It is undefined if *`src1`* is out of this range. The value 0 corresponds to the least significant bits, with higher values corresponding to elements with serially more significant bits.

`src0` has the compound type *`srcTypesrcLength`*.

See 4.17 Operands (p. 50). The normal rules for source and destination operands apply but using the packed type's element compound type:

- The source and destination type (`s`, `u`, `f`) must match.
- For integer types, the source compound type size must be at least the size of the packed source type's element size. The destination register must be the size of the destination compound type. If the destination compound type size is bigger than the source type's element size, then the value will be sign-extended for `s` and zero-extended for `u`.
- For `f32` and `f64` types, the destination compound type must match the packed source type's element type. The destination register must be the size of the destination compound type.
- For `f16` type, the destination register must be an `s` register. The packed element value will be converted from the memory representation to the register representation. See 4.21 Floating-Point Numbers (p. 55).

5.9.3 Controls in src2 for shuffle Operation

`src2` of type `b32` or `b64` contains a set of bit selectors as shown in the table below.

The second column shows where the bits are copied to in the destination.

Table 5–13 Bit Selectors for shuffle Operation

src2 Bits for Packed Data Types s8x4 and u8x4	Copied to
1-0 selects one of four bytes from *`src0`*	*`dest`* bits 7-0
3-2 selects one of four bytes from *`src0`*	*`dest`* bits 15-8
5-4 selects one of four bytes from *`src1`*	*`dest`* bits 23-16
7-6 selects one of four bytes from *`src1`*	*`dest`* bits 31-24

src2 Bits for Packed Data Types s8x8 and u8x8	Copied to
2-0 selects one of eight bytes from *`src0`*	*`dest`* bits 7-0
5-3 selects one of eight bytes from *`src0`*	*`dest`* bits 15-8
8-6 selects one of eight bytes from *`src0`*	*`dest`* bits 23-16
11-9 selects one of eight bytes from *`src0`*	*`dest`* bits 31-24

src2 Bits for Packed Data Types s8x8 and u8x8	Copied to
14-12 selects one of eight bytes from *src1*	*dest* bits 39-32
17-15 selects one of eight bytes from *src1*	*dest* bits 47-40
20-18 selects one of eight bytes from *src1*	*dest* bits 55-48
23-21 selects one of eight bytes from *src1*	*dest* bits 63-56

src2 Bits for Packed Data Types s16x2, u16x2, and f16x2	Copied to
0 selects one of two 16-bit values from *src0*	*dest* bits 15-0
1 selects one of two 16-bit values from *src1*	*dest* bits 31-16

src2 Bits for Packed Data Types s16x4, u16x4, and f16x4	Copied to
1-0 selects one of four 16-bit values from *src0*	*dest* bits 15-0
3-2 selects one of four 16-bit values from *src1*	*dest* bits 31-16
5-4 selects one of four 16-bit values from *src0*	*dest* bits 47-32
7-6 selects one of four 16-bit values from *src1*	*dest* bits 63-48

src2 Bits for Packed Data Type f32x2	Copied to
0 selects one of two 32-bit values from *src0*	*dest* bits 31-0
1 selects one of two 32-bit values from *src1*	*dest* bits 63-32

5.9.4 Common Uses for shuffle Operation

Common uses for the shuffle operation include broadcast, swap, and rotate.

Broadcast

Broadcast the least significant data element into the destination:

```
shuffle_u8x4 dest, src0, src1, 0;
```

src2 is the constant 00 00 00 00 in bits.

Broadcast the second data element into the destination:

```
shuffle_u8x4 dest, src0, src1, 0x55;
```

src2 is the constant 01 01 01 01 in bits.

Broadcast the third data element into the destination:

```
shuffle_u8x4 dest, src0, src1, 0xaa;
```

src2 is the constant 10 10 10 10 in bits.

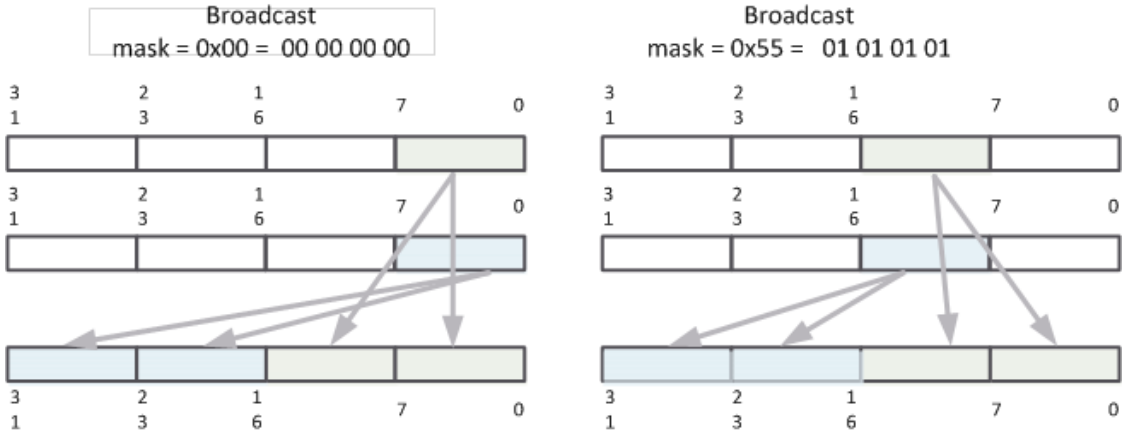
Broadcast the most significant data element into the destination:

```
shuffle_u8x4 dest, src0, src0, 0xff;
```

src2 is the constant 11 11 11 11 in bits.

See the figure below.

Figure 5–1 Example of Broadcast



Swap

Swap (switch the order of data elements; the reverse is 0x1b):

```
shuffle_u8x4 dest, src0, src0, 0x1b;
```

src2 is the constant 00 01 10 11 in bits.

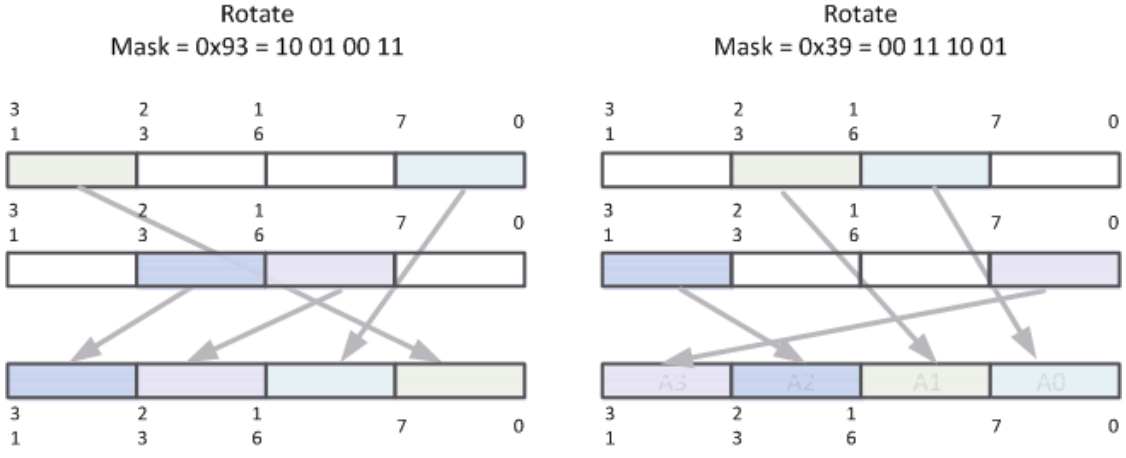
Rotate

To rotate:

- 0x93 is the left rotate (shifting data to the left); the most significant data element is moved to the least significant position.
- 0x39 is the right rotate (shifting data to the right); the least significant data element is moved to the most significant position.

See the figure below, which is an example of a shuffle with two specific masks.

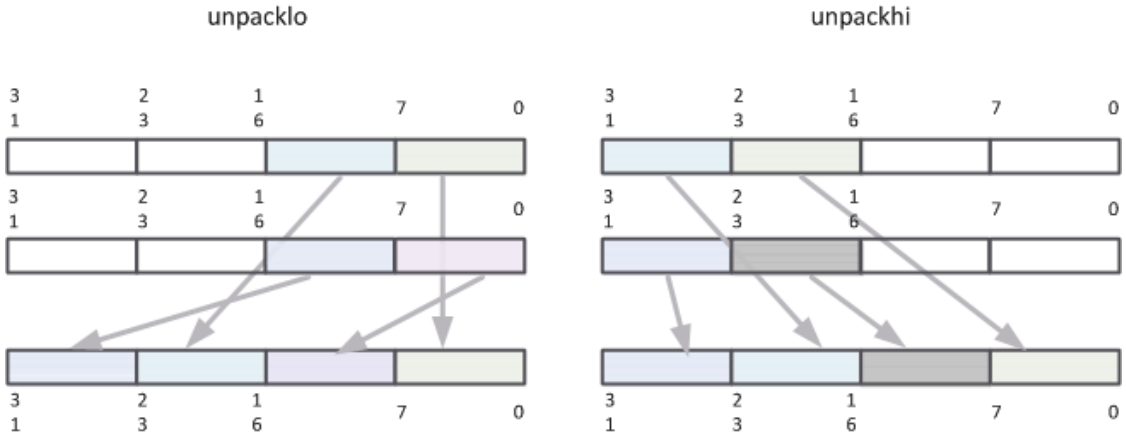
Figure 5–2 Example of Rotate



5.9.5 Examples of unpacklo and unpackhi Operations

See the figure below.

Figure 5–3 Example of Unpack



Examples

```
shuffle_u8x4 $s10, $s12, $s12, 0x55;
unpacklo_u8x4 $s1, $s2, 72;
unpackhi_f16x2 $s3, $s3,$s4;

// Packing with no conversions:
pack_f32x2_f32 $d1, $s2, 1;
pack_f32x4_f32 $q1, $s2, 3;
pack_u32x2_u32 $d1, $s1, 2;
pack_s64x2_s64 $q1, $d1, 0;

// Packing with integer truncation:
pack_u8x4_u32 $s1, $s2, $s3, 2;
pack_s16x4_s64 $d1, $d1, $d2, 0;
pack_u32x2_u64 $d1, $d2, $d3, 0;

// Packing an f16 and converting from the
// implementation-defined register representation:
pack_f16x2_f16 $s1, $s2, s3, 1;
pack_f16x4_f16 $d1, $d2, s3, 3;

// Unpacking with no conversions:
unpack_f32_f32x2 $s1, $d2, 1;
unpack_f32_f32x4 $s1, $q2, 3;
unpack_u32_u32x4 $s1, $q1, 2;
unpack_s64_s64x2 $d1, $q1, 0;

// Unpacking with integer sign or zero extension:
unpack_u32_u8x4 $s1, $s2, 2;
unpack_s32_s16x4 $s1, $d1, 0;
unpack_u64_u32x4 $d1, $q1, 2;
unpack_s64_s32x2 $d1, $d2, 0;

// Unpacking an f16 and converting to the
// implementation-defined register representation:
unpack_f16_f16x2 $s1, $s2, 1;
unpack_f16_f16x4 $s1, $d2, 3;

// Unpacking a u8 and converting to f32:
unpack_f32_u8x4 $s1, $s2, 0;
unpack_f32_u8x4 $s1, $s2, 1;
unpack_f32_u8x4 $s1, $s2, 2;
unpack_f32_u8x4 $s1, $s2, 3;
```

5.10 Bit Conditional Move (cmov) Operation

The `cmov` operation performs a bit conditional move.

There is a packed form of this operation.

5.10.1 Syntax

Table 5–14 Syntax for Bit Conditional Move (cmov) Operation

Opcode and Modifiers	Operands
`cmov_`*`TypeLength`*	*`dest, src0, src1, src2`*

Explanation of Modifiers (see Table 4–2 (p. 46))
`Type`: For the regular operation: `b`. For the packed operation: `s`, `u`, `f`.
`Length`: For the regular operation, *`Length`* can be 1, 32, 64. Applies to *`src1`*, and *`src2`*. For the packed operation, *`Length`* can be any packed type.

Explanation of Operands
`dest`: Destination register. For the packed form, if the length is 32 bits, then *`dest`* must be an `s` register; if the length is 64 bits, then *`dest`* must be a `d` register; if the length is 128 bits, then *`dest`* must be a `q` register.
`src0`, *`src1`*, *`src2`*: Sources. For the regular operation, *`src0`* must be a control (`c`) register or an immediate value and is of type `b1`. For the packed operation, if the *`Length`* is 32 bits, then *`src0`* must be an `s` register or constant value of type `u`*`Length`*; if the *`Length`* is 64 bits, then *`src0`* must be a `d` register or constant value of type `u`*`Length`*; if the *`Length`* is 128 bits, then *`src0`* must be a `q` register or constant value of type `u`*`Length`*. For the packed operation, each element in *`src0`* is assumed to contain either all 1's (true) or all 0's (false); results are undefined for other *`src0`* values.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.9 BRIG Syntax for Bit Conditional Move (cmov) Operation (p. 317).

5.10.2 Description

The regular form of `cmov` conditionally moves either of two 1-bit, 32-bit, 64-bit, or 128-bit values into the destination register *`dest`*. If the source *`src0`* is false (0), the destination is set to the value of *`src2`*; otherwise, the destination is set to the value of *`src1`*.

The packed form of `cmov` conditionally moves each element of the packed type independently. If the element in *`src0`* is false (0), the corresponding destination element is set to the corresponding element of *`src2`*; otherwise, the destination is set to the corresponding element of *`src1`*.

Examples

```
cmov_b32 $s1, $c3, $s1, $s2;
cmov_b64 $d1, $c3, $d1, $d2;
cmov_b32 $s1, $c0, $s1, $s2;

cmov_u8x4 $s1, $s0, $s1, $s2;
cmov_s8x4 $s1, $s0, $s1, $s2;
cmov_s8x8 $d1, $d0, $d1, $d2;
```

5.11 Floating-Point Arithmetic Operations

5.11.1 Overview

HSAIL provides a rich set of floating-point operations.

Most HSAIL floating-point operations follow the IEEE/ANSI Standard 754-2008 for floating-point arithmetic. However, there are important differences:

- Flags are supported using the DETECT exception policy and related operations. See 13.3 Hardware Exception Policies (p. 227).
- Some operations are fast approximations (the `nsqrt` operation is an example). See 5.13 Floating-Point Native Functions Operations (p. 104).
- Many operations that are not in the IEEE/ANSI Standard 754-2008 are provided.

The `ftz` (flush to zero) modifier, which forces subnormal values to zero, is supported on most operations.

If the Base profile has been specified and the operation supports the `ftz` modifier, then the `ftz` modifier must be specified. (See 17.2.2 Base Profile Requirements (p. 251).) Otherwise, if the operation supports the `ftz` modifier, it is optional, and if omitted the operation must not flush subnormal values to zero. In all other cases, the `ftz` modifier must not be specified.

If `ftz` is specified, it is performed on all source operands before the operation is performed. Any exceptions generated by the operation and subsequent rounding are based on the flushed source values.

All four IEEE/ANSI Standard 754-2008 rounding modes are supported for some operations: `up`, `down`, `near`, and `zero`. If the `round` modifier is omitted, and the operation supports a rounding mode, `near` is assumed. See 4.11 Rounding Modes (p. 39). If the Base profile has been specified, it is an error to use any rounding mode except `near`. (See 17.2.2 Base Profile Requirements (p. 251).)

It is implementation-defined if rounding generates underflow based on the value before or after rounding, but an implementation must use the same method for all operations. If the rounding specified by the operation does generate an underflow exception and `ftz` is specified, then the result must be set to 0.0 and the inexact exception generated if not already generated by the rounding. Note that the flush to zero of the result is required to be based on the generation of underflow, not on the result produced by rounding.

For all operations except `abs`, `copysign`, and `neg`:

- If one or more of the inputs is a signaling NaN, an invalid operation exception must be generated. (See Chapter 13 Exceptions (p. 225).)
- NaN payloads are supported for both double and single floating-point. If one or more inputs are NaNs, the result must be a quiet NaN. (The exception to this rule is `min` and `max` when one of the inputs is a NaN and the other is a number; see the Description below.) The NaN produced must be one of the following:
  - If the Base profile has been specified, it is implementation-defined which NaN will be returned and if the NaN payload is preserved. However, the sign must be preserved. (See 17.2.2 Base Profile Requirements (p. 251).)
  - Otherwise the NaN produced must be bit-identical to one of the inputs (it is implementation-defined which NaN will be returned), except signaling NaNs must be converted to quiet NaNs.
- If an operation produces a NaN, it must produce a quiet NaN.
- Operations are required to follow IEEE/ANSI Standard 754-2008 in generation of exceptions and generation of returned values if exceptions are generated.

For `abs`, `copysign`, and `neg`, the operation is performed as a bit operation, only acting upon the sign bit:

- They do not generate any exceptions, including underflow or inexact, nor invalid operation if any of their inputs are signaling NaNs.
- They do not flush subnormal values to 0.0.
- They do not convert signaling NaNs to quiet NaNs.
- For `abs` and `neg`, if the source is a NaN, its exact value must be preserved, except the sign bit is set to 0 for `abs`, and inverted for `neg`.
- For `copysign`, if either operand is a NaN, the result is still the exact bits of *`src0`* with the sign bit set to the sign bit value of *`src1`*.

HSAIL supports packed versions of some floating-point arithmetic operations. The value for each element of the packed result is the same as would be produced by the non-packed version of the operation, including handling of the `ftz` and rounding modifiers. For the packed `f16` types, both the source and destination elements are represented using the 16-bit memory representation, not the implementation-defined register representation.

5.11.2 Syntax

Table 5–15 Syntax for Floating-Point Arithmetic Operations

Opcode and Modifiers	Operands
abs_***TypeLength***	***dest, src0***
add_ftz_*round*_***TypeLength***	***dest, src0, src1***
ceil_ftz_***TypeLength***	***dest, src0***
copysign_***TypeLength***	***dest, src0, src1***
div_ftz_*round*_***TypeLength***	***dest, src0, src1***
floor_ftz_***TypeLength***	***dest, src0***

Opcode and Modifiers	Operands
fma_ftz_*round*_***TypeLength***	***dest, src0, src1, src2***
fract_ftz_***TypeLength***	***dest, src0***
max_ftz_***TypeLength***	***dest, src0, src1***
min_ftz_***TypeLength***	***dest, src0, src1***
mul_ftz_*round*_***TypeLength***	***dest, src0, src1***
neg_***TypeLength***	***dest, src0***
rint_ftz_***TypeLength***	***dest, src0***
sqrt_ftz_*round*_***TypeLength***	***dest, src0***
sub_ftz_*round*_***TypeLength***	***dest, src0, src1***
trunc_ftz_***TypeLength***	***dest, src0***

Explanation of Modifiers
`ftz`: Required if the Base profile has been specified, otherwise optional. If specified, forces subnormal values to zero, otherwise subnormal values are not flushed to zero.
round: Optional rounding mode. If the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251)) only `near`; otherwise `up`, `down`, `zero`, or `near`.
Type: `f` (see Table 4–2 (p. 46)).
Length: 16, 32, 64 (see Table 4–2 (p. 46)).

Explanation of Operands
dest: Destination register.
src0, *src1*, *src2*: Sources. Can be a register or immediate value.

Exceptions (see Chapter 13 Exceptions (p. 225))
Floating-point exceptions are allowed.

Table 5–16 Syntax for Packed Versions of Floating-Point Arithmetic Operations

Opcode and Modifiers	Operands
abs_***Control*_*TypeLength***	***dest, src0***
add_ftz_*round*_***Control*_*TypeLength***	***dest, src0, src1***
ceil_ftz_***Control*_*TypeLength***	***dest, src0***
copysign_***Control*_*TypeLength***	***dest, src0, src1***
div_ftz_*round*_***Control*_*TypeLength***	***dest, src0, src1***
floor_ftz_***Control*_*TypeLength***	***dest, src0***
fract_ftz_***Control*_*TypeLength***	***dest, src0***
max_ftz_***Control*_*TypeLength***	***dest, src0***

Opcode and Modifiers	Operands
min_ftz_***Control_TypeLength***	***dest*, *src0*, *src1***
mul_ftz_*round*_***Control_TypeLength***	***dest*, *src0*, *src1***
neg_***Control_TypeLength***	***dest*, *src0***
rint_ftz_***Control_TypeLength***	***dest*, *src0***
sqrt_ftz_*round*_***Control_TypeLength***	***dest*, *src0***
sub_ftz_*round*_***Control_TypeLength***	***dest*, *src0*, *src1***
trunc_ftz_***Control_TypeLength***	***dest*, *src0***

Explanation of Modifiers (see 4.15 Packing Controls for Packed Data (p. 47))
`ftz`: See table above.
round: See table above.
Control for `abs`, `ceil`, `floor`, `fract`, `neg`, `rint`, `sqrt`, and `trunc`: `p` or `s`. See 4.15 Packing Controls for Packed Data (p. 47).
Control for `add`, `copysign`, `div`, `max`, `min`, `mul`, and `sub`: `pp`, `ps`, `sp`, or `ss`.
TypeLength: `f16x2`, `f16x4`, `f16x8`, `f32x2`, `f32x4`, `f64x2`. See 4.14.2 Packed Data (p. 46).

Explanation of Operands
dest: Destination register.
src0, *src1*, *src2*: Sources. Can be a register or immediate value.

Exceptions (see Chapter 13 Exceptions (p. 225))
Floating-point exceptions are allowed.

For BRIG syntax, see 19.10.1.10 BRIG Syntax for Floating-Point Arithmetic Operations (p. 318).

5.11.3 Description

`abs`
: Copies a floating-point operand *src0* to the destination *dest*, setting the sign bit to 0 (positive). No rounding is performed.

  The `ftz` modifier is not supported.

`add`
: Performs the IEEE/ANSI Standard 754-2008 standard floating-point add.

`ceil`

Rounds the floating-point source *`src0`* toward positive infinity to produce a floating-point integral number that is assigned to the destination *`dest`*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

`copysign`

Copies a floating-point operand *`src0`* to the destination *`dest`*, but the sign of the destination is copied from the sign bit of *`src1`*. NaN payloads are preserved.

The `ftz` modifier is not supported.

`div`

Computes source *`src0`* divided by source *`src1`* and stores the result in the destination *`dest`*. This operation follows IEEE/ANSI Standard 754-2008 rules.

`div` must return a correctly rounded result in the Full profile and return a result within 2.5 ULP (unit of least precision) of the mathematically accurate value in the Base profile. (See Chapter 17 Profiles (p. 249).)

`floor`

Rounds the floating-point source *`src0`* toward negative infinity to produce a floating-point integral number that is assigned to the destination *`dest`*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

`fma`

The floating-point `fma` (fused multiply add) computes *`src0 * src1 + src2`* with unbounded range and precision. The resulting value is then rounded once using the specified rounding mode.

No underflow, overflow, or inexact exception can be generated for the multiply. However, these exceptions can be generated by the addition. Thus, `fma` differs from a `mul` followed by an `add`.

`fma` is not supported as a packed operation, because it takes three source operands.

`fract`

Sets the destination *`dest`* to the fractional part of source *`src0`*.

`fract_f16` returns `min_f16(x – floor(x),` *implementation-defined*.

`fract_f32` returns `min_f32(x – floor(x), 0x1.fffffep-1f)`.

`fract_f64` returns `min_f64(x - floor(x), 0x1.fffffffffffffp-1)`.

In all cases, the `min` is used to ensure that the result of the `fract` operation of a small negative number is not 1 so that the result is in the half-open interval [0, 1).

Because the register format for `f16` is implementation-defined (see 4.21 Floating-Point Numbers (p. 55)), the value used in the `min_f16` operation must be the largest value that can be exactly represented that is less than 1.0. For example, if the register representation of `f16` is the same as the memory representation, then `0x1.ffcp-1` must be used. Note that if the result of `fract` is stored to memory, it might become 1.0 due to the conversion to the memory format of `f16` (see 4.21 Floating-Point Numbers (p. 55)).

`max`
Computes the maximum of source *`src0`* and source *`src1`* and stores the result in the destination *`dest`*.

`max` implements the maxNum operation as described in IEEE/ANSI Standard 754-2008. If one of the inputs is a quiet NaN and the other input is not a NaN, then the non-NaN input is returned; otherwise NaN inputs are handled as described in 5.11.1 Overview (p. 96).

`min`
Computes the minimum of source *`src0`* and source *`src1`* and stores the result in the destination *`dest`*.

`min` implements the minNum operation as described in IEEE/ANSI Standard 754-2008. If one of the inputs is a quiet NaN and the other input is not a NaN, then the non-NaN input is returned; otherwise NaN inputs are handled as described in 5.11.1 Overview (p. 96).

`mul`
Multiplies source *`src0`* by source *`src1`* (following IEEE/ANSI Standard 754-2008 rules) and stores the result in the destination *`dest`*.

`neg`
Copies a floating-point operand *`src0`* to a destination *`dest`*, reversing the sign bit. `neg` is not the same as `sub(0, x)`. Consider `neg` of +0.0. `neg` preserves NaN payloads; only the sign bit is changed. `neg` does no rounding.

The `ftz` modifier is not supported.

`rint`
Rounds the floating-point source *`src0`* toward the nearest integral number, choosing the even integral value if there is a tie, to produce a floating-point integral number that is assigned to the destination *`dest`*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

`sub`
Subtracts source *`src1`* from source *`src0`* and places the result in the destination *`dest`*. The answer is computed according to IEEE/ANSI Standard 754-2008 rules.

`sqrt`
Sets the destination *`dest`* to the square root of source *`src0`*.

If *`src0`* is negative, must return a quiet NaN and generate the invalid operation exception.

`sqrt` returns the correctly rounded result for the Full profile and a result within 1 ULP of the mathematically accurate value for the Base profile. (See Chapter 17 Profiles (p. 249).)

`trunc`
Rounds the floating-point source *`src0`* toward zero to produce a floating-point integral number that is assigned to the destination *`dest`*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

Examples of Regular (Nonpacked) Operations

```
abs_f32 $s1,$s2; abs_f64 $d1,$d2;
add_f32 $s3,$s2,$s1;
add_f64 $d3,$d2,$d1;
copysign_f32 $s3,$s2,$s1;
copysign_f64 $d3,$d2,$d1;
div_f32 $s3,1.0f,$s1;
div_f64 $d3,1.0,$d0;
fma_f32 $s3,1.0f,$s1,23f;
fma_f64 $d3,1.0,$d0, $d3;
max_f32 $s3,1.0f,$s1;
max_f64 $d3,1.0,$d0;
min_f32 $s3,1.0f,$s1;
min_f64 $d3,1.0,$d0;
mul_f32 $s3,1.0f,$s1;
mul_f64 $d3,1.0,$d0;
neg_f32 $s3,1.0f;
neg_f64 $d3,1.0;
sub_f32 $s3,1.0f,$s1;
sub_f64 $d3,1.0,$d0;
fract_f32 $s0, 3.2f;
```

Examples of Packed Operations

```
version 1:0;
function &packed_ops (arg_u8x4 %x)() {
  abs_p_f16x2 $s1, $s2;
  abs_p_f32x2 $d1, $d1;
  neg_p_f16x2 $s1, $s2;
  add_pp_f16x2 $s1, $s0, $s3;
};
```

5.12 Floating-Point Classify (class) Operation

The floating-point classify (`class`) operation tests properties of a floating-point number in source *`src0`*, storing a 1 in the destination *`dest`* if any of the conditions specified in *`src1`* are true.

If all conditions are false, *`dest`* is set to zero.

5.12.1 Syntax

Table 5–17 Syntax for Floating-Point Classify (class) Operation

Opcode and Modifiers	Operands
**`class_b1_`*`sourceTypeSourceLength`*	***`dest, src0, src1`***

Explanation of Modifiers (see Table 4–2 (p. 46))
sourceType: f
sourceLength: 16, 32, 64

Explanation of Operands
dest: Destination. Must be a control (c) register.
src0: Source whose properties are being tested. Must be a register or immediate value of type *sourceTypeSourceLength* (see 4.17 Operands (p. 50)).
src1: Source bit set specifying the conditions being tested. Must be a register or immediate value of compound type u32 (see 4.17 Operands (p. 50)) See Table 5–18 (p. 103).

Table 5–18 Conditions and Source Bits

Condition being tested	Bit value
Signaling NaN	0x001
Quiet NaN	0x002
Negative infinity	0x004
Negative normal	0x008
Negative subnormal	0x010
Negative zero	0x020
Positive zero	0x040
Positive subnormal	0x080
Positive normal	0x100
Positive infinity	0x200

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.11 BRIG Syntax for Floating-Point Classify (class) Operation (p. 319).

5.12.2 Description

The values of Table 5–18 (p. 103) can be combined. Thus, the following code:

```
class_f32 $c1, $s1, 3;
```

will set the register c1 to 1 if $s1 is either a signaling or quiet NaN.

Examples

```
class_b1_f32 $c1, $s1, 3;
class_b1_f32 $c1, $s1, $s2;
class_b1_f64 $c1, $d1, $s2;
class_b1_f64 $c1, $d1, 3;
```

5.13 Floating-Point Native Functions Operations

The floating-point native functions operations provide fast approximate implementation values. They are intended to be used where speed is preferred over accuracy. Native operations can be used in device-specific libraries, and thus will know the accuracy of the operations on that device. They can also be used in code that first performs tests to ensure the current device's native operations meet the accuracy requirements of the algorithm.

These operations are expected to take advantage of hardware acceleration.

If one or more of the inputs is a signaling NaN, an invalid operation exception must be generated. If the operation produces a NaN, it must be a non-signaling NaN.

These operations do not support rounding modes or the flush to zero (`ftz`) modifier. It is implementation-defined how they round the result, whether or not subnormal values are flushed to zero, if NaN payloads are preserved (regardless of the profile specified), or if exceptions (other than for signaling NaNs) are generated. See 17.2 Profile-Specific Requirements (p. 250).

5.13.1 Syntax

Table 5–19 Syntax for Floating-Point Native Functions Operations

Opcode and Modifiers	Operands
ncos_f32	***dest, src***
nexp2_f32	***dest, src***
nfma_*TypeLength*	***dest, src0, src1, src2***
nlog2_f32	***dest, src***
nrcp_*TypeLength*	***dest, src***
nrsqrt_*TypeLength*	***dest, src***
nsin_f32	***dest, src***
nsqrt_*TypeLength*	***dest, src***

Explanation of Modifiers
Type: `f` (see Table 4–2 (p. 46)).
Length: 16, 32, 64 (see Table 4–2 (p. 46)).

Explanation of Operands
dest: Destination register.
src, *src0*, *src1*, *src2*: Sources. Can be a register or immediate value.

Exceptions (see Chapter 13 Exceptions (p. 225))
Standard floating-point exceptions are allowed.

For BRIG syntax, see 19.10.1.12 BRIG Syntax for Floating-Point Native Functions Operations (p. 319).

5.13.2 Description

`ncos`
Computes the cosine of the angle in source *src* and stores the result in the destination *dest*. The angle *src* is in radians. For `ncos`, input values outside the range [–512π, +512π] may be treated as 1.0.

`nexp2`
Computes the base-2 exponential of a value.

`nfma`
The floating-point `nfma` (native fused multiply add) computes a *src0* * *src1* + *src2* and stores the result in the destination *dest*.

`nlog2`
Finds the base-2 logarithm of a value.

`nrcp`
Computes the floating-point reciprocal.

`nrsqrt`
Computes the reciprocal of the square root.

`nsin`
Computes the sine of the angle in source *src* and stores the result in the destination *dest*. The angle *src* is in radians. For `nsin`, input values outside the range [–512π, +512π] may be treated as 1.0.

`nsqrt`
Computes the square root.

Examples

```
ncos_f32 $s1, $s0;

nexp2_f32 $s1, $s0;

nfma_f32 $s3, 1.0f, $s1, 23.0f;
nfma_f64 $d3, 1.0L, $d0, $d3;

nlog2_f32 $s1, $s0;

nrcp_f32 $s1, $s0;

nrsqrt_f32 $s1, $s0;

nsin_f32 $s1, $s0;
```

5.14 Multimedia Operations

These operations support fast multimedia operations. The operations work on special packed formats that have up to four values packed into a single 32-bit register.

5.14.1 Syntax

Table 5–20 Syntax for Multimedia Operations

Opcode	Operands
bitalign_b32	***dest*, *src0*, *src1*, *src2***
bytealign_b32	***dest*, *src0*, *src1*, *src2***
lerp_u8x4	***dest*, *src0*, *src1*, *src2***
packcvt_u8x4_f32	***dest*, *src0*, *src1*, *src2*, *src3***
unpackcvt_f32_u8x4	***dest*, *src0*, *src1***
sad_u32_u32	***dest*, *src0*, *src1*, *src2***
sad_u32_u16x2	***dest*, *src0*, *src1*, *src2***
sad_u32_u8x4	***dest*, *src0*, *src1*, *src2***
sad_hi_u32_u8x4	***dest*, *src0*, *src1*, *src2***

Explanation of Operands
dest: The destination must be an s register.
src0, *src1*, *src2*, *src3*: Sources. Can be a register, immediate value, or WAVESIZE, except *src1* for unpackcvt must be an immediate with value 0, 1, 2, or 3.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.13 BRIG Syntax for Multimedia Operations (p. 320).

5.14.2 Description

`bitalign`

Used to align 32-bits within 64-bits of data on an arbitrary bit boundary. *src2* is treated as a u32 value and the least significant 5 bits used to specify a shift amount. The 32-bit *src0* and *src1* are treated as the least significant and most significant bits of a 64-bit value respectively, which is shifted right by the shift amount of bits, and the least significant 32 bits returned.

```
uint32 shift = src2 & 31;
uint64_t value = (((uint64_t)src1) << 32) | ((uint64_t)src0);
uint32_t dest = (uint32_t)((value >> shift) & 0xffffffff);
```

If *src0* contains `0xA3A2A1A0` and *src1* contains `0xB3B2B1B0`, then:

- `bitalign` *dest*, *src0*, *src1*, `8` results in destination *dest* containing `0xB0A3A2A1`.
- `bitalign` *dest*, *src0*, *src1*, `16` results in destination *dest* containing `0xB1B0A3A2`.
- `bitalign` *dest*, *src0*, *src1*, `24` results in destination *dest* containing `0xB2B1B0A3`.

`bytealign`

Used to align 32-bits within 64-bits of data on an arbitrary byte boundary. *src2* is treated as a u32 value and the least significant 2 bits used to specify a shift amount. The 32-bit *src0* and *src1* are treated as the least significant and most significant bits of a 64-bit value respectively, which is shifted right by the shift amount of bytes, and the least significant 32 bits returned.

```
uint32 shift = (src2 & 3) * 8;
uint64_t value = (((uint64_t)src1) << 32) | ((uint64_t)src0);
uint32_t dest = (uint32_t)((value >> shift) & 0ffffffff);
```

If *src0* contains `0xA3A2A1A0` and *src1* contains `0xB3B2B1B0`, then:

- `bytealign` *dest*, *src0*, *src1*, `1` results in destination *dest* containing `0xB0A3A2A1`.
- `bytealign` *dest*, *src0*, *src1*, `2` results in destination *dest* containing `0xB1B0A3A2`.
- `bytealign` *dest*, *src0*, *src1*, `3` results in destination *dest* containing `0xB2B1B0A3`.

`lerp`

Linear interpolation (lerp) for multimedia format data. Computes the unsigned 8-bit pixel average.

Treating the sources as four 8-bit packed unsigned values, this operation adds each byte of *src0* and *src1* and the least significant bit of each byte of *src2* and then divides each result by 2.

```
dest = (((((src0 >> 24) & 0xff) + ((src1 >> 24) & 0xff) +
       ((src2 >> 24) & 0x1)) >> 1)) & 0xff) << 24) |
       (((((src0 >> 16) & 0xff) + ((src1 >> 16) & 0xff) +
       ((src2 >> 16) & 0x1)) >> 1)) & 0xff) << 16) |
       (((((src0 >> 8) & 0xff) + ((src1 >> 8) & 0xff) +
       ((src2 >> 8) & 0x1)) >> 1)) & 0xff) << 8) |
       (((src0 & 0xff) + (src1 & 0xff) + (src2 & 0x1)) >> 1)) & 0xff)
```

`packcvt`

Takes four floating-point numbers, converts them to unsigned integer values, and packs them into a packed `u8x4` value. Conversion is performed using round to nearest even. Values greater than 255.0 are converted to 255. Values less than 0.0 are converted to 0.

```
dest = (((uint32_t)(cvt_neari_sat_u8_f32(src0))) << 0)  |
       (((uint32_t)(cvt_neari_sat_u8_f32(src1))) << 8)  |
       (((uint32_t)(cvt_neari_sat_u8_f32(src2))) << 16) |
       (((uint32_t)(cvt_neari_sat_u8_f32(src3))) << 24);
```

`unpackcvt`

Unpacks a single element from a packed `u8x4` value and converts it to an `f32`. *src1* specifies the element and must be an immediate `u32` with a value of 0, 1, 2, or 3.

```
shift = src1 * 8;
dest = cvt_f32_u8((src0 >> shift) & 0xff);
```

`sad`

Computes the sum of the absolute differences of *src0* and *src1* and then adds *src2* to the result. `src0` and `src1` are either `u32`, `u16x2`, or `u8x4` and the absolute difference is performed treating the values as unsigned. The *dest* and *src2* are `u32` .

- `sad_u32_u32`:

```
uint32_t abs_diff(uint32_t a, uint32_t b) {
  return a < b ? b - a : a - b;
}

dest = abs_diff(src0, src1) + src2;
```

- `sad_u32_u16x2`:

```
uint32_t abs_diff(uint16_t a, uint16_t b) {
  return a < b ? b - a : a - b;
}

dest = abs_diff((src0 >> 16) & 0xffff, (src1 >> 16) & 0xffff) +
       abs_diff((src0 >>  0) & 0xffff, (src1 >>  0) & 0xffff) + src2;
```

- `sad_u32_u8x4`:

```
uint32_t abs_diff(uint8_t a, uint8_t b) {
  return a < b ? b - a : a - b;
}

dest = abs_diff((src0 >> 24) & 0xff, (src1 >> 24) & 0xff) +
       abs_diff((src0 >> 16) & 0xff, (src1 >> 16) & 0xff) +
       abs_diff((src0 >>  8) & 0xff, (src1 >>  8) & 0xff) +
       abs_diff((src0 >>  0) & 0xff, (src1 >>  0) & 0xff) + src2;
```

`sadhi`

Same as `sad` except the sum of absolute differences is added to the most significant 16 bits of *`dest`*. *`dest`* and *`src2`* are treated as a `u16x2`. *`src0`* and *`src1`* are treated as `u8x4`.

`sadhi_u16x2_u8x4` can be used in combination with `sad_u32_u8x4` to store two sets of sum of absolute differences results in a single s register as a `u16x2`. In this case, care must be taken that the `sad_u32_u8x4` will not overflow the least significant 16 bits.

- `sadhi_u16x2_u8x4`:

```
uint32_t abs_diff(uint8_t a, uint8_t b) {
  return a < b ? b - a : a - b;
}

dest = (abs_diff((src0 >> 24) & 0xff, (src1 >> 24) & 0xff) << 16) +
       (abs_diff((src0 >> 16) & 0xff, (src1 >> 16) & 0xff) << 16) +
       (abs_diff((src0 >>  8) & 0xff, (src1 >>  8) & 0xff) << 16) +
       (abs_diff((src0 >>  0) & 0xff, (src1 >>  0) & 0xff) << 16) +
        src2;
```

Examples

```
bitalign_b32 $s5, $s0, $s1, $s2;

bytealign_b32 $s5, $s0, $s1, $s2;

lerp_u8x4 $s5, $s0, $s1, $s2;

packcvt_u8x4_f32 $s1, $s2, $s3, $s9, $s3;

unpackcvt_f32_u8x4 $s5, $s0, 0;
unpackcvt_f32_u8x4 $s5, $s0, 1;
unpackcvt_f32_u8x4 $s5, $s0, 2;
unpackcvt_f32_u8x4 $s5, $s0, 3;

sad_u32_u32 $s5, $s0, $s1, $s6;
sad_u32_u16x2 $s5, $s0, $s1, $s6;
sad_u32_u8x4 $s5, $s0, $s1, $s6;

sadhi_u16x2_u8x4 $s5, $s0, $s1, $s6;
```

5.15 Segment Checking (segmentp) Operation

The `segmentp` operation tests whether or not a given flat address is within a specific memory segment.

See also 5.16 Segment Conversion Operations (p. 111).

5.15.1 Syntax

Table 5–21 Syntax for Segment Checking (segmentp) Operation

Opcode and Modifiers	Operands
segmentp_*segment*_b1_*srcTypesrcLength*	***dest, src***

Explanation of Modifiers
segment: Can be `global`, `group`, `private`, `kernarg`, `readonly`, `spill`, or `arg` (see 2.8 Segments (p. 13))
srcType: u (see Table 4–2 (p. 46))
srcLength: 32, 64. The size of the source address. Must match the address size of *segment* (see Table 2–3 (p. 20)).

Explanation of Operands
dest: Destination register. Must be a control (c) register.
src: Source for the flat address that is being checked. Can be a register or immediate value. (See Table 2–3 (p. 20).)

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.14 BRIG Syntax for Segment Checking (segmentp) Operation (p. 320).

5.15.2 Description

This operation sets the destination *dest* to true (1) if the flat address is within the address range of the specified segment. If the source is a register, it must match the size of a flat address. See 2.10 Small and Large Machine Models (p. 20).

Because implementations are allowed to merge certain segments (see Table 2–1 (p. 11)), `segmentp` might return different results. For example, the following operation must return true on implementations that merge the spill and private segments, and it must return false on implementations that keep them separate:

```
spill_b64 %x;
lda_spill_u32 $s0, [%x];
stof_spill_u64_u32 $d1, $s0;
segmentp_private_b1_u64 $c1, $d1;
```

See 2.9 Flat Memory and Agents (p. 18) for information about segments that can be combined.

Examples

```
segmentp_private_b1_u32 $c1, $s0; // small machine model
segmentp_global_b1_u32 $c1, $s0;  // small machine model
segmentp_group_b1_u64 $c1, $d0;   // large machine model
```

5.16 Segment Conversion Operations

The segment conversion operations convert a flat address into a segment address, or a segment address into a flat address.

See also 5.15 Segment Checking (segmentp) Operation (p. 110).

5.16.1 Syntax

Table 5–22 Syntax for Segment Conversion Operations

Opcodes and Modifiers	Operands
ftos_*segment*_*destTypedestLength*_*srcTypesrcLength*	***dest*, *src***
stof_*segment*_*destTypedestLength*_*srcTypesrcLength*	***dest*, *src***

Explanation of Modifiers
segment: `global`, `group`, `private`, `kernarg`, `readonly`, `spill`, or `arg`. (See 2.8 Segments (p. 13).)
destType: u. (See Table 4–2 (p. 46).)
destLength: 32, 64. The size of the destination address. For `ftos`, must be the address size of *segment*; for `stof`, must be the flat address size. (See Table 2–3 (p. 20).)
srcType: u. (See Table 4–2 (p. 46).)
srcLength: 32, 64. The size of the source address. For `ftos`, must be the flat address size; for `stof`, must be the address size of *segment*. (See Table 2–3 (p. 20)).

Explanation of Operands
dest: Destination register. (See Table 2–3 (p. 20).)
src: Source to be converted. Can be a register or immediate value. (See Table 2–3 (p. 20).)

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.1.15 BRIG Syntax for Segment Conversion Operations (p. 320).

5.16.2 Description

`ftos`
: Converts the flat address specified by *`src`* into a segment address and stores the result in the destination register *`dest`*. If the source is not in the specified segment or a segment that the implementation has combined with it, the result is undefined. (See 2.9 Flat Memory and Agents (p. 18).)

  The destination register size must match the size of the *`segment`* address. If the source is a register, it must match the size of a flat address. (See 2.10 Small and Large Machine Models (p. 20).

`stof`
: Converts the segment address specified by *`src`* into a flat address and stores the result in the destination register *`dest`*. The destination register size must match the flat address size. If the source is a register, it must match the size of the *`segment`* address. (See 2.10 Small and Large Machine Models (p. 20).

Examples

```
// large machine model conversions
stof_private_u64_u32 $d1, $s1;
ftos_group_u32_u64 $s1, $d2;
ftos_global_u64_u64 $d1, $d2;

// small machine model conversions
stof_private_u32_u32 $s1, $s2;
ftos_group_u32_u32 $s1, $s2;
ftos_global_u32_u32 $s1, $s2;
```

5.17 Compare (cmp) Operation

The compare (`cmp`) operation compares two numeric values. The value written depends on the type of destination *`dest`*.

`cmp` compares register-sized values, with one exception: for `f16`, `cmp` uses the implementation-defined `f16` register format for register operands, and immediate `f16` values are converted to the implementation-defined register format before the comparison (see 4.21 Floating-Point Numbers (p. 55)).

`cmp` also supports packed operands, returning one result per element.

The `ftz` (flush to zero) modifier, which forces subnormal values to zero, is supported if the source operand type is floating-point.

If the operation supports `ftz` and the Base profile has been specified, then `ftz` must be specified (see 17.2.2 Base Profile Requirements (p. 251)). Otherwise, it is optional.

If `ftz` is specified, the source operands that are subnormal values must be flushed to zero before performing the compare operation.

If the source operands are floating-point, and the comparison operation is not an `s` form, and one or more of them is a signaling NaN, then an invalid operation exception must be generated (see Chapter 13 Exceptions (p. 225)).

Floating-point comparison is required to follow IEEE/ANSI Standard 754-2008.

See Table 5–23 (p. 113) and Table 5–24 (p. 114).

5.17.1 Syntax

Table 5–23 Syntax for Compare (cmp) Operation

Opcode and Modifiers	Operands
`cmp_`*`op`*`_ftz_`*`destTypedestLength_srcTypesrcLength`	*`dest, src0, src1`*

Explanation of Modifiers (see Table 4–2 (p. 46))
op for bit types: `eq` and `ne`.
op for integer source types: `eq`, `ne`, `lt`, `le`, `gt`, `ge`.
op for floating-point source types: `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `equ`, `neu`, `ltu`, `leu`, `gtu`, `geu`, `num`, `nan` and signaling NaN forms `seq`, `sne`, `slt`, `sle`, `sgt`, `sge`, `sequ`, `sneu`, `sltu`, `sleu`, `sgtu`, `sgeu`, `snum`, `snan`.
`ftz`: Only valid for floating-point source types. Required if the Base profile has been specified, otherwise optional. If specified, forces subnormal values to zero, otherwise subnormal values are not flushed to zero.
`destTypedestLength`: Describes the destination.
`destType`: `b`, `u`, `s`, `f`.
`destLength`: 32, 64; 1 if source type is `b`; 16 if source type is `f`.
`srcTypesrcLength`: Describes the two sources.
`srcType`: `b`, `u`, `s`, `f`.
`srcLength`: 32, 64; 1 if source type is `b`; 16 if source type is `f`.

Explanation of Operands
`dest`: Destination register.
`src0`, *`src1`*: Sources. Each source can be a register, immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
sNaN floating-point numbers generate the invalid operation exception, except for the `s` comparison forms.

Table 5–24 Syntax for Packed Version of Compare (cmp) Operation

Opcode and Modifiers	Operands
`cmp_`***`op`***`_ftz_pp_`***`uLength_TypeLength`***	***`dest`***`, `***`src0`***`, `***`src1`***

Explanation of Modifiers (see 4.14.2 Packed Data (p. 46))
op: See table above.
`ftz`: See table above.
Type: `s`, `u`, `f`.
Length: `8x4`, `8x8`, `8x16`, `16x2`, `16x4`, `16x8`, `32x2`, `32x4`, `64x2`

Explanation of Operands
dest: Destination register. This operation performs an element-by-element compare and puts the result in the destination. *dest* must be a packed register of equal dimension as the sources. Each element in the packed destination is written to either all 1's (for true) or all 0's (for false) based on the result of each element-wise compare.
src0, *src1*: Sources. Must be a packed register or a constant value.

Exceptions (see Chapter 13 Exceptions (p. 225))
sNaN floating-point numbers generate the invalid operation exception, except for the s comparison forms.

For BRIG syntax, see 19.10.1.16 BRIG Syntax for Compare (cmp) Operation (p. 320).

5.17.2 Description for cmp Operation

The table below shows the value written into the destination *dest*.

Type of *dest*	True	False
floating-point	1.0	0.0
integer	0xffffffff	0
control	1	0

`eq`, `ne`
: These are the only compares that support the bit types as inputs.

`num`
: Returns true if both floating-point source operands are numeric values (not NaN).

`nan`
: Returns true if either floating-point source operand is NaN.

The ordered integer and floating-point comparisons are: `eq`, `ne`, `lt`, `le`, `gt`, `ge`. If either floating-point source operand is NaN, the result is false.

The unordered floating-point comparisons are: `equ`, `neu`, `ltu`, `leu`, `gtu`, `geu`. If both operands are numeric values (not NaN), then these comparisons have the same result as the ordered compare. If either operand is NaN, then the result of these comparisons is true.

Floating-point comparison ignores the sign of zero (so +0.0 equals –0.0), and infinite operands of the same sign compare as equal.

There are `s` forms of both the floating-point ordered and unordered comparisons. For example, `sle` is the `s` form of `le`. The difference between the `s` forms and non-`s` forms is in the way sNaNs are treated. For most floating-point operations, if one or both sources is an sNaN, an invalid operation exception will be generated. The floating-point compares are the exception: the non-`s` forms can trigger an exception while the `s` forms never generate an exception.

Examples

```
cmp_eq_b1_b1 $c1, $c2, 0;
cmp_eq_b32_b1 $s1, $c2, 0;
cmp_eq_f32_b1 $s1, $c2, 1;

cmp_ne_b1_b1 $c1, $c2, 0;
cmp_ne_b32_b1 $s1, $c2, 0;
cmp_ne_f32_b1 $s1, $c2, 1;

cmp_lt_b1_b32 $c1, $s2, 0;
cmp_lt_b32_b32 $s1, $s2, 0;
cmp_lt_f32_f32 $s1, $s2, 0.0f;

cmp_gt_b1_b32 $c1, $s2, 0;
cmp_gt_b32_b32 $s1, $s2, 0;
cmp_gt_f32_b32 $s1, $s2, 0.0f;

cmp_equ_b1_f32 $c1, $s2, 0f;
cmp_equ_b1_f64 $c1, $d1, $d2;

cmp_sltu_b1_f32 $c1, $s2, 0f;
cmp_sltu_b1_f64 $c1, $d1, $d2;

cmp_lt_pp_u8x4_u8x4 $s1, $s2, $s3;
cmp_lt_pp_u16x2_f16x2 $s1, $s2, $s3;
cmp_lt_pp_u32x2_f32x2 $d1, $d2, $d3;
```

5.18 Conversion (cvt) Operation

5.18.1 Overview

The conversion operation converts a value with a particular type and length to another value with a different type and/or length, or applies rounding to a value with a particular type and length to another value with the same type and length.

Conversion operations generally specify different types and lengths for both the destination and the source operands.

For floating point source types, if the source value is a signaling NaN, an invalid operation exception must be generated. If the source is a NaN and the result is a floating-point type, then the result must be a quiet NaN with the following value:

- If the source and destination floating-point types are not the same, then the source NaN payload is not preserved, because the types are different sizes. However, the sign must be preserved.
- Otherwise if the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251)), it is implementation-defined if the NaN payload of the source is preserved. However, the sign must be preserved.
- Otherwise the NaN produced must be bit-identical to the source, except a signaling NaN must be converted to a quiet NaN.

The `ftz` (flush to zero) modifier, which forces subnormal values to zero, is supported for conversion operations if the source type is floating-point. `ftz` is not allowed in any other cases.

If `ftz` is allowed and the Base profile has been specified, then `ftz` must be specified (see 17.2.2 Base Profile Requirements (p. 251)). Otherwise, if `ftz` is allowed, it is optional.

If `ftz` is specified, the source operand must be flushed to zero if it is a subnormal value, before performing the conversion operation. Any exceptions generated by the conversion are based on the flushed source value.

If the source operand type is `b1` or an integer type, it is allowed to have a register that is wider than the source type length. In this case, the least significant bits are used.

If the destination operand type is `b1` or an integer type, it is allowed to have a register that is wider than the destination type length. In this case, the conversion operations first transform the source to the destination type. The converted result is then zero-extended for `b1` and `u` types, and sign-extended for `s` types, to the size of the register.

A conversion acts as a move operation if the source and destination operand types are the following: `b1`, integer types with the same size, or floating-point types with the same size. For the `b1` and integer types, the move can be preceded by a truncation if the source operand is larger than the type, and followed by a zero extension or sign extension if the destination register is larger than the type.

No packed formats are supported.

Table 5–25 (p. 117) shows how the first step of the conversion operation does the transformation. The table uses the notation defined in Table 5–26 (p. 117).

Table 5–25 Conversion Methods

	Source b1	**Source u8**	**Source s8**	**Source u16**	**Source s16**	**Source f16**	**Source u32**	**Source s32**	**Source f32**	**Source u64**	**Source s64**	**Source f64**
Destination b1	-	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest
Destination u8	zext	-	-	chop	chop	h2u	chop	chop	f2u	chop	chop	d2u
Destination s8	zext	-	-	chop	chop	h2s	chop	chop	f2s	chop	chop	d2s
Destination u16	zext	zext	sext	-	-	h2u	chop	chop	f2u	chop	chop	d2u
Destination s16	zext	zext	sext	-	-	h2s	chop	chop	f2s	chop	chop	d2s
Destination f16	u2h	u2h	s2h	u2h	s2h	-	u2h	s2h	f2h	u2h	s2h	d2h
Destination u32	zext	zext	sext	zext	sext	h2u	-	-	f2u	chop	chop	d2u
Destination s32	zext	zext	sext	zext	sext	h2s	-	-	f2s	chop	chop	d2s
Destination f32	u2f	u2f	s2f	u2f	s2f	h2f	u2f	s2f	-	u2f	s2f	d2f
Destination u64	zext	zext	sext	zext	sext	h2u	zext	sext	f2u	-	-	d2u
Destination s64	zext	zext	sext	zext	sext	h2s	zext	sext	f2s	-	-	d2s
Destination f64	u2d	u2d	s2d	u2d	s2d	h2d	u2d	s2d	f2d	u2d	s2d	-

Table 5–26 Notation for Conversion Methods

ztest	For integer types, 1 if any input bit is 1, 0 if all bits are 0. For floating-point types, 1 if a non-zero number, NaN, +inf or –inf; 0 if +0.0 or –0.0.
chop	Delete all upper bits till the value fits.
zext	Extend the value adding zeros on the left.
sext	Extend the value, using sign extension.
f2u	Convert 32-bit floating-point to unsigned.
f2h	Convert 32-bit floating-point to 16-bit floating-point (half).
d2h	Convert 64-bit floating-point (double) to 16-bit floating-point (half).
h2f	Convert 16-bit floating-point (half) to 32-bit floating-point.
h2u	Convert 16-bit floating-point (half) to unsigned.
h2d	Convert 16-bit floating-point (half) to 64-bit floating-point (double).
d2u	Convert 64-bit floating-point (double) to unsigned.
f2s	Convert 32-bit floating-point to signed.
h2s	Convert 16-bit floating-point (half) to signed.
d2s	Convert 64-bit floating-point (double) to signed.
d2f	Convert 64-bit floating-point (double) to 32-bit floating-point.
s2f	Convert signed to 32-bit floating-point.
s2h	Convert signed to 16-bit floating-point (half).
s2d	Convert signed to 64-bit floating-point (double).

u2f	Convert unsigned to 32-bit floating-point.
u2h	Convert unsigned to 16-bit floating-point (half).
u2d	Convert unsigned to 64-bit floating-point (double).
-	Treat as a move.

5.18.2 Syntax

Table 5–27 Syntax for Conversion (cvt) Operation

Opcode and Modifiers	Operands
cvt_ftz_*round*_***destTypedestLength_srcTypesrcLength***	*dest*, *src*

Explanation of Modifiers
`ftz`: Only valid if *srcType* is floating-point. Required if the Base profile has been specified, otherwise optional. If specified, forces subnormal floating-point source and floating-point destination values to zero, otherwise they are not flushed to zero.
round: Only valid if *destType* and/or *srcType* is floating-point, unless both are floating-point types and *destType* size is equal to or larger than *srcType* size. Possible values are `up`, `down`, `zero`, `near`, `upi`, `downi`, `zeroi`, `neari`, `upi_sat`, `downi_sat`, `zeroi_sat`, and `neari_sat`. However, the allowed values depend on the *destType*, *srcType*, and whether the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251)). In some cases, *round* can be omitted, and defaults to `near` or `zeroi` as appropriate. See 5.18.3 Rules for Rounding for Conversions (p. 118), 5.18.4 Description of Integer Rounding Modes (p. 119), and 5.18.5 Description of Floating-Point Rounding Modes (p. 120).
destType: `b`, `u`, `s`, `f`. (For `b`, only `b1` is supported.) See Table 4–2 (p. 46).
destLength: 1, 8, 16, 32, 64. (For 1, only `b1` is supported.) See Table 4–2 (p. 46).
srcType: `b`, `u`, `s`, `f`. See Table 4–2 (p. 46).
srcLength: 1, 8, 16, 32, 64. 1 is only allowed for *srcType* of `b`. 1 and 8 are not allowed for *srcType* of `f`. See Table 4–2 (p. 46).

Explanation of Operands
dest: Destination register.
src: Source. Can be a register, immediate value, or (if *srcType* is an integer type) `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
Floating-point exceptions are allowed.

For BRIG syntax, see 19.10.1.17 BRIG Syntax for Conversion (cvt) Operation (p. 321).

5.18.3 Rules for Rounding for Conversions

Rounding for conversions follows the rules shown in Table 5–28 (p. 119).

If the type of rounding is "floating-point," the rounding mode can be omitted, in which case it defaults to `near`.

If the type of rounding is "integer," the desired rounding control must be explicitly specified. (This makes it clear that integer conversion is occurring.)

If the type of rounding is "none," then no rounding mode must be specified.

Table 5–28 Rules for Rounding for Conversions

From	To	Type of rounding	Default rounding
`f`	`f` (same size)	none (must not specify rounding)	none (no rounding performed)
`f`	`f` (smaller size)	floating-point	`near`
`f`	`f` (larger size)	none (must not specify rounding)	none (no rounding performed)
`s` or `u`	`f`	floating-point	`near`
`f`	`s` or `u`	integer	`zeroi`
`f`	`b1`	none (must not specify rounding)	none (always converts using ztest)
`b1`	`f`	none (must not specify rounding)	none (always converts to 0.0 or 1.0)
`b1`, `s`, or `u`	`b1`, `s`, or `u`	none (must not specify rounding)	none (no rounding performed)

5.18.4 Description of Integer Rounding Modes

Integer rounding modes are used for floating-point to integer conversions. Integer rounding modes are invalid in all other cases.

There are both regular and saturating integer rounding modes. They differ in the way they handle numeric results that are outside the range of the destination integer type. The floating-point source, after any flush to zero, is first rounded to an integral value according to the rounding mode. Then this rounded result is checked to determine if it is in range of the destination integer type.

A value is outside the range if it is a NaN, +inf, –inf, less than the smallest value that can be represented by the destination integer type, or greater than the largest value that can be represented by the destination integer type:

- For regular integer rounding modes, if the value is out of range, the result is undefined and will generate an invalid operation exception.
- For saturating integer rounding modes, if the value is out of range, the value is clamped to the range of the destination type, with NaN converted to 0.

If the source operand is a signaling NaN, an invalid operation exception must be generated (see Chapter 13 Exceptions (p. 225)).

An inexact exception must be generated if the source value, after any flush to zero, is in range but not an integral value.

The integer rounding mode can be omitted, in which case it defaults to `zeroi`. If the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251)), only `zeroi` and `zeroi_sat` are allowed.

The regular integer rounding modes might execute faster than the saturating integer rounding modes.

Regular Integer Rounding Modes

The regular integer rounding modes are:

- `upi` — Rounds up to the nearest integer greater than or equal to the exact result.
- `downi` — Rounds down to the nearest integer less than or equal to the exact result.
- `zeroi` — Rounds to the nearest integer toward zero.
- `neari` — Rounds to the nearest integer. If there is a tie, chooses an even integer.

Examples are:

If `$s1` has the value 1.6, then:

```
cvt_upi_f32_s32 $s2, $s1;   // sets $s2 = 2
cvt_downi_f32_s32 $s2, $s1; // sets $s2 = 1
cvt_zeroi_f32_s32 $s2, $s1; // sets $s2 = 1
cvt_neari_f32_s32 $s2, $s1; // sets $s2 = 2
```

If `$s1` has the value –1.6, then:

```
cvt_upi_f32_s32 $s2, $s1;   // sets $s2 = -1
cvt_downi_f32_s32 $s2, $s1; // sets $s2 = -2
cvt_zeroi_f32_s32 $s2, $s1; // sets $s2 = -1
cvt_neari_f32_s32 $s2, $s1; // sets $s2 = -2
```

Saturating Integer Rounding Modes

The saturating integer rounding modes are:

- `upi_sat`
- `downi_sat`
- `zeroi_sat`
- `neari_sat`

If the source is a NaN, then the result is 0. Otherwise, the corresponding regular integer rounding mode is first performed. Then for unsigned destination types: If the rounded result is –inf or less than 0.0, then 0 is stored; if the rounded result is +inf or greater than $2^{destLength}-1$, then $2^{destLength}-1$ is stored. Otherwise for signed destination types: If the rounded result is –inf or less than $-2^{destLength-1}$, then $-2^{destLength-1}$ is stored; if the rounded result is +inf or greater than $2^{destLength-1}-1$, then $2^{destLength-1}-1$ is stored.

5.18.5 Description of Floating-Point Rounding Modes

The floating-point rounding modes are:

- `up` — Rounds up to the nearest representable value that is greater than the infinitely precise result.
- `down` — Rounds down to the nearest representable value that is less than the infinitely precise result.
- `zero` — Rounds to the nearest representable value that is no greater in magnitude than the infinitely precise result.
- `near` — Rounds to the nearest representable value. If there is a tie, chooses the one with an even least significant digit.

Floating-point rounding modes are allowed in the following cases:

- A floating-point rounding mode is allowed for conversions from a floating-point type to a smaller floating-point type. These conversions can lose precision.

  The floating-point rounding mode can be omitted, in which case it defaults to `near`. If the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251), only `near` is allowed.

  Rounding is required to follow IEEE/ANSI Standard 754-2008 in generation of exceptions and generation of returned values if exceptions are generated.

  If the source operand is a signaling NaN, then an invalid operation exception must be generated (see Chapter 13 Exceptions (p. 225)), and a non-signaling NaN must be produced with the same sign as the source NaN.

  After any flush to zero, if the converted value is not the same as the source value, then an inexact exception must be generated.

  If the source value is infinity, then the result is an infinity of the same sign. The inexact exception is not generated.

  It is implementation-defined if conversion generates underflow based on the value before or after rounding, but an implementation must use the same method for all operations. If the rounding specified by the conversion does generate an underflow exception, and `ftz` is specified, then the result must be set to 0.0 and the inexact exception generated if not already generated by the rounding. Note that the flush to zero of the result is required to be based on the generation of underflow, not on the result produced by rounding.
- A floating-point rounding mode is allowed for integer to floating-point conversions. The floating-point rounding mode can be omitted, in which case it defaults to `near`. If the Base profile has been specified (see 17.2.2 Base Profile Requirements (p. 251), only `near` is allowed.

  If the source value cannot be exactly represented in the destination type, then an inexact exception must be generated.

Floating-point rounding modes are invalid in all other cases.

Examples

```
cvt_f32_f64 $s1, $d1;
cvt_upi_u32_f32 $s1, $s2;
cvt_ftz_f32_f32 $s1, $s2;
cvt_u32_f32 $s1, $s2;
cvt_f16_f32 $s1, $s2;
cvt_s32_u8 $s1, $s2;
cvt_s32_b1 $s1, $c2;
cvt_f32_f16 $s1, $s2;
cvt_s32_f32 $s1, $s2;
cvt_ftz_upi_s8_f32 $s1, $s2;
```

Chapter 6

Memory Operations

This chapter describes the HSAIL memory operations.

6.1 Memory and Addressing

Note: See also 6.7 Examples of Memory Operations (p. 145).

Memory operations transfer data between registers and memory:

- The load operations move contents from memory to a register.
- The store operations move contents of a register into memory.
- The atomicReturn operations read a value from memory, update the memory location, and set the destination to the original value.
- The atomicNoReturn operations read a value from memory and update the memory location. (They do not have a destination.)

A flat memory, global, readonly, or kernarg segment address is a 32- or 64-bit value, depending on the machine model. A group, private, spill, or arg segment address is always 32 bits regardless of machine model. See 2.10 Small and Large Machine Models (p. 20)). Each operation indicates the type of address.

Memory operations can do either of the following:

- Specify the particular segment used, in which case the address is relative to the start of the segment.
- Use flat addresses, in which case hardware will recognize when an address is within a particular segment.

See 2.8.3 Addressing for Segments (p. 17).

6.1.1 How Addresses Are Formed

The format of an address expression is given in 4.18 Address Expressions (p. 52).

Every address expression has one or both of the following:

- Name in square brackets.

  If the operation uses segment addressing, the name is converted to the corresponding segment address. The behavior is undefined if the name is not in the same segment specified in the memory operation.

- Register plus or minus an offset in square brackets.

  Either the register or the offset can be optional. The size of the register must match the size of the address required by the operation. For example, an `s` register must be used for a group segment address, a `d` register must be used for a global segment address in the large machine model, and an `s` register must be used for a global address in the small machine model. See Table 2–3 (p. 20).

An address is formed from an address expression as follows:

1. Start with address 0.
2. If there is an identifier, add the byte offset of the variable referred to by the identifier within its segment to the address. If the memory operation specifies flat addressing, add the byte address of the start of the variable's segment. The segment of the variable must be the same as the segment specified in the operation using the address.
3. If there is a register, add the value of the register to the address.
4. If there is an offset, add or subtract the offset. The offset is read as a 64 bit integer constant (see 4.13.1 Integer Constants (p. 41)).

All address arithmetic is done using unsigned two's complement arithmetic truncated to the size of the address.

The address formed is then translated to an effective address to determine which memory location is accessed. See 2.8.3 Addressing for Segments (p. 17).

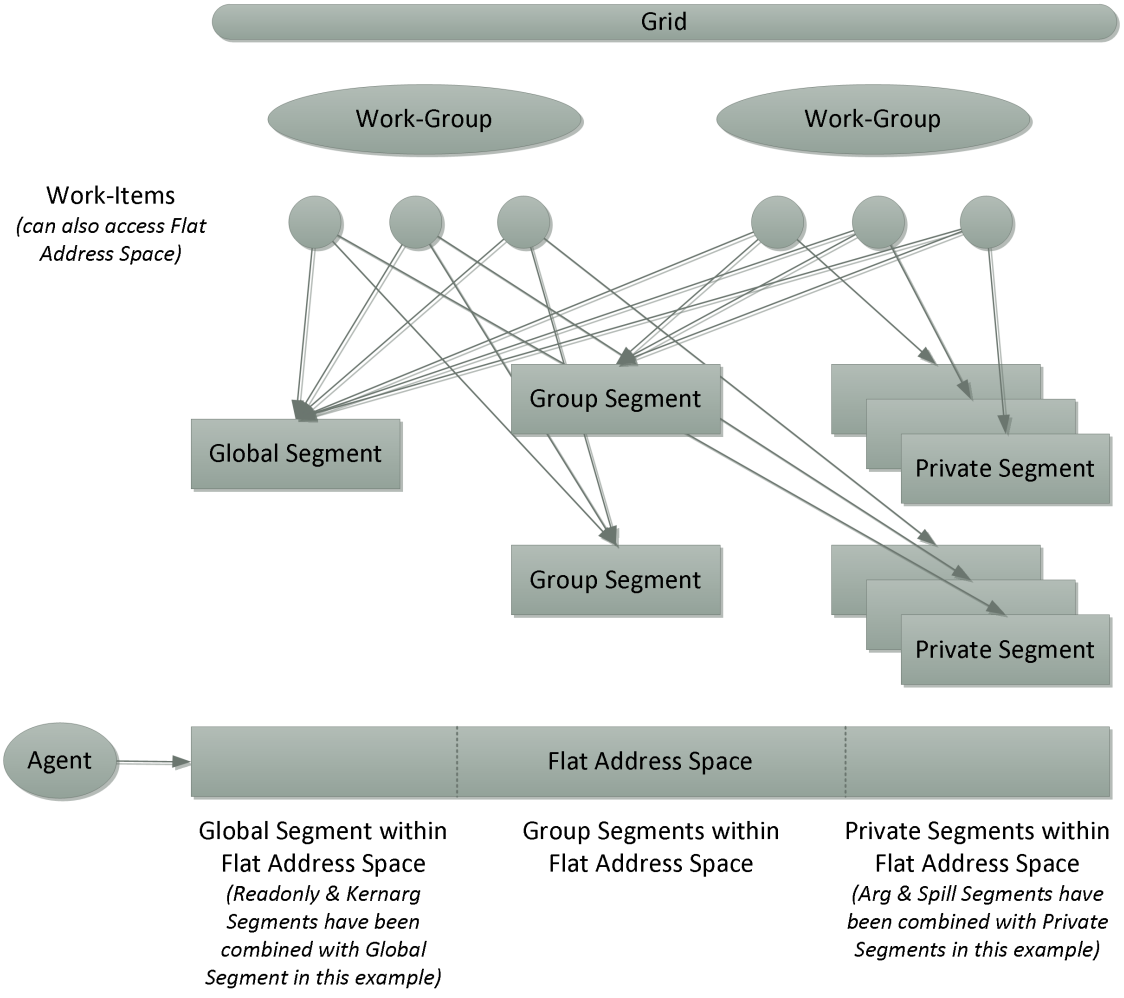
If the resulting effective address value is outside the memory segment specified by the operation, or is a flat address that is outside any segment, the result of the memory segment operation is undefined.

For more information, see 4.18 Address Expressions (p. 52).

6.1.2 Memory Hierarchy

See Figure 6–1 (p. 125). This figure shows the memory hierarchy for an implementation that has combined spill and arg with the private segment, and kernarg and readonly with the global segment.

Figure 6–1 Memory Hierarchy



The address does not need to be naturally aligned to a multiple of the item size.

The segment converting operations (`ftos` and `stof`) convert addresses between flat address and segment address.

The segment checking operation (`segmentp`) can be used to check which segment contains a particular flat address.

6.1.3 Alignment

A memory operation of size `n` bytes is "naturally aligned" if and only if its address is an integer multiple of `n`. For example, naturally aligned 8-byte stores can only be to addresses 0, 8, 16, 24, 32, and so forth.

HSAIL implementations can perform certain memory operations as a series of steps.

For example, an unaligned store might be implemented as a series of aligned stores, as follows: A load (store) is naturally aligned if the address is a multiple of the amount of data loaded (stored). Thus, storing four bytes at address 3 is not naturally aligned. Under certain conditions, implementations could split this up into four separate 1-byte stores.

6.1.4 Equivalence Classes

Equivalence classes can be used to provide alias information to the finalizer.

Equivalence classes are specified with the `ld` and `st` operations.

There are 256 equivalence classes.

Class 0, the default, is general memory. It can interact with all other classes.

The finalizer will assume that any two memory operations in different classes N > 0 and M > 0 (with N not equal to M) do not overlap and can be reordered. Equivalence classes in different segments never overlap.

For example, memory specified by the `ld` or `st` operations as class 1 can only interact with class 1 and class 0 memory.

Memory specified as class 2 can only interact with class 2 and class 0 memory.

Memory specified as class 3 can only interact with class 3 and class 0 memory. And so on.

6.2 Load (ld) Operation

The load (`ld`) operation loads from memory using a segment or flat address expression (see 4.18 Address Expressions (p. 52)) and places the result into one or more registers.

There are four variants of the `ld` operation, depending on the number of destinations: one, two, three, or four.

The size of the value loaded is specified by the operation's compound type. The value is stored into the destination register following the rules in 4.17 Operands (p. 50). Integer values are sign-extended or zero-extended to fit the destination register size. `f16` values are converted to the implementation-defined register format (see 4.21 Floating-Point Numbers (p. 55)). No conversions are performed on other types. Use an explicit `cvt` operation if floating-point conversion is required.

6.2.1 Syntax

Table 6–1 Syntax for Load (ld) Operation

Opcode and Modifiers	Operands
ld_*width_segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	***dest, address***
ld_v2_*width_segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	***(dest0,dest1), address***
ld_v3_*width_segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	***(dest0,dest1,dest2), address***
ld_v4_*width_segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	***(dest0,dest1,dest2,dest3), address***

Explanation of Modifiers
`v2`, `v3`, and `v4`: Optimization flags. See the Description below.
`width`: Optional. Specifies the number of consecutive work-items in flattened ID order that are guaranteed to load the same value. See the Description below.
`segment`: Optional segment: `global`, `group`, `private`, `kernarg`, `readonly`, `spill`, or `arg`. If omitted, flat is used. See 2.8 Segments (p. 13).
`aligned`: Optional. See the Description below.
`sem`: Optional memory semantics flag. See the Description below.
`equiv(`*`n`*`)`: Optional. *`n`* is an equivalence class. If omitted, class 0 is used. See 6.1.4 Equivalence Classes (p. 126).
`Type`: `u`, `s`, `f`. The *`Type`* specifies how the value is expanded to the size of the destination. See Table 4–2 (p. 46).
`Length`: 8, 16, 32, 64. The *`Length`* specifies the amount of data fetched from memory, and the amount to increment the address when the destination is a vector operand. See Table 4–2 (p. 46).
`TypeLength` can also be `b128`, in which case *`dest`* must be a `q` register; or `roimg`, `rwimg`, or `samp`, in which case `dest` must be a `d` register.

Explanation of Operands
`dest`, *`dest0`*, *`dest1`*, *`dest2`*, *`dest3`*: Destination registers.
`address`: Address to be loaded from.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the `aligned` modifier has been specified.

For BRIG syntax, see 19.10.2 BRIG Syntax for Memory Operations (p. 321).

6.2.2 Description

v2, v3, and v4

These flags are strictly optimizations. When v2, v3, or v4 is used, HSAIL will load consecutive values into multiple registers. The address is incremented by the *`Length`* in the operation. Front ends should generate vector forms whenever possible.

The following forms are equivalent but the vector form is often faster.

Slow form:

```
ld_s32 $d0, [$s1];
ld_s32 $d1, [$s1+4];
```

Fast form using the vector:

```
ld_v2_s32 ($d0,$d1), [$s1];
```

width

Can be used to specify that consecutive work-items in flattened ID order will load the same address.

The value can be `width(n)`, `width(WAVESIZE)`, or `width(all)`.

Implementations are allowed to have a single work-item read the value and then broadcast the result to the other work-items. Some implementations can use this modifier to speed up computations.

If work-items specified by `width` do not load the same address, the behavior is undefined.

If `width` is not provided, every work-item can load data from a different address (in other words, `width(1)` is the default).

See 2.13.1 Width Modifier (p. 22)).

`aligned`

If specified, indicates that the implementation can rely on the source address having the natural alignment of the destination type. On some implementations, this might be more efficient. It is undefined if a memory load marked as aligned is in fact unaligned: on some implementations this might result in incorrect values being loaded or memory exceptions being generated. See 18.6 Unaligned Access (p. 255). If `aligned` is omitted, the implementation must correctly handle the source address being unaligned.

sem

Memory semantics flag, most commonly used for synchronization of memory operations and read/write image operations.

For read/write images, the memory semantic flag applies only to read/write image operations on the same HSA component.

Memory operations are the load, store, and atomic operations defined in this chapter. Read/write image operations are the `rdimage`, `ldimage`, `stimage`, `atomicimage`, and `atomicimagenoret` operations defined in Chapter 7 Image Operations (p. 151).

The flag can be `acq` or `part_acq`:

- `acq` creates a downward fence. This means that memory and read/write image operations can be moved (by the implementation) down after the `ld_acq`, but no memory or read/write image operation can be moved before this point. An `ld_acq` marks this memory operation as synchronizing: all synchronizing operations will stay in order.

  One common use of `ld_acq` is to acquire a lock for synchronization. In this case, no code in a critical section after the lock can be moved out of the critical section above the load of the lock. However, code before the lock is acquired can be moved into the critical section.

- `part_acq` specifies that no memory or read-write image operation can be moved before this point as far as this work-group can see, but other work-groups are allowed to see the memory and read-write image operations on the same HSA component in a different order.

If omitted, the load will be an ordinary (not synchronizing) load.

6.2.3 Additional Information

If *segment* is present, the address expression must be a segment address of the same kind. If *segment* is omitted, the address expression must be a flat address. See 6.1.1 How Addresses Are Formed (p. 123).

It is not valid to use a flat load operation with an identifier. The following code is not valid:

```
ld_b64 $s1, [&g]; // not valid because address expression is a segment
                  // address, but a flat address is required.
```

If `ld_v2`, `ld_v3`, or `ld_v4` is used, then all the registers must be the same size.

Subword integer type values are extended to fill the destination register. `s` types are sign-extended, `u` types are zero-extended. Rules for this are:

- `ld_s8` — Loads a value between –128 and 127 inclusive into the destination register.
- `ld_u8` — Loads a value between 0 and 255 inclusive into the destination register.
- `ld_s16` — Loads a value between –32768 and 32767 into the destination register.
- `ld_u16` — Loads a value between 0 and 65535 inclusive into the destination register.
- `ld_s32` — Loads a value between –2,147,483,648 and 2,147,483,647 inclusive into the destination register.
- `ld_u32` — Loads a value between 0 and 4,294,967,296 inclusive into the destination register.

The `ld_s32` and `ld_u32` operations result in identical results when targeting a 32-bit register, because no sign extension or zero extension is required. They are provided to make the code more readable.

`ld_s` and `ld_u` always produce the same value when the operation size is the same as the destination size. However, `ld_s` sign-extends while `ld_u` zero-extends when the destination is larger than the destination size.

For example, `ld_u8 $d2` loads one byte and zero-extends to 64 bits.

A front-end compiler should use `ld_s` when the sign is relevant and `ld_u` when it is not. Then readers of the program can better understand the significance of what is being loaded.

In some cases, a memory system might break up a single load into fragments, treating the single `ld` operation as though it was made up of multiple separate small loads. Between each small load, other work-items can write the same addresses, and the work-items might see a mix of old and new values. A load has read atomicity of size *b* if it cannot read as fragments that are smaller than *b* bits.

In the small machine model, *b* is 32 (see 2.10 Small and Large Machine Models (p. 20)). Consider the following load:

```
ld_global_v4_f64 ($d1,$d3,$d2,$d5), [&x];
```

Implementations are allowed to break up this load into at most eight (nine if the address is unaligned) separate loads, each loading 32 bits.

In the large machine model, *b* is 64, so implementations can break the same load into at most four (five if the address is unaligned) separate loads, each loading 64 bits.

All forms of loads have 64-bit (32-bit for small machine model) naturally aligned read atomicity. This property also applies to each fragment of an unaligned load.

Ordinary loads have the following characteristics:

- They happen as a single operation within a window of fragment size.
- Loads that are larger than a fragment can be broken up into unordered loads of naturally aligned loads, each no larger than a fragment.
- Ordinary loads can be reordered by either the finalizer or hardware. Load reordering can be prevented by using some kind of synchronization operation. For example, a `ld_acq` acts like a partial fence; no memory operation after the `ld_acq` can be moved before the `ld_acq`.
- An ordinary load is allowed to satisfy an ordinary store from its own or a different work-item (before the load is visible to all agents in the system).
- An ordinary load is not allowed to fulfill an atomic operation from its own work-item (before the load is visible to all agents in the system).

Examples

```
ld_global_f32 $s1, [&x];
ld_global_s32 $s1, [&x];
ld_global_f16 $s1, [&x];
ld_global_f64 $d1, [&x];
ld_global_aligned_f64 $d1, [&x];
ld_width(64)_global_f16 $s1, [&x];
ld_width_global_aligned_f16 $s1, [&x];
ld_global_acq_f32 $s1, [&x];
ld_global_acq_f64 $d1, [&x];
ld_global_acq_equiv(2)_f32 $s1, [&x];
ld_global_acq_equiv(2)_f32 $s1, [$s3+4];
ld_arg_acq_equiv(2)_f32 $s1, [&y];
ld_private_f32 $s1, [$s3+4];
ld_spill_f32 $s1, [$s3+4];
ld_f32 $s1, [$s3+4];
ld_aligned_f32 $s1, [$s3+4];
ld_v3_s32 ($s1,$s1,$s6), [$s3+4];
ld_v4_f32 ($s1,$s1,$s6,$s2), [$s3+4];
ld_v2_equiv(9)_f32 ($s1,$s2), [$s3+4];
ld_group_equiv(0)_u32 $s0, [$s2];
ld_equiv(1)_u64 $d3, [$s4+32];
ld_v2_equiv(1)_u64 ($d1,$d2), [$s0+32];
ld_width(8)_v4_f32 ($s1,$s1,$s6,$s2), [$s3+4];
ld_equiv(1)_u64 $d6, [128];
ld_width(4)_v2_equiv(9)_f32 ($s1,$s2), [$s3+4];
ld_width(64)_u32 $s0, [$s2];
ld_width(1024)_equiv(1)_u64 $d6, [128];
ld_width(all)_equiv(1)_u64 $d6, [128];
ld_readonly_rwimg $d1, [&rwimage1];
ld_global_roimg $d2, [&roimage1];
ld_kernarg_samp $d3, [&sampler1];
```

6.3 Store (st) Operation

The store (st) operation stores a value from a register or immediate (see 4.17 Operands (p. 50)) into memory using a segment or flat address expression (see 4.18 Address Expressions (p. 52)).

There are four variants of the store operation, depending on the number of sources: one, two, three, or four.

6.3.1 Syntax

Table 6–2 Syntax for Store (st) Operation

Opcode and Modifiers	Operands
st_*segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	***src0, address***
st_v2_*segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	**(*src0*,*src1*), *address***
st_v3_*segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	**(*src0*,*src1*,*src2*), *address***
st_v4_*segment*_aligned_*sem*_equiv(*n*)_***TypeLength***	**(*src0*,*src1*,*src2*,*src3*), *address***

Explanation of Modifiers
v2, v3, and v4: Optimization flags. See the Description below.
segment: Optional segment: global, group, private, spill, or arg. If omitted, flat is used. See 2.8 Segments (p. 13).
aligned: Optional. If specified, indicates that the store operation can rely on the destination address having the natural alignment of the destination type. If omitted, the store operation must allow the destination address to be unaligned. See the Description below.
sem: Optional memory semantics flag. See the Description below.
equiv(*n*): Optional. *n* is an equivalence class. If omitted, class 0 is used. See 6.1.4 Equivalence Classes (p. 126).
Type: u, s, f. The *Type* specifies how the value is expanded to the size of the destination. See Table 4–2 (p. 46).
Length: 8, 16, 32, 64. The *Length* specifies the amount of data fetched, and the amount to increment the address when the destination is a vector operand. See Table 4–2 (p. 46).
TypeLength can also be b128, in which case *src0* must be a q register; or roimg, rwimg, or samp, in which case dest must be a d register.

Explanation of Operands
src0, *src1*, *src2*, *src3*: Sources. Must be registers or immediates.
address: Address to be stored into.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the aligned modifier has been specified.

For BRIG syntax, see 19.10.2 BRIG Syntax for Memory Operations (p. 321).

6.3.2 Description

v2, v3, and v4
: These flags are strictly optimizations. When v2, v3, or v4 is used, HSAIL will store consecutive values from multiple registers. The address is incremented by the size of the operation type between elements. Front ends should generate vector forms whenever possible.

  For example, this code:

  ```
  st_v4_u8 ($s1, $s2, $s3, $s4), [120];
  ```

  does the following:

  - Stores the lower 8 bits of `$s1` into address 120.
  - Stores the lower 8 bits of `$s2` into address 121.
  - Stores the lower 8 bits of `$s3` into address 122.
  - Stores the lower 8 bits of `$s4` into address 123.

  On certain hardware implementations, it is faster to write 64 or 128 bits in a single operation.

`aligned`
: If the `aligned` modifier is not present, then the finalizer must generate code to correctly handle unaligned memory stores.

  If the `aligned` modifier is present,then the finalizer can generate code that assumes the memory store is naturally aligned. On some implementations this might be more efficient.

  It is undefined if a memory store marked as aligned is in fact unaligned. On some implementations, this can result in incorrect values being stored, values in other memory locations being modified, and memory exceptions being generated. See 18.6 Unaligned Access (p. 255).

sem

Memory semantics flag, most commonly used for synchronization of memory operations and read/write image operations.

For read/write images, the memory semantic flag applies only to read/write image operations on the same HSA component.

Memory operations are the load, store, and atomic operations defined in this chapter. Read/write image operations are the `rdimage`, `ldimage`, `stimage`, `atomicimage`, and `atomicimagenoret` operations defined in Chapter 7 Image Operations (p. 151).

The flag can be `rel` or `part_rel`:

- `rel` creates an upward fence (basically a one-way fence operation). That is, memory and read/write image operations can be moved before the `st_rel` but no memory or read-write image operation can be moved after the `st_rel`. A `st_rel` is a synchronizing operation that will stay ordered with all other synchronizing operations.
- `part_rel` specifies that no memory or read-write image operations before this point can be moved after this point as far as this work-group can see, but other work-groups are allowed to see the memory or read-write image operations on the same HSA component in a different order.

If omitted, the store will be an ordinary (not synchronizing) store.

6.3.3 Additional Information

If *segment* is present, the address expression must be a segment address of the same kind. If *segment* is omitted, the address expression must be a flat address. See 6.1.1 How Addresses Are Formed (p. 123).

It is not valid to use a flat store operation with an identifier. The following code is not valid:

```
st_b64 $s1, [&g]; // not valid because address expression is a segment
                  // address, but a flat address is required.
```

If `st_v2`, `st_v3`, or `st_v4` is used, then all the registers must be the same size.

Subword integer values are extracted from the least significant bits of the source register. Storing a 256 with a `st_s8` writes a zero (least significant 8 bits) into memory. For other integer types, the size of the source and destination must match.

For `f32` and `f64`, the size of the source and destination must match. If a conversion is required, then it should be done explicitly using a `cvt` operation.

For `f16`, if the source is a register, it must be an `s` register. It is converted from the implementation-defined register representation to the memory representation before the store (see 4.21 Floating-Point Numbers (p. 55)).

In some cases, a memory system might break up a single store into fragments, treating the single `st` operation as though it was made up of multiple separate small stores. Between each fragment store, other work-items can read the same addresses, and the work-items might see a mix of old and new values. A store has write atomicity of size b if it cannot be broken up into fragments that are smaller than b bits.

In the small machine model, b is 32 (see 2.10 Small and Large Machine Models (p. 20)). Consider the following store:

```
st_v4_global_f64 ($d1, $d3, $d2, $d5), [&x];
```

Implementations are allowed to break up this store into at most eight (nine if the address is unaligned) separate stores, each storing 32 bits.

In the large machine model, *b* is 64, so implementations can break the same store into at most four (five if the address is unaligned) separate stores, each storing 64 bits.

All forms of stores have 64-bit (32-bit for small machine model) naturally aligned write atomicity. This property also applies to each fragment of an unaligned store.

Ordinary stores have the following characteristics:

- They happen as a single operation within a window of fragment size.
- Stores that are larger than a fragment can be broken up into unordered stores of naturally aligned stores, each no larger than a fragment.
- Ordinary stores can be reordered by either the finalizer or hardware. Store reordering can be prevented by using some kind of synchronization operation. For example, a `st_rel` acts like a partial fence; no memory operation before the `st_rel` can be moved after the `st_rel`.
- An ordinary store is allowed to satisfy an ordinary load from its own or a different work-item (before the store is visible to all agents in the system).
- An ordinary store is not allowed to fulfill an atomic operation from its own work-item (before the store is visible to all agents in the system).

Examples

```
st_global_f32 $s1, [&x];
st_global_aligned_f32 $s1, [&x];
st_global_u8 $s1, [&x];
st_global_u16 $s1, [&x];
st_global_u32 $s1, [&x];
st_global_f16 $s1, [&x];
st_global_f64 $d1, [&x];
st_global_aligned_f64 $d1, [&x];
st_global_rel_f32 $s1, [&x];
st_global_rel_f64 $d1, [&x];
st_global_rel_equiv(2)_f32 $s1, [&x];
st_rel_equiv(2)_f32 $s1, [$s3+4];
st_private_f32 $s1, [$s3+4];
st_global_f32 $s1, [$s3+4];
st_spill_f32 $s1, [$s3+4];
st_arg_f32 $s1, [$s3+4];
st_f32 $s1, [$s3+4];
st_aligned_f32 $s1, [$s3+4];
st_v4_f32 ($s1,$s1,$s6,$s2), [$s3+4];
st_v2_equiv(9)_f32 ($s1,$s2), [$s3+4];
st_v3_s32 ($s1,$s1,$s6), [$s3+4];
st_group_equiv(0)_u32 $s0, [$s2];
st_equiv(1)_u64 $d3, [$s4+32];
st_aligned_equiv(1)_u64 $d3, [$s4+32];
st_v2_equiv(1)_u64 ($d1,$d2), [$s0+32];
st_equiv(1)_u64 $d6, [128];
st_group_rwimg $d1, [&rwimage2];
st_private_rwimg $d1, [&rwimage2];
st_global_roimg $d2, [&roimage2];
st_kernarg_samp $d3, [&sampler2];
```

6.4 Atomic Operations: atomic and atomicnoret

There are two types of atomic operations, which may be implemented in a variety of ways: ordinary and synchronizing.

Synchronizing atomics act like an acquire or an acquire and release. In the acquire-only case, a synchronizing atomic acts like a downward fence. In the acquire and release case, a synchronizing atomic forms a full fence: no memory operation can be moved before or after.

In all cases, the atomic operation is executed atomically such that it is not possible for any work-item or agent in the system to observe or modify the memory location during the atomic sequence.

It is guaranteed that when a work-item issues an atomic operation on a memory address, no write to the same address from outside the current atomic operation by any work-item can occur between the atomic read and write.

If multiple atomic operations from different work-items target the same address, the operations are serialized in an undefined order.

Ordinary atomic operations have the following characteristics:

- An atomic sequence happens as a single operation. No other memory operation can change the data between the load and store of the atomics.
- Unaligned atomics are not allowed and can result in undefined behavior or generate a memory exception.
- Ordinary atomics that do not return values can be reordered by the finalizer or by hardware.

Synchronizing atomics first do an acquire to read the memory value and then, possibly, a release to store the changed value.

Atomic accesses to segments other than global and group by means of a flat address is undefined behavior.

HSAIL provides two kinds of atomic operations:

- Atomic operations, which read, modify, and write. Each atomic operation returns the value that is read before the modification.
- Atomic no return operations, which are the same as the atomic operations but do not return a value.

For both atomic and atomic no return operations, the address must be naturally aligned to a multiple of the access size. If the addresses are not naturally aligned, then the result is undefined and might generate a memory exception.

Acquire and acquire/release semantics are allowed for both atomic and atomic no return operations.

For atomic operations, the *dest* (destination) field is required. Compilers should identify cases where the result is not needed and whenever possible, they should generate the faster atomic no return operations if they do not need the result.

For more information, see 6.5 Atomic (atomic) Operations (p. 136) and 6.6 Atomic No Return (atomicnoret) Operations (p. 142).

6.5 Atomic (atomic) Operations

The atomic operations read a value from memory, set the destination to the original value, and update the memory location.

These operations atomically load the value at *address* into *dest*, perform a reduction operation with modifier data and *src0* (and, with `atomic_cas`, with *src1*), and store the result of the operation at *address*, overwriting the original value.

For atomic operations, accesses to private, spill, and arg memory are illegal.

6.5.1 Syntax

Table 6–3 Syntax for Atomic Operations

Opcode and Modifiers	Operands
atomic_and_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_or_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_xor_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_exch_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_add_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_sub_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_inc_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_dec_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_max_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_min_*segment*_*sem*_***TypeLength***	***dest, address, src0***
atomic_cas_*segment*_*sem*_***TypeLength***	***dest, address, src0, src1***

Explanation of Modifiers
segment: Optional segment: `global` or `group`. If omitted, flat is used, and *address* must be in the global or group segment. See 2.8 Segments (p. 13).
sem: Optional memory semantics flag, most commonly used for synchronization. If omitted, the load will be an ordinary (not synchronizing) load. The flag can be `acq` (acquire), `ar` (acquire and release), or `part_ar` (partial acquire and release). `acq` specifies that no memory operation (load, store, or atomic) can be moved before this point. `ar` specifies that no memory operation (load, store, or atomic) can be moved before this point. In addition, no memory operation can be moved after this point. `part_ar` specifies that no memory operation (load, store, or atomic) can be moved before this point as far as this work-group can see, but other work-groups are allowed to see the memory operations in a different order.
Type: `b` for `and`, `or`, `xor`, `exch`, `cas`; `u` and `s` for `add`, `sub`, `max`, `min`; `u` for `inc` and `dec`. See Table 4–2 (p. 46).
Length: 32, 64. See Table 4–2 (p. 46). 64 is not allowed for small machine model (see 2.10 Small and Large Machine Models (p. 20)).

Explanation of Operands
dest: Destination register.
address: Source location in the specified segment. Must be an address.
src0, *src1*: Sources. Can be a register, immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned.

For Brig syntax, see 19.10.2 BRIG Syntax for Memory Operations (p. 321).

6.5.2 Description of Atomic and Atomic No Return Operations

`and`
ANDs the contents of the *`address`* with the value in *`src0`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`or`
ORs the contents of the *`address`* with the value in *`src0`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`xor`
XORs the contents of the *`address`* with the value in *`src0`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`exch`
Replaces the contents of the *`address`* with *`src0`*. Sets *`dest`* to the original contents of the *`address`*.

Note: There is no atomic no return version of this operation.

`add`
Adds (using integer arithmetic) the contents of the *`address`* with the value in *`src0`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`sub`
Subtracts (using integer arithmetic) the contents of the *`address`* from the value in *`src0`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`min`, `max`
Compares *`scr0`* to the value in the *`address`* and sets the *`address`* to the minimum/maximum of the original value and *`src0`*. Then, for the atomic operation, sets *`dest`* to the original contents of the *`address`*.

`inc`
Increments the contents of the *`address`* using the formula:

```
[address] = ([address] >= src0) ? 0 : [address] + 1
```

After the operation, the contents of the *`address`* will be in the range [0, *`src0`*] inclusive. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

Note: Only unsigned increment is available.

`dec`
Decrements the contents of the *`address`* using the formula:

```
[address] = (([address] == 0) || ([address] > src0)) ? src0 :
     [address] - 1
```

After the operation, the contents of the *`address`* will be in the range [0, *`src0`*] inclusive. For the atomic operation, sets *`dest`* to the original contents of the *`address`*.

Note: Only unsigned decrement is available.

`cas`

Compare and swap. If the original contents of the *`address`* are equal to *`src0`*, then the contents of the location are replaced with *`src1`*. For the atomic operation, sets *`dest`* to the original contents of the *`address`*, regardless of whether the replacement was done.

Examples

```
atomic_and_global_ar_u32 $s1, [&x], 23;
atomic_and_global_u32 $s1, [&x], 23;
atomic_and_group_u32 $s1, [&x], 23;
atomic_and_u32 $s1, [$s2], 23;

atomic_or_global_ar_u64 $d1, [&x], 23;
atomic_or_global_u64 $d1, [&x], 23;
atomic_or_group_u64 $d1, [&x], 23;
atomic_or_u64 $d1, [$s4], 23;

atomic_xor_global_ar_b64 $d1, [&x], 23;
atomic_xor_global_b64 $d1, [&x], 23;
atomic_or_group_u64 $d1, [&x], 23;
atomic_or_u64 $d1, [$s3], 23;

atomic_cas_global_ar_b64 $d1, [&x], 23, 12;
atomic_cas_global_b64 $d1, [&x], 23, 1;
atomic_cas_group_u64 $d1, [&x], 23, 9;
atomic_cas_u64 $d1, [$s5], 23, 12;

atomic_exch_global_ar_b64 $d1, [&x], 23;
atomic_exch_global_b64 $d1, [&x], 23;
atomic_exch_group_u64 $d1, [&x], 23;
atomic_exch_u64 $d1, [$s4], 23;

atomic_add_global_ar_b64 $d1, [&x], 23;
atomic_add_global_b64 $d1, [&x], 23;
atomic_add_group_u64 $d1, [&x], 23;
atomic_add_u64 $d1, [$s6], 23;

atomic_sub_global_ar_b64 $d1, [&x], 23;
atomic_sub_global_b64 $d1, [&x], 23;
atomic_sub_group_u64 $d1, [&x], 23;
atomic_sub_u64 $d1, [$s3], 23;

atomic_inc_global_ar_b64 $d1, [&x], 23;
atomic_inc_global_b64 $d1, [&x], 23;
atomic_inc_group_u64 $d1, [&x], 23;
atomic_inc_u64 $d1, [$s3], 23;

atomic_dec_global_ar_b64 $d1, [&x], 23;
atomic_dec_global_b64 $d1, [&x], 23;
atomic_dec_group_u64 $d1, [&x], 23;
atomic_dec_u64 $d1, [$s4], 23;

atomic_max_global_ar_s64 $d1, [&x], 23;
atomic_max_global_b64 $d1, [&x], 23;
atomic_max_group_u64 $d1, [&x], 23;
atomic_max_u64 $d1, [$s5], 23;

atomic_min_global_ar_s64 $d1, [&x], 23;
atomic_min_global_b64 $d1, [&x], 23;
atomic_min_group_u64 $d1, [&x], 23;
```

```
atomic_and_global_ar_b32 $s1, [&x], 23;
atomic_and_global_b32 $s1, [&x], 23;
atomic_and_group_b32 $s1, [&x], 23;
atomic_and_b32 $s1, [$s2], 23;

atomic_or_global_ar_b64 $d1, [&x], 23;
atomic_or_global_b64 $d1, [&x], 23;
atomic_or_group_b64 $d1, [&x], 23;
atomic_or_b64 $d1, [$s4], 23;

atomic_xor_global_ar_b64 $d1, [&x], 23;
atomic_xor_global_b64 $d1, [&x], 23;
atomic_xor_group_b64 $d1, [&x], 23;
atomic_xor_b64 $d1, [$s3], 23;

atomic_cas_global_ar_b64 $d1, [&x], 23, 12;
atomic_cas_global_b64 $d1, [&x], 23, 1;
atomic_cas_group_b64 $d1, [&x], 23, 9;
atomic_cas_b64 $d1, [$s5], 23, 12;

atomic_exch_global_ar_b64 $d1, [&x], 23;
atomic_exch_global_b64 $d1, [&x], 23;
atomic_exch_group_b64 $d1, [&x], 23;
atomic_exch_b64 $d1, [$s4], 23;

atomic_add_global_ar_u64 $d1, [&x], 23;
atomic_add_global_s64 $d1, [&x], 23;
atomic_add_group_u64 $d1, [&x], 23;
atomic_add_s64 $d1, [$s6], 23;

atomic_sub_global_ar_u64 $d1, [&x], 23;
atomic_sub_global_s64 $d1, [&x], 23;
atomic_sub_group_u64 $d1, [&x], 23;
atomic_sub_s64 $d1, [$s3], 23;

atomic_inc_global_ar_u64 $d1, [&x], 23;
atomic_inc_global_u64 $d1, [&x], 23;
atomic_inc_group_u64 $d1, [&x], 23;
atomic_inc_u64 $d1, [$s3], 23;

atomic_dec_global_ar_u64 $d1, [&x], 23;
atomic_dec_global_u64 $d1, [&x], 23;
atomic_dec_group_u64 $d1, [&x], 23;
atomic_dec_u64 $d1, [$s4], 23;

atomic_max_global_ar_s64 $d1, [&x], 23;
atomic_max_global_u64 $d1, [&x], 23;
atomic_max_group_s64 $d1, [&x], 23;
atomic_max_u64 $d1, [$s5], 23;

atomic_min_global_ar_s64 $d1, [&x], 23;
atomic_min_global_u64 $d1, [&x], 23;
atomic_min_group_s64 $d1, [&x], 23;
```

```
atomic_min_u64 $d1, [$s7], 23;
atomic_min_u64 $d1, [$s7], 23;
```

6.6 Atomic No Return (atomicnoret) Operations

The atomic no return operations read a value from memory and update the memory location. (The atomic no return operations do not have a destination.)

The atomic no return operations atomically load the value at location *address*, perform the operation with modifier data and operand *src0* (and, with atomic_noret_cas, with *src1*), and store the result in location *address*, overwriting the original value.

6.6.1 Syntax

Table 6–4 Syntax for Atomic No Return Operations

Opcodes and Modifiers	Operands
atomicnoret_and_segment_sem_***TypeLength***	***address, src0***
atomicnoret_or_segment_sem_***TypeLength***	***address, src0***
atomicnoret_xor_segment_sem_***TypeLength***	***address, src0***
atomicnoret_add_segment_sem_***TypeLength***	***address, src0***
atomicnoret_sub_segment_sem_***TypeLength***	***address, src0***
atomicnoret_inc_segment_sem_***TypeLength***	***address, src0***
atomicnoret_dec_segment_sem_***TypeLength***	***address, src0***
atomicnoret_max_segment_sem_***TypeLength***	***address, src0***
atomicnoret_min_segment_sem_***TypeLength***	***address, src0***
atomicnoret_cas_segment_sem_***TypeLength***	***address, src0, src1***

Explanation of Modifiers
segment: Optional segment: global or group. If omitted, flat is used, and *address* must be in the global or group segment. See 2.8 Segments (p. 13).
sem: Optional memory semantics flag, most commonly used for synchronization. If omitted, the load will be an ordinary (not synchronizing) load. The flag can be acq (acquire), ar (acquire and release), or part_ar (partial acquire and release). acq specifies that no memory operation (load, store, or atomic) can be moved before this point. ar specifies that no memory operation (load, store, or atomic) can be moved before this point. In addition, no memory operation can be moved after this point. part_ar specifies that no memory operation (load, store, or atomic) can be moved before or after this point as far as this work-group can see, but other work-groups are allowed to see the memory operations in a different order.
Type: b for and, or, xor, cas; u and s for add, sub, max, min; u for inc, dec. See Table 4–2 (p. 46).
Length: 32, 64. See Table 4–2 (p. 46). 64 is not allowed for small machine model (see 2.10 Small and Large Machine Models (p. 20)).

Explanation of Operands
address: Source location in the specified segment. Must be an address.
src0, *src1*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned.

For BRIG syntax, see 19.10.2 BRIG Syntax for Memory Operations (p. 321).

6.6.2 Description

See 6.5.2 Description of Atomic and Atomic No Return Operations (p. 138).

The atomic no return operations change memory in the same way as the atomic operations but do not have a destination.

Examples

```
atomicnoret_and_global_ar_b32 [&x], 23;
atomicnoret_and_global_b32 [&x], 23;
atomicnoret_and_group_b32 [&x], 23;
atomicnoret_and_b32 [$s1], 23;

atomicnoret_or_global_ar_b64 [&x], 23;
atomicnoret_or_global_b64 [&x], 23;
atomicnoret_or_group_b64 [&x], 23;
atomicnoret_or_b64 [$s2], 23;

atomicnoret_xor_global_ar_b64 [&x], 23;
atomicnoret_xor_global_b64 [&x], 23;
atomicnoret_xor_group_b64 [&x], 23;
atomicnoret_xor_b64 [$s3], 23;

atomicnoret_cas_global_ar_b64 [&x], 23, 12;
atomicnoret_cas_global_b64 [&x], 23, 1;
atomicnoret_cas_group_u64 [&x], 23, 9;
atomicnoret_cas_u64 [$s2], 23, 12;

atomicnoret_add_global_ar_u64 [&x], 23;
atomicnoret_add_global_s64 [&x], 23;
atomicnoret_add_group_u64 [&x], 23;
atomicnoret_add_s64 [$s4], 23;

atomicnoret_sub_global_ar_u64 [&x], 23;
atomicnoret_sub_global_s64 [&x], 23;
atomicnoret_sub_group_u64 [&x], 23;
atomicnoret_sub_s64 [$s5], 23;

atomicnoret_inc_global_ar_u64 [&x], 23;
atomicnoret_inc_global_u64 [&x], 23;
atomicnoret_inc_group_u64 [&x], 23;
atomicnoret_inc_u64 [$s2], 23;

atomicnoret_dec_global_ar_u64 [&x], 23;
atomicnoret_dec_global_u64 [&x], 23;
atomicnoret_dec_group_u64 [&x], 23;
atomicnoret_dec_u64 [$s6], 23;

atomicnoret_max_global_ar_u64 [&x], 23;
atomicnoret_max_global_s64 [&x], 23;
atomicnoret_max_group_u64 [&x], 23;
atomicnoret_max_s64 [$s3], 23;

atomicnoret_min_global_ar_u64 [&x], 23;
atomicnoret_min_global_s64 [&x], 23;
atomicnoret_min_group_u64 [&x], 23;
atomic_noret_min_s64 [$s4], 23;
```

6.7 Examples of Memory Operations

6.7.1 Examples Without Synchronization

This section shows examples without synchronization. The memory operations in these examples are not ordered (provided the addresses are different).

Example 1

HSAIL uses relaxed ordering. This means that the memory system may not see the operation in sequenced-before order, using memory operations without synchronization, unless the addresses are the same.

Work-item 0	Work-item 1
`@h0: st_global_u32 1, [&a]`	`@k0: st_global_u32 1, [&b]`
`@h1: ld_global_u32 $s0, [&b]`	`@k1: ld_global_u32 $s1, [&a]`
Initially, `&a` and `&b` = 0.	
`$s0` = 0 and `$s1` = 0 is allowed.	

For each sequence, there are a number of edge constraints. First, there are the constraints added because readers have to follow writers. For example, `k1` (the reader) has to happen before `h0` changes the value.

There are also constraints caused by synchronization. If the graph with all constraints has a cycle, the output is not legal. But if there are no cycles, the result is allowed.

In this example, the code can appear to the memory system as `h1 >> k1 >> h0 >> k0`.

There is no cycle, so this is legal.

Even though `h0` appears first (in sequenced-before order) before `h1`, there is no requirement that the operations appear in text order (sequenced-before order) to the memory system.

Example 2

In HSAIL, stores need not stay in order.

Work-item 0	Work-item 1
`@h0: st_global_u32 1, [&a]`	`@k0: ld_global_u32 $s0, [&b]`
`@h1: st_global_u32 1, [&b]`	`@k1: ld_global_u32 $s1, [&a]`
Initially, `&a` and `&b` = 0.	
`$s0` = 1 and `$s1` = 0 is allowed.	

Because no operations are marked with acquire or release, the stores and loads can be reordered (in any order).

Any order where `h1 >> k0 and k1 >> h0` has the allowed result.

This example shows that work-items cannot reliably synchronize by spinning on a flag without using acquire and release.

Example 3

HSAIL does not require write-read causality.

Work-item 0	Work-item 1	Work-item 2
`@h0: st_global_u32 1, [&a]`	`@k0: ld_global_u32 $s0, [&a]`	`@j0: ld_global_u32 $s1, [&b]`
`@h1:`	`@k1: st_global_u32 1, [&b]`	`@j1: ld_global_u32 $s2, [&a]`
Initially, `&a` and `&b` = 0.		
`$s0` = 1, `$s1` = 1, and `$s2` = 0 is allowed.		

`$s0` = 1 implies `h0 >> k0`, `$s1` = 1 implies `k1 >> j0`, and `$s2` = 0 implies `j1 >> h0`.

Thus, HSAIL requires only `j1 >> h0 >> k0` and `k1 >> j0`.

Example 4

Register dependence does not force the order of memory operations.

Work-item 0	Work-item 1
`@h0: ld_global_u32 $s0, [&a]`	`@j0: st_global_u32 20, [100]`
`@h1: ld_global_u32 $s1, [$s0]`	`@j1: st_global_rel_u32 100, [&a]`
Initially, `&a` and location 100 = 0.	
`$s1` == 0 and `$s0` == 100 is allowed.	

If `$s1` == 0 then `h1 >> j0`.

If `$s0` == 100 then `j1 >> h1`.

Because of the `st_rel`, `j0 >> j1`. Thus, the order is `j0 >> j1 >> h1 >> h0`, which does not have a cycle.

Because this seems to violate dependence order, it is useful to consider how this can come about.

Work-item 0 is allowed to prefetch load `h1`. One reason it might do this is that code before these operations reads address 96, and the implementation reads in large cache lines.

Later, work-item 1 reads the new value of `&a`, which is 100. Then it reads the value of location 100, but because there is no synchronization, it can use the previously prefetched value of 0.

6.7.2 Examples Where Reusing an Address Forces Order

Example 6

Writes to the same address stay in order.

Work-item 0
`@h0: st_global_u32 1 [&a]`
`@h1: st_global_u32 2, [&a]`
`@l2: ld_global_u32 $s0, [&a]`

Initially, `&a` = 0.
HSAIL does not allow `$s0` = 1.

Example 7

Loads in a single work-item do not reorder across stores when their addresses are the same.

Work-item 0
`@h0: st_global_u32 1, [&a]`
`@l2: ld_global_u32 $s0, [&a]`

Initially, `&a` = 0.
HSAIL does not allow `$s0` = 0.

Example 8

This example shows where stores are transitively visible.

Work-item 0	Work-item 1	Work-item 2
`@h0: st_global_rel_u32 1, [&a]`	`@k0: ld_global_u32 $s0, [&a]`	`@j0: ld_global_acq_u32 $s1, [&b]`
`@h1:`	`@k1: st_global_rel_u32 1, [&b]`	`@j1: ld_global_u32 $s2, [&a]`
Initially, `&a` and `&b` = 0 .		
HSAIL does not allow `$s0` = 1 and `$s1` = 1 and `$s2` = 0.		

If `$s0` = 1 then `h0 >> k0`.

If `$s1` = 1 then `k1 >> j0`.

If `$s2` = 0 then `j1 >> h0`.

Because of the synchronization, `k0 >> k1` and `j0 >> j1`. Thus, the order is `h0 >> k0 >> k1 >> j0 >> j1 >> h0`, which has an illegal cycle.

6.7.3 Examples With One-Sided Synchronization

The examples in this section show that operations with synchronization on one direction do not force an ordering on the other direction (but force some partial ordering).

Example 9

HSAIL implementations are allowed to reorder memory operations, even when some of the operations are marked as release.

Work-item 0	Work-item 1
`@h0: st_global_rel_u32 1, [&a]`	`@k0: st_global_rel_u32 1, [&b]`
`@h1: ld_global_u32 $s0, [&b]`	`@k1: ld_global_u32 $s1, [&a]`
Initially, `&a` and `&b` = 0.	
`$s0` = 0 and `$s1` = 0 is allowed.	

The code can appear to the memory system as `h1 >> k1 >> h0 >> k0`.

In HSAIL, release does not prevent operations from moving up, but it does prevent operations from moving down.

In HSAIL, for two work-items to communicate, both the writer and the reader need to use explicit synchronization.

Example 10

Stores cannot be moved before an `ld_acq` operation.

Work-item 0	Work-item 1
`@h0: ld_global_acq_u32 $s0, [&a]`	`@k0: ld_global_acq_u32 $s1, [&b]`
`@h1: st_global_u32 1, [&b]`	`@k1: st_global_u32 1, [&a]`
Initially, `&a` and `&b` = 0.	
HSAIL does not allow `$s0` = 1 and `$s1` = 1.	

If `$s0` = 1, then `k1 >> h0`.

If `$s1` = 1, then `h1 >> k0`.

But the `ld_acq` adds edges `h0 >> h1` and `k0 >> k1`. Thus, the order must be `h1 >> k0 >> k1 >> h0 >> h1`, which has an illegal cycle.

6.7.4 Examples With Two-Sided Synchronization

In these examples, two-sided synchronization prevents reordering.

Example 11

HSAIL implementations are not allowed to reorder memory operations that are marked with acquire/release.

Work-item 0	Work-item 1
`@h0: st_global_u32 1, [&a]`	`@k0: ld_global_acq_u32 $s0, [&b]`
`@h1: st_rel_u32 1, [&b]`	`@k1: ld_global_u32 $s1, [&a]`
Initially, `&a` and `&b` = 0.	
HSAIL does not allow `$s0` = 1 and `$s1` = 0.	

Because of acquire/release, `h0 >> h1` and `k0 >> k1`.

If `$s0` = 1, then `h1 >> k0`.

But if `$s1` = 0, then `k1 >> h0`. Thus, the order is `kh0 >> h1 >> k0 >> k1 >> h0`, which is an illegal cycle and not allowed.

Example 12

In this example, two-sided synchronization prevents reordering.

Work-item 0	Work-item 1
`@h0: st_global_rel_u32 1, [&a]`	`@k0: st_global_rel_u32 1, [&b]`
`@h1: ld_global_acq_u32 $0, [&b]`	`@k1: ld_global_acq_u32 $s1, [&a]`
Initially, `&a` and `&b` = 0.	
HSAIL does not allow `$s0` = 0 and `$s1` = 0.	

Ordinary loads can move up before a `st_rel` but not an `ld_acq`.

If `$s0 == 0`, then `h1 >> k0`.

If `$s1 == 0`, then `k1 >> h0`.

Because of synchronization, `k0 >> k1` and `h0 >> h1`. Thus, there is a cycle `h1 >> k0 >> k1 >> h0 >> h1`.

Example 13

In this example, stores are seen in consistent order.

Work-item 0	Work-item 1	Work-item 2	Work-item 3
`@h0: st_global_rel_u32 1, [&a]`	`@k0: st_global_rel_u32 1, [&b]`	`@j0: ld_global_acq_u32 $s0, [&a]`	`@m0: ld_global_acq_u32 $s2, [&b]`
		`@j1: ld_global_acq_u32 $s1, [&b]`	`@m1: ld_global_acq_u32 $s3, [&a]`

Initially, `&a` and `&b` = 0.
HSAIL does not allow `$s0` and `$s1` = 0, `$s2` = 1, and `$s3` = 0.

If `$s0` == 1, then `h0 >> j0` .

If `$s1` == 0, then `j1 >> k0`.

If `$s2` = 1, then `k0 >> m0`.

If `$s3` = 0, then `m1 >> h0`.

Because of synchronization, `j0 >> j1` and `m0 >> m1`. Thus, the order is `j0>> j1 >> k0 >> m0 >> m1 >> h0 >> j0`, which has an illegal cycle.

Example 14

In this example, synchronized atomics are seen in consistent order.

Work-item 0	Work-item 1	Work-item 2	Work-item 3
`@h0: atomic_exch_global_ar_u32 $s4, [&a], 1`	`@k0: atomic_exch_global_ar_u32 $s5, [&b], 1`	`@j0: ld_global_acq_u32 $s0, [&a]`	`@m0: ld_global_acq_u32 $s2, [&b]`
		`@j1: ld_global_acq_u32 $s1, [&b]`	`@m1: ld_global_acq_u32 $s3, [&a]`

Initially, `&a` and `&b` = 0.
HSAIL does not allow `$s0` = `$s1` = 0, `$s2` = 1 and `$s3` = 0.

This example has the same analysis as Example 13.

Chapter 7

Image Operations

This chapter describes how image and sampler objects are used in HSAIL and also describes the read image, load image, store image, and atomic image operations.

Note: For background information, see the OpenCL™ Specification: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

7.1 Images in HSAIL

7.1.1 What Are Images?

Images are a graphics feature that can sometimes be useful in data-parallel computing. Image memory is a special kind of memory access that can make use of dedicated hardware often provided for graphics. Many implementations will provide such dedicated hardware to speed up image operations.

Reasons to use an image include:

- Implementations can have special caches that are optimized for 2D accesses.
- Implementations can have dedicated hardware that can be used for specialized operations. One such operation is filtering, which is a way to determine a value for a coordinate from the values in the image that are near the coordinate. Certain filtering modes (like linear) form averages around the coordinate. Mathematically, this tends to smooth out the values or "filter" out high-frequency changes.
- Implementations can have special out-of-bounds hardware support.
- Images can be addressed in one, two, or three dimensions using integer or normalized coordinates. Coordinates can be either integers or floating-point values in the 0.0 to 1.0 range.
- Image memory offers different addressing modes, as well as data filtering, for some specific image formats.
- Images have many special compression modes that can save bandwidth.

While images are frequently used to hold visual data, an HSAIL program can use an image to hold any kind of data.

In all HSAIL implementations, the use of images provides a collection of capabilities that extend the simple CPU memory view.

A 1D image and a linear array differ because implementations are allowed to insert gaps in images whenever that helps performance. Implementations are allowed to reorder the memory locations of 2D and 3D images in any order.

Image implementations can create caching hints using read-only images.

Images can be used to optimize write operations by delaying them until the next kernel execution.

7.1.2 How Images Are Described

An image is an array of one, two, or three dimensions.

Each element in an image has four values called components. The components are named `rgba`.

Each element in the image has the same format.

Each image has a fixed size for each dimension.

Associated with each image are two objects:

- An *image object* that describes how the image is structured.
- A *sampler object* that describes how a particular read is to be performed.

HSAIL can create an image object, but it cannot fill in the image data. The application creates the image and then binds it to the name of the image object by using a runtime library call.

Conceptually, inside an image object is a reference to the image data.

Image objects in HSAIL are opaque: the only access to the image data referenced by the image object is through HSAIL image operations, not through `ld`, `st`, `atomic`, or `atomicnoret` operations.

Image objects can be passed as kernel and function arguments and can be copied between memory and registers using `ld`, `st`, and `mov` operations. Note that these operations are copying the image object that references the image data, not the image data. The memory address of an image object can be taken using the `lda` operation, but again this is the address of the image object, not the image data.

An image object can be defined as either of the following:

- Read-only (with type `roimg`)
- Read-write (with type `rwimg`)

Image data memory that is referenced by an `roimg` object is read-only within the kernel dispatch that uses it. The image data can, however, still be written by the host CPU. Within a different kernel dispatch, the same image data can be referenced by an `rwimg` object, and within that kernel dispatch the image data is read-write. However, it is undefined if, during the execution of a specific kernel dispatch, the image data referenced by an `roimg` object within that kernel dispatch is written during the execution of the kernel. This writing includes through aliased paths, both by regular memory operations and by image operations using `rwimg` image objects, and by any agent.

For more information, see 7.1.4 Image Objects (p. 154).

Just like image objects, sampler objects are opaque. They can be passed as kernel and function arguments and can be copied between memory and registers using `ld`, `st`, and `mov` operations. The memory address of a sample object can be taken using the `lda` operation. HSAIL provides operations to access properties of samplers.

Sampler objects have type `samp`.

For more information, see 7.1.7 Sampler Objects (p. 160).

7.1.3 Image Geometry

Each image has an associated geometry.

There are six kinds of image geometry:

- 1D (one-dimensional image)
- 2D (two-dimensional image)
- 3D (three-dimensional image)
- 1DA (one-dimensional image array)
- 2DA (two-dimensional image array)
- 1DB (one-dimensional image buffer)

1D
: A 1D image contains data that can be addressed with a single coordinate: width. The coordinate can be either of the following:
  - An unnormalized integer value greater than or equal to 0 and less than the width of the image
  - A normalized floating-point value between 0 and 1

2D
: A 2D image is addressed by two coordinates: width and height. The coordinates can be one of the following:
  - A pair of unnormalized integer values greater than or equal to 0 and less than the width (height) of the image
  - A pair of normalized floating-point values between 0 and 1

3D
: A 3D image is addressed by three coordinates: width, height, and depth. The coordinates can be one of the following:
  - A triple of unnormalized integer values greater than or equal to 0 and less than the height or depth of the image
  - A triple of normalized floating-point values between 0 and 1

1DA

A 1DA image is an array of a homogeneous collection of one-dimensional images, all with the same size and format.

Each element in the 1DA is addressable with two coordinates:

- The first selects the coordinate within the image.
- The second indexes the underlying 1D image.

If the coordinates are floating-point, some rules apply to the second coordinate:

- The second coordinate must be unnormalized.
- The second coordinate is rounded to the nearest even integer.
- If the second coordinate is out of bounds, it is clamped so that the result is greater than or equal to 0 and less than or equal to the array size –1.

The most important difference between 1DA images and 2D images is that samplers never combine values across images.

2DA

A 2DA image is an array of two-dimensional images, addressed by three coordinates:

- The first two are the coordinates of an element in the image.
- The third selects the image.

The images must all have the same size and format.

1DB

A 1DB image buffer is a specialized kind of one-dimensional image with the following restrictions:

- The coordinates must be unnormalized, not normalized.
- Samplers cannot be used.

Note: Graphic systems frequently support many additional image formats, cubemaps, three-dimensional arrays, and so forth. HSAIL has just enough graphics to support common programming languages like OpenCL. The BRIG enumeration for geometry includes additional geometry values that can be used by extensions. See 19.2.9 BrigImageGeometry (p. 264).

7.1.4 Image Objects

An image object describes how an image is structured. It contains pieces of information that hold the properties of an image:

- Width in elements.
- Height in elements. Must be 1 if the image is 1D or 1DA.
- Depth in elements. Must be 1 if the image is 1D, 2D, 1DA, or 2DA.
- Array size. Must be 1 if the image is not a 1DA or a 2DA.

- Image format: Specifies how the data is stored in memory. See Table 7–1 (p. 156).
- Image order: Specifies the presence and order of components in memory. See Table 7–2 (p. 156).

An image object also contains a reference to the actual image data.

Image objects can be defined in three places:

- As a global or readonly variable outside of a function or kernel. They are not allowed inside a function or kernel, because image objects require external linkage so the agent can initialize the image data.
- As a type in an arg definition
- As a type in a kernarg definition

An image object always has a size of 8 bytes and a natural alignment of 8 bytes.

The format of an image object is device-specific.

The HSA runtime provides operations for an application program to create image objects including creating and associating image data with an image object for a specific HSA component. The runtime ensures that the correct device-specific image object format is used when it is:

- Passed as the argument to a kernel dispatch
- A global or readonly segment variable referenced in an address expression

It is undefined to use an image object in any HSA component other than the HSA component that originally accessed it from a kernel argument or global or readonly segment variable. This is because different HSA components might have different representations for an image object.

An image object declared as a global or readonly variable can have its properties defined by providing an initializer that is a list containing pairs of keyword = values.

For example, the following defines 12 read-only objects and 14 read-write objects:

```
global_roimg &name0 = {width = 5, height = 4, depth = 6,
             format = unorm_short_101010, order = rgbx};
extern global_roimg &name1;
extern global_roimg &ArrayOfroimgs[10];
extern global_rwimg &namedrwimg12;
extern global_rwimg &namedrwimg2;
extern global_rwimg &namedrwimg3;
extern global_rwimg &ArrayOfrwimgs[10];
global_rwimg &namedrwimgWithInit = {width = 5, height = 4, depth = 6,
             format = unorm_short_101010, order = rgbx};
```

Each of the properties is called a `TOKEN_PROPERTY` in Extended Backus-Naur Form.

The `query` operations can be used to query the attributes of an image. See 7.7 Query Image and Query Sampler Operations (p. 173).

See the tables below. For information about how to read the tables, see 7.1.5 How Images Are Accessed (p. 157) and 7.1.6 Bits Per Pixel (bpp) (p. 158).

Table 7–1 Enumeration for Image Format Properties

Enumeration Name	No. of Components	Normalized?	Signed?	Bits	Type
snorm_int8	four	yes	yes	8	integer
snorm_int16	four	yes	yes	16	integer
unorm_int8	four	yes	no	8	integer
unorm_int16	four	yes	no	16	integer
unorm_int24	one	yes	no	r=bits[24:00]	integer
unorm_short_565	three	yes	no	r=bits[15:11]	integer
				g=bits[10:05]	
				b=bits[04:00]	
unorm_short_555	three	yes	no	r=bits[14:10]	integer
				g=bits[09:05]	
				b=bits[04:00]	
				bit 15 ignored	
unorm_int_101010	three	yes	no	31:30 ignored	integer
				r=bits[29:20]	
				g=bits[19:10]	
				b=bits[09:00]	
signed_int8	four	no	yes	8	integer
signed_int16	four	no	yes	16	integer
signed_int32	four	no	yes	32	integer
unsigned_int8	four	no	no	8	integer
unsigned_int16	four	no	no	16	integer
unsigned_int32	four	no	no	32	integer
half_float	four	-	-	16	floating-point
float	four	-	-	32	floating-point

Table 7–2 Enumeration for Image Order Properties

Enumeration Name	Components	Border	Components' Image Format
r	(r,0f,0f,1f)	0f,0f,0f,1f	
rx	(r,0f,0f,1f)	0f,0f,0f,0f	
a	(pf,0f,0f,a)	0f,0f,0f,a	
rg	(r,g,0f,1f)	0f,0f,0f,1f	
rgx	(r,g,0f,1f)	0f,0f,0f,0f	
ra	(r,0f,0f,a)	0f,0f,0f,0f	
rgb	(r,g,b,1f)	0f,0f,0f,1f	`unorm_short_565`
			`unorm_short_555`
			`unorm_int_101010`
srgb, srgbx	(r,g,b,1f)	0f,0f,0f,0f	`unorm_int8`
rgbx	(r,g,b,1f)	0f,0f,0f,0f	
rgba	(r,g,b,a)	0f,0f,0f,0f	

Enumeration Name	Components	Border	Components' Image Format
bgra	(b,g,r,a)	0f,0f,0f,0f	unorm_int8
			snorm_int8
			signed_int8
			unsigned_int8
argb	(r,g,b,a)	0f,0f,0f,0f	unorm_int8
			snorm_int8
			signed_int8
			unsigned_int8
srgba	(r,g,b,a)	0f,0f,0f,0f	unorm_int8
sbgra	(b,g,r,a)	0f,0f,0f,0f	unorm_int8
intensity	(i,i,i,i)	0f,0f,0f,0f	unorm_int8
			unorm_int16
			snorm_int8
			snorm_int16
			half_float
			float
luminance	(1,1,1,1f)	0f,0f,0f,1f	unorm_int8
			unorm_int16
			snorm_int8
			snorm_int16
			half_float
			float

7.1.5 How Images Are Accessed

To access the data in an image, a program loads an image object into a `d` register using a load (`ld`) operation with a source type of `roimg` or `rwimg`. This does not load the image data; instead, it loads an opaque handle that can be used to access the image data. It then uses this register as the source of the read image (`rdimage`, load image (`ldimage`), store image (`stimage`), and atomic image (`atomicimage` and `atomicimagenoret`) operations.

The differences between the `rdimage` operation and the `ldimage` operation are:

- `rdimage` takes a sampler object and therefore supports additional modes.
- The value returned for out-of-bounds references for `rdimage` depends on the sampler object; `ldimage` always returns 0.

The sampler object is provided to the image operations in the same way as the image object: it is loaded into a `d` register.

Both an image and sampler object in a `d` register can be moved to another `d` register using the move (`mov`) operation, and stored to another variable of the same type using the store (`st`) operation. This allows them to be passed by value into a function, and returned by value from a function.

It is undefined if the d register used in an image operation was not loaded with a value that ultimately originated from a global, readonly, or kernal argument variable. For images, the type of image in the original variable (roimg or rwimg) must match in all operations that use the value. For samplers, the original variable and all operations that use the value must specify the sampler type (samp). These operations include load (ld), store (st), move (mov), the image operations (rdimage, ldimage, stimage, atomicimage, and atomicimagenoret), and the image and sampler query operations (see 7.7 Query Image and Query Sampler Operations (p. 173)).

The address of an image or sampler object can be taken using the lda operation. This allows them to be passed by reference. It is undefined if the address returned is used by a load or store operation that does not specify the same type as the original image or sampler object.

Each image has an image format that indicates how the data is stored in memory. For example, unorm_short_555 specifies that data is stored as three 5-bit fields in a 16-bit (2-byte) field of memory. See Table 7–1 (p. 156).

Each image has an image order, which describes the memory layout. See Table 7–2 (p. 156).

Reads from an image retrieve four values (rgba). Stores to an image expect four values. However, to save space, the data in memory are allowed to have fewer than four components and are allowed to have the components in a different order.

For example, the image order r specifies that only the r component is stored in memory. On a read, the gba components get default values. On a store, only the first component is written into memory.

To access data from an image, the operations require coordinates. Coordinates can be:

- Float-normalized (meaning in the range 0.0 to 1.0 as mapped to the image size)
- Float-unnormalized (0 to the size of the coordinate dimension)
- Integer (0 to the size of the coordinate dimension)

When a rdimage operation accesses an image using normalized float coordinates, it is possible that the value used in the operation does not exactly map to a coordinate value of the image. In this case, the resulting unnormalized coordinate accessed will be near the location of the normalized value. A sampler specifies how the coordinate is chosen. (An example is the use of a weighted average of the values around the coordinate.)

Normalized coordinates are not restricted to the 0 to 1.0 range. For values outside of this range, assorted transformations, again determined by the sampler, can be applied. (For example, the coordinates could be clamped to the edge, or the coordinates could be wrapped.) The rdimage operation results in four values, computed from the image based on the sampler.

7.1.6 Bits Per Pixel (bpp)

Associated with each combination of image format and image order there is a number called the bits per pixel (bpp).

The bpp value is the number of bits needed to hold one element of an image. The bpp value is the size of a component (from the format) times the number of components (from the order of components).

For example, the image order `r` has one component per element if the element is in `half_float` image format (16-bit). The `bpp` value is 1*16 = 16 bits. If the image format is `float` (32-bit), then the `bpp` is 1*32 = 32 bits.

The image order `bgra` has four components per element. If the image format is `unorm_int8`, then the `bpp` is 4*8 = 32 bits.

The image order `ra` has two components per element. If the image format is `float`, then the `bpp` is 2*32 = 64 bits.

Table 7–3 Supported Image Orders and Image Formats

	unorm	snorm	uint	sint	float	bpp
intensity, luminance						
8	Y	Y	Y	Y	-	8
16	Y	Y	Y	Y	Y	16
32	-	-	-	-	Y	32
r						
8	Y	Y	Y	Y	-	8
16	Y	Y	Y	Y	Y	16
24	Y	-	-	-	-	24
32	-	-	Y	Y	Y	32
rx, a						
8	Y	Y	Y	Y	-	8
16	Y	Y	Y	Y	Y	16
32	-	-	Y	Y	Y	32
rg, rgx, ra						
16_16	Y	Y	Y	Y	Y	32
32_32	-	-	Y	Y	Y	64
rgb, rgbx						
5_6_5	Y	-	-	-	-	16
5_5_5	Y	-	-	-	-	16
rgba, bgra, argb						
8_8_8	Y	Y	Y	Y	-	32
16_16_16_16	Y	Y	Y	Y	Y	64
32_32_32	-	-	Y	Y	Y	96
32_32_32_32	-	-	Y	Y	Y	128
srgb, srgbx, srgba, sbgra						
8_8_8_8	Y	-	-	-	-	32

In the table:

- Y means supported.
- - means not supported.

7.1.7 Sampler Objects

Sampler objects are used to specify how to process reads (`rdimage` operations) for images.

Sampler objects can be defined in three places:

- As a global or readonly variable outside of a function or kernel. They are not allowed inside a function or kernel, because sampler objects require external linkage so the agent can initialize them.
- As a type in an arg definition
- As a type in a kernarg definition

Properties of a sampler object are:

- `coord`: `normalized` (coordinates are in range [0.0 to 1.0]) or `unnormalized` (coordinates are integers).
- `filter`: `nearest` or `linear`. Linear filtering cannot be used with read-write images.
- `boundaryU`: boundary mode for first component: `wrap`, `clamp`, `mirror`, `mirroronce`, or `border`.
- `boundaryV`: boundary mode for second component: `wrap`, `clamp`, `mirror`, `mirroronce`, or `border`.
- `boundaryW`: boundary mode for third component: `wrap`, `clamp`, `mirror`, `mirroronce`, or `border`.

`mirror` makes the image look as though it was infinitely big; no coordinate returns the border color.

`mirroronce` makes the image look twice as big, where the additional half of the new image mirrors the original.

Coordinates outside of the doubled image return the border color.

See 7.1.8 Rules to Process Coordinates (p. 161) and 7.1.9 Image Boundary Modes (p. 161).

A sampler object always has a size of 8 bytes and a natural alignment of 8 bytes.

The format of a sampler object is device-specific.

The HSA runtime provides operations for an application program to create sampler objects to pass as arguments to kernels for a specific HSA component. The runtime ensures that the correct device-specific sampler object format is used when it is:

- Passed as the argument to a kernel dispatch
- A global or readonly segment variable referenced in an address expression

It is undefined to use a sampler object in any HSA component other than the HSA component that originally accessed it from a kernel argument or global or readonly segment variable. This is because different HSA components might have different representations for a sampler object.

A sampler object declared as a global or readonly variable can have its properties defined by providing an initializer that is a list containing pairs of keyword = values.

An example of a sampler object is:

```
global_samp &y1 = {coord = normalized, filter = nearest, boundaryU = clamp,
                   boundaryV = clamp, boundaryW = clamp};
```

Implementations of HSAIL are expected to provide efficient ways to organize images to take advantage of 2D spatial locality. Because the three-coordinate address is available, implementations can reorder or pad the image data in arbitrary ways.

7.1.8 Rules to Process Coordinates

If the value of `u`, `v`, or `w` is INF or NaN, the result is undefined.

`coord = normalized` specifies that all coordinates are in the range [0.0 to 1.0].

`coord = unnormalized` specifies that all cordinates are in the range 0 to dimension -1

`filter = nearest` specifies that the image element selected is the nearest (in Manhattan distance) to the specified (`u`, `v`, `w`) coordinates.

`filter = linear` selects a line block of two elements (for 1D images), a 2x2 square block of elements (for 2D images), or a 2x2x2 cube block of elements (for 3D images) around the input coordinate, and combines the selected values using linear interpolation. The result is formed as the weighted average of the values in each element in the block. The weights are the fractional distance from the element center to the coordinate.

Certain boundary modes are connected with formats and normalized coordinates:

- The array indexes (1DA and 2DA) are always treated as clamped.
- `wrap`, `mirror`, and `mirroronce` require normalized coordinates.

7.1.9 Image Boundary Modes

`boundaryU`, `boundaryV`, and `boundaryW` determine what happens when the coordinate (`u`, `v`, `w`) is out of range. Once the coordinate has been converted to an unnormalized value, it is possible that the coordinate value is outside of the image. In that case, either the coordinate gets mapped back into range by the boundary mode for each component, or a special value called the *border color* is used.

Each component of the coordinate is processed separately. If the coordinate `x` is outside of the image (that is, either `x < 0` or `x > dim-1`, where `dim` is the image dimension for this component), then the coordinate is transformed.

The code below shows how the boundary mode affects the transformation. It follows these steps:

1. The coordinate is converted to an integer if the coordinate is normalized and nearest filtering is used: `x = floor (u * dim)`.
2. If the coordinate is normalized and linear filtering is used: `x = floor (u* dim - 0.5)`.
3. If the coordinate is not normalized and nearest filtering is used: `x = floor(u)`.
4. If the coordinate is not normalized and linear filtering is used: `x = floor(u - 0.5)`.

```
if ((x < 0) || (x > dim - 1)) {
    switch (boundarymode)
    {
    case wrap:
       x = x mod dim;

       if (x < 0) { x += dim; }

       break;
    case mirror:
        {
            if(x < 0){x = -x - 1;}
            bool flip = (x/dim) & 1;
            x &= dim;
            if (flip) {x = dim - x - 1;}
            break;
        }
    case clamp:
        x = max( 0, min( x, dim - 1 ) );
        break;
    case mirroronce:
        if (x < 0) {x = -x - 1;}
        x = max( 0, min( x, dim - 1 ) );
        break;
    case border:
                // special case: instead of fetching from the image,
                // use the border color
        break;
    }
```

If boundary mode in the sampler is `clamp`, then out-of-range image coordinates return the border color. The border color selected depends on the image channel order. See the following table:

Table 7–4 Image Channel Order and Border Color

Image Channel Order	Border Color
A, INTENSITY, Rx, RA, RGx, RGBx, ARGB, BGRA, RGBA, sRGB, sRGBx, sRGBA, or sBGRA	(0.0f, 0.0f, 0.0f, 0.0f)
R, RG, RGB, or LUMINANCE	(0.0f, 0.0f, 0.0f, 1.0f)

If the image order has less than four components, `rdimage` and `ldimage` first figure out the return values, ignoring the missing components, and then return a four-element result, filling in the missing values from the Components column in Table 7–2 (p. 156).

7.1.10 Image Formats and Output Types

HSAIL requires that implementations support the image formats in the table below for each output type.

Table 7–5 Image Formats and Output Types

Image Format	f32 Output Type	u32 Output Type	s32 Output Type
snorm_int8	[–1.0 to 1.0]	*	*
snorm_int16	[–1.0 to 1.0]	*	*
unorm_int8	[0.0 to 1.0]	*	*
unorm_int16	[0.0 to 1.0]	*	*
unorm_int24	[0.0 to 1.0]	*	*
unorm_short_565	[0.0 to 1.0]	*	*
unorm_short_555	[0.0 to 1.0]	*	*
unorm_short_101010	[0.0 to 1.0]	*	*
signed_int8	*	u32	*
signed_int16	*	u32	*
signed_int32	*	u32	*
unsigned_int8	*	*	u32
unsigned_int16	*	*	u32
unsigned_int32	*	*	u32
half_float	float	*	*
float	float	*	*

In the table, * means "undefined."

7.2 Read Image (rdimage) Operation

The read image (`rdimage`) operation performs an image memory lookup using an image coordinate vector.

7.2.1 Syntax

Table 7–6 Syntax for Read Image Operation

Opcode and Modifiers	Operands
rdimage_v4_1d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *sampler*, *coordWidth***
rdimage_v4_2d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *sampler*, (*coordWidth*, *coordHeight*)**
rdimage_v4_3d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *sampler*, (*coordWidth*, *coordHeight*, *coordDepth*)**
rdimage_v4_1da_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *sampler*, (*coordWidth*, *coordArrayIndex*)**
rdimage_v4_2da_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *sampler*, (*coordWidth*, *coordHeight*, *coordArrayIndex*)**

Explanation of Modifiers
`v4`: Specifies the number of elements returned by the operation. Always 4.
`1d`, `2d`, `3d`, `1da`, `2da`: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be `1d` (width), `2d` (width and height), `3d` (width, height, and depth), `1da` (height and array index), or `2da` (width, height and array index). `1db` is not supported. See 7.1.3 Image Geometry (p. 153).
`destType`: Destination type: `u32`, `s32`, or `f32`. See Table 4–2 (p. 46).
`imageType`: Image object type: `roimg`, `rwimg`. See Table 4–4 (p. 47).
`coordType`: Source coordinate element type: `u32`, `f32`. See Table 4–2 (p. 46).

Explanation of Operands
`destR`, *`destG`*, *`destB`*, *`destA`*: Destination. Must be an `s` register.
`image`: A source operand `d` register that contains a value of an image object of type `imageType`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type `imageType`, or from an arg segment variable that is of type `imageType` that was initialized with a value that is of type `imageType`.
`sampler`: A source operand `d` register that contains a value of a sampler object. It is always of type `samp`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type `samp`, or from an arg segment variable that is of type `samp` that was initialized with a value that is of type `samp`.
`coordWidth`, *`coordHeight`*, *`coordDepth`*, *`coordArrayIndex`*: A source `s` register of type *`coordType`* that specifies the coordinates being read.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.2.2 Description

The read image (`rdimage`) operation performs an image memory lookup using an image coordinate vector. The operation loads data from a read-write or read-only image, specified by source operand *`image`* at coordinates given by source operands *`coordWidth`*, *`coordHeight`*, *`coordDepth`*, and *`coordArrayIndex`*, into destination operands *`destR`*, *`destG`*, *`destB`*, and *`destA`*. A sampler specified by source operand *`sampler`* defines how to process the read.

`rdimage` used with integer coordinates has restrictions on the sampler:

- `coord` must be `unnormalized`.
- `filter` must be `nearest`.
- The boundary mode must be `clamp` or `border`.

`rdimage` used with read-write images has restrictions on the sampler:

- `filter` must be `nearest`.

Examples

```
ld_global_rwimg $d1, [%rwimg1];
ld_kernarg_roimg $d2, [%roimg2];
ld_readonly_samp $d3, [%samp1];
rdimage_v4_1d_s32_rwimg_f32 ($s0, $s1, $s5, $s3), $d1, $d3, $s6;
rdimage_v4_1da_s32_rwimg_f32 ($s0, $s1, $s2, $s3), $d1, $d3,
   ($s6, $s7);
rdimage_v4_2da_s32_rwimg_f32 ($s0, $s1, $s3, $s4), $d1, $d3,
   ($s6, $s9, $s12);
rdimage_v4_2d_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d2, $d3,
   ($s6, $s9);
rdimage_v4_3d_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d2, $d3,
   ($s6, $s9, $s2);
```

7.3 Load Image (ldimage) Operation

The load image (`ldimage`) operation loads from image memory using an image coordinate vector.

7.3.1 Syntax

Table 7–7 Syntax for Load Image Operation

Opcode and Modifiers	Operands
ldimage_v4_1d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *coordWidth***
ldimage_v4_2d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, (*coordWidth*, *coordHeight*)**
ldimage_v4_3d_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, (*coordWidth*, *coordHeight*, *coordDepth*)**
ldimage_v4_1da_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, (*coordWidth*, *coordArrayIndex*)**
ldimage_v4_2da_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, (*coordWidth*, *coordHeight*, *coordArrayIndex*)**
ldimage_v4_1db_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *coordByteIndex***

Explanation of Modifiers
`v4`: Specifies the number of elements returned by the operation. Always 4.
`1d`, `2d`, `3d`, `1da`, `2da`, `1db`: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be `1d` (width), `2d` (width and height), `3d` (width, height, and depth), `1da` (height and array index), `2da` (width, height and array index), or `1db` (byte index). See 7.1.3 Image Geometry (p. 153).
`destType`: Destination type: `u32`, `s32`, or `f32`. See Table 4–2 (p. 46).
`imageType`: Image object type: `roimg`, `rwimg`. See Table 4–4 (p. 47).
`coordType`: Source coordinate element type: `u32`. See Table 4–2 (p. 46).

Explanation of Operands
`destR`, *`destG`*, *`destB`*, *`destA`*: Destination. Must be an `s` register.
`image`: A source operand `d` register that contains a value of an image object of type `imageType`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type `imageType`, or from an arg segment variable that is of type `imageType` that was initialized with a value that is of type `imageType`.
`coordWidth`, *`coordHeight`*, *`coordDepth`*, *`coordArrayIndex`*: A source `s` register of type *`coordType`* that specifies the coordinates being read.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.3.2 Description

The load image (`ldimage`) operation loads from image memory using an image coordinate vector. The operation loads data from a read-write or read-only image, specified by source operand *`image`* at integer coordinates given by source operands *`coordWidth`*, *`coordHeight`*, *`coordDepth`*, and *`coordArrayIndex`*, into destination operands *`destR`*, *`destG`*, *`destB`*, and *`destA`*.

While `ldimage` does not have a sampler, it works as though there is a sampler with `coord = unnormalized` and `filter = nearest`.

If a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0), no load is performed and 0 is returned.

The differences between the `ldimage` operation and the `rdimage` operation are:

- `rdimage` takes a sampler and therefore supports additional modes.
- The value returned if a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0) for `rdimage` depends on the sampler; `ldimage` always returns 0.

For the `1db` geometry, coordinates are in bytes. For all other geometries, coordinates are in elements.

Examples

```
ld_global_rwimg $d1, [%rwimg1];
ld_kernarg_roimg $d2, [%roimg2];
ldimage_v4_3d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1,
   ($s4, $s5, $s6);
ldimage_v4_1da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1,
   ($s4, $s5);
ldimage_v4_1db_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2, $s4;
ldimage_v4_2da_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2,
   ($s4, $s1, $s2);
```

7.4 Store Image (stimage) Operation

The store image (`stimage`) operation stores to image memory using an image coordinate vector.

7.4.1 Syntax

Table 7–8 Syntax for Store Image Operation

Opcode and Modifiers	Operands
stimage_v4_1d_*srcType_imageType_coordType*	**(*srcR*, *srcG*, *srcB*, *srcA*), *image*, *coordWidth***
stimage_v4_2d_*srcType_imageType_coordType*	**(*srcR*, *srcG*, *srcB*, *srcA*), *image*, (*coordWidth*, *coordHeight*)**
stimage_v4_3d_*srcType_imageType_coordType*	**(*srcR*, *srcG*, *srcB*, *srcA*), *image*, (*coordWidth*, *coordHeight*, *coordDepth*)**
stimage_v4_1da_*srcType_imageType_coordType*	**(*srcR*, *srcG*, *srcB*, *srcA*), *image*, (*coordWidth*, *coordArrayIndex*)**
stimage_v4_2da_*srcType_imageType_coordType*	**(*srcR*, *srcG*, *srcB*, *srcA*), *image*, (*coordWidth*, *coordHeight*, *coordArrayIndex*)**
stimage_v4_1db_*destType_imageType_coordType*	**(*destR*, *destG*, *destB*, *destA*), *image*, *coordByteIndex***

Explanation of Modifiers
`v4`: Specifies the number of elements returned by the operation. Always 4.
`1d`, `2d`, `3d`, `1da`, `2da`, `1db`: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be `1d` (width), `2d` (width and height), `3d` (width, height, and depth), `1da` (height and array index), `2da` (width, height and array index), or `1db` (byte index). See 7.1.3 Image Geometry (p. 153).
`srcType`: Source type: `u32`, `s32`, or `f32`. See Table 4–2 (p. 46).
`imageType`: Image object type: `rwimg`. See Table 4–4 (p. 47).
`coordType`: Source coordinate element type: `u32`. See Table 4–2 (p. 46).

Explanation of Operands
`srcR`, *`srcG`*, *`srcB`*, *`srcA`*: Source data. Must be an `s` register.
`image`: A source operand `d` register that contains a value of an image object of type `imageType`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type `imageType`, or from an arg segment variable that is of type `imageType` that was initialized with a value that is of type `imageType`.
`coordWidth`, *`coordHeight`*, *`coordDepth`*, *`coordArrayIndex`*: A source `s` register of type *`coordType`* that specifies the coordinates being read.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.4.2 Description

The store image (`stimage`) operation stores to image memory using an image coordinate vector. The operation stores data specified by source operands *`srcR`*, *`srcG`*, *`srcB`*, and *`srcA`* to a read-write image specified by source operand *`image`* at integer coordinates given by source operands *`coordWidth`*, *`coordHeight`*, *`coordDepth`*, *`coordArrayIndex`*, and *`coordByteIndex`*.

If a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0), no store is performed.

The source elements are interpreted left-to-right as `r`, `g`, `b`, and `a` components of the image format. These elements are written to the corresponding components of the image element. Source elements that do not occur in the image element are ignored.

For example, an image format of `r` has only one component in each element, so only source operand *`srcR`* is stored.

For the `1db` geometry, coordinates are in bytes. For all other geometries, coordinates are in elements.

Type conversions are performed as needed between the source data type specified by *`srcType`* (`s32`, `u32`, or `f32`) and the destination image data element type and format.

Examples

```
ld_global_rwimg $d1, [%rwimg1];
stimage_v4_3d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1,
   ($s4, $s5, $s6);
stimage_v4_2da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1,
   ($s4, $s5, $s6);
stimage_v4_1da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1,
   ($s4, $s5);
stimage_v4_1db_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, $s4;
```

7.5 Atomic Image (atomicimage) Operations

The atomic image (`atomicimage`) operations perform an atomic operation on image memory using an image coordinate vector. They do both a read and a write to the same coordinates which is done atomically.

7.5.1 Syntax

Table 7–9 Syntax for Atomic Image Operations

Opcode and Modifiers	Operands
atomicimage_and_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_or_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_xor_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_exch_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_add_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_sub_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_inc_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_dec_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_min_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_max_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0***
atomicimage_cas_geom_destType_imageType_coordType	***dest*, *image*, *coord*, *src0*, *src1***

Explanation of Modifiers
geom: Image geometry: `1d`, `2d`, `3d`, `1da`, `2da`, `1db`. Specifies the number and meaning of coordinates required to access an image element. Can be `1d` (width), `2d` (width and height), `3d` (width, height, and depth), `1da` (height and array index), `2da` (width, height and array index), or `1db` (byte index). See 7.1.3 Image Geometry (p. 153).
destType:Type of formatted image data element: `u32`, `s32`, `b32`, `u64`, `s64`, `b64`, depending on the type of operation. Length of 64 is not allowed for small machine model (see 2.10 Small and Large Machine Models (p. 20). `add`, `sub`, `min`, and `max` apply to `u32`, `s32`, `u64`, and `s64` types. `inc` and `dec` apply to `u32` and `u64` types. `and`, `or`, `xor`, `exch`, and `cas` apply to `b32` and `b64` types. See Table 4–2 (p. 46).
imageType: Image object type: `roimg`, `rwimg`. See Table 4–4 (p. 47).
coordType: Source coordinate element type: `u32`. See Table 4–2 (p. 46).

Explanation of Operands
dest: Destination, the original value of the formatted image data element being updated. Must be an `s` register for 32-bit types and a `d` register for 64-bit types.
image: A source operand `d` register that contains a value of an image object of type `imageType`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type *imageType*, or from an arg segment variable that is of type *imageType* that was initialized with a value that is of type *imageType*.
coord: Source that specifies the coordinates being atomically updated, as either a single operand or a vector operand of two or three elements. Each coordinate element must be an `s` register of type *coordType*. The number of coordinate elements is specified by the *geom* modifier: single operand for 1D images, a two-element vector operand for 2D images, and a three-element vector operand for 3D images. The meaning of each coordinate element is the same as for the `stimage` operation.
src0: The formatted value of type *destType* used to combine with the formatted image data element specified by *coord*. Can be a register, immediate value, or `WAVESIZE`.
src1: Used only for `atomicimage_cas` (compare and swap). If the contents of the *image* are equal to *src0*, then the contents of the location are replaced with *src1*. Their type is *destType*. Can be a register, immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.5.2 Description

The `atomicimage` operations perform an atomic operation on 32-bpp or 64-bpp (bits per pixel) formatted image data element. The *destType* is used to determine the bits per pixel size of the formatted image data element. It must match the image format.

All coordinates of atomic image operations must be integer values.

If a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0), then an atomic image operation has no effect and returns 0.

All images used in atomic image operations must be read-write (type `rwimg`).

For atomic image operations that operate on 32-bit types, the bits per pixel (bpp) of the image must be 32. For 64-bit types, the bits per pixel must be 64.

The atomic operation performed is the same as for the regular atomic (`atomic`) operations described in 6.5 Atomic (atomic) Operations (p. 136), except the operation is performed on the formatted image data element specified by the coordinates.

The atomic image operations are the same as the atomic no return image (`atomicimagenoret`) operations (see 7.6 Atomic Image No Return (atomicimagenoret) Operations (p. 171)) except they return the contents of the memory location before the operation.

Examples

```
ld_global_rwimg $d1, [%rwimg1];
ld_global_rwimg $d2, [%rwimg2];
atomicimage_and_3d_b32_rwimg_u32 $s1, $d1, ($s0, $s3, $s1), $s1;
atomicimage_and_2d_b32_rwimg_u32 $s2, $d1, ($s0, $s3), $s2;
atomicimage_and_1d_b32_rwimg_u32 $s3, $d1, $s1, $s10;
atomicimage_or_1d_b32_rwimg_u32 $s3, $d1, $s1, $s3;
atomicimage_xor_1db_b32_rwimg_u32 $s0, $d1, ($s1, $s2), $s3;
atomicimage_add_3d_s32_rwimg_u32 $s4, $d1, ($s0, $s3, $s1), $s2;
atomicimage_sub_2d_s64_rwimg_u32 $d3, $d2, ($s0, $s3), $d4;
atomicimage_min_1d_u64_rwimg_u32 $d3, $d2, $s1, $d4;
atomicimage_max_1da_u64_rwimg_u32 $d3, $d2, ($s1, $s2), $d4;
atomicimage_exch_2da_b64_rwimg_u32 $d3, $d2, ($s1, $s2, $s3), $d4;
atomicimage_cas_1d_b32_rwimg_u32 $s10, $d1, $s1, $s3, $s4;
```

7.6 Atomic Image No Return (atomicimagenoret) Operations

The `atomicimagenoret` operations perform an atomic operation on 32-bpp or 64-bpp (bits per pixel) data.

7.6.1 Syntax

Table 7–10 Syntax for Atomic Image No Return Operations

Opcode and Modifiers	Operands
atomicimagenoret_and_geom_destType_imageType_coordType_	*image, coord, src0*
atomicimagenoret_or_geom_destType_imageType_coordType_	*image, coord, src0*
atomicimagenoret_xor_geom_destType_imageType_coordType_	*image, coord, src0*
atomicimagenoret_add_geom_destType_imageType_coordType_	*image, coord, src0*
atomicimagenoret_sub_geom_destType_imageType_coordType_	*image, coord, src0*

Opcode and Modifiers	Operands
atomicimagenoret_inc_geom_destType_imageType_coordType_	***image, coord, src0***
atomicimagenoret_dec_geom_destType_imageType_coordType_	***image, coord, src0***
atomicimagenoret_min_geom_destType_imageType_coordType_	***image, coord, src0***
atomicimagenoret_max_geom_destType_imageType_coordType_	***image, coord, src0***
atomicimagenoret_cas_geom_destType_imageType_coordType_	***image, coord, src0, src1***

Explanation of Modifiers
geom: Image geometry: `1d`, `2d`, `3d`, `1da`, `2da`, `1db`. Specifies the number and meaning of coordinates required to access an image element. Can be `1d` (width), `2d` (width and height), `3d` (width, height, and depth), `1da` (height and array index), `2da` (width, height and array index), or `1db` (byte index). See 7.1.3 Image Geometry (p. 153).
destType:Type of formatted image data element: `u32`, `s32`, `b32`, `u64`, `s64`, `b64`, depending on the type of operation. Length of 64 is not allowed for small machine model (see 2.10 Small and Large Machine Models (p. 20). `add`, `sub`, `min`, and `max` apply to `u32`, `s32`, `u64`, and `s64` types. `inc` and `dec` apply to `u32` and `u64` types. `and`, `or`, `xor`, `exch`, and `cas` apply to `b32` and `b64` types. See Table 4–2 (p. 46).
imageType: Image object type: `roimg`, `rwimg`. See Table 4–4 (p. 47).
coordType: Source coordinate element type: `u32`. See Table 4–2 (p. 46).

Explanation of Operands
image: A source operand `d` register that contains a value of an image object of type `imageType`. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type `imageType`, or from an arg segment variable that is of type `imageType` that was initialized with a value that is of type *imageType*.
coord: Source that specifies the coordinates being atomically updated, as either a single operand or a vector operand of two or three elements. Each coordinate element must be an `s` register of type *coordType*. The number of coordinate elements is specified by the *geom* modifier: single operand for 1D images, a two-element vector operand for 2D images, and a three-element vector operand for 3D images. The meaning of each coordinate element is the same as for the `stimage` operation.
src0: The formatted value of type *destType* used to combine with the formatted image data element specified by *coord*. Can be a register, immediate value, or `WAVESIZE`.
src1: Used only for `atomicimagenoret_cas` (compare and swap). If the contents of the *image* are equal to *src0*, then the contents of the location are replaced with *src1*. Their type is *destType*. Can be a register, immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.6.2 Description

The atomic image no return (`atomicimagenoret`) operations are the same as the atomic image (`atomicimage`) operations except they do not return the original value of the image data element being updated. They therefore do not support the `exch` operation.

For more information, see 7.5 Atomic Image (atomicimage) Operations (p. 169).

Examples

```
ld_global_rwimg $d1, [%rwimg1];
ld_global_rwimg $d2, [%rwimg2];
atomicimagenoret_and_3d_b32_rwimg_u32 $d1, ($s0, $s3, $s1), $s1;
atomicimagenoret_and_2d_b32_rwimg_u32 $d1, ($s0, $s3), $s2;
atomicimagenoret_and_1d_b32_rwimg_u32 $d1, $s1, $s10;
atomicimagenoret_or_1d_b32_rwimg_u32 $d1, $s1, $s3;
atomicimagenoret_xor_1db_b32_rwimg_u32 $d1, ($s1, $s2), $s3;
atomicimagenoret_add_3d_s32_rwimg_u32 $d1, ($s0, $s3, $s1), $s2;
atomicimagenoret_sub_2d_s64_rwimg_u32 $d2, ($s0, $s3), $d4;
atomicimagenoret_min_1d_u64_rwimg_u32 $d2, $s1, $d4;
atomicimagenoret_max_1da_u64_rwimg_u32 $d2, ($s1, $s2), $d4;
atomicimagenoret_cas_1d_b32_rwimg_u32 $d1, $s1, $s3, $s4;
```

7.7 Query Image and Query Sampler Operations

The query image and query sampler operations query an attribute of an image object or a sampler object.

7.7.1 Syntax

Table 7–11 Syntax for Query Image and Query Sampler Operations

Opcode	Operands
queryimagewidth_*destType*_*imageType*	***dest, image***
queryimageheight_*destType*_*imageType*	***dest, image***
queryimagedepth_*destType*_*imageType*	***dest, image***
queryimagearray_*destType*_*imageType*	***dest, image***
queryimageorder_*destType*_*imageType*	***dest, image***
queryimageformat_*destType*_*imageType*	***dest, image***
querysamplercoord_*destType*_samp	***dest, sampler***
querysamplerfilter_*destType*_samp	***dest, sampler***

Explanation of Modifier
destType: Destination type: u32. See Table 4–2 (p. 46).
imageType: Image object type: roimg, rwimg. See Table 4–4 (p. 47).
samp: Sampler object type. See Table 4–4 (p. 47).

Explanation of Operands
dest: Destination register or type u32.
image: A source operand d register that contains a value of an image object of type *imageType*. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type *imageType*, or from an arg segment variable that is of type *imageType* that was initialized with a value that is of type *imageType*.
sampler: A source operand d register that contains a value of a sampler object. It is always of type samp. It is undefined if the value was not originally loaded from a global, readonly, or kernarg segment variable of type samp, or from an arg segment variable that is of type samp that was initialized with a value that is of type samp.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.3 BRIG Syntax for Image Operations (p. 322).

7.7.2 Description

Each query returns a 32-bit value giving a property of the source:

Query	Returns
width	Image width in elements.
height	Image height in elements. Must be 1 if the image is 1D or 1DA.
depth	Image depth in elements. Must be 1 if the image is 1D, 2D, 1DA, or 2DA.
array	The number of image data sets in a 1DA or 2DA image, or 1 for image types that are not arrays.
order	An image order property (see Table 7–2 (p. 156)) encoded as an integer according to 19.2.10 BrigImageOrder (p. 264).
format	An image format property (see Table 7–1 (p. 156)) encoded as an integer according to 19.2.8 BrigImageFormat (p. 263).
coord	A sampler coordinate property encoded as an integer according to 19.2.21 BrigSamplerCoord (p. 272).
filter	A sampler filter property encoded as an integer according to 19.2.22 BrigSamplerFilter (p. 272).

Examples

```
ld_global_rwimg $d1, [%rwimg1];
ld_kernarg_roimg $d2, [%roimg2];
ld_readonly_samp $d3, [%samp1];
queryimagewidth_u32_rwimg $s1, $d1;
queryimageheight_u32_rwimg $s0, $d1;
queryimagedepth_u32_rwimg $s0, $d1;
queryimagearray_u32_roimg $s1, $d2;
queryimageorder_u32_roimg $s0, $d2;
queryimageformat_u32_roimg $s0, $d2;
querysamplercoord_u32_samp $s0, $d3;
querysamplerfilter_u32_samp $s0, $d3;
```

Chapter 8

Branch Operations

This chapter describes the direct and indirect branch operations.

8.1 Branches in HSAIL

Like many programming languages, HSAIL supports branch operations that can alter the control flow.

The branch operations are:

- `brn` — Unconditional branch
- `cbr` — Conditional branch

There are two forms of each branch operation: direct and indirect.

8.1.1 Direct Branches

The direct branch operations work this way:

- `brn` (the unconditional branch) transfers control to a label.
- `cbr` (the conditional branch) checks a source and transfers control to either the label (if the source is true), or the statement after the `cbr` (if the source is false).

An unconditional direct branch could be written in pseudocode as:

```
goto k1;
// some code
:k1;
```

and might be translated into HSAIL as:

```
brn @k1;
// some code
@k1:
```

A conditional direct branch could be written in pseudocode as:

```
if (condition)
   {
   // then statements
   }
else
   {
   // else statements
   }
```

and might be translated into HSAIL as:

```
// compute the condition into $c0
cbr $c0, @k1;
// code for the else statements
brn @join;
@k1:
// code the then statements
@join:
```

Because HSAIL allows implementations to execute code in wavefronts with size greater than 1, branches can sometimes introduce performance issues.

For example, a single `cbr` operation might transfer control to the label for work-items where the source is true and to the operation after the `cbr` for work-items where the source is false. In this case, the wavefront is said to diverge, and the code is inside divergent control flow. See 2.13 Divergent Control Flow (p. 21).

In divergent control flow, an implementation is allowed to execute all the work-items where the condition has evaluated to be true, with the other work-items waiting, followed by execution of the work-items where the condition was evaluated to be false.

In the above example, the time to execute it would be the sum of the time it takes to execute the `if` block plus the time it takes to execute the `else` block, if the `cbr` diverged. If the `cbr` does not diverge, then the time to execute the example would only be the time it takes for the non-divergent path to execute. That is, either the `if` block or the `else` block but not both.

8.1.2 Indirect Branches

Indirect branch operations can have multiple targets.

All work-items take the branch, but because the target is in a register, they can go to different targets.

You can use the optional argument *`possible-targets`* to list the possible targets. (Some implementations might generate more efficient code if the list of possible targets is known when the code is finalized.)

`possible-targets` can be any of the following:

- Omitted.

  If *`possible-targets`* is omitted, the branch can target (jump to) any label in the current function that appears in an `ldc` operation (see 5.8 Copy (Move) Operations (p. 83)) or in the initializer of a global or readonly variable definition.

  Implementations might produce slower code when this option is used.

  ```
  // method 1 - any label that appeared in an ldc

  ldc_u32 $s1, @label;
  brn $s1;
  cbr $c1, $s1;
  //...
  @label:
  // ...
  ```

- A label (in square brackets) of a `labeltargets` statement.

In this case, the branch can target (jump to) any label in the list of targets in the `labeltargets` statement.

The behavior is undefined if the value in the target register is not in the list of targets in the `labeltargets` statement.

A programmer might use this method if a lot of conditional indirect branches go to the same places.

```
// method 2 - using label targets

@tab: labeltargets @a1, @a2;
ld_global_u32 $s1, [&x][$s0];
brn_width(4) $s1, [@tab];
brn $s1, [@tab];

cbr_width(4) $c1, $s1, [@tab];
cbr $c1, $s1, [@tab];

@a1:
// ...
@a2:
// ...
```

- An identifier (in square brackets) of a `u32` array with a label initializer.

  In this case, the branch can target (jump to) any label in the initializer list.

  The behavior is undefined if the value in the target register is not in the initializer list.

```
// method 3 - using an array of targets
global_u32 %x[] = {@a3, @a4};
// ...
brn_width(all) $s1, [%x];
// ...
brn $s1, [%x];

// each work-item uses its own value of $c1 to determine if
// it branches and its own value of $s1 to determine where to go

cbr_width(all) $c1, $s1, [%x];
// ...
cbr $c1, $s1, [%x];

@a1:
// ...
@a2:
// ...
@a3:
// ...
@a4:
// ...
```

Because a register might hold different values in different work-items, even a `brn` operation can diverge.

8.2 Direct and Indirect Branch Operations

8.2.1 Syntax

Table 8–1 Syntax for Unconditional Direct Branch Operation

Opcode and Modifier	Operands
brn	***direct-branch-target***

Table 8–2 Syntax for Conditional Direct Branch Operation

Opcode and Modifier	Operands
cbr_*width*	***src0, direct-branch-target***

Table 8–3 Syntax for Unconditional Indirect Branch Operation

Opcode and Modifier	Operands
brn_*width*	***src0***, *possible-targets*

Table 8–4 Syntax for Conditional Indirect Branch Operation

Opcode and Modifier	Operands
cbr_*width*	***src0, src1***, *possible-targets*

Explanation of Modifiers
width: Optional: `width(`*n*`)`, `width(WAVESIZE)`, or `width(all)`. For the direct form of `brn`, `width(all)` is the only possible value, so there is no need for the width modifier. For `cbr` and the indirect form of `brn`, the width modifier specifies the number of consecutive work-items in flattened ID order that are guaranteed to branch to the same address. If this modifier is omitted, each work-item in the work-group is allowed to branch independently (in other words, `width(1)` is the default). For example, when `width(all)` is specified, the value in *src1* must be uniform over each work-group (the same in all work-items in the work-group). For more information, see 8.3 Using the Width Modifier (p. 182).

Explanation of Operands
src0: For brn, *src0* is the address of the destination. Must be an s or d register, depending on the memory model, that holds the address of a label. See Table 2–3 (p. 20). For cbr, *src0* determines if the branch is taken. Must be a control (c) register.
direct-branch-target: Target of the direct branch. Must be a label.
src1: Target of the conditional indirect branch. Must be an s or d register, depending on the memory model, that holds the address of a label. See Table 2–3 (p. 20).
possible-targets: Optional. Possible targets of the indirect branch. Either a label in square brackets of a labeltargets statement or an identifier (in square brackets) of a u32 array with a label initializer. See 8.4 Label Targets (labeltargets Statement) (p. 183).

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.4 BRIG Syntax for Branch Operations (p. 322).

8.2.2 Description

If the source *src0* of the cbr is true (non-zero), work-items will branch to the target; otherwise the remaining work-items will fall through and execute the operation after the branch.

If the target is a label, the flow can only go to the label or the fall-through.

If the target is in a register, there might be multiple targets.

See also 8.4 Label Targets (labeltargets Statement) (p. 183) and 8.3 Using the Width Modifier (p. 182).

Examples

```
cbr_width(2) $c0, @label;
cbr_width(all) $c0, @label;
cbr $c0, @label;

brn @label2;

cbr $c1, $s1;
cbr_width(2) $c1, $s1;
cbr_width(all) $c1, $s1;

global_u32 %jumptable[3] = {@label, @label2, @label3};
global_u32 %jumptable2[3] = {@label, @label2, @label3};

cbr $c1, $s1, [@targets];
cbr_width(2) $c1, $s1, [%jumptable];
cbr_width(all) $c1, $s1, [@targets];

@targets: labeltargets @label1, @label2, @label3;
brn $s1, [@targets];
brn $s1, [%jumptable2];
// ...
@label1:
// ...
@label2:
// ...
@label3:
// ...
```

8.3 Using the Width Modifier

Sometimes a finalizer can generate more efficient code if it knows details about how divergent a branch might be.

Sometimes it is possible to know that a subset of the work-items will transfer to the same target, even when all the work-items will not. HSAIL uses the width modifier to provide this information.

If the application knows that n (n being a power of 2) consecutive work-items in flattened ID order will transfer to the same target, this can be specified as `width(`*n*`)`.

For example, see the following pseudocode (part of a reduction):

```
for (unsigned int s = 512; s>=64; s>>=1)
{
   int id = workitemid(0);
   if (id < s) {
      sdata[id] += sdata[id + s];
   }
   barrier;
}
```

`s` will have the values 512, 256, 128, 64, and consecutive work-items in groups of 64 will always go the same way.

For best performance, the `if` should be coded as `cbr_width(64)`.

`width(all)` indicates that all work-items in the work-group will transfer to the same location. If a developer knows or a compiler determined that the condition in the example above was independent of the work-item ID, then a possibly more efficient way to code the example would be to use the `width(all)` modifier:

```
cbr_width(all) $c0, @l1;
// ...
@l1:
```

This specifies that either all work-items will go to `@l1` or none of them will.

`width(WAVESIZE)` can be used to indicate that all work-items in the implementation-defined wavefront size will transfer to the same location. This requires that the kernel algorithm has been explicitly written to use `WAVESIZE` appropriately. This in turn may require that the kernel is dispatched using values dependent on the wavefront size. For example, the algorithm may require that the work-group size and dynamic group memory allocation be a function of the wavefront size. The wavefront size for a particular finalized kernel can be obtained by a runtime query. Using `width(WAVESIZE)` may allow the finalizer to optimize.

The width modifier does not cause the finalizer to group work-items into wavefronts in a different way (the assignment of work-items to wavefronts is fixed). The width modifier allows a finalizer to determine if every work-item in a wavefront will do the operation in the same way.

See also 2.13.1 Width Modifier (p. 22).

8.4 Label Targets (labeltargets Statement)

When a register is the target of a branch operation, the finalizer can often generate better code if it knows the possible targets. The `labeltargets` statement can be used to supply this information to the finalizer.

The `labeltargets` statement is used only to provide a label list. It does not allocate any storage.

The syntax is:

`label``: labeltargets` *`ListofLabels`*

For example:

```
@targets: labeltargets @l1, @l2, @l3;
// ...
cbr $c1, $s0, [@targets];
// ...
@l1:
// ...
@l2:
// ...
@l3:
// ...
```

The `labeltargets` statement can only appear inside a kernel or function.

It is not legal to transfer control out of the kernel or function.

Every target in the *ListofLabels* must be inside the same kernel or function.

If a `labeltargets` statement is supplied, it is undefined to jump to a label not listed in the `labeltargets` statement.

If a `labeltargets` statement is not supplied, it is legal to jump to any label defined in the current kernel or function that has appeared as the source of an `ldc` operation or in the initializer of a global or readonly variable.

The *label* in a `labeltargets` statement is not a legal target for a jump, and cannot appear in an `ldc` operation or in an initializer list.

For more information, see 8.2 Direct and Indirect Branch Operations (p. 180).

Chapter 9

Parallel Synchronization and Communication Operations

This chapter describes operations used for cross work-item communication.

9.1 Memory Fence Modifier

Some parallel synchronization operations have a memory fence modifier that specifies which memory segments the operation forces to become synchronized.

There are two memory segments that can be synchronized with a memory fence: the group segment and the global segment.

Implementations cannot move memory operations to the synchronized memory segment over a memory fence operation. A memory fence operation is always a two-way fence. It waits for all prior loads and stores to the synchronized memory segment to complete. Synchronizing a memory segment ensures that the work-item executing the memory fence operation will not proceed until all previous values it has stored to that memory segment have become visible to other work-items and agents, and that all previous values it has loaded from that memory segment have completed. It also ensures that any memory store operations after the memory fence operation will not become visible to other work-items or agents unless those being made visible by the fence operation are visible, and that any memory load operations after the memory fence operation will see the values made visible by memory fence operations that have previously been executed by other work-items or agents.

The group segment is only synchronized with other work-items in the same work-group. It does not synchronize with group segment memory operations of work-items in other work-groups, which operate on a different group segment.

The global segment can be synchronized with all work-items and agents, or be synchronized just with the work-items in the same work-group (termed a partial memory fence). A global segment fence includes both memory operations to the global segment (`ld`, `st`, `atomic`, and `atomicnoret`), and operations to read-write images (`ldimage`, `rdimage`, `stimage`, `atomicimage`, and `atomicimagenoret`).

A partial global segment memory fence is not guaranteed to make global segment memory or read-write image stores by the work-item visible to work-items in other work-groups or other agents. The global segment memory values stored by work-items in other work-groups or other agents are not guaranteed to become visible to the work-items of this work-group, even if the other work-items or agents execute a global memory fence operation. Because changes to global memory must eventually happen, limiting synchronization to the work-group does not give an implementation permission to delete a global memory store operation even if it can determine that no work-item in the work-group will read the changed location.

A partial global segment memory fence is appropriate when work-items will write to global segment memory or read-write images, and other work-items will read back those values, but all communication will only happen between work-items in the same work-group. A partial global segment memory fence might be more efficient on some implementations than a regular global segment fence.

For example, the amount of data the work-items within a work-group are exchanging might be too large to fit into group memory. In this case, they could use the global segment, and partial global segment memory fences, because the data is only being shared by work-items in the same work-group. In some implementations (for example, ones that share an L1 cache over a work-group), the use of a partial memory fence might allow an implementation to reduce memory traffic and so would be more efficient than a regular global segment memory fence.

The valid values for the memory fence are:

- `fnone` — specifies that no memory fence is performed. Some operations do not allow this value.
- `fgroup` — specifies that all prior group segment memory operations by the work-item must complete. Global segment or read-write image operations are not guaranteed to have completed.
- `fglobal` — specifies that all prior global segment operations and read-write image operations by the work-item must complete. Group segment operations are not guaranteed to have completed.
- `fboth` — specifies that group segment operations by the work-item, and global segment and read-write image operations by the work-item, must complete.
- `fpartial` — specifies that all prior global segment operations and read-write image operations by the work-item must complete with respect to other work-items in the work-group. Global segment and read-write image operations by work-items in other work-groups and other agents and group segment operations are not guaranteed to have completed.
- `fpartialboth` — specifies that group segment operations, global segment operations, and read-write image operations by the work-item must complete with respect to other work-items in the work-group. Global segment, or read-only image operations by work-items in other work-groups, or by other agents, are not guaranteed to have completed.

If an operation has an optional memory fence modifier, it specifies the value used if it is omitted.

9.2 barrier Operation

The `barrier` operation is used to synchronize work-item execution in a work-group and optionally act as a memory fence.

9.2.1 Syntax

Table 9–1 Syntax for barrier Operation

Opcode and Modifiers
barrier_*width*_*fence*

Explanation of Modifiers
width: Optional. Must be `width(`*n*`)` or `width(WAVESIZE)`. Used to indicate the communication pattern among work-items. Specifies the number of consecutive work-items in flattened ID order that can communicate, guaranteeing that there is no communication between work-items outside the consecutive work-items. If this modifier is omitted, any work-item in the work-group can communicate with any other. In other words, the default is equivalent to `width(all)`. `width(all)` cannot be specified, because it would be the same as the default. See 2.13.1 Width Modifier (p. 22).
fence: Optional. The memory segment that the `barrier` operation forces to become synchronized. Either `fnone`, `fgroup`, `fglobal`, `fboth`, `fpartial`, or `fpartialboth`. If omitted, `fboth` is used. See 9.1 Memory Fence Modifier (p. 185).

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.5 BRIG Syntax for Parallel Synchronization and Communication Operations (p. 323).

9.2.2 Description

The `barrier` operation is used to synchronize work-items in a work-group such that all work-items must have reached this point before any one of them can go farther.

If the `barrier` operation is used inside of divergent control flow, the result is undefined. (See 2.13 Divergent Control Flow (p. 21).) The underlying threading model is undefined by the specification, so some architectures might reach deadlock in the presence of divergent barriers while others might not correctly synchronize.

A `barrier` operation can be used in a loop provided the loop introduces no divergent control flow. This requires that all work-items in the work-group execute the loop the same number of iterations.

The `barrier` operation also acts as a memory fence. No work-items in a work-group can proceed past the `barrier` until prior stores have become visible. The memory fence modifier (*fence*) is used to specify which stores, and to which work-items and agents the stores must be visible. See 9.1 Memory Fence Modifier (p. 185). Because a barrier ensures that all work-items of a work-group are synchronized, this also ensures that all work-items of the work-group perform the memory fence.

See also 9.3 Fine-Grain Barrier (fbar) Operations (p. 188).

If an implementation has a wavefront size that is greater than or equal to *n*, the implementation is free to remove the `barrier`. However, even if the `barrier` is removed, memory operations cannot be moved over the barrier location.

Examples

```
barrier;
barrier_width(64);
barrier_fgroup;
barrier_fglobal;
```

9.3 Fine-Grain Barrier (fbar) Operations

9.3.1 Overview: What Is an Fbarrier?

In certain situations, barrier synchronization (which is synchronization over all work-items in a work-group) is too coarse. Applications might find it convenient to synchronize at a finer level, over a subset of the work-items within the work-group. A fine-grain barrier object called an *fbarrier* is needed for this subset. The work-items in the subset are said to be members of the fbarrier.

An fbarrier is defined using the `fbarrier` statement which can appear either in compilation unit scope or in kernel or function scope. For example:

```
fbarrier &fb;
```

Fbarriers are used to synchronize only between work-items within a work-group that are wavefront uniform. As such, an fbarrier has work-group persistence (see 2.9.1 Persistence Rules (p. 19)): it has the same allocation, persistence, and addressability rules as a group segment variable. The naming and visibility of an fbarrier follows the same rules as variables.

An fbarrier is an opaque entity and its size and representation are implementation-defined. It is also implementation-defined in which kind of memory fbarriers are allocated. For example, an fbarrier can use dedicated hardware, or can use memory in the group or global segments. An implementation is allowed to limit the number of fbarriers it supports, but must support a minimum of 32 per work-group. The total number of fbarriers supported by a compute unit might limit the number of work-groups that can be executed simultaneously. An implementation can use group segment memory to implement fbarriers, which will reduce the amount of group segment memory available to group segment variables. If a kernel uses more fbarriers than an HSA component supports, then an error must be reported by the finalizer.

An fbarrier conceptually contains three fields:

- Unsigned integer `member_count` — the number of wavefronts in the work-group that are members of the fbarrier.
- Unsigned integer `arrive_count` — the number of wavefronts in the work-group that are either currently waiting on the fbarrier or have arrived at the fbarrier.
- SetOfWavefrontId `wait_set` — the set of wavefronts currently waiting on the fbarrier.

An fbarrier is an opaque object and can only be accessed using fbarrier operations. An implementation is free to implement the semantics implied by these conceptual fields in any way it chooses, and is not restricted to having these exact fields.

The fbarrier operations are described below. They can refer to the fbarrier they operate on by the identifier of the fbarrier statement.

The address of an fbarrier can be taken with the `ldf` operation. This returns a `u32` value in a register that can also be used by fbarrier operations to specify which fbarrier to operate on.

9.3.2 Syntax

Table 9–2 Syntax for fbar Operations

Opcodes	Operands
`initfbar`	***src***
`joinfbar`*`_width`*	***src***
`waitfbar`*`_width_fence`*	***src***
`arrivefbar`*`_width_fence`*	***src***
`leavefbar`*`_width`*	***src***
`releasefbar`	***src***
`ldf_u32`	***dest, fbarrierName***

Explanation of Modifier
`width`: Optional: `width(`*`n`*`)`, `width(WAVESIZE)`, or `width(all)`. If *n* is specified, it must be a multiple of `WAVESIZE`. If the width modifier is omitted, it defaults to `width(WAVESIZE)`. See 2.13.1 Width Modifier (p. 22)).
`fence`: Optional. The memory segment that the fbarrier operation forces to become synchronized. Either `fnone`, `fgroup`, `fglobal`, `fboth`, `fpartial`, or `fpartialboth`. If omitted, defaults to `fboth`. See 9.1 Memory Fence Modifier (p. 185)).

Explanation of Operands
`src`: Either the name of an fbarrier, or an s register containing a value produced by an `ldf` operation. If a register, its compound type is `u32`.
`fbarrierName`: Name of the fbarrier on which to operate.
`dest`: An s register.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.5 BRIG Syntax for Parallel Synchronization and Communication Operations (p. 323).

9.3.3 Description

`initfbar`
Before an fbarrier can be used by any work-item in the work-group, it must be initialized.

The *src* operand specifies the fbarrier to initialize.

`initfbar` conceptually sets the `member_count` and `arrive_count` to 0, and the `wait_set` to empty. On some implementations, this operation might perform allocation of additional resources associated with the fbarrier.

An fbarrier must not be initialized if it is already initialized. This implies only one work-item of the work-group must perform the `initfbar` operation at a time.

An fbarrier must be initialized because a finalizer cannot know the full set of fbarriers used by a work-group in the presence of dynamic group memory allocation.

There must not be a race condition between the work-item that executes the `initfbar` and any other work-items in the work-group that execute fbarrier operations on the same fbarrier. This requirement can be satisfied by using the `barrier` operation, or the `waitfbar`operation (on another fbarrier) between the `initfbar` and the fbarrier operations that use it.

Once an fbarrier has been initialized, its memory cannot be modified by any operation except fbarrier operations until it is released by an `releasefbar` operation.

`joinfbar`
Causes the work-item to become a member of the fbarrier.

The *src* operand specifies the fbarrier to join.

This operation (which includes the value of the *src* operand) must be wavefront uniform (see 2.14 Uniform Operations (p. 23)). This implies that all active work-items of a wavefront must be members of the same fbarriers.

`joinfbar` conceptually atomically increments the `member_count` for the wavefront.

A work-item must not join an fbarrier that has not been initialized, nor join an fbarrier of which it is already a member.

`waitfbar`

Indicates that the work-item has arrived at the fbarrier, and causes execution of the work-item to wait until all other work-items of the same work-group that are members of the same fbarrier have arrived at the fbarrier.

The *src* operand specifies the fbarrier on which to wait.

An `waitfbar` operation can also optionally perform a memory fence to ensure that any data being communicated becomes visible. The memory fence is performed after all work-items that are members of the fbarrier have arrived at the fbarrier and before the work-items waiting at the fbarrier proceed with execution. See 9.1 Memory Fence Modifier (p. 185).

This operation (which includes the value of the *src* operand) must be wavefront uniform (see 2.14 Uniform Operations (p. 23)). This implies that all active work-items of a wavefront arrive at an `waitfbar` together.

`waitfbar` conceptually atomically increments the `arrive_count` for the wavefront, and adds the wavefront to the `wait_set`. It then atomically checks and waits until the `arrive_count` equals the `member_count`, at which point any wavefronts in the `wait_set` are allowed to proceed, the `arrive_count` is reset to 0, and the `wait_set` reset to empty.

A work-item must not wait on an fbarrier that has not been initialized, nor wait on an fbarrier of which it is not a member.

`arrivefbar`

Indicates that the work-item has arrived at the fbarrier, but does not wait for other work-items that are members of the fbarrier to arrive at the same fbarrier. If the work-item is the last of the fbarrier members to arrive, then any work-items waiting on the fbarrier can proceed and the fbarrier is reset.

The *src* operand specifies the fbarrier on which to arrive.

An `arrivefbar` operation can also optionally perform a memory fence before proceeding to ensure that any data being communicated becomes visible. See 9.1 Memory Fence Modifier (p. 185).

This operation (which includes the value of the *src* operand) must be wavefront uniform (see 2.14 Uniform Operations (p. 23)). This implies that all active work-items of a wavefront arrive at an `arrivefbar` together.

`arrivefbar` conceptually atomically increments the `arrive_count` for the wavefront, and checks if the `arrive_count` equals the `member_count`. If it does, then atomically any wavefronts in the `wait_set` are allowed to proceed, the `arrive_count` is reset to 0, and the `wait_set` is reset to empty.

A work-item must not arrive at an fbarrier that has not been initialized, nor arrive at an fbarrier of which it is not a member.

After a work-item has arrived at an fbarrier, it cannot wait, arrive, or leave the same fbarrier unless the fbarrier has been satisfied and the `arrive_count` has been reset to 0.

`leavefbar`

Indicates that the work-item is no longer a member of the fbarrier. It does not wait for other work-items that are members of the fbarrier to arrive. If the work-item is the last of the fbarrier members to arrive, then any work-items waiting on the fbarrier can proceed and the fbarrier is reset.

The *src* operand specifies the fbarrier to leave.

Every work-item that joins an fbarrier must leave the fbarrier before it exits.

An `leavefbar` operation does not perform a memory fence before proceeding. An explicit `sync` operation can be used if that is required in order to make any data being communicated visible.

This operation (which includes the value of the *src* operand) must be wavefront uniform (see 2.14 Uniform Operations (p. 23)). This implies that all active work-items of a wavefront must be members of the same fbarriers.

`leavefbar` conceptually atomically decrements the `member_count` for the wavefront, and checks if the `arrive_count` equals the `member_count` . If it does, then atomically any wavefronts in the `wait_set` are allowed to proceed, the `arrive_count` is reset to 0, and the `wait_set` is reset to empty.

A work-item must not leave an fbarrier that has not been initialized, nor leave an fbarrier of which it is not a member.

`releasefbar`

Before all work-items of a work-group exit, every fbarrier that has been initialized by a work-item of the work-group using `initfbar` must be released.

The *src* operand specifies the fbarrier to release.

Once released, the fbarrier is no longer considered initialized. An fbarrier must not be released if it is not already initialized. This implies that only one work-item of the work-group must perform the `releasefbar` operation at a time.

An fbarrier must have no members when released. This implies that every work-item that joins an fbarrier must leave the fbarrier before it exits.

An fbarrier must be released, because some implementations might need to deallocate the additional resources allocated to an fbarrier when it was initialized.

There must not be a race condition between the other work-items in the work-group that execute fbarrier operations on the same fbarrier and the work-item that executes the `releasefbar`. This requirement can be satisfied by using the `barrier` operation, or the `waitfbar` operation (on another fbarrier) between the fbarrier operations that use it and the `releasefbar`.

`ldf`

Places the address of an fbarrier into the destination *dest*. The address has work-group persistence (see 2.9.1 Persistence Rules (p. 19)) and the value can only be used in work-items that belong to the same work-group as the work-item that executed the `ldf` operation. The compound type *dest* is always `u32` regardless of the machine model (see 2.10 Small and Large Machine Models (p. 20)). The value returned can be used with fbarrier operations to specify which fbarrier they are to operate on.

9.3.4 Additional Information About Fbarrier Operations

Additional information about the use of fbarrier operations:

- Fbarrier operations are allowed in divergent code. In fact, this is a primary reason to use fbarriers rather than the `barrier` operation, which can only be used in work-group uniform code. However, fbarrier usage must be wavefront uniform.
- The fbarrier operation that arrives at an fbarrier does not need to be the same operation in each wavefront. The operation simply needs to reference the same fbarrier.
- The fbarrier operations that operate on a particular fbarrier do not need to be in the same code block. They are allowed to be in both the kernel body and different function bodies.
- Fbarriers can be used in functions. If the function is called in divergent code, then an fbarrier can be passed by reference as an argument so the function has an fbarrier that has all the work-items that are calling it as members. The function can use this to synchronize usage of its own fbarriers.
- An fbarrier can be initialized and released multiple times. While not initialized, the group memory associated with an fbarrier can be used for other purposes. However, on some implementations, the cost to initialize and release an fbarrier might make it preferable to only perform these operations once per work-group fbarrier, and then reuse the same fbarrier by using `joinfbar` and `leavefbar`. A `barrier` operation, or `waitfbar` (to another fbarrier) operation can be used between the `leavefbar` and `joinfbar` operations to avoid race conditions between the fbarrier operations that use the fbarrier for different purposes.

When using fbarrier operations, the following rules must be satisfied or the execution behavior is undefined:

- All work-items that are members of an fbarrier must perform either an `waitfbar`, `arrivefbar`, or `leavefbar` on the fbarrier; otherwise, deadlock will occur when a work-item performs an `waitfbar` on the fbarrier.
- No work-item is allowed to be a member of any fbarrier when it exits. It must perform an `leavefbar` on every fbarrier on which it performs an `joinfbar`.

- While a work-item is waiting on an fbarrier, it is allowed for other work-items in the same work-group to perform `joinfbar`, `waitfbar`, `arrivefbar`, and `leavefbar` operations. All but `joinfbar` can cause the waiting work-items to be allowed to proceed, either because the `arrive_count` is incremented to match the `member_count`, or the `member_count` is decremented to match the `arrive_count`.

  However, there must not be a race condition between `joinfbar` operations and `waitfbar`, `arrivefbar`, and `leavefbar`, operations such that the order in which they are performed might affect the number of members the fbarrier has when a wait is satisfied.

  One way to satisfy this requirement is by using the `barrier` operation, or the `waitfbar` operation (on another fbarrier), between the `joinfbar` and `waitfbar`, `arrivefbar`, and `leavefbar` operations. This ensures that all work-items have become members before any start arriving at the fbarrier. However, other uses of `barrier` and`waitfbar` (on another fbarrier) operations can also ensure the race condition free requirement.

- Similarly, there cannot be a race condition between an `arrivefbar` operation and other fbarrier operations that could result in the same work-item performing more than one fbarrier operation on the same fbarrier without the fbarrier having been satisfied and the `arrive_count` being reset to 0.

  This requirement can also be satisfied by using a `barrier` or `waitfbar` (on another fbarrier) operation after the `arrivefbar` operation.

9.3.5 Pseudocode Examples

To use fbarriers in divergent code, it is necessary to create an fbarrier with only the work-items that are executing the divergent code. This can be done by creating an fbarrier with all the work-items and then using `leavefbar` on the non-interesting divergent paths as shown in Example 1.

```
Example 1: Using leavefbar to create an fbarrier that only contains
           divergent work-items.
01: fbarrier %fb1;
02: if (workitemflatid_u32 == 0) {
03:   initfbar %fb1;
04: }
05: barrier_fnone;
06: joinfbar %fb1; // start with all work-items
07: barrier_fnone;
08: if (cond1) { // cond1 must be WAVESIZE uniform
09:   ...
10:   if (cond2) { // cond2 must be WAVESIZE uniform
11:     ...
12:     waitfbar_fglobal %fb1; // fb1 only has work-items for which
                               // cond1 && cond2 is true as other
                               // work-items have left on
                               // lines 16 and 19.
13:     ...
14:     leavefbar %fb1;
15:   } else {
16:     leavefbar %fb1;
17:   }
18: } else {
19:   leavefbar %fb1;
20: }
21: barrier_fnone;
22: if (workitemflatid_u32 == 0) {
23:   releasefbar %fb1;
24: }
```

Or an fbarrier can be created that has all the work-items on all divergent paths, and then using this to synchronize creating another fbarrier that only the work-items executing the desired divergent path join as shown in Example 2.

Example 2: Using joinfbar to create an fbarrier that only contains divergent work-items.

```
01: fbarrier %fb0;
02: fbarrier %fb1;
03: if (workitemflatid_u32 == 0) {
05:   initfbar %fb0;
06:   initfbar %fb1;
07: }
08: barrier_fnone;
09: joinfbar %fb0; // fb0 has all work-items of work-group
10: barrier_fnone;
11: if (cond1) {    // cond1 must be WAVESIZE uniform
12:   ...
13:   if (cond2) {  // cond2 must be WAVESIZE uniform
14:     joinfbar %fb1;
15:     waitfbar_fnone %fb0; // wait for all work-items to either
                             // join fb1 on line 14 or arrive at
                             // line 21 or 24
16:     ...
17:     waitfbar_fglobal %fb1; // fb1 only has work-items for which
                               // cond1 && cond2 is true
18:     ...
19:     leavefbar %fb1;
20:   } else {
21:     waitfbar_fnone %fb0;
22:   }
23: } else {
24:   waitfbar_fnone %fb0;
25: }
26: leavefbar %fb0;
27: barrier_fnone;
28: if (workitemflatid_u32 == 0) {
29:   releasefbar %fb0;
30:   releasefbar %fb1;
31: }
```

The following example uses two fbarriers to allow producer and consumer wavefronts to overlap execution.

```
Example 3: Producer/consumer using two fbarriers that allow
           producer and consumer wavefront executions to overlap.

kernel producerConsumer()
{
  // Declare the fbarriers.
  fbarrier %producer_fb;
  fbarrier %consumer_fb;

  // Use a single work-item to initialize the fbarriers.
  if (workitemflatid_u32 == 0) {
    initfbar [%produced_fb];
    initfbar [%consumed_fb];
  }
  // Wait for fbarriers to be initialized before using them.
  // No memory fence required as no data has been produced yet.
  barrier_fnone;

  // All work-items join both fbarriers.
  joinfbar [%fb_produced];
  joinfbar [%fb_consumed];
  // Wait for all fbarriers to join to prevent a race condition
  // between join and subsequent wait.
  // No memory fence required as no data has been produced yet.
  barrier_fnone;

  // Ensure all produces and consumers are in the same wavefront
  // so that the fbarrier operations are wavefront uniform.
  producer = ((workitemflatid_u32 / WAVESIZE) & 1) == 1;

  if (producer) {
    for (i = 0 to n) {
      // Producer compute new data.

      // Wait until all consumers have processed the previous
      // data before storing the new data.
      // No need for a memory fence as consumer is producing no data
      // used by the consumer.
      waitfbar_fnone [%consumed_fb];
      // fill in new data in some group segment buffer data.
      // Tell the consumers the data is ready.
      // Using arrive allows the producer to continue computing new data
      // before all consumers have read this data.
      // Memory fence should correspond to segment holding data to
      // make sure it is visible to consumer.
      arrivefbar_fgroup [%produced_fb];
  } else {
    // Tell producer ready to receive new data. This is the
    // initial state of a consumer.
    // No memory barrier required as consumer is not producing any data.
    arrivefbar_fnone [%consumed_fb];

    for (j = 0 to n) {
      // Wait for all producers to store new data.
      // Memory fence should correspond to segment holding data to make
      // sure it is visible to consumer.
```

```
      waitfbar_fgroup [%produced_fb];

      // Consumer reads the new data

      // Only need to tell producer have read data if there is
      // another value to be produced.
      if (j != n) {
        // Tell producer have read new data.
        // Using arrive allows the consumer to start processing the data
        // before all consumers have read the data.
        // No memory barrier required as consumer is not producing any data.
        arrivefbar_fnone [%consumed_fb];
      }

      // Consumer processes new data.
    }
  }
  // Ensure each work-item leaves the fbarriers it has
  // joined before it terminates.
  leavefbar %producer_fb;
  leavefbar %consumer_fb;

  // Wait for fbarriers to be finished with before releasing them.
  // No memory fence required as no data has been produced.
  barrier_fnone;

  // Use a single work-item to release the fbarriers.
  if (workitemflatid_u32 == 0) {
    releasefbar %produced_fb;
    releasefbar %consumed_fb;
  }
  }
```

Examples

```
fbarrier %fb;
initfbar %fb;
joinfbar %fb;
waitfbar_fglobal %fb;
waitfbar %fb;
arrivefbar_fpartial %fb;
leavefbar %fb;
releasefbar %fb;
ldf_u32 $s0, %fb;
joinfbar $s0;
```

9.4 Synchronization (sync) Operation

The sync operation makes sure that the writes in the program preceding the sync have been fully committed to memory before the program continues.

9.4.1 Syntax

Table 9–3 Syntax for sync Operation

Opcode and Modifier
sync_*fence*

Explanation of Modifier
fence: Optional. The memory segment that the sync operation forces to become synchronized. Either `fgroup`, `fglobal`, `fboth`, `fpartial`, or `fpartialboth`. `fnone` is not allowed, because this is a memory synchronization operation. If omitted, `fboth` is used. See 9.1 Memory Fence Modifier (p. 185).

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.5 BRIG Syntax for Parallel Synchronization and Communication Operations (p. 323).

9.4.2 Description

The sync operation is always a two-way fence. It waits for all prior loads and stores to complete. See 9.1 Memory Fence Modifier (p. 185).

For example:

```
st_fglobal_u32 1, [&x];
sync;           // will wait till 1 is stored into memory
```

The sync operation can be used in conditional code.

Examples

```
sync;
sync_fgroup;
```

9.5 Cross-Lane Operations

These operations perform work across lanes in a wavefront.

9.5.1 Syntax

Table 9–4 Syntax for Cross-Lane Operations

Opcodes	Operands
`countlane_u32`	*`dest`*, *`src0`*
`countuplane_u32`	*`dest`*
`masklane_b64`	*`dest`*, *`src0`*
`sendlane_b32`	*`dest`*, *`src0`*, *`src1`*
`receivelane_b32`	*`dest`*, *`src0`*, *`src1`*

Explanation of Operands
`dest`: Destination register. For `countlane`, `countuplane`, `sendlane`, and `receivelane`, *`dest`* must be an `s` register. For `masklane`, *`dest`* is a `d` register.
`src0`, *`src1`*: Sources. For `countlane` and `masklane`, can be a `c` register, `s` register, an immediate value, or `WAVESIZE`. For `sendlane` and `receivelane`, can be an `s` register, an immediate value, or `WAVESIZE`.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.5 BRIG Syntax for Parallel Synchronization and Communication Operations (p. 323).

9.5.2 Description

`countlane`
: Counts the number of work-items in the current wavefront that have a non-zero source *`src0`*. The operation returns a value greater than or equal to 0 and less than or equal to `WAVESIZE`.

`countuplane`
: Forms a prefix sum of the number of executing work-items in the current wavefront. That is, the operation sets the destination *`dest`* in each work-item to the count of the number of earlier (in flattened work-item order) work-items that execute the statement.

  Because `countuplane` gives each executing work-item in the wavefront a unique value, it is often used in compaction.

`masklane`
: Returns a bit mask that shows which work-items in the wavefront have a non-zero source *`src0`*. The affected bit position within *`dest`* corresponds to each work-item's lane ID. Any remaining bits are cleared.

`sendlane`

Transfers the value in *`src0`* to the destination lane specified in *`src1`*. The value in *`src1`* must be greater than or equal to 0 and less than `WAVESIZE`; otherwise the result is undefined.

If either the receiving or sending lane is inactive, the value returned in the receiving lane is undefined.

If the value in *`src1`* is duplicated in different lanes, so that multiple lanes are sending to the same lane, the value of *`dest`* in the receiving lane will be one of the values of *`src0`* from a sending lane. If no lane sends to *`dest`*, the value of *`dest`* is unchanged.

The `sendlane` operation is independent from the `receivelane` operation; they do not form a matched pair.

`receivelane`

Transfers the data value in *`src0`* to the current lane from the source lane specified in *`src1`*. The value in *`src1`* must be greater than or equal to 0 and less than `WAVESIZE`; otherwise the result is undefined.

If either the receiving or sending lane is inactive, the value returned in the receiving lane is undefined.

The `receivelane` operation is independent from the `sendlane` operation; they do not form a matched pair.

Examples

```
countlane_u32 $s1, $s2;

masklane_b64 $d1, $s0;

countuplane_u32 $s1;

sendlane_b32 $s1, $s2, 3;
sendlane_b32 $s1, 3, $s2;
sendlane_b32 $s1, $s2, $s2;

receivelane_b32 $s1, $s2, $s2;
receivelane_b32 $s1, 23, $s2;
```

Chapter 10 Functions

This chapter describes how to use functions in HSAIL.

See also Chapter 11 Operations Related to Functions (p. 211).

10.1 Functions in HSAIL

Like other programming languages, HSAIL provides support for user functions.

In order that HSAIL can execute efficiently on a wide range of compute units, an abstract method is used for passing arguments, with the finalizer determining what to do. This is necessary because, on a GPU, stacks are not a good use of resources, especially if each work-item has its own stack. If an application is simultaneously running, for example, 30,000 work-items, then the stack-per-work-item is very limited. Having one return address per wavefront (not one address per work-item) is desirable.

Implementations should map the abstractions into appropriate hardware.

Functions cannot be nested, but functions can be recursive.

10.1.1 Example of a Simple Function

The simplest function has no arguments and does not return a value. It is written in HSAIL as follows:

```
function &foo()()
{
    ret;
};
function &bar()()
{
   {call &foo;} //start argument scope
};
```

Execution of the `call` operation transfers control to `foo`, implicitly saving the return address. Execution of the `ret` operation within `foo` transfers control to the operation following the call.

10.1.2 Example of a More Complex Function

Here is a more complex example of a function:

```
//  Call a compare function with two floating-point arguments
//  Allocate multiple arg objects to hold arguments

// ...
{
   arg_f32 %a;
   arg_f32 %b;

// Fill in the arguments
   st_arg_f32 4f, [%a];
   st_arg_f32 $s0, [%b];
   arg_f32 %res;
   call &compare (%res)(%a, %b);
   ld_arg_f32 $s0, [%res];
}  // End argument scope

// More code
};

function &compare (arg_f32 %res) (arg_f32 %left, arg_f32 %right)
{
  ld_arg_f32 $s0, [%left];
  ld_arg_f32 $s1, [%right];
  cmp_eq_f32_f32 $s0, $s1, $s0;
  st_arg_f32 $s0, [%res];
  ret;
};
```

10.1.3 Function Pointers

Function pointers can be set up in two ways:

- A load code address operation can store a pointer to a function into a 32-bit or 64-bit register. (See Table 2–3 (p. 20).) A register holding a function pointer can then be used in a call statement. It is not valid to modify any function pointer with arithmetic operations.
- A host CPU can use a runtime library call to obtain the address of a non-static function. That address can then be passed into a kernel.

10.1.4 Functions That Do Not Return a Result

Functions that do not return a result are declared with an empty return arguments list:

```
function &foo ()(arg_u32 %in)
{  // does not return a value
   ret;
};
```

10.2 Argument Passing Rules

Currently, HSAIL supports only a single output argument from a function. Additional results can always be passed by allocating space in the caller and passing an address. Later versions might allow additional output parameters.

Arguments in calls must be variables in the arg segment. (See 10.4 Arg Segment (p. 207).) Arguments are pass-by-value.

It is not legal to use the same name as both an input and an output argument.

The type and size of actual arguments must be compatible with the formal parameters. An actual argument is compatible with a formal parameter if one of these three properties holds:

- The two have identical properties, type, size, and alignment.
- Both are arrays with the same size and alignment and the elements have identical properties.
- The argument is the last parameter and both are arrays with elements that have identical properties, both arrays have the same alignment, and the formal is an array with unspecified size. See 10.5 Variadic Functions (p. 208).

10.3 Function Declarations, Function Definitions, and Function Signatures

Functions cannot be nested, but functions can be recursive.

Every function must be declared or defined prior to being called.

After a function has been declared, a `call` operation can use the function as a target. See 11.1 call Operation (p. 211).

10.3.1 Function Declaration

A function declaration is a function without a code block. A function declaration declares a function, providing attributes, the function name, and names and types of the output and input arguments.

For example:

```
function &fun (arg_u32 %out) (arg_u32 %in0, arg_u32 %in1);
```

10.3.2 Function Definition

A function definition defines a function. It is a function declaration followed by a code block:

```
function &fun (arg_u32 %out) (arg_u32 %in0, arg_u32 %in1)
{
    ret;
};
function &caller()()
{
  {
     arg_u32 %input1;
     arg_u32 %input2;
     call &fnWithTwoArgs ()(%input1, %input2); // call of a function
                                               // all work-items called
  }
  // ...
};
```

10.3.3 Function Signature

A function signature does not describe a single function: it describes a set of functions that have the same types for arguments.

A function signature declares a type of function. Obviously, the type cannot be called, but it can be used to describe the target of a call by means of a function pointer.

A function signature contains attributes, the signature name, and the types (and optional names) of the output and input arguments.

Syntactically, a signature is much like a function. A signature defines a name, which must start with an ampersand (`&`). The name can be used in indirect function calls.

The syntax is:

`'signature'` *`name outputFormalArgs inputFormalArgs`*

where:

- *`outputFormalArgs`* is a parenthesized list of zero or one argument description optionally followed by an identifier.
- *`inputFormalArgs`* is a parenthesized list of zero or more argument descriptions optionally followed by identifiers.

An argument description contains up to three parts:

- An optional `align` *`n`*
- A type
- An optional identifier, which may have an array size

In the following example, assume that `$d2` in each work-item contains the address of a function whose arguments meet this signature:

```
signature &bar_t (arg_u32 ) (align 8 arg_f32, arg_f32 %x[10]);
signature &fun_t (arg_u32) (arg_u32, arg_u32);
function &caller1 ()()
{
   {
   arg_u32 %out;
   arg_u32 %in1;
   arg_u32 %in2;
call $d2 (%out)(%in1, %in2) &fun_t;
// ...
   }
};
```

This is a call of some function that takes two `u32` arguments and returns a `u32` result. The particular target function is selected by the contents of register `$d2`. Each work-item has its own `$d2`, so this might call many different functions.

The behavior is undefined if the register points to a function that does not match the signature.

10.4 Arg Segment

When a call to a function is executed by a work-item, argument passing is done through variables in the arg segment. Arg variables cannot be read or written by other work-items.

Arg variables are allocated much like other variables but must be in an argument scope. See 4.9 Argument Scope (p. 38). A function can allocate an arbitrary number of arg variables. Each implementation is allowed to limit the number of bytes used for the allocation of arg variables but must support a minimum of 64 bytes.

Arg variables can be used in `ld`, `st`, and `lda` operations.

The variants of `st` can modify arg variables, arg variables can be passed as arguments, and `lda` can be used to take the address of an arg variable.

It is not legal to take the address of an arg variable that is not a formal parameter (that is an argument defined in the current argument block).

Arg variables that are used to pass values into a function are live from the point of the definition to the end of the argument scope.

Input arguments cannot be used after the call, but output arguments can be used to read returned values.

Each work-item can set a different value into its own arg variable.

Calls to functions operate as described below.

In the caller:

1. Allocate arg locations to hold arguments.
2. Store the values into the input arguments.
3. Make the call.
4. Optionally load the result.

In the callee:

1. The arg comes into the function as a formal argument.
2. Code should use loads to access the arguments.
3. The callee can take the address of the arg variable and pass it to additional subroutines.
4. Store the result into an output argument.

An arg object can be the address of an array that is allocated to private memory. The arg object can be used to bundle up a sequence of actual arguments and then pass the entire arg object to the function.

Array arg objects are useful in the following cases:

- To pass a large number of arguments to a function
- To pass a variable number of arguments to a function
- To pass arguments of different types to a function
- When code needs to take the address of an argument list

The finalizer can implement arg objects as physical registers or can map the arg object into memory. Each finalizer release will document the argument convention for each target chip.

10.5 Variadic Functions

A variadic function is a function that accepts a variable number of arguments.

In HSAIL, variadic functions are declared with a last argument, which is an array with no specified size (for example, `uint32 extra_args[]`). The matching actual argument must be a fixed-size array.

An array with unspecified size is only allowed as the last argument definition of a function declaration, function definition, or function signature, or for a global symbol definition that has an initializer. For more information, see 4.13.1 Integer Constants (p. 41).

10.5.1 Example of a Variadic Function

The example function below computes the sum of a list of floating-point values.

The first argument to the function is the size of the list.

The second argument is an array of floating-point values.

```
function &maxofN(arg_f32 %r)(arg_u32 %n, align 8 arg_u8 %last[])
{
   ld_arg_u32 $s0, [%n];         // s0 holds the number to add
   mov_b32 $s1, 0;               // s1 holds the sum
   mov_b32 $s3, 0;               // s3 is the offset into last
   @loop:
   cmp_eq_b1_u32 $c1, $s0, 0;    // see if the count is zero
   cbr $c1, @done;               // if it is,  jump to done
   ld_arg_f32 $s4, [%last][$s3]; // load a value
   add_f32 $s1, $s1, $s4;        // add the value
   add_u32 $s3, $s3, 4;          // advance the offset to the next element
   sub_u32 $s0, $s0, 1;          // decrement the count
   brn @loop;
   @done:
   st_arg_f32 $s1, [%r];
   ret;
};

kernel &adder()
{ // here is an example caller passing in 4 32-bit floats
  {
     align 8 arg_u8 %n[16];
     st_arg_f32 1.2f, [%n][0];
     st_arg_f32 2.4f, [%n][4];
     st_arg_f32 3.6f, [%n][8];
     st_arg_f32 6.1f, [%n][12];
     arg_u32 %count;
     st_arg_u32 4, [%count];
     arg_f32 %sum;
     call &maxofN(%sum)(%count, %n);
     ld_arg_f32  $s0, [%sum];
  }
  // ...  %s0 holds the sum
};
```

10.6 align Field

`align` is an optional field indicating the alignment of the arg object in bytes.

The number *n* can be 1, 2, 4, 8, or 16.

Without `align`, the variable is naturally aligned. That is, it is allocated at an address that is a multiple of the variable's type.

For example:

```
{
 arg_u32 %x;            // holds one 32-bit integer value
 arg_f64 %y[3];         // holds three 64-bit float doubles
 align 8 arg_b8 %a[16]; // holds 16 bytes on an 8-byte boundary
}
```

`align` is useful when you want to pass different types to the same function.

Consider a function `foo` that is a simplified version of `printf`. `foo` takes in two formal arguments. The first argument is an integer 0 or 1. That argument determines the type of the second argument, which is either a double or a character:

```
function &top ()()
{
   // ...
   global_f64 %d;
   global_u8 %c[4];
   ld_global_f64 $d0, [%d];
   ld_global_u8 $s0, [%c];
   {
      align 8 arg_b8 %sk[12]; // ensures that sk starts on an 8-byte
                              // boundary so that both 32-bit and
                              // 64-bit stores are naturally aligned
      st_arg_u32 $s0, [%sk][8]; // stores 32 bits into the back of sk
      st_arg_u64 $d0, [%sk][0]; // stores 64 bits into the front of sk
      call &foo(%sk);
    }
  // ...
};
function &foo () (align 8 arg_b8 %z[])
{
   // ...
   ret;
};
```

Chapter 11

Operations Related to Functions

This chapter describes operations related to functions.

See also Chapter 10 Functions (p. 203).

11.1 call Operation

The `call` operation stores the address of the next operation, so execution can resume at that point after executing a `ret` operation.

11.1.1 Syntax

Table 11–1 Syntax for call Operation

Opcode and Modifiers	Operands
`call`	***`functionName`*, *`inputArgs`***
`call`	***`functionName`*, *`outputArg`*, *`inputArgs`***
`call`_*`width`*	***`src`*, *`inputArgs`*, *`targets`***
`call`_*`width`*	***`src`*, *`outputArg`*, *`inputArgs`*, *`targets`***

Explanation of Modifier
`width`: Optional: Specifies the number of consecutive work-items in flattened ID order that are guaranteed to call the same target for indirect calls. See the Description below.

Explanation of Operands
`functionName`: Name of the function to call.
`inputArgs`: Parenthesized list of zero or more call arguments separated by commas.
`outputArg`: Parenthesized list of zero or one call argument.
`src`: A register, either `s` or `d`, depending on the memory model, that holds the address of the function to call. See Table 2–3 (p. 20).
`targets`: Comma-separated list in square brackets of global identifiers or a single entry that is a signature. Each identifier in the comma-separated list is a function, in which case the list is complete. All possible targets are named. All functions in the list must take the same type arguments.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.6 BRIG Syntax for Operations Related to Functions (p. 324).

11.1.2 Description

Calls must appear inside of an argument scope, even if the call has no arguments. See 4.9 Argument Scope (p. 38).

Arguments must be variables in the arg segment.

Return arguments are optional.

Arguments are pass-by-value.

In order to finalize the abstract call API, the finalizer might need to know the alignment, type, and length of arguments that are passed at the call site. This information can be supplied at the end of the call, either by a comma-separated list of all possible targets or by a signature indicating the types and lengths of all arguments of the call. The finalizer might be able to generate more efficient code if the list of targets is supplied.

width

Specifies the number of consecutive work-items in flattened ID order that are guaranteed to call the same target for indirect calls.

The value can be `width(`*n*`)`, `width(WAVESIZE)`, or `width(all)`.

Because work-items are executed in wavefronts, a single indirect call can reach multiple targets if the register has different values in different work-items. The optional width modifier can be used to inform the finalizer that consecutive work-items in flattened ID order will call the same target.

If work-items specified by the width modifier do not call the same target, the behavior is undefined.

If the width modfiier is not provided, every work-item can call a different address (in other words, `width(1)` is the default).

See 2.13.1 Width Modifier (p. 22)).

Example

```
function &bar (arg_u32 %r) (arg_f32 %a);
signature &barone_t (arg_u32 %r) (arg_f32 %a);
function &foo (arg_u32 %r) (arg_f32 %a);
function &Example()(arg_u32 %arg1) {
    ldc_u64 $d1, &foo;
    cbr $c0, @lab;
    ldc_u64 $d1, &bar;
    @lab: st_arg_f32 2f, [%arg1];
    {
       arg_u32 %res;

       // call foo or bar using an explicit list
       // foo and bar are the two potential targets
       // $s1 can contain the address of foo or bar

       call_width(all) $d1 (%res)(%arg1) [&foo, &bar];
    }
    { arg_u32 %res;

    // call foo or bar using a signature

    call_width(all) $d1 (%res)(%arg1) &barone_t;
    }
};
```

11.2 Return (ret) Operation

The return (`ret`) operation returns from a function back to the caller's environment. `ret` can also be used to exit a kernel.

If the program does not have a `ret` operation before the exit of the kernel or function's code block, the finalizer will place the `ret` operation at the end of the function.

11.2.1 Syntax

Table 11–2 Syntax for ret Operation

Opcode
ret

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.6 BRIG Syntax for Operations Related to Functions (p. 324).

11.2.2 Description

Within a function, a `ret` operation inside of divergent control flow causes control to transfer to the end of the function, where the work-item waits for all the other work-items in the same wavefront. Once all work-items in a wavefront have reached the end of the function, the function returns.

Within a kernel, a `ret` operation inside of divergent control flow causes control to transfer to the end of the kernel, where the work-item waits for all the other work-items in the same work-group. Once all work-items in a work-group have reached the end of the kernel, the work-group finishes.

As the return is executed for a function, all values in the return arguments list are copied to the corresponding actual arguments in the call site.

A `ret` operation executed in a kernel will wait for all work-items to complete and then terminate the kernel execution.

Example

```
ret;
```

11.3 System Call (syscall) Operation

The system call (`syscall`) operation is used to call runtime library-supplied functions that will be executed by a host CPU. These are typically functions that are best implemented on a host CPU.

Each implementation and runtime library provides a list of such functions.

Some common functions might be `vprintf`, `malloc`, `free`, `fileio`, and so forth.

11.3.1 Syntax

Table 11–3 Syntax for System Call (syscall) Operation

Opcode	Operands
`syscall_`*`TypeLength`	***`dest`*`, n, src0, src1, src2`**

Explanation of Modifier
`TypeLength`: u32, u64 (see Table 4–2 (p. 46)).

Explanation of Operands
dest: Destination. Must be a register of size *TypeLength*.
n: Number, of type *TypeLength*, for the runtime library function to call. Must be an immediate value or WAVESIZE.
src0, *src1*, *src2*: Sources. Each source is of type *TypeLength* and must be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.6 BRIG Syntax for Operations Related to Functions (p. 324).

11.3.2 Description

The following steps occur:

1. The `syscall` operation transfers an operation number and up to three operands of `u32` or `u64` data from an HSA component to a host CPU.
2. The work-item waits for the host CPU to finish.
3. The host CPU calls the associated (by operation number) runtime library routine passing the data from each work-item.
4. A 32-bit or 64-bit result is returned and the work-item is resumed.

Developers who want to pass more data to a specific `syscall` operation can use buffers in global memory. HSAIL does not support a large argument list.

Examples

```
syscall_u32 $s1, 3, $s2, $s3, $s4;
syscall_u64 $d1, 10, $d2, 100, $d4;
```

11.4 Allocate Memory (alloca) Operation

The allocate memory (`alloca`) operation is used by kernels or functions to allocate per-work-item private memory at run time.

The allocated memory is freed automatically when the kernel or function exits.

11.4.1 Syntax

Table 11–4 Syntax for Allocate Memory (alloca) Operation

Opcode	Operands
`alloca_private_u32`	*`dest`*, *`src`*

Explanation of Operands
`dest`: Destination. Must be a 32-bit register.
`src`: Source. Can be a 32-bit register, immediate value, or `WAVESIZE` (see 4.17 Operands (p. 50)). The value of *`src`* is the minimum amount of space (in bytes) requested.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.6 BRIG Syntax for Operations Related to Functions (p. 324).

11.4.2 Description

The `alloca` operation sets the destination *`dest`* to the private segment address of the allocated memory. The memory can then be accessed with `ld_private` and `st_private` operations.

Whenever a particular alignment is needed, the application might need to allocate additional space and adjust the returned address to achieve the desired alignment.

The size is specified in bytes. However, an implementation is allowed to allocate more than requested. For example, the request can be rounded up to ensure that a stack pointer maintains a certain alignment.

The behavior is undefined if not enough private memory is available to satisfy the requested size.

Example

```
alloca_private $s1, 24;
```

Chapter 12

Special Operations

This chapter describes special operations used to access predefined values.

12.1 Syntax

The table below shows the syntax for the special operations in alphabetical order.

Table 12–1 Syntax for Special Operations

Opcodes and Modifier	Operands
cleardetectexcept_u32	***exceptionsNumber***
clock_u64	***dest***
cuid_u32	***dest***
currentworkgroupsize_u32	***dest, dimNumber***
debugtrap_u32	***src***
dim_u32	***dest***
dispatchid_u64	***dest***
dispatchptr_global_u*Length*	***dest***
getdetectexcept_u32	***dest***
gridgroups_u32	***dest, dimNumber***
gridsize_u32	***dest, dimNumber***
laneid_u32	***dest***
maxcuid_u32	***dest***
maxwaveid_u32	***dest***
nop	
nullptr_*segment*_u*Length*	***dest***
qid_u32	***dest***
qptr_global_u*Length*	***dest***
setdetectexcept_u32	***exceptionsNumber***
waveid_u32	***dest***
workgroupid_u32	***dest, dimNumber***
workgroupsize_u32	***dest, dimNumber***
workitemabsid_u32	***dest, dimNumber***
workitemflatabsid_u32	***dest***
workitemflatid_u32	***dest***
workitemid_u32	***dest, dimNumber***

Explanation of Modifiers
segment: Optional segment name.
Length: 32, 64. Must match the address size for the specified segment (see Table 2–3 (p. 20)).

Explanation of Operands
dest: Destination. Must be an s register, except for clock and dispatchid, which require a d register. For nullptr, must be a register with a size that matches the address size of the segment address. See Table 2–3 (p. 20).
dimNumber: Source that selects the dimension (X, Y, or Z). 0, 1, and 2 are used for X, Y, and Z, respectively. Must be an immediate value of data type u32.
exceptionsNumber: Source that specifies the set of exceptions. bit:0=INVALID_OPERATION, bit:1=DIVIDE_BY_ZERO, bit:2=OVERFLOW, bit:3=UNDERFLOW, bit:4=INEXACT; all other bits are ignored. Must be an immediate value of data type u32.

Exceptions (see Chapter 13 Exceptions (p. 225))
No exceptions are allowed.

For BRIG syntax, see 19.10.7 BRIG Syntax for Special Operations (p. 324).

12.2 Description

`cleardetectexcept`
: Clears DETECT exception flags specified in *exceptionsNumber* for the work-group containing the work-item. The result is undefined if used inside divergent control flow, and might lead to deadlock. See 12.3 Additional Information on DETECT Exception Operations (p. 222).

`clock`
: Stores the current value of a 64-bit unsigned cycle counter in a d register specified by the destination *dest*. The same clock is required to be used by all compute units of a single HSA component, but is not required to be the same clock used by other HSA components or the host CPU. The clock must count at a fixed frequency. The HSA runtime can be queried to determine the frequency of the clock for each HSA component, and to convert the clock value of a specific HSA component into the global time domain.

`cuid`
: Returns a 32-bit unsigned number identifying the compute unit on which the work-item is currently executing and stores the result in the destination *dest* a number between 0 and `maxcuid`. `cuid` is helpful in determining the load balance of a kernel. Implementations are allowed to move in-flight computations between compute units, so the value returned can be different each time `cuid` is executed.

`currentworkgroupsize`
Accesses the work-group size that the currently executing work-item belongs to for the *`src`* dimension (see 2.2 Work-Groups (p. 7)) and stores the result in the destination *`dest`*.

Because the grid is not required to be a multiple of the work-group size, there can be partial work-groups. The `currentworkgroupsize` operation returns the work-group size that the current work-item belongs to. The value returned by this operation will only be different from that returned by the `workgroupsize` operation if the current work-item belongs to a partial work-group.

If it is known that the kernel is always dispatched without partial work-groups, then it might be more efficient to use the `workgroupsize` operation.

If the kernel was dispatched with fewer dimensions than *`src`*, then `currentworkgroupsize` returns 1 for the unused dimensions.

`debugtrap`
Halts the current wavefront and transfers control to the debugger. The source *`src`* is passed to the debugger and can be used to identify the trap.

`dim`
Returns the number of dimensions in use by this dispatch and stores the result in the destination *`dest`*. See 2.1 Overview of Grids, Work-Groups, and Work-Items (p. 5).

`dispatchid`
Returns a 64-bit dispatch identifier (dispatch ID) that is unique for the queue used for this dispatch and stores the result in the destination *`dest`*.

The combination of the dispatch ID and the queue ID is globally unique. Debuggers might find this useful.

`dispatchptr`
Sets the destination *`dest`* to the global segment address of the AQL dispatch packet that invoked this kernel execution. The format of the dispatch packet is defined in the *HSA System Architecture Specification*.

`getdetectexcept`
Returns the current value of DETECT exception flags, which is a summarization for all work-items in the work-group containing the work-item, and stores the result in the destination `dest`. The bits in the result indicate if that exception has been generated in any work-item within the work-group containing the current work-item, as modified by any preceding `cleardetectexcept` or `cleardetectexcept` operations executed by any work-item in the work-group containing the current work-item. The bits correspond to the exceptions as follows: bit 0 is INVALID_OPERATION, bit 1 is DIVIDE_BY_ZERO, bit 2 is OVERFLOW, bit 3 is UNDERFLOW, bit 4 is INEXACT, and other bits are ignored. The result is undefined if used inside divergent control flow, and might lead to deadlock. See 12.3 Additional Information on DETECT Exception Operations (p. 222).

`gridgroups`
Returns the upper bound for work-group identifiers (IDs) (that is, the number of work-groups) within the grid and stores the result in the destination *`dest`*.

If the grid was launched with fewer dimensions than the work-item absolute ID, then `gridgroups` returns 1 for the unused dimensions.

`gridgroups` is always equal to `gridsize` divided by `workgroupsize` rounded up to the nearest integer.

`gridsize`
Returns the upper bound for work-item absolute identifiers (IDs) within the grid and stores the result in the destination *`dest`*.

If the grid was launched with fewer dimensions than the work-item absolute ID, then `gridsize` returns 1 for the unused dimensions.

`laneid`
Returns the identifier (ID) of the work-item's lane within the wavefront, a number between 0 and `WAVESIZE` – 1, and stores the result in the destination *`dest`*.

The compile-time macro `WAVESIZE` can be used to generate code that depends on the wavefront size.

`maxcuid`
Returns the number of compute units –1 for this HSA component and stores the result in the destination *`dest`*. For example, if an HSA component has four compute units, `maxcuid` will be 3.

`maxwaveid`
Returns the number of wavefronts –1 that can run at the same time on a compute unit and stores the result in the destination *`dest`*. All compute units of an HSA component must support the same value for `maxwaveid`. For example, if a maximum of four wavefronts can execute at the same time on a compute unit, `maxwaveid` will be 3.

`nop`
A NOP (no operation). Used to leave space in an HSAIL program.

`nullptr`
Sets the destination *`dest`* to a value that is not a legal address within the segment. If *`segment`* is omitted, *`dest`* is set to the value of the null pointer value used by the host CPUs for flat addressing.

The value for each segment, and for flat addressing, is dependent on the host operating system. All agents must use the same values as the host CPUs.

The runtime will provide API calls so that agents can determine the value returned by `nullptr` for each segment.

`qid`
Returns a 32-bit queue identifier (queue ID) that is unique for the currently created queues within a process and stores the result in the destination *`dest`*. Queue IDs may be reused during the lifetime of the process, and are not guaranteed to be unique between processes.

The combination of the queue ID and the dispatch ID is globally unique. Debuggers might find this useful.

`qptr`
Sets the destination *`dest`* to to the global segment address of the AQL queue object on which the dispatch packet that invoked this kernel execution is queued. The format of the queue object is defined in the *HSA System Architecture Specification*.

`setdetectexcept`
Sets DETECT exception flags specified in *`exceptionsNumber`* for the work-group containing the current work-item. The result is undefined if used inside divergent control flow, and might lead to deadlock. See 12.3 Additional Information on DETECT Exception Operations (p. 222).

`waveid`
Returns an identifier (ID) for the wavefront on this compute unit, a number between 0 and `maxwaveid`, and stores the result in the destination *`dest`*.

For example, if a maximum of four wavefronts can execute at the same time on a compute unit, the possible `waveid` values will be 0, 1, 2, and 3.

The value is unique across all currently executing wavefronts on the same compute unit. The number will be reused when the wavefront is finished and a new wavefront starts.

Programs might use this value to address non-persistent global storage.

`workgroupid`
Accesses the work-group identifier (ID) within the grid.

This operation computes the three-dimensional ID of the work-group, selects one dimension, and stores the result in the destination *`dest`*.

If the grid was launched with fewer than three dimensions, `workgroupid` returns 0 for the unused dimensions.

`workgroupsize`
Accesses the work-group size specified when the kernel was dispatched for the *`src`* dimension (see 2.2 Work-Groups (p. 7)) and stores the result in the destination *`dest`*.

Because the grid is not required to be a multiple of the work-group size, there can be partial work-groups. If there can be partial work-groups, the `currentworkgroupsize` operation should be used to get the work-group size for the work-group that the currently executing work-item belongs to.

If it is known that the kernel is always dispatched without partial work-groups, then `currentworkgroupsize` and `workgroupsize` will always be the same, and it might be more efficient to use `workgroupsize`.

If the kernel was dispatched with fewer dimensions than *`src`*, then `workgroupsize` returns 1 for the unused dimensions.

`workitemabsid`
Accesses the work-item absolute identifier (ID) within the entire grid and stores the result in the destination *`dest`*.

If the work-group was launched with fewer dimensions than *`src`*, `workitemabsid` returns 0 for the unused dimensions.

`workitemflatabsid`
Accesses the flattened form of the work-item absolute identifier (ID) within the entire grid and stores the result in the destination *`dest`*.

`workitemflatid`
Accesses the flattened form of the work-item identifier (ID) within the work-group and stores the result in the destination *`dest`*.

`workitemid`
Accesses the work-item identifier (ID) within the work-group and stores the result in the destination *`dest`*.

If the work-group was launched with fewer dimensions than *`src`*, `workitemid` returns 0 for the unused dimensions.

12.3 Additional Information on DETECT Exception Operations

DETECT exception processing operates on the five exceptions specified in 13.2 Hardware Exceptions (p. 225)).

DETECT exception processing is performed independently for each work-group. Each work-group conceptually maintains a 5-bit `exception_detected` field which is initialized to 0 before any wavefront in the work-group starts executing. This field can be implemented in group memory and so might reduce the amount of memory available for group segment variables. However, an implementation is free to implement the semantics implied by the `cleardetectexcept`, `setdetectexcept`, and `getdetectexcept` operations in any way it chooses, including by using dedicated hardware.

If any of the five exceptions occurs in any work-item of the work-group, the bit corresponding to the exception is conceptually set in the `exception_detected` field.

The `cleardetectexcept`, `setdetectexcept`, and `getdetectexcept` operations conceptually operate on the `exception_detected` field, and their execution must be work-group uniform. If they are used inside of divergent control flow, the result is undefined, and might lead to deadlock. (See 2.13 Divergent Control Flow (p. 21).) These operations can be used in a loop, provided the loop introduces no divergent control flow. This requires that all work-items in the work-group execute the loop the same number of iterations.

The work-group `exception_detected` field is not implicitly saved when the work-items of the work-group complete execution. If the user wants to save the value, then explicit HSAIL code must be used. For example, the kernel might perform a `getdetectexcept` operation at the end and atomically `or` the result into a global memory location specified by a kernel argument. This will accumulate the results from all work-groups of a kernel dispatch.

When a kernel is finalized, the set of exceptions that are enabled for DETECT can be specified. In addition, they can be specified in the kernel by the `enabledetectexceptions` control directive. The exceptions enabled for DETECT is the union of both these sources.

If any function that the kernel calls, either directly or indirectly, has an `enabledetectexceptions` control directive that includes exceptions not specified by either the kernel's `enabledetectexceptions` control directive or the finalizer option, then it is undefined if those exceptions will be enabled for DETECT.

An implementation is only required to correctly report DETECT exceptions that were enabled when the kernel was finalized. It is implementation-defined if exceptions not enabled for DETECT when the kernel was finalized are correctly reported.

On some implementations, if one or more exceptions are enabled for DETECT, the code produced might have lower performance than if no exceptions were enabled for DETECT. However, an implementation should attempt to make the performance near that of a kernel finalized with no exceptions enabled for DETECT.

If any exceptions are enabled for the DETECT policy, there are some restrictions on the optimizations that are permitted by the finalizer. In general, the intent is that effective optimization can still be performed according to the optimization level specified to the finalizer (see 18.9 Exceptions (p. 255)).

Examples

```
clock_u64 $d6;              // return the current time
debugtrap_u32 $s1;          // halt and transfer control to debugger
cuid_u32 $s7;               // access the compute unit id within the
                            // HSA component
maxcuid_u32 $s6;            // access number of compute units on the
                            // HSA component
waveid_u32 $s3;             // access the wavefront ID within the
                            // HSA component
maxwaveid_u32 $s4;          // access the maximum number of waves
                            // that can be executing at the same
                            // time by the HSA component
dispatchid_u64 $d0;         // access the dispatch ID
dispatchptr_global_u64 $d0; // access the address of the
                            // AQL dispatch packet
laneid_u32 $s1;             // access the lane ID
qid_u32 $s0;                // access the queue ID
cleardetectexcept_u32 $s0;
getdetectexcept_u32 $s1;
setdetectexcept_u32 $s2;
nop; // no operation
nullptr_group_u32 $s0;    // null pointer value for group segment
nullptr_global_u64 $d1;   // null pointer value for global segment
gridsize_u32 $s2, 2;      // access the number of work-items in the
                          // grid Z dimension
gridgroups_u32 $s2, 2;    // access the number of work-groups in the
                          // grid Z dimension
workgroupsize_u32 $s1, 0; // access the number of work-items in the
                          // non-partial work-groups in the X dimension
currentworkgroupsize_u32 $s1, 0; // access the number of work-items in
                                 // the current work-group in the X
                                 // dimension, which might be partial
workitemabsid_u32 $s1, 0; // access the work-item absolute ID in the
                          // X dimension
workgroupid_u32 $s1, 0; // access the work-group ID in the X dimension
workgroupid_u32 $s1, 1; // access the work-group ID in the Y dimension
workgroupid_u32 $s1, 2; // access the work-group ID in the Z dimension
workitemid_u32 $s1, 0;  // access the work-item ID in the X dimension
workitemid_u32 $s1, 1;  // access the work-item ID in the Y dimension
workitemid_u32 $s1, 2;  // access the work-item ID in the Z dimension
```

Chapter 13

Exceptions

This chapter describes HSA exception processing.

13.1 Kinds of Exceptions

Three kinds of exceptions are supported:

- Hardware-detected exceptions such as divide by zero.

  See 13.2 Hardware Exceptions (p. 225).

- Software-triggered exceptions corresponding to the higher-level catch and throw operations.

  Software exceptions can be generated by `syscall`, but HSAIL provides no special operations for handling software exceptions.

- Debug-related exceptions generated by `debugtrap`.

  Debug exceptions normally invoke the debugger. If no debugger is present, they are handled as software exceptions.

13.2 Hardware Exceptions

HSAIL requires the hardware to generate certain exceptions, and provides a mechanism to control these exceptions by means of hardware exception policies (see 13.3 Hardware Exception Policies (p. 227)). These exceptions include the five floating-point exceptions specified in IEEE/ANSI Standard 754-2008. HSAIL also allows, but does not require, an implementation to generate a divide by zero exception if integer division or remainder with a divisor of zero is performed.

In addition, HSAIL allows, but does not require, an implementation to generate other exceptions, such as invalid address and memory exception. However, HSAIL does not provide support to control these exceptions by means of the HSAIL exception policies. It is therefore implementation-defined how they are handled.

The exceptions supported by the HSAIL exception policies are:

- Overflow

  The floating-point exponent of a value is too large to be represented.

  An operation that generates the overflow exception must also generate the inexact exception.

- Underflow

  A non-zero floating-point value is so small that it cannot be represented without an extraordinary loss of accuracy. The value can be represented only as zero or a subnormal number.

  An operation that generates the underflow exception must also generate the inexact exception if the result is zero or a subnormal value that is not the exact value.

- Division by zero

  A finite non-zero floating-point value is divided by zero.

  It is implementation defined if integer `div` or `rem` operations with a divisor of zero will generate a divide by zero exception.

- Invalid operation

  Operations are performed on values for which the results are not defined. These are:

  - Operations on signaling NaN (sNan) floating-point values.
  - Multiplication: `mul`(0.0, infinity) or `mul`(infinity, 0.0).
  - Fused multiply add: `fma`(0.0, infinity, c) or `fma`(infinity, 0.0, c) unless c is a quiet NaN, in which case it is implementation-defined if an exception is generated.
  - Addition, subtraction, or fused multiply add: magnitude subtraction of infinities, such as: `add`(positive infinity, negative infinity), `sub`(positive infinity, positive infinity).
  - Division: `div`(0.0, 0.0) or `div`(infinity, infinity).
  - Square root: `sqrt`(negative).
  - Conversion: A `cvt` with a floating-point source type, an integer destination type, and a nonsaturating rounding mode, when the source value is a NaN, infinity, or the rounded value, after any flush to zero, cannot be represented precisely in the integer type of the destination.

- Inexact

  A computed floating-point value cannot be represented exactly, so a rounding error is introduced.

  This exception is very common.

  An operation might generate both an underflow and inexact exception. An operation that generates an overflow exception will also generate an inexact exception.

13.3 Hardware Exception Policies

HSA supports DETECT and BREAK policies for each of the five exceptions specified in 13.2 Hardware Exceptions (p. 225):

- DETECT

  A compute unit must maintain a status bit for each of the five supported hardware exceptions for each work-group it is executing. All status bits are set to 0 at the start of a work-group. If an exception is generated in any work-item, the corresponding status bit will be set for its work-group. The `cleardetectexcept`, `getdetectexcept`, and `setdetectexcept` operations can be used to read and write the per work-group status bits.

  The DETECT policy is independent of the BREAK policy.

  In order that DETECT exceptions are correctly reported, it is necessary to specify them when the finalizer is invoked, or in an `enabledetectexceptions` control directive in the kernel. It is undefined if enabled DETECT exceptions are correctly updated if all external functions called directly or indirectly by the kernel are not also finalized with the exceptions enabled for the DETECT policy.

  Specifying DETECT exceptions to the finalizer might result in the code produced having lower performance. However, an implementation should attempt to make the performance reduction minimal.

  If any exceptions are enabled for the DETECT policy, there are some restrictions on the optimizations that are permitted by the finalizer. In general, the intent is that effective optimization can still be performed according to the optimization level specified to the finalizer (see 18.9 Exceptions (p. 255)).

  See 12.3 Additional Information on DETECT Exception Operations (p. 222).

- BREAK

  When an exception occurs and the BREAK policy is enabled for that exception, all compute units of the HSA component must stop execution of the HSAIL instruction stream for that kernel dispatch. It is required that if an instruction generates an exception that is not enabled for the BREAK exception policy, then it will not cause execution of the kernel dispatch to halt. Execution is halted at a machine instruction boundary, and this is not required to be at an HSAIL operation boundary. The machine instruction each wavefront is executing when the compute unit is halted is termed the halted machine instruction.

  The halted instruction for all work-items that executed a machine instruction that generated an exception that is enabled for the BREAK policy before the compute unit stopped execution must be the machine instruction that generated the exception. These are termed the excepting work-items. The machine instruction that generated the exception is termed the excepting machine instruction. The wavefronts containing the excepting work-items are termed the excepting wavefronts.

  The other wavefronts that are currently executing the kernel dispatch, but do not contain an excepting work-item, are termed non-excepting wavefronts. The work-items they contain are termed non-excepting work-items.

  For each of the excepting work-items, it is required that the machine state must be as if the excepting machine instruction had never executed. This includes updating of machine registers, writing to memory, setting the DETECT exception bits, and updating any other machine state. It is required to indicate the set of

excepting work-items, together with the set of exceptions each generated. A single excepting work-item may generate more than one exception. All exceptions enabled for the BREAK policy must be included. It is allowed, but not required, to include other exceptions that the excepting instruction generated.

All non-excepting work-items, whether in an excepting wavefront or non-excepting wavefront, that are enabled are required to behave as if either: they had not executed the halted machine instruction and therefore not modified machine state, including setting any DETECT exception status bits; or they had completed execution of the halted machine instruction and modified the machine state including any DETECT exception status bits. They are not allowed to only partially update the machine state.

For both excepting and non-excepting wavefronts, it is required to provide an indication of which enabled work-items in each wavefront have completed execution of the halted machine instruction, and which are as if they had not executed the halted machine instruction. It is allowed for a wavefront to have some enabled work-items that have completed, and some that have not completed, the halted machine instruction.

It is allowed for execution of the non-excepting wavefronts to proceed some number of machine instructions after an exception enabled for the BREAK policy has been generated by some other wavefront. Indeed, this execution might result in additional exceptions enabled for the BREAK policy being generated, and so result in additional excepting wavefronts. However, an implementation should attempt to keep the number of machine instructions executed by other wavefronts to a minimum in order to improve the precision of exception information presented by debuggers.

It is required that the machine state of an excepting work-item can be modified. This must include updating of machine registers, writing to memory, setting the DETECT exception bits, updating any other machine state, and changing the work-item to indicate that it is as if the excepting machine instruction had completed execution. It is required that execution can be resumed, which will result in all halted wavefronts continuing execution. For each wavefront, it is required that all enabled work-items that are as if the halted machine instruction had not been completed, will first complete execution of the halted machine instruction, before all enabled work-items in the same wavefront continue execution with the next machine instruction.

When the finalizer is invoked, or in an `enablebreakexceptions` control directive in the kernel, it must be specified which exceptions can be enabled for BREAK when it is dispatched. It is undefined if an exception enabled for BREAK when a kernel was finalized will correctly halt execution if it occurs, unless all external functions called directly or indirectly by the kernel are also finalized with that exception enabled for BREAK.

Specifying one or more exceptions might result in code that executes with lower performance.

If any exceptions are enabled for the BREAK policy, there are some restrictions on the optimizations that are permitted by the finalizer. In general, the intent is that effective optimization can still be performed according to the optimization level specified to the finalizer (see 18.9 Exceptions (p. 255)).

If an exception is generated that is not enabled for the BREAK policy, or if execution is resumed after having been halted due to generation of either the same or different exception that is enabled for the BREAK policy, then execution continues after updating of the DETECT status bit if the DETECT policy is enabled for that exception.

The operation generating the exception completes and produces the result specified for that exceptional case. Generating an exception does not affect execution unless the BREAK policy is enabled for that exception, and execution is not resumed, except for the side effect of updating the corresponding DETECT bit if the DETECT policy is enabled for that exception, or any side effects resulting from halting execution due to an exception enabled for the BREAK policy.

No HSAIL operations can be used to change which exceptions are enabled for the DETECT or BREAK policy: that can only be achieved at finalize time through the finalizer detect exceptions option, or an `enabledetectexceptions` control directive in the kernel.

Chapter 14

Directives

This chapter describes the directives.

See also 8.4 Label Targets (labeltargets Statement) (p. 183).

14.1 extension Directive

The `extension` directive enables additional opcodes that can be used in the compilation unit. It must appear after the `version` directive but before the first HSAIL declaration or definition. This allows a finalizer to identify all extensions by only inspecting the directives at the start of a compilation unit: it does not need to scan the entire compilation unit.

An `extension` directive applies to all kernels and functions in the compilation unit up to the next `version` directive or end of input.

The syntax is:

`extension` *`string`*

The string is the name of the extension.

For example, if a finalizer from a vendor named *foo* was to support an extension named *bar,* an application could enable it using code like this:

```
extension "foo:bar";
```

The string `"CORE"` specifies that no extensions are allowed:

```
extension "CORE";
```

If the `"CORE" extension` directive is present, the only other `extension` directives allowed in the same compilation unit are other `"CORE"` directives. Otherwise, multiple non-`"CORE" extension` directives are allowed in a compilation unit: a finalizer must enable all opcodes for all `extension` directives that specify the vendor of the finalizer for the compilation unit.

14.1.1 How to Set Up Finalizer Extensions

HSAIL opcodes are 32 bits in the binary format. Each extension uses the upper 16 bits as an identifier for the extension and the lower 16 bits to identify the specific opcode.

For example, assume that a particular finalizer named `xyz`has implemented an extension called `newext`. The finalizer could choose to number this extension target as extension `0x23`, with a `max3_f32` operation (with number `0x00230001`). The operation could return the maximum value of three floating-point inputs.

The code would be:

```
version 1:0:$full:$large;
extension "xyz:newext";
kernel &max3Vector(kernarg_u32 %A,
                   kernarg_u32 %B,
                   kernarg_u32 %C,
                   kernarg_u32 %D
                      )
{
   workitemabsid $s0, 0;  // s0 is the absolute ID
   mul_u32 $s0, $s0, 4; // 4* absolute ID (into bytes)

   ld_kernarg_u32 $s4, [%A];
   add_u32 $s1, $s0, $s4;
   ld_global_f32 $s10, [$s1];

   ld_kernarg_u32 $s4, [%B];
   add_u32 $s1, $s0, $s4;
   ld_global_f32 $s11, [$s1];

   ld_kernarg_u32 $s4, [%C];
   add_u32 $s1, $s0, $s4;
   ld_global_f32 $s12, [$s1];

// The finalizer supports new opcode:
   newext_max3_f32 $s11, $s10, $s11, $s12;

   ld_kernarg_u32 $s4, [%D];
   add_u32 $s10, $s0, $s4;
   st_global_f32 $s10, [$s10];
   ret;
};
```

If the finalizer does not support the extension, the program should fail to finalize at run time.

14.2 Block Section Directives for Debugging and Runtime Information

HSAIL allows a compiler to embed information for either a debugger or a runtime library inside an HSAIL program. The embedded information is contained in a block section containing directives.

Blocks can appear in either of these places:

- Inside a function or kernel. (These blocks apply to the function or kernel.)
- At the top level. (These blocks apply to all following code.)

More than one block section can appear in a program.

14.2.1 Syntax for a Block Section

Syntax is:

```
block string
series of one or more blockstring or blocknumeric directives;
endblock;
```

An example is:

```
block "rti"
blockstring "meta info about this function";
endblock;
```

The string on the `block` directive can be either `"debug"` (for debugger information) or `"rti"` (for runtime information).

The `blockstring` directive contains a string in quotes specifying the information to be passed.

The `blocknumeric` directive has the following format:

- `blocknumeric_b8` — Comma-separated list of byte values
- `blocknumeric_b32` — Comma-separated list of integer values
- `blocknumeric_b64` — Comma-separated list of integer values

The `endblock` directive ends the block.

Because the `block` directive starts a block of statements, it does not have a trailing semicolon, but the other statements in the block do.

Blocks labeled with `"debug"` are placed in the BRIG `.debug` section. See 19.9 .debug Section (p. 314).

Blocks labeled with `"rti"` are placed in the BRIG `.directive` section. See 19.6 .directive Section (p. 281).

For information about BRIG syntax, see 19.5 Block Sections in BRIG (p. 279).

14.2.2 Example of a Block Section for Debug Data

This example shows a block section for debug data:

```
block "debug"
blocknumeric_b8 255, 23, 10, 23;
blocknumeric_b32 1255, 0x323, 10, 23;
blocknumeric_b64 0x123456781, 0x323, 10, 23;
blockstring "this is a string";
endblock;
```

14.2.3 Using a Block Section for Runtime Information

Examples of why a compiler might use a block section include:

- To let the runtime know that it needs to treat some argument in a special way.
- To hold reflection information.
- To hold information about configuration control.

14.2.4 Example of a Block Section for Runtime Data

Example:

```
version 1:0:$full:$large;
global_u32 &a  = 0;
block "rti"
blockstring "meta info global";
endblock;

static global_u32 &c;
function &foo () (align 8 arg_b8 %z[12])
{
    block "rti"
    blockstring "meta info about this function";
    endblock;
    ret;
};

kernel &square (kernarg_u32 %___square__a)
{
    block "rti"
    blockstring "meta info about this kernel";
    endblock;
    ret;
};
```

14.3 file Directive

The `file` directive is used to map numbers to character strings. The numbers can then be used in `loc` statements to indicate that code following the `loc` statement originated in the file whose name is in the character string.

The syntax is:

```
file number string
```

number is an index into a table for use by later `loc` statements. *number* must be a positive integer that is unique to all other `file` directives.

string is a file name surrounded by quotes. An empty file name also requires quotes.

See the following example:

```
file 1 "this is a file";
```

See also 14.4 loc Directive (p. 234).

14.4 loc Directive

Use the `loc` directive to specify the line number in the source file for the HSAIL code.

The syntax is:

```
loc filenum linenum [column] [options]
```

filenum is the number on a prior `file` directive.

linenum is the line number within that file.

column is an optional column within the line.

options are debug options. Currently, options must be 0; later releases of HSAIL will support additional options.

For example:

```
loc 1 20 0;  // file 1 line 20
```

See also 14.3 file Directive (p. 234).

14.5 pragma Directive

The `pragma` directive can be used to pass a string of information to the finalizer.

The syntax is:

```
pragma string
```

The `pragma` directive can appear anywhere in the HSAIL code.

If the pragma applies to a kernel or function, then it must be placed in the kernel or function scope, and only applies to that kernel or function. This allows the finalizer to locate all pragmas for a kernel or function without having to read all file scope directives. It also allows an HAIL linker to process functions independently, because no pragmas outside the function can alter its behavior.

The finalizer implementation defines rules for what portion of the kernel or function the pragma applies to and what happens if the same pragma appears multiple times.

The finalizer implementation determines the interpretation of `pragma` strings.

You cannot use this directive to change the semantics of the HSAIL virtual machine.

14.6 Control Directives for Low-Level Performance Tuning

HSAIL provides control directives to allow implementations to pass information to the finalizer. These directives are used for low-level performance tuning. See Table 14–1 (p. 235).

Table 14–1 Control Directives for Low-Level Performance Tuning

Directive	Arguments
`enablebreakexceptions`	*exceptionsNumber*
`enabledetectexceptions`	*exceptionsNumber*
`maxdynamicgroupsize`	*size*
`maxflatgridsize`	*count*
`maxflatworkgroupsize`	*count*
`requestedworkgroupspercu`	*nc*

Directive	Arguments
`requireddim`	*nd*
`requiredgridsize`	*nx*, *ny*, *nz*
`requiredworkgroupsize`	*nx*, *ny*, *nz*
`requirenopartialworkgroups`	

Explanation of Operands
`exceptionsNumber`: Source that specifies the set of exceptions. bit:0=INVALID_OPERATION, bit:1=DIVIDE_BY_ZERO, bit:2=OVERFLOW, bit:3=UNDERFLOW, bit:4=INEXACT; all other bits are ignored. Must be an immediate value of data type `u32`.
`size`: The number of bytes. Treated as data type `u32`. Must be an immediate value.
`count`: The number of work-items. Treated as data type `u32`. Must be an immediate value greater than 0, or `WAVESIZE`.
nc: The number of work-groups. Treated as data type `u32`. Must be an immediate value greater than 0.
nd: The number of dimensions. Treated as data type `u32`. Must be an immediate value with the value 1, 2 or 3.
nx, *ny*, *nz*: The work-group range. Treated as data type `u32`. Must be an immediate value greater than 0, or `WAVESIZE`.

See also 19.2.6 BrigControlDirective (p. 262).

The control directives must appear in the code block of a kernel, function, or argument scope, and only apply to that kernel or function, and possibly all the functions it calls directly or indirectly. This allows the finalizer to locate all control directives for a kernel or function without having to read all file scope directives. It also allows an HAIL linker to process functions independently, because no control directives outside the function can alter its behavior.

The rules for what portion of the kernel or function the control directive applies to, and what happens if the same control directive appears multiple times, or in functions called by the code block, are specified by each control directive.

If the runtime library also supports arguments for the limits specified by the directives, the directives take precedence over any constraints passed to the finalizer by the runtime.

`enablebreakexceptions`

Specifies the set of exceptions that must be enabled for the BREAK policy (see 13.3 Hardware Exception Policies (p. 227)). *`exceptionsNumber`* must be an immediate value of data type `u32`. The bits correspond to the exceptions as follows: bit 0 is INVALID_OPERATION, bit 1 is DIVIDE_BY_ZERO, bit 2 is OVERFLOW, bit 3 is UNDERFLOW, bit 4 is INEXACT, and other bits are ignored. It can be placed in either a kernel or a function code block.

The exceptions enabled for the BREAK policy is the union of the exceptions specified by all the `enablebreakexceptions` control directives in the kernel code block and the enable break exceptions argument specified when the finalizer is invoked. The setting applies to the whole kernel and all functions it calls in the same compilation unit.

If the functions called directly or indirectly by the kernel contain `enablebreakexceptions` control directives, then it is undefined if exceptions specified in them are enabled if they are not also enabled by the kernel or finalizer option.

It is undefined if enabled BREAK exceptions are correctly updated in functions called directly or indirectly by the kernel in other compilation units, unless they contain `enablebreakexceptions` control directives or the finalizer was invoked specifying them in the enable break exceptions argument.

`enabledetectexceptions`

Specifies the set of exceptions that must be enabled for the DETECT policy (see 13.3 Hardware Exception Policies (p. 227)). *`exceptionsNumber`* must be an immediate value of data type `u32`. The bits correspond to the exceptions as follows: bit 0 is INVALID_OPERATION, bit 1 is DIVIDE_BY_ZERO, bit 2 is OVERFLOW, bit 3 is UNDERFLOW, bit 4 is INEXACT, and other bits are ignored. It can be placed in either a kernel or a function code block.

The exceptions enabled for the DETECT policy is the union of the exceptions specified by all the `enabledetectexceptions` control directives in the kernel code block and the enable detect exceptions argument specified when the finalizer is invoked. The setting applies to the whole kernel and all functions it calls in the same compilation unit.

If the functions called directly or indirectly by the kernel contain `enabledetectexceptions` control directives, then it is undefined if exceptions specified in them are enabled if they are not also enabled by the kernel or finalizer option.

It is undefined if enabled DETECT exceptions are correctly updated in functions called directly or indirectly by the kernel in other compilation units, unless they contain `enabledetectexceptions` control directives or the finalizer was invoked specifying them in the enable detect exceptions argument.

`maxdynamicgroupsize`

Specifies the maximum number of bytes of dynamic group memory (see 4.24 Dynamic Group Memory Allocation (p. 62)) that will be allocated for a dispatch of the kernel. *`size`* must be an immediate value of data type `u32` with a value greater than or equal to 0. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

This value can be used by the finalizer to determine the maximum number of bytes of group memory used by each work-group. The finalizer can add this value to the group memory required for all group segment variables used by the kernel and all functions it calls and to the group memory used to implement other HSAIL features such as fbarriers and the detect exception operations. This can allow the finalizer to determine the expected number of work-groups that can be executed by a compute unit and allow more resources to be allocated to the work-items if it is known that fewer work-groups can be executed due to group memory limitations. This can also allow the finalizer to determine that there is free group memory that it can use for other purposes such as spilling.

The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same value.

If the value for maximum dynamic group size is specified when the finalizer is invoked, it must match the value given in any `maxdynamicgroupsize` control directive.

`maxflatgridsize`

Specifies the maximum maximum number of work-items that will be in the grid when the kernel is dispatched. `count` must be an immediate value of data type `u32` with a value greater than 0, or `WAVESIZE`. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched with a grid range when the product of the X, Y, and Z elements of the grid range is greater than this value. A finalizer might be able to generate better code for the `workitemabsid`, `workitemflatid`, and `workitemflatabsid` operations if the absolute grid size is less than $2^{24}-1$, because faster `mul24` operations can be used. The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same values.

If the value for maximum absolute grid size is specified when the finalizer is invoked, the value must be less than or equal to the corresponding value given in any `maxflatgridsize` control directive, and will override the control directive value. The value specified must also be greater than or equal to the product of the values specified by `requiredgridsize`.

`maxflatworkgroupsize`

Specifies the maximum number of work-items that will be in the work-group when the kernel is dispatched. `count` must be an immediate value of data type `u32` with a value greater than 0, or `WAVESIZE`. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched when the product of the X, Y, and Z elements of the work-group range is greater than this value. A finalizer might be able to generate better code for barriers if it knows that the work-group size is less than or equal to the wavefront size. A finalizer might be able to generate better code for the `workitemflatid` operation if the total work-group size is less than $2^{24}-1$, because faster `mul24` operations can be used. The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same values.

If the value for maximum absolute work-group size is specified when the finalizer is invoked, the value must be less than or equal to the corresponding value given by any `maxflatgroupsize` control directive, and will override the control directive value. The value specified must also be greater than or equal to the product of the values specified by `requiredworkgroupsize`.

`requestedworkgroupspercu`

Specifies the desired number of work-groups that can execute on a single compute unit. *nc* must be an immediate value of data type `u32` with a value greater than 0. It can be placed in either a kernel or a function code block. The finalizer should attempt to generate code that will meet this request. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

This can be used by the finalizer to determine the number of resources that should be allocated to a single work-group and work-item. For example, a low value might allow more resources to be allocated, resulting in higher per work-item performance, as it is known there will never be more than the specified number of work-groups actually executing on the compute unit. Conversely, a high value might allocate fewer resources, resulting in lower per work-item performance, which is offset by the fact that it allows more work-groups to actually execute on the compute unit.

The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same value.

If the value for requested work-groups per compute unit is specified when the finalizer is invoked, the value must match the value given in any `requestedworkgroupspercu` control directive.

`requireddim`

Specifies the number of dimensions that will be used when the kernel is dispatched. *nd* must be an immediate value of data type `u32` with the value 1, 2, or 3. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched with a dimensions value that does not match the required dimension.

With the use of this operation, a finalizer might be able to generate better code for the `workitemid`, `workitemabsid`, `workitemflatid`, and `workitemflatabsid` operations, because the terms for dimensions above the value specified can be treated as 1.

The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same value.

If `requireddim` is specified (either by a control directive or when the finalizer was invoked), it must be consistent with `requiredgridsize` and `requiredworkgroupsize` if specified: if the value is 1, then their Y and Z dimensions must be 1; if 2, then their Z dimension must be 1.

If the value for required dimensions is specified when the finalizer is invoked, the value must match the value in any `requireddim` control directive.

`requiredgridsize`

Specifies the grid size that will be used when the kernel is dispatched. The X, Y, Z dimensions of the range correspond to *nx*, *ny*, *nz* respectively. They must be an immediate value of data type `u32` with a value greater than 0, or `WAVESIZE`. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched with a grid range that does not match these dimensions. A finalizer might be able to generate better code for the `gridsize` operation. Also, if the total grid size is less than $2^{24}-1$, then faster `mul24` operations might be able to be used for the `workitemid`, `workitemabsid`, `workitemflatid`, and `workitemflatabsid` operations, because the terms for dimensions above the value specified can be treated as 1. In conjunction with `requiredworkgroupsize`, a finalizer might also be able to generate better code for `gridgroup`and `currentworkgroupsize` operations (because it can determine if there are any partial work-groups).

The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same values.

If the values for required grid size are specified when the finalizer is invoked, they must match the corresponding values given in any `requiredgridsize` control directive. The product of the values must also be less than or equal to any value specified by `maxflatgridsize`.

`requiredworkgroupsize`

Specifies the work-group size that will be used when the kernel is dispatched. The X, Y, Z dimensions of the range correspond to *nx*, *ny*, *nz* respectively. They must be an immediate value of data type `u32` with a value greater than 0, or `WAVESIZE`. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched with a work-group range that does not match these dimensions.

A finalizer might be able to generate better code for barriers if it knows that the work-group size is less than or equal to the wavefront size. This directive might also allow better code for the `workgroupsize`, `workitemid`, `workitemabsid`, `workitemflatid`, and `workitemflatabsid` operations.

The control directive applies to the whole kernel and all functions it calls. If multiple control directives are present in the kernel or the functions it calls, they must all have the same values.

If the values for required work-group size are specified when the finalizer is invoked, they must match the corresponding values given in any `requiredworkgroupsize` control directive. The product of the values must also be less than or equal to any value specified by `maxflatworkgroupsize` .

`requirenopartialworkgroups`

Specifies that the kernel must be dispatched with no partial work-groups. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

It is undefined if the kernel is dispatched with any dimension of the grid size not being an exact multiple of the corresponding dimension of the work-group size.

A finalizer might be able to generate better code for `currentworkgroupsize` if it knows there are no partial work-groups, because the result becomes the same as the `workgroupsize` operation. An HSA component might be able to dispatch a kernel more efficiently if it knows there are no partial work-groups.

The control directive applies to the whole kernel and all functions it calls. It can appear multiple times in a kernel or function. If it appears in a function (including external functions), then it must also appear in all kernels that call that function (or have been specified when the finalizer was invoked), either directly or indirectly.

If require no partial work-groups is specified when the finalizer is invoked, the kernel behaves as if the `requirenopartialworkgroups` control directive has been specified.

Chapter 15

version Statement

This chapter describes the `version` statement.

15.1 Syntax of the version Statement

The `version` statement specifies the HSAIL version of the code and gives the attributes of the required target architecture.

A single `version` statement must appear at the top of each program. Duplicate `version` statements are allowed, which enables programs to be concatenated.

Each additional `version` statement must have the same major and minor numbers as the previous statement.

Compilation scope ends at a statement with a `version` statement or at the end of the compilation unit.

The syntax is:

```
version major : minor : profile : machine_model
```

`major`
Specifies that major version changes are incompatible and that this stream of operations can only be compiled and executed by systems with the same major number.

Major number changes are incompatible, so a kernel or function compiled with one major number cannot call a function compiled with a different major number.

`minor`
Specifies that this stream of operations can only be compiled and executed by systems with the same or larger minor number.

Minor number changes correspond to added functionality. Minor changes are compatible, so kernels or functions compiled at one minor level can call functions compiled at a different minor level, provided the implementation supports both minor versions.

`profile`
Specifies which profile is used during finalization (see Chapter 17 Profiles (p. 249)). Possibilities are:

- `$base` — The Base profile should be used. Inclusion of this option indicates that the associated HSAIL uses or requires features of the Base profile.
- `$full` — The Full profile should be used. Inclusion of this option indicates that the associated HSAIL uses or requires features of the Full profile.

For more information, see Chapter 17 Profiles (p. 249).

machine_model

Specifies which machine model is used during finalization (see 2.10 Small and Large Machine Models (p. 20)). Possibilities are:

- `$large` —Specifies large model, in which all flat and global addresses are 64 bits.
- `$small` — Specifies small model, in which all flat and global addresses are 32 bits. A legacy host CPU application executing in 32-bit mode might want program data-parallel sections in small mode.

For more information, see 2.10 Small and Large Machine Models (p. 20).

It is a linker error for multiple files to have different major version numbers, different profiles, or different machine models and to attempt to link to the same executable.

Examples

```
version 1:0: $full : $large;
version 1:0: $base : $small;
```

Chapter 16

Libraries

This chapter describes how to write HSAIL code for libraries.

16.1 Library Restrictions

HSAIL provides support for separately compiled libraries.

Code written for a library has the following restrictions:

- Every externally callable routine in the library should have external linkage.
- Every non-externally-callable routine in the library should have static linkage.
- Every HSAIL source file that contains a call to a library should have a declaration specifying `extern` for each library function that it will call.

16.2 Library Example

An example of library code is shown below:

```
group_f32 &xarray[100]; // the library gets part of this array
extern function &libfoo(arg_u32 %res) (arg_u32 %sptr);
static function &a()(arg_u32 %formal);

kernel &main()
{
   {
      arg_u32 %in;
      // give the library part of the group memory
      lda_group_u32 $s1, [&xarray][4];
      st_arg_u32 $s1, [%in];
      arg_u32 %out;
        call &libfoo(%out)(%in);
        ld_arg_u32 $s2, [%out];
   }
   {
      arg_u32 %in1;
      st_arg_u32 $s2, [%in1];
      call &a(%in1);
      // $s2 has the library call result
   }
   // ...
};

static function &a ()(arg_u32 %formal)
{
   // get the result of the library call
   ld_arg_u32 $s1, [%formal];
   // ...
};

// now for the second compile unit - the library

static function &l1()(arg_u32 %input);
function &libfoo(arg_u32 %res) (arg_u32 %sptr)

{
   ld_arg_u32 $s1, [%sptr];
   ld_group_u32 $s2, [$s1];    // library reads some group data
   st_group_u32 $s2, [$s1+4];  // library reads some group data
   {
      arg_u32 %s;

      // give a function in the library part of the shared array
      add_u32 $s4, $s2, 20;
      st_arg_u32 $s2, [%s];
      call &l1(%s);
    }
    // ...
};

static function &l1()(arg_u32 %input)
{
   ld_arg_u32 $s6, [%input];
   // library passed address in group memory is now $s6
```

```
// ...
};
```

Chapter 17

Profiles

This chapter describes the HSAIL profiles.

17.1 What Are Profiles?

HSAIL provides two kinds of profiles:

- Base
- Full

HSAIL profiles are provided to guarantee that the implementation supports a required feature set and meets a given set of program limits. The strictly defined set of HSAIL profile requirements provides portability assurance to users that a certain level of support is present.

The Base profile indicates that an implementation targets smaller systems that provide better power efficiency without sacrificing performance. Precision is possibly reduced in this profile to improve power efficiency.

The Full profile indicates that an implementation targets larger systems that have hardware that can guarantee higher-precision results without sacrificing performance.

The following rules apply to profiles:

- A finalizer can choose to support either or both profiles.
- A single profile applies to the entire compilation unit.
- An application is not allowed to mix profiles.
- The required profile must be selected by a modifier on the `version` statement. See 15.1 Syntax of the version Statement (p. 243).
- Both the large and small machine models are supported in each profile.
- The profile applies to all declared options.

Both profiles are required to support the following:

- Ability to load, store and convert all of the floating-point types (`f16`, `f32`, `f64`).
- The `f16` type and all operations on the type. `f16` precision requirements are a minimum requirement. Implementations can optionally provide additional precision and range when computing `f16` values in `s` registers. `f16` results are not required to be bit-reproducible across different HSA implementations (see 4.21 Floating-Point Numbers (p. 55)).

- All 32-bit floating-point (`f32`) operations according to the declared profile.
- Handling of the `syscall` and `debugtrap` exceptions.

The runtime library should provide a mechanism that enables an application to determine which features are available.

Both profiles are required to support all HSAIL requirements, except as specified in 17.2 Profile-Specific Requirements (p. 250).

See Appendix B Limits (p. 341) for details on limits that apply to both profiles.

17.2 Profile-Specific Requirements

This section describes the requirements that an implementation must adhere to in order to claim support of the Full or Base profile.

17.2.1 Full Profile Requirements

Implementations of the Full profile are required to provide the following support:

- On all supported floating-point types:
  - Must provide IEEE/ANSI Standard 754-2008 a correctly rounded result for add/subtract/multiply/divide/fma and square root operations.
  - Must support all 64-bit floating-point (`f64`) operations.
  - Must support all four HSAIL-defined rounding modes.
  - Must support floating-point subnormal values.
  - Must support the `ftz` modifier and IEEE/ANSI Standard 754-2008 gradual underflow.
  - Must generate invalid operation exceptions for signaling NaN sources. In general, NaN payloads and sign must be preserved, and signaling NaNs must be converted to silent NaNs.
- Full exception handling feature set:
  - Must support the DETECT and BREAK exception policies (see 13.3 Hardware Exception Policies (p. 227)) for the five exceptions specified in 13.2 Hardware Exceptions (p. 225).

17.2.2 Base Profile Requirements

Implementations of the Base profile are required to provide the following support:

- On all supported floating-point types:
  - Must provide IEEE/ANSI Standard 754-2008 a correctly rounded result for add/subtract/multiply/fma operations.
  - Must provide div operations within 2.5 ULP (unit of least precision) of the mathematically accurate result.
  - Must provide square root operations within 1 ULP of the mathematically accurate result.
  - Must follow these rounding mode rules: All floating-point operations (except `cvt`) that support the floating-point rounding mode must only support the `near` rounding mode. The `cvt` operation from a floating-point type to a smaller floating-point type, and from integer type to floating-point type, must only support the `near` rounding mode. The `cvt` operation from floating-point type to integer type must only support `zeroi` and `zeroi_sat` (which correspond to the standard floating-point to integer conversion of C).
  - Must flush subnormal values to zero. All HSAIL floating-point operations must specify the `ftz` modifier (when `ftz` is valid).
  - Must generate invalid operation exceptions for signaling NaN sources. In general, NaN sign must be preserved, and signaling NaNs must be converted to silent NaNs. However, it is not expected to preserve NaN payloads.
- Minimal exception handling feature set:
  - Must support the DETECT exception policy (see 13.3 Hardware Exception Policies (p. 227)) for the exceptions specified in 13.2 Hardware Exceptions (p. 225).

Chapter 18

Guidelines for Compiler Writers

This chapter provides guidelines for compiler writers.

18.1 Register Pressure

The most important optimization for a high-level compiler is to minimize register pressure.

Code should be scheduled to use as few registers as possible. On the other hand, it is often important to try to move memory operations together either by using the vector forms (`v2`, `v3`, and `v4`) or by making loads and stores consecutive. Each high-level compiler will have to approach this carefully.

High-level compilers should use the spill segment to hold register spills, because the finalizer might be able to deploy extra registers and remove the spills.

18.2 Using Lower-Precision Faster Operations

When a source language permits, for example by means of a fast math compiler option, a high-level compiler can use faster but lower-precision substitutions for slower operations. For example, `div(`*`src0`*`, `*`src1`*`)` could be replaced by *`src0`* `* nrcp(`*`src1`*`)` whenever the lower precision is permitted.

18.3 Functions

Functions are often quite expensive. High-level compilers should inline whenever possible.

Inlining is critical because the finalizer might assume that all functions are recursive and allocate significant arg space. Given that a typical HSAIL implementation is able to execute thousands of work-items simultaneously, programs with functions can frequently run out of arg space.

Common performance ratios might be: one "call" takes as long as 1000 "adds," one indirect call takes as long as 10,000 "adds."

A simple high-level compiler could give up some performance by using an array for a stack as:

```
private_b8 %stacklike[8];
lda_private_u32 $s4, [%stacklike]; // the stack pointer

// Fill in the arguments
st_private_f32 4f, [$s4];
add_u32 $s4, $s4, 4;
st_private_f32 $s0, [$s4];
add_u32 $s4, $s4, 4;
{
   arg_u32 %sp;
   st_arg_u32 $s4, [%sp];
   call &stackfn (%sp)(%sp);
   ld_arg_u32 $s4, [%sp];
}
add_u32 $s4, $s4,-8;

// More code

};

function &stackfn (arg_u32 %sp) (arg_u32 %sp1)
{
   ld_arg_u32 $s4, [%sp];
   add_u32 $s4, $s4, 4;
   st_private_u32 $s1, [$s4];
   ld_private_b32 $s0, [$s4-8];
   ld_private_b32 $s0, [$s4-4];
   cmp_eq_b32_f32 $s0, $s1, $s0;
   ld_private_u32 $s1, [$s4];
   ret;
};
```

18.4 Frequent Rounding Mode Changes

Some implementations might choose to change the rounding mode of floating-point operations by changing the value of some state register. This might require flushing the floating-point pipeline, which can be quite slow. On such implementations, frequent changes of IEEE/ANSI Standard 754-2008 rounding modes can be very slow. Compilers are advised to group floating-point operations so that operations with the same mode are adjacent when possible.

18.5 Wavefront Size

Some applications might be able to maximize performance with knowledge of the wavefront size. Tool developers need to be careful about wavefront size assumptions, because programs coded for a single wavefront size might generate wrong answers if they are executed on machines with a different wavefront size.

Considering that wavefronts are important to get maximal performance but are not necessary to ensure correct results, you should, as a general rule, try to avoid control flow divergence. Work-items in a wavefront are numbered consecutively, so this

could be achieved by trying to code kernels so that consecutive work-items take the same path.

This is similar to the need to write cache-aware code for best performance on a CPU.

18.6 Unaligned Access

While HSAIL supports unaligned accesses for loads and stores, these are quite expensive and should be avoided. Unaligned accesses are not atomic, and atomic and atomic no return operations do not support unaligned access.

If a load or store is known to be naturally aligned, it should be marked with the `aligned` modifier. This might allow the finalizer to generate more efficient code on some implementations. A front-end compiler may be able to determine this either due to restrictions in the language it is compiling, or by analysis based on variable allocation. However, incorrectly marked aligned memory accesses might result in undefined results and generate memory exceptions on some implementations.

18.7 When to Use Flat Addressing

In general, segment addressing is faster than flat-address addressing, because a 32-bit register is used to hold the address. However, a segment is limited to 4 GB in size.

A high-level compiler should attempt to identify where a segment address should be used and avoid unnecessary 64-bit addressing operations.

18.8 Arg Arguments

While the calling convention allows arg arguments, every finalizer has the option to pass some of the arguments in high-speed machine registers. High-level compiler developers should read the microarchitecture guide for the chip for details.

18.9 Exceptions

If any exceptions are enabled for the BREAK policy (see 13.3 Hardware Exception Policies (p. 227)), there are some restrictions on the optimizations that are permitted by the finalizer. In general, however, the intent is that effective optimizations can still be performed according to the optimization level specified to the finalizer.

For exceptions enabled for the BREAK or DETECT policy, the finalizer should ensure that optimizations do not result in generating exceptions that would not have happened without the optimization, or in eliminating exceptions that would have been generated for non-dead code had the optimization not been done. However, optimization is allowed to change the order and number of enabled exceptions that are generated.

For example, for exceptions enabled for the BREAK or DETECT policy:

- A set of instructions that produce a result used in a visible result that can generate an exception cannot be transformed into a set of instructions that produce the same result but do not generate the exception. However, such transformations are allowed if the exception generated is not enabled for the BREAK or DETECT policy. For example, a divide by the immediate 0.0 could be folded to a multiply by +infinity if the divide by zero exception is not enabled.
- It is allowed to eliminate instructions that are dead, even if they could generate enabled exceptions. Namely, it is not necessary to prevent eliminating code whose only (side) effect is to cause an exception. Operations such as `syscall` and `debugtrap`, whose sole purpose is to generate an exception, must always be preserved if in reachable code.
- Instruction reordering is allowed to change the order of exceptions, as long as all enabled exceptions will still happen at least once. This allows transformations such as constant expression elimination, expression reassociation, and folding to be performed which can change the order that exceptions are generated, and can result in the same exception being generated fewer times. These optimizations are important to achieve performance comparable to code being executed without exceptions enabled.
- Code hoisting out of a loop and partial redundancy elimination, which can cause an exception where there previously was none, must not be permitted. For example, hoisting a loop invariant expression out of a loop, where the expression could cause an exception, must be guarded to ensure it is not executed if the loop count is 0. However, it should still be legal to hoist the expression provided it is guarded, which will also change both the order and number of times that exceptions can be generated.

Chapter 19

BRIG: HSAIL Binary Format

This chapter describes BRIG, the HSAIL binary format.

19.1 What Is BRIG?

The BRIG representation describes all aspects of the textual representation of HSAIL except:

- The textual layout.
- The textual format used to define constants and offsets. It just describes the value required by the operation, which may be truncated from the textual value specified.
- The use of explicit modifier values that are the default value used when the modifier is omitted (such as for `width`, `equiv`, `near` or `zeroi` rounding mode, and `fboth` memory fence modifier).
- The use of initializers to specify the size of an array. The textual form of HSAIL allows the size of an array to be omitted from a variable definition if it has an initializer, in which case it defaults to the number of elements in the initializer. In BRIG, the variable definition is represented as if it had been explicitly declared with a size.
- The options for the `version` statement when they do not need to be specified.

The finalizer is not required to read text format programs directly.

19.1.1 BRIG Sections

BRIG format is made up of five sections:

- `.string` — String section, containing all character strings and byte data used in the compilation unit. See 19.4 .string Section (p. 278).
- `.directive` — The directives, which provide information for the finalizer. The directives do not generate code. See 19.6 .directive Section (p. 281).
- `.code` — All of the executable operations. Most operations contain offsets to the `.operand` section. See 19.7 .code Section (p. 296).
- `.operand` — The operands, such as immediate constants, registers, and address expressions, that appear in the operations. See 19.8 .operand Section (p. 309).
- `.debug` — The debug information generated by the high-level compiler. The finalizer does not modify the `.debug` section. See 19.9 .debug Section (p. 314).

HSAIL does not specify ways to read and write these sections and any associated header information.

As an example, a pragma is a directive containing:

- An offset to the code (the byte offset where the pragma applies).
- An offset to the `.string` section. The way the finalizer processes the HSAIL code after the code offset may be affected by the contents of the string.

19.1.2 Format of Entries in the Sections

Every section starts with a `BrigSectionHeader` that contains the size of the section (see 19.3 Section Header (p. 277)).

The `BrigSectionHeader` is followed by the entries of the section with no gaps between each entry. Every entry is a multiple of four bytes, so every entry starts on a 4-byte boundary.

The largest type used in all BRIG structures is 32 bits, so every entry is naturally aligned. There must be no bytes after the last entry of a section and the end of the section. All entries in the `.directive`, `.code`, and `.operand` sections have a similar format.

Entries are variable-size. Each entry starts with a 16-bit unsigned integer containing the length of the entry in bytes, followed by a 16-bit kind field indicating the type of the entry.

While knowledge of the kind of an entry would enable the finalizer to calculate the length in most cases, the length is described explicitly. This is because future expansion of BRIG directives might add additional fields at the end of entries. The use of a length field will allow old finalizers to process new BRIG sections (ignoring any new fields).

All entries in the `.string` section consist of a 32-bit unsigned integer containing the number of bytes of data, then the bytes of the data, followed by enough zero-pad bytes to make the entry a multiple of 4.

BRIG structures are accessible in C style using structs. (C++ classes are not used.)

All BRIG values are stored in little endian format.

A number of BRIG structures (for example, `BrigDirectiveLabelList` and `BrigDirectiveSignature`, among others) are variable-sized entities. Such structures (except those in the `.string` section) have an element count in addition to the entry size. Variable-size BRIG entries might need to be padded with trailing zeros to reach a length that is a multiple of 4 bytes. In this case, the entry size is the total number of bytes in the entry including the padding bytes, and the element count contains the number of data items actually used.

In BRIG, both operations and operands are typed. Some operations have different types for their operands, so this makes it easy for a finalizer to determine an operand's type, and allows for validation tools. The types must match or the code is in error. In particular, the required size of the immediate integer operand is defined by the type of the operation where it is used.

19.2 Support Types

This section defines the various types and enumerations used in the structures present in each BRIG section.

19.2.1 Section Offsets

These types are used to reference a structure in a specific section. The value is the byte offset relative to the start of the section to the beginning of the referenced structure. The value 0 is reserved to indicate that the offset does not reference any structure:

```
typedef uint32_t BrigDirectiveOffset32_t;
typedef uint32_t BrigCodeOffset32_t;
typedef uint32_t BrigOperandOffset32_t;
typedef uint32_t BrigStringOffset32_t;
```

19.2.2 Section Structure Kinds

The `.directive`, `.code`, and `.operand` sections can each have a number of kinds of structures. The following enumerations are used to identify the kind of structure:

- `BrigDirectiveKinds` is used to specify the kind of `.directive` section structure:

```
typedef uint16_t BrigDirectiveKinds16_t;
enum BrigDirectiveKinds {
  BRIG_DIRECTIVE_ARG_SCOPE_END = 0,
  BRIG_DIRECTIVE_ARG_SCOPE_START = 1,
  BRIG_DIRECTIVE_BLOCK_END = 2,
  BRIG_DIRECTIVE_BLOCK_NUMERIC = 3,
  BRIG_DIRECTIVE_BLOCK_START = 4,
  BRIG_DIRECTIVE_BLOCK_STRING = 5,
  BRIG_DIRECTIVE_COMMENT = 6,
  BRIG_DIRECTIVE_CONTROL = 7,
  BRIG_DIRECTIVE_EXTENSION = 8,
  BRIG_DIRECTIVE_FBARRIER = 9,
  BRIG_DIRECTIVE_FILE = 10,
  BRIG_DIRECTIVE_FUNCTION = 11,
  BRIG_DIRECTIVE_IMAGE = 12,
  BRIG_DIRECTIVE_IMAGE_INIT = 13,
  BRIG_DIRECTIVE_KERNEL = 14,
  BRIG_DIRECTIVE_LABEL = 15,
  BRIG_DIRECTIVE_LABEL_INIT = 16,
  BRIG_DIRECTIVE_LABEL_TARGETS = 17,
  BRIG_DIRECTIVE_LOC = 18,
  BRIG_DIRECTIVE_PRAGMA = 19,
  BRIG_DIRECTIVE_SAMPLER = 20,
  BRIG_DIRECTIVE_SAMPLER_INIT = 21,
  BRIG_DIRECTIVE_SCOPE = 22,
  BRIG_DIRECTIVE_SIGNATURE = 23,
  BRIG_DIRECTIVE_VARIABLE = 24,
  BRIG_DIRECTIVE_VARIABLE_INIT = 25,
  BRIG_DIRECTIVE_VERSION = 26
 };
```

- `BrigInstKinds` is used to specify the kind of `.code` section structure:

```
typedef uint16_t BrigInstKinds16_t;
enum BrigInstKinds {
  BRIG_INST_NONE = 0,
  BRIG_INST_BASIC = 1,
  BRIG_INST_ATOMIC = 2,
  BRIG_INST_ATOMIC_IMAGE = 3,
  BRIG_INST_BAR = 4,
  BRIG_INST_BR = 5,
  BRIG_INST_CMP = 6,
  BRIG_INST_CVT = 7,
  BRIG_INST_FBAR = 8,
  BRIG_INST_IMAGE = 9,
  BRIG_INST_MEM = 10,
  BRIG_INST_ADDR = 11,
  BRIG_INST_MOD = 12,
  BRIG_INST_SEG = 13,
  BRIG_INST_SOURCE_TYPE = 14
};
```

- `BrigOperandKinds` is used to specify the kind of `.operand` section structure:

```
typedef uint16_t BrigOperandKinds16_t;
enum BrigOperandKinds {
  BRIG_OPERAND_IMMED = 0,
  BRIG_OPERAND_WAVESIZE = 1,
  BRIG_OPERAND_REG = 2,
  BRIG_OPERAND_REG_VECTOR = 3,
  BRIG_OPERAND_ADDRESS = 4,
  BRIG_OPERAND_LABEL_REF = 5,
  BRIG_OPERAND_ARGUMENT_REF = 6,
  BRIG_OPERAND_ARGUMENT_LIST = 7,
  BRIG_OPERAND_FUNCTION_REF = 8,
  BRIG_OPERAND_FUNCTION_LIST = 9,
  BRIG_OPERAND_SIGNATURE_REF = 10,
  BRIG_OPERAND_FBARRIER_REF = 11
};
```

19.2.3 BrigAluModifierMask

`BrigAluModifierMask` defines bit masks that can be used to access the modifiers for arithmetic logic unit operations.

```
typedef uint16_t BrigAluModifier16_t;
enum BrigAluModifierMask {
  BRIG_ALU_ROUND = 15,
  BRIG_ALU_FTZ = 16
};
```

- `BRIG_ALU_ROUND` — a bit mask that can be used to select the rounding mode. Values are specified by the `BrigRound` enumeration. See 19.2.19 BrigRound (p. 271).
- `BRIG_ALU_FTZ` — a bit mask that can be used to select the setting for the `ftz` (floating-point flush subnormals to zero) modifier. A 0 value means it is absent and a non-0 value means it is present.

19.2.4 BrigAtomicOperation

`BrigAtomicOperation` is used to specify the type of atomic operation. For more information, see 6.4 Atomic Operations: atomic and atomicnoret (p. 135).

```
typedef uint8_t BrigAtomicOperation8_t;
enum BrigAtomicOperation {
  BRIG_ATOMIC_AND = 0,
  BRIG_ATOMIC_OR = 1,
  BRIG_ATOMIC_XOR = 2,
  BRIG_ATOMIC_CAS = 3,
  BRIG_ATOMIC_EXCH = 4,
  BRIG_ATOMIC_ADD = 5,
  BRIG_ATOMIC_INC = 6,
  BRIG_ATOMIC_DEC = 7,
  BRIG_ATOMIC_MIN = 8,
  BRIG_ATOMIC_MAX = 9,
  BRIG_ATOMIC_SUB = 10
};
```

19.2.5 BrigCompareOperation

`BrigCompareOperation` is used to specify the type of compare operation. For more information, see 5.17 Compare (cmp) Operation (p. 112).

```
typedef uint8_t BrigCompareOperation8_t;
enum BrigCompareOperation {
  BRIG_COMPARE_EQ = 0,
  BRIG_COMPARE_NE = 1,
  BRIG_COMPARE_LT = 2,
  BRIG_COMPARE_LE = 3,
  BRIG_COMPARE_GT = 4,
  BRIG_COMPARE_GE = 5,
  BRIG_COMPARE_EQU = 6,
  BRIG_COMPARE_NEU = 7,
  BRIG_COMPARE_LTU = 8,
  BRIG_COMPARE_LEU = 9,
  BRIG_COMPARE_GTU = 10,
  BRIG_COMPARE_GEU = 11,
  BRIG_COMPARE_NUM = 12,
  BRIG_COMPARE_NAN = 13,
  BRIG_COMPARE_SEQ = 14,
  BRIG_COMPARE_SNE = 15,
  BRIG_COMPARE_SLT = 16,
  BRIG_COMPARE_SLE = 17,
  BRIG_COMPARE_SGT = 18,
  BRIG_COMPARE_SGE = 19,
  BRIG_COMPARE_SGEU = 20,
  BRIG_COMPARE_SEQU = 21,
  BRIG_COMPARE_SNEU = 22,
  BRIG_COMPARE_SLTU = 23,
  BRIG_COMPARE_SLEU = 24,
  BRIG_COMPARE_SNUM = 25,
  BRIG_COMPARE_SNAN = 26,
  BRIG_COMPARE_SGTU = 27
};
```

19.2.6 BrigControlDirective

`BrigControlDirective` is used to specify the type of control directive. For more information, see 14.6 Control Directives for Low-Level Performance Tuning (p. 235).

```
typedef uint16_t BrigControlDirective16_t;
enum BrigControlDirective {
  BRIG_CONTROL_NONE = 0,
  BRIG_CONTROL_ENABLEBREAKEXCEPTIONS = 1,
  BRIG_CONTROL_ENABLEDETECTEXCEPTIONS = 2,
  BRIG_CONTROL_MAXDYNAMICGROUPSIZE = 3,
  BRIG_CONTROL_MAXFLATGRIDSIZE = 4,
  BRIG_CONTROL_MAXFLATWORKGROUPSIZE = 5,
  BRIG_CONTROL_REQUESTEDWORKGROUPSPERCU = 6,
  BRIG_CONTROL_REQUIREDDIM = 7,
  BRIG_CONTROL_REQUIREDGRIDSIZE = 8,
  BRIG_CONTROL_REQUIREDWORKGROUPSIZE = 9,
  BRIG_CONTROL_REQUIRENOPARTIALWORKGROUPS = 10
};
```

19.2.7 BrigExecutableModifierMask

`BrigExecutableModifierMask` defines bit masks that can be used to access properties about an executable kernel or function.

```
typedef uint8_t BrigExecutableModifier8_t;
enum BrigExecuteableModifierMask {
  BRIG_EXECUTABLE_LINKAGE = 3,
  BRIG_EXECUTABLE_DECLARATION = 4
};
```

- `BRIG_EXECUTABLE_LINKAGE` — Values are specified by the `BrigLinkage` enumeration. See 19.2.11 BrigLinkage (p. 265).
- `BRIG_EXECUTABLE_DECLARATION` — 0 means this is a definition and has a code block; 1 means this is a declaration only and has no code block.

See 19.6.8 BrigDirectiveExecutable (p. 285).

19.2.8 BrigImageFormat

`BrigImageFormat` is used to specify the image format. For more information, see 7.1.4 Image Objects (p. 154).

```
typedef uint8_t BrigImageFormat8_t;
enum BrigImageFormat {
  BRIG_FORMAT_SNORM_INT8 = 0,
  BRIG_FORMAT_SNORM_INT16 = 1,
  BRIG_FORMAT_UNORM_INT8 = 2,
  BRIG_FORMAT_UNORM_INT16 = 3,
  BRIG_FORMAT_UNORM_SHORT_565 = 4,
  BRIG_FORMAT_UNORM_SHORT_555 = 5,
  BRIG_FORMAT_UNORM_SHORT_101010 = 6,
  BRIG_FORMAT_SIGNED_INT8 = 7,
  BRIG_FORMAT_SIGNED_INT16 = 8,
  BRIG_FORMAT_SIGNED_INT32 = 9,
  BRIG_FORMAT_UNSIGNED_INT8 = 10,
  BRIG_FORMAT_UNSIGNED_INT16 = 11,
  BRIG_FORMAT_UNSIGNED_INT32 = 12,
  BRIG_FORMAT_HALF_FLOAT = 13,
  BRIG_FORMAT_FLOAT = 14,
  BRIG_FORMAT_UNORM_INT24 = 15
};
```

Values 16 through 64 are available for extensions.

19.2.9 BrigImageGeometry

`BrigImageGeometry` is used to specify the number of coordinates needed to access an image. For more information, see 7.1.3 Image Geometry (p. 153).

```
typedef uint8_t BrigImageGeometry8_t;
enum BrigImageGeometry {
  BRIG_GEOMETRY_1D = 0,
  BRIG_GEOMETRY_2D = 1,
  BRIG_GEOMETRY_3D = 2,
  BRIG_GEOMETRY_1DA = 3,
  BRIG_GEOMETRY_1DB = 4,
  BRIG_GEOMETRY_2DA = 5
};
```

Values 6 through 255 are available for extensions.

19.2.10 BrigImageOrder

`BrigImageOrder` is used to specify the order of image components. For more information, see 7.1.4 Image Objects (p. 154).

```
typedef uint8_t BrigImageOrder8_t;
enum BrigImageOrder {
  BRIG_ORDER_R = 0,
  BRIG_ORDER_A = 1,
  BRIG_ORDER_RX = 2,
  BRIG_ORDER_RG = 3,
  BRIG_ORDER_RGX = 4,
  BRIG_ORDER_RA = 5,
  BRIG_ORDER_RGB = 6,
  BRIG_ORDER_RGBA = 7,
  BRIG_ORDER_RGBX = 8,
  BRIG_ORDER_BGRA = 9,
  BRIG_ORDER_ARGB = 10,
  BRIG_ORDER_INTENSITY = 11,
  BRIG_ORDER_LUMINANCE = 12,
  BRIG_ORDER_SRGB = 13,
  BRIG_ORDER_SRGBX = 14,
  BRIG_ORDER_SRGBA = 15,
  BRIG_ORDER_SBGRA = 16
};
```

Values 17 through 255 are available for extensions.

19.2.11 BrigLinkage

`BrigLinkage` is used to specify linkage. For more information, see 4.23 Linkage: External, Static, and None (p. 61).

```
typedef uint8_t BrigLinkage8_t;
enum BrigLinkage {
  BRIG_LINKAGE_NONE = 0,
  BRIG_LINKAGE_STATIC = 1,
  BRIG_LINKAGE_EXTERN = 2
};
```

19.2.12 BrigMachineModel

`BrigMachineModel` is used to specify the kind of machine model. For more information, see 2.10 Small and Large Machine Models (p. 20).

```
typedef uint8_t BrigMachineModel8_t;
enum BrigMachineModel {
  BRIG_MACHINE_SMALL = 0,
  BRIG_MACHINE_LARGE = 1
};
```

19.2.13 BrigMemoryFence

`BrigMemoryFence` is used to specify the kind of memory fence performed by an operation. If an operation has a memory fence modifier (`fence`) but does not explicitly specify the kind of memory fence, then the value used must be the default value defined by the operation. If the operation does not support the memory fence

modifier, then BRIG_FENCE_NONE must be used. See 9.1 Memory Fence Modifier (p. 185).

```
typedef uint8_t BrigMemoryFence8_t;
enum BrigMemoryFence {
  BRIG_FENCE_NONE = 0,
  BRIG_FENCE_GROUP = 1,
  BRIG_FENCE_GLOBAL = 2,
  BRIG_FENCE_BOTH = 3,
  BRIG_FENCE_PARTIAL = 4,
  BRIG_FENCE_PARTIAL_BOTH = 5
};
```

19.2.14 BrigMemoryModifierMask

BrigMemoryModifierMask defines bit masks that can be used to access the modifiers for memory operations.

```
typedef uint16_t BrigMemoryModifier8_t;
enum BrigMemoryModifierMask {
  BRIG_MEMORY_SEMANTIC = 15,
  BRIG_MEMORY_ALIGNED = 16
  };
```

- BRIG_MEMORY_SEMANTIC — A bit mask that can be used to select the semantics of the memory operation. Values are specified by the BrigMemorySemantic enumeration. If the operation does not support the memory semantic modifier, then this must be BRIG_SEMANTIC_NONE. If the operation supports the memory semantic modifier but does not specify it, then this must be BRIG_SEMANTIC_REGULAR. See 19.2.15 BrigMemorySemantic (p. 266).
- BRIG_MEMORY_ALIGNED — A bit mask that can be used to select the setting for the aligned modifier. A 0 value means it is absent and a non-0 value means it is present. If the operation does not support the aligned modifier, then this must be 0.

19.2.15 BrigMemorySemantic

BrigMemorySemantic is used to specify the semantics for a memory operation. For more information, see 6.7 Examples of Memory Operations (p. 145).

```
typedef uint8_t BrigMemorySemantic8_t;
enum BrigMemorySemantic {
  BRIG_SEMANTIC_NONE = 0,
  BRIG_SEMANTIC_REGULAR = 1,
  BRIG_SEMANTIC_ACQUIRE = 2,
  BRIG_SEMANTIC_RELEASE = 3,
  BRIG_SEMANTIC_ACQUIRE_RELEASE = 4,
  BRIG_SEMANTIC_PARTIAL_ACQUIRE = 5,
  BRIG_SEMANTIC_PARTIAL_RELEASE = 6,
  BRIG_SEMANTIC_PARTIAL_ACQUIRE_RELEASE = 7
};
```

19.2.16 BrigOpcode

`BrigOpcode` is used to specify the opcode for the HSAIL operation.

```
typedef uint16_t BrigOpcode16_t;
enum BrigOpcode {
  BRIG_OPCODE_NOP = 0,
  BRIG_OPCODE_ABS = 1,
  BRIG_OPCODE_ADD = 2,
  BRIG_OPCODE_BORROW = 3,
  BRIG_OPCODE_CARRY = 4,
  BRIG_OPCODE_CEIL = 5,
  BRIG_OPCODE_COPYSIGN = 6,
  BRIG_OPCODE_DIV = 7,
  BRIG_OPCODE_FLOOR = 8,
  BRIG_OPCODE_FMA = 9,
  BRIG_OPCODE_FRACT = 10,
  BRIG_OPCODE_MAD = 11,
  BRIG_OPCODE_MAX = 12,
  BRIG_OPCODE_MIN = 13,
  BRIG_OPCODE_MUL = 14,
  BRIG_OPCODE_MULHI = 15,
  BRIG_OPCODE_NEG = 16,
  BRIG_OPCODE_REM = 17,
  BRIG_OPCODE_RINT = 18,
  BRIG_OPCODE_SQRT = 19,
  BRIG_OPCODE_SUB = 20,
  BRIG_OPCODE_TRUNC = 21,
  BRIG_OPCODE_MAD24 = 22,
  BRIG_OPCODE_MAD24HI = 23,
  BRIG_OPCODE_MUL24 = 24,
  BRIG_OPCODE_MUL24HI = 25,
  BRIG_OPCODE_SHL = 26,
  BRIG_OPCODE_SHR = 27,
  BRIG_OPCODE_AND = 28,
  BRIG_OPCODE_NOT = 29,
  BRIG_OPCODE_OR = 30,
  BRIG_OPCODE_POPCOUNT = 31,
  BRIG_OPCODE_XOR = 32,
  BRIG_OPCODE_BITEXTRACT = 33,
  BRIG_OPCODE_BITINSERT = 34,
  BRIG_OPCODE_BITMASK = 35,
  BRIG_OPCODE_BITREV = 36,
  BRIG_OPCODE_BITSELECT = 37,
  BRIG_OPCODE_FIRSTBIT = 38,
  BRIG_OPCODE_LASTBIT = 39,
  BRIG_OPCODE_COMBINE = 40,
  BRIG_OPCODE_EXPAND = 41,
  BRIG_OPCODE_LDA = 42,
  BRIG_OPCODE_LDC = 43,
  BRIG_OPCODE_MOV = 44,
  BRIG_OPCODE_SHUFFLE = 45,
  BRIG_OPCODE_UNPACKHI = 46,
  BRIG_OPCODE_UNPACKLO = 47,
  BRIG_OPCODE_PACK = 48,
  BRIG_OPCODE_UNPACK = 49,
  BRIG_OPCODE_CMOV = 50,
  BRIG_OPCODE_CLASS = 51,
  BRIG_OPCODE_NCOS = 52,
  BRIG_OPCODE_NEXP2 = 53,
```

```
BRIG_OPCODE_NFMA = 54,
BRIG_OPCODE_NLOG2 = 55,
BRIG_OPCODE_NRCP = 56,
BRIG_OPCODE_NRSQRT = 57,
BRIG_OPCODE_NSIN = 58,
BRIG_OPCODE_NSQRT = 59,
BRIG_OPCODE_BITALIGN = 60,
BRIG_OPCODE_BYTEALIGN = 61,
BRIG_OPCODE_PACKCVT = 62,
BRIG_OPCODE_UNPACKCVT = 63,
BRIG_OPCODE_LERP = 64,
BRIG_OPCODE_SAD = 65,
BRIG_OPCODE_SADHI = 66,
BRIG_OPCODE_SEGMENTP = 67,
BRIG_OPCODE_FTOS = 68,
BRIG_OPCODE_STOF = 69,
BRIG_OPCODE_CMP = 70,
BRIG_OPCODE_CVT = 71,
BRIG_OPCODE_LD = 72,
BRIG_OPCODE_ST = 73,
BRIG_OPCODE_ATOMIC = 74,
BRIG_OPCODE_ATOMICNORET = 75,
BRIG_OPCODE_RDIMAGE = 76,
BRIG_OPCODE_LDIMAGE = 77,
BRIG_OPCODE_STIMAGE = 78,
BRIG_OPCODE_ATOMICIMAGE = 79,
BRIG_OPCODE_ATOMICIMAGENORET = 80,
BRIG_OPCODE_QUERYIMAGEARRAY = 81,
BRIG_OPCODE_QUERYIMAGEDEPTH = 82,
BRIG_OPCODE_QUERYIMAGEFORMAT = 83,
BRIG_OPCODE_QUERYIMAGEHEIGHT = 84,
BRIG_OPCODE_QUERYIMAGEORDER = 85,
BRIG_OPCODE_QUERYIMAGEWIDTH = 86,
BRIG_OPCODE_QUERYSAMPLERCOORD = 87,
BRIG_OPCODE_QUERYSAMPLERFILTER = 88,
BRIG_OPCODE_CBR = 89,
BRIG_OPCODE_BRN = 90,
BRIG_OPCODE_BARRIER = 91,
BRIG_OPCODE_ARRIVEFBAR = 92,
BRIG_OPCODE_INITFBAR = 93,
BRIG_OPCODE_JOINFBAR = 94,
BRIG_OPCODE_LEAVEFBAR = 95,
BRIG_OPCODE_RELEASEFBAR = 96,
BRIG_OPCODE_WAITFBAR = 97,
BRIG_OPCODE_LDF = 98,
BRIG_OPCODE_SYNC = 99,
BRIG_OPCODE_COUNTLANE = 100,
BRIG_OPCODE_COUNTUPLANE = 101,
BRIG_OPCODE_MASKLANE = 102,
BRIG_OPCODE_SENDLANE = 103,
BRIG_OPCODE_RECEIVELANE = 104,
BRIG_OPCODE_CALL = 105,
BRIG_OPCODE_RET = 106,
BRIG_OPCODE_SYSCALL = 107,
BRIG_OPCODE_ALLOCA = 108,
BRIG_OPCODE_CLEARDETECTEXCEPT = 109,
```

```
  BRIG_OPCODE_CLOCK = 110,
  BRIG_OPCODE_CUID = 111,
  BRIG_OPCODE_CURRENTWORKGROUPSIZE = 112,
  BRIG_OPCODE_DEBUGTRAP = 113,
  BRIG_OPCODE_DIM = 114,
  BRIG_OPCODE_DISPATCHID = 115,
  BRIG_OPCODE_DISPATCHPTR = 116,
  BRIG_OPCODE_GETDETECTEXCEPT = 117,
  BRIG_OPCODE_GRIDGROUPS = 118,
  BRIG_OPCODE_GRIDSIZE = 119,
  BRIG_OPCODE_LANEID = 120,
  BRIG_OPCODE_MAXCUID = 121,
  BRIG_OPCODE_MAXWAVEID = 122,
  BRIG_OPCODE_NULLPTR = 123,
  BRIG_OPCODE_QID = 124,
  BRIG_OPCODE_QPTR = 125,
  BRIG_OPCODE_SETDETECTEXCEPT = 126,
  BRIG_OPCODE_WAVEID = 127,
  BRIG_OPCODE_WORKGROUPID = 128,
  BRIG_OPCODE_WORKGROUPSIZE = 129,
  BRIG_OPCODE_WORKITEMABSID = 130,
  BRIG_OPCODE_WORKITEMFLATABSID = 131,
  BRIG_OPCODE_WORKITEMFLATID = 132,
  BRIG_OPCODE_WORKITEMID = 133
};
```

19.2.17 BrigPack

`BrigPack` is used to specify the kind of packing control for packed data. For more information, see 4.15 Packing Controls for Packed Data (p. 47).

```
typedef uint8_t BrigPack8_t;
enum BrigPack {
  BRIG_PACK_NONE = 0,
  BRIG_PACK_PP = 1,
  BRIG_PACK_PS = 2,
  BRIG_PACK_SP = 3,
  BRIG_PACK_SS = 4,
  BRIG_PACK_S = 5,
  BRIG_PACK_P = 6,
  BRIG_PACK_PPSAT = 7,
  BRIG_PACK_PSSAT = 8,
  BRIG_PACK_SPSAT = 9,
  BRIG_PACK_SSSAT = 10,
  BRIG_PACK_SSAT = 11,
  BRIG_PACK_PSAT = 12
};
```

19.2.18 BrigProfile

`BrigProfile` is used to specify the kind of profile. For more information, see 17.1 What Are Profiles? (p. 249).

```
typedef uint8_t BrigProfile8_t;
enum BrigProfile {
  BRIG_PROFILE_BASE = 0,
  BRIG_PROFILE_FULL = 1
};
```

19.2.19 BrigRound

`BrigRound` is used to specify rounding. For more information, see 4.11 Rounding Modes (p. 39) and 5.18.3 Rules for Rounding for Conversions (p. 118).

If the operation does not support a rounding mode, then `BRIG_ROUND_NONE` must be used. Otherwise, the appropriate rounding mode must be used.

If the operation supports a rounding mode but does not explicitly specify one, then `BRIG_ROUND_FLOAT_NEAR_EVEN` or `BRIG_ROUND_INTEGER_ZERO` must be specified as appropriate, not `BRIG_ROUND_NONE`.

```
typedef uint8_t BrigRound8_t;
enum BrigRound {
  BRIG_ROUND_NONE = 0,
  BRIG_ROUND_FLOAT_NEAR_EVEN = 1,
  BRIG_ROUND_FLOAT_ZERO = 2,
  BRIG_ROUND_FLOAT_PLUS_INFINITY = 3,
  BRIG_ROUND_FLOAT_MINUS_INFINITY = 4,
  BRIG_ROUND_INTEGER_NEAR_EVEN = 5,
  BRIG_ROUND_INTEGER_ZERO = 6,
  BRIG_ROUND_INTEGER_PLUS_INFINITY = 7,
  BRIG_ROUND_INTEGER_MINUS_INFINITY = 8,
  BRIG_ROUND_INTEGER_NEAR_EVEN_SAT = 9,
  BRIG_ROUND_INTEGER_ZERO_SAT = 10,
  BRIG_ROUND_INTEGER_PLUS_INFINITY_SAT = 11,
  BRIG_ROUND_INTEGER_MINUS_INFINITY_SAT = 12
};
```

19.2.20 BrigSamplerBoundaryMode

`BrigSamplerBoundaryMode` is used to specify the boundary mode for the `boundaryU`, `boundaryV`, and `boundaryW` fields in the sampler object. For more information, see 7.1.7 Sampler Objects (p. 160).

```
typedef uint8_t BrigSamplerBoundaryMode8_t;
enum BrigSamplerBoundaryMode {
  BRIG_BOUNDARY_CLAMP = 0,
  BRIG_BOUNDARY_WRAP = 1,
  BRIG_BOUNDARY_MIRROR = 2,
  BRIG_BOUNDARY_MIRRORONCE = 3,
  BRIG_BOUNDARY_BORDER = 4
};
```

Values 5 through 255 are available for extensions.

19.2.21 BrigSamplerCoord

`BrigSamplerCoord` is used to specify the setting for the `coord` field in the sampler object. For more information, see 7.1.7 Sampler Objects (p. 160).

```
enum BrigSamplerCoord {
  BRIG_COORD_NORMALIZED = 0,
  BRIG_COORD_UNNORMALIZED = 1
};
```

Values 5 through 255 are available for extensions.

19.2.22 BrigSamplerFilter

`BrigSamplerFilter` is used to specify the setting for the `filter` field in the sampler object. For more information, see 7.1.7 Sampler Objects (p. 160).

```
enum BrigSamplerFilter {
  BRIG_FILTER_NEAREST = 0,
  BRIG_FILTER_LINEAR = 1
};
```

Values 2 through 63 are available for extensions.

19.2.23 BrigSamplerModifierMask

`BrigSamplerModifierMask` defines bit masks that can be used to access the properties of a sampler object. For more information, see 7.1.7 Sampler Objects (p. 160).

```
typedef uint8_t BrigSamplerModifier8_t;
enum BrigSamplerModifierMask {
  BRIG_SAMPLER_FILTER = 63,
  BRIG_SAMPLER_COORD = 64,
  BRIG_SAMPLER_COORD_UNNORMALIZED = 64
};
```

- `BRIG_SAMPLER_FILTER` — A six-bit field specifying one of the filter modes defined in `BrigSamplerFilter`. See 19.2.22 BrigSamplerFilter (p. 272).
- `BRIG_SAMPLER_COORD` — A one-bit field specifying one of the two coord modes defined in `BrigSamplerCoord`. For convenience, the value for `BRIG_COORD_UNNORMALIZED` can be tested for using the bit mask `BRIG_SAMPLER_COORD_UNNORMALIZED`. See 19.2.21 BrigSamplerCoord (p. 272).

19.2.24 BrigSegment

`BrigSegment` is used to specify the memory segment for a symbol. For more information, see 2.8 Segments (p. 13).

```
typedef uint8_t BrigSegment8_t;
enum BrigSegment {
  BRIG_SEGMENT_NONE = 0,
  BRIG_SEGMENT_FLAT = 1,
  BRIG_SEGMENT_GLOBAL = 2,
  BRIG_SEGMENT_READONLY = 3,
  BRIG_SEGMENT_KERNARG = 4,
  BRIG_SEGMENT_GROUP = 5,
  BRIG_SEGMENT_PRIVATE = 6,
  BRIG_SEGMENT_SPILL = 7,
  BRIG_SEGMENT_ARG = 8
};
```

Values 9 through 16 are available for extensions.

19.2.25 BrigSymbolModifierMask

`BrigSymbolModifierMask` defines bit masks that can be used to access properties about a symbol.

```
typedef uint8_t BrigSymbolModifier8_t;
enum BrigSymbolModifierMask {
  BRIG_SYMBOL_LINKAGE = 3,
  BRIG_SYMBOL_DECLARATION = 4,
  BRIG_SYMBOL_CONST = 8,
  BRIG_SYMBOL_ARRAY = 16,
  BRIG_SYMBOL_FLEX_ARRAY = 32
};
```

- `BRIG_SYMBOL_LINKAGE` — Values are specified by the `BrigLinkage` enumeration. See 19.2.11 BrigLinkage (p. 265).
- `BRIG_SYMBOL_DECLARATION` — 1 means this is a declaration (an external); 0 means this is a definition.
- `BRIG_SYMBOL_CONST` — 1 means `const`; 0 means read/write.
- `BRIG_SYMBOL_ARRAY` — 1 means vector (size in `dim`); 0 means scalar.
- `BRIG_SYMBOL_FLEX_ARRAY` — 1 means flexible array (array with no explicit size), `BRIG_SYMBOL_ARRAY` must be set, and `dim` must be 0. An array declared in the textual form without a size, but with an initializer, is not considered a flexible array because its size is defined by the size of the initializer. In this case, `BRIG_SYMBOL_FLEX_ARRAY` must not be set, and `dim` must be set to the size of the initializer.

See 19.6.19 BrigDirectiveSymbol (p. 294).

19.2.26 BrigType

`BrigType` is used to specify the data compound type of operations, operands, variables, arguments, initializers, and block numeric values.

The `BrigType` enumeration is encoded to make it easy to determine if the type is packed, and if so to determine the packed element compound type and the bit size of the packed type.

The base type is encoded in the bottom 5 bits, and the packed type size recorded in the next 2 bits.

For the packed type size: 0 means not a packed type, 1 means a 32-bit packed type, 2 means a 64-bit packed type, and 3 means a 128-bit packed type. Masks, shifts, and enumeration values are provided to access the base type and access and test the packed type size.

For more information, see 4.14 Data Types (p. 46).

```
enum {
  BRIG_TYPE_PACK_SHIFT = 5,
  BRIG_TYPE_BASE_MASK = (1 << BRIG_TYPE_PACK_SHIFT) - 1,
  BRIG_TYPE_PACK_MASK = 3 << BRIG_TYPE_PACK_SHIFT,

  BRIG_TYPE_PACK_NONE = 0 << BRIG_TYPE_PACK_SHIFT,
  BRIG_TYPE_PACK_32 = 1 << BRIG_TYPE_PACK_SHIFT,
  BRIG_TYPE_PACK_64 = 2 << BRIG_TYPE_PACK_SHIFT,
  BRIG_TYPE_PACK_128 = 3 << BRIG_TYPE_PACK_SHIFT
};
```

```
typedef uint16_t BrigType16_t;
enum BrigType {
  BRIG_TYPE_NONE = 0,

  BRIG_TYPE_U8 = 1,
  BRIG_TYPE_U16 = 2,
  BRIG_TYPE_U32 = 3,
  BRIG_TYPE_U64 = 4,

  BRIG_TYPE_S8 = 5,
  BRIG_TYPE_S16 = 6,
  BRIG_TYPE_S32 = 7,
  BRIG_TYPE_S64 = 8,

  BRIG_TYPE_F16 = 9,
  BRIG_TYPE_F32 = 10,
  BRIG_TYPE_F64 = 11,

  BRIG_TYPE_B1 = 12,
  BRIG_TYPE_B8 = 13,
  BRIG_TYPE_B16 = 14,
  BRIG_TYPE_B32 = 15,
  BRIG_TYPE_B64 = 16,
  BRIG_TYPE_B128 = 17,

  BRIG_TYPE_SAMP = 18,
  BRIG_TYPE_ROIMG = 19,
  BRIG_TYPE_RWIMG = 20,

  BRIG_TYPE_FBAR = 21,

  BRIG_TYPE_U8X4 = BRIG_TYPE_U8 | BRIG_TYPE_PACK_32,
  BRIG_TYPE_U8X8 = BRIG_TYPE_U8 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_U8X16 = BRIG_TYPE_U8 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_U16X2 = BRIG_TYPE_U16 | BRIG_TYPE_PACK_32,
  BRIG_TYPE_U16X4 = BRIG_TYPE_U16 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_U16X8 = BRIG_TYPE_U16 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_U32X2 = BRIG_TYPE_U32 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_U32X4 = BRIG_TYPE_U32 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_U64X2 = BRIG_TYPE_U64 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_S8X4 = BRIG_TYPE_S8 | BRIG_TYPE_PACK_32,
  BRIG_TYPE_S8X8 = BRIG_TYPE_S8 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_S8X16 = BRIG_TYPE_S8 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_S16X2 = BRIG_TYPE_S16 | BRIG_TYPE_PACK_32,
  BRIG_TYPE_S16X4 = BRIG_TYPE_S16 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_S16X8 = BRIG_TYPE_S16 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_S32X2 = BRIG_TYPE_S32 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_S32X4 = BRIG_TYPE_S32 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_S64X2 = BRIG_TYPE_S64 | BRIG_TYPE_PACK_128,
```

```
  BRIG_TYPE_F16X2 = BRIG_TYPE_F16 | BRIG_TYPE_PACK_32,
  BRIG_TYPE_F16X4 = BRIG_TYPE_F16 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_F16X8 = BRIG_TYPE_F16 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_F32X2 = BRIG_TYPE_F32 | BRIG_TYPE_PACK_64,
  BRIG_TYPE_F32X4 = BRIG_TYPE_F32 | BRIG_TYPE_PACK_128,

  BRIG_TYPE_F64X2 = BRIG_TYPE_F64 | BRIG_TYPE_PACK_128
};
```

19.2.27 BrigVersion

The literal values of `BrigVersion` define the versions of BRIG defined by this revision of the HSAIL specification.

```
enum BrigVersion {
  BRIG_VERSION_MAJOR = 0,
  BRIG_VERSION_MINOR = 0
};
```

- `BRIG_VERSION_MAJOR` — The major version of this revision of the HSAIL specification. This is the value used in the version directive `major` field. BRIG with a major version different from this value is not compatible with this revision of the HSAIL specification.
- `BRIG_VERSION_MINOR` — The minor version of this revision of the HSAIL specification. This is the value used in the version directive `minor` field. BRIG is compatible with this revision of the HSAIL specification only if it has the same major version and a minor version less than or equal to this value.

19.2.28 BrigWidth

`BrigWidth` is used to specify the width modifier. Because the width must be a power of 2 between 1 and 2^{31} inclusive, only enumerations for the power of 2 values are present, and they are numbered as $\log_2(n) + 1$ of the value. In addition, `width(all)` and `width(WAVESIZE)` have an enumeration value that comes after the explicit numbered enumerations. This makes it is easy for a finalizer to determine if a width value is greater than or equal to the wavefront size by simply doing a comparison of greater than or equal with the enumeration value that corresponds to the actual wavefront size of the implementation. For more information, see 2.13.1 Width Modifier (p. 22).

```
typedef uint8_t BrigWidth8_t;
enum BrigWidth {
  BRIG_WIDTH_NONE = 0,
  BRIG_WIDTH_1 = 1,
  BRIG_WIDTH_2 = 2,
  BRIG_WIDTH_4 = 3,
  BRIG_WIDTH_8 = 4,
  BRIG_WIDTH_16 = 5,
  BRIG_WIDTH_32 = 6,
  BRIG_WIDTH_64 = 7,
  BRIG_WIDTH_128 = 8,
  BRIG_WIDTH_256 = 9,
  BRIG_WIDTH_512 = 10,
  BRIG_WIDTH_1024 = 11,
  BRIG_WIDTH_2048 = 12,
  BRIG_WIDTH_4096 = 13,
  BRIG_WIDTH_8192 = 14,
  BRIG_WIDTH_16364 = 15,
  BRIG_WIDTH_32768 = 16,
  BRIG_WIDTH_65536 = 17,
  BRIG_WIDTH_131072 = 18,
  BRIG_WIDTH_262144 = 19,
  BRIG_WIDTH_524288 = 20,
  BRIG_WIDTH_1048576 = 21,
  BRIG_WIDTH_2097152 = 22,
  BRIG_WIDTH_4194304 = 23,
  BRIG_WIDTH_8388608 = 24,
  BRIG_WIDTH_16777216 = 25,
  BRIG_WIDTH_33554432 = 26,
  BRIG_WIDTH_67108864 = 27,
  BRIG_WIDTH_134217728 = 28,
  BRIG_WIDTH_268435456 = 29,
  BRIG_WIDTH_536870912 = 30,
  BRIG_WIDTH_1073741824 = 31,
  BRIG_WIDTH_2147483648 = 32,
  BRIG_WIDTH_WAVESIZE = 33,
  BRIG_WIDTH_ALL = 34
};
```

19.3 Section Header

The first entry in every section must be `BrigSectionHeader`, which consists of a 32-bit word that is the byte size of the section. There are no section termination flags. Any code that generates Brig needs to correctly fill in each section's size. This also allows a section offset of 0 to indicate no value, because the first entry in each section starts at an offset of 4.

Syntax is:

```
struct BrigSectionHeader {
  uint32_t size;
  };
```

Field is:

- `size` — size in bytes of the section

19.4 .string Section

The `.string` section must start with a `BrigSectionHeader` entry. See 19.3 Section Header (p. 277).

The `.string` section is used both to store textual strings used for identifiers within HSAIL and for the value of variable initializers and block numeric and block string values.

An entry comprises both the length of the data in bytes and the actual bytes of the data.

An offset value into the `.string` section references the start of the `BrigString`, not the data, which starts at `data` within `BrigString`.

Entries for HSAIL identifiers and block string values are stored as ASCII character strings without null termination. The length is the number of characters in the identifier.

Data entries are stored as raw bytes with no terminating byte. The length is the number of bytes in the data.

In both cases, the length does not include the number of padding bytes that must be added to make the entry a multiple of 4.

Each `BrigString` starts on a 4-byte boundary. Any required padding bytes after the data to make the entry a multiple of 4 bytes must be 0.

There must be only one entry for each unique `BrigString` value. Thus, the value of two entries is equal if and only if their `.string` section offsets are equal. This allows a finalizer to quickly determine if two strings used as an identifier are equal by simply comparing the offset values. Entries being used for identifiers and those used for data are not distinguished in the `.string` section. The uniqueness property applies across all entries regardless of how they are being used. Indeed, an entry can be used both as an identifier and as the initializer data for a variable.

Syntax is:

```
struct BrigString {
  uint32_t byteCount;
  uint8_t bytes[1];
  };
```

Fields are:

- `byteCount` — Number of bytes in the string data. Does not include any padding bytes that have to be added to ensure the next `BrigString` starts on a 4-byte boundary. Therefore, to locate the start of the next `BrigString`, the value `((7 + byteCount) / 4) * 4)` must be added to the offset of the current `BrigString`.
- `bytes` — Variable-sized. Must be allocated with `(((byteCount + 3) / 4) * 4)` elements. Any elements after `byteCount` – 1 must be 0. Bytes 0 to `byteCount` – 1 contain the data.

19.5 Block Sections in BRIG

19.5.1 Overview

A block section can appear in either the `.directive` section or the `.debug` section.

Each block section is identified by a name. More than one block section can have the same name.

Each block section starts with a `BrigBlockStart` and ends with a `BrigBlockEnd`.

Block sections cannot nest.

The table below shows the block section structures in alphabetical order.

Table 19–1 Block Section Structures

Name	Description
`BrigBlockEnd`	Ends a block section. See 19.5.2 BrigBlockEnd (p. 279).
`BrigBlockNumeric`	List of numeric values in the block section. See 19.5.3 BrigBlockNumeric (p. 279).
`BrigBlockStart`	Starts a block section. See 19.5.4 BrigBlockStart (p. 280).
`BrigBlockString`	A reference to a null-terminated character string. See 19.5.5 BrigBlockString (p. 281).

19.5.2 BrigBlockEnd

`BrigBlockEnd` ends a block section.

Syntax is:

```
struct BrigBlockEnd {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_BLOCK_END`.

19.5.3 BrigBlockNumeric

`BrigBlockNumeric` is a variable-size list of numeric values. All the values should have the same type.

More than one `BrigBlockNumeric` can be in a single block section.

Syntax is:

```
struct BrigBlockNumeric {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigType16_t type;
  uint16_t reserved;
  uint32_t elementCount;
  BrigStringOffset32_t data;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_BLOCK_NUMERIC`.
- `BrigType16_t type` — Type of the data. Must be `BRIG_TYPE_B8`, `BRIG_TYPE_B16`, `BRIG_TYPE_B32`, or `BRIG_TYPE_B64`.
- `uint16_t reserved` — Must be 0.
- `uint32_t elementCount` — Number of data elements that are present in the structure.
- `BrigStringOffset32_t data` — The offset in the `.string` section to the entry for the data bytes. The number of data bytes must be `elementCount` times the byte size of `type`.

19.5.4 BrigBlockStart

`BrigBlockStart` starts a block section.

It provides a name that can be used to separate information used by different debuggers or runtimes.

More than one `BrigBlockStart` can have the same name.

Syntax is:

```
struct BrigBlockStart {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_BLOCK_START`.

- `BrigCodeOffset32_t code` — Byte offset into the `.code` section. The block applies immediately before the operation at that location. For example, a block supplying debug information for a function would have the byte offset set to the first operation in the function.
- `BrigStringOffset32_t name` — Byte offset into the place in the `.string` section where the name appears. Can be either `"debug"` or `"rti"`. All `BrigBlockStart` directives with name `"debug"` appear in the `.debug` section. All `BrigBlockStart` directives with name `"rti"` appear in the `.directive` section. All other block directives are placed in the same section as the preceding `BrigBlockStart`.

19.5.5 BrigBlockString

`BrigBlockString` is a reference to a character string.

Syntax is:

```
struct BrigBlockString {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigStringOffset32_t string;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_BLOCK_STRING`.
- `BrigStringOffset32_t string` — The offset in the `.string` section to the entry for the string characters. The number of data bytes must be the length of the string.

19.6 .directive Section

The `.directive` section must start with a `BrigSectionHeader` entry. See 19.3 Section Header (p. 277).

19.6.1 Overview

Directives are statements, corresponding with functions, kernels, and global declarations. They are not code. Directives appear in the `.directive` section in the same order in which they appear in the text format.

Directives that have limited scope have a special placement rule: Immediately after a function or kernel directive, BRIG requires the directives that describe the arguments to be in a certain order. Return arguments are first, followed by input arguments, followed by the directives that apply only to the function or kernel.

All directives contain a reference to code using the `code` field. The directive applies to the operation at `code` and the following operations.

For directives that are outside of a function or kernel, the `code` value is the first operation of the first function or kernel where the directive applies. In a compilation unit with no functions, `code` would specify the start of the `.code` section. Directives must be ordered with respect to `code` values. It is not valid in BRIG for directive `A` to follow directive `B` in the `.directive` section and have `A.code < B.code`.

If the directive applies after all code in the `.code` section, the offset should be the size of the `.code` section.

The table below shows the directives in alphabetical order.

Table 19–2 Structures in the .directive Section

Name	Description
`BrigDirectiveBase`	Helper type. See 19.6.3 BrigDirectiveBase (p. 283).
`BrigDirectiveCallableBase`	Helper type. See 19.6.4 BrigDirectiveCallableBase (p. 283).
`BrigDirectiveArgScope`	See 19.6.5 BrigDirectiveArgScope (p. 284).
`BrigDirectiveComment`	Comment string. See 19.6.6 BrigDirectiveComment (p. 284).
`BrigDirectiveControl`	Assorted compiler controls. See 19.6.7 BrigDirectiveControl (p. 284).
`BrigDirectiveExecutable`	Describes a kernel or function. See 19.6.8 BrigDirectiveExecutable (p. 285).
`BrigDirectiveExtension`	Used to enable device-specific extensions. See 19.6.9 BrigDirectiveExtension (p. 287).
`BrigDirectiveFbarrier`	Used for fbarrier definitions. See 19.6.10 BrigDirectiveFbarrier (p. 287).
`BrigDirectiveFile`	File descriptor. See 19.6.11 BrigDirectiveFile (p. 288).
`BrigDirectiveImageInit`	Declare an image object. See 19.6.12 BrigDirectiveImageInit (p. 288).
`BrigDirectiveLabel`	Declare a label. See 19.6.13 BrigDirectiveLabel (p. 289).
`BrigDirectiveLabelList`	Entries in a variable-size label list used in a jump table. See 19.6.14 BrigDirectiveLabelList (p. 290).
`BrigDirectiveLoc`	Source-level line position. See 19.6.15 BrigDirectiveLoc (p. 290).
`BrigDirectivePragma`	More compiler controls. See 19.6.16 BrigDirectivePragma (p. 291).
`BrigDirectiveSamplerInit`	Declares a sampler object. See 19.6.17 BrigDirectiveSamplerInit (p. 292).
`BrigDirectiveSignature`	Declares a function signature. See 19.6.18 BrigDirectiveSignature (p. 292).
`BrigDirectiveSymbol`	Declares a symbol. See 19.6.19 BrigDirectiveSymbol (p. 294).
`BrigDirectiveVariableInit`	Declares a variable. See 19.6.20 BrigDirectiveVariableInit (p. 295).
`BrigDirectiveVersion`	HSAIL version and target information. See 19.6.21 BrigDirectiveVersion (p. 296).

See also 19.5 Block Sections in BRIG (p. 279).

19.6.2 Declarations and Definitions in the Same Compilation Unit

If the same symbol (variable, image, sampler, or function) is both declared and defined in the same compilation unit, all references to the symbol in the Brig representation must refer to the definition, even if the definition comes after the use. If there are multiple declarations and no definitions, then all uses must refer to the first declaration in lexical order. This avoids a finalizer needing to traverse the entire Brig compilation unit to determine if there is a definition for a symbol in the compilation unit.

19.6.3 BrigDirectiveBase

The `.directive` section should not include any items of type `BrigDirectiveBase`. The declaration is only a helper type so that tools processing Brig can use pointers to a `BrigDirectiveBase` as a generic pointer to any directive, which all start with this field layout.

Syntax is:

```
struct BrigDirectiveBase {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Can be any member of the `BrigDirectiveKinds` enumeration. See 19.2.2 Section Structure Kinds (p. 259).
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the directive appears.

19.6.4 BrigDirectiveCallableBase

The `.directive` section should not include any items of type `BrigDirectiveCallableBase`. The declaration is only a helper type so that tools processing Brig can use pointers to a `BrigDirectiveCallableBase` as a generic pointer to any `BrigDirectiveExecutable` or `BrigDirectiveSignature` directive, which start with this field layout.

Syntax is:

```
struct BrigDirectiveCallableBase {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
  uint16_t inArgCount;
  uint16_t outArgCount;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_KERNEL`, `BRIG_DIRECTIVE_FUNCTION`, or `BRIG_DIRECTIVE_SIGNATURE`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the first byte of the first instruction appears.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section where the name of the kernel, function, or signature appears.

- `uint16_t inArgCount` — The number of input parameters to the kernel, function, or signature.
- `uint16_t outArgCount` — The number of output (returned value) parameters of the kernel, function, or signature. For kernels, this will always be 0.

19.6.5 BrigDirectiveArgScope

`BrigDirectiveArgScope` is an argument scope.

Syntax is:

```
struct BrigDirectiveArgScope {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_ARGSCOPE_END` or `BRIG_DIRECTIVE_ARGSCOPE_START`.
- `BrigCodeOffset32_t code` — The place in the `.code` section where the argument scope starts or ends, respectively.

19.6.6 BrigDirectiveComment

`BrigDirectiveComment` is a comment string.

Syntax is:

```
struct BrigDirectiveComment {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_COMMENT`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the comment appears.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section where the text of the comment (including the `//`) appears.

19.6.7 BrigDirectiveControl

`BrigDirectiveControl` specifies assorted compiler controls, such as the maximum number of work-items in a work-group. For information on placement and scope of

control directives, see 14.6 Control Directives for Low-Level Performance Tuning (p. 235).

Syntax is:

```
struct BrigDirectiveControl {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigControlDirective16_t control;
  BrigType16_t type;
  uint16_t reserved;
  uint16_t valueCount;
  BrigOperandOffset32_t values[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_CONTROL`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the control applies.
- `BrigControlDirective16_t control` — Used to select the type of control, maximum size of a work-group, number of work-groups per compute unit, or controls on optimization. See 19.2.6 BrigControlDirective (p. 262).
- `BrigType16_t type` — The Brig type of the values.
- `uint16_t reserved` — Must be 0.
- `uint16_t valueCount` — Number of values.
- `BrigOperandOffset32_t values[1]` — A variable-sized array of values that apply to the control directive. Must have `valueCount` elements. Each element is a byte offset to operands in the `.operand` section. The operand must either be `BRIG_OPERAND_IMMED` or `BRIG_OPERAND_WAVESIZE`.

19.6.8 BrigDirectiveExecutable

`BrigDirectiveExecutable` describes a kernel or function.

Kernels are arranged in the `.directive` section as:

1. `BrigDirectiveExecutable` with kind of `BRIG_DIRECTIVE_KERNEL`
2. Zero or more source parameters
3. Zero or more directives that are scoped to the kernel
4. The next top-level item

Functions are arranged in the `.directive` section as:

1. `BrigDirectiveExecutable` with kind of `BRIG_DIRECTIVE_FUNCTION`
2. Zero or more destination parameters

3. Zero or more source parameters
4. Zero or more directives that are scoped to the function
5. The next top-level item

Syntax is:

```
struct BrigDirectiveExecutable {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
  uint16_t inArgCount;
  uint16_t outArgCount;
  BrigDirectiveOffset32_t firstInArg;
  BrigDirectiveOffset32_t firstScopedDirective;
  BrigDirectiveOffset32_t nextTopLevelDirective;
  uint32_t instCount;
  BrigExecutableModifier8_t modifier;
  uint8_t reserved[3];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_KERNEL` or `BRIG_DIRECTIVE_FUNCTION`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the first operation of the kernel or function appears. If this is a function declaration, or a kernel or function definition with an empty code block, then this should be the offset of the next operation in the `.code` section, or equal to the `.code` section size if there are no more operations.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section giving the name of the kernel or function.
- `uint16_t inArgCount` — The number of input parameters to the kernel or function.
- `uint16_t outArgCount` — The number of output parameters from the function. Must be 0 for kernels.
- `BrigDirectiveOffset32_t firstInArg` — Byte offset to the location in the `.directive` section of the first input parameter. If there are no input parameters, then this must be the same value as `firstScopedDirective`.
- `BrigDirectiveOffset32_t firstScopedDirective` — Byte offset to the location in the `.directive` section of the first directive inside the scope of this kernel or function. If this is a function declaration with no code block (indicated by `modifier` and `BRIG_EXECUTABLE_DECLARATION` being non-zero), or if the kernel or function definition has no directives, then this must be the same value as `nextTopLevelDirective`.

- `BrigDirectiveOffset32_t nextTopLevelDirective` — Byte offset to the location in the `.directive` section of the next directive outside the scope of this kernel or function. If there are no more directives, then this must be the size of the `.directive` section.
- `uint32_t instCount` — The number of instructions (not bytes) in this kernel or function. If a function or kernel with no instructions, or a function declaration, then this must be 0.
- `BrigExecutableModifier8_t modifier;` — Modifier for the kernel or function. The `BRIG_EXECUTABLE_LINKAGE` must be `BRIG_LINKAGE_NONE` for kernels. The `BRIG_EXECUTABLE_DECLARATION` must be 0 for kernels because they always have a code block. See 19.2.7 BrigExecutableModifierMask (p. 263).
- `uint8_t reserved[3]` — Must be 0.

19.6.9 BrigDirectiveExtension

`BrigDirectiveExtension` is used to enable a device-specific extension. For more information, see 14.1 extension Directive (p. 231).

Syntax is:

```
struct BrigDirectiveExtension {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_FBARRIER`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the fbarrier symbol is defined.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section where the name of the fbarrier appears.

19.6.10 BrigDirectiveFbarrier

`BrigDirectiveFbarrier` is used for fbarrier definitions.

Syntax is:

```
struct BrigDirectiveFbarrier {
                uint16_t size;
                BrigDirectiveKinds16_t kind;
                BrigCodeOffset32_t code;
                BrigStringOffset32_t name;
                };
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_EXTENSION`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the extension applies.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section where the name of the extension appears.

19.6.11 BrigDirectiveFile

`BrigDirectiveFile` specifies the file descriptor. This is similar to the C preprocessor `#file`.

The `BrigDirectiveFile` must appear before any `BrigDirectiveLoc` that uses the file ID.

For more information, see 14.3 file Directive (p. 234).

Syntax is:

```
struct BrigDirectiveFile {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  uint32_t fileid;
  BrigStringOffset32_t filename;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_FILE`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the `file` directive appears. See 14.3 file Directive (p. 234).
- `uint32_t fileid` — An integer value that can be used by a `BrigDirectiveLoc` to refer to the file.
- `BrigStringOffset32_t filename` — Byte offset to the place in the `.string` section where the name of the file appears.

19.6.12 BrigDirectiveImageInit

`BrigDirectiveImageInit` specifies the initializer for an image definition. For more information, see 7.1.4 Image Objects (p. 154).

Syntax is:

```
struct BrigDirectiveImageInit {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  uint32_t width;
  uint32_t height;
  uint32_t depth;
  uint32_t array;
  BrigImageOrder8_t order;
  BrigImageFormat8_t format;
  uint16_t reserved;
};
```

Fields are:

- `size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_IMAGEINIT`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the image initializer appears.
- `uint32_t width` — The image width; 0 if not specified.
- `uint32_t height` — The image height; 0 if not specified. This is always 0 for a 1D image.
- `uint32_t depth` — The image depth; 0 if not specified. This is always 0 for a 1D or 2D image.
- `uint32_t array` — The number of 1DA and 2DA images. Must be 1 for all other image types.
- `BrigImageOrder8_t order` — Order for the components. Components of an image can be reordered when values are read from or written to memory. A member of the BrigImageOrder enumeration. See 19.2.10 BrigImageOrder (p. 264).
- `BrigImageFormat8_t format` — Format for storing images. Images can be stored in assorted packed formats. A member of the BrigImageFormat enumeration. See 19.2.8 BrigImageFormat (p. 263).
- `uint16_t reserved` — Must be 0.

19.6.13 BrigDirectiveLabel

`BrigDirectiveLabel` declares a label.

Syntax is:

```
struct BrigDirectiveLabel {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_LABEL`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the label appears. Label directives cannot be at the top level. They must be inside the scope of a function or a kernel.
- `BrigStringOffset32_t name` — Byte offset to the place in the `.string` section table where the name of the label appears.

19.6.14 BrigDirectiveLabelList

`BrigDirectiveLabelList` is used for a `labeltargets` statement (see 8.4 Label Targets (labeltargets Statement) (p. 183)) and a variable label initializer (see 8.1.2 Indirect Branches (p. 178)).

This is a variable-size directive.

Syntax is:

```
struct BrigDirectiveLabelList {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigDirectiveOffset32_t label;
  uint16_t labelCount;
  uint16_t reserved;
  BrigDirectiveOffset32_t labels[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_LABEL_LIST` or `BRIG_DIRECTIVE_LABEL_INIT`.
- `BrigCodeOffset32_t code` — Byte offset to the place in the `.code` section where the label list appears.
- `BrigDirectiveOffset32_t label` — The offset to the directive for the label used to refer to the label list in the HSAIL textual representation.
- `uint16_t labelCount` — Number of labels in the list.
- `uint16_t reserved` — Must be 0.
- `BrigDirectiveOffset32_t labels[1]` — A variable-sized array of labels containing the offsets of each label in the `.directive` section. Must have `labelCount` elements.

19.6.15 BrigDirectiveLoc

`BrigDirectiveLoc` specifies the source-level line position. This is similar to the `.line` cpp directive. For more information, see 14.4 loc Directive (p. 234).

Syntax is:

```
struct BrigDirectiveLoc {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  uint32_t fileid;
  uint32_t line;
  uint32_t column;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_LOC`.
- `BrigCodeOffset32_t code` — Byte offset into the `.code` section. The instructions starting at that offset up to the next `BrigDirectiveLoc` are assumed to correspond to the source location defined by this directive.
- `uint32_t fileid` — An integer value that is used to look up the corresponding `fileid` in a `BrigDirectiveFile` statement, which then gives the name of the source file.
- `uint32_t line` — The finalizer and other tools should assume that the operation at `code` corresponds to `line`. Multiple `BrigDirectiveLoc` statements can refer to the same `line`.
- `uint32_t column` — The finalizer and other tools should assume that the operation at `code` corresponds to `column`. Multiple `BrigDirectiveLoc` statements can refer to the same `column`.

19.6.16 BrigDirectivePragma

`BrigDirectivePragma` allows additional directives to be given to control the compiler. For more information, see 14.5 pragma Directive (p. 235).

Syntax is:

```
struct BrigDirectivePragma {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_PRAGMA`.
- `BrigCodeOffset32_t code` — Byte offset into the place in the `.code` section where the pragma applies.
- `BrigStringOffset32_t name` — Byte offset into the place in the `.string` section where the text of the pragma appears.

19.6.17 BrigDirectiveSamplerInit

`BrigDirectiveSamplerInit` declares a sampler object. For more information, see 7.1.7 Sampler Objects (p. 160).

Syntax is:

```
struct BrigDirectiveSamplerInit {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigSamplerModifier8_t modifier;
  BrigSamplerBoundaryMode8_t boundaryU;
  BrigSamplerBoundaryMode8_t boundaryV;
  BrigSamplerBoundaryMode8_t boundaryW;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_SAMPLER_INIT`.
- `BrigSamplerModifier8_t modifier` — Modifier for the sampler. See 19.2.23 BrigSamplerModifierMask (p. 272).
- `BrigSamplerBoundaryMode8_t boundaryU` — The boundary mode for borders for the first coordinate. Must be a member of the `BrigSamplerBoundaryMode` enumeration. See 19.2.20 BrigSamplerBoundaryMode (p. 271).
- `BrigSamplerBoundaryMode8_t boundaryV` — The boundary mode for borders for the second coordinate. Must be a member of the `BrigSamplerBoundaryMode` enumeration. See 19.2.20 BrigSamplerBoundaryMode (p. 271). This is ignored if this is a 1D image, and must be `BRIG_BOUNDARY_UNKNOWN`.
- `BrigSamplerBoundaryMode8_t boundaryW` — The boundary mode for borders for the third coordinate. Must be a member of the `BrigSamplerBoundaryMode` enumeration. See 19.2.20 BrigSamplerBoundaryMode (p. 271). This is ignored if this is a 1D or 2D image, and must be `BRIG_BOUNDARY_UNKNOWN`.

  For arrays (1DA or 2DA), this coordinate is the array selector. It is never normalized and always uses `BRIG_BOUNDARY_CLAMP`.

19.6.18 BrigDirectiveSignature

`BrigDirectiveSignature` declares a function signature. This is a variable-size directive. The size depends on the number of input and output types passed as arguments to the function.

For more information, see 10.3.3 Function Signature (p. 206).

Syntax is:

```
struct BrigDirectiveSignature {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
  uint16_t inArgCount;
  uint16_t outArgCount;
  struct {
    BrigType16_t type;
    uint8_t align;
    BrigSymbolModifier8_t modifier;
    uint32_t dimLo;
    uint32_t dimHi;
  } args[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_SIGNATURE`.
- `BrigCodeOffset32_t code` — Byte offset into the place in the `.code` section where the function signature applies.
- `BrigStringOffset32_t name` — Byte offset into the place in the `.string` section where the name appears.
- `uint16_t inArgCount` — The number of source types.
- `uint16_t outArgCount` — The number of return types.
- `args[]` — Variable-sized. Must be be allocated with (`inArgCount` + `outArgCount`) elements that contain descriptions of each signature argument.
  - `BrigType16_t type` — The Brig type of the argument.
  - `uint8_t align` — The required alignment of the argument in bytes. Possible values are 0, 1, 2, 4, 8, and 16. The value 0 indicates natural alignment, 1 means any alignment, 2 is any even byte boundary, 4 is any multiple of four, and so forth.
  - `BrigSymbolModifier8_t modifier` — Modifier for the argument. Only the last argument of a function can be a flexible array indicated by `BRIG_SYMBOL_FLEX_ARRAY`. Linkage must always be `BRIG_LINKAGE_NONE` for signature arguments. A signature argument cannot be marked `BRIG_SYMBOL_CONST`. See 19.2.25 BrigSymbolModifierMask (p. 273).
  - `uint32_t dimLo` — `dimLo` is combined with `dimHi` to form a 64-bit dimension for the argument:

    ```
    dim = (uint64_t(dimHi) << 32) | uint64_t(dimLo)
    ```

    `dim` must be 0 unless `modifier` indicates the argument is an array that is not a flexible array.
  - `uint32_t dimHi` — See above.

19.6.19 BrigDirectiveSymbol

`BrigDirectiveSymbol` is used for variable, sampler, and image declarations or definitions.

Syntax is:

```
struct BrigDirectiveSymbol {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t name;
  BrigDirectiveOffset32_t init;
  BrigType16_t type;
  BrigSegment8_t segment;
  uint8_t align;
  uint32_t dimLo;
  uint32_t dimHi;
  BrigSymbolModifier8_t modifier;
  uint8_t reserved[3];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_VARIABLE`, `BRIG_DIRECTIVE_IMAGE`, or `BRIG_DIRECTIVE_SAMPLER`.
- `BrigCodeOffset32_t code` — Byte offset into the `.code` section showing where the variable, image, or sampler symbol is declared or defined.
- `BrigStringOffset32_t name` — Byte offset into the place in the `.string` section where the symbol name appears.
- `BrigDirectiveOffset32_t init` — Must be 0 if there is no initializer; otherwise must reference `BRIG_DIRECTIVE_VARIABLE_INIT`, `BRIG_DIRECTIVE_IMAGE_INIT`, or `BRIG_DIRECTIVE_SAMPLER_INIT` if kind is `BRIG_DIRECTIVE_VARIABLE`, `BRIG_DIRECTIVE_IMAGE`, or `BRIG_DIRECTIVE_SAMPLER`, respectively.
- `BrigType 16_t type` — The Brig type of of the symbol. Must be `BRIG_TYPE_ROIMG` or `BRIG_TYPE_RWIMG` if and only if kind is `BRIG_DIRECTIVE_IMAGE`. Must be `BRIG_TYPE_SAMP` if and only if kind is `BRIG_DIRECTIVE_SAMPLER`.
- `BrigSegment8_t segment` — Segment that will hold the symbol. A member of the `BrigSegment` enumeration. See 19.2.24 BrigSegment (p. 272).
- `uint8_t align` — The required symbol alignment in bytes. Possible values are 0, 1, 2, 4, 8, and 16. The value 0 means natural alignment, 1 means any alignment, 2 is any even byte boundary, 4 is any multiple of four, and so forth.
- `uint32_t dimLo` — `dimLo` is combined with `dimHi` to form a 64-bit value:

  ```
  dim = (uint64_t(dimHi) << 32) | uint64_t(dimLo)
  ```

  The `BRIG_SYMBOL_ARRAY` and `BRIG_SYMBOL_FLEX_ARRAY` bits of the `modifier` field indicate if the symbol is an array, and if so if it is a flexible array (an array without a specified size) respectively. See 19.2.25 BrigSymbolModifierMask (p. 273).

If the symbol is an array with a size, then `dim` must be the number of elements in the array. If the symbol is not an array or is a flexible array, then `dim` must be 0. An array declared in the textual form without a size, but with an initializer, is not considered a flexible array. In this case, the value of `dim` is related to the directive referenced by the `init` field: if `BrigDirectiveVariableInit`, then `dim` must be the same value as `elementCount` field of the `BrigDirectiveVariableInit`; if `BrigDirectiveImageInit` or `BrigDirectiveSamplerInit`, then `dim` must be 1 because HSAIL allows only a single initializer for image and sampler objects.

- `uint32_t dimHi` — See above.
- `BrigSymbolModifier8_t modifier` — Modifier for the symbol. See 19.2.25 BrigSymbolModifierMask (p. 273).
- `uint8_t reserved[3]` — Must be 0.

19.6.20 BrigDirectiveVariableInit

`BrigDirectiveVariableInit` declares a variable.

A `BrigDirectiveVariableInit` gives alignment, type, and other information about the variable. A `BrigDirectiveVariableInit` is used for most variable declarations (but not for image-related objects).

Syntax is:

```
struct BrigDirectiveVariableInit {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  BrigStringOffset32_t data;
  uint32_t elementCount;
  BrigType16_t type;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_VARIABLE_INIT`.
- `BrigCodeOffset32_t code` — Byte offset into the place in the `.code` section where the initializer of the variable is declared.
- `BrigStringOffset32_t data` — Byte offset into the place in the `.string` section where the data value is available. The data size must be `elementCount` times the size of `type`.
- `uint32_t elementCount` — Number of initialization values that are present in the data. If `elementCount` is less than the greater of 1 and the `dim` value of the `BrigDirectiveSymbol` referencing this `BrigDirectiveVariableInit`, then when the runtime allocates and initializes the variable, any elements after `elementCount` must be initialized to 0. This is true even if `elementCount` is 0.
- `BrigType16_t type` — Type of each element.
- `uint16_t reserved` — Must be 0.

19.6.21 BrigDirectiveVersion

`BrigDirectiveVersion` specifies the HSAIL version and target information. For more information, see Chapter 15 version Statement (p. 243).

This directive must be the first directive in the `.directive` section.

Additional `BrigDirectiveVersion` directives are allowed, but they must all be the same value.

Syntax is:

```
struct BrigDirectiveVersion {
  uint16_t size;
  BrigDirectiveKinds16_t kind;
  BrigCodeOffset32_t code;
  uint16_t major;
  uint16_t minor;
  BrigProfile8_t profile;
  BrigMachineModel8_t machineModel;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigDirectiveKinds16_t kind` — Must be `BRIG_DIRECTIVE_VERSION`.
- `BrigCodeOffset32_t code` — Byte offset into the `.code` section showing where the version information applies.
- `uint16_t major` — The HSAIL major version. Must be `BRIG_VERSION_MAJOR` to be compatible with this revision of the HSAIL specification. See 19.2.27 BrigVersion (p. 276).
- `uint16_t minor` — The HSAIL minor version. When generating BRIG, must be `BRIG_VERSION_MINOR`. When consuming BRIG, must be less than or equal to `BRIG_VERSION_MINOR` to be compatible with this revision of the HSAIL specification. See 19.2.27 BrigVersion (p. 276).
- `BrigProfile8_t profile` — The profile. A member of the `BrigProfile` enumeration. See 19.2.18 BrigProfile (p. 270)
- `BrigMachineModel8_t machineModel` — The machine model. A member of the `BrigMachineModel` enumeration. See 19.2.12 BrigMachineModel (p. 265).
- `uint16_t reserved;` — Must be 0.

19.7 .code Section

The `.code` section must start with a `BrigSectionHeader` entry. See 19.3 Section Header (p. 277).

19.7.1 Overview

All HSAIL operations that might generate ISA are stored in the `.code` section.

All `BrigInst*` start with the `BrigInstBase` layout. See 19.7.2 BrigInstBase (p. 297).

Each operation in the `.code` section starts with a 32-bit word containing a `size` (in bytes) and an operation `kind` (format).

The `size` and `kind` are followed by an `opcode`, a `type`, and five offsets to operands. As with text format, the destination operand is first, followed by source operands. Operations that use fewer than five operands must set the remaining operand fields to 0. As a special exception, the `BrigInstNone` format only contains the `size` and `kind`. See 19.7.15 BrigInstNone (p. 307).

If an operation does not produce a value, the value `BRIG_TYPE_NONE` must be used for the `type`. Examples of these operations are `call`, `ret`, `cbr`, and `brn`, among others.

The table below shows the possible formats for the operations in alphabetical order. Every operation uses one of these formats.

Table 19–3 Formats of Operations in the .code Section

Name	Description
`BrigInstBase`	Field layout used for most operations. See 19.7.2 BrigInstBase (p. 297).
`BrigInstBasic`	Field layout used for all operations that require no extra modifier information. See 19.7.3 BrigInstBasic (p. 298).
`BrigInstAddr`	Address operations. See 19.7.4 BrigInstAddr (p. 298).
`BrigInstAtomic`	Atomic operations. See 19.7.5 BrigInstAtomic (p. 299).
`BrigInstAtomicImage`	Atomic image-related operations. See 19.7.6 BrigInstAtomicImage (p. 300).
`BrigInstBar`	Barrier and sync operations. See 19.7.7 BrigInstBar (p. 301).
`BrigInstBr`	Branch and call operations with certain modifiers. See 19.7.8 BrigInstBr (p. 301).
`BrigInstCmp`	Compare operation. See 19.7.9 BrigInstCmp (p. 302).
`BrigInstCvt`	Convert operation. See 19.7.10 BrigInstCvt (p. 303).
`BrigInstFbar`	Fbarrier operations. See 19.7.11 BrigInstFbar (p. 304).
`BrigInstImage`	Image-related operations. See 19.7.12 BrigInstImage (p. 304).
`BrigInstMem`	Memory operations other than load/store. See 19.7.13 BrigInstMem (p. 305).
`BrigInstMod`	Operations with a single modifier, such as a rounding mode. See 19.7.14 BrigInstMod (p. 306).
`BrigInstNone`	Special operation that is always ignored. See 19.7.15 BrigInstNone (p. 307).
`BrigInstSeg`	Segment checking and segment conversion operations. See 19.7.16 BrigInstSeg (p. 307).
`BrigInstSourceType`	Operations that have different types for their destination and source operands. See 19.7.17 BrigInstSourceType (p. 308).

19.7.2 BrigInstBase

The `.code` section should not include any items of type `BrigInstBase`. The declaration is only a helper type so that tools processing Brig can use pointers to a `BrigInstBase` as a generic pointer to any instruction (except `BrigInstBase`), which all start with this field layout.

Syntax is:

```
struct BrigInstBase {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_BASE`.
- `BrigOpcode16_t opcode` — Opcode associated with the operation.
- `BrigType16_t type` — Data type of the destination of the operation. If the operation does not use a structure that provides a source type, this can also be the type of the source operands.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.

19.7.3 BrigInstBasic

The `BrigInstBasic` format is used for all operations that require no extra modifier information.

Syntax is:

```
struct BrigInstBasic {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_BASIC`.
- `BrigOpcode16_t opcode` — Opcode associated with the operation.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.

19.7.4 BrigInstAddr

The `BrigInstAddr` format is used for address operations.

Syntax is:

```
struct BrigInstAddr {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigSegment8_t segment;
  uint8_t reserved[3];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_ADDR`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the operation does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See 19.2.24 BrigSegment (p. 272).
- `uint8_t reserved` — Must be 0.

19.7.5 BrigInstAtomic

The `BrigInstAtomic` format is used for atomic and atomic no return operations.

Syntax is:

```
struct BrigInstAtomic {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigSegment8_t segment;
  BrigMemorySemantic8_t memorySemantic;
  BrigAtomicOperation8_t atomicOperation;
  int8_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `uint16_t kind` — Must be `BRIG_INST_ATOMIC`.
- `BrigOpcode16_t opcode` — Opcode for an `atomic` or `atomicnoret` operation.
- `BrigDataType16_t type` — Data type of the memory destination of the operation.

- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the operation does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See 19.2.24 BrigSegment (p. 272).
- `BrigMemorySemantic8_t memorySemantic` — Memory semantics of the atomic operation. See 19.2.15 BrigMemorySemantic (p. 266).
- `BrigAtomicOperation8_t atomicOperation` — An atomic suboperation such as `add` or `or`.
- `int8_t reserved` — Must be 0.

19.7.6 BrigInstAtomicImage

The `BrigInstAtomicImage` format is used for atomic image and atomic image no return operations.

This format is similar to `BrigInstAtomic`, but includes an additional field for image information.

Syntax is:

```
struct BrigInstAtomicImage {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t imageType;
  BrigType16_t coordType;
  BrigImageGeometry8_t geometry;
  BrigAtomicOperation8_t atomicOperation;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_ATOMIC_IMAGE`.
- `BrigOpcode16_t opcode` — Opcode for an `atomicimage` or `atomicimagenoret` operation.
- `BrigType8_t type` — Data type of the memory destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t imageType` — Type of the image. Must be `BRIG_TYPE_RWIMG`.
- `BrigType16_t coordType` — Type of the coordinates. Must be `BRIG_TYPE_U32`.
- `BrigImageGeometry8_t geometry` — Image geometry: 1D, 2D, 3D, 1DA, 2DA, or 1DB. See 19.2.9 BrigImageGeometry (p. 264).

- `BrigAtomicOperation8_t atomicOperation` — An atomic suboperation such as and or `or`. See 19.2.4 BrigAtomicOperation (p. 261).
- `uint16_t reserved;` — Must be 0.

19.7.7 BrigInstBar

The `BrigInstBar` format is used for the `barrier` and `sync` operations.

Syntax is:

```
struct BrigInstBar {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigMemoryFence8_t memoryFence;
  BrigWidth8_t width;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_BAR`.
- `BrigOpcode16_t opcode` — Opcode, such as `BRIG_OPCODE_BARRIER` or `BRIG_OPCODE_SYNC`.
- `BrigType8_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigMemoryFence8_t memoryFence;` — Memory fence specified by the operation. See 19.2.13 BrigMemoryFence (p. 265).
- `BrigWidth8_t width` — The width modifier. If the operation does not support the width modifier, then this must be `BRIG_WIDTH_NONE`. If the operation supports the width modifier but does not specify it, then this must be `BRIG_WIDTH_ALL` (the default for `BRIG_OPCODE_BARRIER`). If the operation specifies `width(all)`, then this must be `BRIG_WIDTH_ALL`. Otherwise, this must be the corresponding enumeration value from `BrigWidth` (see 19.2.28 BrigWidth (p. 276)).
- `uint16_t reserved` — Must be 0.

19.7.8 BrigInstBr

The `BrigInstBr` format is used for the branch and call operations.

Syntax is:

```
struct BrigInstBr {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigAluModifier16_t modifier;
  BrigWidth8_t width;
  uint8_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_BR`.
- `BrigOpcode16_t opcode` — Opcode: `BRIG_OPCODE_BRN`, `BRIG_OPCODE_CBR`, or `BRIG_OPCODE_CALL`.
- `BrigType8_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. See 19.2.3 BrigAluModifierMask (p. 261).
- `BrigWidth8_t width` — The width modifier. If the operation does not support the width modifier, then this must be `BRIG_WIDTH_ALL` (the default for the direct branch and direct call operations). If the operation supports the width modifier but does not specify it, then this must be `BRIG_WIDTH_1` (the default for indirect branch and indirect call operations). If the operation specifies `width(all)`, then this must be `BRIG_WIDTH_ALL`. Otherwise, this must be the corresponding enumeration value from `BrigWidth` (see 19.2.28 BrigWidth (p. 276)).
- `uint8_t reserved` — Must be 0.

19.7.9 BrigInstCmp

The `BrigInstCmp` format is used for compare operations. The compare operation needs a special format because it has a comparison operator and a second type.

Syntax is:

```
struct BrigInstCmp {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t sourceType;
  BrigAluModifier16_t modifier;
  BrigCompareOperation8_t compare;
  BrigPack8_t pack;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_CMP`.
- `BrigOpcode16_t opcode` — Opcode. Must be `BRIG_OPCODE_CMP`.
- `BrigType16_t type` — Data type of the destination of the compare operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. See 19.2.3 BrigAluModifierMask (p. 261).
- `BrigCompareOperation8_t compare` — The specific comparison (greater than, less than, and so forth).
- `BrigPack8_t pack` — Packing control. See 19.2.17 BrigPack (p. 270).
- `uint16_t reserved` — Must be 0.

For packed compares, the value of `type` must be `u` with the same length as `sourceType`.

19.7.10 BrigInstCvt

The `BrigInstCvt` format is used for convert operations.

Syntax is:

```
struct BrigInstCvt {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t sourceType;
  BrigAluModifier16_t modifier;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_CVT`.
- `BrigOpcode16_t opcode` — Opcode. Must be `BRIG_OPCODE_CVT`.
- `BrigType16_t type` — Data type of the destination of the convert operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. See 19.2.3 BrigAluModifierMask (p. 261).

19.7.11 BrigInstFbar

The `BrigInstFbar` format is used for fbarrier operations.

Syntax is:

```
struct BrigInstFbar {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigMemoryFence8_t memoryFence;
  BrigWidth8_t width;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_FBAR`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation. Must be `BRIG_TYPE_NONE`, because fbarrier operations do not have a destination type.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigMemoryFence8_t memoryFence` — Memory fence specified by the operation. If the operation does not support the memory fence modifier, then this must be `BRIG_FENCE_NONE`. If the operation does support the memory fence modifier but does not specify it, then this must be `BRIG_FENCE_BOTH` (the default for fbarrier operations). See See 19.2.13 BrigMemoryFence (p. 265).
- `BrigWidth8_t width` — The width modifier. If the operation does not support the width modifier, then this must be `BRIG_WIDTH_NONE`. If the operation supports the width modifier but does not specify it, then this must be `BRIG_WIDTH_WAVESIZE` (the default for fbarrier operations). (See 19.2.28 BrigWidth (p. 276).
- `uint16_t reserved` — Must be 0.

19.7.12 BrigInstImage

The `BrigInstImage` format is used for the image operations.

Syntax is:

```
struct BrigInstImage {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t imageType;
  BrigType16_t coordType;
  BrigImageGeometry8_t geometry;
  uint8_t reserved[3];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_IMAGE`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t imageType` — Type of the image. Must be`BRIG_INST_ROIMG` or `BRIG_INST_RWIMG`.
- `BrigType16_t coordType` — Type of the coordinates.
- `BrigImageGeometry8_t geometry` — Image geometry: 1D, 2D, 3D, 1DA, 2DA, or 1DB. See 19.2.9 BrigImageGeometry (p. 264).
- `uint8_t reserved[3]` — Must be 0.

19.7.13 BrigInstMem

The `BrigInstMem` format is used for memory operations.

Syntax is:

```
struct BrigInstMem {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigSegment8_t segment;
  BrigMemoryModifier8_t modifier;
  uint8_t equivClass;
  BrigWidth8_t width;
}
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_MEM`.

- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the operation does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See 19.2.24 BrigSegment (p. 272).
- `BrigMemoryModifier8_t modifier` — Memory modifier flags of the operation. See 19.2.14 BrigMemoryModifierMask (p. 266).
- `uint8_t equivClass` — Memory equivalence class. If no equivalence class is explicitly given, then the value must be set to 0, which is general memory that can interact with all other equivalence classes. See 6.1.4 Equivalence Classes (p. 126).
- `BrigWidth8_t width` — The width modifier. If the operation does not support the width modifier, then this must be `BRIG_WIDTH_NONE`. If the operation supports the width modifier but does not specify it, then this must be `BRIG_WIDTH_1` (the default for load operations). If the operation specifies `width(all)`, then this must be `BRIG_WIDTH_ALL`. Otherwise, this must be the corresponding enumeration value from `BrigWidth` (see 19.2.28 BrigWidth (p. 276)).

19.7.14 BrigInstMod

The `BrigInstMod` format is used for ALU operations with a modifier.

Syntax is:

```
struct BrigInstMod {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigAluModifier16_t modifier;
  BrigPack8_t pack;
  uint8_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_MOD`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.

- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. If an operation does not have a rounding modifier, then `BRIG_ALU_ROUND` must be set to `BRIG_ROUND_NONE`. If an operation does not have an `ftz` modifier, then `BRIG_ALU_FTZ` must not be set. See 19.2.3 BrigAluModifierMask (p. 261).
- `BrigPack8_t pack` — Packing control. If the operation does not have a packing modifier, this must be set to `BRIG_PACK_NONE`. See 19.2.17 BrigPack (p. 270).
- `uint8_t reserved` — Must be 0.

19.7.15 BrigInstNone

The `BrigInstNone` format is a special format that allows a tool to overwrite long operations with short ones, provided the tool sets the remaining words to be a `BrigInstNone` format.

`BrigInstNone` is the one structure that does not contain all the fields of `BrigInstBase`. This allows it to be as small as as four bytes. It can also be used to cover any number of 4-bytes by setting the `size` field accordingly, in which case any bytes after the `BrigInstNone` structure must be set to 0.

Syntax is:

```
struct BrigInstNone {
  uint16_t size;
  BrigInstKinds16_t kind;
};
```

Fields are:

- `uint16_t size` — The number of bytes covered by the `BrigInstNone` structure. Must be a multiple of 4. If `size` is greater than the size of the `BrigInstNone` structure (4 bytes), then any extra bytes must be set to 0.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_NONE` (which has the value 0).

19.7.16 BrigInstSeg

The `BrigInstSeg` format is used for segment checking, segment conversion, query image, and query sampler operations.

Syntax is:

```
struct BrigInstSeg {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t sourceType;
  BrigSegment8_t segment;
  uint8_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_SEG`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t sourceType` — Type of the source. For the `query_image` and `query_sampler` operations, must be `BRIG_TYPE_NONE`, because they have no source type specified.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the operation does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See 19.2.24 BrigSegment (p. 272).
- `uint8_t reserved` — Must be 0.

19.7.17 BrigInstSourceType

The `BrigInstSourceType` format is used for operations that have different types for their destination and source operands.

Syntax is:

```
struct BrigInstSourceType {
  uint16_t size;
  BrigInstKinds16_t kind;
  BrigOpcode16_t opcode;
  BrigType16_t type;
  BrigOperandOffset32_t operands[5];
  BrigType16_t sourceType;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigInstKinds16_t kind` — Must be `BRIG_INST_SOURCE_TYPE`.
- `BrigOpcode16_t opcode` — Opcode.
- `BrigType16_t type` — Data type of the destination of the operation.
- `BrigOperandOffset32_t operands[5]` — Byte offset to operands in the `.operand` section. Unused operands must be 0.
- `BrigType16_t sourceType` — Type of the source.
- `uint16_t reserved` — Must be 0.

19.8 .operand Section

The `.operand` section must start with a `BrigSectionHeader` entry. See 19.3 Section Header (p. 277).

19.8.1 Overview

The `.operand` section contains the operands that appear in the HSAIL operations.

It is legal, but not required, for multiple operation operands to refer to the same operand in the `.operand` section if they are syntactically the same. This can reduce the size of the `.operand` section.

All operand structures start with the `BrigOperandBase` field layout. See 19.8.2 BrigOperandBase (p. 309).

The table below shows the structures in the `.operand` section in alphabetical order.

Table 19–4 Structures in the .operand Section

Name	Description
`BrigOperandBase`	Helper type. See 19.8.2 BrigOperandBase (p. 309).
`BrigOperandAddress`	Used for [name]. See 19.8.3 BrigOperandAddress (p. 310).
`BrigOperandImmed`	A numeric value. See 19.8.4 BrigOperandImmed (p. 310).
`BrigOperandList`	Used for return and input arguments. See 19.8.5 BrigOperandList (p. 311).
`BrigOperandRef`	A single argument. See 19.8.6 BrigOperandRef (p. 312).
`BrigOperandReg`	A register (`c`, `s`, or `d`). See 19.8.7 BrigOperandReg (p. 313).
`BrigOperandRegVector`	Used for (register, register) or (register, register, register, register). See 19.8.8 BrigOperandRegVector (p. 313).
`BrigOperandWavesize`	The `wavesize` operand. See 19.8.9 BrigOperandWavesize (p. 314).

19.8.2 BrigOperandBase

The `.operand` section should not include any items of type `BrigOperandBase`. The declaration is only a helper type so that tools processing Brig can use pointers to a `BrigOperandBase` as a generic pointer to any operand, which all start with this field layout.

Syntax is:

```
struct BrigOperandBase {
  uint16_t size;
  BrigOperandKinds16_t kind;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Can be any member of the `BrigOperandKinds` enumeration. See 19.2.2 Section Structure Kinds (p. 259).

19.8.3 BrigOperandAddress

`BrigOperandAddress` is used for `[name]`.

Syntax is:

```
struct BrigOperandAddress {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigDirectiveOffset32_t symbol;
  BrigStringOffset32_t reg;
  uint32_t offsetLo;
  uint32_t offsetHi;
  BrigType16_t type;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_ADDRESS`.
- `BrigDirectiveOffset32_t symbol` — Offset in `.directive` section pointing to the symbol directive for the name. If the compilation unit has both a declaration and a definition, then this must point to the definition, even if it occurs lexically later than the operation. See 19.6.2 Declarations and Definitions in the Same Compilation Unit (p. 282).
- `BrigStringOffset32_t reg` — Byte offset in the `.string` section to the register name.
- `uint32_t offsetLo` — Byte offset to add to the address.

  `offsetLo` is combined with `offsetHi` to form a 64-bit offset for the address:

  ```
  offset = (uint64_t(offsetHi) << 32) | uint64_t(offsetLo)
  ```

  If the address size is 32 bits, then `dimHi` must be 0.
- `uint32_t offsetHi` — See above. Must be 0 if type size is 32.
- `BrigType16_t type` — Must be `u32` or `u64` depending on the segment of the operation referencing this operand and the machine model in the `version` directive. This is not the compound type of the memory referenced by the address that is available from the operation's `type` field. See Table 2–3 (p. 20).
- `uint16_t reserved` — Must be 0.

19.8.4 BrigOperandImmed

`BrigOperandImmed` is used for a numeric value.

Syntax is:

```
struct BrigOperandImmed {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigType16_t type;
  uint16_t byteCount;
  uint8_t bytes[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_IMMED`.
- `BrigType16_t type` — The compound type of the source operand. See 4.17 Operands (p. 50).
- `uint16_t byteCount` — The number of bytes in the immediate value. Must match the byte size of `type` and can be 1, 2, 4, 8, or 16.
- `uint8_t bytes[]` — Variable-sized. Must be allocated with `(((byteCount + 3) / 4) * 4)` elements. Any elements after `byteCount - 1` must be 0. The immediate value is composed of `byteCount` bytes from `bytes`, with index 0 being the least significant bits.

19.8.5 BrigOperandList

`BrigOperandList` is used for the list of arguments to a function or a list of function names or function signatures. Lists of function names or function signatures are needed when the call statement has a list of possible targets.

Syntax is:

```
struct BrigOperandList {
  uint16_t size;
  BrigOperandKinds16_t kind;
  uint16_t elementCount;
  uint16_t reserved;
  BrigDirectiveOffset32_t elements[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes, including the variable-sized `args` list.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_ARGUMENT_LIST` or `BRIG_OPERAND_FUNCTION_LIST`.

- `uint16_t elementCount` — Number of elements in `elements`. Can be 0.
- `uint16_t reserved` — Must be 0.
- `BrigDirectiveOffset32_t elements[]` — Variable-sized array. Must be allocated with `elementCount` elements.

  If `BRIG_OPERAND_ARGUMENT_LIST`, must reference `BRIG_DIRECTIVE_SYMBOL` with `BRIG_SEGMENT_ARG` segment.

  If `BRIG_OPERAND_FUNCTION_LIST`, must reference `BRIG_DIRECTIVE_FUNCTION` directive. If the compilation unit has both a declaration and a definition of the function, then this must point to the definition, even if it occurs lexically later than the operation. See 19.6.2 Declarations and Definitions in the Same Compilation Unit (p. 282).

19.8.6 BrigOperandRef

`BrigOperandRef` is used for a label, argument, function, signature, or fbarrier.

Syntax is:

```
struct BrigOperandRef {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigDirectiveOffset32_t ref;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_LABEL_REF`, `BRIG_OPERAND_ARGUMENT_REF`, `BRIG_OPERAND_FUNCTION_REF`, `BRIG_OPERAND_SIGNATURE_REF`, or `BRIG_OPERAND_SIGNATURE_FBARRIER`.
- `BrigDirectiveOffset32_t ref` — Byte offset to the place in the `.directive` section where the label, argument, function, signature, or fbarrier is declared.

  If `kind` is `BRIG_OPERAND_LABEL_REF`, must reference a directive with kind of `BRIG_DIRECTIVE_LABEL` if the label is for an operation, or `BRIG_DIRECTIVE_LABEL_LIST` if the label is for a `labeltargets` statement. If for a `labeltargets` statement, the `label` field of `BrigDirectiveLabelList` will reference the label.

  If `kind` is `BRIG_OPERAND_ARGUMENT_REF`, must reference a directive with kind of `BRIG_DIRECTIVE_SYMBOL` with the `BRIG_SEGMENT_ARG` segment.

  If `kind` is `BRIG_OPERAND_FUNCTION_REF`, must reference a directive with kind of `BRIG_DIRECTIVE_FUNCTION`. If the compilation unit has both a declaration and a definition of the function, then this must point to the definition, even if it occurs lexically later than the operation. See 19.6.2 Declarations and Definitions in the Same Compilation Unit (p. 282).

  If `kind` is `BRIG_OPERAND_SIGNATURE_REF`, must reference a directive with kind of `BRIG_DIRECTIVE_FUNCTION` or `BRIG_DIRECTIVE_SIGNATURE`.

  If `kind` is `BRIG_OPERAND_SIGNATURE_FBARRIER`, must reference a directive with kind of `BRIG_DIRECTIVE_FBARRIER`.

19.8.7 BrigOperandReg

`BrigOperandReg` is used for a register (`c`, `s`, or `d`).

Syntax is:

```
struct BrigOperandReg {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigStringOffset32_t reg;
  BrigType16_t type;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_REG`.
- `BrigStringOffset32_t reg` — Byte offset to the place in the `.string` section where the register name occurs.
- `BrigType16_t type` — The compound type of the operand. See 4.17 Operands (p. 50).
- `uint16_t reserved` — Must be 0.

19.8.8 BrigOperandRegVector

`BrigOperandRegVector` is used for certain operations to allow vector register forms.

Syntax is:

```
struct BrigOperandRegVector {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigType16_t type;
  uint16_t regCount;
  BrigStringOffset32_t regs[1];
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_REG_VECTOR`.
- `BrigType16_t type` — The compound type of each of the registers in the vector register operand. See 4.17 Operands (p. 50).
- `uint16_t regCount` — Number of registers. Can be 2, 3, or 4.
- `BrigStringOffset32_t regs[]` — Variable-sized array. Must be allocated with `regCount` elements. Each element is the byte offset in the `.string` section to the register name.

19.8.9 BrigOperandWavesize

`BrigOperandWavesize` is the `wavesize` operand, which is a compile-time value equal to the size of a wavefront.

Syntax is:

```
struct BrigOperandWavesize {
  uint16_t size;
  BrigOperandKinds16_t kind;
  BrigType16_t type;
  uint16_t reserved;
};
```

Fields are:

- `uint16_t size` — Size of the structure in bytes.
- `BrigOperandKinds16_t kind` — Must be `BRIG_OPERAND_WAVESIZE`.
- `BrigType16_t type` — The compound type of the source operand. See 4.17 Operands (p. 50).
- `uint16_t reserved` — Must be 0.

19.9 .debug Section

The `.debug` section must start with a `BrigSectionHeader` entry. See 19.3 Section Header (p. 277).

The `.debug` section contains one or more block sections. See 19.5 Block Sections in BRIG (p. 279).

19.10 BRIG Syntax for Operations

This section describes the BRIG syntax for operations.

19.10.1 BRIG Syntax for Arithmetic Operations

19.10.1.1 BRIG Syntax for Integer Arithmetic Operations

Table 19–5 BRIG Syntax for Integer Arithmetic Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_ABS	`BRIG_INST_BASIC` if default modifier is used; otherwise `BRIG_INST_MOD`	*dest*	*src*	
BRIG_OPCODE_ADD	`BRIG_INST_BASIC` if default modifier is used; otherwise `BRIG_INST_MOD`	*dest*	*src*	*src*
BRIG_OPCODE_BORROW	`BRIG_INST_BASIC`	*dest*	*src*	*src*
BRIG_OPCODE_CARRY	`BRIG_INST_BASIC`	*dest*	*src*	*src*
BRIG_OPCODE_DIV	`BRIG_INST_BASIC`	*dest*	*src*	*src*

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_MAX	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*
BRIG_OPCODE_MIN	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*
BRIG_OPCODE_MUL	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*
BRIG_OPCODE_MULHI	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*
BRIG_OPCODE_NEG	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	
BRIG_OPCODE_REM	BRIG_INST_BASIC	*dest*	*src*	*src*
BRIG_OPCODE_SUB	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.1.2 BRIG Syntax for Integer Optimization Operation

Table 19–6 BRIG Syntax for Integer Optimization Operation

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_MAD	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.1.3 BRIG Syntax for 24-Bit Integer Optimization Operations

Table 19–7 BRIG Syntax for 24-Bit Integer Optimization Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_MAD24	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*
BRIG_OPCODE_MAD24HI	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*
BRIG_OPCODE_MUL24	BRIG_INST_BASIC	*dest*	*src*	*src*	
BRIG_OPCODE_MUL24HI	BRIG_INST_BASIC	*dest*	*src*	*src*	

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.1.4 BRIG Syntax for Integer Shift Operations

Table 19–8 BRIG Syntax for Integer Optimization Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_SHL	BRIG_INST_BASIC	*dest*	*src*	*src*
BRIG_OPCODE_SHR				

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

The pack field of BRIG_INST_BASIC should be set to BRIG_PACK_PS for packed source types and to BRIG_PACK_NONE otherwise.

19.10.1.5 BRIG Syntax for Individual Bit Operations

Table 19–9 BRIG Syntax for Individual Bit Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_AND	BRIG_INST_BASIC	*dest*	*src*	*src*
BRIG_OPCODE_NOT	BRIG_INST_BASIC	*dest*	*src*	
BRIG_OPCODE_OR	BRIG_INST_BASIC	*dest*	*src*	*src*
BRIG_OPCODE_POPCOUNT	BRIG_INST_SOURCE_TYPE	*dest*	*src*	
BRIG_OPCODE_XOR	BRIG_INST_BASIC	*dest*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.1.6 BRIG Syntax for Bit String Operations

Table 19–10 BRIG Syntax for Bit String Operations

Opcode	Format	Oper. 0	Oper. 1	Oper. 2	Oper. 3	Oper. 4
BRIG_OPCODE_BITEXTRACT	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*	
BRIG_OPCODE_BITINSERT	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*	*src*
BRIG_OPCODE_BITMASK	BRIG_INST_BASIC	*dest*	*src*	*src*		
BRIG_OPCODE_BITREV	BRIG_INST_BASIC	*dest*	*src*			
BRIG_OPCODE_BITSELECT	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*	
BRIG_OPCODE_FIRSTBIT	BRIG_INST_SOURCE_TYPE	*dest*	*src*			
BRIG_OPCODE_LASTBIT	BRIG_INST_SOURCE_TYPE	*dest*	*src*			

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.1.7 BRIG Syntax for Copy (Move) Operations

Table 19–11 BRIG Syntax for Copy (Move) Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_COMBINE	BRIG_INST_SOURCE_TYPE	*dest*	*vector*	
BRIG_OPCODE_EXPAND	BRIG_INST_SOURCE_TYPE	*vector*	*src*	
BRIG_OPCODE_LDA	BRIG_INST_MEM	*dest*	*address*	
BRIG_OPCODE_LDC	BRIG_INST_BASIC	*dest*	*label-or-function*	
BRIG_OPCODE_MOV	BRIG_INST_BASIC	*dest*	*src*	

dest: must be BRIG_OPERAND_REG.

vector: must be BRIG_OPERAND_REG_VECTOR.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

address: must be BRIG_OPERAND_ADDRESS.

label-or-function: must be BRIG_OPERAND_LABELREF or BRIG_OPERAND_FUNCTION_REF.

19.10.1.8 BRIG Syntax for Packed Data Operations

Table 19–12 BRIG Syntax for Packed Data Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_SHUFFLE	BRIG_INST_BASIC	*dest*	*src*	*src*	*number*
BRIG_OPCODE_UNPACKHI	BRIG_INST_BASIC	*dest*	*src*	*src*	
BRIG_OPCODE_UNPACKLO	BRIG_INST_BASIC	*dest*	*src*	*src*	
BRIG_OPCODE_PACK	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*src*	*src*
BRIG_OPCODE_UNPACK	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*src*	

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

number: must be BRIG_OPERAND_IMMED.

The pack field of BRIG_INST_BASIC should be set to BRIG_PACK_NONE.

19.10.1.9 BRIG Syntax for Bit Conditional Move (cmov) Operation

Table 19–13 BRIG Syntax for Bit Conditional Move (cmov) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_CMOV	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src:must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

The pack field of BRIG_INST_BASIC should be set to BRIG_PACK_PP for packed source types and to BRIG_PACK_NONE otherwise.

19.10.1.10 BRIG Syntax for Floating-Point Arithmetic Operations

Table 19–14 BRIG Syntax for Floating-Point Arithmetic Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_ABS	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		
BRIG_OPCODE_ADD	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_CEIL	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		
BRIG_OPCODE_COPYSIGN	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_DIV	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_FLOOR	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		
BRIG_OPCODE_FMA	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	*src*
BRIG_OPCODE_FRACT	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		
BRIG_OPCODE_MAX	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_MIN	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_MUL	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_NEG	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_RINT	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	BRIG_INST_BASIC	*dest*	*src*	
BRIG_OPCODE_SQRT	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		
BRIG_OPCODE_SUB	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*	*src*	
BRIG_OPCODE_TRUNC	BRIG_INST_BASIC if default modifier is used; otherwise BRIG_INST_MOD	*dest*	*src*		

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG or BRIG_OPERAND_IMMED.

19.10.1.11 BRIG Syntax for Floating-Point Classify (class) Operation

Table 19–15 BRIG Syntax for Floating-Point Classify (class) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_CLASS	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG or BRIG_OPERAND_IMMED.

19.10.1.12 BRIG Syntax for Floating-Point Native Functions Operations

Table 19–16 BRIG Syntax for Floating-Point Native Functions Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_NCOS	BRIG_INST_BASIC	*dest*	*src*		
BRIG_OPCODE_NEXP2		*dest*	*src*		
BRIG_OPCODE_NFMA		*dest*	*src*	*src*	*src*
BRIG_OPCODE_NLOG2		*dest*	*src*		
BRIG_OPCODE_NRCP		*dest*	*src*		
BRIG_OPCODE_NRSIN		*dest*	*src*		
BRIG_OPCODE_NRSQRT		*dest*	*src*		
BRIG_OPCODE_NSQRT		*dest*	*src*		

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG or BRIG_OPERAND_IMMED.

19.10.1.13 BRIG Syntax for Multimedia Operations

Table 19–17 BRIG Syntax for Multimedia Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_BITALIGN	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*
BRIG_OPCODE_BYTEALIGN	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*
BRIG_OPCODE_LERP	BRIG_INST_BASIC	*dest*	*src*	*src*	*src*
BRIG_OPCODE_PACKCVT	BRIG_INST_SOURCE_TYPE	*dest*	*vector*		
BRIG_OPCODE_UNPACKCVT	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*number*	
BRIG_OPCODE_SAD	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*src*	*src*
BRIG_OPCODE_SADHI	BRIG_INST_SOURCE_TYPE	*dest*	*src*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

vector: must be BRIG_OPERAND_REG_VECTOR.

number: must be BRIG_OPERAND_IMMED with value 0, 1, 2, 3, or 4.

19.10.1.14 BRIG Syntax for Segment Checking (segmentp) Operation

Table 19–18 BRIG Syntax for Segment Checking (segmentp) Operation

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_SEGMENTP	BRIG_INST_SEG	*dest*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG or BRIG_OPERAND_IMMED.

19.10.1.15 BRIG Syntax for Segment Conversion Operations

Table 19–19 BRIG Syntax for Segment Conversion Operations

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_FTOS	BRIG_INST_SEG	*dest*	*src*
BRIG_OPCODE_STOF			

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG or BRIG_OPERAND_IMMED.

19.10.1.16 BRIG Syntax for Compare (cmp) Operation

Table 19–20 BRIG Syntax for Compare (cmp) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_CMP	BRIG_INST_CMP	*dest*	*src*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

The pack field of BRIG_INST_CMP should be set to BRIG_PACK_PP for packed source types and to BRIG_PACK_NONE otherwise.

19.10.1.17 BRIG Syntax for Conversion (cvt) Operation

Table 19–21 BRIG Syntax for Conversion (cvt) Operation

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_CVT	BRIG_INST_CVT	*dest*	*src*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.2 BRIG Syntax for Memory Operations

Table 19–22 BRIG Syntax for Memory Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_LD	BRIG_INST_MEM	*reg-or-vector*	*address*		
BRIG_OPCODE_ST	BRIG_INST_MEM	*reg-or-vector-or-num*	*address*		
BRIG_OPCODE_ATOMIC	BRIG_INST_ATOMIC	*dest*	*address*	*src*	
BRIG_OPCODE_ATOMIC (for **atomic_cas**)	BRIG_INST_ATOMIC	*dest*	*address*	*src*	*src*
BRIG_OPCODE_ATOMICNORET	BRIG_INST_ATOMIC	*address*	*src*		
BRIG_OPCODE_ATOMICNORET (for **atomicnoret_cas**)	BRIG_INST_ATOMIC	*address*	*src*	*src*	

reg-or-vector: must be BRIG_OPERAND_REG or BRIG_OPERAND_REG_VECTOR.

address: must be BRIG_OPERAND_ADDRESS.

reg-or-vector-or-num: must be BRIG_OPERAND_REG, BRIG_OPERAND_REG_VECTOR, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.3 BRIG Syntax for Image Operations

Table 19–23 BRIG Syntax for Image Operations

Opcode	Format	Oper. 0	Oper. 1	Oper. 2	Oper. 3	Oper. 4
BRIG_OPCODE_RDIMAGE	BRIG_INST_IMAGE	*4-vector-reg*	*image*	*sampler*	*reg-or-vector*	
BRIG_OPCODE_LDIMAGE	BRIG_INST_IMAGE	*4-vector-reg*	*image*	*reg-or-vector*		
BRIG_OPCODE_STIMAGE	BRIG_INST_IMAGE	*4-vector-reg*	*image*	*reg-or-vector*		
BRIG_OPCODE_ATOMICIMAGE	BRIG_INST_ATOMIC_IMAGE	*dest*	*image*	*reg-or-vector*	*src*	
BRIG_OPCODE_ATOMICIMAGE (for **atomicimage_cas**)	BRIG_INST_ATOMIC_IMAGE	*dest*	*image*	*reg-or-vector*	*src*	*src*
BRIG_OPCODE_ATOMICIMAGENORET	BRIG_INST_ATOMIC_IMAGE	*image*	*reg-or-vector*	*src*		
BRIG_OPCODE_ATOMICIMAGENORET (for **atomicimagenoret_cas**)	BRIG_INST_ATOMIC_IMAGE	*image*	*reg-or-vector*	*src*	*src*	
BRIG_OPCODE_QUERYIMAGEWIDTH	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYIMAGEHEIGHT	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYIMAGEDEPTH	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYIMAGEARRAY	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYIMAGEORDER	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYIMAGEFORMAT	BRIG_INST_SOURCE_TYPE	*dest*	*image*			
BRIG_OPCODE_QUERYSAMPLERCOORD	BRIG_INST_SOURCE_TYPE	*dest*	*sampler*			
BRIG_OPCODE_QUERYSAMPLERFILTER	BRIG_INST_SEG	*dest*	*sampler*			

4-vector-reg: must be BRIG_OPERAND_REG_VECTOR.

image: must be BRIG_OPERAND_REG.

sampler: must be BRIG_OPERAND_REG.

reg-or-vector: must be BRIG_OPERAND_REG or BRIG_OPERAND_REG_VECTOR.

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.4 BRIG Syntax for Branch Operations

If the operation is a direct BRIG_OPCODE_BRN or BRIG_OPCODE_CBR operation and BRIG_OPERAND_LABEL_REF is used, the ref in BrigOperandRef must be to the BrigDirectiveLabel for the label.

If the operation is an indirect BRIG_OPCODE_BRN or BRIG_OPCODE_CBR operation and BRIG_OPERAND_LABEL_REF is used, the ref must be to the BrigDirectiveLabelList for the labeltargets statement that has the label. The label field of BrigDirectiveLabelList contains the offset of the BrigDirectiveLabel for the label.

Table 19–24 BRIG Syntax for Branch Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_BRN	BRIG_INST_BR	*label-or-register*		
BRIG_OPCODE_BRN	BRIG_INST_BR	*dest*	*label-or-address*	
BRIG_OPCODE_CBR	BRIG_INST_BR	*dest*	*label-or-register*	
BRIG_OPCODE_CBR	BRIG_INST_BR	*dest*	*dest*	*label-or-address*

label-or-register: must be BRIG_OPERAND_LABEL_REF or BRIG_OPERAND_REG.

dest: must be BRIG_OPERAND_REG.

label-or-address: must be BRIG_OPERAND_LABEL_REF or BRIG_OPERAND_ADDRESS.

19.10.5 BRIG Syntax for Parallel Synchronization and Communication Operations

Table 19–25 BRIG Syntax for Parallel Synchronization and Communication Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_SYNC	BRIG_INST_BAR			
BRIG_OPCODE_BARRIER	BRIG_INST_BAR			
BRIG_OPCODEINITFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_JOINFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_WAITFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_ARRIVEFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_LEAVEFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_RELEASEFBAR	BRIG_INST_FBAR	*fbarrier-or-reg*		
BRIG_OPCODE_LDF	BRIG_INST_BASIC	*dest*	*fbarrier*	
BRIG_OPCODE_COUNTLANE	BRIG_INST_BASIC	*dest*	*src*	
BRIG_OPCODE_COUNTUPLANE	BRIG_INST_BASIC	*dest*		
BRIG_OPCODE_MASKLANE	BRIG_INST_BASIC	*dest*	*src*	
BRIG_OPCODE_SENDLANE	BRIG_INST_BASIC	*dest*	*src*	*src*
BRIG_OPCODE_RECEIVELANE	BRIG_INST_BASIC	*dest*	*src*	*src*

fbarrier-or-reg: must be BRIG_OPERAND_FBARRIER_REF or BRIG_OPERAND_FBARRIER_REG.

fbarrier: must be BRIG_OPERAND_FBARRIER_REF.

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

19.10.6 BRIG Syntax for Operations Related to Functions

Table 19–26 BRIG Syntax for Operations Related to Functions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
BRIG_OPCODE_RET	BRIG_INST_BASIC					
BRIG_OPCODE_ALLOCA	BRIG_INST_SEG	*dest*	*src*			
BRIG_OPCODE_SYSCALL	BRIG_INST_BASIC	*dest*	*number*	*src*	*src*	*src*
BRIG_OPCODE_CALL	BRIG_INST_BR	*out-args*	*func-or-reg*	*in-args*		
BRIG_OPCODE_CALL	BRIG_INST_BR	*out-args*	*func-or-reg*	*in-args*	*funcs*	

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

number: must be BRIG_OPERAND_IMMED or BRIG_OPERAND_WAVESIZE.

out-args: output arguments; must be BRIG_OPERAND_ARGUMENT_LIST.

in-args: input arguments; must be BRIG_OPERAND_ARGUMENT_LIST.

func-or-reg: must be BRIG_OPERAND_FUNCTION_REF or BRIG_OPERAND_REG.

funcs: must be BRIG_OPERAND_FUNCTION_LIST.

19.10.7 BRIG Syntax for Special Operations

Table 19–27 BRIG Syntax for Special Operations

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_CLEARDETECTEXCEPT	BRIG_INST_BASIC	*exceptionsNumber*	
BRIG_OPCODE_CLOCK	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_CUID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_CURRENTWORKGROUPSIZE	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_DEBUGTRAP	BRIG_INST_BASIC	*src*	
BRIG_OPCODE_DIM	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_DISPATCHID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_DISPATCHPTR	BRIG_INST_SEG	*dest*	
BRIG_OPCODE_GETDETECTEXCEPT	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_GRIDGROUPS	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_GRIDSIZE	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_LANEID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_MAXCUID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_MAXWAVEID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_NOP	BRIG_INST_BASIC		
BRIG_OPCODE_NULLPTR	BRIG_INST_SEG	*dest*	

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_QID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_QPTR	BRIG_INST_SEG	*dest*	
BRIG_OPCODE_SETDETECTEXCEPT	BRIG_INST_BASIC	*exceptionsNumber*	
BRIG_OPCODE_WAVEID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_WORKGROUPID	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_WORKGROUPSIZE	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_WORKITEMABSID	BRIG_INST_BASIC	*dest*	*dimNumber*
BRIG_OPCODE_WORKITEMFLATABSID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_WORKITEMFLATID	BRIG_INST_BASIC	*dest*	
BRIG_OPCODE_WORKITEMID	BRIG_INST_BASIC	*dest*	*dimNumber*

dest: must be BRIG_OPERAND_REG.

src: must be BRIG_OPERAND_REG, BRIG_OPERAND_IMMED, or BRIG_OPERAND_WAVESIZE.

dimNumber: must be BRIG_OPERAND_IMMED with the value 0, 1, or 2 corresponding to the X, Y, and Z dimensions respectively.

exceptionsNumber: must be BRIG_OPERAND_IMMED. bit:0=INVALID_OPERATION, bit:1=DIVIDE_BY_ZERO, bit:2=OVERFLOW, bit:3=UNDERFLOW, bit:4=INEXACT; all other bits are ignored.

Appendix A

HSAIL Grammar in Extended Backus-Naur Form (EBNF)

This appendix shows the HSAIL grammar in Extended Backus–Naur Form (EBNF). Symbol meanings are:

- :== assignment
- [] option
- {} repetition
- | alternative

```
sequenceOfPrograms      ::= program { program }
program                 ::= version topLevelStatements
topLevelStatements      ::= { topLevelStatement } topLevelStatement
topLevelStatement       ::= directive
                          | TOKEN_COMMENT
                          | globalDecl
                          | kernel
                          | function
globalDecl              ::= globalInitializableDecl
                          | globalUninitializableDecl
                          | declprefix globalImageDecl
                          | declprefix globalReadOnlyImageDecl
                          | declprefix globalSamplerDecl
                          | functionDecl
                          | functionSignature
directive               ::= pragma
                          | extension
                          | block
                          | control
                          | fileDecl
bodyStatements          ::= { bodyStatement } bodyStatement
bodyStatement           ::= TOKEN_COMMENT
                          | block
                          | pragma
                          | declprefix localInitializableDecl
                          | declprefix localUninitializableDecl
                          | argblock
                          | location
                          | label
                          | labeltarget
                          | operation
bodyStatementNested     ::= TOKEN_COMMENT
                          | pragma
                          | block
                          | declprefix initializableDecl
                          | declprefix uninitializableDecl
                          | location
                          | label
                          | labeltarget
                          | operation
argblock                ::= "{" argStatements "}"
argStatements           ::= { argStatement } argStatement
argStatement            ::= bodyStatementNested
                          | declprefix argUninitializableDecl
                          | call
operation               ::= Instruction0
                          | Instruction1
                          | Instruction2
                          | Instruction3
                          | Instruction4
                          | cmp
                          | mul
                          | combine
                          | expand
                          | mov
                          | seg
```

```
                                | alloca
                                | sad
                                | packcvt
                                | unpackcvt
                                | bitextract
                                | pack
                                | unpack
                                | lda
                                | ldc
                                | imageread
                                | ld
                                | st
                                | cvt
                                | atomicRet
                                | atomicNoRet
                                | imageAtomicRet
                                | imageAtomicNoRet
                                | sync
                                | bar
                                | fbar
                                | syscall
                                | ret
                                | branch
                                | query
                                | imagestore
                                | imageload
sectionitems                ::= { sectionitem } sectionitem
block                       ::= section sectionitems endsection
declprefix                  ::= [ alignment externOrStatic |
                                  externOrStatic alignment |
                                  externOrStatic | alignment | const
                                  alignment externOrStatic | const
                                  externOrStatic alignment | const
                                  externOrStatic | const alignment |
                                  alignment const externOrStatic |
                                  externOrStatic const alignment |
                                  externOrStatic const | alignment const |
                                  alignment externOrStatic const |
                                  externOrStatic alignment const ]
globalSamplerDecl           ::= initializableAddress "_Samp"
                                  TOKEN_GLOBAL_IDENTIFIER
                                  optArrayDimensions sobInitializer ";"
globalImageDecl             ::= initializableAddress "_rwimg"
                                  TOKEN_GLOBAL_IDENTIFIER
                                  optArrayDimensions imageInitializer ";"
globalReadOnlyImageDecl     ::= initializableAddress "_roimg"
                                  TOKEN_GLOBAL_IDENTIFIER
                                  optArrayDimensions imageInitializer ";"
version                     ::= "version" TOKEN_INTEGER_CONSTANT ":"
                                          TOKEN_INTEGER_CONSTANT ":"
                                          profile ":"
                                          machineModel ";"
profile                     ::= "$full"
                                | "$base"
machineModel                ::= "$small"
                                | "$large"
```

```
addressableOperand      ::= "[" nonRegister "]"
nonRegister             ::= TOKEN_GLOBAL_IDENTIFIER
                          | TOKEN_LOCAL_IDENTIFIER
identifier              ::= TOKEN_GLOBAL_IDENTIFIER
                          | TOKEN_LOCAL_IDENTIFIER
                          | Register
identifierList          ::= { identifier "," } identifier
decimalConstant         ::= "+" TOKEN_INTEGER_CONSTANT
                          | "-" TOKEN_INTEGER_CONSTANT
                          | TOKEN_INTEGER_CONSTANT
decimalList             ::= { decimalConstant "," } decimalConstant
decimalInitializer      ::= "{" decimalList "}"
                          | decimalConstant
floatList               ::= { TOKEN_DOUBLE_CONSTANT "," }
                            TOKEN_DOUBLE_CONSTANT
labelInitializer        ::= "{" labelList "}"
floatInitializer        ::= "{" floatList "}"
                          | TOKEN_DOUBLE_CONSTANT
singleList              ::= { TOKEN_SINGLE_CONSTANT "," }
                            TOKEN_SINGLE_CONSTANT
singleInitializer       ::= "{" singleList "}"
                          | TOKEN_SINGLE_CONSTANT
packedConstant          ::= dataTypeId "(" decimalList ")"
                          | dataTypeId "(" singleList ")"
                          | dataTypeId "(" floatList ")"
packedList              ::= { packedConstant "," } packedConstant
packedInitializer       ::= "{" packedList "}"
                          | packedConstant
addressSpaceIdentifier  ::= "_readonly"
                          | "_kernarg"
                          | "_global"
                          | "_private"
                          | "_arg"
                          | "_group"
                          | "_spill"
optaddressSpace         ::= [ addressSpaceIdentifier ]
extDataTypeId           ::= dataTypeId
                          | "_roimg"
                          | "_rwimg"
                          | "_samp"
vectorToken             ::= "_v2"
                          | "_v3"
                          | "_v4"
alignment               ::= "align" TOKEN_INTEGER_CONSTANT
arrayDimensionSet       ::= ( "[" TOKEN_INTEGER_CONSTANT |
                              arrayDimensionSet "["
                              TOKEN_INTEGER_CONSTANT | "[" ) "]"
optArrayDimensions      ::= [ arrayDimensionSet ]
optInitializer          ::= [ "=" ( decimalInitializer |
                              floatInitializer | singleInitializer |
                              labelInitializer ) ]
fileDecl                ::= "file" TOKEN_INTEGER_CONSTANT
                              TOKEN_STRING ";"
argumentDecl            ::= declprefix "arg" extDataTypeId
                              TOKEN_LOCAL_IDENTIFIER
                              optArrayDimensions
```

```
kernelArgumentDecl       ::= declprefix "kernarg" extDataTypeId
                             TOKEN_LOCAL_IDENTIFIER
                             optArrayDimensions
argumentListBody         ::= [ { argumentDecl "," } argumentDecl ]
kernelArgumentListBody   ::= [ { kernelArgumentDecl "," }
                             kernelArgumentDecl ]
argList                  ::= "(" argumentListBody ")"
returnArgList            ::= "(" argumentListBody ")"
kernelArgumentList       ::= "(" kernelArgumentListBody ")"
signatureArguments       ::= [ signatureArgumentList [
                             signatureArgumentList ] ]
twoCallArgs              ::= callArgs [ callArgs ]
functionDefinition       ::= declprefix "function"
                             TOKEN_GLOBAL_IDENTIFIER returnArgList
                             argList
function                 ::= functionDefinition codeblock
functionDecl             ::= declprefix "function"
                             TOKEN_GLOBAL_IDENTIFIER returnArgList
                             argList ";"
signatureType            ::= ( alignment "arg" extDataTypeId
                             TOKEN_LOCAL_IDENTIFIER | "arg"
                             extDataTypeId TOKEN_LOCAL_IDENTIFIER |
                             alignment "arg" extDataTypeId | "arg"
                             extDataTypeId ) optArrayDimensions
signatureTypes           ::= { signatureType "," } signatureType
signatureArgumentList    ::= "(" ( signatureTypes ")" | ")" )
functionSignature        ::= "signature" TOKEN_GLOBAL_IDENTIFIER
                             signatureArguments ";"
section                  ::= "block" TOKEN_STRING
sectionitem              ::= (
                             | "blockstring" TOKEN_STRING
                             | "blocknumeric" dataTypeId decimalList
                             | "blocknumeric" dataTypeId packedList
                             | "blocknumeric" dataTypeId singleList
                             | "blocknumeric" dataTypeId floatList ) ";"
endsection               ::= "endblock" ";"
kernelName               ::= "kernel" TOKEN_GLOBAL_IDENTIFIER
kernelheader             ::= kernelName kernelArgumentList
kernel                   ::= kernelheader codeblock
codeblockend             ::= "}" ";"
codeblock                ::= "{" [ bodyStatements ] codeblockend
labelList                ::= { TOKEN_LABEL "," } TOKEN_LABEL
imageInitializer         ::= [ "=" "{" imageList "}" ]
sobInitializer           ::= [ "=" "{" soblist "}" ]
imageList                ::= { imageInit "," } imageInit
soblist                  ::= { sobInit "," } sobInit
imageInit                ::= "format" "=" TOKEN_PROPERTY
                           | "order" "=" TOKEN_PROPERTY
                           | tobNumeric "=" TOKEN_INTEGER_CONSTANT
sobInit                  ::= ( "coord" | "filter" | "boundaryU" |
                             "boundaryV" | "boundaryW" ) "="
                             TOKEN_PROPERTY
globalInitializableDecl  ::= declprefix initializableAddress dataTypeId
                             TOKEN_GLOBAL_IDENTIFIER optArrayDimensions
                             optInitializer ";"
localInitializableDecl   ::= initializableAddress dataTypeId
```

```
                                TOKEN_LOCAL_IDENTIFIER optArrayDimensions
                                optInitializer ";"
Register              ::= TOKEN_CREGISTER
                        | TOKEN_DREGISTER
                        | TOKEN_QREGISTER
                        | TOKEN_SREGISTER
labeltarget           ::= label "labeltargets" labelList ";"
control ::= "enablebreakexceptions" baseOperand ";"
          | "enabledetectexceptions" baseOperand ";"
          | "maxdynamicgroupsize" baseOperand ";"
          | "maxflatgridsize" baseOperand "," baseOperand ","
            baseOperand ";"
          | "maxflatworkgroupsize" baseOperand "," baseOperand ","
            baseOperand ";"
          | "requestedworkgroupspercu" baseOperand ";"
          | "requireddim" baseOperand ";"
          | "requiredgridsize" baseOperand "," baseOperand","
            baseOperand ";"
          | "requiredworkgroupsize " baseOperand "," baseOperand ","
            baseOperand ";"
          | "requirenopartialworkgroups" ";"
pragma                ::= "pragma" TOKEN_STRING ";"
extension             ::= "extension" TOKEN_STRING ";"
externOrStatic        ::= "extern"
                        | "static"
const                 ::= "const"
globalUninitializableAddress ::= "private"
                               | "group"
localUninitializableAddress  ::= "private"
                               | "group"
                               | "spill"
initializableAddress         ::= "readonly"
                               | "global"
globalUninitializableDecl ::= globalUninitializableAddress dataTypeId
                              TOKEN_GLOBAL_IDENTIFIER optArrayDimensions ";"
localUninitializableDecl  ::= localUninitializableAddress dataTypeId
                              TOKEN_LOCAL_IDENTIFIER optArrayDimensions ";"
argUninitializableDecl    ::= "arg" dataTypeId identifier
                              optArrayDimensions ";"
location                  ::= "loc" TOKEN_INTEGER_CONSTANT
                              TOKEN_INTEGER_CONSTANT
                              TOKEN_INTEGER_CONSTANT ";"
label                     ::= TOKEN_LABEL ":"
baseOperand               ::= decimalConstant
                            | packedConstant
                            | TOKEN_DOUBLE_CONSTANT
                            | TOKEN_SINGLE_CONSTANT
                            | TOKEN_WAVESIZE
offsetAddressableOperand ::= "[" Register "+" TOKEN_INTEGER_CONSTANT "]"
                            | "[" Register "-" TOKEN_INTEGER_CONSTANT "]"
                            | "[" Register "]"
                            | "[" TOKEN_INTEGER_CONSTANT "]"
operand                   ::= baseOperand
                            | identifier
pairaddressableOperand    ::= addressableOperand
                              offsetAddressableOperand
```

```
memoryOperand            ::= addressableOperand
                           | offsetAddressableOperand
                           | pairaddressableOperand
arrayOperand             ::= operand
                           | arrayOperandList
arrayOperandList         ::= "(" identifierList ")"
Instruction1Opcode       ::= "clock"
                           | "countuplane"
                           | "cuid"
                           | "debugtrap"
                           | "dim"
                           | "dispatchid"
                           | "cleardetectexcept"
                           | "getdetectexcept"
                           | "setdetectexcept"
                           | "laneid"
                           | "maxcuid"
                           | "maxwaveid"
                           | "qid"
                           | "waveid"
                           | "workitemflatabsid"
                           | "workitemflatid"
Instruction2Opcode       ::= "abs"
                           | "bitmask"
                           | "bitrev"
                           | "countlane"
                           | "currentworkgroupsize"
                           | "ncos"
                           | "neg"
                           | "nexp2"
                           | "nlog2"
                           | "nrcp"
                           | "nrsqrt"
                           | "nsin"
                           | "nsqrt"
                           | "gridgroups"
                           | "gridsize"
                           | "masklane"
                           | "mov"
                           | "not"
                           | "sqrt"
                           | "workgroupid"
                           | "workgroupsize"
                           | "workitemabsid"
                           | "workitemid"
Instruction2OpcodeFTZ    ::= "ceil"
                           | "floor"
                           | "fract"
                           | "rint"
                           | "trunc"
Instruction3Opcode       ::= "add"
                           | "borrow"
                           | "carry"
                           | "copysign"
                           | "div"
                           | "rem"
```

```
                                | "sub"
                                | "shl"
                                | "shr"
                                | "and"
                                | "or"
                                | "xor"
                                | "unpackhi"
                                | "unpacklo"
                                | "class"
                                | "sendlane"
                                | "receivelane"
Instruction3OpcodeFTZ       ::= "max"
                                | "min"
Instruction4Opcode          ::= "fma"
                                | "mad"
                                | "bitselect"
                                | "bitinsert"
                                | "shuffle"
                                | "cmov"
                                | "bitalign"
                                | "bytealign"
                                | "lerp"
Instruction4OpcodeFTZ       ::= "nfma"
atomicOperationId           ::= "_and"
                                | "_or"
                                | "_xor"
                                | "_exch"
                                | "_add"
                                | "_sub"
                                | "_inc"
                                | "_dec"
                                | "_max"
                                | "_min"
comparisonId                ::= "_eq"
                                | "_ne"
                                | "_lt"
                                | "_le"
                                | "_gt"
                                | "_ge"
                                | "_equ"
                                | "_neu"
                                | "_ltu"
                                | "_leu"
                                | "_gtu"
                                | "_geu"
                                | "_num"
                                | "_nan"
                                | "_seq"
                                | "_sne"
                                | "_slt"
                                | "_sle"
                                | "_sgt"
                                | "_sge"
                                | "_snum"
                                | "_snan"
                                | "_sequ"
```

```
                                | "_sneu"
                                | "_sltu"
                                | "_sleu"
                                | "_sgtu"
                                | "_sgeu"
intRounding                 ::= "_upi"
                                | "_downi"
                                | "_zeroi"
                                | "_neari"
                                | "_upi_sat"
                                | "_downi_sat"
                                | "_zeroi_sat"
                                | "_neari_sat"
floatRounding               ::= "_up"
                                | "_down"
                                | "_zero"
                                | "_near"
packing                     ::= "_pp"
                                | "_ps"
                                | "_sp"
                                | "_ss"
                                | "_s"
                                | "_p"
                                | "_pp_sat"
                                | "_ps_sat"
                                | "_sp_sat"
                                | "_ss_sat"
                                | "_s_sat"
                                | "_p_sat"
tobNumeric                  ::= "width"
                                | "height"
                                | "depth"
dataTypeId                  ::= "_u32"
                                | "_s32"
                                | "_s64"
                                | "_u64"
                                | "_b1"
                                | "_b32"
                                | "_f64"
                                | "_f32"
                                | "_b64"
                                | "_b8"
                                | "_b16"
                                | "_s8"
                                | "_s16"
                                | "_u8"
                                | "_u16"
                                | "_f16"
                                | "_b128"
                                | "_u8x4"
                                | "_s8x4"
                                | "_u16x2"
                                | "_s16x2"
                                | "_f16x2"
                                | "_f32x2"
                                | "_u8x8"
```

```
                                | "_s8x8"
                                | "_u16x4"
                                | "_s16x4"
                                | "_f16x4"
                                | "_u8x16"
                                | "_s8x16"
                                | "_u16x8"
                                | "_s16x8"
                                | "_f16x8"
                                | "_f32x4"
                                | "_s32x4"
                                | "_u32x4"
                                | "_f64x2"
                                | "_s64x2"
                                | "_u64x2"
optFTZ                        ::= [ "_ftz" ]
optRoundingMode               ::= [ roundingMode ]
optPacking                    ::= [ packing ]
roundingMode                  ::= "_ftz"
                                  | "_ftz" floatRounding
                                  | floatRounding
                                  | intRounding
Instruction0                  ::= "nop" ";"
Instruction1                  ::= ( "nullptr" | "dispatchptr" | "qptr" )
                                    optaddressSpace dataTypeId |
                                    Instruction1Opcode
                                    optRoundingMode dataTypeId ) operand ";"
Instruction2                  ::= ( "popcount" dataTypeId dataTypeId |
                                    "firstbit" dataTypeId dataTypeId |
                                    "lastbit" dataTypeId dataTypeId |
                                    Instruction2OpcodeFTZ optFTZ optPacking
                                    dataTypeId | Instruction2Opcode
                                    optRoundingMode optPacking dataTypeId )
                                    operand "," operand ";"
syscall                   ::= "syscall" dataTypeId operand "," baseOperand ","
                              operand "," operand "," operand ";"
Instruction3              ::= ( Instruction3Opcode optRoundingMode
                              optPacking dataTypeId |
                              Instruction3OpcodeFTZ optFTZ optPacking
                              dataTypeId ) operand "," operand ","
                              operand ";"
mul                             ::= ( "mul" optRoundingMode optPacking
                                    dataTypeId | "mulhi" optPacking
                                    dataTypeId | "mul24hi" dataTypeId |
                                    "mul24" dataTypeId | "mad24" dataTypeId
                                    operand "," | "mad24hi" dataTypeId
                                    operand "," ) operand "," operand ","
                                    operand ";"
Instruction4                  ::= ( Instruction4Opcode optRoundingMode |
                                    Instruction4OpcodeFTZ optFTZ optPacking
                                    ) dataTypeId operand "," operand ","
                                    operand "," operand ";"
packcvt                       ::= "packcvt" dataTypeId dataTypeId operand ","
                                    operand "," operand "," operand "," operand ";"
unpackcvt                     ::= "unpackcvt" dataTypeId dataTypeId operand ","
                                    operand "," operand ";"
```

```
sad                     ::= ( "sad" | "sadhi" ) dataTypeId dataTypeId
                            operand "," operand "," operand "," operand ";"
pack                    ::= "pack" dataTypeId dataTypeId operand ","
                            operand "," operand "," operand ";"
unpack                  ::= "unpack" dataTypeId dataTypeId operand ","
                            operand "," operand ";"
optWidth                ::= [ "_width" "(" ( "all" | TOKEN_WAVESIZE |
                            TOKEN_INTEGER_CONSTANT ) ")" ]
bitextract              ::= "bitextract" dataTypeId operand "," operand ","
                            operand "," operand "," operand ";"
branchop                ::= "cbr" optWidth
branchopbrn             ::= "brn" optWidth
branch                  ::= ( branchop operand "," TOKEN_LABEL |
                            branchop operand "," identifier |
                            branchop operand "," operand "," "["
                            identifier "]" | branchop operand ","
                            operand "," "[" TOKEN_LABEL "]" |
                            branchopbrn identifier | branchopbrn
                            TOKEN_LABEL | branchopbrn identifier ","
                            "[" identifier "]" | branchopbrn
                            identifier "," "[" TOKEN_LABEL "]" ) ";"
OperandList             ::= { operand "," } operand
callArgs                ::= "(" ( OperandList ")" | ")" )
call                    ::= "call" optWidth operand twoCallArgs
                            optCallTargets ";"
optCallTargets          ::= [ CallTargets ]
CallTargets             ::= "[" identifierList "]"
                          | TOKEN_GLOBAL_IDENTIFIER
optmemSemantic          ::= [ acqrel | acq ]
atomicRet               ::= ( "atomic" atomicOperationId
                            optaddressSpace optmemSemantic
                            dataTypeId operand "," memoryOperand |
                            "atomic_cas" optaddressSpace
                            optmemSemantic dataTypeId operand ","
                            memoryOperand "," operand ) "," operand
                            ";"
atomicNoRet             ::= ( "atomicnoret" atomicOperationId
                            optaddressSpace optmemSemantic
                            dataTypeId memoryOperand |
                            "atomicnoret_cas" optaddressSpace
                            optmemSemantic dataTypeId memoryOperand
                            "," operand ) "," operand ";"
cvtModifier1            ::= floatRounding
                          | "_ftz"
                          | "_ftz" floatRounding
                          | intRounding
                          | "_ftz" intRounding
optcvtModifier          ::= [ cvtModifier1 ]
cvt                     ::= "cvt" optcvtModifier dataTypeId
                            dataTypeId operand "," operand ";"
ldModifier              ::= { ( vectorToken | addressSpaceIdentifier
                          | acq | equiv |aligned ) }
equiv                   ::= "_equiv" "(" TOKEN_INTEGER_CONSTANT ")"
acq                     ::= "_acq"
                          | "_part_acq"
                          | "_rel"
```

```
                                | "_part_rel"
acqrel                        ::= "_ar"
                                | "_part_ar"
aligned                       ::== "aligned"
ldop                          ::= "ld" optWidth
ld                            ::= ldop ldModifier dataTypeId arrayOperand
                                  "," memoryOperand ";"
memFence                      ::= "_fnone"
                                | "_fgroup"
                                | "_fglobal"
                                | "_fboth"
                                | "_fpartial"
                                | "_fpartialboth"
optmemFence                 ::= [ memFence ]
sync                        ::= "sync" optmemFence ";"
bar                         ::= "barrier" optWidth optmemFence ";"
fbar                        ::= "initfbar" addressSpaceIdentifier operand ";"
                              | "joinfbar" optWidth addressSpaceIdentifier
                                operand ";"
                              | "waitfbar" optWidth optmemFence
                                addressSpaceIdentifier operand ";"
                              | "arrivefbar" optWidth optmemFence
                                addressSpaceIdentifier operand ";"
                              | "leavefbar" optWidth addressSpaceIdentifier
                                operand ";"
                              | "releasefbar" addressSpaceIdentifier operand ";"
stofss                       ::= "ftos"
                               | "stof"
seg                           ::= ( "segmentp" | stofss )
                                  addressSpaceIdentifier dataTypeId dataTypeId
                                  operand "," operand ";"
alloca                        ::= "alloca" addressSpaceIdentifier dataTypeId
                                  operand "," operand ";"
combine                       ::= "combine" vectorToken dataTypeId dataTypeId
                                  operand "," arrayOperandList ";"
expand                        ::= "expand" vectorToken dataTypeId dataTypeId
                                  arrayOperandList "," operand ";"
lda                           ::= "lda" optaddressSpace dataTypeId operand ","
                                  memoryOperand ";"
ldc                           ::= "ldc" dataTypeId operand "," (
                                  TOKEN_LABEL ";" | identifier ";" )
ret                           ::= "ret" ";"
cmp                           ::= "cmp" comparisonId optFTZ optPacking dataTypeId
                                  dataTypeId operand "," operand "," operand ";"
st                            ::= "st" ldModifier dataTypeId arrayOperand
                                  "," memoryOperand ";"
geometryId                    ::= "_1d"
                                | "_2d"
                                | "_3d"
                                | "_1db"
                                | "_1da"
                                | "_2da"
imageread  ::= "rdimage" "_v4" geometryId dataTypeId dataTypeId dataTypeId
               arrayOperandList "," operand "," operand "," arrayOperand ";"
imageload  ::= "ldimage" "_v4" geometryId dataTypeId dataTypeId dataTypeId
               arrayOperandList "," operand "," arrayOperand ";"
```

```
imagestore ::= "stimage" "_v4" geometryId dataTypeId dataTypeId dataTypeId
               arrayOperandList "," operand "," arrayOperand ";"
imageAtomicRet ::= "atomicimage"
               ( atomicOperationId
               geometryId dataTypeId dataTypeId dataTypeId
               operand "," operand "," arrayOperand "," operand ";"
             | "_cas"
               geometryId dataTypeId dataTypeId dataTypeId
               operand "," operand "," arrayOperand "," operand ","
               operand ";" )
imageAtomicNoRet ::= "atomicimagenoret"
                     ( atomicOperationId
                     geometryId dataTypeId dataTypeId dataTypeId
                     operand "," arrayOperand "," operand ";"
                   | "_cas"
                     geometryId dataTypeId dataTypeId dataTypeId
                     operand "," arrayOperand "," operand "," operand ";" )
queryOp          ::= "queryimageorder"
                   | "queryimageformat"
                   | "queryimagearray"
                   | "queryimagewidth"
                   | "queryimagedepth"
                   | "queryimageheight"
                   | "querysamplercoord"
                   | "querysamplerfilter"
query            ::= queryOp dataTypeId operand "," Operand ";
```

Appendix B

Limits

This appendix lists the maximum or minimum values that HSA implementations must support:

- Equivalence classes: Every implementation must support exactly 256 classes.
- Work-group size: Every implementation must support work-group sizes of 256 or larger. The work-group size is the product of the three work-group dimensions.
- Wavefront size: Every implementation must have a wavefront size that is a power of 2 in the range from 1 to 64 inclusive.
- Flattened ID (work-item flattened ID, work-item absolute flattened ID, and work-group flattened ID): Every implementation must support flattened IDs of $2^{32} - 1$.
- Number of work-groups: The only limit on the number of work-groups in a single kernel dispatch is a consequence of the size of the flattened IDs. Because each flattened ID is guaranteed to fit in 32 bits, the maximum number of work-groups in a single grid is limited to $2^{32} - 1$.
- Grid dimensions: Every implementation must support up to $2^{32} - 1$ sizes in each grid dimension. The product of the three is also limited to $2^{32} - 1$.
- Number of fbarriers: Every implementation must support at least 32 fbarriers per work-group.
- Size of group memory: Every implementation must support at least 32K bytes of group memory per compute unit for group segment variables. This amount might be reduced if an implementation uses group memory for the implementation of other HSAIL features such as fbarriers (see 9.3 Fine-Grain Barrier (fbar) Operations (p. 188)) and the exception detection operations (see 12.3 Additional Information on DETECT Exception Operations (p. 222)).

- Size of private memory: Every implementation must support at least 64k bytes of private memory per work-group.
- Image data type support. Every HSAIL implementation must support images as defined in 7.1 Images in HSAIL (p. 151). Support of images includes the following limits:
  - 3D images: Every implementation must support 3D image sizes up to (8192 x 8192 x 2048).
  - 2D images: Every implementation must support 2D image sizes up to (8192 x 8192).
  - Number of read images: Every implementation must support at least eight read-only images.
  - Number of read/write images: Every implementation must support at least one read/write image.
  - Number of samplers: Every implementation must support at least eight samplers.

Appendix C

Glossary of HSAIL Terms

acquire synchronizing operation
: A memory operation marked with acquire (an `ld_acq`, `atomic_ar`, or `atomic_noret_ar` operation).

active work-group
: A work-group executing in a compute unit.

active work-item
: A work-item in an active work-group. At an operation, an active work-item is one that executes the current operation.

agent
: A device that participates in the HSA memory model. Agents can generate and dispatch AQL packets to HSA components. See 1.1 What Is HSAIL? (p. 1).

AQL
: Architected Queuing Language. An AQL packet is an HSA-standard packet format. AQL dispatch packets are used to dispatch new kernels on the HSA component and specify the launch dimensions, instruction code, kernel arguments, completion detection, and more. Other AQL packets may also be supported in the future.

arg segment
: A memory segment used to pass arguments into and out of functions. See 2.8.1 Types of Segments (p. 14) and 10.4 Arg Segment (p. 207).

BRIG
: The HSAIL binary format. See Chapter 19 BRIG: HSAIL Binary Format (p. 257).

compound type
: A type made up of a base data type and a length. See 4.14.1 Base Data Types (p. 46).

compute unit
: A piece of virtual hardware capable of executing the HSAIL instruction set. The work-items of a work-group are executed on the same compute unit. An HSA component is composed of one or more compute units. See 2.1 Overview of Grids, Work-Groups, and Work-Items (p. 5).

dispatch
: A runtime operation that performs several chores, one of which is to launch a kernel. See 2.1 Overview of Grids, Work-Groups, and Work-Items (p. 5).

dispatch ID
: An identifier for a dispatch operation that is unique for the queue used for the dispatch. The combination of the dispatch ID and the queue ID is globally unique.

divergent control flow
: A situation in which kernels include branches and the execution of different work-items grouped into a wavefront might not be uniform. See 2.13 Divergent Control Flow (p. 21).

external linkage
: A condition in which a name of a function or variable in one compilation unit can refer to (is linked together with) an object with the same name defined in a different compilation unit. By default, all definitions of names starting with an ampersand (&) in a compilation unit are visible to other compilation units and have external linkage. See 4.23.1 External Linkage (p. 61).

fbarrier
: A fine-grain barrier that applies to a subset of a work-group. See 9.3 Fine-Grain Barrier (fbar) Operations (p. 188).

finalizer
: A back-end compiler that translates HSAIL code into native ISA for a compute unit.

finalizer extension
: An operation specific to a finalizer. Finalizer extensions are specified in the `extension` directive and accessed like all HSAIL operations. See 14.1 extension Directive (p. 231).

flattened absolute ID
: The result after a work-group absolute ID or work-item absolute ID is flattened into one dimension. See 2.3.4 Work-Item Flattened Absolute ID (p. 9).

global segment
: A memory segment in which memory is visible to all work-items in all HSA components and to all host CPUs. See 2.8.1 Types of Segments (p. 14).

grid
: A multidimensional, rectangular structure containing work-groups. A grid is formed when a program launches a kernel. See 1.2 HSAIL Virtual Language (p. 2).

group segment
: A memory segment in which memory is visible to a single work-group. See 2.8.1 Types of Segments (p. 14).

host CPU
: An agent that also supports the native CPU instruction set and runs the host operating system and the HSA runtime. As an agent, the host CPU can dispatch commands to an HSA component using memory operations to construct and enqueue AQL packets. In some systems, a host CPU can also act as an HSA component (with appropriate HSAIL finalizer and AQL mechanisms). See 1.1 What Is HSAIL? (p. 1).

HSA component
: An agent that supports the HSAIL instruction set and the AQL packet format. As an agent, an HSA component can dispatch commands to any HSA component (including itself) using memory operations to construct and enqueue AQL packets. An HSA component is composed of one or more compute units. See 1.1 What Is HSAIL? (p. 1).

HSA implementation
: A combination of (1) hardware components that execute one or more machine instruction set architectures (ISAs), (2) a compiler, linker, and loader, (3) a finalizer that translates HSAIL code into the appropriate native ISA if the hardware components cannot support HSAIL natively, and (4) a runtime system.

HSAIL
: Heterogeneous System Architecture Intermediate Language. A virtual machine and a language. The instruction set of the HSA virtual machine that preserves virtual machine abstractions and allows for inexpensive translation to machine code.

illegal operation
: An operation that a finalizer is allowed (but not required) to complain about.

image object
: An object type that describes how a read-only or read-write image is structured. See 7.1.4 Image Objects (p. 154).

invalid address
: Refer to the sections on shared virtual memory and error reporting in the *HSA System Architecture Specification* for more information on invalid addresses.

kernarg segment
: A memory segment used to pass arguments into a kernel. See 2.8.1 Types of Segments (p. 14).

kernel
: A section of code executed in a data-parallel way by a compute unit. Kernels are written in HSAIL and then separately translated by a finalizer to the target instruction set. See 1.1 What Is HSAIL? (p. 1).

lane
: An element of a wavefront. The wavefront size is the number of lanes in a wavefront. Thus, a wavefront with a wavefront size of 64 has 64 lanes. See 2.6 Wavefronts, Lanes, and Wavefront Sizes (p. 11).

natural alignment
: Alignment in which a memory operation of size n bytes has an address that is an integer multiple of n. For example, naturally aligned 8-byte stores can only be to addresses 0, 8, 16, 24, 32, 40, and so forth. See 4.22 Declaring and Defining Identifiers (p. 56).

private segment
: A memory segment in which memory is visible only to a single work-item. Used for read-write memory. See 2.8.1 Types of Segments (p. 14).

queue ID
: An identifier for a queue in a process. Each queue ID is unique in the process. The combination of the queue ID and the dispatch ID is globally unique.

read atomicity
: A condition of a load such that it must be read in its entirety. A load has read atomicity of size *b* if it cannot be read as fragments that are smaller than *b* bits.

readonly segment
: A memory segment for read-only memory. See 2.8.1 Types of Segments (p. 14).

release synchronizing operation
: A memory operation marked with release (an `st_rel`, `atomic_ar`, or `atomic_noret_ar` operation).

sampler object
: An object type that describes how a particular read on an image is to be performed. See 7.1.7 Sampler Objects (p. 160).

segment
: A contiguous addressable block of memory. Segments have size, addressability, access speed, access rights, and level of sharing between work-items. Also called memory segment. See 2.8 Segments (p. 13).

serial order
: A sequential execution of operations such that all effects of each operation appear to complete before the effects of the next operation. The HSAIL program sequence uses serial order to order memory accesses (loads, stores, and atomics).

spill segment
: A memory segment used to load or store register spills. See 2.8.1 Types of Segments (p. 14).

static linkage
: A condition in which a definition or declaration in one compilation unit is marked `static` and is not visible to any other compilation unit. See 4.23.2 Static Linkage (p. 62).

uniform operation
: An operation that produces the same result over a set of work-items. The set of work-items could be the grid, the work-group, the slice of work-items specified by the width modifier, or the wavefront. See 2.14 Uniform Operations (p. 23).

wavefront
: A group of work-items executing on a single instruction pointer. See 2.6 Wavefronts, Lanes, and Wavefront Sizes (p. 11).

wavefront size
: An implementation-defined constant, ranging from 1 to 64, specifying the size of a wavefront. See 2.6.2 Wavefront Size (p. 12).

work-group
: A collection of work-items. See 2.2 Work-Groups (p. 7).

work-group ID
: The identifier of a work-group expressed in three dimensions. See 2.2.1 Work-Group ID (p. 7).

work-group flattened ID
: The work-group ID flattened into one dimension. See 2.2.2 Work-Group Flattened ID (p. 8).

work-item
: The simplest element of work. See 2.3 Work-Items (p. 8).

work-item absolute ID
: The identifier of a work-item (within the grid) expressed in three dimensions. See 2.3.3 Work-Item Absolute ID (p. 9).

work-item flattened ID
: The work-item ID flattened into one dimension. See 2.3.2 Work-Item Flattened ID (p. 9).

work-item flattened absolute ID
: The work-item absolute ID flattened into one dimension. See 2.3.4 Work-Item Flattened Absolute ID (p. 9).

work-item ID
: The identifier of a work-item (within the work-group) expressed in three dimensions. See 2.3.1 Work-Item ID (p. 8).

write atomicity
: A condition of a store such that it must be written in its entirety. A store has write atomicity of size b if it cannot be broken up into fragments that are smaller than b bits.

Index