



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

Publication #: 49828 • Rev: Version 1.0 Provisional • Issue Date: 5 June 2014



© 2014 HSA Foundation. All rights reserved.

The contents of this document are provided in connection with the HSA Foundation specifications. This specification is protected by copyright laws and contains material proprietary to the HSA Foundation. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of HSA Foundation. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

HSA Foundation grants express permission to any current Founder, Promoter, Supporter Contributor, Academic or Associate member of HSA Foundation to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the HSA Foundation web-site should be included whenever possible with specification distributions.

HSA Foundation makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. HSA Foundation makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the HSA Foundation, or any of its Founders, Promoters, Supporters, Academic, Contributors, and Associates members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

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
- 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
- Vinod Tipparaju AMD
- Tony Tye AMD (Spec. Editor)
- Micah Villmow AMD
- Jem Davies ARM
- Ian Devereux ARM

- Robert Elliott ARM
- Alexander Galazin ARM
- Rune Holm ARM
- Kurt Shuler Arteris
- Andrew Richards Codeplay
- John Glossner General Processor Technologies
- 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
- Richard Bagley MediaTek Inc.
- Roy Ju MediaTek Inc.
- Trent Lo MediaTek Inc.
- Chien-Ping Lu MediaTek Inc. (Workgroup Chair)
- 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.

Document Conventions

Convention	Description
Boldface	In syntax tables, indicates a required item.
<i>Italics</i>	In text, indicates the name of a document or a new term that is described in the Appendix B Glossary of HSAIL Terms (p. 395) . In syntax tables, indicates a variable representation of a modifier or operand.
Monospace text	Indicates actual syntax.
<i>n</i>	Indicates the generic use of a number.

HSA Information Sources

- *HSA Platform System Architecture Specification* describes the HSA system architecture.
- *HSA Core API Programmers Reference Manual* describes the HSA runtime.
- The OpenCL™ Specification: <http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>

Contents

Chapter 1 Overview	1
1.1 What Is HSAIL?	1
1.2 HSAIL Virtual Language	2
Chapter 2 HSAIL Programming Model	5
2.1 Overview of Grids, Work-Groups, and Work-Items	5
2.2 Work-Groups	7
2.2.1 Work-Group ID	7
2.2.2 Work-Group Flattened ID	8
2.3 Work-Items	8
2.3.1 Work-Item ID	8
2.3.2 Work-Item Flattened ID	9
2.3.3 Work-Item Absolute ID	9
2.3.4 Work-Item Flattened Absolute ID	9
2.4 Scalable Data-Parallel Computing	10
2.5 Active Work-Groups and Active Work-Items	10
2.6 Wavefronts, Lanes, and Wavefront Sizes	11
2.6.1 Example of Contents of a Wavefront	11
2.6.2 Wavefront Size	12
2.7 Types of Memory	13
2.8 Segments	13
2.8.1 Types of Segments	15
2.8.2 Shared Virtual Memory	18
2.8.3 Addressing for Segments	19
2.8.4 Memory Segment Access Rules	20
2.8.5 Memory Segment Isolation	23
2.9 Small and Large Machine Models	24
2.10 Base and Full Profiles	25
2.11 Race Conditions	25
2.12 Divergent Control Flow	26
2.12.1 Uniform Operations	27
2.12.2 Using the Width Modifier with Control Transfer Operations	29
2.12.3 Post-Dominator and Immediate Post-Dominator	30
Chapter 3 Examples of HSAIL Programs	31

3.1 Vector Add Translated to HSAIL	31
3.2 Transpose Translated to HSAIL	32
Chapter 4 HSAIL Syntax and Semantics	35
4.1 Two Formats	35
4.2 Program	35
4.2.1 Agent Id	36
4.2.2 Call Convention Id	36
4.2.3 Finalization and Code Descriptors	36
4.3 Module	38
4.3.1 Annotations	40
4.3.2 Kernel	41
4.3.3 Function	43
4.3.4 Signature	45
4.3.5 Code Block	46
4.3.6 Arg Block	47
4.3.7 Operation	48
4.3.8 Variable	49
4.3.9 Fbarrier	52
4.3.10 Declaration and Definition Qualifiers	53
4.4 Source Text Format	59
4.5 Strings	61
4.6 Identifiers	62
4.6.1 Syntax	62
4.6.2 Scope	63
4.7 Registers	64
4.8 Constants	66
4.8.1 Integer Constants	67
4.8.2 Floating-Point Constants	69
4.8.3 Packed Constants	72
4.8.4 How Text Format Constants Are Converted to Bit String Constants	72
4.9 Labels	74
4.10 Initializers and Array Declarations	74
4.11 Storage Duration	76
4.12 Linkage	77
4.12.1 Program Linkage	77
4.12.2 Module Linkage	78
4.12.3 Function Linkage	78

4.12.4 Arg Linkage	79
4.12.5 None Linkage	79
4.13 Data Types	79
4.13.1 Base Data Types	79
4.13.2 Packed Data Types	80
4.13.3 Opaque Data Types	81
4.14 Packing Controls for Packed Data	81
4.14.1 Ranges	82
4.14.2 Packed Constants	83
4.14.3 Examples	85
4.15 Subword Sizes	86
4.16 Operands	86
4.16.1 Operand Compound Type	86
4.16.2 Rules for Operand Registers	87
4.17 Vector Operands	88
4.18 Address Expressions	88
4.19 Floating Point	90
4.19.1 Floating-Point Numbers	91
4.19.2 Rounding	92
4.19.3 Flush to Zero (ftz)	92
4.19.4 Not A Number (NaN)	93
4.19.5 Floating Point Exceptions	94
4.20 Dynamic Group Memory Allocation	95
4.21 Kernarg Segment	96

Chapter 5 Arithmetic Operations 99

5.1 Overview of Arithmetic Operations.....	99
5.2 Integer Arithmetic Operations.....	99
5.2.1 Syntax.....	99
5.2.2 Description.....	101
5.3 Integer Optimization Operation.....	105
5.3.1 Syntax.....	105
5.3.2 Description.....	106
5.4 24-Bit Integer Optimization Operations.....	106
5.4.1 Syntax.....	106
5.4.2 Description.....	107
5.5 Integer Shift Operations.....	108
5.5.1 Syntax.....	108

5.5.2 Description for Standard Form.....	109
5.5.3 Description for Packed Form.....	109
5.6 Individual Bit Operations.....	110
5.6.1 Syntax.....	110
5.6.2 Description.....	111
5.7 Bit String Operations.....	112
5.7.1 Syntax.....	112
5.7.2 Description.....	113
5.8 Copy (Move) Operations.....	117
5.8.1 Syntax.....	117
5.8.2 Description.....	118
5.8.3 Additional Information About lda.....	119
5.9 Packed Data Operations.....	119
5.9.1 Syntax.....	120
5.9.2 Description.....	121
5.9.3 Controls in src2 for shuffle Operation.....	123
5.9.4 Common Uses for shuffle Operation.....	124
5.9.5 Examples of unpacklo and unpackhi Operations.....	126
5.10 Bit Conditional Move (cmov) Operation.....	127
5.10.1 Syntax.....	127
5.10.2 Description.....	128
5.11 Floating-Point Arithmetic Operations.....	129
5.11.1 Syntax.....	129
5.11.2 Description.....	130
5.12 Floating-Point Bit Operations.....	133
5.12.1 Syntax.....	133
5.12.2 Description.....	135
5.13 Native Floating-Point Operations.....	136
5.13.1 Syntax.....	136
5.13.2 Description.....	137
5.14 Multimedia Operations.....	138
5.14.1 Syntax.....	138
5.14.2 Description.....	139
5.15 Segment Checking (segmentp) Operation.....	142
5.15.1 Syntax.....	142
5.15.2 Description.....	143
5.16 Segment Conversion Operations.....	143
5.16.1 Syntax.....	143

5.16.2 Description.....	144
5.17 Compare (cmp) Operation.....	145
5.17.1 Syntax.....	146
5.17.2 Description.....	147
5.18 Conversion (cvt) Operation.....	150
5.18.1 Overview.....	150
5.18.2 Syntax.....	151
5.18.3 Rules for Rounding for Conversions.....	152
5.18.4 Description of Integer Rounding Modes.....	153
5.18.5 Description of Floating-Point Rounding Modes.....	155
Chapter 6 Memory Operations	157
6.1 Memory and Addressing.....	157
6.1.1 How Addresses Are Formed.....	157
6.1.2 Memory Hierarchy.....	158
6.1.3 Alignment.....	159
6.1.4 Equivalence Classes.....	160
6.2 Memory Model.....	160
6.2.1 Memory Order.....	162
6.2.2 Memory Scope.....	165
6.2.3 Memory Synchronization Segments.....	167
6.2.4 Non-Memory Synchronization Segments.....	167
6.2.5 Agent Allocation.....	168
6.2.6 Kernel Dispatch Memory Synchronization.....	168
6.2.7 Execution Barrier.....	170
6.2.8 Flat Addresses.....	171
6.3 Load (ld) Operation.....	171
6.3.1 Syntax.....	171
6.3.2 Description.....	172
6.3.3 Additional Information.....	174
6.4 Store (st) Operation.....	175
6.4.1 Syntax.....	176
6.4.2 Description.....	177
6.4.3 Additional Information.....	177
6.5 Atomic Memory Operations.....	179
6.6 Atomic (atomic) Operations.....	180
6.6.1 Syntax.....	180
6.6.2 Description of Atomic and Atomic No Return Operations.....	181

6.7 Atomic No Return (atomicnoret) Operations	185
6.7.1 Syntax	185
6.7.2 Description	186
6.8 Notification (signal) Operations	187
6.8.1 Syntax	189
6.8.2 Description of Signal Operations	190
6.9 Memory Fence (memfence) Operation	193
6.9.1 Syntax	194
6.9.2 Description	194
Chapter 7 Image Operations	197
7.1 Images in HSAIL	197
7.1.1 Why Use Images?	197
7.1.2 Image Overview	198
7.1.3 Image Geometry	199
7.1.4 Image Format	201
7.1.4.1 Channel Order	201
7.1.4.2 Channel Type	203
7.1.4.3 Bits Per Pixel (bpp)	206
7.1.5 Image Access Permission	207
7.1.6 Image Coordinate	208
7.1.6.1 Coordinate Normalization Mode	209
7.1.6.2 Addressing Mode	210
7.1.6.3 Filter Mode	211
7.1.7 Image Creation and Image Handles	214
7.1.8 Sampler Creation and Sampler Handles	217
7.1.9 Using Image Operations	218
7.1.10 Image Memory Model	220
7.2 Read Image (rdimage) Operation	222
7.2.1 Syntax	222
7.2.2 Description	223
7.3 Load Image (ldimage) Operation	224
7.3.1 Syntax	224
7.3.2 Description	225
7.4 Store Image (stimage) Operation	226
7.4.1 Syntax	226
7.4.2 Description	227
7.5 Query Image and Query Sampler Operations	228

7.5.1 Syntax.....	228
7.5.2 Description.....	229

Chapter 8 Branch Operations 231

8.1 Syntax.....	231
8.2 Description.....	232

Chapter 9 Parallel Synchronization and Communication Operations 235

9.1 Barrier Operations.....	235
9.1.1 Syntax.....	235
9.1.2 Description.....	235
9.2 Fine-Grain Barrier (fbar) Operations.....	236
9.2.1 Overview: What Is an Fbarrier?.....	236
9.2.2 Syntax.....	237
9.2.3 Description.....	238
9.2.4 Additional Information About Fbarrier Operations.....	241
9.2.5 Pseudocode Examples.....	242
9.3 Execution Barrier.....	246
9.4 Cross-Lane Operations.....	248
9.4.1 Syntax.....	249
9.4.2 Description.....	249

Chapter 10 Function Operations 253

10.1 Functions in HSAIL.....	253
10.1.1 Example of a Simple Function.....	253
10.1.2 Example of a More Complex Function.....	254
10.1.3 Functions That Do Not Return a Result.....	254
10.2 Function Call Argument Passing.....	254
10.3 Function Declarations, Function Definitions, and Function Signatures.....	257
10.3.1 Function Declaration.....	257
10.3.2 Function Definition.....	258
10.3.3 Function Signature.....	258
10.4 Variadic Functions.....	259
10.5 align Qualifier.....	259
10.6 Direct Call (call) Operation.....	260
10.6.1 Syntax.....	260
10.6.2 Description.....	261
10.7 Switch Call (scall) Operations.....	261

10.7.1 Syntax.....	262
10.7.2 Description.....	262
10.8 Indirect Call (icall, ldi) Operations.....	263
10.8.1 Syntax.....	264
10.8.2 Description.....	265
10.9 Return (ret) Operation.....	267
10.9.1 Syntax.....	267
10.9.2 Description.....	268
10.10 Allocate Memory (alloca) Operation.....	268
10.10.1 Syntax.....	268
10.10.2 Description.....	269
Chapter 11 Special Operations	271
11.1 Dispatch Packet Operations.....	271
11.1.1 Syntax.....	271
11.1.2 Description.....	272
11.2 Exception Operations.....	274
11.2.1 Syntax.....	274
11.2.2 Description.....	275
11.2.3 Additional Information.....	275
11.3 User Mode Queue Operations.....	277
11.3.1 Syntax.....	277
11.3.2 Description.....	278
11.4 Miscellaneous Operations.....	280
11.4.1 Syntax.....	280
11.4.2 Description.....	281
Chapter 12 Exceptions	285
12.1 Kinds of Exceptions.....	285
12.2 Hardware Exceptions.....	285
12.3 Hardware Exception Policies.....	287
12.4 Debugger Exceptions.....	289
12.5 Handling Signaled Exceptions.....	289
12.5.1 HSA Runtime Debugger Interface Not Active.....	289
12.5.2 HSA Runtime Debugger Interface Active.....	289
Chapter 13 Directives	291
13.1 extension Directive.....	291

13.1.1 extension CORE.....	291
13.1.2 extension IMAGE.....	292
13.1.3 How to Set Up Finalizer Extensions.....	292
13.2 loc Directive.....	293
13.3 pragma Directive.....	294
13.4 Control Directives for Low-Level Performance Tuning.....	295
Chapter 14 version Statement	303
14.1 Syntax of the version Statement.....	303
Chapter 15 Libraries	305
15.1 Library Restrictions.....	305
15.2 Library Example.....	305
Chapter 16 Profiles	307
16.1 What Are Profiles?.....	307
16.2 Profile-Specific Requirements.....	308
16.2.1 Base Profile Requirements.....	309
16.2.2 Full Profile Requirements.....	310
Chapter 17 Guidelines for Compiler Writers	311
17.1 Register Pressure.....	311
17.2 Using Lower-Precision Faster Operations.....	311
17.3 Functions.....	311
17.4 Frequent Rounding Mode Changes.....	312
17.5 Wavefront Size.....	312
17.6 Control Flow Optimization.....	312
17.7 Memory Access.....	313
17.8 Unaligned Access.....	314
17.9 Constant Access.....	314
17.10 Segment Address Conversion.....	314
17.11 When to Use Flat Addressing.....	315
17.12 Arg Arguments.....	315
17.13 Exceptions.....	315
Chapter 18 BRIG: HSAIL Binary Format	317
18.1 What Is BRIG?.....	317
18.2 BrigModule.....	318

18.2.1 Format of Entries in the BRIG Sections.....	318
18.3 Support Types.....	319
18.3.1 Section Offsets.....	319
18.3.2 BrigAlignment.....	320
18.3.3 BrigAllocation.....	320
18.3.4 BrigAluModifierMask.....	320
18.3.5 BrigAtomicOperation.....	321
18.3.6 BrigBase.....	321
18.3.7 BrigCompareOperation.....	322
18.3.8 BrigControlDirective.....	322
18.3.9 BrigExecutableModifierMask.....	323
18.3.10 BrigImageChannelOrder.....	323
18.3.11 BrigImageChannelType.....	324
18.3.12 BrigImageGeometry.....	324
18.3.13 BrigImageQuery.....	324
18.3.14 BrigKind.....	325
18.3.15 BrigLinkage.....	326
18.3.16 BrigMachineModel.....	326
18.3.17 BrigMemoryModifierMask.....	326
18.3.18 BrigMemoryOrder.....	326
18.3.19 BrigMemoryScope.....	326
18.3.20 BrigOpcode.....	327
18.3.21 BrigPack.....	330
18.3.22 BrigProfile.....	330
18.3.23 BrigRegister.....	330
18.3.24 BrigRound.....	330
18.3.25 BrigSamplerAddressing.....	331
18.3.26 BrigSamplerCoordNormalization.....	331
18.3.27 BrigSamplerFilter.....	332
18.3.28 BrigSamplerQuery.....	332
18.3.29 BrigSectionIndex.....	332
18.3.30 BrigSectionHeader.....	332
18.3.31 BrigSegCvtModifierMask.....	333
18.3.32 BrigSegment.....	333
18.3.33 BrigType.....	333
18.3.34 BrigUint64.....	336
18.3.35 BrigVariableModifierMask.....	336
18.3.36 BrigVersion.....	336

18.3.37 BrigWidth.....	337
18.4 hsa_data Section.....	338
18.5 hsa_code Section.....	339
18.5.1 Directive Entries.....	339
18.5.1.1 Declarations and Definitions in the Same Module.....	340
18.5.1.2 BrigDirectiveArgBlock.....	340
18.5.1.3 BrigDirectiveComment.....	341
18.5.1.4 BrigDirectiveControl.....	341
18.5.1.5 BrigDirectiveExecutable.....	341
18.5.1.6 BrigDirectiveExtension.....	343
18.5.1.7 BrigDirectiveFbarrier.....	343
18.5.1.8 BrigDirectiveLabel.....	344
18.5.1.9 BrigDirectiveLoc.....	344
18.5.1.10 BrigDirectiveNone.....	345
18.5.1.11 BrigDirectivePragma.....	345
18.5.1.12 BrigDirectiveVariable.....	346
18.5.1.13 BrigDirectiveVersion.....	348
18.5.2 Instruction Entries.....	349
18.5.2.1 BrigInstBase.....	350
18.5.2.2 BrigInstAddr.....	350
18.5.2.3 BrigInstAtomic.....	351
18.5.2.4 BrigInstBasic.....	352
18.5.2.5 BrigInstBr.....	352
18.5.2.6 BrigInstCmp.....	353
18.5.2.7 BrigInstCvt.....	353
18.5.2.8 BrigInstImage.....	353
18.5.2.9 BrigInstLane.....	354
18.5.2.10 BrigInstMem.....	354
18.5.2.11 BrigInstMemFence.....	355
18.5.2.12 BrigInstMod.....	356
18.5.2.13 BrigInstQueryImage.....	357
18.5.2.14 BrigInstQuerySampler.....	357
18.5.2.15 BrigInstQueue.....	357
18.5.2.16 BrigInstSeg.....	358
18.5.2.17 BrigInstSegCvt.....	358
18.5.2.18 BrigInstSignal.....	359
18.5.2.19 BrigInstSourceType.....	359
18.6 hsa_operand Section.....	359

18.6.1	BrigOperandAddress.....	360
18.6.2	BrigOperandCodeList.....	361
18.6.3	BrigOperandCodeRef.....	361
18.6.4	BrigOperandData.....	362
18.6.5	BrigOperandImageProperties.....	362
18.6.6	BrigOperandOperandList.....	363
18.6.7	BrigOperandReg.....	364
18.6.8	BrigOperandSamplerProperties.....	364
18.6.9	BrigOperandString.....	365
18.6.10	BrigOperandWavesize.....	365
18.7	BRIG Syntax for Operations.....	366
18.7.1	BRIG Syntax for Arithmetic Operations.....	366
18.7.1.1	BRIG Syntax for Integer Arithmetic Operations.....	366
18.7.1.2	BRIG Syntax for Integer Optimization Operation.....	366
18.7.1.3	BRIG Syntax for 24-Bit Integer Optimization Operations.....	367
18.7.1.4	BRIG Syntax for Integer Shift Operations.....	367
18.7.1.5	BRIG Syntax for Individual Bit Operations.....	367
18.7.1.6	BRIG Syntax for Bit String Operations.....	368
18.7.1.7	BRIG Syntax for Copy (Move) Operations.....	368
18.7.1.8	BRIG Syntax for Packed Data Operations.....	369
18.7.1.9	BRIG Syntax for Bit Conditional Move (cmov) Operation.....	369
18.7.1.10	BRIG Syntax for Floating-Point Arithmetic Operations.....	369
18.7.1.11	BRIG Syntax for Floating-Point Bit Operations.....	370
18.7.1.12	BRIG Syntax for Native Floating-Point Operations.....	371
18.7.1.13	BRIG Syntax for Multimedia Operations.....	371
18.7.1.14	BRIG Syntax for Segment Checking (segmentp) Operation.....	371
18.7.1.15	BRIG Syntax for Segment Conversion Operations.....	372
18.7.1.16	BRIG Syntax for Compare (cmp) Operation.....	372
18.7.1.17	BRIG Syntax for Conversion (cvt) Operation.....	372
18.7.2	BRIG Syntax for Memory Operations.....	372
18.7.3	BRIG Syntax for Image Operations.....	374
18.7.4	BRIG Syntax for Branch Operations.....	374
18.7.5	BRIG Syntax for Parallel Synchronization and Communication Operations.....	375
18.7.6	BRIG Syntax for Function Operations.....	376
18.7.7	BRIG Syntax for Special Operations.....	376
18.7.7.1	BRIG Syntax for Dispatch Packet Operations.....	376
18.7.7.2	BRIG Syntax for Exception Operations.....	377
18.7.7.3	BRIG Syntax for User Mode Queue Operations.....	377

18.7.7.4 BRIG Syntax for Miscellaneous Operations.....	378
--	-----

Chapter 19 HSAIL Grammar in Extended Backus-Naur Form 379

19.1 HSAIL Lexical Grammar in Extended Backus-Naur Form (EBNF).....	379
19.2 HSAIL Syntax Grammar in Extended Backus-Naur Form (EBNF).....	381

Appendix A Limits 393

Appendix B Glossary of HSAIL Terms 395

Index 401

Figures

Chapter 2 HSAIL Programming Model

Figure 4–1 A Grid and Its Work-Groups and Work-Items	5
Figure 4–2 TOKEN_WAVESIZE Syntax Diagram	12

Chapter 4 HSAIL Syntax and Semantics

Figure 6–1 module Syntax Diagram	39
Figure 6–2 version Syntax Diagram	39
Figure 6–3 profile Syntax Diagram	39
Figure 6–4 machineModel Syntax Diagram	39
Figure 6–5 moduleDirective Syntax Diagram	40
Figure 6–6 moduleStatement Syntax Diagram	40
Figure 6–7 annotations Syntax Diagram	40
Figure 6–8 annotation Syntax Diagram	41
Figure 6–9 TOKEN_COMMENT Syntax Diagram	41
Figure 6–10 kernel Syntax Diagram	42
Figure 6–11 kernelHeader Syntax Diagram	42
Figure 6–12 kernFormalArgumentList Syntax Diagram	43
Figure 6–13 kernFormalArgument Syntax Diagram	43
Figure 6–14 function Syntax Diagram	44
Figure 6–15 functionHeader Syntax Diagram	44
Figure 6–16 funcOutputFormalArgumentList Syntax Diagram	44
Figure 6–17 funcInputFormalArgumentList Syntax Diagram	45
Figure 6–18 funcFormalArgumentList Syntax Diagram	45
Figure 6–19 funcFormalArgument Syntax Diagram	45
Figure 6–20 signature Syntax Diagram	45
Figure 6–21 sigOutputFormalArgumentList	46
Figure 6–22 sigInputFormalArgumentList Syntax Diagram	46
Figure 6–23 sigFormalArgumentList Syntax Diagram	46
Figure 6–24 sigFormalArgument Syntax Diagram	46
Figure 6–25 codeBlock Syntax Diagram	46
Figure 6–26 codeBlockDirective Syntax Diagram	47
Figure 6–27 codeBlockDefinition	47
Figure 6–28 codeBlockStatement Syntax Diagram	47
Figure 6–29 argBlock Syntax Diagram	48
Figure 6–30 argBlockDefinition	48
Figure 6–31 argBlockStatement Syntax Diagram	48

Figure 6–32 moduleVariable Syntax Diagram	51
Figure 6–33 codeBlockVariable Syntax Diagram	51
Figure 6–34 argBlockVariable Syntax Diagram	51
Figure 6–35 variable Syntax Diagram	51
Figure 6–36 variableSegment Syntax Diagram	52
Figure 6–37 moduleFbarrier Syntax Diagram	53
Figure 6–38 codeBlockFbarrier Syntax Diagram	53
Figure 6–39 fbarrier Syntax Diagram	53
Figure 6–40 optDeclQual Syntax Diagram	54
Figure 6–41 declQual Syntax Diagram	54
Figure 6–42 linkageQual Syntax Diagram	54
Figure 6–43 optAlignQual Syntax Diagram	54
Figure 6–44 optAllocQual Syntax Diagram	54
Figure 6–45 optConstQual Syntax Diagram	55
Figure 6–46 TOKEN_STRING Syntax Diagram	61
Figure 6–47 TOKEN_GLOBAL_IDENTIFIER Syntax Diagram	62
Figure 6–48 TOKEN_LOCAL_IDENTIFIER Syntax Diagram	62
Figure 6–49 TOKEN_LABEL_IDENTIFIER Syntax Diagram	62
Figure 6–50 identifier Syntax Diagram	62
Figure 6–51 TOKEN_CREGISTER Syntax Diagram	64
Figure 6–52 TOKEN_SREGISTER Syntax Diagram	64
Figure 6–53 TOKEN_DREGISTER Syntax Diagram	65
Figure 6–54 TOKEN_QREGISTER Syntax Diagram	65
Figure 6–55 registerNumber Syntax Diagram	65
Figure 6–56 TOKEN_INTEGER_CONSTANT Syntax Diagram	67
Figure 6–57 decimalIntegerConstant Syntax Diagram	67
Figure 6–58 hexIntegerConstant Syntax Diagram	67
Figure 6–59 octalIntegerConstant Syntax Diagram	68
Figure 6–60 TOKEN_HALF_CONSTANT Syntax Diagram	69
Figure 6–61 TOKEN_SINGLE_CONSTANT Syntax Diagram	69
Figure 6–62 TOKEN_DOUBLE_CONSTANT Syntax Diagram	69
Figure 6–63 decimalFloatConstant Syntax Diagram	70
Figure 6–64 hexFloatConstant Syntax Diagram	70
Figure 6–65 ieeeHalfConstant Syntax Diagram	70
Figure 6–66 ieeeSingleConstant Syntax Diagram	70
Figure 6–67 ieeeDoubleConstant Syntax Diagram	70
Figure 6–68 packedConstant Syntax Diagram	83
Figure 6–69 integerList Syntax Diagram	83

Figure 6–70 integerConstant Syntax Diagram	83
Figure 6–71 halfList Syntax Diagram	83
Figure 6–72 halfConstant Syntax Diagram	84
Figure 6–73 singleList Syntax Diagram	84
Figure 6–74 singleConstant Syntax Diagram	84
Figure 6–75 doubleList Syntax Diagram	84
Figure 6–76 doubleConstant Syntax Diagram	84

Chapter 5 Arithmetic Operations

Figure 7–1 Example of Broadcast	125
Figure 7–2 Example of Rotate	126
Figure 7–3 Example of Unpack	126

Chapter 6 Memory Operations

Figure 8–1 Memory Hierarchy	159
-----------------------------------	-----

Chapter 13 Directives

Figure 15–1 pragma Syntax Diagram	294
---	-----

Tables

Chapter 2 HSAIL Programming Model

Table 4–1 Wavefronts 0 Through 6	11
Table 4–2 Memory Segment Access Rules	21
Table 4–3 Machine Model Data Sizes	25

Chapter 4 HSAIL Syntax and Semantics

Table 6–1 Text Constants and Results of the Conversion	73
Table 6–2 Base Data Types	79
Table 6–3 Packed Data Types and Possible Lengths	80
Table 6–4 Opaque Data Types	81
Table 6–5 Packing Controls for Operations With One Source Input	82
Table 6–6 Packing Controls for Operations With Two Source Inputs	82

Chapter 5 Arithmetic Operations

Table 7–1 Syntax for Integer Arithmetic Operations	99
Table 7–2 Syntax for Packed Versions of Integer Arithmetic Operations	100
Table 7–3 Syntax for Integer Optimization Operation	105
Table 7–4 Syntax for 24-Bit Integer Optimization Operations	106
Table 7–5 Syntax for Integer Shift Operations	108
Table 7–6 Syntax for Individual Bit Operations	110
Table 7–7 Inputs and Results for <code>popcount</code> Operation	112
Table 7–8 Syntax for Bit String Operations	112
Table 7–9 Inputs and Results for <code>firstbit</code> and <code>lastbit</code> Operations	116
Table 7–10 Syntax for Copy (Move) Operations	117
Table 7–11 Syntax for Shuffle and Interleave Operations	120
Table 7–12 Syntax for Pack and Unpack Operations	120
Table 7–13 Bit Selectors for <code>shuffle</code> Operation	123
Table 7–14 Syntax for Bit Conditional Move (<code>cmove</code>) Operation	127
Table 7–15 Syntax for Floating-Point Arithmetic Operations	129
Table 7–16 Syntax for Packed Versions of Floating-Point Arithmetic Operations	130
Table 7–17 Syntax for Floating-Point Bit Operations	133
Table 7–18 Class Operation Source Operand Condition Bits	134
Table 7–19 Syntax for Packed Versions of Floating-Point Bit Operations	134
Table 7–20 Syntax for Native Floating-Point Operations	136
Table 7–21 Syntax for Multimedia Operations	138
Table 7–22 Syntax for Segment Checking (<code>segmentp</code>) Operation	142

Table 7–23 Syntax for Segment Conversion Operations	143
Table 7–24 Syntax for Compare (cmp) Operation	146
Table 7–25 Syntax for Packed Version of Compare (cmp) Operation	146
Table 7–26 Floating-Point Comparisons	148
Table 7–27 Conversion Methods	150
Table 7–28 Notation for Conversion Methods	151
Table 7–29 Syntax for Conversion (cvt) Operation	151
Table 7–30 Rules for Rounding for Conversions	152
Table 7–31 Integer Rounding Modes	154

Chapter 6 Memory Operations

Table 8–1 Syntax for Load (ld) Operation	171
Table 8–2 Syntax for Store (st) Operation	176
Table 8–3 Syntax for Atomic Operations	180
Table 8–4 Syntax for Atomic No Return Operations	185
Table 8–5 Syntax for Signal Operations	189
Table 8–6 Syntax for memfence Operation	194

Chapter 7 Image Operations

Table 9–1 Image Geometry Properties	199
Table 9–2 Channel Order Properties	201
Table 9–3 Channel Type Properties	203
Table 9–4 Channel Order, Channel Type and Image Geometry Combinations	207
Table 9–5 Image Handle Properties	215
Table 9–6 Image Operation Combinations	219
Table 9–7 Syntax for Read Image Operation	222
Table 9–8 Syntax for Load Image Operation	224
Table 9–9 Syntax for Store Image Operation	226
Table 9–10 Syntax for Query Image and Query Sampler Operations	228
Table 9–11 Query Image Properties	229
Table 9–12 Query Sampler Properties	229

Chapter 8 Branch Operations

Table 10–1 Syntax for Branch Operations	231
---	-----

Chapter 9 Parallel Synchronization and Communication Operations

Table 11–1 Syntax for Barrier Operations	235
Table 11–2 Syntax for fbar Operations	237
Table 11–3 Syntax for Cross-Lane Operations	249

Chapter 10 Function Operations

Table 12–1 Syntax for direct call Operation	260
Table 12–2 Syntax for switch call Operation	262
Table 12–3 Syntax for indirect call Operations	264
Table 12–4 Syntax for ret Operation	267
Table 12–5 Syntax for Allocate Memory (alloca) Operation	268

Chapter 11 Special Operations

Table 13–1 Syntax for Dispatch Packet Operations	271
Table 13–2 Syntax for Exception Operations	274
Table 13–3 Syntax for Exception Operations	277
Table 13–4 Syntax for Miscellaneous Operations	281

Chapter 13 Directives

Table 15–1 Control Directives for Low-Level Performance Tuning	295
--	-----

Chapter 18 BRIG: HSAIL Binary Format

Table 20–1 Formats of Directives in the hsa_code Section	340
Table 20–2 Formats of Instructions in the hsa_code Section	349
Table 20–3 Formats of Operands in the hsa_operand Section	360
Table 20–4 BRIG Syntax for Integer Arithmetic Operations	366
Table 20–5 BRIG Syntax for Integer Optimization Operation	366
Table 20–6 BRIG Syntax for 24-Bit Integer Optimization Operations	367
Table 20–7 BRIG Syntax for Integer Optimization Operation	367
Table 20–8 BRIG Syntax for Individual Bit Operations	367
Table 20–9 BRIG Syntax for Bit String Operations	368
Table 20–10 BRIG Syntax for Copy (Move) Operations	368
Table 20–11 BRIG Syntax for Packed Data Operations	369
Table 20–12 BRIG Syntax for Bit Conditional Move (cmov) Operation	369
Table 20–13 BRIG Syntax for Floating-Point Arithmetic Operations	369
Table 20–14 BRIG Syntax for Floating-Point Classify (class) Operation	370
Table 20–15 BRIG Syntax for Native Floating-Point Operations	371
Table 20–16 BRIG Syntax for Multimedia Operations	371
Table 20–17 BRIG Syntax for Segment Checking (segmentp) Operation	371
Table 20–18 BRIG Syntax for Segment Conversion Operations	372
Table 20–19 BRIG Syntax for Compare (cmp) Operation	372
Table 20–20 BRIG Syntax for Conversion (cvt) Operation	372
Table 20–21 BRIG Syntax for Memory Operations	372
Table 20–22 BRIG Syntax for Image Operations	374

Table 20–23 BRIG Syntax for Branch Operations	374
Table 20–24 BRIG Syntax for Parallel Synchronization and Communication Operations	375
Table 20–25 BRIG Syntax for Operations Related to Functions	376
Table 20–26 BRIG Syntax for Dispatch Packet Operations	376
Table 20–27 BRIG Syntax for Exception Operations	377
Table 20–28 BRIG Syntax for User Mode Queue Operations	377
Table 20–29 BRIG Syntax for Miscellaneous Operations	378

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 programming 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.
- Other agents that can participate in the HSA memory model. These include dedicated hardware to perform specialized tasks such as video encoding and decoding.

An HSA component does not need to execute HSAIL code directly: it can execute ISA generated from HSAIL code by a finalizer provided by the runtime. 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 (global addresses are 64 bits)
- Small mode (global addresses are 32 bits)

For more information, see [2.9 Small and Large Machine Models \(p. 24\)](#).

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 16 Profiles \(p. 307\)](#). 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 \(p. 31\)](#).

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

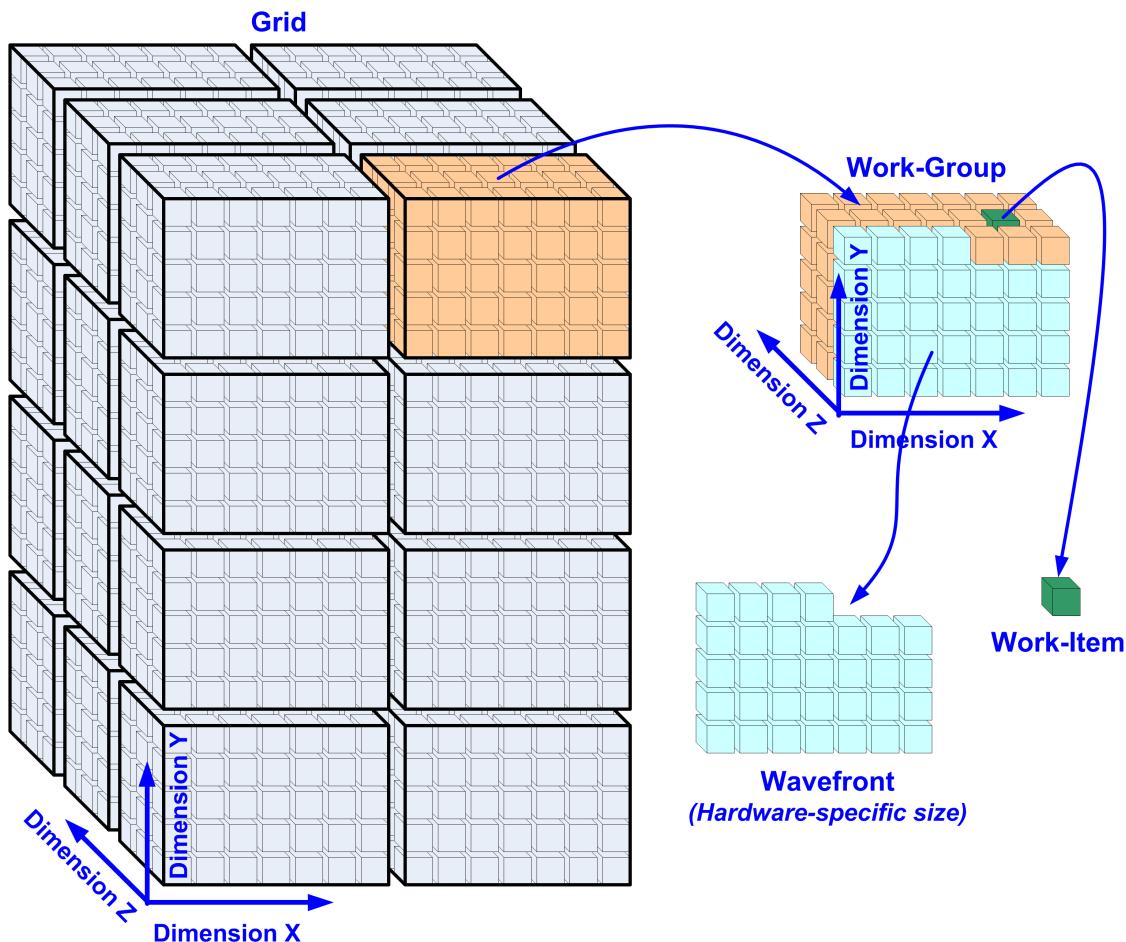
HSAIL Programming Model

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 4–1 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.

An HSA implementation ensures that all queues are serviced and dispatches the kernel ISA associated with the queued dispatch packets on the HSA component with which the queue is associated, causing 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 `requiredworkgroupsizes` control directive.

The combination of the dispatch ID and the queue ID can be used to identify a kernel dispatch within the application. Operations in a kernel can access these IDs by means of the `packetid` and `queueid` special operations. See [11.1 Dispatch Packet Operations \(p. 271\)](#) and [11.3 User Mode Queue Operations \(p. 277\)](#).

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, then 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; all other grid and work-group size components must be non-0.

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 be either 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:

```
work-group size = workgroupsize0 * workgroupsize1 * workgroupsize2
```

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 dispatch packet operations. See [11.1 Dispatch Packet Operations \(p. 271\)](#).

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 = workgroupid0 +
                         workgroupid1 * workgroupsize0 +
                         workgroupid2 * workgroupsize0 * workgroupsize1
```

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 11 Special Operations \(p. 271\)](#).)

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_i is the size of the work-group or 1.

For each dimension i , the set of values of ID_i is the dense set $[0, 1, 2, \dots, \max_i - 1]$.

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:

```
work-item flattened ID = ID0 + ID1 * max0 + ID2 * max0 * max1
```

where:

```
ID0 = workitemid (dimension 0)
ID1 = workitemid (dimension 1)
ID2 = workitemid (dimension 2)
max0 = workgroupsize (dimension 0)
max1 = 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:

```
work-item flattened absolute ID = ID0 + ID1 * max0 + ID2 * max0 * max1
```

where:

```
ID0 = workitemabsid (dimension 0)
ID1 = workitemabsid (dimension 1)
ID2 = workitemabsid (dimension 2)
max0 = gridsize (dimension 0)
max1 = 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 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 4–1 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	54-63

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	55-63

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

Figure 4-2 TOKEN_WAVESIZE Syntax Diagram

TOKEN_WAVESIZE

→ [WAVESIZE] →

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`. This can be used in any operation operand where an integer or bit immediate value is allowed, and as the argument to the width modifier. It is not supported for directive operands unless indicated otherwise. See [2.12 Divergent Control Flow \(p. 26\)](#).

`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 \(p. 13\)](#).

- Registers

There are four register sizes:

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

Registers are untyped.

For more information, see [4.7 Registers \(p. 64\)](#).

- 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 \(p. 197\)](#).

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 both work-items executed by HSA components and threads executed by other agents.

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.

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.

No isolation guarantee between segments is provided. See [2.8.5 Memory Segment Isolation \(p. 23\)](#).

2.8.1 Types of Segments

There are seven types of segments:

- Global

The *global segment* can be used to hold variables that are shared by all agents.

Global segment variables can either have program or agent allocation. See the `alloc` qualifier description in section [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

- Global memory variables with program allocation has a single allocation for the variable which is visible to all agents, including all HSA components executing an application. The address of the variable allocation in global memory can be read and written by any agent, including any work-item of any kernel dispatch executed by any HSA component.
- Global memory variables with agent allocation have multiple allocations, one for each HSA component that is a member of the HAIL program. Each allocation has a distinct global segment address and is only visible to the associated HSA component. The address of each variable allocation in global memory can only be read and written by work-items of any kernel dispatch executed by the associated HSA component. In addition, the host CPU agent can access all allocations by using the HSA runtime and specifying the HSA component.

The visibility of global memory is further constrained by the memory model (see [6.2 Memory Model \(p. 160\)](#)). See [4.10 Initializers and Array Declarations \(p. 74\)](#) for a description of the visibility of variable initializers.

All global memory is persistent across the application execution.

Global memory can be set before the execution of a kernel dispatch, either explicitly by HSAIL variable definition initializers, by the HSA runtime variable definition API, by the execution of other kernel dispatches, by the application executing on a host CPU agent, or by other agents.

Global segment variables can be marked `const` in which case their value must not be changed for their storage duration after they have been allocated and initialized. A `const` variable HSAIL definition must have an initializer. A non-`const` HSAIL variable definition can optionally have an initializer. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

Standard page protections (for example, read-only, read-write, and protected) apply to global memory. See the *HSA Platform System Architecture Specification* for information.

Global memory can be accessed using a flat address that is not in the range reserved for the group or private memory.

- Group

The *group segment* is used to hold variables that are shared by the work-items of a work-group.

Group memory is visible to the work-items of a single work-group of a kernel dispatch. An address of a variable in group memory can be read and written by any work-item in the work-group with which it is associated, but not by work-items in other work-groups or by other agents. Visibility of group memory is further constrained by the memory model. See [6.2 Memory Model \(p. 160\)](#).

Group memory is persistent across the execution of the work-items in the work-group of the kernel dispatch with which it is associated.

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

One specific implementation-defined range of flat addresses is reserved for group memory. See [2.8.3 Addressing for Segments \(p. 19\)](#).

- Private

The *private segment* can be used to hold variables that are local to a single work-item.

Private memory is visible only to a single work-item of a kernel dispatch. An address of a variable in private memory can be read and written only by the work-item with which it is associated, but not by any other work-items or other agents.

Private memory is persistent across the execution of the work-item with which it is associated.

Private memory is uninitialized when the work-item starts.

One specific implementation-defined range of flat addresses is reserved for private memory. See [2.8.3 Addressing for Segments \(p. 19\)](#).

- Kernarg

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

Kernarg memory is visible to all work-items of the kernel dispatch with which it is associated. An address of a variable in kernarg memory can be read by any work-item in the kernel dispatch with which it is associated, but not by work-items in other kernel dispatches. Other agents must not modify the kernarg memory while the kernel dispatch it is associated with is executing.

Kernarg memory is persistent across the execution of the kernel dispatch with which it is associated.

Kernarg memory is initialized to the values specified by the agent that dispatches the kernel.

Kernarg memory cannot be accessed using a flat address.

- **Readonly**

The *readonly segment* can be used to hold variables that remain constant during the execution of a kernel dispatch. However, the values can be changed from one kernel dispatch execution to another by the host CPU agent using the HSA runtime. Accesses to the readonly segment might perform better than accesses to global memory on some implementations.

HSA components are only permitted to perform read operations on the addresses of variables that reside in readonly memory.

All readonly memory is persistent across the application.

Readonly segment variables have agent allocation. Each variable has multiple allocations, one for each HSA component that is a member of the HAIL program, each allocation with a distinct address. Each HSA component can only access its associated allocation. The host CPU agent can access all allocations by using the HSA runtime and specifying the HSA component. See the `alloc` qualifier description in section [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

Readonly memory can be set and made visible before the execution of a kernel dispatch, either explicitly by HSAIL variable definition initializers, by the HSA runtime variable definition API, or by the application executing on a host CPU agent using the HSA runtime. However, it is undefined if a readonly variable allocation value for an HSA component is changed while a kernel dispatch that uses that variable is executing on that HSA component. See [4.10 Initializers and Array Declarations \(p. 74\)](#).

Readonly segment variables can be marked `const` in which case their value must not be changed for their storage duration after they have been allocated and initialized. A `const` variable HSAIL definition must have an initializer. A non-`const` HSAIL variable definition can optionally have an initializer. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

Readonly memory cannot be accessed using a flat address.

It is implementation-defined whether read-only memory protections are applied to the readonly segment variables while a kernel dispatch is executing.

- **Spill**

HSAIL has a fixed number of registers, and the *spill segment* can be used to load or store register spills. This also serves as a hint to the finalizer, which might be able to generate better code by promoting spills into available ISA registers.

Spill memory is visible only to a single work-item of a kernel dispatch. A spill segment variable can be read and written only by the work-item with which it is associated, but not by any other work-items or other agents.

Spill segment variables can only be defined in a kernel or function code block, not outside a kernel or function. The address of a spill segment variable cannot be taken with an `lda` operation. These restrictions make it easier for a finalizer to promote spill segment variables to ISA registers.

If temporary variables for a single work-item are required that do require their address to be taken, then they can be defined in the private segment. Such variables would not be easy for a finalizer to promote into ISA registers.

Spill memory is persistent across the execution of the work-item with which it is associated.

Spill memory is uninitialized when the work-item starts.

Spill memory cannot be accessed using a flat address.

- Arg

The *arg segment* is used to pass arguments into and out of functions.

Arg memory is visible only to a single work-item of a kernel dispatch while it executes an arg block and the corresponding function call. An arg segment variable defined in an arg block can be accessed only by the work-item with which it is associated, but not by any other work-items or other agents. In an arg block it can be written if it corresponds to a call input actual argument, and read if it corresponds to a call output actual argument; in the called function the input formal arguments can only be read and the called function output formal argument can only be written.

The address of an arg segment variable cannot be taken with an `lda` operation. This makes it easier for a finalizer to allocate arg segment variables to ISA registers.

Arg memory is persistent across the execution of an arg block and associated called function of a work-item of a kernel dispatch with which it is associated.

Arg memory is uninitialized when the work-item starts execution of an arg block.

Arg memory cannot be accessed using a flat address.

For more information, see [10.2 Function Call Argument Passing \(p. 254\)](#).

Also see:

- [4.6.2 Scope \(p. 63\)](#)
- [4.11 Storage Duration \(p. 76\)](#)
- [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#)

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 four sharing levels: shared over all work-items (global), shared over all work-items of a single dispatch (kernarg), 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.

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.

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

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 segment, 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 execution of work-items in 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 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 memory, step 1 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 \(p. 157\)](#).

See also:

- [5.15 Segment Checking \(segmentp\) Operation \(p. 142\)](#)
- [5.16 Segment Conversion Operations \(p. 143\)](#)

2.8.4 Memory Segment Access Rules

The persistence of a memory segment specifies how stores in the segment can be seen by other loads. See [Table 4-2 \(p. 21\)](#).

Table 4–2 Memory Segment Access Rules

Segment	HSA Component interaction (HSAIL)	Non-HSA Component Agent interaction	Persistence	Allocation	Definition can be initialized?	Where can variables be defined?	Can be accessed by a flat address?
Global	General global space; non-const variables read-write; const variables read-only and value must not change during storage duration of variable.	Read-write by all agents.	Application	Program or agent	Optional for non-const variables; required for const variables	module; kernel or function code block	Yes
Readonly	Read-only; value must not change during execution of kernel dispatch.	Can be written by host CPU agent using HSA runtime, provided no kernel dispatch is executing that is using variable.	Application	Agent	Optional	module; kernel or function code block	No
Kernarg	Holds kernel arguments; read-only; value must not change during execution of kernel dispatch.	Initial values provided by the agent when the kernel dispatch is queued. Initial values must not be changed while kernel dispatch is executing.	Kernel	Automatic	No	kernel formal argument list	No

Segment	HSA Component interaction (HSAIL)	Non-HSA Component Agent interaction	Persistence	Allocation	Definition can be initialized?	Where can variables be defined?	Can be accessed by a flat address?
Group	Read-write.	Inaccessible.	Work-group	Automatic	No	module; kernel or function code block	Yes
Arg	Holds function input and output arguments; actual input arguments can be written, actual output argument can be read, formal input arguments can be read and formal output argument can be written; cannot have address taken with <code>lda</code> operation.	Inaccessible.	Work-item	Automatic	No	kernel or function arg block; function formal arguments	No
Private	Holds work-item local variables; read-write.	Inaccessible.	Work-item	Automatic	No	module; kernel or function code block	Yes
Spill	Holds spilled register values; read-write; cannot have address taken with <code>lda</code> operation.	Inaccessible.	Work-item	Automatic	No	kernel or function code block	No

Each segment has one of the following persistence values:

- Application: If the allocation is program, then stores in one kernel dispatch or agent thread can be seen by loads of another kernel dispatch or agent thread in the same application execution. If the allocation is agent, then stores in one kernel dispatch execution, or performed by the host CPU agent using the HSA runtime and specifying the HSA component, can be seen by any kernel dispatch executing on the same HSA component in the same application execution. Note, the readonly segment variables for an HSA component cannot be changed while a kernel dispatch that access the variables is executing on that HSA component.
- Kernel: stores in one kernel dispatch execution can be seen by loads in the same kernel dispatch execution. Note, the kernarg segment values cannot be changed while kernel dispatch is executing.
- 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, and the function argument list arg variables, can only be accessed while the function is being executed by the work-item. Arg variables declared in an argument scope can only be accessed while the containing argument scope is being executed by the work-item. See [4.6.2 Scope \(p. 63\)](#) and [4.11 Storage Duration \(p. 76\)](#).

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 or other agent thread executed 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 segment kind. This allows a `segmentp` operation 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. See [5.15 Segment Checking \(`segmentp`\) Operation \(p. 142\)](#).

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.

For further information on:

- Allocation, see the `alloc` qualifier description in section [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).
- Initializers, see [4.10 Initializers and Array Declarations \(p. 74\)](#).

2.8.5 Memory Segment Isolation

An implementation is not required to isolate the memory for each segment. This means it may be possible to access the memory of one segment using addresses in another segment. This may permit work-items or other agents to use the aliased addresses to access variables in segments that are defined as being inaccessible.

However, while the kernel dispatch executes, results are undefined if a variable allocated in one segment is accessed in another segment:

- Results are undefined whether the variable is defined explicitly in HSAIL or is allocated dynamically by any agent including a host CPU.
- Results are undefined whether the variable is accessed using a segment address or a corresponding flat address.

- Results are undefined whether the access is done by another work-item in the same kernel dispatch, the work-items in other kernel dispatches, or by other agents, including a host CPU.
- An implementation is not required to detect or generate an exception if such an access occurs.

This allows an implementation considerable freedom in how it can implement segments:

- An implementation could use special dedicated hardware:
 - Readonly and/or kernarg variables could be allocated in a specialized read-only cache.
 - Special hardware could be used to accelerate arg and spill memory. For example, by promoting them to ISA registers.
 - Group addresses could be mapped to special scratch-pad memory allocated for each HSA component compute unit.
- An implementation could use addresses in global memory:
 - If used to implement group memory, the implementation must adjust the group segment and flat 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 address.
 - If used to implement private memory, the implementation must adjust the segment or flat addresses used by each work-item so that different work-items access different memory locations for the same private segment address. For example, this could be done:
 - By using separate contiguous memory areas for each work-item.
 - By expanding the segment or flat address into multiple interleaved addresses, one for every work-item in a wavefront. This could be implemented by special hardware.

2.9 Small and Large Machine Models

HSAIL supports two machine models. Machine models determine the size of certain data values and are not compatible. [Table 4-3 \(p. 25\)](#) shows the 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 [14.1 Syntax of the version Statement \(p. 303\)](#).

Table 4–3 Machine Model Data 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
Fbarrier address	32-bit	32-bit
Address expression offset	32-bit	64-bit
Atomic value	32-bit	32-bit & 64-bit
Signal value	32-bit	64-bit

The small machine model has these constraints:

- 64-bit atomic operations are not supported.
- 64-bit signal value operations are not supported.
- For register plus offset addressing, the offset is truncated to 32 bits.

The large machine model has these constraints:

- 32-bit signal value operations are not supported.

2.10 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 [14.1 Syntax of the version Statement \(p. 303\)](#).

For more information, see [Chapter 16 Profiles \(p. 307\)](#).

2.11 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. See [6.2.1 Memory Order \(p. 162\)](#).

2.12 Divergent Control Flow

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

When a *wavefront* executes a branch that can transfer to multiple targets (namely a conditional branch `cbr` or switch branch `sbr`, see [Chapter 8 Branch Operations \(p. 231\)](#)), or a function call that can invoke multiple functions (namely a switch call `scall` or indirect call `icall`, see [Chapter 10 Function Operations \(p. 253\)](#)), it is possible that the work-items in the *wavefront* take different paths. This causes the *wavefront* to enter divergent control flow.

For example, a single `cbr` operation will transfer control to the label for work-items where the source condition is true and to the operation after the `cbr` for work-items where the source condition is false. Similarly, a `sbr` or `scall` operation might transfer to different labels or functions respectively for work-items which have different values for the source index. Finally, an `icall` operation could transfer to different functions for work-items that have different values for the indirect function descriptor. In these cases, the *wavefront* is said to diverge, and the code is inside 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.

For example, in divergent control flow, an implementation may execute all the work-items that transfer to the same target up to a reconvergence point, with the other work-items waiting, followed by execution of the all the work-items that transfer to the next target, and so forth until all the possible targets are processed. Then execution can continue by all work-items from the reconvergence point.

In the case of a `cbr` there can only be up to two possible targets, but an `sbr`, `scall` and `icall` could potentially have many more. For example, a conditional 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_b1 $c0, @k1;
// code for the else statements
br @join;
@k1:
// code the then statements
@join:
```

The time to execute this would be the sum of the time it takes to execute the THEN 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 THEN block or the ELSE block but not both.

HSAIL requires that implementations reconverge control flow no later than the immediate post-dominator (as described in [2.12.3 Post-Dominator and Immediate Post-Dominator \(p. 30\)](#)), but may reconverge earlier. This requirement can limit certain optimizations that involve cloning control flow (see [17.6 Control Flow Optimization \(p. 312\)](#)).

Divergent control flow can also occur within control flow that is already divergent. In this case there are the same issues and requirements, except they only apply to the work-items that are active in the parent divergent path being executed.

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. See [6.2.1 Memory Order \(p. 162\)](#).

2.12.1 Uniform Operations

If the set of work-items that make up the dispatch grid can be partitioned into a set of slices, and if for each independent slice an operation behaves the same for each work-item in the slice each time it is evaluated for a particular evaluation property, then the operation is termed a *uniform operation* with respect to the slice and evaluation property. Note that the operation does not have to behave the same for the work-items of different slices, and does not have to behave the same each time the same operation is evaluated. The operation only has to behave the same for the work-items in a single slice for a single evaluation of the operation.

Certain HSAIL memory, image, control flow, function and parallel synchronization and communication operations allow the uniformity of the operation to be specified by an optional width modifier. These operations specify the evaluation property and default slice algorithm that will be used if the width modifier is omitted. In addition, some special operations are required to be uniform.

There are three kinds of uniform evaluation properties:

result uniform

Specifies that all active work-items in the slice will produce the same result value. Note that the operation may be in divergent code and only some of the work-items in the slice may be active. Only the active work-items are required to produce the same result value, the inactive work-items are not executing the operation and so do not use the result of the operation even if it is result uniform.

For example, a load operation is result uniform if all active work-items in the slice will load the same value, independent of each work-item's source operand address. This may allow a finalizer to generate more efficient code by executing the load once and broadcasting the result to all active work-items in the slice.

For another example, a conditional branch operation is result uniform if all work-items in the slice either take the branch or do not take the branch. Conceptually the result value of the conditional branch is the code address of the next instruction. This may allow a finalizer to deduce that operations in divergent code are execution uniform if the control flow is reducible and all conditional control flow in the control flow nest is result uniform. See also [2.12.2 Using the Width Modifier with Control Transfer Operations \(p. 29\)](#).

execution uniform

Specifies that all work-items in the slice will either be active or inactive. It will never be the case that some are active and some are inactive. Therefore, if the operation is executed, it will be executed by all work-items in the slice. Note, execution uniform does not specify that each work-item in the same slice will have the same values for the source operands, and produce the same values when the operation is executed.

For example, a cross-lane operation is execution uniform if all work-items in the slice will execute it. This may allow a finalizer to use special ISA instructions.

communication uniform

Specifies that all active work-items in the slice will only communicate with other active work-items in the slice. No communication will happen between work-items that are in different slices. Communication between work-items can be accomplished by using atomic memory operations (to both the global and group segments), signal operations, the clock operation, the DETECT exception special operations, and the execution synchronization operations (barrier and fbarrier).

For example, a `barrier_width(n)` indicates that only the work-items in a work-group's slice are participating in some form of communication. If an implementation has a waveform size that is greater than or equal to n , it is free to optimize the ISA generated for the barrier because the gang-scheduled execution of work-items in wavefronts will ensure execution synchronization of the communicating work-items.

The uniform slice algorithm can be specified by the width modifier:

`width(all)`

Each slice is comprised of all the work-items of a single work-group.

`width(n)`

The value of n must be a power of 2 between 1 and 2^{31} inclusive. Work-items are in the same slice if they are in the same work-group and if the integral part of the work-items' flattened ID (see [2.3.2 Work-Item Flattened ID \(p. 9\)](#)) divided by n are the same. Note that all slices will not be of size n if the size of all work-groups is not a multiple of n .

`width(WAVESIZE)`

Same as `width(n)` where *n* is set to the implementation-defined number of work-items in a wavefront (see [2.6 Wavefronts, Lanes, and Wavefront Sizes \(p. 11\)](#)).

Note that 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. See [2.6 Wavefronts, Lanes, and Wavefront Sizes \(p. 11\)](#).

If a slice is not a multiple of `WAVESIZE`, then the last wavefront of the slice is partial. Any lanes that do not correspond to work-items are treated as inactive for result uniform, and only the lanes corresponding to work-items are considered for execution uniform.

The default for the width modifier if it is omitted depends on the operation, and can either be `width(1)`, `width(WAVESIZE)`, or `width(all)`.

The width modifier is only a performance hint, and can be ignored by an implementation.

2.12.2 Using the Width Modifier with Control Transfer Operations

Sometimes a finalizer can generate more efficient code if it knows details about how divergent control flow 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 specify the result uniformity of the target of conditional and switch branches. All active work-items in the same slice are guaranteed to branch to the same target.

If the width modifier is omitted for control transfer operations, it defaults to `width(1)`, indicating each active work-item can transfer to a target independently.

If active work-items specified by the width modifier do not transfer to the same target, the behavior is undefined.

If a width modifier is used, then:

- If a conditional branch (`cbr`), then the value in `src` must be the same for all active work-items in the same slice as it is used to determine the target of the branch.
- If a switch branch (`sbr`) or switch call (`scall`), then the index value in `src` does not have to be the same for all active work-items in the same slice, but the label or function selected by those index values must be the same for all active work-items in the same slice. It is the target that must be uniform, not the index value.
- If an indirect call (`icall`), then the value in `src` must be the same for all active work-items in the same slice as it is used to determine the indirect function being called.

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 with a width modifier of `width(64)`.

`width(all)` indicates that all work-items in the work-group will transfer to the same target. If a developer knows, or a compiler can determine, 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 which specifies that either all active work-items will go to the target label 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 target. 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 HSA component can be obtained by a runtime query. Using `width(WAVESIZE)` may allow the finalizer to optimize.

2.12.3 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_b1 $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.

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;

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_b1            $c0, @BB0_2;
    br                @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];
    br                @BB0_1;
};

```

3.2 Transpose Translated to HSAIL

The code below shows one way to write a transpose.

```

version 1:0:$full:$small;

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:           // %entry
// BB#0:
    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.

4.1 Two Formats

HSAIL modules can be represented in either of two formats:

- Text format
- Binary format (BRIG)

This chapter describes the text format.

The chapters describing HSAIL operations show syntax for the text format.

For more information about BRIG, see [Chapter 18 BRIG: HSAIL Binary Format \(p. 317\)](#).

The HSA runtime finalizer operates on the BRIG format.

4.2 Program

An application can use the HSA runtime to create zero or more HSAIL programs, to which it can add zero or more HSAIL modules. The HSAIL module is the unit of HSAIL generation, and can contain multiple symbol declarations and definitions. Linking of symbol declarations to symbol definitions between modules is done within the context of the HSAIL program (see [4.12 Linkage \(p. 77\)](#)). In addition, the application can provide symbol definitions to an HSAIL program, and can obtain the address of symbols defined by the HSAIL program, using the HSA runtime.

An HSAIL module can be added to zero or more HSAIL programs. Distinct instances of the symbols it defines are created within each program, and symbol declarations are only linked to the definitions provided by other modules in the same program.

When an HSAIL program is created, one or more HSA components that are part of the HSA platform must be specified, together with the machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)) and profile (see [Chapter 16 Profiles \(p. 307\)](#)). The machine model address size for the global segment must match the size used by the application (see [Table 4-3 \(p. 25\)](#)). All modules added to the program must have the same machine model and profile as the program.

4.2.1 Agent Id

When an HSAIL program is created, one or more HSA components that are part of the HSA platform must be specified. The set of agents associated with a program cannot be changed after it has been created.

Within a program, each of these HSA component members has a unique identifier in the range 0 to the number of agent members minus one. The same HSA component may have a different agent identifier in different HSAIL programs that it is a member.

4.2.2 Call Convention Id

Each HSA component can support one or more call conventions. For example, an HSA component may have different call conventions that each use a different number of isa registers to allow different numbers of wavefronts to execute on a compute unit.

When the HSA runtime is used to create an HSAIL program (see [4.2 Program \(p. 35\)](#)) the set of agents that are members of the program must be specified. The HSA runtime determines a dense call convention id ordering for the program. The first agent is assigned call convention ids 0 to the number of call conventions it supports minus one. The next agent is assigned the next range of call conventions according to the number it supports, and so on. Note that the same agent may have a different range of call convention ids in different programs of which it is a member.

When finalizing a kernel or indirect function, the specific call convention required can be specified, or the finalizer can be requested to choose the best call convention based on the kernel. The call convention used by the code produced by the finalizer is recorded in the code descriptor produced by the finalizer.

An HSA runtime query is available to determine the range of call convention ids used for a particular agent of a particular program.

4.2.3 Finalization and Code Descriptors

The HSA runtime finalizer can be used to generate code for kernels and indirect functions from a specific program for a specific HSA component. A kernel can only be finalized once per program per agent. An indirect function can only be finalized once per program per agent per call convention. Only code for the HSA components specified when the program was created can be requested.

The program must contain a definition for the requested kernels and indirect functions amongst the modules that have been added to the program. The modules of the program must collectively define all variables, fbarriers, kernels and functions referenced by operations in the code block of:

- The kernel and indirect functions being finalized.
- The transitive closure of all functions specified by `call` or `scall` operations starting with the kernel and indirect functions being finalized. See [Chapter 10 Function Operations \(p. 253\)](#).

When invoking the finalizer, one or more kernels and indirect functions can be requested. On some implementations specifying multiple kernels and indirect functions can produce code with better performance than finalizing the kernels and indirect function individually. For example, it can allow the finalizer to generate

common code for shared functions which can reduce code footprint and improve instruction cache performance.

The HSA runtime allocates a kernel descriptor for every kernel definition in a program, and an indirect function descriptor for every indirect function definition in the program. Both kinds of descriptors are represented as an array of code descriptors in global segment memory. Each code descriptor provides the information about a finalization of the kernel or indirect function for a specific HSA component, and for indirect functions, a specific call convention of that HSA component.

The kernel descriptor array is indexed by the agent id. The kernel descriptor and agent id are available by the `ldk` and `agent_id` operations respectively, or by an HSA runtime query. See [4.2.1 Agent Id \(p. 36\)](#) and [11.3 User Mode Queue Operations \(p. 277\)](#).

The indirect function descriptor is indexed by the call convention id which is performed implicitly by the `icall` operation. The indirect function descriptor address is available by the `ldi` operation or an HSA runtime query. See [4.2.2 Call Convention Id \(p. 36\)](#) and [10.8 Indirect Call \(icall, ldi\) Operations \(p. 263\)](#)).

The finalizer updates the code descriptor corresponding to the agent or call convention for each kernel or indirect function that it finalizers.

The layout of a code descriptor is defined by the HSA runtime. It includes a kind field that indicates whether the code descriptor contains finalized code. If it has been finalized, then for kernels the information needed to create a User Mode Queue kernel dispatch packet is available, including:

- The byte size of the group segment for a single work-group. This includes:
 - Module scope and function scope group segment variables used by the kernel or any functions it calls directly or indirectly.
 - Any finalizer allocated temporary space. For example, in the implement of exception operations or fbarriers.

This does not include any dynamically allocated group segment space (see [4.20 Dynamic Group Memory Allocation \(p. 95\)](#)).

- The byte size of the private segment for a single work-item. This includes:
 - Module scope and function scope private segment variables.
 - Space for function scope spill segment variables allocated in memory.
 - Space for argument scope arg segment variables allocated in memory.
 - Any space needed for saved HSAIL or ISA registers due to calls.
 - Any other finalizer introduced temporaries including spilled ISA registers and space for function call stack.

These include those both objects used directly by the kernel as well as any functions it calls directly or indirectly.

This size may need to be increased appropriately for kernels that call indirect functions using signatures or have recursive function calls.

- The 64-bit opaque code handle to the finalized code that includes the executable ISA for the HSA component. It can be used for the kernel dispatch packet kernel object address field.

The code descriptor also includes other information that may be useful to a high-level language runtime to invoke and manage the kernel's execution. For example, the size

and alignment of the kernarg segment and the call convention used by the code of the kernel.

For an indirect function, a code descriptor includes:

- The 64-bit opaque code handle to the finalized code that includes the executable ISA for a single call convention of the HSA component.

The HSA runtime updates the kernel descriptor as kernel and indirect functions are finalized. The update is done as follows:

1. The code generated is made available for execution on the associated agent for kernel dispatches launched after the HSA runtime has completed finalization.
2. All the fields in the global segment code descriptor are set.
3. Finally, a store release at system scope of the code descriptor kind field to indicate that the kernel or indirect function code is available.

Once code has been generated for a kernel for a specific HSA component, it can be executed by adding a kernel dispatch packet to a User Mode Queue associated with the HSA component. The information required to create the kernel dispatch packet is available in the code descriptor addressed by using the agent id to index the kernel descriptor.

For an indirect function, the code is only made available to kernel dispatches launched after the indirect function has been finalized. Therefore, prior to executing a kernel, all indirect functions that it will call must have been finalized for the HSA component with the call convention used by the kernel code. The `icall` operation, used to call indirect functions, implicitly access the code descriptor addressed by using the call convention id of the executing kernel to index the indirect function descriptor. See [10.8 Indirect Call \(icall, ldi\) Operations \(p. 263\)](#) and [4.2.2 Call Convention Id \(p. 36\)](#).

Appropriate memory synchronization is needed to access the code descriptor since it is updated concurrently by the HSA runtime. Memory synchronization may include using an acquire fence at system scope on the kernel dispatch packet if the code descriptor may have changed since the HSA component executed the last packet acquire fence at system scope, or using a load acquire at system scope on the code descriptor kind field if the code descriptor may change during the execution of the kernel dispatch. This applies both to accesses performed explicitly to create kernel dispatch packets, and implicitly by the `icall` operation.

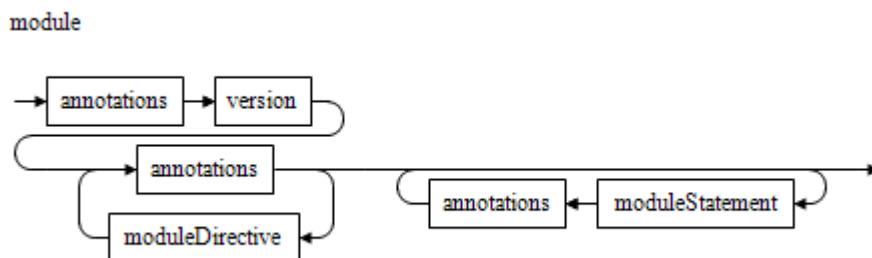
The code will remain available to execute until the HSA runtime is used to destroy the HSAIL program with which the code is associated. All HSAIL programs created by the application are implicitly destroyed when the application terminates.

4.3 Module

A module is the basic building block for HSAIL programs. When HSAIL is generated it is represented as a module.

A module begins with a version statement, is followed by zero or more module directives, and ends with zero or more module statements.

Figure 6–1 module Syntax Diagram



The `version` statement specifies the HSAIL language version and the required target architecture. For more information, see [Chapter 14 version Statement \(p. 303\)](#).

Figure 6–2 version Syntax Diagram

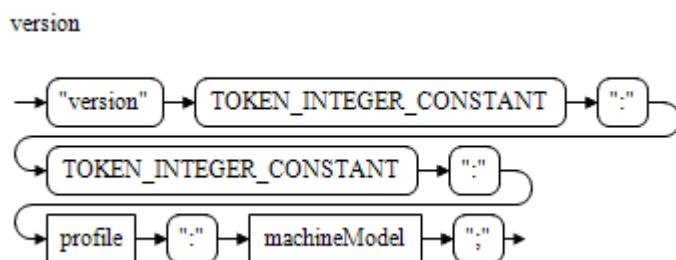


Figure 6–3 profile Syntax Diagram

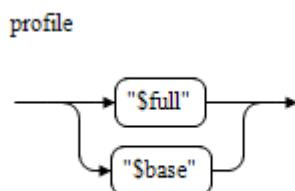
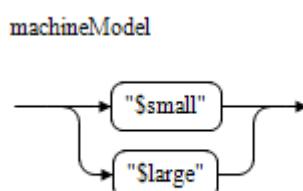


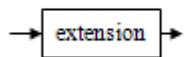
Figure 6–4 machineModel Syntax Diagram



A module directive can be the extension directive which must precede other HSAIL statements and applies to the whole module. See [Chapter 13 Directives \(p. 291\)](#).

Figure 6–5 moduleDirective Syntax Diagram

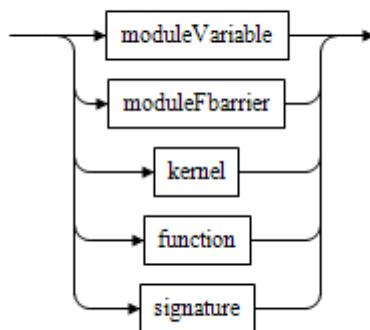
```
moduleDirective
```



A module statement can be a module variable, module fbarrier, kernel, function or signature.

Figure 6–6 moduleStatement Syntax Diagram

```
moduleStatement
```



4.3.1 Annotations

Comments, file and line number location information, and pragmas can be interleaved with other HSAIL statements.

Figure 6–7 annotations Syntax Diagram

```
annotations
```

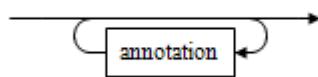
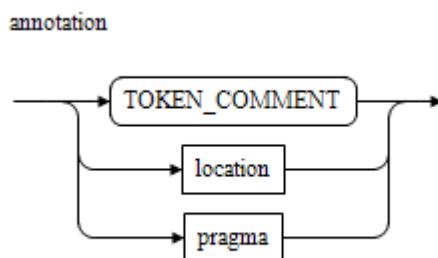


Figure 6–8 annotation Syntax Diagram



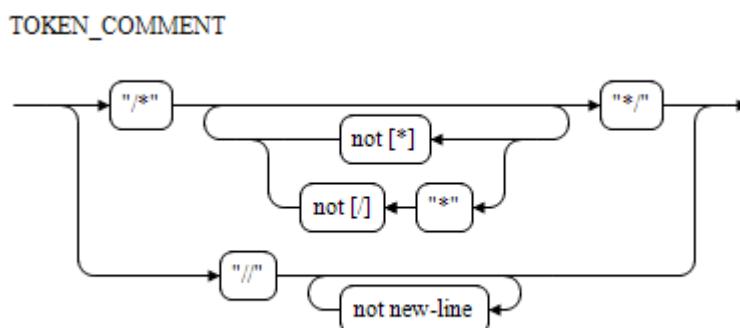
Comments that can span multiple lines use non-nested /* and */. The comment starts at the /* and extends to the next */, which might be on a different line.

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.

Figure 6–9 TOKEN_COMMENT Syntax Diagram



For more information on location and pragma directives, see [Chapter 13 Directives \(p. 291\)](#).

4.3.2 Kernel

A kernel can either be a declaration or a definition.

A kernel declaration establishes the name, formal arguments and linkage of a kernel.

A kernel definition establishes the same characteristics as a declaration, and in addition defines the kernel's code block. A definition causes memory for the kernel descriptor to be allocated, and initialized to indicate no finalized code is available for any HSA component that is a member of the program, when the module is added to a program. The memory is destroyed when the HSA runtime is used to destroy the program. All HSAIL programs created by the application are implicitly destroyed when the application terminates. See [4.2.3 Finalization and Code Descriptors \(p. 36\)](#).

A kernel with the same name can be declared in a module zero or more times, but can be defined at most once.

All kernels with the same name in a module denote the same kernel and must be compatible.

Kernel declaration and definitions are compatible if they:

- have the same kernel formal arguments,
- and have the same linkage.

If the kernel has program linkage, then there can be at most one definition of a kernel with program linkage with that name amongst all the modules in the same program. All kernels with program linkage in any module of the same program that have the same name denote the same kernel and must be compatible. This allows a kernel to be defined in one module, but used in another module of the same program. Otherwise, the kernel has module linkage and can only be referenced within the same module. See [4.12 Linkage \(p. 77\)](#).

At the time a kernel is finalized, or is referenced by an `ldk` operation of another kernel or indirect function being finalized, there must be a definition for it in one of the modules that belong to the program.

A single module can contain multiple kernel declarations and definitions.

A kernel declaration or definition consists of `decl` if a declaration, followed by its linkage, the `kernel` keyword, the kernel name, the kernel formal argument list, the code block if a definition, and terminated by a semicolon (`;`).

The arguments of a kernel declaration have no linkage as they are not referenced by any operations.

The arguments of a kernel definition have function linkage and can only be referenced within the function scope they are defined.

Figure 6–10 kernel Syntax Diagram

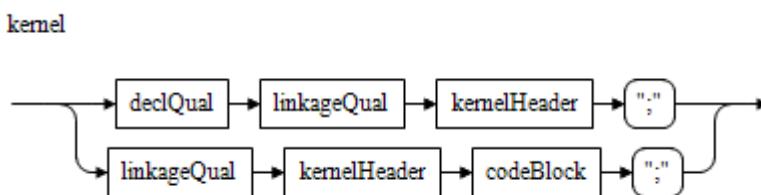


Figure 6–11 kernelHeader Syntax Diagram

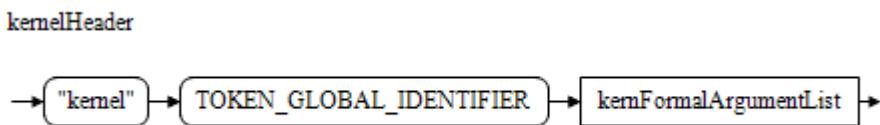


Figure 6–12 kernFormalArgumentList Syntax Diagram

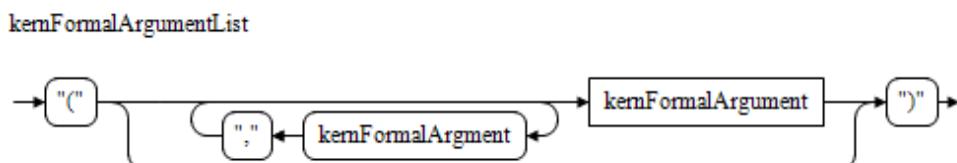


Figure 6–13 kernFormalArgument Syntax Diagram



4.3.3 Function

A function can either be a declaration or a definition.

A function declaration establishes the name, output formal arguments, input formal arguments, whether it is an indirect function and linkage of a function.

A function definition establishes the same characteristics as a declaration, and in addition defines the function's code block. A definition of an indirect function causes memory for the indirect function descriptor to be allocated, and initialized to indicate no finalized code is available for any call convention of the program, when the module is added to a program. The memory is destroyed when the HSA runtime is used to destroy the program. All HSAIL programs created by the application are implicitly destroyed when the application terminates. See [4.2.3 Finalization and Code Descriptors \(p. 36\)](#).

A function with the same name can be declared in a module zero or more times, but can be defined at most once.

All functions with the same name in a module denote the same function and must be compatible.

Function declaration and definitions are compatible if they:

- have the same function output and input formal arguments,
- match whether they are indirect or not,
- and have the same linkage.

If the function has program linkage, then there can be at most one definition of a function with program linkage with that name amongst all the modules in the same program. All functions with program linkage in any module of the same program that have the same name denote the same function and must be compatible. This allows a function to be defined in one module, but used in another module of the same program. Otherwise, the function has module linkage and can only be referenced within the same module. See [4.12 Linkage \(p. 77\)](#).

At the time an indirect function is finalized, or a function is referenced by a `call`, `scall` or, for an indirect function, an `ldi` operation of another kernel or indirect function being finalized (including any indirect references from functions they call by `call` and `scall` operations), there must be a definition for it in one of the modules that belong to the program. It is not required that an indirect function called by an `icall` operation has a definition since the actual indirect function will not be determined until the kernel is executed. An indirect function has limitations on the symbols it can reference. See [10.8 Indirect Call \(icall, ldi\) Operations \(p. 263\)](#)

A single module can contain multiple function declarations and definitions.

A function declaration or definition consists of `decl` if a declaration, followed by its linkage, an optional `indirect` keyword to specify an indirect function, the `function` keyword, the function name, the function output formal argument list, the function input formal argument list, the code block if a definition, and terminated by a semicolon (`;`).

The arguments of a function declaration have no linkage as they are not referenced by any operations.

The arguments of a function definition have function linkage and can only be referenced within the function scope they are defined.

Figure 6–14 function Syntax Diagram

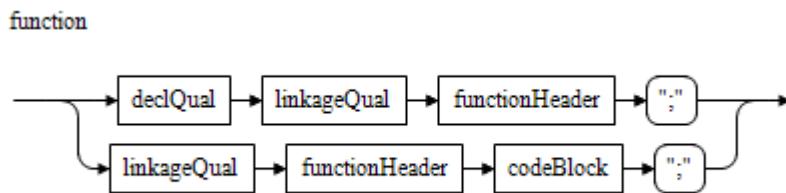


Figure 6–15 functionHeader Syntax Diagram

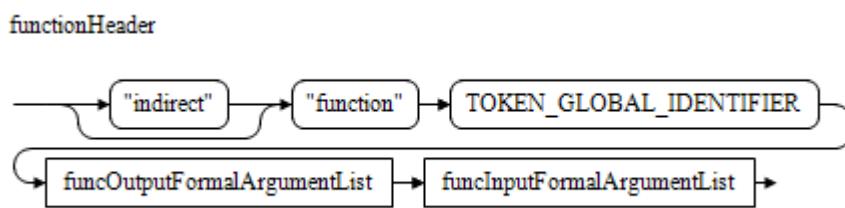


Figure 6–16 funcOutputFormalArgumentList Syntax Diagram

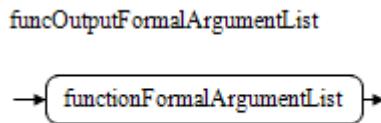


Figure 6–17 funcInputFormalArgumentList Syntax Diagram

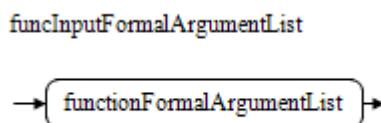


Figure 6–18 funcFormalArgumentList Syntax Diagram

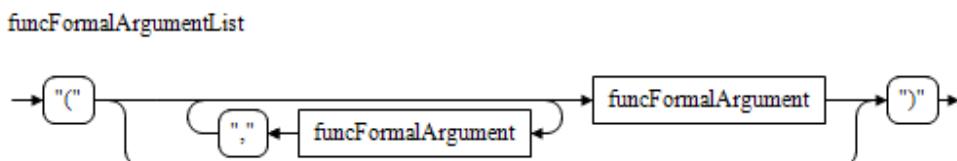


Figure 6–19 funcFormalArgument Syntax Diagram



For more information, see [Chapter 10 Function Operations \(p. 253\)](#).

4.3.4 Signature

A function signature does not describe a single function: it defines a type of function which describes a set of functions that have the same types of arguments. It therefore cannot be called directly, but instead is used to describe the target of an indirect function call `icall` operation.

Syntactically, a signature is much like a function.

The arguments of a signature have no linkage as they are not referenced by any operations.

Figure 6–20 signature Syntax Diagram

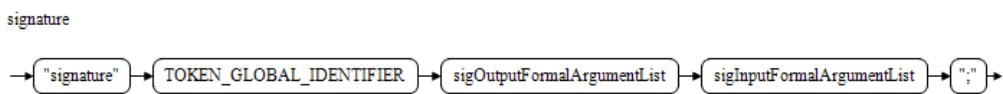


Figure 6–21 sigOutputFormalArgumentList

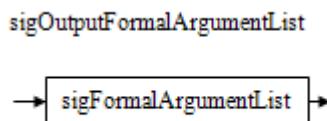


Figure 6–22 sigInputFormalArgumentList Syntax Diagram

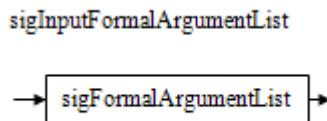


Figure 6–23 sigFormalArgumentList Syntax Diagram

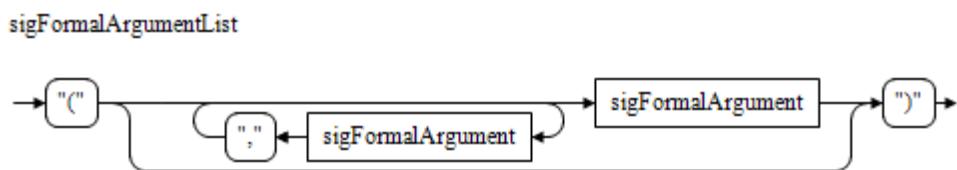


Figure 6–24 sigFormalArgument Syntax Diagram

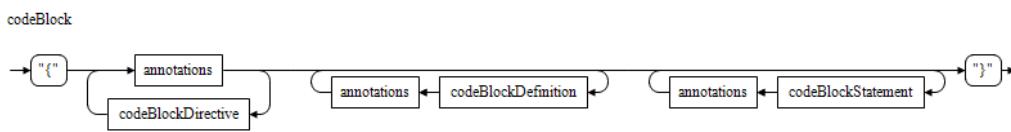


For more information, see [Chapter 10 Function Operations \(p. 253\)](#).

4.3.5 Code Block

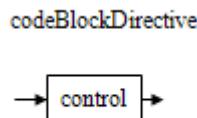
A code block consists of zero or more code block directives, followed by zero or more code block definitions, followed by zero or more code block statements, all surrounded by curly braces (`{ }`).

Figure 6–25 codeBlock Syntax Diagram



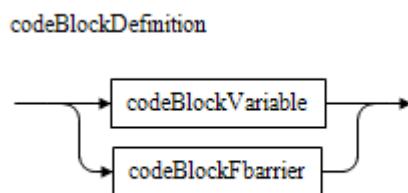
A code block directive can be a control directive which must proceed other HSAIL statements in the code block and applies to the kernel or function with which the code block is associated.

Figure 6–26 codeBlockDirective Syntax Diagram



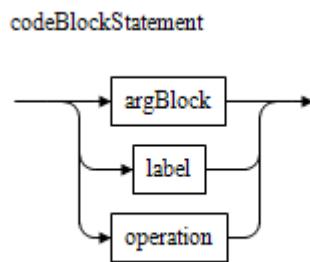
A code block definition can be a code block variable or code block fbarrier.

Figure 6–27 codeBlockDefinition



A code block statement can be an arg block, label or operation (except a call operation which is only allowed in an arg block). The code block statements contain the bulk of the code in an HSAIL module.

Figure 6–28 codeBlockStatement Syntax Diagram



For more information on:

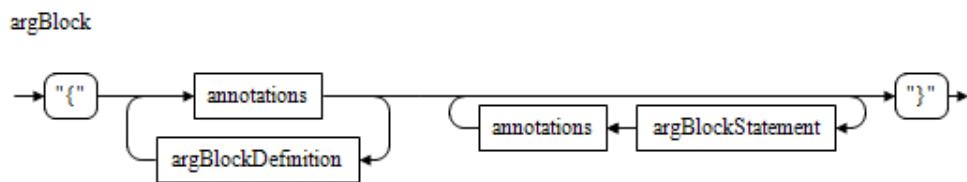
- Control directives, see [Chapter 13 Directives \(p. 291\)](#).
- Labels, see [4.9 Labels \(p. 74\)](#).

4.3.6 Arg Block

An arg block consists of zero or more arg block definitions, followed by zero or more arg block statements, all surrounded by curly braces ({}). An arg block is used to pass

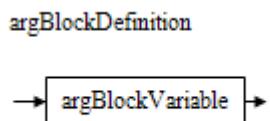
argument values into and out of a call to a function. See [10.2 Function Call Argument Passing \(p. 254\)](#).

Figure 6–29 argBlock Syntax Diagram



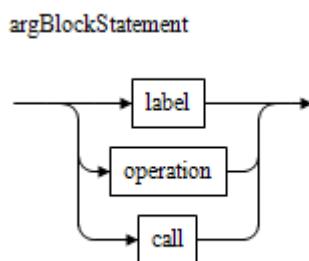
An arg block definition can be an arg block variable.

Figure 6–30 argBlockDefinition



An arg block statement can be a label or operation (including a call operation).

Figure 6–31 argBlockStatement Syntax Diagram



For more information, see [10.2 Function Call Argument Passing \(p. 254\)](#).

4.3.7 Operation

An operation is an executable HSAIL instruction.

The example below shows four operations:

```

global_f32 %array[256];
@start: workitemid_u32 $s1, 0;
    shl_u32      $s1, $s1, 2;           // multiply by 4
    ld_global_u32 $s2, [%array][$s1]; // reads array[4 * workid]
    add_f32      $s2, $s2, 0.5F;       // add 1/2
  
```

Operations consist of an opcode usually followed by an underscore followed by a type followed by a comma-separated list of zero or more operands and ending with a semicolon (;). Some operations use special syntax for certain operands.

Operands can be registers, constants, address expressions or the identifier of a label, kernel, function, signature or fbarrier. Some operations also support lists of operands surrounded by parenthesis (()) or square braces([]). The destination operand is first, followed by source operands. See [4.16 Operands \(p. 86\)](#).

HSAIL allows a finalizer implementation to implement extensions that add additional features to HSAIL which may include operations. A *finalizer extension* is enabled using the `extension` directive. Any operations enabled by a finalizer extension are accessed like all other HSAIL operations. For more information, see [13.1.3 How to Set Up Finalizer Extensions \(p. 292\)](#).

For more information, see:

- [Chapter 5 Arithmetic Operations \(p. 99\)](#)
- [Chapter 6 Memory Operations \(p. 157\)](#)
- [Chapter 7 Image Operations \(p. 197\)](#)
- [Chapter 8 Branch Operations \(p. 231\)](#)
- [Chapter 9 Parallel Synchronization and Communication Operations \(p. 235\)](#)
- [Chapter 10 Function Operations \(p. 253\)](#)
- [Chapter 11 Special Operations \(p. 271\)](#)

4.3.8 Variable

A module variable can either be a declaration or a definition. A code block or arg block variable can only be a definition.

A variable declaration establishes the name, segment, data type, array dimensions, linkage and variable qualifiers of a variable.

A variable definition establishes the same characteristics as a declaration, and in addition for some segments can specify an initializer. For global and readonly segment variables, a definition causes memory for the variable to be allocated, and initialized if it has an initializer, when the module is added to a program. The memory is destroyed when the HSA runtime is used to destroy the program. All HSAIL programs created by the application are implicitly destroyed when the application terminates.

A module variable with the same name can be declared in a module zero or more times, but can be defined at most once.

All module variables with the same name in a module denote the same variable and must be compatible.

Variable declaration and definitions are compatible if they:

- have the same segment,
- have the same data type,

- have the same linkage,
- have the same variable qualifiers.
- have matching array dimension declarations:
 - have no array dimension specified, or
 - have an array dimension specified with matching array dimension size:
 - A definition with an initializer that has an array dimension that is empty has an array dimension size equal to the number of elements in the array initializer.
 - A declaration with an array dimension that is empty matches a declaration or definition with an array dimension of any size.
 - Otherwise the array dimension sizes must be the same.

There can only be one code block or arg block variable with a specific name in the scope of its identifier. The same name is allowed as a code block or arg block variable in a different scope. For example, there can be multiple function scope variables with the same name if they are defined in different functions or kernels. See [4.6.2 Scope \(p. 63\)](#).

A code block variable has function linkage and can only be referenced within the function scope it is defined.

An arg block variable has arg linkage and can only be referenced within the arg block it is defined.

If the module variable has program linkage, then there can be at most one definition of a module variable with program linkage with that name amongst all the modules in the same program. All module variables with program linkage in any module of the same program that have the same name denote the same variable and must be compatible. This allows a module variable to be defined in one module, but used in another module. Otherwise, the module variable has module linkage and can only be referenced within the same module. See [4.12 Linkage \(p. 77\)](#).

At the time a kernel or indirect function is finalized, there must be a definition for all the variables referenced by address expressions of operations that are part of the kernel or indirect function (including any indirect references from operations in functions they call by `call` and `scall` operations) in one of the modules that belong to the program.

A single module can contain multiple variable declarations and definitions.

A module variable declaration or definition consists of `decl` if it is a declaration, followed by its linkage, the optional variable qualifiers, a segment, a data type, the variable name, an optional array dimension, an optional initializer if a definition for a segment that allows initializers, and terminated by a semicolon (`;`).

A code block or arg block variable can only be a definition and has function and arg linkage respectively. Therefore, it is defined the same as a module variable except `decl` and linkage are not specified.

Figure 6–32 moduleVariable Syntax Diagram

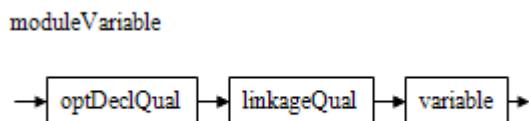


Figure 6–33 codeBlockVariable Syntax Diagram

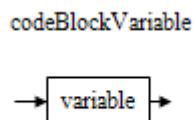


Figure 6–34 argBlockVariable Syntax Diagram

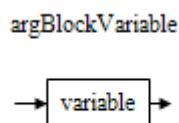
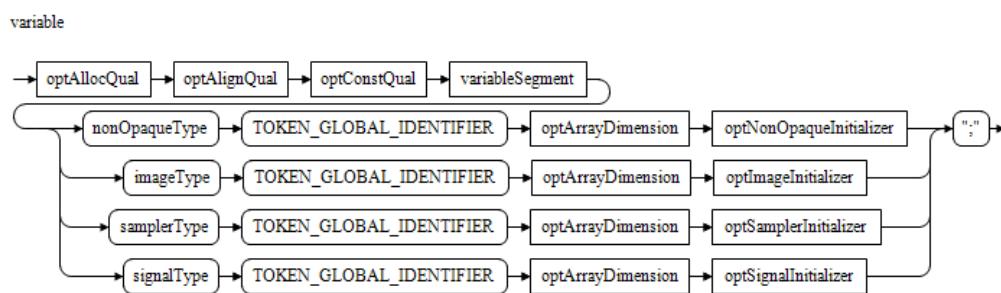


Figure 6–35 variable Syntax Diagram

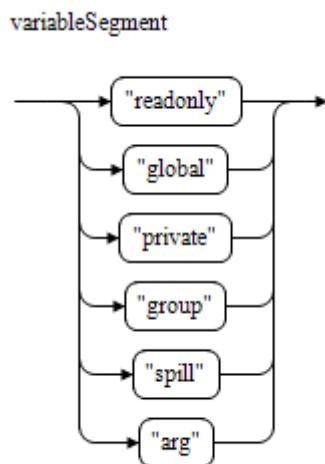


A variable segment can be one of the following:

- Readonly (only allowed for module and code block variables)
- Global (only allowed for module and code block variables)
- Group (only allowed for module and code block variables)
- Private (only allowed for module and code block variables)
- Spill (only allowed for code block variables)
- Arg (only allowed for arg block variables)

The syntax for kernarg and arg segment formal argument variables is defined in [4.3.2 Kernel \(p. 41\)](#) and [4.3.3 Function \(p. 43\)](#) respectively.

Figure 6–36 variableSegment Syntax Diagram



An initializer is only allowed for variable definitions for the following segments:

- Global
- Readonly

The data type can be one of the data types described in [4.13 Data Types \(p. 79\)](#), except for `b1`.

For more information on:

- Variable initializers and array dimensions, see [4.10 Initializers and Array Declarations \(p. 74\)](#).
- Identifier scopes, see [4.6.2 Scope \(p. 63\)](#).
- Variable storage duration, see [4.11 Storage Duration \(p. 76\)](#).

4.3.9 Fbarrier

A module fbarrier can either be a declaration or a definition. A code block fbarrier can only be a definition.

An fbarrier declaration or definition establishes the name of a fine-grain barrier.

A module fbarrier with the same name can be declared in a module zero or more times, but can be defined at most once.

All module fbarriers with the same name in a module denote the same fine-grain barrier.

If a module fbarrier has program linkage, then there can be at most one definition of an fbarrier with program linkage with that name amongst all the modules in the same program. All module fbarriers with program linkage in any module of the same program that have the same name denote the same fine-grain barrier. This allows a module fbarrier to be defined in one module, but used in another module of the same program. Otherwise, the module fbarrier has module linkage and can only be referenced within the same module. See [4.12 Linkage \(p. 77\)](#).

There can only be one code block fbarrier with a specific name in the scope of its identifier. The same name is allowed as a code block fbarrier in a different scope. For example, there can be multiple function scope fbarriers with the same name if they are defined in different functions or kernels. See [4.6.2 Scope \(p. 63\)](#).

A code block fbarrier has function linkage and can only be referenced within the function scope it is defined.

At the time a kernel is finalized, there must be a definition for all the fine-grain barriers referenced by fbarrier operations that are part of the kernel (including any indirect references from operations in functions they call by `call` and `scall` operations) in one of the modules that belong to the program.

A single module can contain multiple fbarrier declarations and definitions.

A module fbarrier declaration or definition consists of `decl` if a declaration, followed by its linkage, the fbarrier name and terminated by a semicolon (`;`).

A code block fbarrier can only be a definition and has function linkage. Therefore, it is defined the same as a module fbarrier except `decl` and linkage are not specified.

Figure 6–37 moduleFbarrier Syntax Diagram

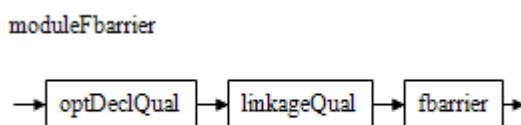


Figure 6–38 codeBlockFbarrier Syntax Diagram

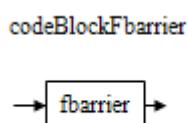
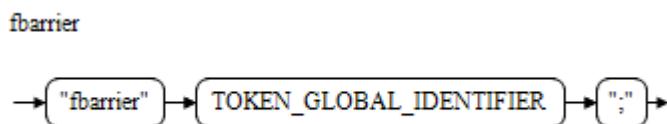


Figure 6–39 fbarrier Syntax Diagram



For more information, see [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#).

4.3.10 Declaration and Definition Qualifiers

There are multiple qualifiers that can be used with certain declarations and definitions.

Figure 6–40 optDeclQual Syntax Diagram

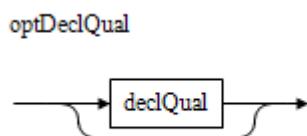


Figure 6–41 declQual Syntax Diagram

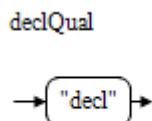


Figure 6–42 linkageQual Syntax Diagram

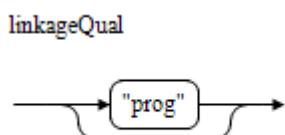


Figure 6–43 optAlignQual Syntax Diagram

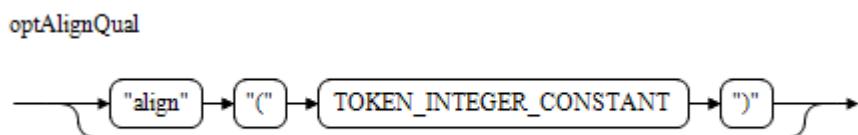


Figure 6–44 optAllocQual Syntax Diagram

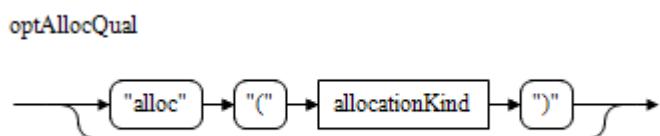
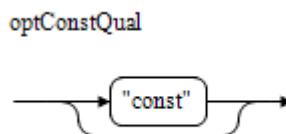


Figure 6–45 optConstQual Syntax Diagram

**decl**

Specifies that the symbol is being declared and not defined. Only allowed for kernels, functions, module variables and module fbarriers. If omitted then the symbol is being defined. Note, code block variables, code block fbarriers and arg block variables are always implicitly definitions.

prog

Specifies that the symbol has program linkage. Only allowed for kernels, functions, module variables and module fbarriers. If omitted:

- Kernels, functions, module variables and module fbarriers default to module linkage.
- Code block variables, code block fbarriers, kernel and function definition formal arguments default to function linkage.
- Arg block variables default to arg linkage.
- Signature definition, kernel declaration and function declaration formal arguments default to none linkage.

See [4.12 Linkage \(p. 77\)](#).

alloc(allocationKind)

Specifies the allocation for a variable. Only available for global segment variables, in both module and function scopes. Valid value of *allocationKind* is *agent*. If omitted defaults to: agent allocation for readonly segment variables; program allocation for global segment variables; none allocation for kernel declaration, function declaration and signature definition formal argument variables; and automatic allocation for all other segment variables.

program allocation

Causes the HSA runtime to perform a single allocation for the variable definition.

In HSAIL all references to the variable access the same single allocation. An *lda* operation performed on the variable returns a segment address that can be used by any agent.

The variable's global segment address can be converted to a flat address and used by any agent.

An HSA runtime query can be used to obtain the segment address of the variable which can be used to access it by any agent.

The definition of the variable may have an initializer. However, image and sampler initializers are not allowed (see [7.1.7 Image Creation and Image Handles \(p. 214\)](#) and [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)).

agent allocation

Causes the HSA runtime to perform a separate allocation for the variable for each agent that is a member of the program. Each separate allocation will have a distinct unique global segment address. It is undefined to access the variable from any agent other than the one it is associated with, except by HSA runtime copy operations. An implementation may allocate such variables on special agent local memory that is not directly accessible from other agents.

In HSAIL, any access to the variable by a kernel executing on an HSA component will access the variable allocation that is associated with that agent. An *lda* operation performed on the variable will obtain the distinct segment address for the allocation associated with the HSA component on which it is executed, but it is undefined for any other agent to access that address. The variable's global segment address can be converted to a flat address, but it is undefined for any other agent to access that address.

An HSA runtime query can be used to obtain the segment address of the variable for a specified agent.

The definition of the variable may have an initializer. Every separate allocation will be initialized. For image and sampler initializers, the format of the agent with which the allocation is associated will be used (see [7.1.7 Image Creation and Image Handles \(p. 214\)](#) and [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)).

automatic allocation

Causes variables to be automatically allocated at the start of the variable's storage duration. See [4.11 Storage Duration \(p. 76\)](#).

none allocation

Causes no allocation and is used for declarations.

align(*n*)

Specifies that the storage for the variable must be aligned on a segment address that is an integer multiple of *n*. Valid values of *n* are 1, 2, 4, 8, 16, 32, 64, 128 and 256.

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

Without an `align` qualifier, the variable will be naturally aligned. That is, the segment 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.

If the segment of the variable can be accessed by a flat address, then the alignment also specifies that the flat address is a multiple of the variable's alignment.

The `lda` operation cannot be used to obtain the address of an arg or spill segment variable. However, any `align` variable qualifier can serve as a hint of how the variable is accessed, and the finalizer may choose to honor the alignment if allocating the variable in memory.

const

Specifies that the variable is a constant variable. A constant variable cannot be written to after it has been defined and initialized. Only global and readonly segment variable definition and declarations can be marked `const`. Kernarg segment variables are implicitly constant variables.

Global and readonly segment variable definitions with the `const` qualifier must have an initializer. If the initial value needs to be specified by the host application, then only provide variable declarations in HSAIL modules, and use the HSA runtime to specify the variable definition together with an initial value.

Memory for constant variables remains constant during the storage duration of the variable. See [4.11 Storage Duration \(p. 76\)](#).

It is undefined to use a store or atomic operation with a constant variable, whether using a segment or flat address expression. It is undefined if implementations will detect stores or atomic operations to constant variables.

The finalizer might place constant variables in specialized read-only caches.

See [17.9 Constant Access \(p. 314\)](#).

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` and `readonly` can appear either inside or outside of a kernel or function. Such variables that appear outside of a kernel or function have module scope. Those defined inside a kernel or function have function scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of multiple variables in the global and readonly segments is implementation defined, except that the memory address is required to honor the alignment requirements of the variable's type and any `align` type qualifier.

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 module scope. Those defined inside a kernel or function have function scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of multiple variables in the group segment is implementation defined, except that the memory address is required to honor the alignment requirements of the variable's type and any `align` type qualifier.

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 module scope. Those defined inside a kernel or function have function scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of multiple variables in the private segment is implementation defined, except that the memory address is required to honor the alignment requirements of the variable's type and any `align` type qualifier.

Kernarg

The value of the variable can be accessed by all work-items in the grid. It is a formal argument of the kernel.

Declarations for `kernarg` must be in a kernel argument list. Such variables in a kernel definition have function scope, and those in a kernel declaration have signature scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of variables in the kernarg segment is defined in [4.21 Kernarg Segment \(p. 96\)](#).

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 function scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of multiple variables in the spill segment is implementation defined, and a finalizer may promote them to ISA registers. The `lda` operation cannot be used to obtain the address of a spill segment variable. However, any `align` variable qualifier can serve as a hint of how the variable is accessed, and the finalizer may choose to honor the alignment if allocating the variable in memory.

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 `arg` block within a kernel or function code block, within a function formal argument list, or within a function signature. Such variables that appear inside an argument scope have argument scope. Those that appear inside a function definition formal argument list have function scope. Those that appear in a function declaration or function signature formal argument list have signature scope. See [4.6.2 Scope \(p. 63\)](#).

The memory layout of multiple variables in the `arg` segment is implementation defined, and a finalizer may be promote them to ISA registers. The `lda` operation cannot be used to obtain the address of an `arg` segment variable. However, any `align` variable qualifier can serve as a hint of how the variable is accessed, and the finalizer may choose to honor the alignment if allocating the variable in memory.

See [4.11 Storage Duration \(p. 76\)](#) for a description of when storage is allocated for variables.

Also see:

- [7.1.7 Image Creation and Image Handles \(p. 214\)](#)
- [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)

Here is an example:

```
function &fib(arg_s32 %r) (arg_s32 %n)
{
    private_s32 %p;           // allocate a private variable
                           // to hold the partial result
    ld_arg_s32 $s1, [%n];
    cmp_lt_b1_s32 $c1, $s1, 3; // if n < 3 go to return
    cbr_b1 $c1, @return;
    {
        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;
};
```

4.4 Source Text Format

Source text sequences are ASCII characters.

The source text 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.

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

The source text is broken into the following lexical tokens:

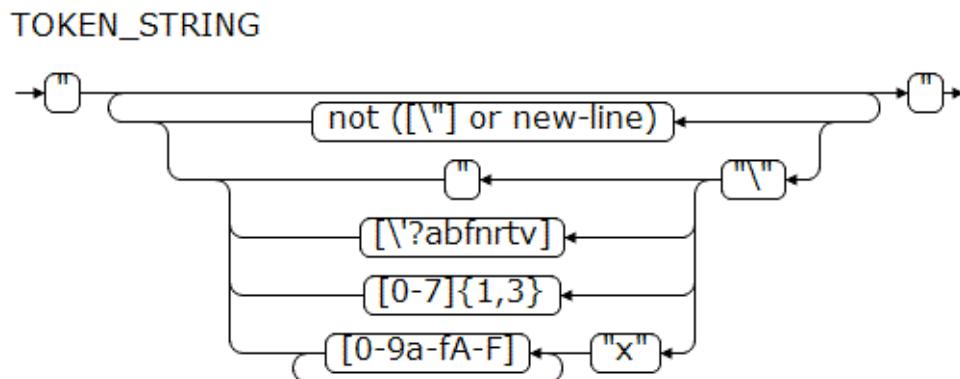
- TOKEN_COMMENT (see [4.3.1 Annotations \(p. 40\)](#))
- TOKEN_GLOBAL_IDENTIFIER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_LOCAL_IDENTIFIER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_LABEL_IDENTIFIER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_CREGISTER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_SREGISTER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_DREGISTER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_QREGISTER (see [4.6 Identifiers \(p. 62\)](#))
- TOKEN_INTEGER_CONSTANT (see [4.8.1 Integer Constants \(p. 67\)](#))
- TOKEN_HALF_CONSTANT (see [4.8.2 Floating-Point Constants \(p. 69\)](#))
- TOKEN_SINGLE_CONSTANT (see [4.8.1 Integer Constants \(p. 67\)](#))
- TOKEN_DOUBLE_CONSTANT (see [4.8.1 Integer Constants \(p. 67\)](#))
- TOKEN_WAVESIZE (see [2.6.2 Wavefront Size \(p. 12\)](#))
- TOKEN_STRING (see [4.5 Strings \(p. 61\)](#))
- Tokens for all the terminal symbols of the HSAIL syntax (see [19.2 HSAIL Syntax Grammar in Extended Backus-Naur Form \(EBNF\) \(p. 381\)](#))

Lexical tokens can be separated by zero or more white space characters: space, horizontal tab, new-line, vertical tab and form-feed.

See [19.1 HSAIL Lexical Grammar in Extended Backus-Naur Form \(EBNF\) \(p. 379\)](#).

4.5 Strings

Figure 6–46 TOKEN_STRING Syntax Diagram



A string is a sequence of characters and escape sequences 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 enter using a regular keyboard and do not display in a consistent way.

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)
 - v - vertical tab character (octal 013)

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.

4.6 Identifiers

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

Figure 6–47 TOKEN_GLOBAL_IDENTIFIER Syntax Diagram

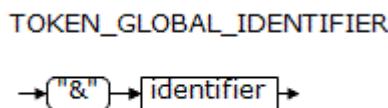


Figure 6–48 TOKEN_LOCAL_IDENTIFIER Syntax Diagram

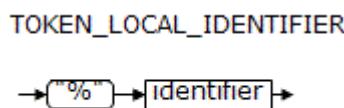


Figure 6–49 TOKEN_LABEL_IDENTIFIER Syntax Diagram

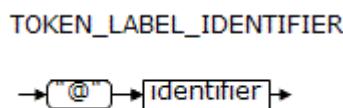
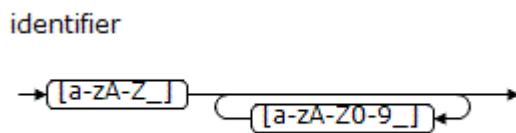


Figure 6–50 identifier Syntax Diagram



4.6.1 Syntax

Identifiers that are register names must start with a dollar sign (\$). See [4.7 Registers \(p. 64\)](#).

Identifiers that are labels must start with an at sign (@). See [4.9 Labels \(p. 74\)](#).

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 module 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`, `TOKEN_DREGISTER`, or `TOKEN_QREGISTER`. See [4.7 Registers \(p. 64\)](#).

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 module. See also [4.3.8 Variable \(p. 49\)](#).

4.6.2 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 within a single module must have different scopes.

There are four kinds of scopes:

- Module
- Function
- Argument
- Signature

Every identifier has scope determined by the placement of the declaration or definition that it names:

- If the declaration or definition appears outside of any function or kernel code block, the identifier has module scope, which extends from the point of declaration or definition to the end of the module.
- If an identifier appears as a formal argument definition in a kernel or function definition, it has function scope, which extends from the point of declaration to the end of the kernel or function's code block.
- If an identifier appears as an arg segment variable definition in an arg block, it has argument scope, which extends from the point of definition to the end of the arg block. See [10.2 Function Call Argument Passing \(p. 254\)](#).
- Label definitions have function scope which extends from the start to the end of the enclosing code block (even if defined in a nested arg block).

- Any registers used in a kernel or function code block are implicitly defined. Registers have function scope which extends from the start to the end of the enclosing code block (even if used in a nested arg block).
- If the definition appears inside a kernel or function code block, the identifier has function scope, which extends from the point of definition to the end of the code block.
- If an identifier appears as a formal argument definition of a kernel declaration, function declaration or signature definition, then it has signature scope, which extends from the point of definition to the end of the kernel declaration, function declaration or signature definition respectively.

HSAIL uses a single name space for each scope for all object kinds. In HSAIL the following object kinds can be named by an identifier: kernel, function, signature, variable, fbarrier, label and register.

Kernels, functions, signatures, variables and fbarriers declared or defined outside a kernel or function with module scope must have unique names within the enclosing module, but are not required to be unique with respect to the module scopes of other modules. The exception is that there can be zero or more declarations and at most one definition of the same object by specifying the same name for matching objects. Additionally, the linkage rules require there only be at most one module scope name that is the definition of an object with program linkage amongst all the modules that belong to the same program.

Variables, fbarriers, labels and registers defined inside a kernel or function must have unique names within the enclosing function scope, but are not required to be unique with respect to other function scopes that can define distinct objects with the same name.

Arg segment variable names defined inside an arg block with argument scope must be unique within the argument scope, but can have the same name as the arg segment variables in other argument scopes, or the names of objects in the enclosing function scope (in which case the arg segment variable name hides the function scope name).

4.7 Registers

Figure 6–51 TOKEN_CREGISTER Syntax Diagram



Figure 6–52 TOKEN_SREGISTER Syntax Diagram

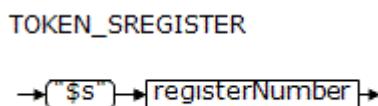


Figure 6–53 TOKEN_DREGISTER Syntax Diagram

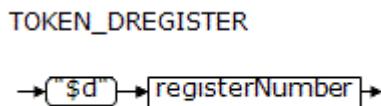


Figure 6–54 TOKEN_QREGISTER Syntax Diagram

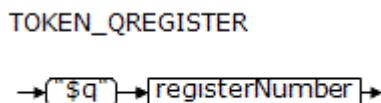
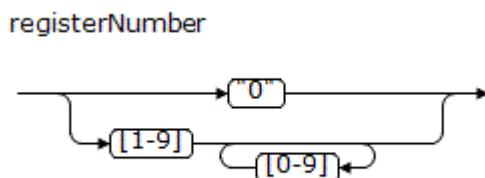


Figure 6–55 registerNumber Syntax Diagram



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 $\$c7$.

In the Extended Backus-Naur Form syntax, a control register is referred to as a TOKEN_CREGISTER.

- 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 $\$s127$.

In the Extended Backus-Naur Form syntax, a 32-bit register is referred to as a TOKEN_SREGISTER.

- 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 $\$d63$.

In the Extended Backus-Naur Form syntax, a 64-bit register is referred to as a TOKEN_DREGISTER.

- 128-bit registers (q registers)

These hold packed data.

Every HSAIL implementation supports 32 128-bit registers.

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

In the Extended Backus-Naur Form syntax, a 128-bit register is referred to as a TOKEN_QREGISTER.

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 per function scope. It is an error if the value $((s_{\max}+1) + 2 * (d_{\max}+1) + 4 * (q_{\max}+1))$ exceeds 128 for any kernel or function definition, where s_{\max} , d_{\max} , and q_{\max} are the highest register number in the kernel or function code block for the corresponding register type, or -1 if no registers of that type are used. For example, if a function code block only uses registers $\$s0$ and $\$s7$, then s_{\max} is 7 not 2. 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.

4.8 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.

All constants have a type determined by where they are used. Constant values are used for:

- Data initialization directives: the constant type is the element type of the variable being initialized.
- Operation source operands: the constant type is the type of the operand, which can be specified by the operation or be a fixed type defined by the operation.
- Operation address expressions: the type is an unsigned integer of the address size. See [Table 4-3 \(p. 25\)](#). This is true for both an absolute address and the address offset.
- Directive and version statement operands: the type of each operand is specified by the directive.
- Other usage: the type used is `u64`. These include array dimensions, image size properties, alignment, equivalence class, and so forth.

4.8.1 Integer Constants

Figure 6–56 TOKEN_INTEGER_CONSTANT Syntax Diagram

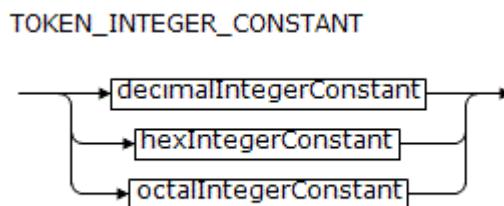


Figure 6–57 decimalIntegerConstant Syntax Diagram

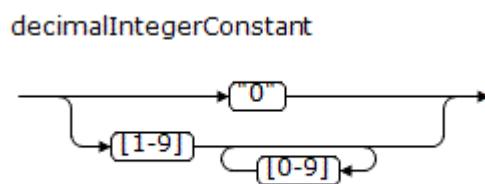


Figure 6–58 hexIntegerConstant Syntax Diagram

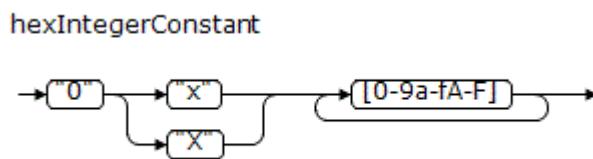
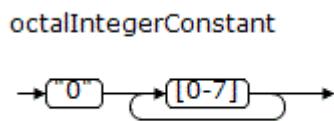


Figure 6–59 octalIntegerConstant Syntax Diagram



Integer constants are 64-bit unsigned values. In the Extended Backus-Naur Form syntax, an integer constant is referred to as a TOKEN_INTEGER_CONSTANT.

Integer constants are only valid for integer types, and for bit types and packed types less than or equal to 64 bits. See [4.13.1 Base Data Types \(p. 79\)](#) and [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#). The type size determines the number of least significant bits of the 64-bit integer constant value that are used; any remaining bits are ignored. For signed integer types, the bits are treated as a two's complement signed value. The exception is the b1 type used in controls, which is interpreted as in C and C++: a zero 64-bit value is treated as 0 and a non-zero 64-bit value is treated as 1 (that is, zero values are False and non-zero values are True).

Some uses of integer constants allow an optional + and – sign before the integer constant. For –, the integer constant value is treated as a two's complement value and negated, regardless of whether the constant type is a signed integer type, and the resulting bits used as the value.

In BRIG, the size of an integer constant immediate operand value must be the number of bytes needed by the constant type. For b1, a single byte is used and must be 0 or 1.

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 64-bit unsigned values, but the operation expects 32-bit signed types. The least significant 32 bits of the 64-bit integer constant are treated as a 32-bit signed value:

```
global_s32 %someident[] = {24, 25};
add_s32 $s1, 24, 25;
```

Integer constants can be written in decimal, hexadecimal, or octal form, following the C++ syntax:

- A decimal integer constant starts with a non-zero digit. See [Figure 6–57 \(p. 67\)](#).
- A hexadecimal integer constant starts with 0x or 0X. See [Figure 6–58 \(p. 67\)](#).
- An octal integer constant starts with 0. See [Figure 6–59 \(p. 68\)](#).

4.8.2 Floating-Point Constants

Figure 6–60 TOKEN_HALF_CONSTANT Syntax Diagram

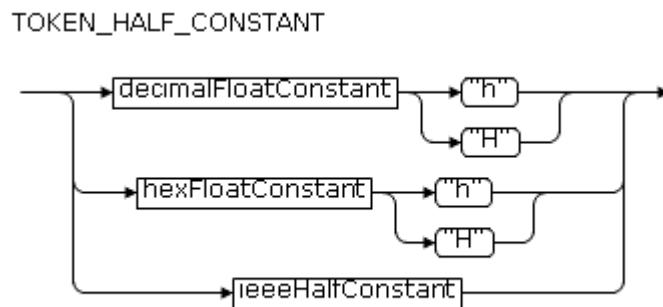


Figure 6–61 TOKEN_SINGLE_CONSTANT Syntax Diagram

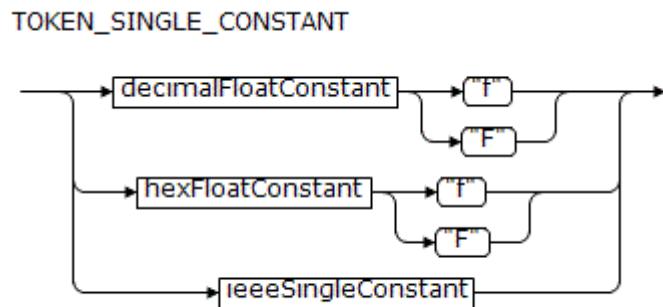


Figure 6–62 TOKEN_DOUBLE_CONSTANT Syntax Diagram

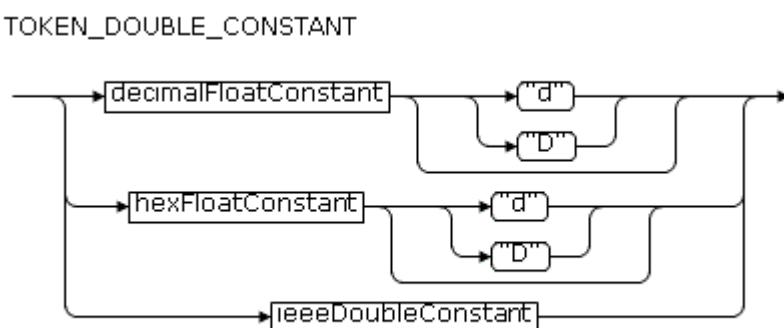


Figure 6–63 decimalFloatConstant Syntax Diagram

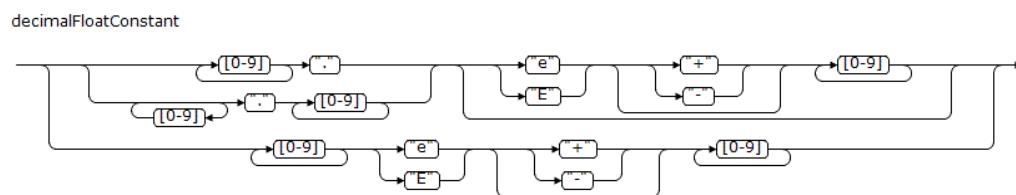


Figure 6–64 hexFloatConstant Syntax Diagram

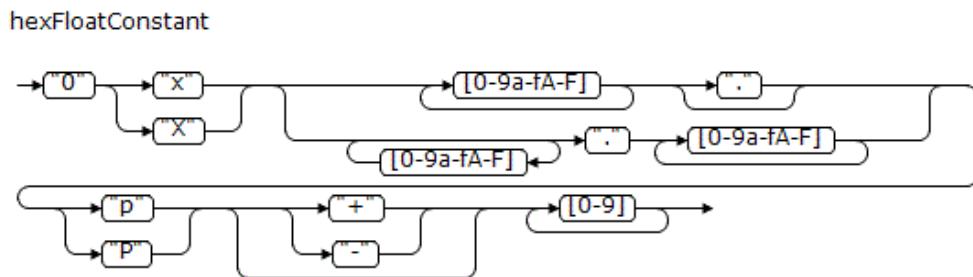


Figure 6–65 ieeeHalfConstant Syntax Diagram

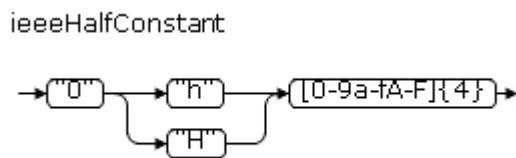


Figure 6–66 ieeeSingleConstant Syntax Diagram

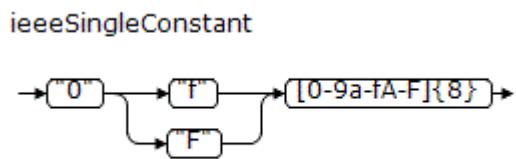
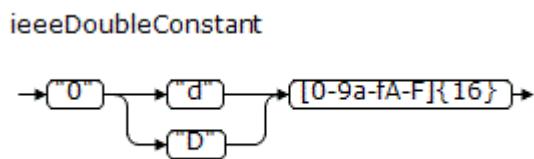


Figure 6–67 ieeeDoubleConstant Syntax Diagram



Floating-point constants are represented as either:

- 16-bit single-precision

It is an error to use a half-precision float constant unless the constant type is `f16` or `b16`. See [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#). In Extended Backus-Naur Form syntax, a half-precision float constant is referred to as a `TOKEN_HALF_CONSTANT`.

- 32-bit single-precision

It is an error to use a single-precision float constant unless the constant type is `f32` or `b32`. See [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#). In Extended Backus-Naur Form syntax, a single-precision float constant is referred to as a `TOKEN_SINGLE_CONSTANT`.

- 64-bit double-precision

It is an error to use a double-precision float constant unless the constant type is `f64` or `b64`. See [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#). In Extended Backus-Naur Form syntax, a double-precision float constant is referred to as a `TOKEN_DOUBLE_CONSTANT`. Neither the 64-bit floating-point type (`f64`) nor the 64-bit double-precision floating-point constant formats are supported by the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

Some uses of floating-point constants allow an optional + and - sign before the floating-point constant. For -, the sign bit of the floating-point representation of the constant type is inverted, no other bits are changed, and the resulting bits are used as the value.

In BRIG, the size of a floating-point constant immediate operand value must be the number of bytes needed by the constant type.

Floating-point constants can be written in decimal or hexadecimal form following the C++ syntax. In addition, they can be specified using the IEEE/ANSI Standard 754-2008 binary interchange format:

- A decimal floating-point constant can be written with a significand part, a decimal exponent part, and a float size suffix. The significand part represents a rational number and consists of a sequence of decimal digits (the whole number) followed by an optional fraction part (a period followed by a sequence of decimal digits). The decimal exponent part is an optionally signed decimal integer that indicates the power of 10. The significand is raised to that power of 10. The float size suffix indicates the type: h or H indicates 16 bits; f or F indicates 32 bits; d or D indicates 64 bits. The float size suffix can be omitted for double-precision decimal float constants, but is required for half-precision and single-precision decimal float constants. The decimal floating-point constant is converted to the memory representation using convert to nearest even (see [4.19.2 Rounding \(p. 92\)](#)). See [Figure 6-63 \(p. 70\)](#).
- A hexadecimal floating-point constant can be written using the C99 format. It consists of a hexadecimal prefix of 0x or 0X, a significand part, a binary exponent part, and a float size 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 float size suffix indicates the type: h or H indicates 16 bits; f or F indicates 32 bits; d or D indicates 64 bits. The float size suffix can be omitted for double-precision hexadecimal float constants, but is required for half-precision and single-precision hexadecimal float constants. See [Figure 6-64 \(p. 70\)](#).
- An IEEE/ANSI Standard 754-2008 binary interchange double-precision floating-point constant begins with 0d or 0D followed by 16 hexadecimal digits. A single-precision floating-point constant begins with 0f or 0F followed by eight hexadecimal digits. A half-precision floating-point constant begins with 0h or 0H followed by four hexadecimal digits.

A double like 12.345 can be written as 0d4028b0a3d70a3d71 or 0x1.8b0a3d70a3d71p+3.

4.8.3 Packed Constants

For information, see [4.14.2 Packed Constants \(p. 83\)](#).

4.8.4 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 kind 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 6–1 Text Constants and Results of the Conversion

Kind of text format constant provided (see 4.8 Constants (p. 66))	Data type of expected value (see 4.13 Data Types (p. 79))			
	Signed/unsigned integer	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 $d0, $d0, 0xffffffff; // Legal: 9 f's stored as 0x0000000ffffffffff.
```

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 0x0000000ffffffffff.

```
add_s8x4 $s0, $s0, 0xffffffff; // Legal: 9 f's truncated and stored as 0xffffffff.
```

The s8x4 is four signed 8-bit elements, for a total of 32 bits, the constant would be truncated and stored as b32 0xffffffff.

It is not possible to provide an integer constant to a 128-bit data type:

```
mov_b128 $q0, 0xffffffff; // Illegal: integer constant is evaluated as 64 bits
                           // and operation requires 128 bits.
add_s32x4 $q0, $q0, 0xffffffff; // Illegal: four elements each a signed 32-bit
                               // number is 128 bits, there is not be enough
                               // data so it is an error.
```

However, a packed constant can be used for a 128-bit data type. For example, 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.
```

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.

```
mov_u64x2 $q1, _u64x2(1, 2); // Legal as packed types match.
mov_u64x2 $q1, _u32x4(1, 2, 3, 4); // Illegal as packed types do not match even
                                         // though size does.
mov_f32 $s1, 1.0f; // Legal as floating-point constant size matches operand size.
mov_f32 $s1, 1.0d; // Illegal as floating-point constant size does not match operand size.
```

The length-only rule is the following: the bits in the constant are used provided the number of bits is the same.

```
mov_b32 $s1, 3.7f; // Legal as size of floating-point constant and operand type are 32.
mov_b32 $s1, 3.7d; // Illegal as floating-point constant and operand type size mismatch.
```

For mov_b32 \$s1, 3.7f, although the types do not match, the lengths do match, so the binary representation of value 3.7f is used.

When `WAVESIZE` is allowed as a constant, it is treated exactly the same as an integer constant with a 64 bit value that is equal to the value of `WAVESIZE`. See [2.6.2 Wavefront Size \(p. 12\)](#).

4.9 Labels

Label identifiers consist of an at sign (@) followed by the name of the identifier (see [4.6 Identifiers \(p. 62\)](#)).

Label definitions consist of a label identifier followed by a colon (:).

Label identifiers cannot be used in any operations except `br`, `cbr`, and `sbr`.

Label identifiers cannot appear in an address expression.

See [Chapter 8 Branch Operations \(p. 231\)](#).

4.10 Initializers and Array Declarations

Variable definitions in the global and readonly segments can specify an initial value. The variable name is followed by an equals (=) sign and the initial value or values for the variable.

It is not possible to initialize variables in segments other than the global and readonly segments.

For a global or readonly segment variable definition with the `const` qualifier, an initializer is required. For a global and readonly segment variable without the `const` qualifier, an initializer is optional.

When a global segment or readonly segment variable is defined it is allocated and an initial value assigned if it has an initializer.

When a global segment variable is initialized a release memory fence on the global segment at system memory scope is performed. Initialization can either be by adding a module that contains a definition with an initializer to a program, or by adding a definition to the program by an HSA runtime operation. It is undefined if a kernel dispatch does not use appropriate memory synchronization to access the variable after it has been initialized.

A readonly segment variable has agent allocation, and so has distinct memory allocations for each agent that is a member of the program (see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#)). When a readonly segment variable is defined and initialized, either by adding a module with a definition to a program, or by adding a definition to the program by an HSA runtime operation, the HSA runtime makes each agent allocation value visible to all subsequent dispatches on the corresponding agent. The HSA runtime can also be used to change the value of a non-`const` readonly variable after it has been defined for a specific agent that is a member of the program: this also makes the value visible to all subsequent dispatches on the corresponding agent. It is undefined to access the agent allocation by kernel dispatches that were executing before the variable's definition initialization, or HSA runtime update.

For the initialization of image and sampler handles, see [7.1.7 Image Creation and Image Handles \(p. 214\)](#) and [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#).

For the initialization of signal handles, the initial value must be the constant value 0 to indicate the null signal handle.

For all other types, any constant value can appear in the initializer. However, WAVELENGTH is not allowed.

The values of an initializer 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 6-1 \(p. 73\)](#) apply. For example, the following initializes the identifier `&x` to the 32-bit value 0x40400000:

```
global_b32 &x = 3.0f;
```

Variables that hold addresses of variables, kernel descriptors or indirect function descriptors should be of type `u` and of size 32 or 64 depending on the machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)).

Array variables are provided to allow the high-level compiler to reserve space. To declare or define an array variable, the variable name is followed with an array dimension declaration. The size of the dimension is either an integer constant of type `u64` or is left empty. WAVELENGTH is not allowed.

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.

If the variable does not specify an array dimension, then the initial value can be specified as a single value. If the variable does specify an array dimension, then the initial value can be specified as a comma separated list of one or more values inside curly braces.

The array dimension of a variable definition can be left empty and set by the length of the initialization list (note that this follows the C++ rules):

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

If there is no initializer, the value of the variable is undefined. If some, but not all of the elements of an array variable are initialized, then:

- If the variable has an image or sampler type then the rest of the elements are undefined.
- Otherwise, the rest of the elements 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
```

It is an error if the size of the initializer is larger than the array dimension size.

The array size of a global or readonly segment variable declaration can be left empty, in which case the size is specified by the variable definition (note that this follows the C++ rules).

The last formal argument of a function or signature can be an array without a specified size. The size passed is determined by the size of the arg segment variable definition passed to the function by the call operation. This is used to support variadic function calls. See [10.4 Variadic Functions \(p. 259\)](#).

Examples

```
global_u32 &loc1;
global_f32 &size = 1.0;
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};
decl global_u32 &c[];
global_u32 &c[4];
global_sig64 &s1 = 0; // Signal handles must be initialized with 0.
global_sig64 &sa[2] = {0, 0x00};
global_sig64 &sa[4] = {0, 0x00}; // All 4 signal handles are initialized to null handle.
global_sig64 &se[] = {0, 0x00, 0}; // Equivalent to &se[3].
```

4.11 Storage Duration

Global and readonly segment variable definitions can be used to allocate blocks of memory. The memory is allocated when the HSAIL module containing the definition is added to the HSAIL program and lasts until the HSAIL program is destroyed. This corresponds to the C++ notion of static storage duration. (See the C++ specification ISO/IEC 14882:2011.)

Kernarg segment variable definitions that appear in a kernel's formal arguments are allocated when a kernel dispatch starts and released when the kernel dispatch finishes.

Group segment variable definitions that appear inside a kernel, or at module 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 segment variable definitions that appear inside any function that can be called by the kernel are allocated the same way. This is because group segment 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 segment memory. Dynamically allocated group segment memory is also allocated the same way (see [4.20 Dynamic Group Memory Allocation \(p. 95\)](#)).

Private and spill segment variable definitions that appear inside a kernel are allocated when a work-item starts executing the kernel, and last until the work-item exits the kernel.

Private segment variable definitions that appear at module scope (spill cannot appear at module scope) and are used by a 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 segment variable definitions 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 segment variable definitions inside an arg block are allocated each time the arg block is entered by a work-item, and last until the work-item exits the arg block.

Recursive calls to a function will allocate multiple copies of private, spill and arg segment variables defined in the function's code block. 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.

Fbarrier definitions have the same allocation as group segment variables.

Kernel and indirect function definitions allocate a kernel descriptor and indirect function descriptor respectively the same way as global segment variable definitions.

For more information see [4.2 Program \(p. 35\)](#) and [4.3 Module \(p. 38\)](#).

4.12 Linkage

Linkage determines the rules that specify how a name (kernel, function, variable or fbarrier) refers to an object. It can allow the same name within a single module, or in multiple modules, to refer to the same object.

See [4.6.2 Scope \(p. 63\)](#).

4.12.1 Program Linkage

A name of a kernel, function, variable or fbarrier in one module can refer to an object with the same name defined in a different module. The two names are linked together. Only one module in a program is allowed to have a definition for the name, and must be marked `prog`. In all other modules that refer to the same object, the name must be a declaration, and must be marked `decl prog`. Objects that can be linked together in this way are said to have *program linkage*.

A name can be both declared and defined in the same module.

Only module scope program linkage declarations can be marked `decl prog`.

Only module scope program linkage definitions can be marked `prog`.

A kernel or function declaration marked `decl prog` cannot have a body, because that would make it a definition.

A variable marked `decl prog` is not a definition, so it cannot have an initializer.

No definition or declaration for the same name can have both module and program linkage in the same module.

Module scope objects are: global, group, private and readonly segment variables; kernel, function and fbarriers.

The finalizer does not allocate space for names marked `decl prog`, only for those marked `prog`.

For example:

```
decl prog function &foo()(); // program linkage declaration:  
                          // says it is defined elsewhere  
                          // in the same module or is  
                          // defined in another module.  
// ...  
  
prog function &foo()() // program linkage definition: contains the body  
{  
    // ... the body  
};
```

4.12.2 Module Linkage

A name of a kernel, function, variable or fbarrier in one module can be restricted to only be visible in a single module. All declarations and definitions with the same name in a single module refer to the same object, and declarations must be marked `decl`. The same name can appear in other modules but refers to a different object. Objects that are linked together in this way are said to have *module linkage*.

A module must have at most one module linkage definition for the name.

A module can have zero or more module linkage declarations for the name.

Only module scope module linkage declarations can be marked `decl`.

Only module scope module linkage definitions can omit `decl`.

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

A variable marked `decl` is not a definition, so it cannot have an initializer.

No definition or declaration for the same name can have both module and program linkage in the same module.

Module scope objects are: global, group, private and readonly segment variables; kernel, function and fbarriers.

The finalizer does not allocate space for names marked `decl`, only for those that are definitions.

For example:

```
decl function &foo()(); // module linkage declaration:  
                      // says it is defined elsewhere  
                      // in the same module.  
// ...  
  
function &foo()()  
{ // module linkage definition: contains the body  
  // ... the body  
};
```

4.12.3 Function Linkage

Definitions in function scope are only visible in the corresponding code block. The same name can appear in different function scopes and refers to different objects. Only definitions are allowed in function scope.

Function scope objects are: global, group, private, spill and readonly segment variables; kernarg segment variables that are kernel definition formal arguments; arg segment variables that are function definition formal arguments; labels and fbarriers.

For example:

```
function &foo()()  
{  
  global_u32 %v; // function linkage definition:  
                  // only visible in function &foo.  
  // ...  
};
```

4.12.4 Arg Linkage

Definitions in argument scope are only visible in the corresponding arg block. The same name can appear in function scopes and different arg scopes and refers to different objects. Only definitions are allowed in argument scope.

Argument scope objects are: arg segment variables in an arg block.

For example:

```
function &foo()()
{
    // ...
    // Start or arg block
    arg_u32 %v; // arg linkage definition:
                 // only visible in arg block.
    // ...
} // end of arg block
// ...
};
```

4.12.5 None Linkage

Definitions in signature scope are only visible in the associated formal argument lists. They do not refer to any object. The same name can appear in other scopes and refer to different objects.

Signature scope objects are: arg segment variables in the formal argument list of kernel declaration, function declarations and signature definitions.

For example:

```
// none linkage: %x only visible in signature and has no allocation.
signature &foo() (arg_u32 %x);
```

4.13 Data Types

4.13.1 Base Data Types

HSAIL has four base data types, each of which supports a number of bit lengths. See [Table 6–2 \(p. 79\)](#).

Table 6–2 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.

The 64-bit floating-point type (`f64`) is not supported by the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)). This includes segment variable declarations, segment variable definitions, double-precision floating-point constants and operations.

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.

4.13.2 Packed Data Types

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 6–3 \(p. 80\)](#).

Table 6–3 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	<code>8x4, 16x2</code>	<code>8x8, 16x4, 32x2</code>	<code>8x16, 16x8, 32x4, 64x2</code>
u	Unsigned integer	<code>8x4, 16x2</code>	<code>8x8, 16x4, 32x2</code>	<code>8x16, 16x8, 32x4, 64x2</code>
f	Floating-point	<code>16x2</code>	<code>16x4, 32x2</code>	<code>16x8, 32x4, 64x2</code>

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

The 64-bit floating-point packed type (`f64x2`) is not supported by the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)). This includes segment variable declarations, segment variable definitions, packed constants and operations.

4.13.3 Opaque Data Types

HSAIL also has the following opaque types:

Table 6–4 Opaque Data Types

Type	Description	Length in bits
<code>roimg</code>	Read-only image handle	64
<code>woimg</code>	Write-only image handle	64
<code>rwimg</code>	Read-write image handle	64
<code>samp</code>	Sampler handle	64
<code>sig32</code>	Signal handle for signal with 32 bit signal value	64
<code>sig64</code>	Signal handle for signal with 64 bit signal value	64

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

The image handle (`roimg`, `woimg`, `rwimg`) and sampler handle (`samp`) types are only supported if the "IMAGE" extension directive has been specified (see [13.1.2 extension IMAGE \(p. 292\)](#)). This includes segment variable declarations, segment variable definitions and operations.

The signal handle type for signals with a 64-bit signal value (`sig64`) is not supported by the small machine model, and the signal handle type for signals with a 32-bit signal value (`sig32`) is not supported by the large machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)). This includes segment variable declarations, segment variable definitions and operations.

For more information see:

- [7.1.7 Image Creation and Image Handles \(p. 214\)](#)
- [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)
- [6.8 Notification \(signal\) Operations \(p. 187\)](#)

4.14 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 [4.13.2 Packed Data Types \(p. 80\)](#).

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

See the tables below.

Table 6–5 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 6–6 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.)

4.14.1 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.

4.14.2 Packed Constants

Figure 6–68 packedConstant Syntax Diagram

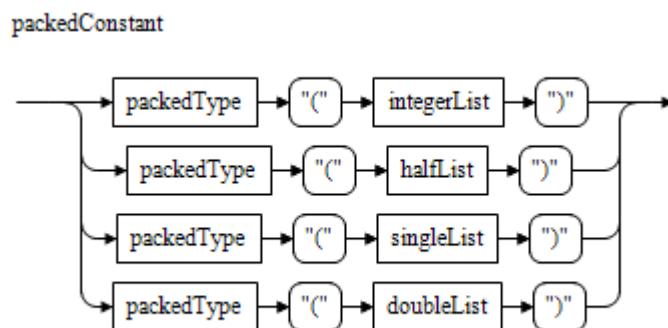


Figure 6–69 integerList Syntax Diagram

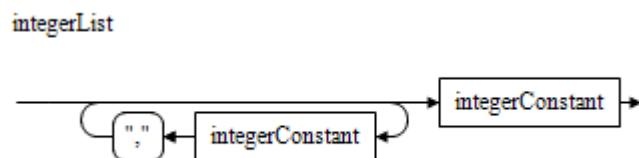


Figure 6–70 integerConstant Syntax Diagram

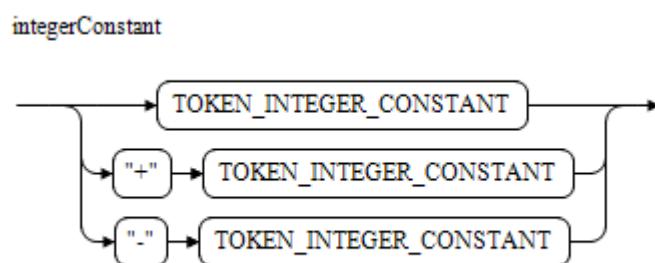


Figure 6–71 halfList Syntax Diagram

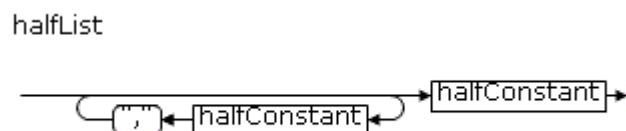


Figure 6–72 halfConstant Syntax Diagram

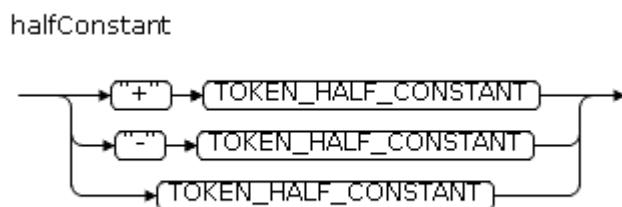


Figure 6–73 singleList Syntax Diagram

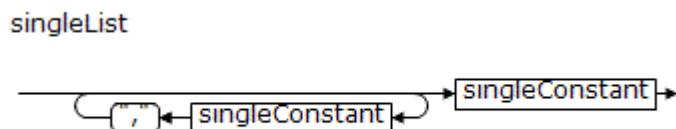


Figure 6–74 singleConstant Syntax Diagram

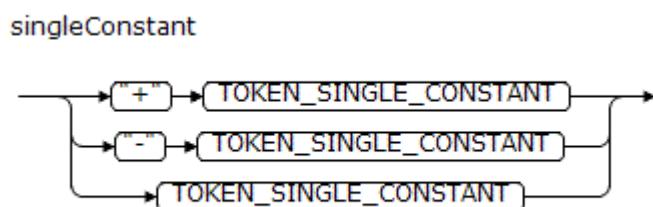


Figure 6–75 doubleList Syntax Diagram

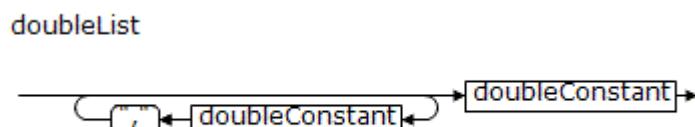
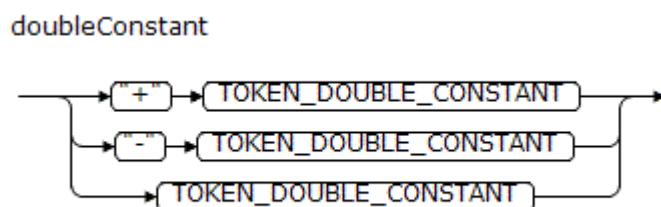


Figure 6–76 doubleConstant Syntax Diagram



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 constant 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 required to be the same size as the packed element type and is read as described in [4.8.2 Floating-Point Constants \(p. 69\)](#). The 64-bit packed floating-point type (f64x2) is not supported by the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

Bit types are not allowed.

Packed constants are only valid for bit types with the same size as the packed constant, and for packed types with the same packed type. See [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#).

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(2.0f, 1.0f);
add_pp_f32x2 $d1, $d2, 0x3f8000040000000;
```

4.14.3 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;
```

4.15 Subword Sizes

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

4.16 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 or `WAVESIZE`. See [4.8 Constants \(p. 66\)](#) and [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 \(p. 88\)](#).
- Memory, image, and some copy (move) operations allow a vector register as source and destination operands. These comprise a list of registers. See [4.17 Vector Operands \(p. 88\)](#).
- Branch operations can take a label and list of labels as a source operand. See [Chapter 8 Branch Operations \(p. 231\)](#).
- Call operations can take a function identifier, list of function identifiers, and signature identifier as a source operand. See [Chapter 10 Function Operations \(p. 253\)](#).

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.

4.16.1 Operand Compound Type

Register, immediate, and address expression operands have an associated compound type. See [4.13 Data Types \(p. 79\)](#). 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 4-3 \(p. 25\)](#).

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 [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#).

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

4.16.2 Rules for Operand Registers

The following rules apply to operand registers:

- If the operand compound type is `b1` then it must be a `c` register.
- If the operand compound type is `f16` then it must be an `s` register. The `s` register representation of `f16` is implementation-defined and might not match the memory representation, so care should be taken to use operations that correctly convert to and from `s` register representation if arithmetic is to be performed. See [4.19.1 Floating-Point Numbers \(p. 91\)](#).

The `1d` and `unpack` operations will convert the source value from the memory representation of `f16` into the `s` register representation. The `st` and `pack` operations will convert the source value from the `s` register representation of `f16` into the memory representation. All other operations will operate on the `s` register representation.

Despite using an `s` register, an `f16` value is not converted to an `f32` value. The `cvt` operation must be used to explicitly perform such a 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 the operand type is `u` or `s` with a size less than 32 bits then it must be a `s` register. (There are no other types less than 32 bits except for `b1` and `f16` which are described above.)

For source operands the size of the compound type dictates the number of least significant bits of the `s` register that are used.

For destination operands the operation is performed in the size of the operand compound type. The result is then zero-extended for `u` types, and sign-extended for `s` types, to 32 bits. For example, an `1d_u16` operation must have an `s` destination register: a 16-bit value is loaded from memory, zero-extended to 32 bits, and stored in the `s` register.

- Otherwise the source operand register size must match the size of its compound type.

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. Similarly, if it is necessary to transfer a `b1` value in a `c` register into an `s` or `d` register, or vice versa, the `cvt` operation must be used to do the appropriate testing to a `b1` value or conversion to a signed or unsigned integer value or a float value. See [5.18 Conversion \(cvt\) Operation \(p. 150\)](#).

4.17 Vector Operands

Several operations support vector operands.

Both destination and source vector operands are written as a comma-separated list of component operands enclosed in parentheses.

A `v2` vector operand contains two component operands, a `v3` vector operand contains three component operands, and a `v4` vector operand contains four component operands.

It is not valid to omit a component operand from the vector operand list.

For a destination vector operand, each component operand must be a register.

For a source vector operand, each component operand can be a register, immediate operand or `WAVESIZE`.

The type of the vector operand applies to each component operand:

- The rules for each register in a vector operand follow the same rules as registers in non-vector operands. Therefore, they must all be the same register type. In a vector operand used as a destination, it is not valid to repeat a register.
- The rules for each constant in a vector operand follow the same rules as constants in non-vector operands. See [4.8.4 How Text Format Constants Are Converted to Bit String Constants \(p. 72\)](#).

In BRIG, the type of the vector operand is the type of each component operand. See [4.16 Operands \(p. 86\)](#).

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.

Examples:

```
group_u32 %x;
readonly_s32 %tbl[256];
ld_group_u16 $s0, [%x]; // via offset
ld_group_u32 $s0, [%x];
ld_group_f32 $s2, [%x][0]; // treat result as floating-point
ld_v4_readonly_f32 ($s0, $s3, $s1, $s2), [%tbl];
ld_READONLY_s32 $s1, [%tbl][12];
ld_v4_READONLY_width(all)_f32 ($s0, $s3, $s9, $s1), [%tbl][2]; // broadcast form
ld_v2_f32 ($s9, $s2), [$s1+8];
st_v2_f32 ($s9, 1.0f), [$s1+16];
st_v4_u32 ($s9, 2, 0xffffffff, WAVESIZE), [$s1+32];
combine_v4_b128_b32 $q0, (3.14f, _f16x2(0.0h, 1.0h), -1, WAVESIZE);
```

See [6.3 Load \(ld\) Operation \(p. 171\)](#).

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.

An operation that uses an address expression operand specifies if it is a flat or segment address by the segment modifier on the operation. If the segment modifier is omitted, the operand is a flat address, otherwise it is a segment address for the segment specified by the modifier.

Address expressions consist of one of the following:

- A variable name in square brackets
- An address in square brackets
- A variable name in square brackets followed by an address in square brackets

An address is one of the following:

- register
- integer constant
- + integer constant
- - integer constant
- register + integer constant
- register - integer constant

If a variable name is specified, the variable must be declared or defined with the same segment as the address expression operand. Therefore, a flat address expression operand cannot use a variable name as variables are always declared or defined in a specific segment. For information about how to declare a variable, see [4.3.8 Variable \(p. 49\)](#).

Addresses are always in bytes. For information about how addresses are formed from an address expression, see [6.1.1 How Addresses Are Formed \(p. 157\)](#).

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 $s2, [%g1][2];   // global segment address
ld_global_f32 $s1, [%g1][0];   // the [0] is optional
ld_global_f32 $s2, [%g1][+4];
lda_global_u64 $d0, [%g1][-4];
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 global segment address 100
ld_group_f32 $s3, [%x][$s2];  // group segment-relative address
ld_group_f16 $s5, [100];       // read 16 bits at absolute global segment address 100
```

See [6.3 Load \(ld\) Operation \(p. 171\)](#).

4.19 Floating Point

HSAIL provides a rich set of floating-point operations. Most follow the IEEE/ANSI Standard 754-2008 for floating-point operations. However, there are important differences:

- If the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)) has been specified:
 - The 64-bit floating-point type (`f64`) is not supported (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).
 - The DETECT and BREAK exception policies are not supported for the five floating point exceptions specified in [12.2 Hardware Exceptions \(p. 285\)](#), therefore operations do not have to generate them as they have no observable effect (see [4.19.5 Floating Point Exceptions \(p. 94\)](#)).
- Floating-point values are stored in IEEE/ANSI Standard 754-2008 binary interchange format encoding except for `f16`. See [4.19.1 Floating-Point Numbers \(p. 91\)](#).
- For operations that follow the IEEE/ANSI Standard 754-2008, the exceptions generated are those corresponding to the set of IEEE/ANSI Standard 754-2008 status flags raised by IEEE/ANSI Standard 754-2008 default exception handling. See [12.2 Hardware Exceptions \(p. 285\)](#).
 - When exceptions are generated the result is that produced by IEEE/ANSI Standard 754-2008 default exception handling.
 - IEEE/ANSI Standard 754-2008 flags are supported using the DETECT exception policy and related operations. See [11.2 Exception Operations \(p. 274\)](#).
- Four IEEE/ANSI Standard 754-2008 rounding modes are supported for some floating-point operations. See [4.19.2 Rounding \(p. 92\)](#).
- The `ftz` (flush to zero) modifier, which forces subnormal values to zero, is supported on most operations. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).
- Operations that produce NaN results have certain requirements. See [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).
- Some operations are fast approximations (the `nsqrt` operation is an example). See [5.13 Native Floating-Point Operations \(p. 136\)](#).
- Many operations that are not in the IEEE/ANSI Standard 754-2008 are provided.
- HSAIL supports packed versions of some floating-point 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`, rounding modifiers, and exceptions.
 - The exceptions generated by the packed operation is the union of the exceptions generated for each element of the packed result.
 - For the packed `f16` types, both the source and destination elements are represented using the 16-bit memory representation, but the result elements are still computed using the `f16` implementation-defined register representation. See [4.19.1 Floating-Point Numbers \(p. 91\)](#).

4.19.1 Floating-Point Numbers

Floating-point data is stored in IEEE/ANSI Standard 754-2008 binary interchange format encoding:

- 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.

4.19.2 Rounding

Four IEEE/ANSI Standard 754-2008 rounding modes are supported 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.

If the `round` modifier is omitted, and the operation supports a rounding mode, `near` is assumed. If the Base profile has been specified, it is an error to use any rounding mode except `near`. See [16.2.1 Base Profile Requirements \(p. 309\)](#).

Floating-point operations that support the rounding modifier first compute the infinitely precise result, and then round it to the destination floating-point type. Rounding is performed according to the IEEE/ANSI Standard 754-2008 including the generation of overflow, underflow and inexact exceptions (see [12.2 Hardware Exceptions \(p. 285\)](#)):

- As specified by IEEE/ANSI Standard 754-2008 Section 7.5, it is implementation-defined if tinniness (a tiny non-zero result) is detected before or after rounding, but an implementation must use the same method for all operations.
- If the result is a NaN then the destination is set to a quiet NaN. See [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).
- Else if the result is infinity then the destination is set to an infinity with the same sign. No exceptions are generated.
- Else if the result is outside the range of representable numbers then the overflow and inexact exceptions are generated. The destination is set to either an appropriately signed infinity or appropriately signed largest representable number according to the rounding mode. `near` always rounds to infinity.
- Else if tinniness is detected and `ftz` is specified, then the destination is set to 0.0 and the underflow exception generated. It is implementation defined if the inexact exception is also generated. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).
- Else the destination is set to the rounded result. In addition:
 - If the rounded result does not exactly equal the value before rounding then the inexact exception is generated.
 - If the rounded result does not exactly equal the value before rounding and tinniness was detected then the underflow exception is generated.

4.19.3 Flush to Zero (ftz)

HSAIL supports the flush to zero `ftz` modifier on many floating-point operations that controls the flushing of source subnormal values, and tiny results to zero.

If an operation supports the `f tz` modifier then:

- If the Base profile has been specified then the `f tz` modifier must be specified. See [16.2.1 Base Profile Requirements \(p. 309\)](#).
- Otherwise, the `f tz` modifier is optional.

If `f tz` is specified on an operation that has floating-point source operands::

- For each floating-point source operand that has a subnormal value, the operation is performed using the value 0.0 instead.
- The result of the operation and any exceptions generated by the operation and any subsequent rounding are based on the flushed source values.

If `f tz` is specified on an operation that has a floating-point destination operand:

- The operation result before rounding is computed as defined by the IEEE/ANSI Standard 754-2008.
- If tinniness is detected (see [4.19.2 Rounding \(p. 92\)](#)), then the destination operand must be set to 0.0 and the underflow exception generated. It is implementation defined if the inexact exception is also generated. These exceptions are in addition to any other exception generated by the operation.
- Otherwise, the result is rounded according to the rounding modifier and stored in the destination operand. See [4.19.2 Rounding \(p. 92\)](#).

4.19.4 Not A Number (NaN)

As required by IEEE/ANSI Standard 754-2008, for all floating-point operations, except the floating-point bit operations (see [5.12 Floating-Point Bit Operations \(p. 133\)](#)) and native floating-point operations (see [5.13 Native Floating-Point Operations \(p. 136\)](#)):

- If one or more of the floating-point source operands is a signaling NaN, an invalid operation exception must be generated. Additionally, if the operation is a signalling comparison form (see [5.17 Compare \(cmp\) Operation \(p. 145\)](#)) and one or more of the source operands is a quiet NaN, then an invalid operation exception must be generated. See [12.2 Hardware Exceptions \(p. 285\)](#).
- If an operation has a floating-point destination operand and produces a NaN, it must produce a quiet NaN.
- If one or more of the floating-point source operands are NaNs, and the operation has a floating-point destination operand, then the result must be a quiet NaN.
 - The exception to this rule is `min` and `max` when one of the inputs is a quiet NaN and the other is a number, in which case the result is the number.

In addition HSAIL requires that when a NaN is produced by these operations, it must be one of the following:

- If the Base profile has been specified, it is implementation-defined what value quiet NaN is returned. It is not required to be bit-identical, after converting a signaling NaN to a quiet NaN, to one of the NaN inputs. See [16.2.1 Base Profile Requirements \(p. 309\)](#).
- If the Full profile has been specified then NaN source operands must be propagated as IEEE/ANSI Standard 754-2008 Section 6.2.3 defines should happen:
 - The quiet NaN produced must be bit-identical to one of the NaN inputs, after converting signalling NaNs to quiet NaNs, except that the sign bits may differ. If multiple inputs are a NaN, it is implementation-defined which NaN will be used. See [16.2.2 Full Profile Requirements \(p. 310\)](#).
 - The cvt operation is an exception to this rule when both the source and the destination are a floating-point type. In this case the source and destination operands are different sizes, and it is implementation defined what quiet NaN is returned. However, if a NaN is converted from a larger floating-point type to a small one and then back to the original larger floating-point type, then the final quiet NaN produced must be bit-identical to the original NaN, after converting signalling NaNs to quiet NaNs, except that the sign bits may differ.

The image operations are an exception to these rules, both when converting component values (see [7.1.4.2 Channel Type \(p. 203\)](#)), and when using a sampler with normalized coordinates (see [7.1.6.1 Coordinate Normalization Mode \(p. 209\)](#)) or a linear filter (see [7.1.6.3 Filter Mode \(p. 211\)](#)). They must not generate an invalid operation exception for signalling NaNs. For both profiles it is implementation defined if NaN values are propagated, or signaling NaNs are converted to quiet NaNs.

4.19.5 Floating Point Exceptions

HSAIL defines the five floating-point exceptions specified in IEEE/ANSI Standard 754-2008 (see [12.2 Hardware Exceptions \(p. 285\)](#)). It also provides a mechanism to control these exceptions by means of the DETECT and BREAK exception policies (see [12.3 Hardware Exception Policies \(p. 287\)](#)). The exception policies are specified when a kernel is finalized and cannot be changed at runtime. An implementation can choose to not generate hardware exceptions that correspond to HSAIL exceptions that are not enabled for the DETECT or BREAK exception policy since their effect is not observable in HSAIL.

- For the Full profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)), the DETECT and BREAK exception policies for the five floating-point exceptions must be supported.
- For the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)), it is not permitted to enable the DETECT or BREAK exception policies for any of the five floating-point exceptions. Therefore, no floating-point operation has to generate any exception as they have no observable effect in HSAIL.

4.20 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:

```
kernel &k1(kernarg_u32 %dynamic_size, kernarg_u32 %more_dynamic_size)
{
    group_u32 %known[2];
    group_u32 %more_known[4];
    group_u32 %dynamic[%dynamic_size]; // illegal: %dynamic_size not a constant value
    group_u32 %more_dynamic[%more_dynamic_size];
        // illegal: %more_dynamic_size not a constant value
    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,  known
  offset 8,  more_known
  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 arrays `dynamic` and `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, dynamic allocation of group memory uses these steps:

1. The application declares the HSAIL kernel with additional arguments, which are group segment offsets for the dynamically sized group memory. The kernel adds these offsets to the group segment base address returned by `groupbaseptr`, and uses the result to access the dynamically sized group memory.
2. The finalizer calculates the amount of group segment memory used by the kernel and the functions it calls directly or indirectly, and reports the size when the kernel is finalized.
3. The application computes the size and alignment of each of the dynamically allocated group segment variables that correspond to each of the additional kernel arguments. It uses this information to compute the group segment offset for each of the additional kernel arguments by starting at the group segment size reported by the finalizer for the kernel. The offsets must be rounded up to meet any alignment requirements.
4. The application dispatches the kernel using the group segment offsets it computed, and specifies the amount of group memory as the sum of the amount reported by the finalizer plus the amount required for the dynamic group memory.

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

```

kernel &k1(kernarg_u32 %dynamic_offset, kernarg_u32 %more_dynamic_offset)
{
    group_u32 %known[2];
    group_u32 %more_known[4];
    groupbaseptr_u32 $s0;
    ld_kernarg_u32 $s1, [%dynamic_offset];
    add_u32 $s1, $s0, $s1;
    ld_kernarg_u32 $s2, [%more_dynamic_offset];
    add_u32 $s2, $s0, $s2;
    st_group_f32 1.0f, [$s1 + 8];
    st_group_f32 2.0f, [$s2];
    //...
};

```

4.21 Kernarg Segment

The kernarg segment is used to hold kernel formal arguments as kernarg segment variables. Kernarg segment variables:

- Are always constant, because all work-items get the same values.
- Are read-only.
- Can only be declared in the list of kernel formal arguments.
- Cannot have initializers, because they get their values from the kernel's dispatch packet.

The memory layout of variables in the kernarg segment is required to be in the same order as the list of kernel formal arguments, starting at offset 0 from the kernel's kernarg segment base address, with no padding between variables except to honor the natural alignment requirements of the variable's type and any `align` qualifier. For information about the `align` qualifier, see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

The base address of the kernarg segment variables for the currently executing kernel dispatch can be obtained by the `kernargbaseptr` operation. The size of the kernel's kernarg segment variables is the size required for the kernarg segment variables and padding, rounded up to be a multiple of 16. The alignment of the base address of the kernel's kernarg segment variables is the larger of 16 and the maximum alignment of the kernel's kernarg segment variables.

HSA requires that the agent dispatching the kernel and the HSA component executing the dispatch have the same endian format.

When a kernel is dispatched, the dispatch packet that is added to the User Mode Queue must point to global segment memory that provides the values for the dispatch's kernarg segment. The global segment memory is required to be allocated using the runtime kernarg memory allocator specifying the HSA component with which the User Mode Queue is associated. It is allowed for a single allocation to be used for multiple dispatch packets on the same HSA component, either by sub-dividing it, or reusing it, provided the following restrictions are observed for the global segment memory pointed at by each dispatch packet:

- The memory must have the kernel's kernarg segment size and alignment.
- The memory must be initialized with the values of the kernel's formal arguments using the same memory layout as the kernel's kernarg segment, starting from offset 0.

- It must be ensured that the memory's initialized values are visible to a thread that performs a load acquire at system scope on the dispatch packet format field and it gets the DISPATCH value. For example, this could be achieved using a store release at system scope on the format field by the same thread that previously did the initialization.
- The memory must not be modified once the dispatch packet is enqueued until the dispatch has completed execution.

Therefore, the layout, size and alignment of the global segment memory used to pass values to the kernarg segment of a kernel can be statically determined, in a device independent manner, by examining the kernel's signature. An implementation is not permitted to require this memory to be any larger, or have greater alignment: for example, to hold additional implementation-specific data used during the execution of the kernel.

For example, the first kernel argument is stored at the base address, the second is stored at the base address + sizeof(first kernarg) aligned based on the type and optional align qualifier of the second argument, and so forth. Arrays are passed by value (see [4.10 Initializers and Array Declarations \(p. 74\)](#)).

It is implementation defined if the machine instructions generated to access the kernel's kernarg segment directly access this global segment memory, or if the values are used to initialize some other implementation-specific memory within the HSA component.

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 kernarg segment address)
```

Such addresses must not be used in store operations.

For more information, see [6.3 Load \(ld\) Operation \(p. 171\)](#) and [5.8 Copy \(Move\) Operations \(p. 117\)](#).

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 [4.16 Operands \(p. 86\)](#)).

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 [4.16 Operands \(p. 86\)](#).

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 7–1 Syntax for Integer Arithmetic Operations

Opcodes and Modifiers	Operands
<code>abs_sLength</code>	<code>dest, src0</code>
<code>add_TypeLength</code>	<code>dest, src0, src1</code>
<code>borrow_TypeLength</code>	<code>dest, src0, src1</code>
<code>carry_TypeLength</code>	<code>dest, src0, src1</code>
<code>div_TypeLength</code>	<code>dest, src0, src1</code>

Opcodes and Modifiers	Operands
<code>max_TypeLength</code>	<code>dest, src0, src1</code>
<code>min_TypeLength</code>	<code>dest, src0, src1</code>
<code>mul_TypeLength</code>	<code>dest, src0, src1</code>
<code>mulhi_TypeLength</code>	<code>dest, src0, src1</code>
<code>neg_sLength</code>	<code>dest, src0</code>
<code>rem_TypeLength</code>	<code>dest, src0, src1</code>
<code>sub_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))*Type:* s, u.*Length:* 32, 64.**Explanation of Operands (see [4.16 Operands \(p. 86\)](#))***dest:* Destination register.*src0, src1:* Sources. Can be a register, immediate value, or WAVESIZE.**Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))**

The only exceptions allowed are for `div` and `rem`, which are permitted to generate a divide by zero exception or an implementation-defined exception for a 0 divisor.

Table 7–2 Syntax for Packed Versions of Integer Arithmetic Operations

Opcodes and Modifiers	Operand
<code>abs_Control_sLength</code>	<code>dest, src0</code>
<code>add_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>max_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>min_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>mul_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>mulhi_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>neg_Control_sLength</code>	<code>dest, src0</code>
<code>sub_Control_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see [4.14 Packing Controls for Packed Data \(p. 81\)](#))

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 [4.13.2 Packed Data Types \(p. 80\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

src0, *src1*: Sources. Can be a register or immediate value.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.1 BRIG Syntax for Integer Arithmetic Operations \(p. 366\)](#).

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_length` is valid.

`abs(-231)` returns -2^{31} for 32-bit operands. `abs(-263)` 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.

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.

carry

The `carry` operation adds the two sources src_0 and src_1 . 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.

div

The `div` operation divides source src_0 by source src_1 and stores the quotient in destination $dest$.

The `div` operation follows the c99 model for signed division: the result 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 src_0 by source src_1 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(-231, -1)` returns 0 for `s32` types. `rem(-263, -1)` returns 0 for `s64` types.

max

The `max` operation computes the maximum of source src_0 and source src_1 and stores the result into the destination $dest$.

min

The `min` operation computes the minimum of source src_0 and source src_1 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.

`mul(-231, -1)` returns -2^{31} for 32-bit operands. `mul(-263, -1)` returns -2^{63} for 64-bit operands.

mulhi

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

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

For example: In the operation -1×1 , 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 $M \times N$, the top M bits of each of the N signed or unsigned products is placed in the packed $M \times N$ 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 \(p. 117\)](#).

neg

The `neg` operation computes $0 - source_{src0}$ and stores the result into the destination `dest`. There are no unsigned versions of `neg`, so only `neg_sLength` is valid.

`neg(-231)` returns -2^{31} for 32-bit operands. `neg(-263)` 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.

Examples of Regular (Nonpacked) Operations

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

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

borrow_s64 $d1, $d2, 23;

carry_s64 $d1, $d2, 23;

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;

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

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_s32x2 $d1, $d1;

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

max_pp_u8x4 $s1, $s0, $s3;
min_pp_u8x4 $s1, $s0, $s3;
mul_pp_u16x4 $d1, $d0, $d3;
mulhi_pp_u8x8 $d1, $d3, $d4;

neg_s_s8x4 $s1, $s2;
neg_s_s8x4 $s1, $s2;

sub_sp_u8x8 $d1, $d0, $d3;

```

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 \(p. 106\)](#).

5.3.1 Syntax

Table 7–3 Syntax for Integer Optimization Operation

Opcode and Modifiers	Operands
<code>mad_TypeLength</code>	<code>dest, src0, src1, src2</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

Type: s, u.

Length: 32, 64.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

src0, src1, src2: Sources. Can be a register, immediate value, or WAVELENGTH.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.2 BRIG Syntax for Integer Optimization Operation \(p. 366\)](#).

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: $((s0 * s1) + s2) \& ((1 << \text{length}) - 1)$.

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 \(p. 105\)](#).

5.4.1 Syntax

Table 7–4 Syntax for 24-Bit Integer Optimization Operations

Opcode and Modifiers	Operands
<code>mad24_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>mad24hi_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>mul24_TypeLength</code>	<code>dest, src0, src1</code>
<code>mul24hi_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see [Table 6-2 \(p. 79\)](#))*Type:* s, u*Length:* 32**Explanation of Operands (see [4.16 Operands \(p. 86\)](#))***dest:* Destination register.*src0, src1, src2:* Sources: Can be a register, immediate value, or WAVESIZE.**Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))**

No exceptions are allowed.

For BRIG syntax, see [18.7.1.3 BRIG Syntax for 24-Bit Integer Optimization Operations \(p. 367\)](#).

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 7–5 Syntax for Integer Shift Operations

Opcode and Modifiers	Operands
<code>shl_TypeLength</code>	<code>dest, src0, src1</code>
<code>shr_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

Type: s, u.

Length: For regular form: 32, 64; for packed form: 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, or 64x2.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

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

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.4 BRIG Syntax for Integer Shift Operations \(p. 367\)](#).

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 7–6 Syntax for Individual Bit Operations

Opcode and Modifiers	Operands
<code>and_TypeLength</code>	<code>dest, src0, src1</code>
<code>or_TypeLength</code>	<code>dest, src0, src1</code>
<code>xor_TypeLength</code>	<code>dest, src0, src1</code>
<code>not_TypeLength</code>	<code>dest, src0</code>
<code>popcount_u32_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

Type: b

Length: 1, 32, 64; popcount does not support b1.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest`: Destination register.

`src0, src1`: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 12 Exceptions (p. 285))
--

No exceptions are allowed.

For BRIG syntax, see [18.7.1.5 BRIG Syntax for Individual Bit Operations \(p. 367\)](#).

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 [Table 7-7 \(p. 112\)](#).

Table 7–7 Inputs and Results for popcount Operation

Input	Result
00000000	0
00ffffff	24
7fffffff	31
01ffffff	25
ffffffff	32
fffff0f00	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 7–8 Syntax for Bit String Operations

Opcode and Modifiers	Operands
<code>bitextract_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>bitinsert_TypeLength</code>	<code>dest, src0, src1, src2, src3</code>
<code>bitmask_TypeLength</code>	<code>dest, src0, src1</code>
<code>bitrev_TypeLength</code>	<code>dest, src0</code>
<code>bitselect_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>firstbit_u32_TypeLength</code>	<code>dest, src0</code>
<code>lastbit_u32_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers (see [Table 6-2 \(p. 79\)](#))

Type: b for bitmask, bitrev, and bitselect; s and u for bitextract, bitinsert, firstbit, and lastbit.

Length: 32, 64.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register. Must match the size of *Length*.

src0, *src1*, *src2*: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.6 BRIG Syntax for Bit String Operations \(p. 368\)](#).

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 `0xffffffff` (all 1's) returns 0
- `firstbit_u32_u32` of `0x7fffffff` (one 0 followed by 31 1's) returns 1
- `firstbit_u32_u32` of `0x01fffffff` (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 [Table 7–9 \(p. 116\)](#).

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 [Table 7–9 \(p. 116\)](#).

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 [Table 7–9 \(p. 116\)](#).

Table 7–9 Inputs and Results for `firstbit` and `lastbit` Operations

Input	Result for <code>firstbit</code>	Result for <code>lastbit</code>
00000000	-1	-1
00ffffff	8	0
7fffffff	1	0
01ffffff	7	0
ffffffff	0	0
fffff0f00	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.

If the Base profile has been specified then the 64-bit floating-point type (`f64`) is not supported (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

For the small machine model `sig64` is not supported, and for the large machine model `sig32` is not supported (see [2.9 Small and Large Machine Models \(p. 24\)](#)).

5.8.1 Syntax

Table 7–10 Syntax for Copy (Move) Operations

Opcode and Modifiers	Operands
<code>combine_v2_b64_b32</code>	<code>dest, (src0,src1)</code>
<code>combine_v4_b128_b32</code>	<code>dest, (src0,src1,src2,src3)</code>
<code>combine_v2_b128_b64</code>	<code>dest, (src0,src1)</code>
<code>expand_v2_b32_b64</code>	<code>(dest0,dest1), src0</code>
<code>expand_v4_b32_b128</code>	<code>(dest0,dest1,dest2,dest3), src0</code>
<code>expand_v2_b64_b128</code>	<code>(dest0,dest1), src0</code>
<code>lda_segment_uLength</code>	<code>dest, address</code>
<code>mov_moveType</code>	<code>dest, src0</code>

Explanation of Modifiers

`segment`: Optional segment: global, group, private, kernarg or readonly. If omitted, flat is used. See [2.8 Segments \(p. 13\)](#).

`Length`: 1, 32, 64, 128 (see [Table 6–2 \(p. 79\)](#)). For `lda` must match the address size (see [Table 4–3 \(p. 25\)](#)).

`moveType`: b1, b32, b64, b128, u32, u64, s32, s64, f16, f32, roimg, woimg, rwimg, samp. In addition, can be `f64` if the Base profile is not specified, `sig32` for small machine model, and `sig64` for large machine model. See [2.9 Small and Large Machine Models \(p. 24\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest, dest0, dest1, dest2, dest3`: Destination.

`src0, src1, src2, src3`: Sources. Can be a register, immediate value, or `WAVESIZE`.

`address`: An address expression. See [4.18 Address Expressions \(p. 88\)](#).

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.7 BRIG Syntax for Copy \(Move\) Operations \(p. 368\)](#).

5.8.2 Description

combine

Combines the values in the multiple source registers src_0, src_1 , and so forth to form a single result, which is stored in the destination register $dest$. src_0 becomes the least significant bits, src_1 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 src_0 into multiple parts and stores them in the multiple destination registers $dest_0, dest_1$, and so forth. The least significant bits of the value are stored in $dest_0$, the next least significant bits in $dest_1$, 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 \(p. 157\)](#). The size of $dest$ must match the address size of the segment. See [Table 4-3 \(p. 25\)](#).

The address of a kernel, function or label cannot be taken. Instead, the `ldk` operation can be used to get the address of a kernel descriptor, the `ldi` operation can be used to get the address of an indirect function descriptor, and the `sbr` operation can be used to achieve the equivalent of indirect branches.

The address of a spill segment variable cannot be taken.

The address of an arg segment variable cannot be taken: neither a function formal argument, nor arg block actual argument.

This operation can also be used to take the byte address of a kernel's formal arguments in the kernarg segment.

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

mov

Copies a value of type $moveType$ from source src_0 into the destination $dest$. It is required that, when moving a value that is of type `roimg`, `woimg`, `rwimg`, `samp`, `sig32` or `sig64`, $moveType$ should be specified accordingly (see [7.1.9 Using Image Operations \(p. 218\)](#) and [6.8 Notification \(signal\) Operations \(p. 187\)](#)). If $moveType$ is `f16`, an implementation is allowed to move only the number of bytes in the register representation of the `f16` type. See [4.19.1 Floating-Point Numbers \(p. 91\)](#).

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_u64 $d0, $d1;
lda_global_u64 $d1, [$d1 + 8];

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 and floating-point operations support packed data as does the cmp operation.

If the Base profile has been specified then the 64-bit packed floating-point type (2xf64) is not supported (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

See also:

- [5.2 Integer Arithmetic Operations \(p. 99\)](#)
- [5.11 Floating-Point Arithmetic Operations \(p. 129\)](#)
- [5.17 Compare \(cmp\) Operation \(p. 145\)](#)

See [Table 7–11 \(p. 120\)](#) and [Table 7–12 \(p. 120\)](#).

5.9.1 Syntax

[Table 7–11 Syntax for Shuffle and Interleave Operations](#)

Opcodes and Modifiers	Operands
<code>shuffle_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>unpacklo_TypeLength</code>	<code>dest, src0, src1</code>
<code>unpackhi_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see [4.13.2 Packed Data Types \(p. 80\)](#))

Type: s, u, f.

Length: 8x4, 8x8, 16x2, 16x4, 32x2

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

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 7–13 \(p. 123\)](#).

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.8 BRIG Syntax for Packed Data Operations \(p. 369\)](#).

[Table 7–12 Syntax for Pack and Unpack Operations](#)

Opcodes and Modifiers	Operands
<code>pack_destType destLength_srcType srcLength</code>	<code>dest, src0, src1, src2</code>
<code>unpack_destType destLength_srcType srcLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers
<i>destType</i> : s, u, f.
<i>srcType</i> : s, u, f.
<i>destLength</i> :
For pack, can be 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2. If the Base profile has been specified, 64x2 is not supported if <i>destType</i> is f.
For unpack, can be 32, 64, and, if <i>destType</i> is f, can be 16. If the Base profile has been specified, 64 is not supported if <i>destType</i> is f.
<i>srcLength</i> :
For pack, can be 32, 64, and, if <i>srcType</i> is f, can be 16. If the Base profile has been specified, 64 is not supported if <i>srcType</i> is f.
For unpack, can be 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2. If the Base profile has been specified, 64x2 is not supported if <i>srcType</i> is f.
See Table 6-2 (p. 79) , Table 6-3 (p. 80) and 16.2.1 Base Profile Requirements (p. 309) .

Explanation of Operands (see 4.16 Operands (p. 86))
<i>dest</i> : Destination register.
<i>src0</i> , <i>src1</i> , <i>src2</i> , <i>src3</i> : Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see Chapter 12 Exceptions (p. 285))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.8 BRIG Syntax for Packed Data Operations \(p. 369\)](#).

5.9.2 Description

`shuffle`

Selects half of the elements of *src0* based on controls in *src2* and copies them into the lower half of the *dest*. It then selects half of the elements of *src1* based on controls in *src2* and copies them into the upper half of the *dest*. *src2* has the fixed compound type of b32. See [5.9.3 Controls in src2 for shuffle Operation \(p. 123\)](#).

`unpacklo`

Copies and interleaves the lower half of the elements from each source into the destination. See [5.9.5 Examples of unpacklo and unpackhi Operations \(p. 126\)](#).

`unpackhi`

Copies and interleaves the upper half of the elements from each source into the destination. See [5.9.5 Examples of unpacklo and unpackhi Operations \(p. 126\)](#).

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 [4.16 Operands \(p. 86\)](#). 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, if the packed destination type's element size is 8 or 16 then the source compound type size must be 32, otherwise it must be the same as 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

unpack

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

src1 has the fixed compound type of `u32`. It specifies the index of the element to unpack.

If the element count is 2 (that is, the *Length* ends with `x2`), the index is specified in the least significant bit of *src1*.

If the element count is 4 (that is, the *Length* ends with `x4`), the index is specified in the least significant 2 bits of *src1*.

If the element count is 8 (that is, the *Length* ends with `x8`), the index is specified in the least significant 3 bits of *src1*.

If the element count is 16 (that is, the *Length* ends with `x16`), the index is specified in the least significant 4 bits of *src1*.

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

src0 has the compound type `srcType srcLength`.

See [4.16 Operands \(p. 86\)](#). 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, if the packed source type's element size is 8 or 16 then the destination compound type size must be 32, otherwise it must be the same as 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

5.9.3 Controls in *src2* for shuffle Operation

src2 of type `b32` 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 7–13 Bit Selectors for shuffle Operation

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

src2 Bits for Packed Data Types s8x8 and u8x8	Copied to
2-0 selects one of eight bytes from <i>src0</i>	<i>dest</i> bits 7-0
5-3 selects one of eight bytes from <i>src0</i>	<i>dest</i> bits 15-8
8-6 selects one of eight bytes from <i>src0</i>	<i>dest</i> bits 23-16
11-9 selects one of eight bytes from <i>src0</i>	<i>dest</i> bits 31-24
14-12 selects one of eight bytes from <i>src1</i>	<i>dest</i> bits 39-32
17-15 selects one of eight bytes from <i>src1</i>	<i>dest</i> bits 47-40
20-18 selects one of eight bytes from <i>src1</i>	<i>dest</i> bits 55-48
23-21 selects one of eight bytes from <i>src1</i>	<i>dest</i> bits 63-56

src2 Bits for Packed Data Types s16x2, u16x2, and f16x2	Copied to
0 selects one of two 16-bit values from <i>src0</i>	<i>dest</i> bits 15-0
1 selects one of two 16-bit values from <i>src1</i>	<i>dest</i> 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 <i>src0</i>	<i>dest</i> bits 15-0
3-2 selects one of four 16-bit values from <i>src1</i>	<i>dest</i> bits 31-16
5-4 selects one of four 16-bit values from <i>src0</i>	<i>dest</i> bits 47-32
7-6 selects one of four 16-bit values from <i>src1</i>	<i>dest</i> bits 63-48

src2 Bits for Packed Data Type f32x2	Copied to
0 selects one of two 32-bit values from <i>src0</i>	<i>dest</i> bits 31-0
1 selects one of two 32-bit values from <i>src1</i>	<i>dest</i> 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.

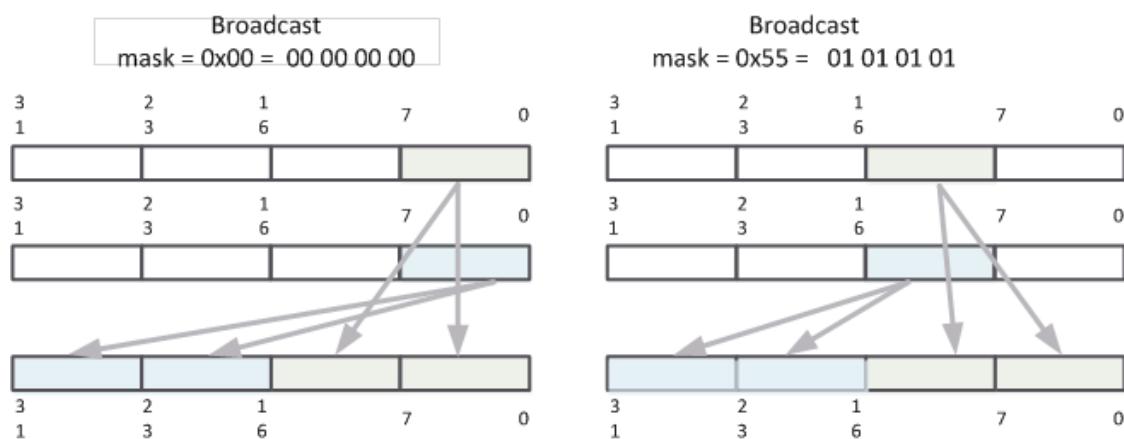
Broadcast the most significant data element into the destination:

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

src2 is the constant 11 11 11 11 in bits.

See the figure below.

Figure 7–1 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.

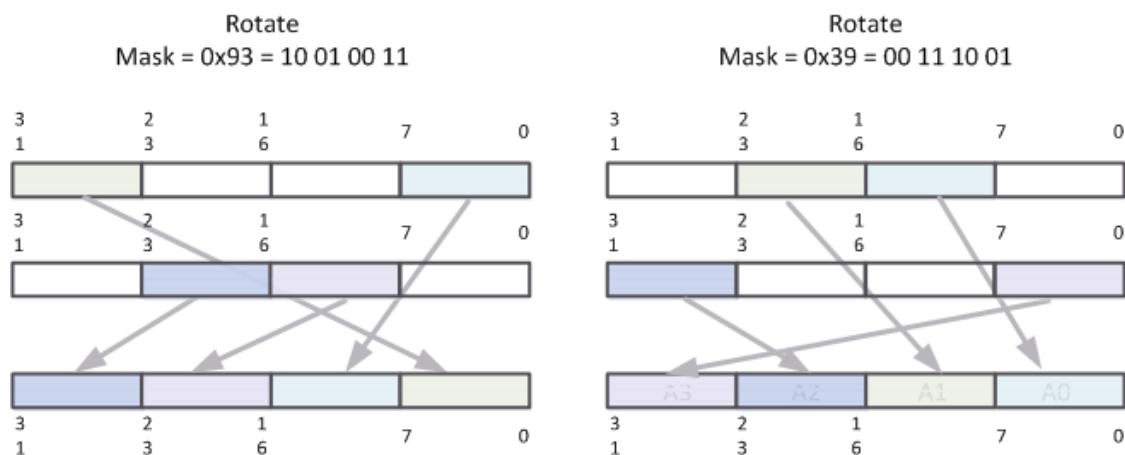
Rotate

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.

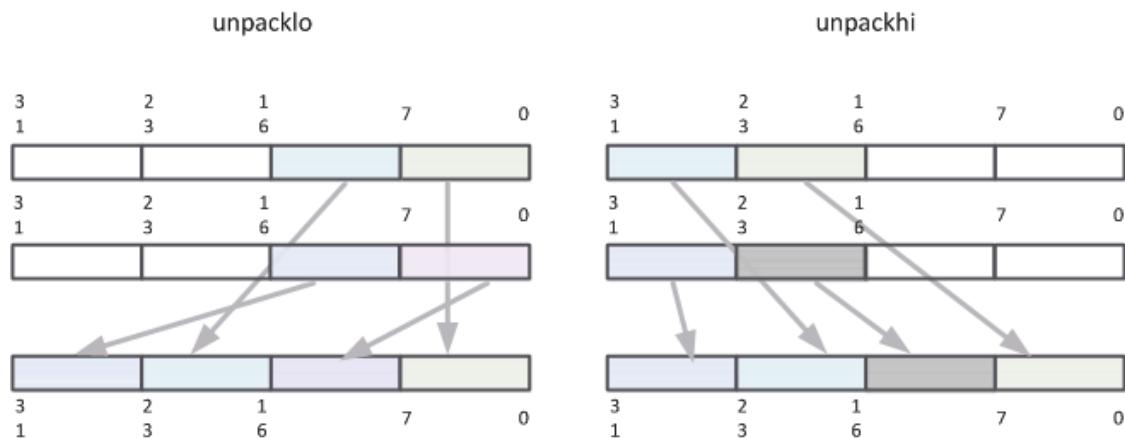
Figure 7–2 Example of Rotate



5.9.5 Examples of unpacklo and unpackhi Operations

See the figure below.

Figure 7–3 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, $s1, $s2, 1;
pack_f32x4_f32 $q1, $q2, $s2, 3;
pack_u32x2_u32 $d1, $d2, $s1, 2;
pack_s64x2_s64 $q1, $q1, $d1, $s1;

// 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;

```

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 7-14 Syntax for Bit Conditional Move (cmov) Operation

Opcode and Modifiers	Operands
<code>cmov_TypeLength</code>	<code>dest, src0, src1, src2</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

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 (see [4.16 Operands \(p. 86\)](#))

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 uLength; if the *Length* is 64 bits, then *src0* must be a d register or constant value of type uLength; if the *Length* is 128 bits, then *src0* must be a q register or constant value of type uLength. 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 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.9 BRIG Syntax for Bit Conditional Move \(cmov\) Operation \(p. 369\)](#).

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

These operations perform floating-point arithmetic and follow the IEEE/ANSI Standard 754-2008. However, there are some important differences. See [4.19 Floating Point \(p. 90\)](#).

5.11.1 Syntax

Table 7-15 Syntax for Floating-Point Arithmetic Operations

Opcode and Modifiers	Operands
<code>add_ftz_round_TypeLength</code>	<code>dest, src0, src1</code>
<code>ceil_ftz_TypeLength</code>	<code>dest, src0</code>
<code>div_ftz_round_TypeLength</code>	<code>dest, src0, src1</code>
<code>floor_ftz_TypeLength</code>	<code>dest, src0</code>
<code>fma_ftz_round_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>fract_ftz_TypeLength</code>	<code>dest, src0</code>
<code>max_ftz_TypeLength</code>	<code>dest, src0, src1</code>
<code>min_ftz_TypeLength</code>	<code>dest, src0, src1</code>
<code>mul_ftz_round_TypeLength</code>	<code>dest, src0, src1</code>
<code>rint_ftz_TypeLength</code>	<code>dest, src0</code>
<code>sqrt_ftz_round_TypeLength</code>	<code>dest, src0</code>
<code>sub_ftz_round_TypeLength</code>	<code>dest, src0, src1</code>
<code>trunc_ftz_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers

`ftz`: Required if the Base profile has been specified, otherwise optional. If specified, subnormal source values and tiny result values are flushed to zero. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

`round`: Optional rounding mode. If the Base profile has been specified, only `near`; otherwise `up`, `down`, `zero`, or `near`. See [4.19.2 Rounding \(p. 92\)](#).

`Type`: f. See [Table 6-2 \(p. 79\)](#).

`Length`: 16, 32, and, if the Base profile has not been specified, 64. See [Table 6-2 \(p. 79\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest`: Destination register.

`src0, src1, src2`: Sources. Can be a register or immediate value.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Floating-point exceptions are allowed.

Table 7–16 Syntax for Packed Versions of Floating-Point Arithmetic Operations

Opcode and Modifiers	Operands
<code>add_ftz_round_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>ceil_ftz_Control_TypeLength</code>	<code>dest, src0</code>
<code>div_ftz_round_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>floor_ftz_Control_TypeLength</code>	<code>dest, src0</code>
<code>fract_ftz_Control_TypeLength</code>	<code>dest, src0</code>
<code>max_ftz_Control_TypeLength</code>	<code>dest, src0</code>
<code>min_ftz_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>mul_ftz_round_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>rint_ftz_Control_TypeLength</code>	<code>dest, src0</code>
<code>sqrt_ftz_round_Control_TypeLength</code>	<code>dest, src0</code>
<code>sub_ftz_round_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>trunc_ftz_Control_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers

`ftz`: Required if the Base profile has been specified, otherwise optional. If specified, subnormal source values and tiny result values are flushed to zero. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

`round`: Optional rounding mode. If the Base profile has been specified, only `near`; otherwise `up`, `down`, `zero`, or `near`. See [4.19.2 Rounding \(p. 92\)](#).

`Control` for `ceil`, `floor`, `fract`, `rint`, `sqrt`, and `trunc`: `p` or `s`.

`Control` for `add`, `div`, `max`, `min`, `mul`, and `sub`: `pp`, `ps`, `sp`, or `ss`.

See [4.14 Packing Controls for Packed Data \(p. 81\)](#).

`TypeLength`: `f16x2`, `f16x4`, `f16x8`, `f32x2`, `f32x4`, and, if the Base profile has not been specified, `f64x2`. See [4.13.2 Packed Data Types \(p. 80\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest`: Destination register.

`src0, src1`: Sources. Can be a register or immediate value.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.10 BRIG Syntax for Floating-Point Arithmetic Operations \(p. 369\)](#).

5.11.2 Description

add

Performs the IEEE/ANSI Standard 754-2008 standard floating-point add.

ceil

Rounds the floating-point source src_0 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.

div

Computes source src_0 divided by source src_1 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 16 Profiles \(p. 307\)](#).

floor

Rounds the floating-point source src_0 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 $src_0 * src_1 + src_2$ 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 src_0 .

fract_f16 returns $\text{min_f16}(x - \text{floor}(x))$, *implementation-defined*.

fract_f32 returns $\text{min_f32}(x - \text{floor}(x))$, $0x1.\text{fffffep-1f}$.

fract_f64 returns $\text{min_f64}(x - \text{floor}(x))$, $0x1.\text{fffffffffffffp-1}$.

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, 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 $0x1.\text{ffcp-1}$ 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

max

Computes the maximum of source src_0 and source src_1 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 [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).

min

Computes the minimum of source src_0 and source src_1 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 [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).

mul

Multiplies source src_0 by source src_1 (following IEEE/ANSI Standard 754-2008 rules) and stores the result in the destination $dest$.

rint

Rounds the floating-point source src_0 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 src_1 from source src_0 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 src_0 .

If src_0 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 16 Profiles \(p. 307\)](#).

trunc

Rounds the floating-point source src_0 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

```
add_f32 $s3,$s2,$s1;
add_f64 $d3,$d2,$d1;
div_f32 $s3,1.0f,$s1;
div_f64 $d3,1.0,$d0;
fma_f32 $s3,1.0f,$s1,23.0f;
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;
sub_f32 $s3,1.0f,$s1;
sub_f64 $d3,1.0,$d0;
fract_f32 $s0, 3.2f;
```

Examples of Packed Operations

```
add_pp_f16x2 $s1, $s0, $s3;
sub_pp_f16x2 $s1, $s0, $s3;
```

5.12 Floating-Point Bit Operations

These operations are performed as floating-point bit operations and follow the IEEE/ANSI Standard 754-2008. See [4.19 Floating Point \(p. 90\)](#).

Since they are bit operations:

- They do not generate any exceptions, including underflow or inexact, nor invalid operation if any of their inputs are signaling NaNs.
- They do not convert signaling NaNs to quiet NaNs.
- The `ftz` modifier is not supported and they do not flush subnormal values to 0.0.
- The rounding modifier is not supported and no rounding is performed.

5.12.1 Syntax

Table 7–17 Syntax for Floating-Point Bit Operations

Opcode and Modifiers	Operands
<code>abs_TypeLength</code>	<code>dest, src0</code>
<code>class_b1_TypeLength</code>	<code>dest, src0, cond</code>
<code>copysign_TypeLength</code>	<code>dest, src0, src1</code>
<code>neg_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers (see [4.16 Operands \(p. 86\)](#) (see [Table 6-2 \(p. 79\)](#))*Type:* f. See [Table 6-2 \(p. 79\)](#).*Length:* 16, 32, and, if the Base profile has not been specified, 64. See [Table 6-2 \(p. 79\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).**Explanation of Operands (see [4.16 Operands \(p. 86\)](#))***dest:* Destination register.*src0, src1:* Sources. Can be a register or immediate value.*cond:* Source bit set specifying the conditions being tested. Must be a register or immediate value of compound type u32. See [Table 7-18 \(p. 134\)](#).

Table 7-18 Class Operation Source Operand Condition 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 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

Table 7-19 Syntax for Packed Versions of Floating-Point Bit Operations

Opcode and Modifiers	Operands
<code>abs_Control_TypeLength</code>	<code>dest, src0</code>
<code>copysign_Control_TypeLength</code>	<code>dest, src0, src1</code>
<code>neg_Control_TypeLength</code>	<code>dest, src0</code>

Explanation of Modifiers

Control for *abs*, and *neg*: *p* or *s*.

Control for *copysign*: *pp*, *ps*, *sp*, or *ss*.

See [4.14 Packing Controls for Packed Data \(p. 81\)](#).

TypeLength: *f16x2*, *f16x4*, *f16x8*, *f32x2*, *f32x4*, and, if the Base profile has not been specified, *f64x2*. See [4.13.2 Packed Data Types \(p. 80\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

src0, *src1*: Sources. Can be a register or immediate value.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.11 BRIG Syntax for Floating-Point Bit Operations \(p. 370\)](#).

5.12.2 Description

abs

Copies a floating-point operand *src0* to the destination *dest*, setting the sign bit to 0 (positive). No rounding is performed.

class

Tests the properties of a floating-point number in source *src0*, storing a 1 in the destination *dest* if any of the conditions specified in *cond* are true. If all properties are false, *dest* is set to 0. *dest* must be a control (c) register.

cond is interpreted using the values of [Table 7-18 \(p. 134\)](#) which can be combined using bitwise OR. All other bits are ignored. Thus, the following code will set the register *c1* to 1 if *\$s1* is either a signaling or quiet NaN:

```
class_b1_f32 $c1, $s1, 3;
```

copysign

Copies a floating-point operand *src0* to the destination *dest*, setting the sign bit to the sign bit of *src1*.

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.

Examples

```
abs_f32 $s1,$s2;
abs_f64 $d1,$d2;
class_b1_f32 $c1, $s1, 3;
class_b1_f32 $c1, $s1, $s2;
class_b1_f64 $c1, $d1, $s2;
class_b1_f64 $c1, $d1, 3;
copysign_f32 $s3,$s2,$s1;
copysign_f64 $d3,$d2,$d1;
neg_f32 $s3,1.0f;
neg_f64 $d3,1.0;
```

Examples of Packed Operations

```
abs_p_f16x2 $s1, $s2;
abs_p_f32x2 $d1, $d1;
neg_p_f16x2 $s1, $s2;
add_pp_f16x2 $s1, $s0, $s3;
```

5.13 Native Floating-Point Operations

The floating-point native functions operations provide fast approximate implementation values. They are expected to take advantage of hardware acceleration and are intended to be used where speed is preferred over accuracy.

For example, they can be used in device-specific libraries which know the accuracy of the native operations on that device. They can also be used in code that first performs tests to ensure they meet the accuracy requirements for every value in the range required by the algorithm.

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 source operand values are flushed to zero, whether or not tiny result values are flushed to zero, if NaN payloads are preserved (regardless of the profile specified), or if exceptions are generated (including those resulting from signaling NaNs).

See [4.19 Floating Point \(p. 90\)](#).

5.13.1 Syntax

Table 7–20 Syntax for Native Floating-Point Operations

Opcode and Modifiers	Operands
<code>ncos_f32</code>	<code>dest, src</code>
<code>nexp2_f32</code>	<code>dest, src</code>
<code>nfma_TypeLength</code>	<code>dest, src0, src1, src2</code>
<code>nlog2_f32</code>	<code>dest, src</code>

Opcode and Modifiers	Operands
<code>nrcp_TypeLength</code>	<code>dest, src</code>
<code>nrsqrt_TypeLength</code>	<code>dest, src</code>
<code>nsin_f32</code>	<code>dest, src</code>
<code>nsqrt_TypeLength</code>	<code>dest, src</code>

Explanation of Modifiers (see [Table 6-2 \(p. 79\)](#))

Type: f.

Length: 16, 32, and, if the Base profile has not been specified, 64. See [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

src, src0, src1, src2: Sources. Can be a register or immediate value.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Standard floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.12 BRIG Syntax for Native Floating-Point Operations \(p. 371\)](#).

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.

`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.

`nssqrt`

Computes the square root.

Examples

```
ncos_f32 $s1, $s0;
nexp2_f32 $s1, $s0;
nfma_f32 $s3, 1.0f, $s1, 23.0f;
nfma_f64 $d3, 1.0D, $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 7–21 Syntax for Multimedia Operations

Opcode	Operands
<code>bitalign_b32</code>	<code>dest, src0, src1, src2</code>
<code>bytealign_b32</code>	<code>dest, src0, src1, src2</code>
<code>lerp_u8x4</code>	<code>dest, src0, src1, src2</code>
<code>packcvt_u8x4_f32</code>	<code>dest, src0, src1, src2, src3</code>
<code>unpackcvt_f32_u8x4</code>	<code>dest, src0, src1</code>
<code>sad_u32_u32</code>	<code>dest, src0, src1, src2</code>
<code>sad_u32_u16x2</code>	<code>dest, src0, src1, src2</code>
<code>sad_u32_u8x4</code>	<code>dest, src0, src1, src2</code>
<code>sadhi_u16x2_u8x4</code>	<code>dest, src0, src1, src2</code>

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

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 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.13 BRIG Syntax for Multimedia Operations \(p. 371\)](#).

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) & 0xffffffff);
```

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, and that adding `src2` (which is treated as the type `u16x2`) also does 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 \(p. 143\)](#).

5.15.1 Syntax

Table 7–22 Syntax for Segment Checking (segmentp) Operation

Opcode and Modifiers	Operands
<code>segmentp_segment_nonull_b1_srcTypesrcLength</code>	<code>dest, src</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

`segment`: Can be `global`, `group` or `private`. See [2.8 Segments \(p. 13\)](#).

`nonull`: Optional. If present, indicates that the `src` operand will not be the `nullptr` address value for the segment. See the Description below.

`srcType`: `u`.

`srcLength`: 32, 64. The size of the source address. Must match the address size of flat addresses. See [Table 4–3 \(p. 25\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`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 4–3 \(p. 25\)](#).

Exceptions (see Chapter 12 Exceptions (p. 285))
--

No exceptions are allowed.

For BRIG syntax, see [18.7.1.14 BRIG Syntax for Segment Checking \(segmentp\) Operation \(p. 371\)](#).

5.15.2 Description

This operation sets the destination *dest* to true (1) if the flat address in source *src* is either the `nullptr` value for the flat address, or 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 [2.9 Small and Large Machine Models \(p. 24\)](#).

If it is known that the *src* operand can never have the flat address null pointer value, then the `nonnull` modifier can be specified. On some implementations this might be more efficient. The result is undefined if the `nonnull` modifier is specified and *src* is the `nullptr` value for the flat address. On some implementations this might result in incorrect values. See [17.10 Segment Address Conversion \(p. 314\)](#).

See [2.8.4 Memory Segment Access Rules \(p. 20\)](#).

Examples

```
segmentp_private_b1_u32 $c1, $s0;           // small machine model
segmentp_global_b1_u32 $c1, $s0;             // small machine model
segmentp_global_nonull_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 \(p. 142\)](#).

5.16.1 Syntax

Table 7–23 Syntax for Segment Conversion Operations

Opcodes and Modifiers	Operands
<code>ftos_segment_nonull_destTypedestLength_srcTypesrcLength</code>	<code>dest, src</code>
<code>stof_segment_nonull_destTypedestLength_srcTypesrcLength</code>	<code>dest, src</code>

Explanation of Modifiers
<i>segment</i> : global, group or private. See 2.8 Segments (p. 13) .
<i>nonnull</i> : Optional. If present, indicates that the <i>src</i> operand will not be the <code>nullptr</code> address value for the segment. See the Description below.
<i>destType</i> : u. See Table 6-2 (p. 79) .
<i>destLength</i> : 32, 64. The size of the destination address. For <code>ftos</code> , must be the address size of <i>segment</i> ; for <code>stof</code> , must be the flat address size. See Table 4-3 (p. 25) .
<i>srcType</i> : u. See Table 6-2 (p. 79) .
<i>srcLength</i> : 32, 64. The size of the source address. For <code>ftos</code> , must be the flat address size; for <code>stof</code> , must be the address size of <i>segment</i> . See Table 4-3 (p. 25) .

Explanation of Operands (see 4.16 Operands (p. 86))
<i>dest</i> : Destination register.
<i>src</i> : Source to be converted. Can be a register or immediate value.

Exceptions (see Chapter 12 Exceptions (p. 285))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.15 BRIG Syntax for Segment Conversion Operations \(p. 372\)](#).

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 *src* is the flat address `nullptr` value, then *dest* is set to the segment address `nullptr` value. 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 [2.9 Small and Large Machine Models \(p. 24\)](#).

If the source is not in the specified segment, the result is undefined. See [2.8.4 Memory Segment Access Rules \(p. 20\)](#).

If it is known that the *src* operand can never have the flat address null pointer value, then the `nonnull` modifier can be specified. On some implementations this might be more efficient. The result is undefined if the `nonnull` modifier is specified and *src* is the `nullptr` value for the flat address. On some implementations this might result in incorrect values. See [17.10 Segment Address Conversion \(p. 314\)](#).

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 [2.9 Small and Large Machine Models \(p. 24\)](#).

If it is known that the *src* operand can never have the segment address null pointer value, then the `nonnull` modifier can be specified. On some implementations this might be more efficient. The result is undefined if the `nonnull` modifier is specified and *src* is the `nullptr` value for the segment address. On some implementations this might result in incorrect values. See [17.10 Segment Address Conversion \(p. 314\)](#).

Examples

```
// large machine model conversions
stof_private_u64_u32 $d1, $s1;
stof_private_nonull_u64_u32 $d1, $s1;
ftos_group_u32_u64 $s1, $d2;
ftos_global_u64_u64 $d1, $d2;
ftos_global_nonull_u64_u64 $d1, $d2;

// small machine model conversions
stof_private_u32_u32 $s1, $s2;
stof_private_nonull_u32_u32 $s1, $s2;
ftos_group_u32_u32 $s1, $s2;
ftos_global_u32_u32 $s1, $s2;
ftos_global_nonull_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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

`cmp` also supports packed operands, returning one result per element.

Floating-point comparison is required to follow IEEE/ANSI Standard 754-2008. See [4.19 Floating Point \(p. 90\)](#).

If the source operands are floating-point, and one or more of them is a signaling NaN, then an invalid operation exception must be generated. Additionally, if the operation is a signalling comparison form and one or more of the source operands is a quiet NaN, then an invalid operation exception must be generated. See [12.2 Hardware Exceptions \(p. 285\)](#).

The `ftz` modifier is supported if the source operand type is floating-point. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

See [Table 7-24 \(p. 146\)](#) and [Table 7-25 \(p. 146\)](#).

5.17.1 Syntax

Table 7–24 Syntax for Compare (cmp) Operation

Opcode and Modifiers	Operands
<code>cmp_op_ftz_destType destLength_srcType srcLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see Table 6–2 (p. 79))
<code>op</code> for bit types: <code>eq</code> and <code>ne</code> .
<code>op</code> for integer source types: <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> .
<code>op</code> for floating-point source types: <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> , <code>equ</code> , <code>neu</code> , <code>ltu</code> , <code>leu</code> , <code>gtu</code> , <code>geu</code> , <code>num</code> , <code>nan</code> and signaling <code>Nan</code> forms <code>sNaN</code> , <code>sne</code> , <code>sLT</code> , <code>sLE</code> , <code>sGT</code> , <code>sGE</code> , <code>sEQ</code> , <code>sNEU</code> , <code>sLTU</code> , <code>sLEU</code> , <code>sGTU</code> , <code>sGEU</code> , <code>sNUM</code> , <code>sNAN</code> .
<code>ftz</code> : Only valid for floating-point source types. Required if the Base profile has been specified, otherwise optional. If specified, subnormal source values are flushed to zero. See 4.19.3 Flush to Zero (ftz) (p. 92) .
<code>destType destLength</code> : Describes the destination.
<code>destType</code> : <code>u</code> , <code>s</code> , <code>f</code> ; <code>b</code> if <code>destLength</code> is 1.
<code>destLength</code> : 32, 64; 1 if source type is <code>b</code> ; 16 if source type is <code>f</code> . If the Base profile has been specified, 64 is not supported if <code>destType</code> is <code>f</code> . See 16.2.1 Base Profile Requirements (p. 309) .
<code>srcType srcLength</code> : Describes the two sources.
<code>srcType</code> : <code>b</code> , <code>u</code> , <code>s</code> , <code>f</code> .
<code>srcLength</code> : 32, 64; 1 if source type is <code>b</code> ; 16 if source type is <code>f</code> . If the Base profile has been specified, 64 is not supported if <code>srcType</code> is <code>f</code> . See 16.2.1 Base Profile Requirements (p. 309) .

Explanation of Operands (see 4.16 Operands (p. 86))
<code>dest</code> : Destination register.
<code>src0, src1</code> : Sources. Each source can be a register, immediate value, or <code>WAVESIZE</code> .

Exceptions (see Chapter 12 Exceptions (p. 285))
<code>sNaN</code> floating-point numbers generate the invalid operation exception. The <code>s</code> comparison forms also generate the invalid operation exception for <code>qNaN</code> floating-point numbers.

Table 7–25 Syntax for Packed Version of Compare (cmp) Operation

Opcode and Modifiers	Operands
<code>cmp_op_ftz_pp_uLength_TypeLength</code>	<code>dest, src0, src1</code>

Explanation of Modifiers (see 4.13.2 Packed Data Types (p. 80))
--

<i>op</i> : See Explanation of Modifiers table above.

<i>ftz</i> : Only valid for floating-point source types. Required if the Base profile has been specified, otherwise optional. If specified, subnormal source values are flushed to zero. See 4.19.3 Flush to Zero (ftz) (p. 92) .

<i>Type</i> : s, u, f.

<i>Length</i> : 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2. If the Base profile has been specified, 64x2 is not supported if <i>Type</i> is f. See 16.2.1 Base Profile Requirements (p. 309) .
--

Explanation of Operands (see 4.16 Operands (p. 86))
--

<i>dest</i> : Destination register. This operation performs an element-by-element compare and puts the result in the destination. <i>dest</i> 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.
--

<i>src0, src1</i> : Sources. Must be a packed register or a constant value.

Exceptions (see Chapter 12 Exceptions (p. 285))
--

sNaN floating-point numbers generate the invalid operation exception. The s comparison forms also generate the invalid operation exception for qNaN floating-point numbers.

For BRIG syntax, see [18.7.1.16 BRIG Syntax for Compare \(cmp\) Operation \(p. 372\)](#).

5.17.2 Description

The table below shows the value written into the destination *dest*. For packed types, the value for the comparison of each element is written into the corresponding element in the destination *dest*.

Type of <i>dest</i>	True	False
f16, f32, f64	1.0	0.0
u8	0xff	0x00
u16	0xffff	0x0000
u32, s32	0xffffffff	0x00000000
u64, s64	0xffffffffffffffff	0x0000000000000000
b1	1	0

num

Numeric. Only supported for floating point source operand types. Returns true if both floating-point source operands are numeric values (not a NaN).

nan

Not A Number. Only supported for floating point source operand types. Returns true if either floating-point source operand is a NaN.

`eq, ne, lt, le, gt, ge`

Ordered comparisons. These correspond to *equal*, *not equal*, *less than*, *less than or equal*, *greater than* and *greater than or equal* respectively. All support both integer and floating point source operand types. Additionally, `eq` and `ne` support the `b1` bit source operand type. For floating-point source operands, if either is a NaN, then the result is false. Otherwise, returns the corresponding comparison performed on the source operands.

`equ, neu, ltu, leu, gtu, geu`

Unordered comparisons. There are unordered forms of all the ordered comparisons. For example, `leu` is the unordered form of `le`. Only supported for floating point source operand types. If either operand is a NaN, then the result is true. Otherwise, returns the same result as the corresponding ordered comparison.

`seq, sne,slt,sle,sgt,sge,sequ,sneu,slt,u,sleu,sgtu,sgeu,snan,snan`

Signalling comparisons. There are signalling forms of the ordered, unordered, `num` and `nan` comparisons. For example, `sle` is the signalling form of `le`. Only supported for floating point source operand types. Returns the same result as the corresponding non-signalling comparison, except that the invalid operation exception must also be generated if either source operand is a quiet NaN.

For the floating point comparisons see [Table 7–26 \(p. 148\)](#):

- The table gives a mapping from the HSAIL floating-point comparisons to the corresponding IEEE/ANSI Standard 754-2008 four mutually exclusive relations *less than* (LT), *equal* (EQ), *greater than* (GT) and *unordered* (UN).
- The HSAIL comparison is true if any of the IEEE/ANSI Standard 754-2008 relations are true.
- The sign of zero is ignored so +0.0 compares *equal* to -0.0.
- Infinite operands of the same sign compare as *equal*.
- Every NaN compares *unordered* with everything, including itself.
- The table also gives the IEEE/ANSI Standard 754-2008 equivalent operation name if available.

Table 7–26 Floating-Point Comparisons

HSAIL	IEEE/ANSI Standard 754-2008	
Comparison Operation	True Relations	Operation
<code>num</code>	EQ, LT, GT	<code>compareQuietOrdered</code>
<code>nan</code>	UN	<code>compareQuietUnordered</code>
<code>eq</code>	EQ	<code>compareQuietEqual</code>
<code>ne</code>	LT, GT	
<code>lt</code>	LT	<code>compareQuietLess</code>
<code>le</code>	EQ, LT	<code>compareQuietLessEqual</code>
<code>gt</code>	GT	<code>compareQuietGreater</code>
<code>ge</code>	EQ, GT	<code>compareQuietGreaterEqual</code>
<code>equ</code>	EQ, UN	
<code>neu</code>	LT, GT, UN	<code>compareQuietNotEqual</code>
<code>ltu</code>	LT, UN	<code>compareQuietLessUnordered</code>
<code>leu</code>	EQ, LT, UN	<code>compareQuietNotGreater</code>
<code>gtu</code>	GT, UN	<code>compareQuietGreaterUnordered</code>

HSAIL	IEEE/ANSI Standard 754-2008	
Comparison Operation	True Relations	Operation
geu	EQ, GT, UN	compareQuietNotLess
snum	EQ, LT, GT	
snan	UN	
seq	EQ	compareSignalingEqual
sne	LT, GT	
slt	LT	compareSignalingLess
sle	EQ, LT	compareSignalingLessEqual
sgt	GT	compareSignalingGreater
sge	EQ, GT	compareSignalingGreaterEqual
sequ	EQ, UN	
sneu	LT, GT, UN	compareSignalingNotEqual
sltu	LT, UN	compareSignalingLessUnordered
sleu	EQ, LT, UN	compareSignalingNotGreater
sgtu	GT, UN	compareSignalingGreaterUnordered
sgeu	EQ, GT, UN	compareSignalingNotLess

Examples

```

cmp_eq_b1_b1 $c1, $c2, 0;
cmp_eq_u32_b1 $s1, $c2, 0;
cmp_eq_s32_b1 $s1, $c2, 1;
cmp_eq_f32_b1 $s1, $c2, 1;

cmp_ne_b1_b1 $c1, $c2, 0;
cmp_ne_u32_b1 $s1, $c2, 0;
cmp_ne_s32_b1 $s1, $c2, 0;
cmp_ne_f32_b1 $s1, $c2, 1;

cmp_lt_b1_u32 $c1, $s2, 0;
cmp_lt_u32_s32 $s1, $s2, 0;
cmp_lt_s32_s32 $s1, $s2, 0;
cmp_lt_f32_f32 $s1, $s2, 0.0f;

cmp_gt_b1_u32 $c1, $s2, 0;
cmp_gt_u32_s32 $s1, $s2, 0;
cmp_gt_s32_s32 $s1, $s2, 0;
cmp_gt_f32_f32 $s1, $s2, 0.0f;

cmp_equ_b1_f32 $c1, $s2, 0.0f;
cmp_equ_b1_f64 $c1, $d1, $d2;

cmp_sltu_b1_f32 $c1, $s2, 0.0f;
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 (cvt) 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 specify different types and/or lengths for the destination and the source operands.

The source and destination operands are not allowed to have the same type and length. If the source operand is an integer type, then the destination type is not allowed to be an integer type with the same size. Use a `mov` operation instead because these cases involve no conversion.

If the source or destination is a floating-point type the conversion is required to follow IEEE/ANSI Standard 754-2008. See [4.19 Floating Point \(p. 90\)](#).

For register operands:

- If the source or destination operand type is `b1` then it must be a `c` register.
- If the source operand has an integer type less than 32 bits in size, then it must be an `s` register. In this case, the least significant source type length bits are used.
- If the destination operand has an integer type less than 32 bit is size, then it must be an `s` register. In this case, the conversion operations first transform the source to the destination type. The converted result is then zero-extended for `u` types, and sign-extended for `s` types, to 32 bits.

No packed formats are supported.

[Table 7-27 \(p. 150\)](#) shows how the first step of the conversion operation does the transformation. The table uses the notation defined in [Table 7-28 \(p. 151\)](#).

Table 7-27 Conversion Methods

	Source <code>b1</code>	Source <code>u8</code>	Source <code>s8</code>	Source <code>u16</code>	Source <code>s16</code>	Source <code>f16</code>	Source <code>u32</code>	Source <code>s32</code>	Source <code>f32</code>	Source <code>u64</code>	Source <code>s64</code>	Source <code>f64</code>
Destina tion <code>b1</code>	-	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest
Destina tion <code>u8</code>	zext	-	-	chop	chop	h2u	chop	chop	f2u	chop	chop	d2u
Destina tion <code>s8</code>	zext	-	-	chop	chop	h2s	chop	chop	f2s	chop	chop	d2s
Destina tion <code>u16</code>	zext	zext	sext	-	-	h2u	chop	chop	f2u	chop	chop	d2u
Destina tion <code>s16</code>	zext	zext	sext	-	-	h2s	chop	chop	f2s	chop	chop	d2s
Destina tion <code>f16</code>	u2h	u2h	s2h	u2h	s2h	-	u2h	s2h	f2h	u2h	s2h	d2h
Destina tion <code>u32</code>	zext	zext	sext	zext	sext	h2u	-	-	f2u	chop	chop	d2u
Destina tion <code>s32</code>	b2s	zext	sext	zext	sext	h2s	-	-	f2s	chop	chop	d2s

	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 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	b2s	zext	sext	zext	sext	h2s	zext	sext	f2s	-	-	d2s
Destination f64	u2d	u2d	s2d	u2d	s2d	h2d	u2d	s2d	f2d	u2d	s2d	-

Table 7–28 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.
b2s	If 0 then all zeros; else all ones.
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).
-	Not allowed.

5.18.2 Syntax

Table 7–29 Syntax for Conversion (cvt) Operation

Opcode and Modifiers	Operands
<code>cvt_ftz_round_destType destLength srcType srcLength</code>	<code>dest, src</code>

Explanation of Modifiers (see [Table 6–2 \(p. 79\)](#))

ftz: Only valid if *srcType* is floating-point. Required if the Base profile has been specified, otherwise optional. If specified, subnormal source values and tiny result values are flushed to zero. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

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, neari_sat, supi, sdowni, szeroi, sneari, supi_sat, sdowni_sat, szeroi_sat, and sneari_sat. However, the allowed values depend on the *destType*, *srcType*, and whether the Base profile has been specified. See [4.19.2 Rounding \(p. 92\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#). In some cases, *round* can be omitted, and defaults to near or zeroi as appropriate. See [5.18.3 Rules for Rounding for Conversions \(p. 152\)](#), [5.18.4 Description of Integer Rounding Modes \(p. 153\)](#), and [5.18.5 Description of Floating-Point Rounding Modes \(p. 155\)](#).

destType: b, u, s, f.

destLength: 1, 8, 16, 32, 64. 1 is only allowed for *destType* of b. 1 and 8 are not allowed for *destType* of f. If the Base profile has been specified, 64 is not supported if *destType* is f. See [16.2.1 Base Profile Requirements \(p. 309\)](#).

srcType: b, u, s, f.

srcLength: 1, 8, 16, 32, 64. 1 is only allowed for *srcType* of b. 1 and 8 are not allowed for *srcType* of f. If the Base profile has been specified, 64 is not supported if *srcType* is f. See [16.2.1 Base Profile Requirements \(p. 309\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

src: Source. Can be a register, immediate value, or (if *srcType* is an integer type) WAVESIZE.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.17 BRIG Syntax for Conversion \(cvt\) Operation \(p. 372\)](#).

5.18.3 Rules for Rounding for Conversions

Rounding for conversions follows the rules shown in [Table 7–30 \(p. 152\)](#).

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 rounding mode can be omitted, in which case it defaults to zeroi.

If the type of rounding is “none,” then no rounding mode must be specified.

Table 7–30 Rules for Rounding for Conversions

From	To	Type of rounding	Default rounding
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

From	To	Type of rounding	Default rounding
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. See [Table 7-31 \(p. 154\)](#).

The integer rounding mode can be omitted, in which case it defaults to zeroi. If the Base profile has been specified, only zeroi, zeroi_sat, szeroi and szeroi_sat are allowed.

If the source operand is a signaling NaN, an invalid operation exception must be generated. See [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).

The `ftz` modifier is supported. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

There are both regular and saturating integer rounding modes. For example, `upi_sat` is the saturating integer rounding mode that corresponds to the `upi` regular integer rounding mode. 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.

The regular integer rounding modes might execute faster than the saturating integer rounding modes.

For the saturating integer rounding modes:

- If the source is a NaN, then the result is 0.
- If the source is not a NaN, 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 $2^{\text{destLength}} - 1$, then $2^{\text{destLength}} - 1$ is stored.
 - For signed destination types:
 - If the rounded result is -inf or less than $-2^{\text{destLength}-1}$, then $-2^{\text{destLength}-1}$ is stored.
 - If the rounded result is +inf or greater than $2^{\text{destLength}-1} - 1$, then $2^{\text{destLength}-1} - 1$ is stored.

There are both non-signalling and signalling forms of the regular and saturating integer rounding modes. For example, `supi` is the signalling form of `upi`. They differ in whether they generate the inexact exception if the source value, after any flush to zero, is in range but not an integral value. The non-signalling forms do not generate an inexact exception and correspond to the IEEE/ANSI Standard 754-2008 inexact conversions. The signalling forms do generate an inexact exception and correspond to the IEEE/ANSI Standard 754-2008 exact conversions. If no exception policy is enabled for the inexact exception, then both forms behave the same way.

Table 7–31 Integer Rounding Modes

Regular Integer Rounding Modes		Saturating Integer Rounding Modes		Regular Integer Rounding Mode Description
Non-Signalling Form	Signalling Form	Non-Signalling Form	Signalling Form	
<code>upi</code>	<code>supi</code>	<code>upi_sat</code>	<code>supi_sat</code>	Rounds up to the nearest integer greater than or equal to the exact result.
<code>downi</code>	<code>sdowni</code>	<code>downi_sat</code>	<code>sdowni_sat</code>	Rounds down to the nearest integer less than or equal to the exact result.
<code>zeroi</code>	<code>szeroi</code>	<code>zeroi_sat</code>	<code>szeroi_sat</code>	Rounds to the nearest integer toward zero.
<code>neari</code>	<code>sneari</code>	<code>neari_sat</code>	<code>sneari_sat</code>	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_s32_f32 $s2, $s1; // sets $s2 = 2
cvt_downi_s32_f32 $s2, $s1; // sets $s2 = 1
cvt_zeroi_s32_f32 $s2, $s1; // sets $s2 = 1
cvt_neari_s32_f32 $s2, $s1; // sets $s2 = 2
```

If `$s1` has the value -1.6, then:

```
cvt_upi_s32_f32 $s2, $s1; // sets $s2 = -1
cvt_downi_s32_f32 $s2, $s1; // sets $s2 = -2
cvt_zeroi_s32_f32 $s2, $s1; // sets $s2 = -1
cvt_neari_s32_f32 $s2, $s1; // sets $s2 = -2
```

5.18.5 Description of Floating-Point Rounding Modes

The floating-point rounding modes are (see [4.19.2 Rounding \(p. 92\)](#)):

- **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, then only **near** is allowed. See [4.19.2 Rounding \(p. 92\)](#).

The **ftz** modifier is supported. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

If the source operand is a NaN, then the result must be a quiet NaN. The NaN payload is not preserved, because the types are different sizes. It is implementation-defined if the sign is preserved. If a signaling NaN, then an invalid operation exception must be generated. See [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).

Otherwise, the infinitely precise source value, after any flush to zero, is rounded to the destination type and stored in the destination operand. The exceptions generated include those produced by rounding. See [4.19.2 Rounding \(p. 92\)](#).

- 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, then only **near** is allowed. See [4.19.2 Rounding \(p. 92\)](#).

The **ftz** modifier is not supported. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

Otherwise, the infinitely precise source value is rounded to the destination type and stored in the destination operand. The exceptions generated include those produced by rounding. See [4.19.2 Rounding \(p. 92\)](#).

Floating-point rounding modes are invalid in all other cases.

Examples

```
cvt_f32_f64 $s1, $d1;  
cvt_upi_u32_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_sat_s8_f32 $s1, $s2;
```

Chapter 6

Memory Operations

This chapter describes the HSAIL memory operations.

6.1 Memory and Addressing

Memory operations transfer data between registers and memory and can define memory synchronization between work-items and other agents:

- The ordinary load and atomic load operations move contents from memory to a register.
- The ordinary store and atomic store operations move contents of a register into memory.
- The atomic read-modify-write memory operations update the contents of a memory location based on the original value of the memory location and the value in a register. Most read-modify-write operations have two forms: one that returns the original value of the memory location into a register; and one that does not return a value and so has no destination operand.
- The memory fence operation defines the memory synchronization between work-items and other agents.

A flat memory, global segment, readonly segment, or kernarg segment address is a 32- or 64-bit value, depending on the machine model. A group segment, private segment, spill segment, or arg segment address is always 32 bits regardless of machine model. See [2.9 Small and Large Machine Models \(p. 24\)](#). 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 \(p. 19\)](#).

6.1.1 How Addresses Are Formed

The format of an address expression is described in [4.18 Address Expressions \(p. 88\)](#).

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 4–3 \(p. 25\)](#).

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. 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 [4.8.1 Integer Constants \(p. 67\)](#).

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 \(p. 19\)](#).

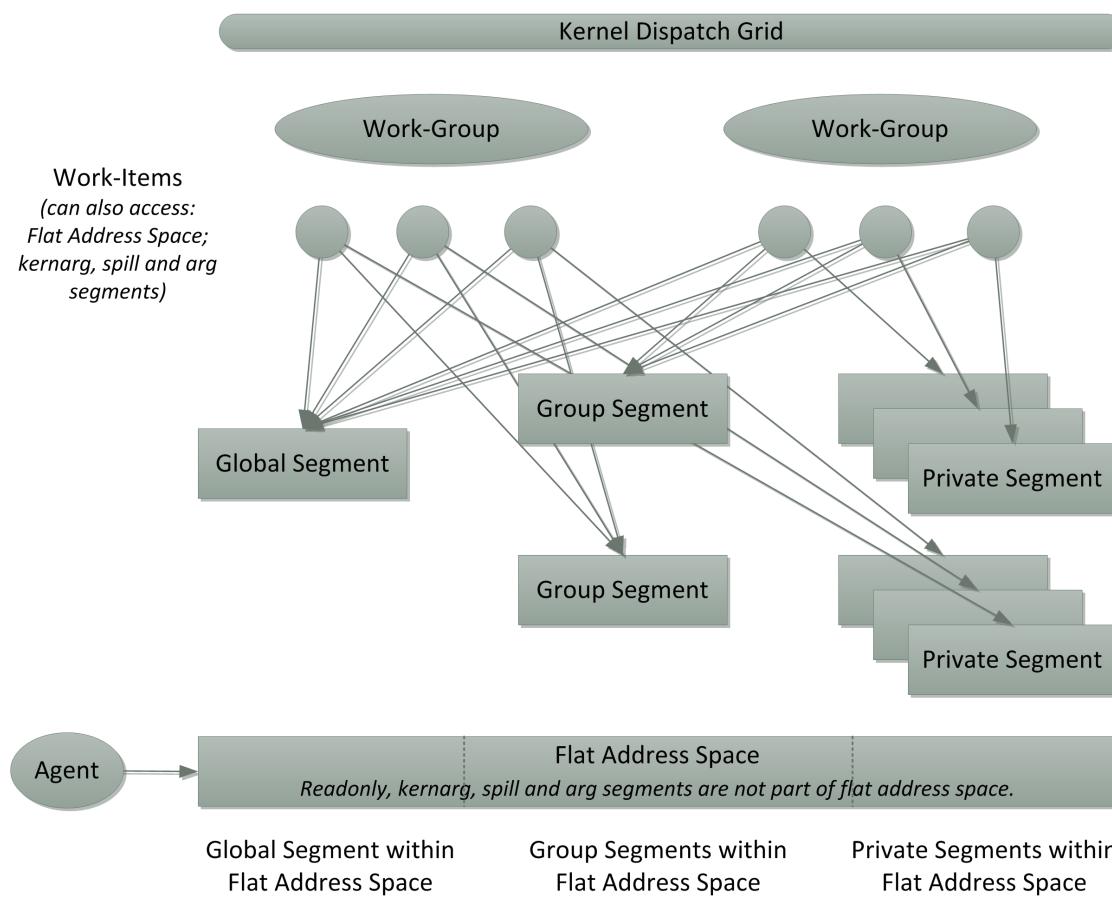
If the resulting effective address value is outside the memory segment specified by the operation, or is a flat address that is outside any segment, the result of the memory segment operation is undefined.

For more information, see [4.18 Address Expressions \(p. 88\)](#).

6.1.2 Memory Hierarchy

[Figure 8–1 \(p. 159\)](#) shows an example of the memory used by an agent executing a kernel dispatch grid.

Figure 8–1 Memory Hierarchy



The addresses used to access memory do not need to be naturally aligned to a multiple of the access 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.

The readonly, kernarg, spill and arg segments are not part of the flat address space.

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 Memory Model

This section maps the HSAIL operations and modifiers to the HSA Memory Model formally defined in the *HSA Platform System Architecture Specification* chapter 3 *HSA Memory Consistency Model*. It also provides an overall informal definition of the memory model.

The ordinary memory operations do not specify any memory synchronization. In contrast the atomic memory operations can optionally specify memory synchronization. Memory synchronization can be used to ensure that different work-items or agents never access the same memory location at the same time.

Ordinary memory operations support unaligned access and do not guarantee atomicity of access. Atomic memory operations are only supported for naturally aligned accesses to 32 bits, and for the large machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)) naturally aligned accesses to 64 bits, and do guarantee atomicity of access.

If two memory operations by different work-items or agents can access a single memory location at the same time, one or both of those accesses is an ordinary memory operation, and one or both modifies the memory location, then the access is defined as a data race. Conversely, if the accesses cannot happen at the same time, or if both accesses to the memory location are reads, or if both accesses are atomic, whether or not the atomic operation specifies memory synchronization, then there is no data race. A program with a data race is undefined, including the results of operations done before the operation that caused the data race. A data race free program must execute as if all work-items and agents observe the same modification order for every memory location accessed.

The atomic memory operations specify both the segment they access and a scope. The scope specifies a set of work-items and agents. Each atomic memory access belongs to a specific segment scope instance based on the segment being accessed, the scope specified by the operation and the scope instance to which the work-item or agent executing it belongs.

There are relaxed atomic operations and synchronizing atomic memory operations:

- The value of a memory location made visible by a relaxed atomic memory operation must become visible to all work-items and agents that belong to the segment scope instance in finite time. However, the value may be made visible to different work-items and agents at different times. So different work-items and agents may observe different relative orders between different memory locations, but they will always see an order of values that is consistent with the single modification order of the memory location.
- The value of a memory location made visible by a synchronizing atomic memory operation is made visible to all work-items and agents that belong to the same scope instance at the same time (sequentially consistent). So different work-items and agents that are members of the same scope instance, observe the same relative orders between different memory locations that have all been updated by synchronizing atomic memory operations belonging to the same segment scope instance.

Memory operations are the load, store, atomic and memory fence operations defined in this chapter. Read/write image operations are the `rdimage`, `ldimage` and `stimage` operations defined in [Chapter 7 Image Operations \(p. 197\)](#) which use a separate image memory model defined in [7.1.10 Image Memory Model \(p. 220\)](#).

6.2.1 Memory Order

The memory synchronization of an operation is specified by the memory order modifier which can have the following values:

- `scacq` specifies the operation is a sequentially consistent acquire memory operation. An acquire memory operation is a synchronizing memory operation. It creates a downward fence. This means that non-synchronizing memory operations by the same work-item can be moved (by the implementation) down after the operation, but no memory operation by the same work-item can be moved before the operation. It is allowed on atomic load, atomic read-modify-write and memory fence operations.
 - For acquire atomic memory operations, whether atomic load or atomic read-modify-write, the value of the location accessed will be no earlier in that location's modification order than those made visible by release operations executed before the acquire memory operation by any work-item or agent. In addition, all subsequent memory operations by the same work-item, whether ordinary or atomic, to any location, will access values no earlier in the location's modification order than those made visible by release operations, by any work-item or agent, executed before the acquire memory operation.
 - For acquire memory fence operations, all subsequent memory operations by the same work-item, whether ordinary or atomic, to any location, will access values no earlier in the location's modification order than those made visible by release operations, by any work-item or agent, executed before the acquire memory fence operation. In addition, any previous atomic load by the same work-item, whether synchronizing or relaxed, will access the value of the location no later in that location's modification order than those made visible by release operations executed before the acquire memory fence operation by any work-item or agent. That is, ordinary memory (but not atomic memory, neither synchronizing nor relaxed) operations by the same work-item can be moved down after the memory fence operation but no memory operations by the same work-item can be moved up before the acquire fence.
- `screl` specifies the operation is a sequentially consistent release memory operation. A release memory operation is a synchronizing memory operation. It creates an upward fence. It is allowed on atomic store, atomic read-modify-write and memory fence operations.
 - For release atomic memory operations, whether atomic store or atomic read-modify-write, all previous memory update operations by the same work-item, whether atomic or ordinary, to any location, will be made visible to other work-items and agents before the value of the location updated by the release memory operation is made visible. That is, memory operations by the same work-item can be moved up before the operation but no memory operations by the same work-item can be moved down after the operation.
 - For release memory fence operations, all previous memory update operations by the same work-item, whether atomic or ordinary, to any location, will be made visible to other work-items and agents before the values of any locations updated by atomic operations by the same work-item, whether synchronizing or relaxed, are made visible. In addition, any subsequent atomic memory operation by the same work-item, whether synchronizing or relaxed, will be made visible to other work-items and

agents after the memory fence operation. That is, ordinary memory (but not atomic memory, neither synchronizing nor relaxed) operations by the same work-item can be moved up before the memory fence operation but no memory operations by the same work-item can be moved down after the release fence.

- `scar` specifies the operation is both a sequentially consistent acquire and sequentially consistent release memory operation. It creates a full fence (basically a two-way fence operation). That is, memory operations by the same work-item can neither be moved up before or down after the operation. It is allowed on atomic read-modify-write and memory fence operations. It has the properties of both an acquire and release operation, and so is a synchronizing memory operation.
- `rlx` specifies the operation is a relaxed memory operation. A relaxed memory operation is not a synchronizing memory operation. This means that non-synchronizing memory operations to other memory locations by the same work-item can be moved (by the implementation) both down after and up before the operation. It is allowed on atomic load, atomic store and atomic read-modify-write memory operations.
 - For atomic store and read-modify-write operations, relaxed atomic updates will be made visible to other work-items and agents in finite time, but can be made visible to different work-items and agent threads at different times.
 - For atomic load memory operations, the value accessed must be no earlier in the location's modification order than required by the preceding acquire memory operation of the same work-item. If subsequent atomic memory operations, whether synchronizing or non-synchronizing, by other work-items or agents, update the same location after the preceding acquire memory operation, then one of those values may be returned. Additionally, such subsequent values must be returned by an atomic relaxed load in finite time from when they were performed.
 - For atomic read-modify-write the rules for both atomic load and atomic store memory operations apply. In addition, the rules for all read-modify write operations described below apply.

Consequently:

- If a work-item or agent thread accesses the value produced by a relaxed memory operation, any following memory operation, whether ordinary or atomic, by the same work-item or agent thread, must not access an earlier value in the location's modification order. This is a restatement of the single modification order requirement for all memory locations.
- In addition, if subsequent relaxed memory operations, by the same work-item or agent thread, accesses the same location after the location has been updated by atomic operations, whether synchronizing or non-synchronizing, by other work-items or agents, then one of the subsequent relaxed memory operations will see a value later in the modification order of the location. This is implied by the requirement that relaxed memory operations become visible in finite time.
- However, there is no guarantee about the values seen for other memory locations. This is a consequence of a relaxed memory operation not being a synchronizing memory operation.
- Nor is there a guarantee about the values seen for the location by other work-items or agent threads for this memory location. The value of a relaxed atomic update must become visible to other work-items and agents in finite time, but can happen at different times for different work-items or agent threads.

An ordinary update memory operation is made visible to other work-items and agents by a release memory operation executed by the same work-item, or a release performed by the packet processor of the HSA component on which it executes. Note that the value updated by an ordinary store may become accessible by other work-items or agents at any time before it is made visible, but it is a data race if any other work-item or agent accesses the memory location before it is made visible. Therefore, a finalizer or hardware can optimize ordinary memory operations provided it ensures the final value of memory locations modified by them is made visible by the associated release memory operation. Such optimizations include reordering loads and stores provided data dependence requirements are met, and common subexpression elimination to avoid multiple loads and stores of the same location.

It is not a data race if a read-modify-write atomic operation consumes an ordinary store of the same work-item provided no other work-item can access or update the location. It is not a data race for a read-modify-write atomic operation to consume the value produced by a relaxed atomic operation of any work-item or agent since the value was updated by an atomic operation. However, since relaxed atomic operations can become visible to different work-items and agents at different times, and it is required that each work-item and agent see the same modification order for each location, the value produced by a relaxed atomic operation can only be used as the source of an atomic read-modify-write operation if it will precede the value produced by the read-modify-write atomic operation in the location modification order. One way an implementation can achieve this is by ensuring that the value produced by a relaxed atomic operation is made visible to all work-items and agents before it is used as the source of a read-modify-write atomic operation.

An implementation must behave as if all synchronizing memory operations stay in order. Every work-item and agent thread must observe the same total ordering of synchronizing memory operations. Therefore, if sequential consistency is required on atomic memory operations it is only necessary to ensure that the memory operation re-orderings allow within a work-item by acquire and release operations are prevented. This can be achieved by:

- using scar on a read-modify-write operations,
- preceding a load acquire with a release memory fence,
- and following a store release with an acquire memory fence.

One common use of acquire and release memory ordering is to implement a lock for synchronization. In this case, no memory operations in a critical section bracketed by the acquire and release memory operations can be moved out of the section. An acquire access of a global variable ensures that the subsequent memory operations in the critical section will read values no older than the value loaded. The update release of a global variable at the end of the critical section will ensure that all the memory updates done in the critical section have been made visible before the value of that variable is made visible. The global variables can therefore be used to control entry of the critical section, and to communicate that the critical section has completed updating memory.

6.2.2 Memory Scope

The scope of an atomic operation and memory fence can also be specified. An operation is only defined as atomic, and a memory fence is only defined as synchronizing, for other work-items and agents that are members of the same scope and specify the same scope in the operations they execute. Otherwise, it is as if the

atomic memory operation was an ordinary memory operation, and a memory fence has no affect.

Consequently, an atomic operation can result in a race condition if an atomic operation in another work-item or agent thread accesses the same memory location using a different scope, or if the other work-item or agent thread is not a member of the same scope. Also, synchronizing memory operations and memory fence operations only ensure updates become visible to other work-items and agent threads that are members of the same scope and access them specifying the same scope.

A release memory operation or release memory fence, can be used to make previous atomic update memory operations by the same work-item visible to other work-items and agents at a scope different to the one they specified. This is no different than the rules for making any ordinary update memory operation visible to other work-items and agents.

The scope of an atomic memory operation or memory fence is specified by the memory scope modifier which can have the following values:

- `wi` specifies work-item scope which includes only the executing work-item. Only supported by the memory fence operation for the image segment.
- `wv` specifies wavefront scope which includes all work-items in the same wavefront as the executing work-item.
- `wg` specifies work-group scope which includes all work-items in the same work-group as the executing work-item.
- `cmp` specifies HSA component scope which includes all work-items on the same HSA component executing kernel dispatches for the same application process as the executing work-item. Only supported for the global segment.
- `sys` specifies the entire HSA system scope which includes all work-items on all HSA components executing kernel dispatches for the same application process, together with all agents executing the same application process as the executing work-item. Only supported for the global segment.

An implementation may only support `sys` scope on certain ranges of virtual addresses. Also, an implementation may support the use of both `cmp` and `sys` scopes more efficiently on certain ranges of virtual addresses. The runtime will provide a memory allocation operation that allows the scope that will be used to access it to be specified: either `cmp`, `sys` or both. Such memory can be used to implement agent allocation (see [6.2.5 Agent Allocation \(p. 168\)](#)). If a memory operation with `sys` scope is performed on a location with an address in the range allocated by the runtime for allocations that only support `cmp` scope:

- To all work-items executing on the same HSA component, the operation behaves as if it was performed with `sys` scope.
- To work-items executing on other HSA Components and other agents the operation behaves as if it was performed with `cmp` scope. Therefore, it is a data race if a work-item in another HSA component or other agent accesses the location.

A narrower memory scope is appropriate when work-items will write to global segment memory, and other work-items will read back those values, but all communication will only happen between members of the narrower scope. Using a narrower memory scope might be more efficient on some implementations than a wider memory scope.

For example, the amount of data the work-items within a work-group are exchanging might be too large to fit into the group segment. In this case, they could use the global segment, and `wg` memory scope, because the data is only being shared by work-items in the same work-group. In implementations that share an L1 cache over a work-group, the use of `wg` memory scope might allow an implementation to reduce memory traffic and so would be more efficient than using a wider memory scope. However, note that the work-items of different work-groups must access different global memory locations otherwise it is a data race. This is because the updates of one work-group are ordinary updates to another work-group since they are not both members of the same `wg` scope.

The `wi` scope is only supported for the memory fence operation with the image segment. See [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#).

6.2.3 Memory Synchronization Segments

A synchronizing memory operation and memory fence only affects memory operations to the segments specified by the operation. This includes both the rules for reordering and visibility specified by the operation's memory order modifier. The segment of an atomic memory operation and memory fence is specified by the segment modifier of the operation and can have the following values:

- `group` specifies the group segment.
- `global` specifies the global segment.
- `image` specifies the image segment. This is not a regular segment, and can only be used with the memory fence operation. It includes all the memory that holds image data, and is implicitly accessed by the image operations. It is only supported if the "IMAGE" extension directive has been specified (see [13.1.2 extension IMAGE \(p. 292\)](#)).

If the memory segment is omitted for an atomic memory operation, it specifies that a flat memory address is being used. See [6.2.8 Flat Addresses \(p. 171\)](#).

For a memory fence operation, a list of one or more segments must be given, specifying for each one the memory scope. If multiple segments are specified, then for all synchronizing memory operations with a segment and scope that matches one of those specified by the memory fence, performed by work-items or agents that are members of the scope: all operations before the memory fence in program order are made visible before all operations after the memory fence. See [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#).

See [2.8 Segments \(p. 13\)](#).

6.2.4 Non-Memory Synchronization Segments

This section specifies the memory model rules for memory accesses to segments that are not memory synchronization segments (see [6.2.3 Memory Synchronization Segments \(p. 167\)](#)). Only ordinary memory operations are supported for these segments.

The `private`, `spill` and `arg` segments can only be accessed by a single work-item and all accesses behave as if in program order.

The `kernarg` segment values are initialized and made visible before a kernel dispatch starts executing, and their values cannot be changed during the execution of the kernel dispatch. Only load operations are allowed.

`readonly` segment locations have agent allocation (see [6.2.5 Agent Allocation \(p. 168\)](#)) and it is undefined if the locations accessed by a kernel dispatch change value during its execution. Only load operations are allowed. The values can only be changed by the host CPU agent using the HSA runtime operations, which makes the values visible to all subsequent kernel dispatch executions on the associated HSA component. See [4.10 Initializers and Array Declarations \(p. 74\)](#).

See [2.8 Segments \(p. 13\)](#).

6.2.5 Agent Allocation

The `global` segment allows variables to have agent allocation, which results in distinct allocations of the variable for each HSA component, each with a distinct global segment address. All `readonly` segment variables have agent allocation. It is undefined to access a location that is part of an agent allocation except from the HSA component that the allocation is associated. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

6.2.6 Kernel Dispatch Memory Synchronization

Before a work-item starts executing, no implicit acquire memory fence is performed. So any ordinary memory operation executed by a work-item, that precedes the first acquire operation executed by the same work-item, may access values in the modification order as old as the value made visible by the last acquire fence performed by the packet processor of the HSA component on which it is executing.

When a work-item completes execution, no implicit release memory fence is performed. So any ordinary memory operations after the last release operation executed by a work-item are not made visible to other work-items in the same kernel dispatch. Neither are they made visible to work-items in other kernel dispatches unless they are made visible by a subsequent release fence performed by the packet processor of the HSA component that executed the work-item.

The dispatch packets processed by the packet processor can specify an acquire and release fence scope of either:

- `cmp`
- `cmp and sys.`

The barrier packets processed by the packet processor can specify a release fence scope of either:

- `cmp`
- `cmp and sys.`

Barrier packets do not support an acquire fence.

Two packets are members of the same `cmp` scope if they are executed on the same HSA component for the same application. Two packets are members of the same `sys` scope if they are executed on any HSA component for the same application. In

addition, any `sys` scope memory operations done by other agents for the same application are members of the `sys` scope.

The packet processor executes packets on a queue in order. A packet processor can execute packets from multiple queues in parallel. A dispatch packet is used to start the execution of a kernel. Subsequent packet processing of a queue does not have to wait for a dispatch packet to complete execution of the kernel. A barrier packet can be used to ensure previous dispatch packets on the same queue have completed execution. For more information on the packet processor see *HSA Platform System Architecture Specification*.

For the packet acquire fence, all values previously made visible at any of the specified scopes will be made visible to subsequent work-items at all of the specified scopes.

A value is previously visible to a packet acquire fence if:

- a preceding release operation with one of the specified scopes was executed by a work-item that is a member of that scope, by either a currently executing, or previously completed, kernel dispatch;
- one of the specified scopes is `sys` and a preceding `sys` release operation was executed by an agent for the same application;
- a preceding packet was executed that made it visible because the packet: was a member of one of the scopes and specified a release fence with one of the specified scopes;
- or the current packet being executed is a barrier packet that makes it visible because the packet specifies a release fence with one of the specified scopes.

The subsequent work-items of a packet acquire fence are:

- If the current packet is a dispatch packet, then the subsequent work-items include all the work-items of the dispatch. Note that the acquire happens before any work-item of the dispatch executes an operation, not at the time a particular work-item starts executing. Therefore, any ordinary loads done by a work-item that may see a value updated by another work-item of the same dispatch will be a data race. Each work-item must execute an acquire operation, or use atomic operations, to access values updated by other work-items of the same dispatch.
- In addition, the subsequent work-items include the work-items of all subsequent dispatch packets that are members of the scope.

For the packet release fence, all values updated by previous memory operations are made visible to subsequent memory operations at the specified scopes.

The previous memory operations of a packet release fence are:

- If the current packet is a dispatch packet, then the previous memory operations include those executed by all the work-items of the dispatch. Note that the release happens after all work-items of the dispatch complete execution, not at the time a particular work-item completes execution. Therefore, any ordinary stores done by a work-item that may be accessed by another work-item of the same dispatch will be a data race. Each work-item must execute a release operation, or use atomic operations, to update values accessed by other work-items of the same dispatch.
- In addition, the previous memory operations include those executed by work-items of all previous dispatch packets that have completed and are members of the scope.
- For memory operations executed by work-items belonging to dispatch packets that have not completed execution, only the memory operations that have been made visible by synchronizing memory operations and memory fences are included. Note that any ordinary memory operations that have been previously executed, but not made visible by the work-item that executed them, are not made visible, and it would be a data race to access them.

The subsequent memory operations of a packet release fence are:

- Any memory operation subsequently executed by a work-item or agent in the same scope. In the case of a work-item, it can belong to a currently executing dispatch packet, or a future dispatch packet.

Because global memory update operations of a kernel can be made visible by the release fence of the dispatch packet that executes it, or by some future packet executed on the same HSA component, an implementation (both hardware and finalizer) cannot delete the update of the final value of global memory locations by the ordinary memory operations of a work-item, even if it can prove it cannot be accessed by any work-item in the kernel dispatch. For example, using ordinary memory operations, or atomic memory operations with a memory scope of work-group or wavefront, does not give an implementation permission to delete a global memory update operation even if it can determine that no work-item in the work-group or wavefront will access the changed location.

To avoid a data race, a memory location updated by an ordinary memory operation, or an atomic memory operation at a scope less than `sys`, must be made visible by a release to `sys` scope before it can be re-allocated by the runtime for use as a system global variable. Consider that an implementation is allowed to make such values accessible to other work-items and agents at any time between the memory operation and a release at `sys` scope. Similarly for locations used for device only coherent variables being released to `cmp` and `sys` scopes.

6.2.7 Execution Barrier

A barrier operation is used to synchronize the execution of the work-items that participate in an associated execution barrier instance. In addition, an execution barrier operation defines a memory ordering of synchronizing memory operations executed by work-items participating in the execution barrier instance with respect to the synchronizing memory operations executed by the other work-items participating in the same execution barrier instance. See [9.3 Execution Barrier \(p. 246\)](#).

6.2.8 Flat Addresses

Synchronizing memory operations that use a flat address are defined as the equivalent segment address synchronizing memory operation using:

- A segment and segment address corresponding to actual flat address when the flat synchronizing memory operation is executed at runtime.
- A memory scope that is the minimum of the memory scope specified by the flat synchronizing memory operation and the widest scope supported by the segment of the actual flat address when the flat synchronizing memory operation is executed at runtime.
- A memory order corresponding to that specified by the flat synchronizing memory operation.

6.3 Load (ld) Operation

The load (ld) operation loads from memory using a segment or flat address expression (see [4.18 Address Expressions \(p. 88\)](#)) and places the result into one or more registers. It is an ordinary non-synchronizing memory operation (see [6.2 Memory Model \(p. 160\)](#)).

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 [4.16 Operands \(p. 86\)](#). 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#). No conversions are performed on other types. Use an explicit cvt operation if floating-point conversion is required.

If the Base profile has been specified then the 64-bit floating-point type (f64) is not supported (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

6.3.1 Syntax

Table 8–1 Syntax for Load (ld) Operation

Opcode and Modifiers	Operands
<code>ld_segment_align(n)_const_equiv(n)_width_TypeLength</code>	<code>dest0, address</code>
<code>ld_v2_segment_align(n)_const_equiv(n)_width_TypeLength</code>	<code>(dest0, dest1), address</code>
<code>ld_v3_segment_align(n)_const_equiv(n)_width_TypeLength</code>	<code>(dest0, dest1, dest2), address</code>
<code>ld_v4_segment_align(n)_const_equiv(n)_width_TypeLength</code>	<code>(dest0, dest1, dest2, dest3), address</code>

Explanation of Modifiers

`v2`, `v3`, and `v4`: Optional vector element count. Used to specify that multiple contiguous memory locations, each of type `TypeLength`, are being loaded. See the Description below.

`segment`: Optional segment: `global`, `group`, `private`, `kernarg`, `readonly`, `spill`, or `arg`. If omitted, `flat` is used. See [2.8 Segments \(p. 13\)](#).

`align(n)`: Optional. Used to specify the byte alignment of the base of the memory being loaded. If omitted, 1 is used indicating no alignment. See the Description below.

`const`: Optional. Used to indicate if the memory loaded is constant. Only allowed for `global` and `flat` `segment`. See the Description below.

`equiv(n)`: Optional: `n` is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See [6.1.4 Equivalence Classes \(p. 160\)](#).

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the result uniformity of the loaded values. All active work-items in the same slice are guaranteed to load the same value(s). If the `width` modifier is omitted, it defaults to `width(1)`, indicating each active work-item can load different value(s). See the Description below.

`Type`: `u`, `s`, `f`. The `Type` specifies how the value is expanded to the size of the destination. See [Table 6–2 \(p. 79\)](#).

`Length`: 8, 16, 32, 64. If the Base profile has been specified, 64 is not supported if `Type` is `f`. 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 6–2 \(p. 79\)](#) and [16.2.1 Base Profile Requirements \(p. 309\)](#).

`TypeLength` can also be `b128`, in which case `destn` must be a `q` register; or `roimg`, `woimg`, `rwimg`, `samp`, `sig32`, or `sig64`, in which case `destn` must be a `d` register.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest0`, `dest1`, `dest2`, `dest3`: Destination registers.

`address`: Address to be loaded from. Must be an address expression for an address in `segment` (see [4.18 Address Expressions \(p. 88\)](#)).

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

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 [18.7.2 BRIG Syntax for Memory Operations \(p. 372\)](#).

6.3.2 Description

v2, v3, and v4

When v2, v3, or v4 is used, HSAIL will load consecutive values into multiple registers. The address is incremented by the size of the *TypeLength* specified 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];
```

`align(n)`

If specified, indicates that the implementation can rely on the *address* operand having an address that is an integer multiple of *n*. Valid values of *n* are 1, 2, 4, 8, 16, 32, 64, 128 and 256. On some implementations, this may allow the load to be performed more efficiently. It is undefined if a memory load marked as aligned is in fact not unaligned to the specified *n*: on some implementations this might result in incorrect values being loaded or memory exceptions being generated. If `align` is omitted, the value 1 is used for *n*, and the implementation must correctly handle the source address being unaligned. Note, for v2, v3, and v4 only the alignment of the first value is specified: the subsequent values are still loaded contiguously according to the size of *TypeLength*. See [17.8 Unaligned Access \(p. 314\)](#).

`const`

If specified, indicates that the load is accessing constant memory. An implementation can rely on the memory locations loaded not being written to for the lifetime of the variable in the program. Only global and readonly segment loads, and flat addresses that refer to constant global segment memory, can be marked `const`. Note, kernarg segment accesses are implicitly constant memory accesses.

On some implementations, knowing a load is accessing constant memory might be more efficient. It is undefined if a memory load marked as constant is changed during the execution of any kernels that are part of the program: on some implementations this might result in incorrect values being loaded. See [17.9 Constant Access \(p. 314\)](#).

width

Because work-items are executed in wavefronts, a single load can access multiple memory locations if the *address* operand evaluates to different addresses in different work-items. The optional width modifier specifies the result uniformity of the loaded value (see [2.12 Divergent Control Flow \(p. 26\)](#)). It can be *width(n)*, *width(WAVESIZE)*, or *width(all)*. All active work-items in the same slice are guaranteed to load the same result. If the width modifier is omitted, it defaults to *width(1)*, indicating each active work-item can load different values.

In the case of *v2*, *v3*, and *v4*, each work-item produces multiple results. The loads of the work-items in a slice are only result uniform if each corresponding result is the same.

Note that a load operation is considered result uniform if the result(s) of all active work-items in the slice are the same, regardless of whether the *address* operand evaluates to the same addresses in each of the work-items.

If active work-items specified by the width modifier do not load the same values, the behavior is undefined.

Implementations are allowed to have a single active work-item read the value and then broadcast the result to the other active work-items. Some implementations can use this modifier to speed up computations.

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 \(p. 157\)](#).

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 (*s8*, *u8*, *s16* and *u16*) are extended to fill the destination *s* 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.

For example, *ld_u8 \$s2, \$d0* loads one byte and zero-extends to 32 bits.

For other integer types, the size of the source and destination must match, and so *ld_s* and *ld_u* operations result in identical results, because no sign extension or zero extension is required. 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.

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`, the destination must be an `s` register. The memory representation is converted to the implementation-defined register representation after the load. See [4.19.1 Floating-Point Numbers \(p. 91\)](#).

For `roimg`, `woimg`, `rwimg`, `samp`, `sig32`, or `sig64` value types, it is required that the compound type specified on the load must match the value type (see [7.1.9 Using Image Operations \(p. 218\)](#) and [6.8 Notification \(signal\) Operations \(p. 187\)](#)).

The `ld` operation is an ordinary non-synchronizing memory operation. It can be reordered by either the finalizer or hardware, and can cause data races. Load reordering and data races can be prevented by using a synchronizing memory operations or memory fences in conjunction with a synchronizing memory operations. For example, a `atomic_ld_acq` acts like a partial fence; no memory operation after the `atomic_ld_acq` can be moved before the `atomic_ld_acq`. See [6.2 Memory Model \(p. 160\)](#).

Examples

```

ld_global_f32 $s1, [%x];
ld_global_s32 $s1, [%x];
ld_global_f16 $s1, [%x];
ld_global_f64 $d1, [%x];
ld_global_align(8)_f64 $d1, [%x];
ld_global_width(WAVESIZE)_f16 $s1, [%x];
ld_global_align(2)_const_width(all)_f16 $s1, [%x];
ld_arg_equiv(2)_f32 $s1, [%y];
ld_private_f32 $s1, [%$s3+4];
ld_spill_f32 $s1, [%$s3+4];
ld_f32 $s1, [%$s3+4];
ld_align(16)_f32 $s1, [%$s3+4];
ld_v3_s32 ($s1,$s2,$s6), [%$s3+4];
ld_v4_f32 ($s1,$s3,$s6,$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), [%$s0+32];
ld_v4_width(8)_f32 ($s1,$s3,$s6,$s2), [%$s3+4];
ld_equiv(1)_u64 $d6, [128];
ld_v2_equiv(9)_width(4)_f32 ($s1,$s2), [%$s3+4];
ld_width(64)_u32 $s0, [%$s2];
ld_equiv(1)_width(1024)_u64 $d6, [128];
ld_equiv(1)_width(all)_u64 $d6, [128];
ld_global_rwimg $d1, [&rwimage1];
ld_READONLY_roimg $d2, [&roimage1];
ld_global_woimg $d2, [&woimage1];
ld_kernarg_samp $d3, [%sampler1];
ld_global_sig32 $d3, [%signal32];
ld_global_sig64 $d3, [%signal64];

```

6.4 Store (st) Operation

The store (`st`) operation stores a value from one or more registers, or an immediate, (see [4.16 Operands \(p. 86\)](#)) into memory using a segment or flat address expression (see [4.18 Address Expressions \(p. 88\)](#)). It is an ordinary non-synchronizing memory operation (see [6.2 Memory Model \(p. 160\)](#)).

There are four variants of the store operation, depending on the number of sources: one, two, three, or four.

If the Base profile has been specified then the 64-bit floating-point type (`f64`) is not supported (see [16.2.1 Base Profile Requirements \(p. 309\)](#)).

6.4.1 Syntax

Table 8–2 Syntax for Store (st) Operation

Opcode and Modifiers	Operands
<code>st_segment_align(n)_equiv(n)_TypeLength</code>	<code>src0, address</code>
<code>st_v2_segment_align(n)_equiv(n)_TypeLength</code>	<code>(src0,src1), address</code>
<code>st_v3_segment_align(n)_equiv(n)_TypeLength</code>	<code>(src0,src1,src2), address</code>
<code>st_v4_segment_align(n)_equiv(n)_TypeLength</code>	<code>(src0,src1,src2,src3), address</code>

Explanation of Modifiers
<code>v2, v3, and v4</code> : Optional vector element count. Used to specify that multiple contiguous memory locations, each of type <code>TypeLength</code> , are being stored. See the Description below.
<code>segment</code> : Optional segment: <code>global</code> , <code>group</code> , <code>private</code> , <code>spill</code> , or <code>arg</code> . If omitted, <code>flat</code> is used. See 2.8 Segments (p. 13) .
<code>align(n)</code> : Optional. Used to specify the byte alignment of the base of the memory being stored. If omitted, 1 is used indicating no alignment. See the Description below.
<code>equiv(n)</code> : Optional: <code>n</code> is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See 6.1.4 Equivalence Classes (p. 160) .
<code>Type</code> : <code>u</code> , <code>s</code> , <code>f</code> . The <code>Type</code> specifies how the value is extracted from the source to match the size of the destination. See Table 6–2 (p. 79) .
<code>Length</code> : 8, 16, 32, 64. If the Base profile has been specified, 64 is not supported if <code>Type</code> is <code>f</code> . The <code>Length</code> specifies the amount of data stored, and the amount to increment the address when the destination is a vector operand. See Table 6–2 (p. 79) and 16.2.1 Base Profile Requirements (p. 309) .
<code>TypeLength</code> can also be <code>b128</code> , in which case <code>srcn</code> must be a <code>q</code> register; or <code>roimg</code> , <code>woimg</code> , <code>rwimg</code> , <code>samp</code> , <code>sig32</code> , or <code>sig64</code> , in which case <code>srcn</code> must be a <code>d</code> register. If <code>roimg</code> , <code>woimg</code> , <code>rwimg</code> or <code>samp</code> then <code>segment</code> must be <code>arg</code> .

Explanation of Operands (see 4.16 Operands (p. 86))
<code>src0, src1, src2, src3</code> : Sources. Can be register, immediate, or <code>WAVESIZE</code> .
<code>address</code> : Address to be stored into. Must be an address expression for an address in <code>segment</code> (see 4.18 Address Expressions (p. 88)).

Exceptions (see Chapter 12 Exceptions (p. 285))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the <code>aligned</code> modifier has been specified.

For BRIG syntax, see [18.7.2 BRIG Syntax for Memory Operations \(p. 372\)](#).

6.4.2 Description

v2, v3, and v4

When v2, v3, or v4 is used, HSAIL will store consecutive values from multiple registers, immediate values or WAVELENGTH. The address is incremented by the size of the *TypeLength* specified the operation.

Front ends should generate vector forms whenever possible. The following forms are equivalent but the vector form is often faster.

Slow form:

```
st_s32 $d0, [$s1];
st_s32 $d1, [$s1+4];
```

Fast form using the vector:

```
st_v2_s32 ($d0,$d1), [$s1];
```

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.

`align(n)`

If specified, indicates that the implementation can rely on the *address* operand having an address that is an integer multiple of *n*. Valid values of *n* are 1, 2, 4, 8, 16, 32, 64, 128 and 256. On some implementations, this may allow the store to be performed more efficiently. It is undefined if a memory store marked as aligned is in fact not unaligned to the specified *n*: on some implementations this might result in incorrect values being stored, values in other memory locations being modified or memory exceptions being generated. If `align` is omitted, the value 1 is used for *n*, and the implementation must correctly handle the source address being unaligned. Note, for v2, v3, and v4 only the alignment of the first value is specified: the subsequent values are still stored contiguously according to the size of *TypeLength*. See [17.8 Unaligned Access \(p. 314\)](#).

6.4.3 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 \(p. 157\)](#).

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 type values (`s8`, `u8`, `s16` and `u16`) are extracted from the least significant bits of the source `s` 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

For `roimg`, `woimg`, `rwimg`, `samp`, `sig32`, or `sig64` value types, it is required that the compound type specified on the store must match the value type (see [7.1.9 Using Image Operations \(p. 218\)](#) and [6.8 Notification \(signal\) Operations \(p. 187\)](#)).

The `roimg`, `woimg`, `rwimg` and `samp` value types are only allowed if `segment` is `arg` (see [7.1.7 Image Creation and Image Handles \(p. 214\)](#) and [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)).

The `st` operation is an ordinary non-synchronizing memory operation. It can be reordered by either the finalizer or hardware, and can cause data races. Store reordering and data races can be prevented by using a synchronizing memory operations or memory fences in conjunction with a synchronizing memory operations. For example, a `atomic_st_rel` acts like a partial fence; no memory operation before the `atomic_st_rel` can be moved after the `atomic_st_rel`. See [6.2 Memory Model \(p. 160\)](#).

Examples

```

st_global_f32 $s1, [%x];
st_global_align(4)_f32 $s1, [%x];
st_global_u8 $s1, [%x];
st_global_u16 $s1, [%x];
st_global_u32 $s1, [%x];
st_global_u32 200, [%x];
st_global_u32 WAVESIZE, [%x];
st_global_f16 $s1, [%x];
st_global_f64 $d1, [%x];
st_global_align(8)_f64 $d1, [%x];
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_align(4)_f32 $s1, [%$s3+4];
st_v4_f32 ($s1,$s1,$s6,$s2), [%$s3+4];
st_v2_align(8)_equiv(9)_f32 ($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_align(16)_equiv(1)_u64 $d3, [%$s4+32];
st_v2_equiv(1)_u64 ($d1,$d2), [%$s0+32];
st_equiv(1)_u64 $d6, [128];
st_arg_roimg $d2, [%roimage2];
st_arg_rwimg $d1, [%rwimage2];
st_arg_woimg $d2, [%woimage2];
st_arg_samp $d3, [%sampler2];
st_global_sig32 $d3, [%signal32];
st_global_sig64 $d3, [%signal64];

```

6.5 Atomic Memory Operations

Atomic memory operations are executed atomically such that it is not possible for any work-item or agent in the system to observe or modify the memory location at the same memory scope during the atomic sequence.

It is guaranteed that when a work-item issues an atomic read-modify-write memory operation on a memory location, no write to the same memory location using the same memory scope from outside the current atomic operation by any work-item or agent can occur between the read and write performed by the operation.

If multiple atomic memory operations from different work-items or agents target the same memory location, the operations are serialized in an undefined order. In particular, if multiple work-items in the same wavefront target the same memory location, they will be serialized in an undefined order.

The address of atomic memory operations must be naturally aligned to a multiple of the access size. If the address is not naturally aligned, then the result is undefined and might generate a memory exception.

Atomic memory operations only allow global segment, group segment and flat addresses. Accesses to segments other than global and group by means of a flat address is undefined behavior.

Most atomic read-modify-write memory operations have two forms:

- `atomic` operations which return the value that was read before the modification. These operations require the `dest` (destination) operand.
- `atomicnoret` operations which do not return a value. These operations do not have a destination operand.

An implementation may execute `atomicnoret` read-modify-write memory operations faster than the corresponding `atomic` read-modify-write memory operations. Therefore, compilers should identify cases where the result of read-modify-write memory operations is not needed and whenever possible, should generate `atomicnoret` operations.

Both `atomic` and `atomicnoret` operations can specify a memory order and memory scope.

For more information, see:

- [6.2 Memory Model \(p. 160\)](#)
- [6.6 Atomic \(`atomic`\) Operations \(p. 180\)](#)
- [6.7 Atomic No Return \(`atomicnoret`\) Operations \(p. 185\)](#)

6.6 Atomic (`atomic`) Operations

The `atomic` memory (`atomic`) operations atomically load the value at `address` into `dest`, and, except for `atomic_ld`, store the result of a reduction operation at `address`, overwriting the original value. The reduction operation is performed on the loaded value and `src0` (and for `atomic_cas`, also with `src1`). `atomic` operations are atomic memory operations that can either be synchronizing or non-synchronizing, all except `atomic_ld` are read-modify-write operations (see [6.2 Memory Model \(p. 160\)](#)).

6.6.1 Syntax

Table 8–3 Syntax for Atomic Operations

Opcode and Modifiers	Operands
<code>atomic_ld_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address</code>
<code>atomic_and_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_or_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_xor_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_exch_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_add_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_sub_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_wrapinc_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>

Opcode and Modifiers	Operands
<code>atomic_wrapdec_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_max_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_min_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0</code>
<code>atomic_cas_segment_order_scope_equiv(n)_TypeLength</code>	<code>dest, address, src0, src1</code>

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 \(p. 13\)](#).

order: Memory order used to specify synchronization. Can be `rlx` (relaxed) and `scacq` (sequentially consistent acquire) for all operations, and for all operations except `ld` can also be `screl` (sequentially consistent release) or `scar` (sequentially consistent acquire and release). See [6.2.1 Memory Order \(p. 162\)](#).

scope: Memory scope used to specify synchronization. Can be `wv` (wavefront) and `wg` (work-group) for `global` or `group` segments, and for `global` segment can also be `cmp` (HSA component) or `sys` (system). For a flat address, any value can be used, but if the address references the `group` segment, `cmp` and `sys` behave as if `wg` was specified. See [6.2.2 Memory Scope \(p. 165\)](#).

equiv(n): Optional: *n* is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See [6.1.4 Equivalence Classes \(p. 160\)](#).

Type: `b` for `ld`, `and`, `or`, `xor`, `exch`, `cas`; `u` and `s` for `add`, `sub`, `max`, `min`; `u` for `wrapinc`, `wrapdec`. See [Table 6-2 \(p. 79\)](#).

Length: 32, 64. See [Table 6-2 \(p. 79\)](#). 64 is not allowed for small machine model. See [2.9 Small and Large Machine Models \(p. 24\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register.

address: Source location in the specified segment. Must be an address expression for an address in *segment* (see [4.18 Address Expressions \(p. 88\)](#)).

src0, src1: Sources. Can be a register, immediate value, or `WAVESIZE`.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Invalid address exceptions are allowed. May generate a memory exception if address is unaligned.

For Brig syntax, see [18.7.2 BRIG Syntax for Memory Operations \(p. 372\)](#).

6.6.2 Description of Atomic and Atomic No Return Operations

ld**Loads the contents of the *address* into *dest*.**`dest = [address];`**Note:** There is no `atomicnoret` version of this operation.**st****Stores the value in *src0* to *address*.**`[address] = src0;`**Note:** There is only an `atomicnoret` version of this operation.**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*.**

```
original = [address];
[address] = original & src0;
dest = original; // Only if atomic operation
```

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*.**

```
original = [address];
[address] = original | src0;
dest = original; // Only if atomic operation
```

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*.**

```
original = [address];
[address] = original ^ src0;
dest = original; // Only if atomic operation
```

exch**Replaces the contents of the *address* with *src0*. Sets *dest* to the original contents of the *address*.**

```
original = [address];
[address] = src0;
dest = original;
```

Note: There is no `atomicnoret` version of this operation.**add****Adds (using integer arithmetic) the value in *src0* to the contents of the memory location with address *address*. For the atomic operation, sets *dest* to the original contents of the *address*.**

```
original = [address];
[address] = original + src0;
dest = original; // Only if atomic operation
```

sub**Subtracts (using integer arithmetic) the value in *src0* from the contents of the memory location with address *address*. For the atomic operation, sets *dest* to the original contents of the *address*.**

```
original = [address];
[address] = original - src0;
dest = original; // Only for atomic operation
```

min, max

Sets the memory location with *address* to the minimum/maximum of the original value and *src0*. For the atomic operations, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = min/max(original, src0);
dest = original; // Only if atomic operation
```

wrapinc

Increments, with wrapping, the contents of the *address* using the formula:

```
original = [address];
[address] = (original >= src0) ? 0 : (original + 1);
dest = original; // Only for atomic operation
```

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.

Note: If a non-wrapping increment is required, then use *add* with the immediate of 1. On some implementations this may perform significantly better than a *wrapinc*.

wrapdec

Decrement, with wrapping, the contents of the *address* using the formula:

```
original = [address];
[address] = ((original == 0) || (original > src0)) ? src0 : (original - 1);
dest = original; // Only for atomic operation
```

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.

Note: If a non-wrapping decrement is required, then use *sub* with the immediate of 1. On some implementations this may perform significantly better than a *wrapdec*.

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.

```
original = [address];
[address] = (original == src0) ? src1 : original;
dest = original; // Only for atomic operation
```

Examples

```

atomic_ld_global_rlx_sys_equiv(49)_b32 $s1, [&x];
atomic_ld_global_scacq_cmp_b32 $s1, [&x];
atomic_ld_group_scacq_wg_b32 $s1, [&x];
atomic_ld_scacq_sys_b64 $d1, [$d0];

atomic_and_global_ar_wg_u32 $s1, [&x], 23;
atomic_and_global_rlx_wv_u32 $s1, [&x], 23;
atomic_and_group_rlx_wg_u32 $s1, [&x], 23;
atomic_and_rlx_sys_u32 $s1, [$d0], 23;

atomic_or_global_scar_sys_u64 $d1, [&x], 23;
atomic_or_global_screl_sys_u64 $d1, [&x], 23;
atomic_or_group_scacq_wv_u64 $d1, [&x], 23;
atomic_or_rlx_sys_u64 $d1, [$d0], 23;

atomic_xor_global_scar_sys_b64 $d1, [&x], 23;
atomic_xor_global_rlx_sys_b64 $d1, [&x], 23;
atomic_xor_group_rlx_wg_u64 $d1, [&x], 23;
atomic_xor_screl_cmp_u64 $d1, [$d0], 23;

atomic_cas_global_scar_sys_b64 $d1, [&x], 23, 12;
atomic_cas_global_rlx_sys_b64 $d1, [&x], 23, 1;
atomic_cas_group_rlx_wg_u64 $d1, [&x], 23, 9;
atomic_cas_rlx_sys_u64 $d1, [$d0], 23, 12;

atomic_exch_global_scar_sys_b64 $d1, [&x], 23;
atomic_exch_global_rlx_sys_b64 $d1, [&x], 23;
atomic_exch_group_rlx_wg_u64 $d1, [&x], 23;
atomic_exch_rlx_sys_u64 $d1, [$d0], 23;

atomic_add_global_scar_sys_b64 $d1, [&x], 23;
atomic_add_global_rlx_sys_b64 $d1, [&x], 23;
atomic_add_group_rlx_wg_u64 $d1, [&x], 23;
atomic_add_screl_sys_u64 $d1, [$d0], 23;

atomic_sub_global_scar_sys_b64 $d1, [&x], 23;
atomic_sub_global_rlx_sys_b64 $d1, [&x], 23;
atomic_sub_group_rlx_wg_u64 $d1, [&x], 23;
atomic_sub_rlx_cmp_u64 $d1, [$d0], 23;

atomic_wrapinc_global_scar_sys_b64 $d1, [&x], 23;
atomic_wrapinc_global_rlx_sys_b64 $d1, [&x], 23;
atomic_wrapinc_group_rlx_wg_u64 $d1, [&x], 23;
atomic_wrapinc_rlx_sys_u64 $d1, [$d0], 23;

atomic_wrapdec_global_scar_sys_b64 $d1, [&x], 23;
atomic_wrapdec_global_rlx_sys_b64 $d1, [&x], 23;
atomic_wrapdec_group_rlx_wg_u64 $d1, [&x], 23;
atomic_wrapdec_rlx_sys_u64 $d1, [$d0], 23;

atomic_max_global_scar_sys_s64 $d1, [&x], 23;
atomic_max_global_rlx_sys_s64 $d1, [&x], 23;
atomic_max_group_rlx_wg_u64 $d1, [&x], 23;
atomic_max_rlx_sys_u64 $d1, [$d0], 23;

atomic_min_global_scar_sys_s64 $d1, [&x], 23;
atomic_min_global_rlx_sys_s64 $d1, [&x], 23;
atomic_min_group_rlx_wg_u64 $d1, [&x], 23;
atomic_min_rlx_sys_u64 $d1, [$d0], 23;

```

6.7 Atomic No Return (atomicnoret) Operations

The atomic no return memory (atomicnoret) operations, except atomicnoret_st, atomically load the value at location *address*, and store the result of a reduction operation at *address*, overwriting the original value. The reduction operation is performed on the loaded value and *src0* (and for atomicnoret_cas, also with *src1*). The atomicnoret_st operation atomically stores the value in *src0* at *address*. The atomicnoret operations do not have a destination, are atomic memory operations that can either be synchronizing or non-synchronizing, and all except atomicnoret_st are read-modify-write operations (see [6.2 Memory Model \(p. 160\)](#)).

6.7.1 Syntax

Table 8–4 Syntax for Atomic No Return Operations

Opcodes and Modifiers	Operands
<code>atomicnoret_st_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_and_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_or_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_xor_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_add_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_sub_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_wrapinc_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_wrapdec_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_max_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_min_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0</code>
<code>atomicnoret_cas_segment_order_scope_equiv(n)_TypeLength</code>	<code>address, src0, src1</code>

Explanation of Modifiers
<i>segment</i> : Optional segment: <code>global</code> or <code>group</code> . If omitted, <code>flat</code> is used, and <i>address</i> must be in the <code>global</code> or <code>group</code> segment. See 2.8 Segments (p. 13) .
<i>order</i> : Memory order used to specify synchronization. Can be <code>r1x</code> (relaxed) and <code>screl</code> (sequentially consistent release) for all operations, and for all operations except <code>st</code> can also be <code>scacq</code> (sequentially consistent acquire) or <code>scar</code> (sequentially consistent acquire and release). See 6.2.1 Memory Order (p. 162) .
<i>scope</i> : Memory scope used to specify synchronization. Can be <code>wv</code> (wavefront) and <code>wg</code> (work-group) for <code>global</code> or <code>group</code> segments, and for <code>global</code> segment can also be <code>cmp</code> (HSA component) or <code>sys</code> (system). For a flat address, any value can be used, but if the address references the <code>group</code> segment, <code>cmp</code> and <code>sys</code> behave as if <code>wg</code> was specified. See 6.2.2 Memory Scope (p. 165) .
<i>equiv(n)</i> : Optional: <i>n</i> is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See 6.1.4 Equivalence Classes (p. 160) .
<i>Type</i> : <code>b</code> for <code>st</code> , <code>and</code> , <code>or</code> , <code>xor</code> , <code>cas</code> ; <code>u</code> and <code>s</code> for <code>add</code> , <code>sub</code> , <code>max</code> , <code>min</code> ; <code>u</code> for <code>wrapinc</code> , <code>wrapdec</code> . See Table 6–2 (p. 79) .
<i>Length</i> : 32, 64. See Table 6–2 (p. 79) . 64 is not allowed for small machine model. See 2.9 Small and Large Machine Models (p. 24) .

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

address:

Source location in the specified segment. Must be an address expression for an address in *segment* (see [4.18 Address Expressions \(p. 88\)](#)).

src0, src1: Sources. Can be a register, immediate value, or WAVESIZE.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Invalid address exceptions are allowed. May generate a memory exception if address is unaligned.

For BRIG syntax, see [18.7.2 BRIG Syntax for Memory Operations \(p. 372\)](#).

6.7.2 Description

See [6.6.2 Description of Atomic and Atomic No Return Operations \(p. 181\)](#).

The `atomicnoret` operations change memory in the same way as the `atomic` operations but do not have a destination.

Examples

```

atomicnoret_st_global_rlx_sys_equiv(49)_b32 [&x], $s1;
atomicnoret_st_global_screl_cmp_b32 [&x], $s1;
atomicnoret_st_group_screl_wg_b32 [&x], $s1;
atomicnoret_st_screl_sys_b64 [$d0], $d1;

atomicnoret_and_global_scar_wg_b32 [&x], 23;
atomicnoret_and_global_rlx_wv_b32 [&x], 23;
atomicnoret_and_group_rlx_wg_b32 [&x], 23;
atomicnoret_and_rlx_sys_b32 [$d0], 23;

atomicnoret_or_global_scar_sys_b64 [&x], 23;
atomicnoret_or_global_screl_sys_b64 [&x], 23;
atomicnoret_or_group_scacq_wv_b64 [&x], 23;
atomicnoret_or_rlx_sys_b64 [$d0], 23;

atomicnoret_xor_global_scar_sys_b64 [&x], 23;
atomicnoret_xor_global_rlx_sys_b64 [&x], 23;
atomicnoret_xor_group_rlx_wg_b64 [&x], 23;
atomicnoret_xor_screl_cmp_b64 [$d0], 23;

atomicnoret_cas_global_scar_sys_b64 [&x], 23, 12;
atomicnoret_cas_global_rlx_sys_b64 [&x], 23, 1;
atomicnoret_cas_group_rlx_wg_u64 [&x], 23, 9;
atomicnoret_cas_rlx_sys_u64 [$d0], 23, 12;

atomicnoret_add_global_scar_sys_u64 [&x], 23;
atomicnoret_add_global_rlx_sys_s64 [&x], 23;
atomicnoret_add_group_rlx_wg_u64 [&x], 23;
atomicnoret_add_screl_sys_s64 [$d0], 23;

atomicnoret_sub_global_scar_sys_u64 [&x], 23;
atomicnoret_sub_global_rlx_sys_s64 [&x], 23;
atomicnoret_sub_group_rlx_wg_u64 [&x], 23;
atomicnoret_sub_rlx_cmp_s64 [$d0], 23;

atomicnoret_wrapinc_global_scar_sys_u64 [&x], 23;
atomicnoret_wrapinc_global_rlx_sys_u64 [&x], 23;
atomicnoret_wrapinc_group_rlx_wg_u64 [&x], 23;
atomicnoret_wrapinc_rlx_sys_u64 [$d0], 23;

atomicnoret_wrapdec_global_scar_sys_u64 [&x], 23;
atomicnoret_wrapdec_global_rlx_sys_u64 [&x], 23;
atomicnoret_wrapdec_group_rlx_wg_u64 [&x], 23;
atomicnoret_wrapdec_rlx_sys_u64 [$d0], 23;

atomicnoret_max_global_scar_sys_u64 [&x], 23;
atomicnoret_max_global_rlx_sys_s64 [&x], 23;
atomicnoret_max_group_rlx_wg_u64 [&x], 23;
atomicnoret_max_rlx_sys_s64 [$d0], 23;

atomicnoret_min_global_scar_sys_u64 [&x], 23;
atomicnoret_min_global_rlx_sys_s64 [&x], 23;
atomicnoret_min_group_rlx_wg_u64 [&x], 23;
atomicnoret_min_rlx_wg_s64 [$d0], 23;

```

6.8 Notification (signal) Operations

Signal operations are used for notification between threads and work-items belonging to a single process potentially executing on different agents in the HSA system. While notification can be performed with regular atomic memory operations, the HSA

platform architecture signals allow implementations to optimize for power and performance during signal operations. For example, spin loops involving atomic memory operations can be replaced with signal wait operations that can be implemented using more efficient hardware features.

Signals are used in the HSA User Mode Queue architecture for notification of packet submission, completion and dependencies. See *HSA Platform System Architecture Specification* for more information on User Mode Queuing. Signals can also be used for user communication between work-items and threads within the same agent and between different agents.

A signal can only be created and destroyed by HSA Runtime operations. It cannot be created or destroyed directly in HSAIL. Only signals that have been created and not destroyed can be used with signal operations.

A signal is referenced by a signal handle. The value of a signal handle is implementation defined, except that the value 0 is reserved and used to represent the null signal handle. The HSA runtime will never create a signal with the null signal handle. The null signal handle must not be used with signal operations.

A signal is opaque, but includes a signal value. The signal value size is 32 bits for the small machine model, and 64 bits for the large machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)). When a signal is created, the size of the signal value is implied by the machine model. A signal handle that references a signal with a 32 bit signal value is of type `sig32`, and one that references a signal with a 64 bit signal value is of type `sig64`. Both signal handle types are 64 bits in size.

The signal value can only be manipulated by the signal operations provided by the HSA Runtime and by the HSAIL signal operations described in this section. It is undefined to access or update a signal value by any other operation, including both ordinary and atomic memory operations. A signal operation specifies the size of the signal value. A signal operation is undefined if the signal handle provided does not reference a signal with the same size of signal value as specified by the signal operation.

Signals are generally intended for notification between agents. Therefore, signal operations interact with the memory model (see [6.2 Memory Model \(p. 160\)](#)) as if the signal value resides in global segment memory, is naturally aligned (see [6.1.3 Alignment \(p. 159\)](#)) and is accessed using atomic memory operations at system scope. However, an implementation is permitted to allocate the signal value in any memory, provided all operations interact with the memory model as if it was allocated in global segment memory.

Signal operations allow a memory ordering to be specified which is used by the atomic memory operation that accesses the signal value. The memory ordering affects how other global segment memory operations performed by the same work-item or thread are made visible.

Signal handles 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 signal handle that references the signal, not the signal. The memory address of a signal handle can be taken using the `lda` operation, but again this is the address of the signal handle, not the signal.

A signal handle global or readonly segment variable can have an initializer, but only the constant value 0 is allowed, which represents the null signal handle.

6.8.1 Syntax

Table 8–5 Syntax for Signal Operations

Opcode and Modifiers	Operands
<code>signal_ld_order_TypeLength_signalType</code>	<code>dest, signalHandle</code>
<code>signal_and_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_or_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_xor_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_exch_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_add_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_sub_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_cas_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0, src1</code>
<code>signal_wait_waitOp_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0</code>
<code>signal_waittimeout_waitOp_order_TypeLength_signalType</code>	<code>dest, signalHandle, src0, timeout</code>
<code>signalnoret_st_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_and_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_or_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_xor_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_add_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_sub_order_TypeLength_signalType</code>	<code>signalHandle, src0</code>
<code>signalnoret_cas_order_TypeLength_signalType</code>	<code>signalHandle, src0, src1</code>

Explanation of Modifiers
<code>order</code> : Memory order used to specify synchronization. Can be <code>r1x</code> (relaxed) for all operations; <code>scacq</code> (sequentially consistent acquire) for all operations except <code>st</code> ; <code>screl</code> (sequentially consistent release) for all operations except <code>ld</code> , <code>wait</code> and <code>waittimeout</code> ; or <code>scar</code> (sequentially consistent acquire and release) for all operations except <code>st</code> , <code>ld</code> , <code>wait</code> and <code>waittimeout</code> . See 6.2.1 Memory Order (p. 162) .
<code>waitOp</code> : The comparison operation to perform. Can be <code>eq</code> (equal) <code>ne</code> (not equal), <code>lt</code> (less than) and <code>gte</code> (greater than or equal).
<code>Type</code> : <code>b</code> for <code>ld</code> , <code>st</code> , <code>and</code> , <code>or</code> , <code>xor</code> , <code>exch</code> , <code>cas</code> ; <code>u</code> and <code>s</code> for <code>add</code> , <code>sub</code> ; <code>s</code> for <code>wait</code> , <code>waittimeout</code> . See Table 6–2 (p. 79) .
<code>Length</code> : 32, 64. See Table 6–2 (p. 79) . Must match the signal value size of <code>signalType</code> . See 2.9 Small and Large Machine Models (p. 24) .
<code>signalType</code> : <code>sig32</code> , <code>sig64</code> . See Table 6–4 (p. 81) . Must be <code>sig32</code> for small machine model and <code>sig64</code> for large machine model. See 2.9 Small and Large Machine Models (p. 24) .

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register of type *TypeLength*.

signalHandle: A source operand *d* register that contains a value of a signal handle of type *signalType*. It is undefined if the value was not originally loaded from a global, readonly, private, spill, or kernarg segment variable of type *signalType*, or from an arg segment variable that is of type *signalType* that was initialized with a value that is of type *signalType*. Must be a signal handle for a signal created by the HSA runtime that has not been destroyed. Must not be the null signal value of 0.

src0, src1: Sources of type *TypeLength*. Can be a register, immediate value, or *WAVESIZE*.

timeout: Timeout value of type *u64*. Specified in same units as the system timestamp . Can be a register, immediate value, or *WAVESIZE*.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

Invalid address exceptions are allowed. May generate a memory exception if signal handle is null or invalid.

For Brig syntax, see [18.7.2 BRIG Syntax for Memory Operations \(p. 372\)](#).

6.8.2 Description of Signal Operations

ld, st, and, or, xor, exch, add, sub, cas

The signal operations have the same definition as the corresponding atomic operations, with the *segment* as *global*, *scope* as *system*, the same *TypeLength*, the *address* operand corresponding to the global segment address of the signal value specified by the *signalHandle* operand and the same other operands. See [6.6 Atomic \(atomic\) Operations \(p. 180\)](#).

The *signalnoret* operations have the same definition as the corresponding *atomicnoret* operations in a similar manner. See [6.7 Atomic No Return \(atomicnoret\) Operations \(p. 185\)](#)

However, an implementation may use special hardware to cause any suspended work-items or threads that are waiting on the signal to be resumed. The exception is the *signal_ld* which does not change the signal value.

wait

The wait operation suspends a work-item's execution until a signal value satisfies a specified condition, a certain amount of time has elapsed or it spuriously returns. The conditions supported are: equal; not equal; less than; and greater than or equal. The signal value is conceptually read using an `atomic_ld` operation, with the `segment` as `global`, `scope` as `system`, and the `address` operand corresponding to the global segment address of the signal value specified by the `signalHandle` operand. The read value is compared to the value specified by `src0` operand using the signed comparison specified by `waitOp`. When the wait operation resumes, the last signal value read is returned in `dest` operand.

A wait operation is required to timeout and resume execution, even if the condition has not been met, no longer than a time interval that is reasonably close to the signal timeout value defined by the HSA runtime. The HSA runtime provides an operation to obtain this value. Additionally, a wait operation can spuriously resume at any time sooner than the timeout (for example, due to system or other external factors) even when the condition has not been met. Conceptually the wait operation behaves as:

```
timer.init(hsa_signal_get_timeout());
do {
    original = [signal_value_address(signalHandle)];
} while (!(original waitOp src0) && !timer.expired() && !spurious_signal_return());
dest = original;
```

However, an implementation can use special hardware to save power and improve performance. For example, a wait operation may suspend thread or work-item execution, and resume it in response to another signal operation that changes the value of a signal value.

Since the wait operation can return spuriously, it is necessary to test the returned value to see if the condition was met. For this reason a wait operation is often used in a loop. For example:

```
// Wait for signal $d1 to be equal to 10
do {
    signal_wait_eq_scacq_s64_sig64 $d0, $d1, 10;
} while ($d0 != 10);
```

A wait operation can be used in divergent code. However, because it suspends execution of a work-item, care should be taken when waiting on a signal that may be updated by a work-item executing in the same waveform, or a work-item later in the flattened work-item order, as deadlock may occur.

The signal values seen by a wait operation are guaranteed to make forward progress in the modification order of the signal value memory location. However, it is not guaranteed that the wait operation will see all values in the modification order. It is therefore possible that a signal value can be updated such that it satisfies the condition of a suspended wait operation, but the wait operation does not observe it before it is changed to a value that does not satisfy its condition, and therefore the wait operation does not resume. By extension, if this scenario happens while multiple threads or work-items are waiting on a signal, some may resume while some may not. It is up to the application to use signals in a way that accounts for this behavior, for example by ensuring signal values only advance, or using multiple signals to coordinate such multiple updates.

A wait operation is not required to resume immediately the signal value satisfies the condition, even if the wait operation does observe a satisfying value.

waittimeout

Same as `wait` except `src1` is used as the timeout value. `src1` is treated as a `u64` and specified in the same units as the system timestamp (see *HSA Platform System Architecture Specification*). The `src1` value is only a hint, and an implementation can choose to timeout either before or after the specified value, but no longer than a time interval that is reasonably close to the signal timeout value defined by the HSA runtime.

```
timer.init(implementation_defined_signal_timeout(src1, hsa_signal_get_timeout()));
do {
    original = [signal_value_address(signalHandle)];
} while (!(original waitOp src0) && !timer.expired() && !spurious_signal_return());
dest = original;
```

Examples

```

signal_ld_rlx_b64_sig64 $d2, $d0;
signal_ld_scacq_b32_sig32 $s2, $d1;

signal_and_scar_u64_sig64 $d2, $d0, 23;
signal_and_u32_sig32 $s2, $d1, 23;

signal_or_scar_u64_sig64 $d2, $d0, 23;
signal_or_screl_u32_sig32_sig32 $s2, $d1, 23;

signal_xor_scar_b64_sig64 $d2, $d0, 23;
signal_xor_rlx_b32_sig32 $s2, $d1, 23;

signal_cas_scar_b64_sig64 $d2, $d0, 23, 12;
signal_cas_rlx_b32_sig32 $s2, $d1, 23, 1;

signal_exch_scar_b64_sig64 $d2, $d0, 23;
signal_exch_rlx_b32_sig32 $s2, $d1, 23;

signal_add_scar_b64_sig64 $d2, $d0, 23;
signal_add_rlx_b32_sig32 $s2, $d1, 23;

signal_sub_scar_b64_sig64 $d2, $d0, 23;
signal_sub_rlx_b32_sig32 $s2, $d1, 23;

signal_wait_eq_rlx_s64_sig64 $d2, $d0, 23;
signal_wait_ne_rlx_s64_sig64 $d2, $d0, $d3;
signal_wait_lt_rlx_s32_sig32 $s2, $d1, WAVESIZE;
signal_wait_gte_rlx_s32_sig32 $s2, $d1, 23;

signal_waittimeout_eq_rlx_s64_sig64 $d2, $d0, 23, $d4;
signal_waittimeout_ne_rlx_s64_sig64 $d2, $d0, $d3, 1000;
signal_waittimeout_lt_rlx_s32_sig32 $s2, $d1, WAVESIZE, $d4;
signal_waittimeout_gte_rlx_s32_sig32 $s2, $d1, 23, $d4;

signalnoret_st_rlx_b64_sig64 $d0, $d2;
signalnoret_st_screl_b32_sig32 $d1, $s2;

signalnoret_and_scar_b64_sig64 $d0, 23;
signalnoret_and_rlx_b32_sig32 $d1, 23;

signalnoret_or_scar_b64_sig64 $d0, 23;
signalnoret_or_screl_b32_sig32 $d1, 23;

signalnoret_xor_scar_b64_sig64 $d0, 23;
signalnoret_xor_rlx_b32_sig32 $d1, 23;

signalnoret_cas_scar_b64_sig64 $d0, 23, 12;
signalnoret_cas_rlx_b32_sig32 $d1, 23, 1;

signalnoret_add_scar_u64_sig64 $d0, 23;
signalnoret_add_rlx_s32_sig32 $d1, 23;

signalnoret_sub_scar_u64_sig64 $d0, 23;
signalnoret_sub_rlx_s32_sig32 $d1, 23;

```

6.9 Memory Fence (memfence) Operation

The memory fence (`memfence`) operation can either be a release memory fence, an acquire memory fence, or both an acquire and a release memory fence. `memfence` operations are synchronizing memory operations (see [6.2 Memory Model \(p. 160\)](#)).

6.9.1 Syntax

Table 8–6 Syntax for memfence Operation

Opcode and Modifier
<code>memfence_order_global(scope)_group(scope)_image(scope)</code>
Explanation of Modifier
<p>order: Memory order used to specify synchronization. Can be <code>scacq</code> (sequentially consistent acquire), <code>screl</code> (sequentially consistent release) or <code>scar</code> (sequentially consistent acquire and release). If the <code>image</code> segment is specified, must be <code>scar</code>. See 6.2.1 Memory Order (p. 162).</p> <p>global, group, image: Optional segments. Must specify at least one. Can appear in any order. The <code>image</code> segment is only allowed if the "IMAGE" extension directive has been specified (see 13.1.2 extension IMAGE (p. 292)). If <code>image</code> is specified, no other segment can be specified. See 6.2.3 Memory Synchronization Segments (p. 167).</p> <p>scope: Memory scope used to specify synchronization for each segment specified. Can be <code>wv</code> (wavefront) and <code>wg</code> (work-group) for <code>global, group</code> or <code>image</code> segments. For <code>global</code> segment can also be <code>cmp</code> (HSA component) or <code>sys</code> (system). For <code>image</code> segment can also be <code>wi</code> (work-item). See 6.2.2 Memory Scope (p. 165).</p>

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.5 BRIG Syntax for Parallel Synchronization and Communication Operations \(p. 375\)](#).

6.9.2 Description

The `memfence` operation allows memory access and updates to be synchronized between work-items and other agents. See [6.2 Memory Model \(p. 160\)](#).

For example:

```
st_global_u32 1, [&x];
memfence_screl_global(sys); // Will ensure 1 is visible to work-items that
                           // subsequently perform an acquire of the global
                           // segment at sys scope.
```

The `memfence` operation can be used in conditional code.

Multiple segments can be specified in the same `memfence` operation to establish a memory ordering between memory operations of the specified segments. This is different to specifying separate `memfence` operations for each segment which only establishes a memory ordering of the memory operations of each segment independent of the other segments: in particular the separate memory fences can be reordered relative to the memory fences of other segments. See [6.2 Memory Model \(p. 160\)](#).

If the `image` segment is specified then the `global` and `group` segments cannot also be specified in the same `memfence` operation. The only way to order image operations

with global or group segment operations is by an execution barrier or a User Mode Queue packet fence. The `image` segment is only supported if the "IMAGE" extension directive has been specified (see [13.1.2 extension IMAGE \(p. 292\)](#)).

The `wi` scope is only supported for the image segment. Image read operations are conceptually performed at a *read-work-item* scope, and image write operations at a *write-work-item* scope. So to make the image writes performed by a single work-item visible to the image reads it performs, it must perform an acquire and release memory fence on the image segment using the `wi` or larger scope.

Examples

```
memfence_scacq_global(sys);  
memfence_screl_group(wg);  
memfence_scacq_global(cmp)_group(wg);  
memfence_scar_image(wi);  
memfence_scar_group(wg)_global(wg);
```


Chapter 7

Image Operations

This chapter describes how images and samplers are used in HSAIL and also describes the associated read, load, store, memory fence and query operations.

The image operations defined in this chapter are only allowed if the "IMAGE" extension directive has been specified. See [13.1.2 extension IMAGE \(p. 292\)](#).

The minimum limits with respect to images are specified in [Appendix A Limits \(p. 393\)](#).

Note: For background information, see:

- The OpenCL™ Specification Version 2.0:
 - *5.3 Image Objects*
 - <http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>
- The OpenCL C Specification Version 2.0:
 - *6.13.14 Image Read and Write Functions*
 - *8. Image Addressing and Filtering*
 - <http://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf>
- The OpenCL Extension Specification Version 2.0:
 - <http://www.khronos.org/registry/cl/specs/opencl-2.0-extensions.pdf>

7.1 Images in HSAIL

7.1.1 Why Use Images?

Images are a graphics feature that can sometimes be useful in data-parallel computing. Images can be accessed in one, two, or three dimensions. 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:

- Special caches and tiling modes that reorder the memory locations of 2D and 3D images. Implementations can also insert gaps in the memory layout to improve alignment. These can save bandwidth by improving data locality and cache line usage compared to traditional linear arrays.
- Image implementations can create caching hints using read-only images.

- Hardware support for out-of-bounds coordinates.
- Image coordinates can be unnormalized, or normalized floating-point values. When a normalized coordinate is used, it is scaled to the image size of the corresponding dimension, allowing values in the range 0.0 to +1.0 to access the entire image.
- The values read and written to an image can be stored in memory as integer values, but returned as unsigned or signed normalized floating-point values in the range 0.0 to +1.0 or -1.0 to +1.0 respectively.
- Values can be converted between linear RGB and sRGB color spaces.
- Image memory offers different addressing modes, as well as data filtering, for some specific image formats. For example, linear filtering is a way to determine a value for a normalized floating-point coordinate by averaging the values in the image that are around the coordinate. Mathematically, this tends to smooth out the values or “filter” out high-frequency changes.

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.

Images can also be used to optimize write operations by delaying them until the next kernel execution.

7.1.2 Image Overview

An image consists of the following information:

- Image geometry
- Image format
- Image size
- Reference to the actual image data

An image is conceptually an array of image elements (also known as pixels). The image elements can either be organized as a single one, two or three dimensional image layer, or as an array of one or two dimensional image layers. The organization is termed the image geometry. An image is indexed by one, two or three coordinates accordingly. The coordinates are named x , y and z . Additionally, there is a one dimensional image buffer geometry that allows image data to be allocated in the global segment. See [7.1.3 Image Geometry \(p. 199\)](#).

The image format specifies the properties of the image elements in terms of their channel order and channel type. Each element in the image has the same image format. See [7.1.4 Image Format \(p. 201\)](#).

There can be implementation dependent restrictions on how an image can be accessed and there are a minimum set of required access permissions for different image formats and geometries. See [7.1.5 Image Access Permission \(p. 207\)](#).

Images are accessed using image coordinates. See [7.1.6 Image Coordinate \(p. 208\)](#).

Images are created by the HSA runtime for a specific agent by specifying the image properties that include the image geometry, image size, image format, image access

permission and image data. Images are referenced by image operations using an opaque image handle. See [7.1.7 Image Creation and Image Handles \(p. 214\)](#).

The `rdimage` image operation uses a sampler to specify how the image coordinates are processed to access the image data. Samplers are created by the HSA runtime for a specific agent by specifying the coordinate processing properties. Samplers are referenced by image operations using an opaque sampler handle. See [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#).

There are a set of image operations that access images, and these have certain limitations on which images they can operate, and how samplers can be used. There are also requirements on how image and sampler handles are used. See [7.1.9 Using Image Operations \(p. 218\)](#).

The image memory model defines the interaction of image operations between different work-items and other agents. See [7.1.10 Image Memory Model \(p. 220\)](#).

7.1.3 Image Geometry

Each image has an associated geometry. See [Table 9–1 \(p. 199\)](#) for a list of the image geometries supported.

Table 9–1 Image Geometry Properties

Image Geometry	Coordinates			Channel Orders	Image Operations	Description
	x	y	z			
1d	width	unused	unused	a, r, rx, rg, rgx, ra, rgba, rgb, rgbx, bgra, argb, abgr, srgb, srgbx, srgba, sbgra, intensity, luminance	<code>rdimage_1d</code> , <code>ldimage_1d</code> , <code>stimage_1d</code>	one-dimensional image
2d	width	height	unused		<code>rdimage_2d</code> , <code>ldimage_2d</code> , <code>stimage_2d</code>	two-dimensional image
3d	width	height	depth		<code>rdimage_3d</code> , <code>ldimage_3d</code> , <code>stimage_3d</code>	three-dimensional image
1da	width	array index	unused		<code>rdimage_1da</code> , <code>ldimage_1da</code> , <code>stimage_1da</code>	one-dimensional image array
2da	width	height	array index		<code>rdimage_2da</code> , <code>ldimage_2da</code> , <code>stimage_2da</code>	two-dimensional image array
1db	width	unused	unused		<code>ldimage_1db</code> , <code>stimage_1db</code>	one-dimensional image buffer
2ddepth	width	height	unused	depth, depth_stencil	<code>rdimage_2ddepth</code> , <code>ldimage_2ddepth</code> , <code>stimage_2ddepth</code>	two-dimensional depth image
2dадепт	width	height	array index		<code>rdimage_2dадепт</code> , <code>ldimage_2дадепт</code> , <code>stimage_2дадепт</code>	two-dimensional depth image array

1D

A 1D image contains image data that is organized in one dimension with a size specified by width. It can be addressed with a single coordinate x .

2D

A 2D image contains image data that is organized in two dimensions with a size specified by width and height. It can be addressed by two coordinates (x, y) corresponding to the width and height respectively.

3D

A 3D image contains image data that is organized in three dimensions with a size specified by width, height and depth. It can be addressed by three coordinates (x, y, z) corresponding to the width, height and depth respectively.

1DA

A 1DA image contains an array of a homogeneous collection of one-dimensional images, all with the same size, format and order, with a size specified by width and array index. It can be addressed by two coordinates (x, y) corresponding to the width and array index respectively.

If a sampler is used, special rules apply to the array index y coordinate. It is always treated as unnormalized even if the sampler specifies normalized. It is rounded to an integral value using round to nearest even integer, and clamped to the range 0 to array size - 1.

An important difference between 1DA and 2D images is that samplers never cause values in different images layers of the array to be combined when computing the returned image element.

2DA

A 2DA image contains an array of a homogeneous collection of two-dimensional images, all with the same size, format and order, with a size specified by width, height and array size. It can be addressed by three coordinates (x, y, z) corresponding to the width, height and array index respectively.

If a sampler is used, special rules apply to the array index z coordinate. It is always treated as unnormalized even if the sampler specifies normalized. It is rounded to an integral value using round to nearest even integer, and clamped to the range 0 to array size - 1.

An important difference between 2DA and 3D images is that samplers never cause values in different images layers of the array to be combined when computing the returned image element.

1DB

A 1DB image contains image data that is organized in one dimension with a size specified by width. It can be addressed with a single coordinate x .

Samplers cannot be used with 1DB images so they cannot be used with the `rdimage` image operation.

An important difference between 1DB and 1D images is that the image data can be allocated in the global segment and can have larger limits on the maximum image size supported. On some implementations this may result in a 1DB image having lower performance than an equivalent 1D image. The image data layout is implementation dependent. Access to the image data using both global segment addressing and image operations is undefined unless the image segment, which is used when image operations access image data, is made coherent with the global segment by appropriate acquire and release memory fences. See [7.1.10 Image Memory Model \(p. 220\)](#)

2DDEPTH

Same as the 2D geometry except the image operations only have a single access component instead of four. Requires that the image component order be `depth` or `depth_stencil`.

2DADEPTH

Same as the 2DA geometry except the image operations only have a single access component instead of four. Requires that the image component order be `depth` or `depth_stencil`.

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 [18.3.12 BrigImageGeometry \(p. 324\)](#).

7.1.4 Image Format

The image format specifies the properties of the image elements in terms of their channel order and channel type. Each element in the image has the same image format. Associated with an image format there is a number called the bits per pixel (`bpp`) which is the number of bits needed to hold one element of an image.

7.1.4.1 Channel Order

Each image element in the image data has one, two, three or four values called memory components (also known as channels). Typically the memory components are named `r`, `g`, `b` and `a` (for red, green, blue, and alpha respectively), which can correspond to the color and transparency of the pixel), although some image orders use other names such as `i`, `L` and `d` (for intensity, luminance and depth respectively).

The image access operations always specify four access components regardless of the number of memory components present in the image data. The exception is the `2DDEPTH` and `2DADEPTH` image geometries which only have one access component.

The channel order specifies how many memory components each image element has and how those memory components are mapped to the four access components. The mapping is also referred to as swizzling. See [Table 9–2 \(p. 201\)](#) for a list of the channel orders supported.

Table 9–2 Channel Order Properties

Channel Order	Memory Components	Access Components	Border Color	Channel Types	Image Geometries
<code>a</code>	(<code>a</code>)	(<code>0,0,0,a</code>)	(<code>0,0,0,0</code>)	<code>snorm_int8, snorm_int16,</code> <code>unorm_int8, unorm_int16,</code> <code>signed_int8, signed_int16,</code>	1D, 2D, 3D, 1DA, 2DA, 1DB
<code>r</code>	(<code>r</code>)	(<code>r,0,0,1</code>)	(<code>0,0,0,1</code>)	<code>signed_int32, unsigned_int8,</code> <code>unsigned_int16,</code>	
<code>rx</code>	(<code>r</code>)	(<code>r,0,0,1</code>)	(<code>0,0,0,0</code>)	<code>unsigned_int32, half_float,</code> <code>float</code>	
<code>rg</code>	(<code>r,g</code>)	(<code>r,g,0,1</code>)	(<code>0,0,0,1</code>)		
<code>rgx</code>	(<code>r,g</code>)	(<code>r,g,0,1</code>)	(<code>0,0,0,0</code>)		
<code>ra</code>	(<code>r,a</code>)	(<code>r,0,0,a</code>)	(<code>0,0,0,0</code>)		
<code>rgba</code>	(<code>r,g,b,a</code>)	(<code>r,g,b,a</code>)	(<code>0,0,0,0</code>)		
<code>rgb</code>	(<code>r,g,b</code>)	(<code>r,g,b,1</code>)	(<code>0,0,0,1</code>)	<code>unorm_short_565,</code> <code>unorm_short_555,</code> <code>unorm_int_101010</code>	
<code>rgbx</code>	(<code>r,g,b</code>)	(<code>r,g,b,1</code>)	(<code>0,0,0,0</code>)		

Channel Order	Memory Components	Access Components	Border Color	Channel Types	Image Geometries
bgra	(b,g,r,a)	(r,g,b,a)	(0,0,0,0)	unorm_int8, snorm_int8, signed_int8, unsigned_int8	
argb	(a,r,g,b)	(r,g,b,a)	(0,0,0,0)		
abgr	(a,b,g,r)	(r,g,b,a)	(0,0,0,0)		
srgb	(r,g,b)	(r,g,b,1)	(0,0,0,1)	unorm_int8 (<i>Component memory type representation uses sRGB, and access type representation uses linear RGB.</i>)	
srgbx	(r,g,b)	(r,g,b,1)	(0,0,0,0)		
srgba	(r,g,b,a)	(r,g,b,a)	(0,0,0,0)		
sbgra	(b,g,r,a)	(r,g,b,a)	(0,0,0,0)	<i>The conversion is done before computing the weighted average when a sampler with linear filtering is used.)</i>	
intensity	(i)	(i,i,i,i)	(0,0,0,0)	unorm_int8, unorm_int16,	
luminance	(L)	(L,L,L,1)	(0,0,0,1)	snorm_int8, snorm_int16, half_float, float	
depth	(d)	(d)	(1)	unorm_int16, unorm_int24, float	2DDEPTH, 2DADEPTH
depth_stencil	(d,s)	(d)	(1)	unorm_int24, float (<i>The stencil value s is not available in HSAIL.</i>)	

7.1.4.1.1 *x*-Form Channel Orders

The *x*-form channel orders differ from the corresponding non-*x*-form channel orders only in the value of the *a* component used for the border color. The *x*-forms use 0, resulting in transparent white, and the non-*x* forms use 1, resulting in opaque white. Thus an *x*-form conceptually behaves the same as the corresponding non-*x*-form image order with an *a* component, such that the *a* component is set to 1 for all elements that are in range of the image dimensions, and 0 for any elements outside the range of the image dimensions. Thus the *x*-form avoids the expense of actually storing the *a* component in the image data. This also allows a sampler with linear filtering and `clamp_to_border` addressing mode to anti-alias the edge of an image with an *x*-form channel order. For example, an `xrgb` channel order behaves like the `anrgba` channel order which has the alpha component set to 1 for in-range elements and 0 for out-of-range elements, but only requires the same amount of image data memory as the `rgb` channel order.

7.1.4.1.2 Standard RGB (*s*-Form) Channel Orders

Standard RGB (sRGB) data roughly displays colors in a linear ramp of luminosity levels such that an average observer, under average viewing conditions, can view them as perceptually equal steps on an average display. For more information see the SRGB color standard, IEC 61996-2-1, at IEC (International Electrotechnical Commission).

The `srgb`, `srgbx`, `srgba`, `sbgra` channel orders differ from the corresponding non-*s*-forms in that they convert the *r*, *g* and *b* components from linear RGB to sRGB values when storing to memory, and from sRGB to linear RGB on read. The *a* channel, if present, is not converted and is always treated as linear. When a sampler is used with linear filtering, the conversion is done before the weighted average is computed.

When reading an *s*-form channel order, the *r*, *g* and *b* memory component values are first converted to sRGB access component values using the channel type conversion method (see [7.1.4.2 Channel Type \(p. 203\)](#)), and then the resulting sRGB access values are converted to a linear RGB access values by evaluating the following formula:

```
access_component = (access_component ≤ 0.04045) ? (access_component / 12.92)
: (((access_component + 0.055) / 1.055)2.4);
```

This conversion must be done such that the infinitely precise inverse conversion applied to the result is within 0.5 ulp of the original value, with the additional requirement that an sRGB access component value of 0.0 or 1.0 is converted to the same linear RGB access component value.

When storing an s-form channel order, the linear RGB r , g and b access component values are first converted to sRGB access component values using the following formula, and then the channel type conversion method is used to convert the resulting sRGB access component values to the memory component value:

```
access_component = (access_component_is_nan) ? 0.0
: (access_component > 1.0) ? 1.0
: (access_component < 0.0) ? 0.0
: (access_component < 0.0031308) ? (access_component * 12.92)
: ((1.055 * c1.0/2.4) - 0.055);
```

This conversion must be done such the result is less than or equal to 0.6 of the infinitely precise result, with the additional requirements that a linear access component value of 0.0 or 1.0 is converted to the same sRGB access component value, and that the result is in the closed interval [0.0, 1.0]. No invalid operation exception is generated if the value is a signalling NaN.

No inexact exception is generated for either conversion.

The HSA runtime allows the same image data to be referenced by a 2D image handle created specifying the s-form channel order and one that was created with the same image geometry, size and format except that the corresponding non-s-form of the channel order was specified. This allows the same image data to be accessed using either sRGB values or linear RGB values. Only one of the handles can be used at a time in a single kernel dispatch if writes are performed.

7.1.4.2 Channel Type

The channel type specifies both the component memory type and the component access type. The component memory type specifies how the value of the memory component is encoded in the image data. The component access type specifies how the value of the memory component is returned by image read operations, or specified to image store operations. Each channel type has a conversion method that is used to converted from the component memory type to the component access type by image read operations, and from the component access type to the component memory type by image write operations. See [Table 9–3 \(p. 203\)](#) for a list of the channel types supported together with their properties.

Table 9–3 Channel Type Properties

Channel Type	Memory Type		Access Type	Conversion Method
	Bit Size	Encoding		
snorm_int8	8	signed integer	f32	SignedNormalize(2^7-1)
snorm_int16	16	signed integer		SignedNormalize($2^{15}-1$)
unorm_int8	8	unsigned integer		UnsignedNormalize(2^8-1)
unorm_int16	16	unsigned integer		UnsignedNormalize($2^{16}-1$)
unorm_int24	24	unsigned integer		UnsignedNormalize($2^{24}-1$)
unorm_short_565	r=5 bits[15:11]	unsigned integer		UnsignedNormalize(2^5-1)
	g=6 bits[10:05]			UnsignedNormalize(2^6-1)
	b=5 bits[04:00]			UnsignedNormalize(2^5-1)

Channel Type	Memory Type		Access Type	Conversion Method
	Bit Size	Encoding		
<code>unorm_short_555</code>	r=5 bits[14:10]	unsigned integer		
	g=5 bits[09:05]			
	b=5 bits[04:00]			
	ignored bit[15]			
<code>unorm_int_101010</code>	r=10 bits[29:20]	unsigned integer		UnsignedNormalize($2^{10}-1$)
	g=10 bits[19:10]			
	b=10 bits[09:00]			
	ignored bits[31:30]			
<code>signed_int8</code>	8	signed integer	s32	SignedClamp($-2^8, 2^8-1$)
<code>signed_int16</code>	16	signed integer		SignedClamp($-2^{15}, 2^{15}-1$)
<code>signed_int32</code>	32	signed integer		Identity()
<code>unsigned_int8</code>	8	unsigned integer	u32	UnsignedClamp(2^8-1)
<code>unsigned_int16</code>	16	unsigned integer		UnsignedClamp($2^{16}-1$)
<code>unsigned_int32</code>	32	unsigned integer		Identity()
<code>half_float</code>	16	float	f32	HalfFloat()
<code>float</code>	32	float		Float()

The memory type is specified as the number of bits occupied by the component (also known as the bit depth), and whether the value is represented as a two's complement signed or unsigned integer or as an IEEE/ANSI Standard 754-2008 for floating-point value (see [4.19.1 Floating-Point Numbers \(p. 91\)](#)). For the packed representations of `unorm_short_555`, `unorm_short_565` and `unorm_short_101010`, the components are the specified bit fields within the image element. For `unorm_short_565` the bit size varies according to whether the r, g or b component.

The access type is the HSAIL type used in the operands of the image operations that specify the image component (see [Table 6–2 \(p. 79\)](#)).

The conversion method can be one of:

`Identity()`

No conversion is performed. On read or write all values are preserved.

`Float()`

On a read or write image operation, it is implementation defined if subnormal values are flushed to zero, if NaN values are propagated in either profile or if signalling NaNs are converted to quiet NaNs (see [4.19.4 Not A Number \(NaN\) \(p. 93\)](#)). All other values are preserved. No invalid operation exception is generated if the value is a signalling NaN. No inexact exception is generated.

HalfFloat()

On a read image operation, the memory component value is converted from $f16$ to $f32$. The conversion must be exact for both normal and subnormal values. The infinity value must be converted to the corresponding infinity value. It is implementation defined if NaN values are propagated in either profile or if signalling NaNs are converted to quiet NaNs (see [4.19.4 Not A Number \(NaN\) \(p. 93\)](#)). No invalid operation exception is generated if the value is a signalling NaN.

On write image operations, the access component value is converted from $f32$ to $f16$. It is implementation defined whether `near` or `zero` rounding mode is used (see [4.19.2 Rounding \(p. 92\)](#)). It is implementation defined if subnormal values resulting from the conversion are flushed to zero. The infinity value must be converted to the corresponding infinity value. It is implementation defined if NaN values are propagated in either profile or if signalling NaNs are converted to quiet NaNs (see [4.19.4 Not A Number \(NaN\) \(p. 93\)](#)). No invalid operation exception is generated if the value is a signalling NaN. No inexact exception is generated.

UnsignedClamp(*upper*)

The unsigned integer access component value is clamped to be in the unsigned integer memory component value closed interval $[0, upper]$.

On a read image operation, the access component is set to the memory component value zero extended to $u32$.

On write image operations, the memory component value is set to:

```
memory_component = min(access_component, upper);
```

SignedClamp(*lower*, *upper*)

The signed integer access component value is clamped to be in the signed integer memory component value closed interval $[lower, upper]$.

On a read image operation, the access component is set to the memory component value sign extended to $s32$.

On write image operations, the memory component value is set to:

```
memory_component = min(max(access_component, lower), upper);
```

UnsignedNormalize(*scale*)

A floating-point access component value in the closed internal [0.0, 1.0] is scaled to the unsigned integer memory component value closed interval [0, *scale*], with values outside that range (including infinity) being clamped to the memory component range and NaN values treated as 0.

On a read image operation, the access component is set to:

```
access_component = min(max(float(memory_component) / float(scale), 0.0), 1.0);
```

This must be done with less than or equal to 1.5 ulp, with the additional requirements:

- If memory component is 0 must return 0.0.
- If memory component is *scale* then must return 1.0.
- Must return a value in the closed interval [0.0, 1.0].

On write image operations, the memory component value is set to:

```
memory_component = min(max(int_neari(access_component * float(scale)), 0), scale);
```

The conversion to integer uses `neari` rounding mode (see [5.18.4 Description of Integer Rounding Modes \(p. 153\)](#)). The result must be in the closed interval of the precise result produced for the access component value $\pm(0.6 / \text{float}(\text{scale}))$. No invalid operation exception is generated if the value is a signalling NaN.

No inexact exception is generated for either conversion.

SignedNormalize(*scale*)

A floating-point access component value in the closed interval [-1.0 to +1.0] is scaled to the signed integer memory component value closed interval [-*scale*-1, +*scale*], with values outside that range (including infinity) being clamped to the memory component range and NaN values treated as 0.

On a read image operation, the access component is set to:

```
access_component = min(max(float(memory_component) / float(scale), -1.0), 1.0);
```

This must be done with less than or equal to 1.5 ulp, with the additional requirements:

- If memory component is -*scale* or -*scale*-1 then must return -1.0.
- If memory component is 0 must return 0.0.
- If memory component is *scale* then must return 1.0.
- Must return a value in the closed interval [-1.0, +1.0].

On write image operations, the memory component value is set to:

```
memory_component = min(max(int_neari(access_component * float(scale)), -scale - 1), scale);
```

The conversion to integer uses `neari` rounding mode (see [5.18.4 Description of Integer Rounding Modes \(p. 153\)](#)). The result must be in the closed interval of the precise result produced for the access component value $\pm(0.6 / \text{float}(\text{scale}))$. No invalid operation exception is generated if the value is a signalling NaN.

No inexact exception is generated for either conversion.

7.1.4.3 Bits Per Pixel (bpp)

Associated with each image format there is a number called the bits per pixel (bpp) which is the number of bits needed to hold one element of an image. The bpp value is

obtained by adding the size of each image component plus any unused bits. The image format channel type specifies the component size, and the channel order specifies the number of components. For example, if the channel order is `rg` (two components per element) and if the channel type is `half_float` (16-bit) then the `bpp` value is $2 \times 16 = 32$ bits. See the `bpp` column of [Table 9-4 \(p. 207\)](#).

7.1.5 Image Access Permission

The image access permissions refer to how an image can be accessed using image operations. If the access permissions of a specific image include:

- read-only, then image read operations are allowed
- write-only, then write operations are allowed
- read-write, then both read and write operations are allowed

Not all combinations of image geometry, channel order and channel type are legal in HSAIL. Furthermore, of the legal combinations, it is implementation defined what access permissions, if any, are supported by a specific HSA component. However, for every HSA component that supports images, there is a minimal set of access permissions that must be supported for specific combinations. The HSA runtime provides a query to determine what access permissions, if any, are supported for a given combination on a particular HSA component. It is undefined if an image operation requires an access permission not supported by the HSA component for a specific image. See [Table 9-4 \(p. 207\)](#) for the legal combinations, and for the minimal required access permissions:

- **Y** means the combination of image geometry, channel order and channel type is legal. All other combinations are not legal.
- **ro** means an HSA component that supports images is required to support the combination for the read-only access permission. Otherwise, it may optionally support it if legal.
- **wo** means an HSA component that supports images is required to support the combination for the write-only access permission. Otherwise, it may optionally support it if legal.
- **rw** means an HSA component that supports images is required to support the combination for the read-write access permission. Otherwise, it may optionally support it if legal.

[Table 9-4 Channel Order, Channel Type and Image Geometry Combinations](#)

Channel Order	Channel Type						Image Geometry	bpp
	Bits	<code>unorm</code>	<code>snorm</code>	<code>uint</code>	<code>sint</code>	<code>float</code>		
<code>r</code>	8	Y (ro,wo,rw)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)		1D, 2D, 3D, 1DA, 2D2, 1DB	8 16 32
	16	Y (ro,wo)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		
	32			Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		
<code>rx, a</code>	8	Y	Y	Y	Y			8 16 32
	16	Y	Y	Y	Y	Y		
	32			Y	Y	Y		
<code>rg</code>	8, 8	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)			16 32
	16, 16	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)		

Channel Order	Channel Type						Image Geometry	bpp
	Bits	unorm	snorm	uint	sint	float		
rgx, ra	32, 32			Y (ro,wo)	Y (ro,wo)	Y (ro,wo)		64
	8, 8	Y	Y	Y	Y			16
	16, 16	Y	Y	Y	Y	Y		32
	32, 32			Y	Y	Y		64
rgb, rgbx	5, 6, 5, 1	Y						16
	5, 5, 5	Y						16
	10, 10, 10, 2	Y						32
rgba	8, 8, 8, 8	Y (ro,wo,rw)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)			32
	16, 16, 16, 16	Y (ro,wo)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		64
	32, 32, 32, 32			Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		128
bgra	8, 8, 8, 8	Y (ro,wo)	Y	Y	Y			32
argb, abgr	8, 8, 8, 8	Y	Y	Y	Y			32
srgb, srgbx	8, 8, 8	Y						24
srgba	8, 8, 8, 8	Y (ro)						32
sbgra	8, 8, 8, 8	Y						32
intensity, luminance	8	Y	Y					8
	16	Y	Y			Y		16
	32					Y		32
depth	16	Y (ro,wo)					2DDEPTH, 2DADEPTH	16
	24	Y						24
	32					Y (ro,wo)		32
depth_stencil	24, 8	Y						32
	32					Y		64

7.1.6 Image Coordinate

Image operations use image coordinates to specify which image element, and for image arrays, which image layer, to access. An image geometry uses either one, two or three coordinates, named `x`, `y` and `z`. These correspond to the width, height, depth and array indices of the image geometry as specified in [Table 9-1 \(p. 199\)](#).

The processing of each image coordinate is controlled by three properties:

- Coordinate normalization mode
- Coordinate addressing mode
- Coordinate filter mode

These properties are specified by a sampler when using an `rdimage` image operation (see [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)). For the `ldimage` and `stimage` image operations, fixed modes are used (see [7.1.6.3 Filter Mode \(p. 211\)](#)). The 1DB image geometry does not support samplers and so cannot be used with the `rdimage` image operation.

7.1.6.1 Coordinate Normalization Mode

The coordinate normalization mode controls how a coordinate value `coord` is converted to an unnormalized coordinate that is used to access an image element. An unnormalized coordinate is a signed value that includes a fractional part. (The pseudo code uses an unspecified floating-point type, but an implementation may use a range reduced signed integer together with a fixed point fractional part.) The conversion depends on the coordinate filter mode (see [7.1.6.3 Filter Mode \(p. 211\)](#)). A coordinate may specify an image element that is outside the range of the corresponding image dimension: the coordinate addressing mode controls how an out of range coordinate is processed (see [7.1.6.2 Addressing Mode \(p. 210\)](#)).

The coordinate normalization mode can be:

unnormalized

An unnormalized coordinate specifies the index of the image element as either a `u32`, `s32` or `f32` data type value:

`u32`

This is always used for `ldimage` and `stimage` image operations which only allow nearest filter mode and unnormalized coordinate normalization mode.

`s32`

This can be used when the sampler for the `rdimage` image operation specifies an unnormalized coordinate normalization mode. For an array index coordinate the nearest filter mode is always used regardless of what is specified by the sampler.

`f32`

This can be used when the sampler for the `rdimage` image operation specifies an unnormalized coordinate normalization mode. It is also used for the array index coordinate for the `rdimage` image operation when the normalized coordinate normalization mode is specified by the sampler, in which case the nearest filter mode is always used regardless of what is specified by the sampler. The coordinate is considered undefined if it has a NaN or Infinity value.

normalized

A normalized coordinate uses a scaled image element index such that the half-open interval $[0.0, 1.0)$ spans the image element index half-open interval of $[0, \text{coord}_{\text{dim}})$ where `coorddim` is the size of the corresponding dimension. It is specified as an `f32` coordinate data type value. The value is multiplied by `coorddim` to determine the image element index. This is used for non-array index coordinates when the sampler for the `rdimage` image operation specifies a normalized coordinate normalization mode. The coordinate is considered undefined if it has a NaN or Infinity value.

A coordinate is converted as follows:

```

normalization(coord) {
    switch (coord_is_array_index ? unnormalized : normalization_mode) {
        case unnormalized:
            switch (coord_data_type) {
                case u32:
                    switch (filter_mode) {
                        case nearest: return float(coord);
                    }
                case s32:
                    switch (coord_is_array_index ? nearest : filter_mode) {
                        case nearest: return float(coord);
                        case linear:   return float(coord) - 0.5;
                    }
                case f32:
                    if (coord_is_nan or coord_is_infinity) return is_undefined;
                    switch (coord_is_array_index ? nearest : filter_mode) {
                        case nearest: return coord;
                        case linear:   return coord - 0.5;
                    }
            }
        case normalized:
            switch (coord_data_type) {
                case f32:
                    if (coord_is_nan or coord_is_infinity) return is_undefined;
                    switch (filter_mode) {
                        case nearest: return coord * coord_dim;
                        case linear:   return (coord * coord_dim) - 0.5;
                    }
            }
    }
}

```

7.1.6.2 Addressing Mode

The coordinate addressing mode controls how out of range coordinates are processed:

`undefined`

The image operation is undefined if the coordinate value is out of range. This mode is always used for all coordinates of `stimage`, and for the non-array coordinates of `ldimage` image operations.

If the coordinates are always known to be inside the image, then using `undefined` can result in improved performance as it allows the implementation to use the most efficient addressing mode. Note that `linear` filter mode can result in coordinates being accessed outside the image even if the coordinates specified to the image operation are inside the image, so using an addressing mode of `undefined` may result in unpredictable values at the edge of the image.

`clamp_to_edge`

Out of range coordinates are clamped to the edge of the image. This mode is always used by the `rdimage` image operation for an array index coordinate regardless of the addressing mode specified by the sampler. It is also always used for an array index coordinate of the `ldimage` image operation.

`clamp_to_border`

If any coordinate used to access an image element is out of range then the border color associated with the channel order of the image is used (see [Table 9–2 \(p. 201\)](#)).

repeat

Out of range coordinates wrap around the image, making the image appear as repeated tiles. It is undefined to specify `repeat` addressing mode unless the normalization mode is normalized.

mirrored_repeat

Out of range coordinates are wrapped in the opposite direction of the previous image repetition, making the image appear as repeated tiles with every other tile a reflection. It is undefined to specify `mirrored_repeat` addressing mode unless the normalization mode is normalized.

The conversions to an integer image element index for non-array index coordinates uses `downi`, whereas `neari` is used for array index coordinates (see [5.18.4 Description of Integer Rounding Modes \(p. 153\)](#)).

The addressing mode is computed as follows:

```
addressing(coord) {
    if (coord_is_undefined) return is_undefined;
    out_of_range = (int_downi(coord) < 0) or (int_downi(coord) > coord_dim - 1);
    if (coord_is_array_index) {
        if ((operation == stimage) and out_of_range) return is_undefined;
        return max(0, min(int_neari(coord), coord_dim - 1));
    }
    if ((normalization_mode == unnormalized) and
        ((addressing_mode == repeat) or (addressing_mode == mirrored_repeat))) return is_undefined;
    if (not out_of_range) return int_downi(coord);
    switch(addressing_mode) {
        case undefined: return is_undefined;
        case clamp_to_edge: return int_downi(max(0, min(coord, coord_dim - 1)));
        case clamp_to_border: return is_border;
        case repeat:
            tile = (coord < 0) ? int_downi(-(-coord / coord_dim) : int_downi(coord / coord_dim);
            return int_downi(coord - (tile * coord_dim));
        case mirrored_repeat:
            mirrored_coord = (coord < 0) ? -coord - 1; : coord;
            tile = int_downi(mirrored_coord / coord_dim);
            mirrored_coord = int_downi(mirrored_coord - (tile * coord_dim));
            if (tile & 1)
                mirrored_coord = (coord_dim - 1) - mirrored_coord;
            }
            return mirrored_coord;
    }
}
```

7.1.6.3 Filter Mode

The coordinate filter mode controls how image elements are selected:

nearest

Specifies that the image element selected is the one with the nearest integral index (in Manhattan distance) that is less than or equal to the specified coordinates. This is also known as point sampling.

linear

Selects a line block of two elements (for 1D and 1DA images), a 2x2 square block of elements (for 2D, 2DA, 2DDEPTH and 2DADEPTH 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. The weighted average is computed for each image element component independently. Note that for image arrays, the weighted average is only computed within the image layer selected by the array index coordinate, not between different image layers. `linear` filter mode is not supported for the 1DB geometry.

The filter mode can result in more than one image element being accessed: these elements are known as texels. In the pseudo code below, each texel is accessed using `load_texel` and `store_texel` which take three image coordinate indices `x_index`, `y_index` and `z_index`. These operations ignore any coordinate indices that are unused by the image geometry (see [Table 9-1 \(p. 199\)](#)). Of the used coordinate indices, if any are *is_undefined*, then the image operation is undefined. For `load_texel`, if any used coordinate index is *is_border* then the border color associated with the channel order of the image is returned (see [Table 9-2 \(p. 201\)](#)). Otherwise, `load_texel` returns the value of the image element with the specified used coordinate indices and `store_texel` stores the value `src` to the image element with the specified used coordinate indices.

`load_texel` converts each memory component of the image element loaded from the memory type to the access type (including conversion from sRGB to linear RGB for the SRGB channel types). Similarly, `store_texel` converts each access component from the access type to the memory type (including conversion from linear RGB to sRGB for the SRGB channel types) before storing in the image element. See [Table 9-3 \(p. 203\)](#) and [7.1.4.1.2 Standard RGB \(s-Form\) Channel Orders \(p. 202\)](#).

`load_texel` and `store_texel` map between memory components and access components as shown in [Table 9-2 \(p. 201\)](#). If the image channel order has fewer than four memory components:

- `load_texel` returns the fixed value from [Table 9-2 \(p. 201\)](#) for any missing memory components
- `store_texel` ignores any access components that have no corresponding memory component

The coordinate properties used by each image operation are:

- `stimage` always uses unnormalized normalization mode, undefined addressing mode and near filter mode.
- `ldimage` always uses unnormalized normalization mode, undefined addressing mode and near filter mode.
- `rdimage` uses the values for normalization mode, addressing mode and filter mode specified by the sampler operand (see [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#)).

The filter mode is computed as follows:

```

nearest (stimage)
    x_index = addressing(normalization(x));
    y_index = addressing(normalization(y));
    z_index = addressing(normalization(z));
    store_texel(x_index, y_index, z_index, src);

nearest (rdimage, ldimage)
    x_index = addressing(normalization(x));
    y_index = addressing(normalization(y));
    z_index = addressing(normalization(z));
    return load_texel(x_index, y_index, z_index);

linear (rdimage)
    x0_index = addressing(normalization(x));
    x1_index = addressing(normalization(x) + 1);
    x_frc   = normalization(x) - floor(normalization(x));
    y0_index = addressing(normalization(y));
    y1_index = addressing(normalization(y) + 1);
    y_frc   = normalization(y) - floor(normalization(y));
    z0_index = addressing(normalization(z));
    z1_index = addressing(normalization(z) + 1);
    z_frc   = normalization(z) - floor(normalization(z));
    switch (geometry) {
        case 1d:
        case 1da:
            return (1 - x_frc) * load_texel(x0_index, y0_index, z0_index)
                  + x_frc      * load_texel(x1_index, y0_index, z0_index);
        case 2d:
        case 2da:
        case 2ddepth:
        case 2ddepth:
            return (1 - x_frc) * (1 - y_frc) * load_texel(x0_index, y0_index, z0_index)
                  + x_frc      * (1 - y_frc) * load_texel(x1_index, y0_index, z0_index)
                  + (1 - x_frc) * y_frc      * load_texel(x0_index, y1_index, z0_index)
                  + x_frc      * y_frc      * load_texel(x1_index, y1_index, z0_index);
        case 3d:
            return (1 - x_frc) * (1 - y_frc) * (1 - z_frc) * load_texel(x0_index, y0_index, z0_index)
                  + x_frc      * (1 - y_frc) * (1 - z_frc) * load_texel(x1_index, y0_index, z0_index)
                  + (1 - x_frc) * y_frc      * (1 - z_frc) * load_texel(x0_index, y1_index, z0_index)
                  + x_frc      * y_frc      * (1 - z_frc) * load_texel(x1_index, y1_index, z0_index)
                  + (1 - x_frc) * (1 - y_frc) * z_frc      * load_texel(x0_index, y0_index, z1_index)
                  + x_frc      * (1 - y_frc) * z_frc      * load_texel(x1_index, y0_index, z1_index)
                  + (1 - x_frc) * y_frc      * z_frc      * load_texel(x0_index, y1_index, z1_index)
                  + x_frc      * y_frc      * z_frc      * load_texel(x1_index, y1_index, z1_index);
        case 1db:
            return not_supported;
    }
}

```

If the coordinate normalization mode is unnormalized (whether `u32`, `s32` or `f32`), the addressing mode is undefined, `clamp_to_edge` or `clamp_to_border` and the filter mode is `nearest`, the image element index must be computed with no loss of precision. For all other combinations, the precision of the computations is implementation defined. To ensure a minimum precision, explicit operations can be used to convert to unnormalized coordinates, and to perform the equivalent of any linear filter mode using component values accessed by image operations that do guarantee a precision.

7.1.7 Image Creation and Image Handles

Each image has a fixed size. The size includes the number of elements for each image layer dimension and number of image layers for image arrays:

- Width size: in elements for one, two and three dimensional image data geometries.
- Height size: in elements for two and three dimensional image data geometries.
- Depth size: in elements for three dimensional image data geometries.
- Array size: in number of image layers for image array geometries.

The HSA runtime can be used to query the image data size and alignment required for an image of a specific size, geometry, format and access permission on a specific agent. This size is implementation dependent for each agent and may include additional padding between the image rows and slices. For example, additional padding may ensure alignment that improves performance.

The row pitch is the size in bytes for a single row, including padding between rows, and must be greater than or equal to the `width * bpp / 8`. For 2D and 3D images, the slice pitch is the size of a single 2D slice, including padding between slices, and must be greater than or equal to `row_pitch * height`.

The HSA runtime can be used to allocate image data of the appropriate size and alignment, that is accessible to image operations executed on a specific agent using a specific access permission.

The HSA runtime can be used to create an opaque image handle by specifying:

- Image geometry
- Image size
- Image format
- Image access permission
- Agent
- Address of image data

An image handle representation is implementation dependent for each agent. The combinations of image geometry, access permission and format supported by an agent are implementation defined, but there is a minimal set that every agent must support (see [Table 9-4 \(p. 207\)](#)). The maximum image size supported for an image geometry, and the total number of created image handles for a specific access permission, is implementation dependent for each agent, but there are minimum limits that all agents must support (see [Appendix A Limits \(p. 393\)](#)). An HSA runtime query is available to obtain the maximum limits supported by an agent.

It is implementation defined if the same image data layout is used for different access permissions to images with the same image geometry, size and format on a specific agent. There is an HSA runtime query to determine if the same data layout is used.

It is undefined to create multiple image handles to the same image data unless:

- The agent is the same.
- The image data was allocated using the HSA runtime such that it is accessible to the agent.

- The image geometry, size and format are the same. The one exception is that if the image format channel type is an s-form it can be the corresponding non-s-form and vice versa (see [7.1.4.1.2 Standard RGB \(s-Form\) Channel Orders \(p. 202\)](#)).
- The image access permission must also match unless the agent uses the same image data layout for all image access permissions with the specified image geometry, size and format.

The HSA runtime can be used to destroy an image handle which reduces the number of created handles. It is undefined to use an image handle after it has been destroyed. The HSA runtime provides operations to convert between a linear image data layout and the implementation defined image data layout, and to copy and erase portions of the image data. See the HSA runtime.

In HSAIL there are three opaque image handle types, `roimg`, `woimg` and `rwimg` (see [Table 6–4 \(p. 81\)](#)). These correspond to the three image access permissions (see [7.1.5 Image Access Permission \(p. 207\)](#)). See [Table 9–5 \(p. 215\)](#).

- A read-only image handle (`roimg`) can only be used to read the image data.
- A write-only image handle (`woimg`) can only be used to write the image data.
- A read-write image handle (`rwimg`) can only be used to both read and write the image data.

Table 9–5 Image Handle Properties

Image Handle Type	Image Access Permission	Image Operations
<code>roimg</code>	<code>ro</code>	<code>rdimage</code> , <code>ldimage</code>
<code>woimg</code>	<code>wo</code>	<code>stimage</code>
<code>rwimg</code>	<code>rw</code>	<code>ldimage</code> , <code>stimage</code>

The only access to the image data referenced by an image handle in a kernel dispatch is through the HSAIL image operations `rdimage`, `ldimage` and `stimage`, not through the memory operations `ld`, `st`, `atomic`, or `atomicnoret`. It is undefined to use an image handle that was created by the HSA runtime with a different access permission than is required by the HSAIL type. It is undefined to use HSAIL image operations on an HSA component for which the image handle was not created, or with an image handle that has an access permission that is not supported by the HSA component for the image's properties. Different HSA components may use different representations for image handles, and their image operations may not be able to access each other's image data allocations. Also see [7.1.10 Image Memory Model \(p. 220\)](#).

Image handle variables can be declared and defined:

- As a global or readonly segment variable declaration or definition, either inside or outside of a function or kernel.
- As an arg segment variable definition in an arg block.
- As a function formal argument definition in the arg segment.
- As a kernel formal argument definition in the kernarg segment.

An image handle type always has a size of 8 bytes and a natural alignment of 8 bytes, but the format is implementation dependent for each agent.

An image handle variable definition in the global or readonly segment can have its properties defined by providing an initializer. For the a global or readonly segment variable definition with the `const` qualifier, an initializer is required. For a global or readonly segment variable without the `const` qualifier, an initializer is optional. Since the representation of image handles and image data is agent specific, it is required that such initialized variables have agent allocation. This ensures that each agent has its own allocation for the variable that is initialized with an image handle for an image with the specified properties using the representation appropriate for that agent. Readonly segment variables are implicitly agent allocation, but the `alloc(agent)` qualifier is required for global segment variables. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

An image initializer is an image handle type prefixed by an underscore, followed by a parenthesized list containing pairs of keyword = value. The variable type must match the initializer image handle type. The geometry of the image and all the properties that apply to that geometry must be specified. The properties can be specified in any order, with no duplications and no properties that do not apply to the specified image geometry.

When a variable definition with an image initializer is added to a program, an image with the specified properties is created for each HSA component that is a member of the program and supports images. The HSA component's image handle is allocated and initialized to reference the corresponding image. The image data is not initialized. When the program is destroyed, both the image and image handle are destroyed.

The image handle variable can be declared as an array. An image handle declared as a global or readonly segment variable array can have an array initializer. If the array is larger than the initializer, then the properties of the remaining elements are undefined. It is an error if the size of the initializer is larger than the array. The size of the array can be omitted and will default to the size of the initializer.

For example, the following defines 3 read-only image handles and 6 read-write image handles:

```
alloc(agent) global_roimg &name0 = _roimg(geometry = 3d,
                                             width = 5, height = 4, depth = 6,
                                             channel_type = unorm_int_101010,
                                             channel_order = rgbx);

decl prog global_roimg &name1;
decl prog global_roimg &ArrayOfroimgs[10];
alloc(agent) global_woimg &name3 = _woimg(geometry = 3d,
                                             width = 5, height = 4, depth = 6,
                                             channel_type = unorm_int_101010,
                                             channel_order = rgbx);

decl prog global_rwimg &namedrwimg12;
decl prog global_rwimg &namedrwimg2;
decl prog global_rwimg &namedrwimg3;
decl prog global_rwimg &ArrayOfrwimgs[10];
alloc(agent) global_rwimg &namedrwimgWithInit[2] =
    { _rwimg(geometry = 3d,
              width = 5, height = 4, depth = 6,
              channel_type = unorm_int_101010,
              channel_order = rgbx),
      _rwimg(geometry = 2d,
              width = 5, height = 4,
              channel_type = unorm_short_555,
              channel_order = rgb)
    };

```

The query operations can be used to query the attributes of an image. See [7.5 Query Image and Query Sampler Operations \(p. 228\)](#).

7.1.8 Sampler Creation and Sampler Handles

Samplers are used to specify how to process image coordinates by the `rdimage` image operation (see [7.1.6 Image Coordinate \(p. 208\)](#)).

The HSA runtime can be used to create an opaque sampler handle by specifying:

- Coordinate normalization mode
- Coordinate addressing mode
- Coordinate filter mode

A sampler handle representation is implementation dependent for each agent. The total number of created sampler handles is implementation dependent for each agent, but there are minimum limits that all agents must support (see [Appendix A Limits \(p. 393\)](#)). An HSA runtime query is available to obtain the maximum limits supported by an agent. The HSA runtime can be used to destroy a sampler handle which reduces the number of created handles. It is undefined to use a sampler handle after it has been destroyed. See the HSA runtime.

In HSAIL there is an opaque sampler handle type `samp` (see [Table 6-4 \(p. 81\)](#)). It is undefined to use the `rdimage` image operation on an HSA component for which the sampler handle was not created. Different HSA components may use different representations for sampler handles.

Sampler handle variables can be declared and defined:

- As a global or readonly segment variable declaration or definition inside or outside of a function or kernel.
- As a arg segment variable definition in an arg block.
- As a function formal argument definition in the arg segment.
- As a kernel formal argument definition in the kernarg segment.

A sampler handle type always has a size of 8 bytes and a natural alignment of 8 bytes, but the format is implementation dependent for each agent.

A sampler handle variable definition in the global or readonly segment can have its properties defined by providing an initializer. For the a global or readonly segment variable definition with the `const` qualifier, an initializer is required. For a global or readonly segment variable without the `const` qualifier, an initializer is optional. Since the representation of sampler handles is agent specific, it is required that such initialized variables have agent allocation. This ensures that each agent has its own allocation for the variable that is initialized with a sampler handle for a sampler with the specified properties using the representation appropriate for that agent. Readonly segment variables are implicitly agent allocation, but the `alloc(agent)` qualifier is required for global segment variables. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

A sampler initializer is `_samp`, followed by a parenthesized list containing pairs of `keyword = value`. The variable type must match the initializer `samp` type. All the properties of a sampler must be specified, in any order, with no duplications. It is an error if unnormalized normalization mode is specified with an addressing mode of `repeat` or `mirrored_repeat`.

When a variable definition with a sampler initializer is added to a program, a sampler with the specified properties is created for each HSA component that is a member of

the program and supports images. The HSA component's sampler handle is allocated and initialized to reference the corresponding sampler. When the program is destroyed, both the sampler and sampler handle are destroyed.

The sampler handle variable can be declared as an array. A sampler handle declared as a global or readonly segment variable array can have an array initializer. If the array is larger than the initializer, then the properties of the remaining elements are undefined. It is an error if the size of the initializer is larger than the array. The size of the array can be omitted and will default to the size of the initializer.

For array image geometries (1DA, 2DA, 2DADEPTH), the array index coordinate ignores the sampler values and is always processed using the unnormalized normalization mode, nearest filter mode and an addressing mode of `clamp_to_edge` but using `neari` instead of `downi` rounding mode (see [7.1.6.3 Filter Mode \(p. 211\)](#)).

Samplers cannot be used with 1DB images which are not supported by the `rdimage` image operation.

An example of a sampler handler is:

```
alloc(agent) global_samp &y1 = _samp(coord = normalized,
                                         filter = nearest,
                                         addressing = clamp_to_edge);
alloc(agent) global_samp &y2[2] = { _samp(coord = unnormalized,
                                         filter = nearest,
                                         addressing = clamp_to_border),
                                         _samp(coord = normalized,
                                         filter = linear,
                                         addressing = mirrored_repeat) };
```

The query operations can be used to query the attributes of a sampler. See [7.5 Query Image and Query Sampler Operations \(p. 228\)](#).

7.1.9 Using Image Operations

The image operations are listed in [Table 9–6 \(p. 219\)](#).

- It is undefined to use an image operation with an image geometry modifier that does not match the geometry of the image. See [Table 9–1 \(p. 199\)](#).
- It is undefined to use the image operations with a combination of image handle type, coordinate type, access type, image geometry and sampler properties not listed in [Table 9–6 \(p. 219\)](#).
- It is undefined to use the image operations on an image with a channel order, channel type and image geometry not specified in [Table 9–4 \(p. 207\)](#).
- It is undefined if the access type of the image operation does not match the access type required by the image's channel type specified in [Table 9–4 \(p. 207\)](#).

Table 9–6 Image Operation Combinations

Image Operation	Image Handle Type	Coordinate Type	Access Type	Sampler			Image Geometry
				coord	filter	addressing	
rdimage	roimg	s32	u32, s32, f32	unnormalized	nearest	undefined, clamp_to_edge, clamp_to_border	1D, 2D, 3D, 1DA, 2DA, 2DDEPTH, 2DADEPTH (1DA, 2DA, 2DADEPTH array index coordinate always treated as unnormalized, clamp_to_edge)
			f32	u32, s32			
			f32		nearest, linear		
			u32, s32	normalized	nearest	undefined, clamp_to_edge, clamp_to_border, repeat, mirrored_repeat	
			f32		nearest, linear		
ldimage	roimg, rwimg	u32	u32, s32, f32	Sampler not allowed (<i>undefined if coordinate not in range 0 to dimension size - 1</i>)			1D, 2D, 3D, 1DA, 2DA, 1DB, 2DDEPTH, 2DADEPTH
stimage	woimg, rwimg						

To access the data in an image, a kernel loads an image handle into a `d` register using a load (`ld`) operation with a source type of `roimg`, `woimg` 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`) or store image (`stimage`) operations.

The differences between the `rdimage` operation and the `ldimage` operation are:

- `rdimage` takes a sampler and therefore supports additional coordinate processing modes.
- The value returned for out-of-bounds references for `rdimage` depends on the sampler.

A sampler is provided to the `rdimage` image operation by using an opaque sampler handle which is loaded into a `d` register with a source type of `samp`.

Both an image and sampler handle in a `d` register can be:

- Moved to another `d` register using the `move (mov)` operation with the corresponding `roimg`, `woimg`, `rwimg` or `samp` type.
- Stored to an `arg` segment variable using a `store (st)` operation with the corresponding `roimg`, `woimg`, `rwimg` or `samp` type. The `arg` segment variable must be:
 - An input actual argument of a call operation in an `arg` block.
 - An output formal argument of a function in a function code block.

This allows image and sampler handles to be passed by value into a function, and returned by value from a function.

A store operation is not allowed on any other segment. This restriction ensures that the actual image or sampler used by an image operation can be statically determined if function calls are inlined. Note, true bindless textures are not supported.

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 kernel argument variable. For image handles, the original variable type (`roimg`, `woimg` or `rwimg`) must match in all operations that use the value. For sampler handles, the original variable and all operations that use the value must specify the sampler handle type (`samp`). These operations include `load (ld)`, `store (st)`, `move (mov)`, the image operations (`rdimage`, `ldimage` and `stimage`), and the image and sampler query operations (`queryimage` and `querysampler`). A function's arguments that are of type `roimg`, `woimg`, `rwimg` or `samp`, must be accessed in the `arg` scope of all calls that invoke it using `load (ld)` and `store (st)` operations with the type of the corresponding function argument.

It is undefined to use an image operation (`rdimage`, `ldimage` and `stimage`) or `imagequery` operation with an image handle value that is not currently created by the HSA runtime for the agent with an image access permission that matches the image type (`roimg`, `woimg` or `rwimg`) of the operation. See [7.1.7 Image Creation and Image Handles \(p. 214\)](#).

It is undefined to use an `rdimage` operation or `samplerquery` operation with an sampler handle value that is not currently created by the HSA runtime for the agent. See [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#).

The address of an image or sampler handle variable 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 handler. Note that this is the address of a handle: in particular, for an image handle it is not the address of the image data.

7.1.10 Image Memory Model

This section maps the HSAIL image operations to the HSA Image Memory Model formally defined in the *HSA Platform System Architecture Specification* section 2.13

Requirement: Images. It also provides an overall informal definition of the memory model.

1. It is undefined to use an image or sampler handler that is invalid:
 - a. It is undefined to access an image using an image or sampler handle that was not created, or was created and subsequently destroyed, by the HSA runtime.
 - b. It is undefined to use an image or sampler handle that was not created by the HSA runtime for the HSA component executing the kernel dispatch.
 - c. It is undefined to use an image handle with an HSAIL type that does not match the access capability used when it was created by the HSA runtime. See [Table 9–5 \(p. 215\)](#).
2. The image elements accessed by an `rdimage` operation with a sampler with a linear filter mode includes all locations accessed to perform the weighted average (see [7.1.6.3 Filter Mode \(p. 211\)](#)).
3. Within a single kernel dispatch:
 - a. It is undefined to use multiple image handles that reference the same image data to access the same image elements unless all accesses are reads. Accesses to image elements using different image handles are not allowed to alias unless all access are reads.
 - b. It is undefined to access the same image element using both image operations (which use the image segment) and memory operations using the global segment, unless all access are reads.
4. Within a single work-item:
 - a. It is undefined to read the same image element that has been written, without the execution of an intervening `memfence_scar_image(wi)` or `memfence_scar_image(wg)` operation (see [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#)).
5. Between different work-items in the same work-group:
 - a. It is undefined for work-item A to read or write the same image element that has been written by work-item B in the same work-group, without B executing a `memfence_scar_image(wg)` operation after the write, followed by A executing a `memfence_scar_image(wg)` before the read (see [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#)).
6. Between different work-items in different work-groups of the same kernel dispatch:
 - a. It is undefined for work-item A to read or write the same image element that has been written by work-item B in a different work-group. The widest memory scope that image elements can be shared is work-group.

7. Between different kernel dispatches or agents:

- a. It is undefined to use the same, or different image handles that reference the same image data, to access the same image elements unless all accesses are reads, or there is intervening synchronization using User Mode Queue packet memory fences (see *HSA Platform System Architecture Specification* section 2.7.1 *Packet header*). Image data sharing between different kernel dispatches and other agents is only at kernel dispatch granularity. The packet fences must specify correctly paired release and acquire, and have matching memory scopes of which both are members.
 - b. The HSA runtime image operations implicitly perform an acquire when they start and a release before they report completion at system memory scope.
 - c. The image segment and global segment are only made coherent at kernel dispatch granularity using the User Mode Queue packet fences.
8. Any access to image data using the global segment must use acquire and release memory ordering at an appropriate memory scope in order to allow sharing. See [6.2 Memory Model \(p. 160\)](#).

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 9–7 Syntax for Read Image Operation

Opcode and Modifiers	Operands
<code>rdimage_v4_1d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, sampler, coordWidth</code>
<code>rdimage_v4_2d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight)</code>
<code>rdimage_v4_3d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight, coordDepth)</code>
<code>rdimage_v4_1da_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordArrayIndex)</code>
<code>rdimage_v4_2da_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight, coordArrayIndex)</code>
<code>rdimage_2dddepth_equiv(n)_destType_imageType_coordType</code>	<code>destR, image, sampler, (coordWidth, coordHeight)</code>
<code>rdimage_2dadepth_equiv(n)_destType_imageType_coordType</code>	<code>destR, image, sampler, (coordWidth, coordHeight, coordArrayIndex)</code>

Explanation of Modifiers
v4: If present, specifies the operation returns 4 components, otherwise only 1 component is returned.
1d, 2d, 3d, 1da, 2da, 2dddepth, 2ddepth: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be 1d (width); 2d or 2dddepth (width and height); 3d (width, height, and depth); 1da (width and array index); or 2da or 2ddepth (width, height and array index). 1db is not supported. See 7.1.3 Image Geometry (p. 199) .
equiv(n): Optional: n is an equivalence class. Used to specify the equivalence class of the image data memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See 6.1.4 Equivalence Classes (p. 160) .
destType: Destination type: u32, s32, or f32. See Table 6-2 (p. 79) .
imageType: Image object type: roimg. See Table 6-4 (p. 81) .
coordType: Source coordinate element type: s32 or f32. See Table 6-2 (p. 79) .

Explanation of Operands (see 4.16 Operands (p. 86))
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. See 7.1.7 Image Creation and Image Handles (p. 214) and 7.1.9 Using Image Operations (p. 218) .
sampler: A source operand d register that contains a value of a sampler object. It is always of type samp. See 7.1.8 Sampler Creation and Sampler Handles (p. 217) and 7.1.9 Using Image Operations (p. 218) .
coordWidth, coordHeight, coordDepth, coordArrayIndex: A source s register, immediate value or WAVESIZE of type coordType that specifies the coordinates being read.

Exceptions (see Chapter 12 Exceptions (p. 285))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see [18.7.3 BRIG Syntax for Image Operations \(p. 374\)](#).

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-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 undefined, clamp_to_edge or clamp_to_border.

1DB images are not supported.

Examples

```

ld_global_roimg $d1, [&roimg1];
ld_kernarg_roimg $d2, [%roimg2];
ld_READONLY_samp $d3, [&samp1];
rdimage_v4_1d_equiv(12)_s32_roimg_f32 ($s0, $s1, $s5, $s3), $d1, $d3, $s6;
rdimage_v4_2d_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d2, $d3, ($s6, $s9);
rdimage_v4_3d_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d2, $d3, ($s6, $s9, $s2);
rdimage_v4_1da_s32_roimg_f32 ($s0, $s1, $s2, $s3), $d1, $d3, ($s6, $s7);
rdimage_v4_2da_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d1, $d3, ($s6, $s9, $s12);
rdimage_2dddepth_s32_roimg_f32 $s0, $d2, $d3, ($s6, $s9);
rdimage_2dadept_s32_roimg_f32 $s0, $d2, $d3, ($s6, $s9, $s10);

```

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 9–8 Syntax for Load Image Operation

Opcode and Modifiers	Operands
<code>ldimage_v4_1d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, coordWidth</code>
<code>ldimage_v4_2d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, (coordWidth, coordHeight)</code>
<code>ldimage_v4_3d_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, (coordWidth, coordHeight, coordDepth)</code>
<code>ldimage_v4_1da_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, (coordWidth, coordArrayIndex)</code>
<code>ldimage_v4_2da_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, (coordWidth, coordHeight, coordArrayIndex)</code>
<code>ldimage_v4_1db_equiv(n)_destType_imageType_coordType</code>	<code>(destR, destG, destB, destA), image, coordByteIndex</code>
<code>ldimage_2dddepth_equiv(n)_destType_imageType_coordType</code>	<code>destR, image, (coordWidth, coordHeight)</code>
<code>ldimage_2dadept_equiv(n)_destType_imageType_coordType</code>	<code>destR, image, (coordWidth, coordHeight, coordArrayIndex)</code>

Explanation of Modifiers
v4: If present, specifies the operation returns 4 components, otherwise only 1 component is returned.
1d, 2d, 3d, 1da, 2da, 1db, 2ddepth, 2ddepth: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be 1d or 1db (width); 2d or 2ddepth (width and height); 3d (width, height, and depth); 1da (width and array index); or 2da or 2ddepth (width, height and array index). See 7.1.3 Image Geometry (p. 199) .
equiv(<i>n</i>): Optional: <i>n</i> is an equivalence class. Used to specify the equivalence class of the image data memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See 6.1.4 Equivalence Classes (p. 160) .
destType: Destination type: u32, s32, or f32. See Table 6-2 (p. 79) .
imageType: Image object type: roimg, rwimg. See Table 6-4 (p. 81) .
coordType: Source coordinate element type: u32. See Table 6-2 (p. 79) .

Explanation of Operands (see 4.16 Operands (p. 86))
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. See 7.1.7 Image Creation and Image Handles (p. 214) and 7.1.9 Using Image Operations (p. 218) .
coordWidth, coordHeight, coordDepth, coordArrayIndex: A source s register, immediate value or WAVESIZE of type coordType that specifies the coordinates being read.

Exceptions (see Chapter 12 Exceptions (p. 285))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see [18.7.3 BRIG Syntax for Image Operations \(p. 374\)](#).

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*, *filter* = *nearest* and *address_mode* = *undefined*. It is undefined if a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0).

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; for ldimage it is undefined.

For all geometries, coordinates are in elements.

Examples

```
ld_global_rwimg $d1, [&rwimg1];
ld_kernarg_roimg $d2, [%roimg2];
ldimage_v4_1d_equiv(12)_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, $s4;
ldimage_v4_2d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s4, $s5);
ldimage_v4_3d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s4, $s5, $s6);
ldimage_v4_1da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s4, $s5);
ldimage_v4_2da_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s4, $s1, $s2);
ldimage_v4_1db_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2, $s4;
ldimage_2dddepth_f32_rwimg_u32 $s1, $d1, ($s4, $s5);
ldimage_2dadepth_f32_rwimg_u32 $s1, $d1, ($s4, $s5, $s6);
```

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 9–9 Syntax for Store Image Operation

Opcode and Modifiers	Operands
stimage_v4_1d_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, coordWidth
stimage_v4_2d_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight)
stimage_v4_3d_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight, coordDepth)
stimage_v4_1da_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, (coordWidth, coordArrayIndex)
stimage_v4_2da_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight, coordArrayIndex)
stimage_v4_1db_equiv(n)_srcType_imageType_coordType	(srcR, srcG, srcB, srcA), image, coordArrayIndex
stimage_2dddepth_equiv(n)_srcType_imageType_coordType	srcR, image, (coordWidth, coordHeight)
stimage_2dadepth_equiv(n)_srcType_imageType_coordType	srcR, image, (coordWidth, coordHeight, coordArrayIndex)

Explanation of Modifiers
v4: If present, specifies the operation takes 4 components, otherwise only 1 component is taken.
1d, 2d, 3d, 1da, 2da, 1db, 2ddepth, 2ddepth: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be 1d or 1db (width); 2d or 2ddepth (width and height); 3d (width, height, and depth); 1da (width and array index); or 2da or 2ddepth (width, height and array index). See 7.1.3 Image Geometry (p. 199) .
equiv(n): Optional: n is an equivalence class. Used to specify the equivalence class of the image data memory locations being accessed. If omitted, class 0 is used indicating may alias any memory location. See 6.1.4 Equivalence Classes (p. 160) .
srcType: Source type: u32, s32, or f32. See Table 6-2 (p. 79) .
imageType: Image object type: woimg, rwimg. See Table 6-4 (p. 81) .
coordType: Source coordinate element type: u32. See Table 6-2 (p. 79) .

Explanation of Operands (see 4.16 Operands (p. 86))
srcR, srcG, srcB, srcA: Source data. Must be an s register, immediate value or WAVESIZE.
image: A source operand d register that contains a value of an image object of type imageType. See 7.1.7 Image Creation and Image Handles (p. 214) and 7.1.9 Using Image Operations (p. 218) .
coordWidth, coordHeight, coordDepth, coordArrayIndex: A source s register, immediate value or WAVESIZE of type coordType that specifies the coordinates being read.

Exceptions (see Chapter 12 Exceptions (p. 285))
Invalid address exceptions are allowed. May generate a memory exception if image data is unaligned.

For BRIG syntax, see [18.7.3 BRIG Syntax for Image Operations \(p. 374\)](#).

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 write-only or read-write image specified by source operand *image* at integer coordinates given by source operands *coordWidth*, *coordHeight*, *coordDepth*, *coordArrayIndex*, and *coordByteIndex*.

It is undefined if a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0).

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 all 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_woimg $d1, [&roimg1];
ld_global_rwimg $d2, [&rwimg1];
stimage_v4_1d_equiv(12)_f32_woimg_u32 ($s1, $s2, $s3, $s4), $d1, $s4;
stimage_v4_2d_f32_woimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s4, $s5);
stimage_v4_3d_f32_woimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s4, $s5, $s6);
stimage_v4_1da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s4, $s5);
stimage_v4_2da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s4, $s5, $s6);
stimage_v4_1db_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, $s4;
stimage_2dddepth_f32_rwimg_u32 $s1, $d2, ($s4, $s5);
stimage_2ddepth_f32_rwimg_u32 $s1, $d2, ($s4, $s5, $s6);
st_arg_rwimg $d2, [%rwimg_arg1];
```

7.5 Query Image and Query Sampler Operations

The query image and query sampler operations query an attribute of an image object or a sampler object.

7.5.1 Syntax

Table 9–10 Syntax for Query Image and Query Sampler Operations

Opcode	Operands
<code>queryimage_geometry_imageProperty_destType_imageType</code>	<code>dest, image</code>
<code>querysampler_samplerProperty_destType</code>	<code>dest, sampler</code>

Explanation of Modifiers

geometry: Image geometry: 1d, 2d, 3d, 1da, 2da, 1db, 2dddepth, 2ddepth. See [7.1.3 Image Geometry \(p. 199\)](#).

imageProperty: Image property: width, height, depth, array, channelorder, chanelltype. height only allowed if *geometry* is 2D, 3D, 2DA, 2DDEPTH or 2DADEPTH; depth only allowed if *geometry* is 3D; array only allowed if *geometry* is 1DA, 2DA or 2DADEPTH. See [Table 9–11 \(p. 229\)](#).

samplerProperty: Sampler property: addressing, coord, filter. See [Table 9–12 \(p. 229\)](#).

destType: Destination type: u32. See [Table 6–2 \(p. 79\)](#).

imageType: Image object type: roimg, woimg, rwimg. See [Table 6–4 \(p. 81\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination register of type u32.

image: A source operand d register that contains a value of an image object of type *imageType*. See [7.1.7 Image Creation and Image Handles \(p. 214\)](#) and [7.1.9 Using Image Operations \(p. 218\)](#).

sampler: A source operand d register that contains a value of a sampler object. It is always of type samp. See [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#) and [7.1.9 Using Image Operations \(p. 218\)](#).

Exceptions (see Chapter 12 Exceptions (p. 285))
--

No exceptions are allowed.

For BRIG syntax, see [18.7.3 BRIG Syntax for Image Operations \(p. 374\)](#).

7.5.2 Description

Each query returns a 32-bit value giving a property of the source:

Table 9–11 Query Image Properties

Image Query	Returns
width	Image width in elements. Allowed for all image geometries.
height	Image height in elements. Only allowed for 2D, 3D, 2DA, 2DDEPTH or 2DADEPTH image geometries.
depth	Image depth in elements. Only allowed for 3D image geometry.
array	The number of image layers. Only allowed for 1DA, 2DA or 2DADEPTH image geometries.
channelorder	An image channel order property encoded as an integer according to 18.3.10 BrigImageChannelOrder (p. 323) .
changetype	An image channel type property encoded as an integer according to 18.3.11 BrigImageChannelType (p. 324) .

Table 9–12 Query Sampler Properties

Sampler Query	Returns
addressing	A sampler addressing mode property encoded as an integer according to 18.3.25 BrigSamplerAddressing (p. 331) . If undefined was specified when the sampler was initialized, it is implementation defined what addressing mode is returned. It may be any of the addressing modes, including undefined.
coord	A sampler coordinate property encoded as an integer according to 18.3.26 BrigSamplerCoordNormalization (p. 331) .
filter	A sampler filter property encoded as an integer according to 18.3.27 BrigSamplerFilter (p. 332) .

Examples

```
ld_global_rwimg $d1, [&rwimg1];
ld_kernarg_roimg $d2, [%roimg2];
ld_kernarg_woimg $d3, [%woimg2];
ld_READONLY_samp $d4, [&samp1];
queryimage_1d_width_u32_rwimg $s1, $d1;
queryimage_2d_height_u32_rwimg $s0, $d1;
queryimage_3d_depth_u32_rwimg $s0, $d1;
queryimage_1da_array_u32_roimg $s1, $d2;
queryimage_2da_channelorder_u32_roimg $s0, $d2;
queryimage_1db_channeltype_u32_roimg $s0, $d2;
queryimage_2ddepth_channeltype_u32_woimg $s0, $d3;
querysampler_addressing_u32 $s0, $d4;
querysampler_coord_u32 $s0, $d4;
querysampler_filter_u32 $s0, $d4;
```

Chapter 8

Branch Operations

Like many programming languages, HSAIL supports branch operations that can alter the control flow.

8.1 Syntax

Table 10–1 Syntax for Branch Operations

Opcode and Modifier	Operands
<code>br</code>	<code>label</code>
<code>cbr_width_b1</code>	<code>src, label</code>
<code>sbr_width_uLength</code>	<code>src [labelList]</code>

Explanation of Modifiers

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. The width modifier specifies the result uniformity of the target for branches. All active work-items in the same slice are guaranteed to branch to the same target. If the width modifier is omitted, it defaults to `width(1)`, indicating each active work-item can branch independently. See [2.12.2 Using the Width Modifier with Control Transfer Operations \(p. 29\)](#).

`Length`: 32, 64.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`src`: Source. Can be a register, immediate value, or `WAVESIZE`.

`label`: Must be an identifier of a label in the same code block as the branch operation.

`labelList`: Must be a comma-separated list of one or more label identifiers that are all in the same code block as the branch operation.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.4 BRIG Syntax for Branch Operations \(p. 374\)](#).

8.2 Description

The label or labels specified in the branch operation must be in the code block, which includes any nested arg blocks, of the kernel or function containing the branch operation. However, the label definition can either be lexically before or after the branch operation. For restrictions on using branches with respect to arg blocks see [10.2 Function Call Argument Passing \(p. 254\)](#).

br

An unconditional branch which transfers control to the label specified.

cbr

A conditional branch which transfers execution to the label specified if the condition value in *src* is true (non-zero), otherwise will *fall through* and execution will continue with the next operation after the cbr operation. *src* must be of type b1.

Since a conditional branch can potentially transfer to more than one target, it can result in control flow divergence which can introduce a performance issue. The width modifier can be used to specify properties about the control flow divergence that may result in the finalizer producing more efficient code. See [2.12 Divergent Control Flow \(p. 26\)](#).

sbr

A switch branch which transfers control to the label in the *labelList* that corresponds to the index value in *src*. If the index value is 0 then the first label is selected, if 1 then the second label, and so forth. It is undefined if the number of labels in *labelList* is less than or equal to the index value. *src* can either be of type u32 or u64.

Since a switch branch can potentially transfer to more than one target, it can result in control flow divergence which can introduce a performance issue. The width modifier can be used to specify properties about the control flow divergence that may result in the finalizer producing more efficient code. See [2.12 Divergent Control Flow \(p. 26\)](#).

It is implementation-defined how a switch branch is finalized to machine instructions. For example: by a cascade of compare and conditional branches; by an indirect branch through a jump table; or a combination of these approaches. The performance of switch branches can therefore potentially be slow for long label lists.

Examples

```
br @label1;  
  
cbr_b1 $c0, @label1;  
cbr_width(2)_b1 $c0, @label2;  
cbr_width(all)_b1 $c0, @label3;  
  
sbr_u32 $s1 [@label1, @label2, @label3];  
sbr_width(2)_u32 $s1, [@label1, @label2, @label3];  
sbr_width(all)_u32 $s1, [@label1, @label2, @label3];  
  
// ...  
@label1:  
// ...  
@label2:  
// ...  
@label3:  
// ...
```


Chapter 9

Parallel Synchronization and Communication Operations

This chapter describes operations used for cross work-item communication.

9.1 Barrier Operations

The `barrier` and `wavebarrier` operations are used to synchronize work-item execution in a work-group and wavefront respectively.

9.1.1 Syntax

Table 11–1 Syntax for Barrier Operations

Opcode and Modifiers
<code>barrier_width</code>
<code>wavebarrier</code>

Explanation of Modifiers

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the communication uniformity among the work-items of a work-group. If omitted, defaults to `width(all)`. See the Description below.

Exceptions (see Chapter 12 Exceptions (p. 285))

No exceptions are allowed.

For BRIG syntax, see [18.7.5 BRIG Syntax for Parallel Synchronization and Communication Operations \(p. 375\)](#).

9.1.2 Description

The `barrier` and `wavebarrier` operations are execution barriers. See [9.3 Execution Barrier \(p. 246\)](#).

The `barrier` operation supports the `width` modifier:

width

A barrier operation can have an optional width modifier that can specify the communication uniformity (see [2.12 Divergent Control Flow \(p. 26\)](#)). If omitted it defaults to `width(all)`. For example, a `barrier_width(n)` can be performed only between the `n` work-items in the same slice. There is no requirement for the work-items in other slices of the same work-group to participate in the barrier at the same time, and no guarantees are made in this respect, provided all work-items of the same work-group do eventually execute it (due to the work-group execution uniform requirement).

If an implementation has a frontend size that is greater than or equal to `n`, it is free to optimize the ISA generated for the barrier when the gang-scheduled execution of work-items in wavefronts will ensure execution synchronization of the communicating work-items. However, even if the barrier is optimized, synchronizing atomic memory operations cannot be moved over the barrier location.

An implementation is allowed to ignore the width modifier and always synchronize execution with all work-items of the work-group.

See also [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#).

Examples

```
barrier;
barrier_width(64);
barrier_width(WAVESIZE);
wavebarrier;
```

9.2 Fine-Grain Barrier (fbar) Operations

9.2.1 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 module scope or in function scope (see [4.3.9 Fbarrier \(p. 52\)](#)). 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 [2.8.4 Memory Segment Access Rules \(p. 20\)](#)): 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.

9.2.2 Syntax

Table 11–2 Syntax for fbar Operations

Opcodes	Operands
<code>initfbar</code>	<code>src</code>
<code>joinfbar_width</code>	<code>src</code>
<code>waitfbar_width</code>	<code>src</code>
<code>arrivefbar_width</code>	<code>src</code>
<code>leavefbar_width</code>	<code>src</code>
<code>releasefbar</code>	<code>src</code>
<code>ldf_u32</code>	<code>dest, fbarrierName</code>

Explanation of Modifier

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the execution uniformity among the work-items of a work-group. If `n` is specified, it must be a multiple of `WAVESIZE`. If the width modifier is omitted, it defaults to `width(WAVESIZE)`. See [2.12 Divergent Control Flow \(p. 26\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

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 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.5 BRIG Syntax for Parallel Synchronization and Communication Operations \(p. 375\)](#).

9.2.3 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 waitfbaroperation (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.

Every fbarrier that has been initialized must be released by an releasefbar operation. Once released, the fbarrier is no longer considered initialized.

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 waveform execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)). This implies that all active work-items of a waveform must be members of the same fbarriers.

`joinfbar` conceptually atomically increments the `member_count` for the waveform.

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

Is an execution barrier, see [9.3 Execution Barrier \(p. 246\)](#).

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.

This operation (which includes the value of the *src* operand) must be waveform execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)). This implies that all active work-items of a waveform arrive at an `waitfbar` together.

`waitfbar` conceptually atomically increments the `arrive_count` for the waveform, and adds the waveform to the `wait_set`. It then atomically checks and waits until the `arrive_count` equals the `member_count`, at which point any waveforms 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

Is an execution barrier, see [9.3 Execution Barrier \(p. 246\)](#).

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.

This operation (which includes the value of the *src* operand) must be waveform execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)). This implies that all active work-items of a waveform arrive at an `arrivefbar` together.

`arrivefbar` conceptually atomically increments the `arrive_count` for the waveform, and checks if the `arrive_count` equals the `member_count`. If it does, then atomically any waveforms 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 execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)). 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`.

1df

Places the address of an fbarrier into the destination *dest*. The address has work-group persistence (see [2.8.4 Memory Segment Access Rules \(p. 20\)](#)) and the value can only be used in work-items that belong to the same work-group as the work-item that executed the `1df` operation. The compound type *dest* is always `u32` regardless of the machine model (see [2.9 Small and Large Machine Models \(p. 24\)](#)). The value returned can be used with fbarrier operations to specify which fbarrier they are to operate on.

9.2.4 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.
- A preceding release memory operation must not be moved (by the implementation) after a `waitfbar` or `arrivefbar` operation by the same work-item. An acquire memory operation must not be moved (by the implementation) before a preceding `waitfbar` or `arrivefbar` operation by the same work-item. However, a non-synchronizing or ordinary memory operation can be moved (by the implementation) across a `waitfbar` or `arrivefbar` operation. See [6.2 Memory Model \(p. 160\)](#).

Because a `waitfbar` or `arrivefbar` ensures that execution of all work-items of a work-group are synchronized, this also ensures:

- that all work-items that are members of the fbarrier perform any preceding release memory operation before any work-item that is performing a `waitfbar` that are member of the fbarrier continues execution after the `waitfbar`,
- and that any acquire memory operation after a `waitfbar` will make all values made visible by release operations performed by any work-item that is a member of the fbarrier before the `waitfbar` or `arrivefbar` that allowed the `waitfbar` to continue.

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.

9.2.5 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;
06: joinfbar %fb1; // start with all work-items
07: barrier;
08: if (cond1) { // cond1 must be WAVESIZE uniform
09:   ...
10:   if (cond2) { // cond2 must be WAVESIZE uniform
11:     ...
12:     memfence_screl_global(sys);
13:     waitfbar %fb1; // fb1 only has work-items for which
                      // cond1 && cond2 is true as other
                      // work-items have left on
                      // lines 18 and 21.
14:     memfence_scacq_global(sys);
15:     ...
16:     leavefbar %fb1;
17:   } else {
18:     leavefbar %fb1;
19:   }
20: } else {
21:   leavefbar %fb1;
22: }
23: barrier;
24: if (workitemflatid_u32 == 0) {
25:   releasefbar %fb1;
26: }
```

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;
09: joinfbar %fb0; // fb0 has all work-items of work-group
10: barrier;
11: if (cond1) { // cond1 must be WAVESIZE uniform
12:     ...
13:     if (cond2) { // cond2 must be WAVESIZE uniform
14:         joinfbar %fb1;
15:         waitfbar %fb0; // wait for all work-items to either
                           // join fb1 on line 14 or arrive at
                           // line 23 or 26
16:     ...
17:     memfence_screl_global(sys);
18:     waitfbar %fb1; // fb1 only has work-items for which
                           // cond1 && cond2 is true
19:     memfence_scacq_global(sys);
20:     ...
21:     leavefbar %fb1;
22: } else {
23:     waitfbar %fb0;
24: }
25: } else {
26:     waitfbar %fb0;
27: }
28: leavefbar %fb0;
29: barrier;
30: if (workitemflatid_u32 == 0) {
31:     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 waveform executions to overlap.

```

kernel producerConsumer(data_item_count)
{
    // Declare the fbarriers.
    fbarrier %produced_fb;
    fbarrier %consumed_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;

    // 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;

    // Ensure all producers and consumers are in the same waveform
    // so that the fbarrier operations are waveform uniform.
    producer = ((workitemflatid_u32 / WAVESIZE) & 1) == 1;

    if (producer) {
        for (i = 1 to data_item_count) {
            // 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 [%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.
            memfence_screl_group(wg);
            arrivefbar [%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 [%consumed_fb];

        for (j = 1 to data_item_count) {
            // 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 [%produced_fb];
            memfence_scacq_group(wg);

            // Consumer reads the new data

            // Only need to tell producer have read data if there is
            // another value to be produced.
            if (j != data_item_count) {
                // 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 [%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;

// 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 %fb;
arrivefbar %fb;
leavefbar %fb;
releasefbar %fb;
ldf_u32 $s0, %fb;
joinfbar $s0;

```

9.3 Execution Barrier

A barrier operation is used to synchronize the execution of the work-items that participate in an associated execution barrier instance:

- For the `barrier` operation the participating work-items are those that are members of the same work-group and each work-group has a distinct execution barrier instance per `barrier` operation.
- For the `wavebarrier` operation the participating work-items are those that are members of the same wavefront and each wavefront has a distinct execution barrier instance per `wavebarrier` operation.
- For the `waitfbar` and `arrivefbar` operations the participating work-items are those that are members of the specified `fbarrier` and each work-group has a distinct execution barrier instance per `fbarrier` definition.

An execution barrier instance is satisfied when all participating work-items have executed a barrier operation that specifies the execution barrier instance.

A communication operation is one that can be used to transfer information between different work-items. These operations are:

- atomic memory (see [6.5 Atomic Memory Operations \(p. 179\)](#))
- memfence (see [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#))
- signals (see [6.8 Notification \(signal\) Operations \(p. 187\)](#))
- cross-lane operations (see [Chapter 9 Parallel Synchronization and Communication Operations \(p. 235\)](#))
- barrier and wavebarrier (see [9.1 Barrier Operations \(p. 235\)](#))
- fbarriers (see [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#))
- clock (see [11.4 Miscellaneous Operations \(p. 280\)](#))
- cleardetectexcept, getdetectexcept and setdetectexcept (see [11.2 Exception Operations \(p. 274\)](#))
- or calls to functions that contain any of these (see [Chapter 10 Function Operations \(p. 253\)](#)).

A work-item that executes a barrier, wavebarrier or waitfbar barrier operation must not proceed with execution until the associated execution barrier instance is satisfied. Within a work-item:

- All communication operations, except relaxed atomics, that proceed the barrier operation in program order, must have completed execution and be visible to other work-items and other agents, before executing the barrier operation.
- All communication operations, except relaxed atomics, that follow the barrier operation in program order must be executed and become visible after the execution barrier instance specified by the barrier operation has been satisfied.

A work-item that executes an arrivefbar barrier operation can proceed with execution without waiting for the specified execution barrier instance to be satisfied. However, the work-item must not execute any barrier operation that specifies the same execution barrier instance until it has been satisfied. Within a work-item:

- All communication operations, except relaxed atomics, that proceed the barrier operation in program order, must have completed execution and be visible to other work-items and other agents, before executing the barrier operation.
- All communication operations, except relaxed atomics, that follow the barrier operation in program order must be executed and become visible after the barrier operation has executed. Note, the execution barrier instance does not have to be satisfied in this case.

For memory related operations, visibility is defined by the memory model (see [6.2 Memory Model \(p. 160\)](#)). Because a barrier operation ensures that execution of all participating work-items are synchronized and prevents synchronizing memory operations from being reordered, this also ensures:

- that all participating work-items perform any preceding release memory operation before any of them continue execution after the barrier,
- and that any acquire memory operation after the barrier, executed by a work-item participating in the barrier, will make all values made visible by release operations performed before the barrier by any work-item that is participating in the same barrier.

For non-memory related communicating operations, visibility is defined as the side effects of the operation have been completed.

Note, a relaxed atomic and ordinary memory operation can be moved (by the implementation) across a barrier operation in either direction unless prevented by an intervening synchronizing memory operation. It is therefore important to use memory fences and other synchronizing memory operations to avoid undefined behavior due to data races. See [6.2 Memory Model \(p. 160\)](#).

Operations not otherwise specified above can be moved across a barrier since their execution order is not detectable from other work-items or other agents.

Therefore, a barrier operation is always required to be execution uniform for the participating work-items: all participating work-items must either execute it, or not execute it. The result is undefined if a barrier operation is used in divergent code with respect to the participating work-items. 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. See [2.12 Divergent Control Flow \(p. 26\)](#).

A barrier operation can be used in a loop provided the loop introduces no divergent control flow with respect to the participating work-items. This requires that all participating work-items execute the loop the same number of iterations.

The number of work-items participating in a barrier operation may be less than or equal to the waveform size either because the operation is `wavebarrier` or is `barrier` when the work-group size is less than or equal to the waveform size. In such cases all participating work-items will be members of the same waveform, and an implementation is free to optimize the ISA generated for the barrier when the gang-scheduled execution of work-items in waveforms will ensure execution synchronization. However, even if such an optimization is performed, synchronizing atomic memory operations cannot be moved over the barrier location.

Note it is undefined to omit a barrier operation and simply rely on gang scheduling to ensure execution synchronization. If execution synchronization is required, even if the number of participating work-item is less than or equal to the waveform size, a barrier operation must be used. The implementation should automatically produce optimized code for such barriers. The `requiredworkgroupsize` and `maxflatworkgroupsize` control directives (see [13.4 Control Directives for Low-Level Performance Tuning \(p. 295\)](#)) can be used to specify the work-group size. This can allow the implementation to optimize the barrier operation when the size is less than or equal to the implementation's waveform size.

9.4 Cross-Lane Operations

These operations perform work across lanes in a waveform. These operations only apply to active work-items within a waveform (see [2.5 Active Work-Groups and Active Work-Items \(p. 10\)](#)).

9.4.1 Syntax

Table 11–3 Syntax for Cross-Lane Operations

Opcodes	Operands
<code>activelaneCount_width_u32_b1</code>	<code>dest, src</code>
<code>activelaneId_width_u32</code>	<code>dest</code>
<code>activelaneMask_v4_width_b64_b1</code>	<code>(dest0, dest1, dest2, dest3), src</code>
<code>activelaneshuffle_width_bLength</code>	<code>dest, src, laneId, identity, useIdentity</code>

Explanation of Modifier

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the execution uniformity among the work-items of a work-group. Each active lane in a wavefront can have different values for the source operands, and produce a different value, regardless of the width modifier. If the width modifier is omitted, it defaults to `width(1)`, indicating each lane of the wavefront can be independently active or inactive. See [2.12 Divergent Control Flow \(p. 26\)](#).

`Length`: 1, 32, 64, 128.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest, dest0, dest1, dest2, dest3`: Destination register.

`src, laneId, identity, useIdentity`: Sources. Can be a register, an immediate value, or `WAVESIZE`.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.5 BRIG Syntax for Parallel Synchronization and Communication Operations \(p. 375\)](#).

9.4.2 Description

activelaneCount

Counts the number of active work-items in the current wavefront that have a non-zero source `src` and puts the result in `dest`. The operation returns a value in the range 0 to `WAVESIZE`.

`src` is treated as a `b1` and `dest` is treated as a `u32`.

activelaneid

Sets the destination *dest* in each active work-item to the count of the number of earlier (in flattened work-item order) active work-items within the same wavefront. The result will be in the range 0 to WAVESIZE – 1.

dest is treated as a u32.

Because *activelaneid* gives each active work-item in the wavefront a unique value, it is often used in compaction. It can be thought of as a prefix sum of the number of active work-items in the current wavefront.

activelanemask

Returns a bit mask in a vector of four d registers that shows which active work-items in the wavefront have a non-zero source *src*. The affected bit position within the registers of *dest* corresponds to each work-item's lane ID. The first register covers lane IDs 0 to 63, the second register 64 to 127, and so on. Any bits corresponding to lane IDs that are greater than or equal to the actual implementations wavefront size must be set to 0.

src is treated as a b1. *dest0*, *dest1*, *dest2* and *dest3* are a vector of four registers each treated as a b64.

activelanesshuffle

If the lane *laneId* (in the same wavefront) is inactive or *useIdentity* is 1, the value in *identity* is transferred to *dest*. Otherwise, the value in *src* of the lane (in the same wavefront) specified by *laneId* is transferred to *dest*. *laneId* must be between 0 and WAVESIZE – 1, otherwise the result is undefined.

src, *identity* and *dest* are treated as a b type of size *Length*; *laneId* is treated as a u32; and *useIdentity* is treated as a b1.

If a lane is not active, it does not receive a value.

It is valid for an active lane to specify itself as the sending lane.

It is valid for multiple active lanes to specify the same active lane as the sending lane.

Conceptually the *dest* operands are updated in parallel, using values for the *src*, *laneId*, *identity* and *useIdentity* operands prior to executing the *activelanesshuffle* operation. This allows any of the source operands and destination operands to be the same register.

See this pseudocode:

```
type result[WAVESIZE];
for(l = 0; l < WAVESIZE; ++l) {
    result[l] = identity;
    if (lane[l].active &&
        !lane[l].useIdentity &&
        lane[lane[l].laneId % WAVESIZE].active) {
        result[l] = lane[lane[l].laneId % WAVESIZE].src;
    }
}
for(l = 0; l < WAVESIZE; ++l) {
    if (lane[l].active) lane[l].dest = result[l];
}
```

Examples

```
activelaneid_u32 $s1, $c0;
activelaneid_u32 $s1;
activelaneid_width(WAVESIZE)_u32 $s1;
activelanemask_v4_b64_b1 ($d0, $d1, $d2, $d3), $c0;
activelanesshuffle_b32 $s1, $s2, $s2, $s3, $c1;
activelanesshuffle_b64 $d1, $d2, 0, 0, 0;
activelanesshuffle_width(ALL)_b128 $q1, $q2, $s2, $q3, $c1;
```


Chapter 10

Function Operations

This chapter describes how to use functions in HSAIL and the related operations.

10.1 Functions in HSAIL

Like other programming languages, HSAIL provides support for user functions. A call operation transfers control to the start of the code block of the user function. Once the function's code block has completed execution, either by reaching the end or by executing a `ret` operation, control is transferred back to the operation immediately after the call operation.

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 frontend (not one address per work-item) is desirable.

Implementations should map the abstractions into appropriate hardware.

Function definitions cannot be nested, but functions can be called recursively.

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()()
{
    { //start argument scope
        call &foo()();
    } //end 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 variables to hold arguments

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;
};

kernel &main()
{
    // ...
    { //start argument scope
        arg_f32 %a;
        arg_f32 %b;
        arg_f32 %res;

        // Fill in the arguments
        st_arg_f32 4.0f, [%a];
        st_arg_f32 $s0, [%b];
        call &compare(%res)(%a, %b);
        ld_arg_f32 $s0, [%res];
    } // End argument scope
    // ...
};
```

The function header specifies the output formal argument, followed by the list of input formal arguments. The call operation specifies a corresponding output actual argument, followed by a list of input actual arguments.

10.1.3 Functions That Do Not Return a Result

Functions that do not return a result are declared with an empty output arguments list:

```
function &foo() (arg_u32 %in)
{ // does not return a value
    ret;
};
```

10.2 Function Call Argument Passing

The argument values passed in and out of a call to a function are termed the actual arguments. Operations in the function definition code block access the actual argument values using the formal arguments of the function definition.

Actual argument definitions are variable definitions in an arg block that specify the arg segment. Formal argument definitions are variable definitions in the function header that specify the arg segment. Variable declaration and definitions that specify the arg segment cannot appear in any other place. See [4.3.6 Arg Block \(p. 47\)](#) and [4.3.3 Function \(p. 43\)](#).

A function specifies a list of zero or more output formal arguments and a list of zero or more input formal arguments. A call operation provides a corresponding list of zero or more output actual arguments and zero or more input actual arguments.

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. For example, by defining a function scope private segment variable. Later versions might allow additional output parameters.

A function can declare an arbitrary number of formal arguments. 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.

Actual arguments are passed into and out of a call to a function using an arg block together with a call operation.

Arguments are pass-by-value. This includes arguments that are defined as arrays.

Within an arg block:

- There are zero or more actual argument definitions.
- Operations to assign values to actual arguments used as input formal arguments of the function being called.
- Exactly one call operation that uses those actual arguments.
- Operations to retrieve a value from the actual argument used as the output formal argument of the function being called.
- In addition, an arg block can have other operations including control flow and label definitions.

Actual argument, and formal argument identifiers must start with a percent (%) sign.

Actual arguments have argument scope which starts from the point of definition to the end of the enclosing arg block, and their lifetime extends to the end of the enclosing arg block. An argument scope name hides a definition with the same name outside the arg block in the enclosing function scope. Each arg block defines a distinct argument scope: the same name can be used for actual arguments in different arg blocks.

Function definition formal arguments have function scope which starts from the point of definition in the function header to the end of the function's code block. See [4.6.2 Scope \(p. 63\)](#).

Each work-item can set a different value into its own arg segment variables. Arg segment variables cannot be read or written by other work-items.

Arg blocks cannot be nested.

Arg blocks can include multiple basic blocks.

It is an error to branch into or out of an arg block.

It is not valid to use an `alloca` operation in an arg block.

It is not legal to use `lda` to take the address of an actual argument in an arg block.

There must be a one to one correspondence between the actual arguments of an arg block, and the formal arguments of the function called by the single call operation in the arg block. Each actual argument must appear exactly once as either an input actual argument or output actual argument of the call operation. It is an error if an actual argument does not appear as one of the call operations input or output arguments, appears more than once as an input or output argument, or appears as both an input and output argument. This requirement applies even if the called function does not

use an input formal argument, or the arg block does not use the output actual argument.

The actual arguments of a call operation must be compatible with the corresponding formal parameters of the function being called. The arguments are compatible if there are the same number of actual and formal input arguments, the same number of actual and formal output arguments, and for each argument one of these properties holds:

- The two have identical type, array dimensions and alignment. The array dimension matches if both do not specify an array dimension, or both specify the same array dimension size.
- The argument is the last input argument and both are arrays with elements that have identical type and alignment, and the formal is an array with unspecified size. See [10.4 Variadic Functions \(p. 259\)](#).

The alignment matches if it has the same value regardless of whether it is explicitly specified by an `align` type qualifier, or has implicit default natural alignment.

For indirect function calls, the formal arguments are specified by a function signature and must match the formal arguments of the function that is actually called at runtime (see [10.3.3 Function Signature \(p. 258\)](#)).

An arg segment variable declared as an array is useful in the following cases:

- To pass a structure to a function.
- To pass a large number of arguments to a function
- To pass a variable number of arguments to a function
- To pass argument values of different types to a function

For actual arguments that correspond to the input formal arguments, it is undefined if they are accessed by any operation other than a `st` operation that is post-dominated (see [2.12.3 Post-Dominator and Immediate Post-Dominator \(p. 30\)](#)) by the call operation.

For the actual argument that corresponds to the output formal argument, it is undefined if it is accessed by any operation other than a `ld` operation that is dominated by the call operation.

It is undefined if the single call operation contained in the arg block is not executed exactly once while executing the arg block. Therefore, it is not allowed to conditionally execute the call operation within the arg block, or loop within the arg block to execute the call operation multiple times. If that is required then the control flow should be placed outside the arg block.

In the code block of the called function definition:

- For input formal arguments, it is undefined if they are accessed by any operation other than a `st` operation.
- For the output formal argument, it is undefined if it is accessed by any operation other than a `ld` operation.
- It is not legal to use `lda` to take the address of a formal argument.

At the start of execution of the function code block, the input formal arguments have the final value stored to the corresponding actual argument of the call operation in the arg block. The input formal argument value for any bytes not stored in the corresponding input actual argument in the calling arg block are undefined.

At the start of execution of the function code block, the output formal argument value is undefined. When execution of the function code block returns to the calling arg block, the output actual argument has the final value stored in the output formal argument. The output actual argument value for any bytes not stored in the called function definition code block are undefined.

An arg segment variable can be used to hold the address of an array that is allocated to private segment memory. The private segment variable can be used to bundle up a sequence of actual arguments and then pass the variable to the function by reference.

A typical call to a function operates as described below:

- In the caller arg block:
 - a. Define actual arguments to hold input and output function arguments.
 - b. Store the values into the input actual arguments.
 - c. Make the call specifying the actual arguments as the input and output function arguments.
 - d. Optionally load the result from the output actual argument after the call.
- In the callee function definition:
 - a. The input arguments come into the function as input formal arguments.
 - b. Code can use loads to access the input formal arguments.
 - c. The callee can copy the formal arguments into private segment variables in order to use lda to obtain a private segment address that can be passed to additional functions.
 - d. Store the result into the output formal argument.

The finalizer can implement arg segment variables as physical registers or can map them into memory.

10.3 Function Declarations, Function Definitions, and Function Signatures

Functions definitions cannot be nested, but functions can be called recursively.

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 [10.6 Direct Call \(call\) Operation \(p. 260\)](#), [10.7 Switch Call \(scall\) Operations \(p. 261\)](#) and [10.8 Indirect Call \(icall, ldi\) Operations \(p. 263\)](#).

10.3.1 Function Declaration

A function declaration is a function header, prefixed by `decl`, 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. See [4.3.3 Function \(p. 43\)](#).

For example:

```
decl 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 header, followed by a code block. See [4.3.3 Function \(p. 43\)](#).

For example:

```
function &fnWithTwoArgs(arg_u32 %out) (arg_u32 %in0, arg_u32 %in1)
{
    ld_arg_u32 $s0, [%in0];
    ld_arg_u32 $s1, [%in1];
    add_u32 $s2, $s0, $s1;
    st_arg_u32 $s2, [%out];
    ret;
};

function &caller() ()
{
    // ...
    {
        arg_u32 %input1;
        arg_u32 %input2;
        arg_u32 %res;
        st_arg_u32 $s1, [%input1];
        st_arg_u32 42, [%input2];
        call &fnWithTwoArgs(%res) (%input1, %input2); // call of a function
                                                       // all work-items called
        ld_arg_u32 $s2, [%res];
    }
    // ...
};
```

10.3.3 Function Signature

A signature is used to describe the type of a function. It cannot be called directly, but instead is used to specify the target of an indirect function call `icall` operation. Syntactically, a signature is much like a function. See [4.3.4 Signature \(p. 45\)](#).

In the following example, assume that `$d2` in each work-item contains the address of an indirect function descriptor 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 %in1;
        arg_u32 %in2;
        arg_u32 %out;
        // ...
        icall $d2(%out) (%in1, %in2) &fun_t;
    }
};
```

This is a call of some indirect 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 indirect functions.

The behavior is undefined if the register does not contain the address of an indirect function descriptor that matches the signature.

10.4 Variadic Functions

A variadic function is a function that accepts a variable number of arguments.

In HSAIL, variadic functions are declared by specifying the last formal argument as an array with no specified size (for example, `uint32 extra_args[]`). The matching actual argument passed by a call operation must be an arg segment variable defined as a fixed-size array.

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 and the second argument is an array of floating-point values.

```
function &sumofN(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_b1 $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
    br @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];
    arg_u32 %count;
    arg_f32 %sum;
    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];
    st_arg_u32 4, [%count];
    call &sumofN(%sum)(%count, %n);
    ld_arg_f32 $s0, [%sum];
}
// ... %s0 holds the sum
};
```

10.5 align Qualifier

`align` is an optional qualifier indicating the alignment of the arg variable in bytes. For information about the `align` qualifier, see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

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 values of 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 &foo() (align(8) arg_b8 %z[])
{
    // ...
    ret;
};

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);
}
// ...
};
```

10.6 Direct Call (`call`) Operation

The `call` operation transfers control to a specific function.

10.6.1 Syntax

Table 12–1 Syntax for direct call Operation

Opcode and Modifiers	Operands
<code>call</code>	<code>function (outputArgs) (inputArgs)</code>

Explanation of Operands (see 4.16 Operands (p. 86))

function: Must be the identifier of a function (either non-indirect or indirect). The function output and input formal arguments must match the *outputArgs* and *inputArgs* specified.

outputArgs: List of zero or one call argument.

inputArgs: List of zero or more comma-separated call arguments.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.6 BRIG Syntax for Function Operations \(p. 376\)](#).

10.6.2 Description

A direct call operation transfers control to a specific function specified by the *function* operand. *function* can be the identifier of a function declaration or definition. The function can be either a non-indirect function or an indirect function. At the time of finalizing, the transitive closure of all functions specified by a `call` or `scall` operation starting at the kernel or indirect function being finalized, must have a definition in some module in the HSAIL program. In addition, all variables and fbarriers they reference must have a definition in some module in the HSAIL program. See [4.2 Program \(p. 35\)](#).

Calls must appear inside of an arg block which is used to pass arguments in and out of the function being called. This is required even if the function has no arguments. See [10.2 Function Call Argument Passing \(p. 254\)](#).

Direct calls are the most efficient form of function calls. An implementation may implement them using a function call stack which can store the arguments, function scope private segment variables and return instruction address so execution can resume after the call operation. The calling convention used could be specialized to a specific call site. It is also allowed to inline the function code block.

Example

```
decl function &foo(arg_u32 %r) (arg_f32 %a);

function &example_call(arg_u32 %res) (arg_u32 %arg1)
{
{
    arg_u32 %a;
    arg_u32 %r;
    st_arg_f32 2.0f, [%a];
    // call &foo
    call_width(all) &foo(%r) (%a);
    ld_arg %s1, [%r];
    st_arg %s1, [%res];
}
};
```

10.7 Switch Call (scall) Operations

The `scall` operation uses an integer index to select the specific function to which control is transferred.

10.7.1 Syntax

Table 12–2 Syntax for switch call Operation

Opcode and Modifiers	Operands
<code>scall_width_uLength</code>	<code>src (outputArgs) (inputArgs) [functionList]</code>

Explanation of Modifier

`width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the result uniformity of the target for switch calls. All active work-items in the same slice are guaranteed to call the same target. If the width modifier is omitted, it defaults to `width(1)`, indicating each active work-item can call a different target. See [2.12.2 Using the Width Modifier with Control Transfer Operations \(p. 29\)](#).

`Length`: 32, 64.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`src`: Source. Can be a register, immediate value, or `WAVESIZE`.

`outputArgs`: List of zero or one call arguments.

`inputArgs`: List of zero or more comma-separated call arguments.

`functionList`: Comma-separated list of global identifiers of both non-indirect and indirect functions. All functions must have the same input and output formal arguments, but do not have to match whether they are indirect functions or not. The functions output and input formal arguments must match the `outputArgs` and `inputArgs` specified.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.6 BRIG Syntax for Function Operations \(p. 376\)](#).

10.7.2 Description

A switch call transfers control to the function in the `functionList` that corresponds to the index value in `src`. If the index value is 0 then the first function is selected, if 1 then the second function, and so forth. It is undefined if the number of functions in `functionList` is less than or equal to the index value. `src` can either be of type `u32` or `u64`.

The functions in `functionList` can be either a non-indirect or an indirect functions. They must all have the same input and output arguments. At the time of finalizing, the transitive closure of all functions specified by a `call` or `scall` operation starting at the kernel or indirect function being finalized, must have a definition in some module in the HSAIL program. In addition, all variables and fbarriers they reference must have a definition in some module in the HSAIL program. See [4.2 Program \(p. 35\)](#).

Since a switch call can potentially transfer to more than one target, it can result in control flow divergence which can introduce a performance issue. The width modifier can be used to specify properties about the control flow divergence that may result in the finalizer producing more efficient code. See [2.12 Divergent Control Flow \(p. 26\)](#).

Calls must appear inside of an arg block which is used to pass arguments in and out of the function being called. This is required even if the functions have no arguments. See [10.2 Function Call Argument Passing \(p. 254\)](#).

It is implementation-defined how a switch call is finalized to machine instructions. For example: by a cascade of compare and conditional branches to direct calls; by an indirect call through a jump table, or a combination of these approaches. The performance of switch calls can therefore potentially be slow for long function lists. An implementation may implement the selected call using a function call stack which can store the arguments, function scope private segment variables and return instruction address so execution can resume after the switch call operation. The calling convention used could be specialized to a specific call site. If cascaded control flow with direct calls is used, it is also allowed to inline any or all of the function code blocks.

Example

```
decl function &foo(arg_u32 %r) (arg_f32 %a);
decl function &bar(arg_u32 %r) (arg_f32 %a);

function &example_scall(arg_u32 %res) (arg_u32 %arg1)
{
    ld_kernarg_u32 $s1, [%arg1];
    {
        arg_u32 %a;
        arg_u32 %r;
        st_arg_f32 2.0f, [%a];
        // call &foo or &bar.
        scall_width(all)_u32 $s1(%r) (%a) [&foo, &bar];
        ld_arg %s1, [%r];
        st_arg %s1, [%res];
    }
};
```

10.8 Indirect Call (icall, idi) Operations

Indirect functions allow an application to incrementally finalize the code for functions that can be called by kernels that have already been finalized. For example, this may be useful for languages that can incrementally load and finalize derived classes. The virtual function table for the derived class will then have indirect function descriptor addresses for the derived class virtual functions that override those of the base class. That may result in a previously finalized kernel calling the derived class functions if passed an object of the derived class.

An indirect function is declared and defined in the same way as a non-indirect function except:

- The function header must use the `indirect` qualifier.
- Indirect functions have limitations to allow them to be called by kernels that have already been finalized. They therefore cannot result in the kernel requiring additional group segment or private segment memory for variables, or additional fbarriers. Therefore the transitive closure of all functions specified by a `call` or `scall` operation starting at the indirect function definition code block, must not:
 - Reference any module scope group or private segment variables.
 - Define any function scope group segment variables.
 - Reference any module scope fbarriers.
 - Define any function scope fbarriers.

10.8.1 Syntax

Table 12–3 Syntax for indirect call Operations

Opcode and Modifiers	Operands
<code>icall_width_uLength</code>	<code>src (outputArgs) (inputArgs) signature</code>
<code>ldi_uLength</code>	<code>dest, indirectFunction</code>

Explanation of Modifier
<code>width</code> : Optional: <code>width(n)</code> , <code>width(WAVESIZE)</code> , or <code>width(all)</code> . Used to specify the result uniformity of the target for indirect calls. All active work-items in the same slice are guaranteed to call the same target. If the <code>width</code> modifier is omitted, it defaults to <code>width(1)</code> , indicating each active work-item can call a different target. See 2.12.2 Using the Width Modifier with Control Transfer Operations (p. 29) .
<code>Length</code> : 32, 64. Must match the address size for the global segment (see Table 4–3 (p. 25)).

Explanation of Operands (see 4.16 Operands (p. 86))
<code>outputArgs</code> : List of zero or one call argument.
<code>inputArgs</code> : List of zero or more comma-separated call arguments.
<code>src</code> : A register.
<code>dest</code> : A register.
<code>indirectFunction</code> : Must be the identifier of an indirect function.
<code>signature</code> : Global identifier of a signature. The signature output and input formal arguments must match the <code>outputArgs</code> and <code>inputArgs</code> specified.

Exceptions (see Chapter 12 Exceptions (p. 285))
No exceptions are allowed.

For BRIG syntax, see [18.7.6 BRIG Syntax for Function Operations \(p. 376\)](#).

10.8.2 Description

ldi

Set *dest* to the global segment address of the indirect function descriptor, within the same HSAIL program as the kernel dispatch being executed, for the indirect function *indirectFunction*. *indirectFunction* can be the identifier of an indirect function declaration or definition. At the time of finalization, the HSAIL program must contain an HSAIL module with a definition for the specified indirect function. The indirect function descriptor address can be used to specify which indirect function an `iCall` operation should call. See [4.2.3 Finalization and Code Descriptors \(p. 36\)](#).

icall

An indirect call transfers control to the indirect function that corresponds to the indirect function descriptor global segment address in *src*. The indirect function being called has formal arguments matching those of *signature*.

An indirect function descriptor address can be obtained in two ways:

- A host CPU agent can use an HSA runtime query to obtain the address of an indirect function descriptor. That address can then be passed into a kernel as a kernel argument or through global segment memory.
- An `ldi` operation.

It is undefined if:

- *src* is not a valid indirect function descriptor address for the same program as the code for the kernel currently executing.
- *src* refers to an indirect function with formal input and output arguments that do not match *signature*.
- *src* refers to an indirect function that was not finalized with the same call convention as the currently executing kernel before the current kernel was launched.
- No global segment acquire at system scope has been performed since the *src* code descriptor for the call convention of the currently executing kernel was updated by the finalizer.

See [4.2.3 Finalization and Code Descriptors \(p. 36\)](#).

At the time of finalizing, the actual indirect function that an `icall` will call at runtime does not have to be finalized.

An indirect function descriptor can contain information about the finalized code for all call conventions supported by the HSA components of a program. Therefore, the same indirect function descriptor address can be used with an `icall` executed by any HSA component that is a member of the same program, provided the indirect function has been appropriately finalized. This makes them suitable to support virtual functions that may be called by kernels executing on any HSA component that is part of the program.

Since an indirect call can potentially transfer to more than one target, it can result in control flow divergence which can introduce a performance issue. The width modifier can be used to specify properties about the control flow divergence that may result in the finalizer producing more efficient code. See [2.12 Divergent Control Flow \(p. 26\)](#).

Calls must appear inside of an arg block which is used to pass arguments in and out of the function being called. This is required even if the functions have no arguments. See [10.2 Function Call Argument Passing \(p. 254\)](#).

Since the exact indirect function that will be called is not known until runtime, indirect calls are the least efficient form of function calls.

Example

```

signature &bar_or_foo_t(arg_u32 %r)(arg_f32 %a);
decl indirect function &bar(arg_u32 %r)(arg_f32 %a);
decl indirect function &foo(arg_u32 %r)(arg_f32 %a);
global_u64 &i;

// First execute kernel to save indirect function descriptor
// address in a global variable.
kernel &example_ldi(kernarg_u32 %arg1)
{
    ld_kernarg_u32 $s0, [%arg1];
    cmp_b1_u32 $c1, $s0, 0;
    ldi_u64 $d1, &foo;
    cbr_b1 $c0, @lab;
    ldi_u64 $d1, &bar;
@lab:
    st_global_u64 $d1, [&i];
};

// Then execute a kernel that uses the global variable to call the
// indirect function. The kernels do not have to execute on the same
// agent. The actual indirect function called must have been
// finalized for the same agent and call convention as the
// finalization of this kernel that will be dispatched, before this
// kernel is launched.
kernel &example_icall(kernarg_u64 %res)
{
    ld_global_u64 $d1, [&i];
    {
        arg_u32 %a;
        arg_u32 %r;
        st_arg_f32 2.0f, [%a];
        // $d1 must contain an indirect function descriptor address
        // of an indirect function that matches the signature &bar_or_foo_t.
        // In this case &foo or &bar are the two potential targets.
        icall_width(all) $d1(%r)(%a) &bar_or_foo_t;
        ld_arg $s1, [%r];
        ld_kernarg_u64 $d1, [%res];
        st_global_u32 $s1, [$d1];
    }
};

```

10.9 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 act as if a ret operation was present at the end of the function.

10.9.1 Syntax

Table 12–4 Syntax for ret Operation

Opcode
ret

Exceptions (see Chapter 12 Exceptions (p. 285))

No exceptions are allowed.

For BRIG syntax, see [18.7.6 BRIG Syntax for Function Operations \(p. 376\)](#).

10.9.2 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.

Example

```
ret;
```

10.10 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.

10.10.1 Syntax

Table 12–5 Syntax for Allocate Memory (alloca) Operation

Opcode	Operands
<code>alloca_align(n)_u32</code>	<code>dest, src</code>

Explanation of Modifiers

`align(n)`: Optional. Used to specify the byte alignment of the base of the memory being allocated. If omitted, 1 is used indicating no alignment. See the Description below.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination. Must be a 32-bit register.

src: Source. Can be a 32-bit register, immediate value, or `WAVESIZE`. The value of *src* is the minimum amount of space (in bytes) requested.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.6 BRIG Syntax for Function Operations \(p. 376\)](#).

10.10.2 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 of the allocated memory is required, it can be specified by the `align(n)` modifier. Valid values of *n* are 1, 2, 4, 8, 16, 32, 64, 128 and 256. The private segment address returned in *dest* is required to be a multiple of *n*. If `align` is omitted, the value 1 is used for *n*, and the returned address will have no guaranteed alignment. It is recommended to specify an alignment that corresponds to the natural alignment of the types used to access the memory returned. Using an alignment larger than necessary may result in lower performance and increased memory usage on some implementations. See [17.8 Unaligned Access \(p. 314\)](#).

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, or to satisfy the alignment requested. An implementation may also choose to allocate the maximum size amongst the active work-items in the wavefront so only a single stack pointer per wavefront has to be maintained. This can result in more private segment memory being required than expected.

The behavior is undefined if not enough private memory is available to satisfy the requested size.

It is not valid to use an `alloca` operation in an argument scope. See [10.2 Function Call Argument Passing \(p. 254\)](#).

Example

```
alloca_u32 $s1, 24;
alloca_align(8)_u32 $s1, 24;
```


Chapter 11

Special Operations

This chapter describes special operations that can be used to perform various miscellaneous actions and queries.

11.1 Dispatch Packet Operations

The dispatch packet operations can be used to obtain information about the currently executing kernel dispatch packet.

11.1.1 Syntax

The table below shows the syntax for the dispatch packet operations in alphabetical order.

Table 13–1 Syntax for Dispatch Packet Operations

Opcodes and Modifier	Operands
<code>currentworkgroupsize_u32</code>	<code>dest, dimNumber</code>
<code>dim_u32</code>	<code>dest</code>
<code>gridgroups_u32</code>	<code>dest, dimNumber</code>
<code>gridsize_u32</code>	<code>dest, dimNumber</code>
<code>packetcompletionsig_signalType</code>	<code>dest</code>
<code>packetid_u64</code>	<code>dest</code>
<code>workgroupid_u32</code>	<code>dest, dimNumber</code>
<code>workgroupsize_u32</code>	<code>dest, dimNumber</code>
<code>workitemabsid_uLength</code>	<code>dest, dimNumber</code>
<code>workitemflatabsid_uLength</code>	<code>dest</code>
<code>workitemflatid_u32</code>	<code>dest</code>
<code>workitemid_u32</code>	<code>dest, dimNumber</code>

Explanation of Modifiers

`signalType`: Must be `sig32` for small machine model and `sig64` for large machine model. See [Table 6–4 \(p. 81\)](#) and [2.9 Small and Large Machine Models \(p. 24\)](#).

`Length`: 32, 64.

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination. For `packetcompletionsig` and `packetid` must be a `d` register; otherwise must be an `s` register. See [Table 4-3 \(p. 25\)](#).

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`.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.1 BRIG Syntax for Dispatch Packet Operations \(p. 376\)](#).

11.1.2 Description

currentworkgroupsize

Accesses the work-group size that the currently executing work-item belongs to for the *dimNumber* 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 *dimNumber*, then `currentworkgroupsize` returns 1 for the unused dimensions.

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 \(p. 5\)](#).

gridgroups

Returns the upper bound for work-group identifiers (IDs) (that is, the number of work-groups) within the grid for the *dimNumber* dimension and stores the result in the destination *dest*.

If the grid was launched with fewer dimensions than *dimNumber*, then `gridgroups` stores 1 in destination *dest*.

`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 for the *dimNumber* dimension and stores the result in the destination *dest*.

If the grid was launched with fewer dimensions than *dimNumber*, then `gridsize` stores 1 in destination *dest*.

packetcompletionsig

Returns the signal handle of the completion signal specified for this dispatch in *dest*. The value may be 0 indicating there is no associated completion signal (see [6.8 Notification \(signal\) Operations \(p. 187\)](#)). See *HSA Platform System Architecture Specification* for more information on User Mode Queuing.

packetid

Returns a 64-bit User Mode queue packet identifier (packet ID) that is unique for the User Mode Queue used for this dispatch and stores the result in the destination *dest*. See *HSA Platform System Architecture Specification* for more information on User Mode Queues.

The combination of the queue ID and the packet ID can be used to identify a kernel dispatch within an application. Debugging tools might find this useful.

workgroupid

Accesses the work-group identifier (ID) within the grid.

This operation computes the three-dimensional ID of the work-group, selects the *dimNumber* 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 *dimNumber* 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 *dimNumber*, then **workgroupsize** stores 1 in destination *dest*.

workitemabsid

Accesses the work-item absolute identifier (ID) within the entire grid and stores the result for the *dimNumber* dimension in the destination *dest*. Can either be returned as a u32 or u64. If u32 then the lower 32 bits of the ID are returned.

If the work-group was launched with fewer dimensions than *dimNumber*, **workitemabsid** stores 0 in destination *dest*.

workitemflatabsid

Accesses the flattened form of the work-item absolute identifier (ID) within the entire grid and stores the result in the destination *dest*. Can either be returned as a u32 or u64. If u32 then the lower 32 bits of the ID are returned.

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 for the *dimNumber* dimension in the destination *dest*.

If the work-group was launched with fewer dimensions than *dimNumber*, *workitemid* stores 0 in the destination *dest*.

Examples

```
currentworkgroupsize_u32 $s1, 0; // access the number of work-items in
                                // the current work-group in the X
                                // dimension, which might be partial
dim_u32 $s3;                  // dispatch dimensions
gridgroups_u32 $s2, 2;        // access the number of work-groups in the
                                // grid Z dimension
gridsize_u32 $s2, 2;          // access the number of work-items in the
                                // grid Z dimension
packetcompletionsig_sig64 $d6; // get current dispatch packet
                                // completion signal handle
packetid_u64 $d0;             // access the dispatch packet ID
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
workgroupsize_u32 $s1, 0;       // access the number of work-items in the
                                // non-partial work-groups in the X dimension
workitemabsid_u32 $s1, 0;       // access the work-item absolute ID in the
                                // X dimension
workitemabsid_u64 $d1, 1;       // access the work-item absolute ID in the
                                // Y dimension
workitemflatabsid_u32 $s1;      // access the work-item flat absolute ID
workitemflatabsid_u64 $d1;      // access the work-item flat absolute ID
workitemflatid_u32 $s1;          // access the work-item flat ID
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
```

11.2 Exception Operations

The exception operations can be used to determine what exceptions have been generated.

11.2.1 Syntax

The table below shows the syntax for the exception operations in alphabetical order.

Table 13–2 Syntax for Exception Operations

Opcodes and Modifier	Operands
<code>cleardetectexcept_u32</code>	<code>exceptionsNumber</code>
<code>getdetectexcept_u32</code>	<code>dest</code>
<code>setdetectexcept_u32</code>	<code>exceptionsNumber</code>

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination. Must be an s register. See [Table 4-3 \(p. 25\)](#).

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 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.2 BRIG Syntax for Exception Operations \(p. 377\)](#).

11.2.2 Description

cleardetectexcept

Clears DETECT exception flags specified in *exceptionsNumber* for the work-group containing the work-item. The result is undefined if the operation is not work-group execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)), and might lead to deadlock.

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 the operation is not work-group execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)), and might lead to deadlock.

setdetectexcept

Sets DETECT exception flags specified in *exceptionsNumber* for the work-group containing the current work-item. The result is undefined if the operation is not work-group execution uniform (see [2.12 Divergent Control Flow \(p. 26\)](#)), and might lead to deadlock.

11.2.3 Additional Information

DETECT exception processing operates on the five exceptions specified in [12.2 Hardware Exceptions \(p. 285\)](#).

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. 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. See [2.12 Divergent Control Flow \(p. 26\)](#).

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 [17.13 Exceptions \(p. 315\)](#).

Examples

```
cleardetectexcept_u32 1; // clear DETECT policy flags
getdetectexcept_u32 $s1; // get DETECT policy flags
setdetectexcept_u32 2;   // set DETECT policy flags
```

11.3 User Mode Queue Operations

The User Mode Queue operations can be used to enqueue work to be executed by other agents. See *HSA Platform System Architecture Specification* for more information on User Mode Queuing.

11.3.1 Syntax

The table below shows the syntax for the User Mode Queue operations in alphabetical order.

Table 13–3 Syntax for Exception Operations

Opcodes and Modifier	Operands
<code>agentcount_u32</code>	<code>dest</code>
<code>agentid_u32</code>	<code>dest</code>
<code>addqueuewriteindex_segment_order_u64</code>	<code>dest, address, src</code>
<code>casqueuewriteindex_segment_order_u64</code>	<code>dest, address, src0, src1</code>
<code>ldk_uLength</code>	<code>dest, kernelName</code>
<code>ldqueuereadindex_segment_order_u64</code>	<code>dest, address</code>
<code>ldqueuewriteindex_segment_order_u64</code>	<code>dest, address</code>
<code>queueid_u32</code>	<code>dest</code>
<code>queueptr_uLength</code>	<code>dest</code>
<code>stqueuereadindex_segment_order_u64</code>	<code>address, src</code>
<code>stqueuewriteindex_segment_order_u64</code>	<code>address, src</code>

Explanation of Modifiers

`segment`: Optional segment. If omitted, flat is used. Only flat and global is allowed. See [2.8 Segments \(p. 13\)](#).

`order`: Memory order used to specify synchronization. See [6.2.1 Memory Order \(p. 162\)](#).

`Length`: 32, 64. Must match the address size for the global segment (see [Table 4–3 \(p. 25\)](#)).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

`dest`: Destination. For `ldk` and `queueptr` must be a register with a size that matches the address size of the global segment or flat address; for `agentcount`, `agentid`, and `queueid` must be an `s` register; otherwise must be a `d` register. See [Table 4–3 \(p. 25\)](#).

`src, src0, src1`: Source. Can be register, immediate or `WAVESIZE`.

`address`: Address expression for an address in the specified segment for a User Mode queue created by the HSA runtime (see [4.18 Address Expressions \(p. 88\)](#)).

`kernelName`: Name of a kernel.

Exceptions (see Chapter 12 Exceptions (p. 285))
--

No exceptions are allowed.

For BRIG syntax, see [18.7.7.3 BRIG Syntax for User Mode Queue Operations \(p. 377\)](#).

11.3.2 Description

`agentcount`

Returns the number of agents, that are members of the same HSAIL program as the currently executing kernel dispatch, in `dest` as a 32-bit unsigned integer. See [4.2.1 Agent Id \(p. 36\)](#).

`agentid`

Returns the 32-bit agent identifier of the HSA component executing the current kernel dispatch, in the same HSAIL program as the executing kernel, in `dest` as a 32-bit unsigned integer. Each HSA component has a unique identifier, in the range 0 to `agentcount`-1, within an HSAIL program of which it is a member. See [4.2.1 Agent Id \(p. 36\)](#).

`addqueuewriteindex`

Atomically adds the unsigned 64 bit value in `src` to the current value of the write index associated with the User Mode Queue with address specified by `address`. Returns the original unsigned 64 bit User Mode Queue Packet ID (Packet ID) value of the write index in `dest`. The new value of the write index must be greater than or equal to the original value: adding a value that causes the write index to wrap is undefined. The add is performed as if a read-modify-write atomic memory operation, to the global segment, at system scope, with memory ordering specified by `order` which can be `r1x` (relaxed), `scacq` (sequentially consistent acquire), `screl` (sequentially consistent release) or `scar` (sequentially consistent acquire release). Can be used to allocate zero or more packet slots in a User Mode Queue when there are multiple producer agents.

`casqueuewriteindex`

Atomically compares `src0` to the current value of the write index associated with the User Mode Queue with address specified by `address`, and if the values are the same sets the write index to `src1`. Returns the original value of the write index in `dest`. The `src0`, `src1` and `dest` are unsigned 64 bit User Mode Queue Packet IDs (Packet ID). `src1` must be greater than or equal to `src0`. The compare-and-swap is performed as a read-modify-write atomic memory operation, to the global segment, at system scope, with memory ordering specified by `order` which can be `r1x` (relaxed), `scacq` (sequentially consistent acquire), `screl` (sequentially consistent release) or `scar` (sequentially consistent acquire release). Can be used to allocate zero or more packet slots in a User Mode Queue in conjunction with the `ldqueuewriteindex` when there are multiple producer agents.

1dk

Set *dest* to the global segment address of the kernel descriptor, within the same HSAIL program as the kernel dispatch being executed, for the kernel *kernelName*. *kernelName* can be the identifier of a kernel declaration or definition. At the time of finalization, the HSAIL program must contain an HSAIL module with a definition for the specified kernel. The kernel descriptor can be indexed by an agent identifier to obtain the information needed to create a User Mode Queue kernel dispatch packet. See [4.2.3 Finalization and Code Descriptors \(p. 36\)](#).

1dqueuereadindex

Atomically loads the current value of the read index associated with the User Mode Queue with address specified by *address* into *dest*. The value is an unsigned 64 bit User Mode Queue Packet ID (Packet ID) for the next packet slot in the User Mode Queue to be consumed. The load is performed as an atomic memory operation, to the global segment, at system scope, with memory ordering specified by *order* which can be `r1x` (relaxed) or `scacq` (sequentially consistent acquire). Can be used in conjunction with **1dqueuewriteindex** to determine how many User Mode Queue slots are available.

1dqueuewriteindex

Atomically loads the current value of the write index associated with the User Mode Queue with address specified by *address* into *dest*. The value is an unsigned 64 bit User Mode Queue Packet ID (Packet ID) for the next packet slot in the User Mode Queue to be allocated. The load is performed as an atomic memory operation, to the global segment, at system scope, with memory ordering specified by *order* which can be `r1x` (relaxed) or `scacq` (sequentially consistent acquire). Can be used in conjunction with **1dqueuereadindex** to determine how many User Mode Queue slots are available.

queueid

Returns a 32-bit User Mode Queue Identifier (Queue ID) for the queue that the currently executing kernel's dispatch packet was submitted to, and stores the result in the destination *dest*.

A queue ID is unique for the User Mode Queues that currently exist within an application. They may be reused during the lifetime of the application, and are not guaranteed to be unique between processes. See *HSA Platform System Architecture Specification* for more information on User Mode Queues.

The combination of the queue ID and the packet ID can be used to identify a kernel dispatch within an application. Debuggers might find this useful.

queueptr

Sets the destination *dest* to either the global segment or flat address of the User Mode 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 Platform System Architecture Specification*.

stqueuereadindex

Atomically stores *src* into the read index associated with the User Mode Queue with address specified by *address*. The value is an unsigned 64 bit User Mode Queue Packet ID (Packet ID) and must be greater than or equal to the current value of the read index. The store is performed as an atomic memory operation, to the global segment, at system scope, with memory ordering specified by *order* which can be `rlx` (relaxed) or `screl` (sequentially consistent release). Only permitted on User Mode Queues that are not associated with an HSA component to indicate zero or more packet slots are being processed or have been completed. For example, to implement a User Mode Queue that supports agent dispatch packets for use as a service queue. Not permitted with User Mode Queues that are associated with an HSA component for which only the associated packet processor is permitted to update the read index.

stqueuewriteindex

Atomically stores *src* into the write index associated with the User Mode Queue with address specified by *address*. The value is an unsigned 64 bit User Mode Queue Packet ID (Packet ID) and must be greater than or equal to the current value of the write index. The store is performed as an atomic memory operation, to the global segment, at system scope, with memory ordering specified by *order* which can be `rlx` (relaxed) or `screl` (sequentially consistent release). Can be used to allocate zero or more packet slots in a User Mode Queue when there is only a single producer agent.

Examples

```
agentcount_u32 $s0;                                // access number of agents within the
                                                    // same HSAIL program
agentid_u32 $s0;                                 // access the agent ID within the same
                                                    // HSAIL program
ldk_u64 $d0, &kernel_one;                         // get address of kernel descriptor
queueid_u32 $s0;                                 // access the queue ID
queueptr_u64 $d2;                                // access the queue address
ldqueuewriteindex_global_rlx_u64 $d3, [$d2];     // load queue write index
add_u64 $d4, $d3, 1;                            // compare-and-swap queue write index
casqueuewriteindex_global_scar_u64 $d1, [$d2], $d3, $d4;
addqueuewriteindex_global_rlx_u64 $d1, [$d2], 2; // add to queue write index
ldqueuereadindex_global_scacq_u64 $d5, [$d2];   // load queue read index
stqueuereadindex_global_screl_u64 [$d2], $d4;    // store queue read index to a non-HSA
                                                    // component User Mode Queue
stqueuewriteindex_global_screl_u64 [$d2], $d4;    // store queue write index
```

11.4 Miscellaneous Operations

The miscellaneous operations include various query and special operations.

11.4.1 Syntax

The table below shows the syntax for the miscellaneous operations in alphabetical order.

Table 13–4 Syntax for Miscellaneous Operations

Opcodes and Modifier	Operands
<code>clock_u64</code>	<code>dest</code>
<code>cuid_u32</code>	<code>dest</code>
<code>debugtrap_u32</code>	<code>src</code>
<code>groupbaseptr_uLength</code>	<code>dest</code>
<code>kernargbaseptr_uLength</code>	<code>dest</code>
<code>laneid_u32</code>	<code>dest</code>
<code>maxcuid_u32</code>	<code>dest</code>
<code>maxwaveid_u32</code>	<code>dest</code>
<code>nop</code>	
<code>nullptr_segment_uLength</code>	<code>dest</code>
<code>waveid_u32</code>	<code>dest</code>

Explanation of Modifiers

Length: 32, 64. Must match the address size for the associated segment: for `nullptr` it is the segment specified; for `groupbaseptr` it is the group segment; and for `kernargbaseptr` if is the kernarg segment (see [Table 4–3 \(p. 25\)](#)).

segment: Optional segment. If omitted, flat is used. Can be flat, global, readonly, group, private and kernarg. See [2.8 Segments \(p. 13\)](#).

Explanation of Operands (see [4.16 Operands \(p. 86\)](#))

dest: Destination. For `nullptr` must be a register with a size that matches the address size of the segment or flat address; for `clock` must be a `d` register; otherwise must be an `s` register. See [Table 4–3 \(p. 25\)](#).

src: Source. Can be register, immediate or `WAVESIZE`.

Exceptions (see [Chapter 12 Exceptions \(p. 285\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.4 BRIG Syntax for Miscellaneous Operations \(p. 378\)](#).

11.4.2 Description

clock

Stores the current value of a 64-bit unsigned system timestamp in a `d` register specified by the destination `dest`. All agents in the HSA system are required to provide a uniform view of time which must not roll over. The system timestamp must count at a constant increment rate in the range 1–400MHz, and the HSA runtime can be queried to determine the frequency. The system timestamp is defined in the *HSA Platform System Architecture Specification*.

The `clock` operation is treated as if it is a read-modify-write relaxed atomic memory operation (see [6.2 Memory Model \(p. 160\)](#)). This ensures that the `clock` operation will not give unexpected results due to being drastically moved as a result of optimization, but still allows optimization to be performed. Consequently:

- A `clock` operation cannot be moved (by the implementation) before a preceding acquire memory operation in the same work-item.
- A `clock` operation cannot be moved (by the implementation) after a following release memory operation in the same work-item.
- The order of two `clock` operations cannot be changed (by the implementation).
- Multiple `clock` operations cannot be combined (by the implementation) to a single operation, including hoisting out of a loop.

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.

debugtrap

Can be used to halt the current waveform and transfer control to the debugger. See [12.4 Debugger Exceptions \(p. 289\)](#). The source `src` is passed to the debugger and can be used to identify the trap.

groupbaseptr

Returns the group segment address of the base of the group segment for the work-group of the work-item executing the operation, and stores the result in the destination `dest`. All group segment variables used by the kernel, and the functions it calls directly or indirectly, are allocated within the group segment address range starting at offset 0 from the group segment base up to the group segment size reported when the kernel was finalized. Note, since all variables must have a segment and flat address that is naturally aligned or specified by the `alloc` variable qualifier (see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#)), the group segment base address will always be aligned to the maximum alignment of the group segment variables used by the kernel.

If the kernel dispatch uses dynamic group memory, it is allocated by setting a group segment size in the kernel dispatch packet that is larger than the size reported when the kernel was finalized. The base of the dynamically allocated group memory for the work-group of a work-item is obtained by adding the group segment size reported when the kernel was finalized, to the group segment address returned by this operation. See [4.20 Dynamic Group Memory Allocation \(p. 95\)](#).

kernargbaseptr

Returns the kernarg segment address of the base of the kernarg segment for the kernel dispatch being executed, and stores the result in the destination *dest*. The first kernarg segment variable is allocated at offset 0 relative to this base address. The address will be at least 16 byte aligned. Additionally, if any of the kernarg segment variables have `align(n)` qualifiers (see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#)) with *n* larger than 16, then the returned address will have alignment at least the maximum *n* specified. See [4.21 Kernarg Segment \(p. 96\)](#).

For example, can be used in functions called directly or indirectly by a kernel dispatch to directly access the kernel arguments.

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 for a flat address. The implementation will ensure no variable is allocated, and no memory allocator will return an address, with the null pointer address.

The null pointer value for the global segment and flat address is dependent on the host operating system. All agents must use the same values, including host CPU agents. The arg and spill segments do not have a null pointer value since the address of variables in these segments cannot be obtained with the `lda` operation. The value for each other segment is agent dependent and different agents may use different values.

The runtime will provide API calls so that agents can determine the value returned by `nullptr` for each segment for each agent.

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.

Implementations are allowed to move in-flight computations within and between compute units, so the value returned can be different each time `waveid` is executed.

Programs might use this value to address non-persistent global storage.

Examples

```
clock_u64 $d6;           // return the current time access the compute unit id within the
                         // HSA component
debugtrap_u32 $s1;        // halt and transfer control to debugger
groupbaseptr_u32 $d2;     // base address for group segment
kernargbaseptr_u64 $d2;   // base address for kernarg segment
laneid_u32 $s1;          // access the lane ID
maxcuid_u32 $s6;          // access number of compute units on the HSA component
maxwaveid_u32 $s4;        // access the maximum number of waves that can be executing at the
                         // same time by the HSA component
nop;                     // no operation
nullptr_group_u32 $s0;    // null pointer value for group segment
nullptr_global_u64 $d1;   // null pointer value for global segment
waveid_u32 $s3;          // access the wavefront ID within the HSA component
```

Chapter 12

Exceptions

This chapter describes HSA exception processing.

12.1 Kinds of Exceptions

Three kinds of exceptions are supported:

- Hardware-detected exceptions such as divide by zero.
See [12.2 Hardware Exceptions \(p. 285\)](#).
- Software-triggered exceptions corresponding to higher-level catch and throw operations.
HSAIL provides no special operations for handling software exceptions. They can be implemented in terms of the HSAIL branch operations.
- Debug-related exceptions generated by `debugtrap` or as a consequence of actions performed by the HSA Runtime debugger interface.
See [12.4 Debugger Exceptions \(p. 289\)](#).

12.2 Hardware Exceptions

HSAIL defines a set of exceptions, and provides a mechanism to control these exceptions by means of hardware exception policies (see [12.3 Hardware Exception Policies \(p. 287\)](#)). The exception policies are specified when a kernel is finalized and cannot be changed at runtime.

HSAIL requires the hardware to generate the exceptions, as defined by the HSAIL operations, that are enabled for at least one of the exception policies. The hardware is not required to generate exceptions that are not enabled for any exception policy.

The 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.

For the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)), it is not permitted to enable any of the exception policies for the five floating-point exceptions.

HSAIL also 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. If such exceptions are generated, it is implementation defined if the exception is signaled. See [12.5 Handling Signaled Exceptions \(p. 289\)](#). If the implementation does not signal the exception, or if execution is resumed after being halted due to signaling the exception, the value returned by the associated operation is undefined. For

example, a load from an address of a non-existent memory page can return an undefined value.

The exceptions supported by the HSAIL exception policies are:

- Overflow

The floating-point exponent of a value is too large to be represented. See [4.19.2 Rounding \(p. 92\)](#).

- Underflow

A non-zero tiny floating-point value is computed and either:

- the `f tz` modifier was specified,
- or the `f tz` modifier was not specified and the value cannot be represented exactly.

See [4.19.2 Rounding \(p. 92\)](#).

- 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.
- Signalling comparisons: comparisons on quiet NaN (qNaN) 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: `sqrte(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 is not represented exactly in the destination. This can occur:

- Due to rounding. See [4.19.2 Rounding \(p. 92\)](#).
- In addition, it is implementation defined if operations with the `ftz` modifier that cause a value to be flushed to zero generate the inexact exception. See [4.19.3 Flush to Zero \(ftz\) \(p. 92\)](#).

This exception is very common.

In addition, the native floating point operations may generate exceptions. However, it is implementation defined if, and which, exceptions they generate. For example, the `nlog2` operation may generate a divide by zero exception when given the value 0.

12.3 Hardware Exception Policies

HSA supports DETECT and BREAK policies for each of the five exceptions specified in [12.2 Hardware Exceptions \(p. 285\)](#). For the Base profile (see [16.2.1 Base Profile](#)

[Requirements \(p. 309\)](#), it is not permitted to enable the DETECT or BREAK exception policies for any of the five floating-point exceptions.

- 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.

See [11.2 Exception Operations \(p. 274\)](#).

- BREAK

A work-item must signal an exception if it executes an instruction that generates an exception that is enabled by the BREAK policy. See [12.5 Handling Signaled Exceptions \(p. 289\)](#).

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 signal an exception 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 to be enabled for the BREAK policy 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 [17.13 Exceptions \(p. 315\)](#).

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 at runtime. That can only be specified at finalize time through the `enable detect` and `enable break exceptions` arguments specified when the finalizer is invoked, or an `enabledetectexceptions` or `enabledetectexceptions` control directive in the kernel, or any functions it calls directly or indirectly, being finalized.

12.4 Debugger Exceptions

Debug exceptions include those generated by the `debugtrap` operation (see [Chapter 11 Special Operations \(p. 271\)](#)), and those the HSA Runtime debugger interface causes to be generated (for example, due to inserted breakpoints or single stepping machine instructions).

When a debugger exception is generated it always signals the exception. See section [12.5 Handling Sighaled Exceptions \(p. 289\)](#). If execution is resumed after being halted due to signaling the exception, execution continues as if the exception had not been signaled, except for any side effects resulting from halting execution.

12.5 Handling Sighaled Exceptions

If an exception is signaled, the behavior depends on if the HSA Runtime debugger interface is active.

12.5.1 HSA Runtime Debugger Interface Not Active

If the HSA Runtime debugger interface is not active, a wavefront that executes an instruction that signals an exception must halt execution of the wavefront. In reasonable time, the HSA Component executing the wavefront must stop initiating new wavefronts for all dispatches executing on the same User Mode queue, and must ensure that all wavefronts currently being executed for those dispatches either complete, or are halted. Any dispatches that complete will have their completion signal updated as normal, however, any dispatch that do not complete the execution of all their associated wavefronts will not have their completion signal updated. The User Mode queue will then be put into the error state. It is not possible to resume the wavefronts of any of the affected dispatches.

12.5.2 HSA Runtime Debugger Interface Active

If the HSA Runtime debugger interface is active, a signaled exception causes the wavefront that executed the instruction to be halted and information about the exception communicated to the HSA Runtime through the debugger interface. The debugger interface can be used to halt other wavefronts, inspect the execution state of halted wavefronts, modify the execution state of halted wavefronts, or resume the execution of halted wavefronts. In addition, the HSA Runtime can put a User Mode queue into an error state which will terminate all wavefronts associated with dispatch packets currently executing on it whether or not they are halted. The following text provides more details.

When a machine instruction is executed by the enabled work-items of a wavefront, the wavefront must be halted if any enabled work-item of the wavefront signals an exception. The machine instruction that signaled the exception is termed the *excepting machine instruction*. If a wavefront is halted, it does not affect the execution of other wavefronts.

Execution is halted at a machine instruction boundary; this is not required to be at an HSAIL operation boundary. The machine instruction that a wavefront was executing when the wavefront was halted is termed the *halted machine instruction*.

The halted machine instruction for all work-items that executed an excepting machine instruction must be the excepting machine instruction. The work-items that execute an excepting machine instruction are termed the excepting work-items. The wavefronts containing the excepting work-items are termed the excepting wavefronts. The enabled work-items of the excepting wavefronts that are not excepting work-items are termed non-excepting work-items.

The HSA Runtime debugger interface provides the ability to also halt other wavefronts. For example, it could halt all the other wavefronts currently executing the same kernel dispatch as the excepting wavefronts. These wavefronts are termed non-excepting wavefronts. The work-items they contain are also termed non-excepting work-items. This functionality might be useful to a high level debugger.

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 signaled.

A single excepting work-item may generate more than one exception. All exceptions enabled for the BREAK policy must be included, together with any other exceptions that the excepting instruction signaled. For the debugtrap exception, the value of the work-item's source operand must also be specified.

All non-excepting work-items, whether in an excepting wavefront or nonexcepting 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 work-items are enabled, and for enabled work-items which 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.

The HSA Runtime debugger interface can be used to modify the machine state of work-items in a halted wavefront. This includes updating of machine registers, writing to memory, setting the DETECT exception bits, updating any other machine state. For enabled work-items it also includes changing the work-item to indicate that it is as if the excepting machine instruction had completed execution.

The HSA Runtime debugger interface can be used to resume the execution of halted wavefronts. For each wavefront resumed, 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 wavefront continue execution with the next machine instruction.

Chapter 13

Directives

This chapter describes the directives.

13.1 extension Directive

The `extension` directive enables additional opcodes that can be used in the module. It must appear after the `version` directive but before the first HSAIL module statement (see [4.3 Module \(p. 38\)](#)). This allows a finalizer to identify all extensions by only inspecting the directives at the start of a module: it does not need to scan the entire module.

An `extension` directive applies to all kernels and functions in the module. An extension only applies to the module in which it appears. Other modules are allowed to have different extensions.

The syntax is:

```
extension string
```

The string is the name of the extension. An extension with an empty string is ignored.

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";
```

If an HSA component does not support an extension that is enabled in a module, then the finalizer for that HSA component must report an error.

A runtime operation can be used to query if an HSA component supports a particular extension, and to get the list of extensions it supports.

13.1.1 extension CORE

The "CORE" extension specifies that no extensions are allowed in the module in which it appears:

```
extension "CORE";
```

If the "CORE" extension directive is present, the only other extension directives allowed in the same module are other "CORE" directives. Otherwise, multiple non-"CORE" extension directives are allowed in a module: a finalizer must enable all opcodes for all extension directives that specify the vendor of the finalizer for the module.

13.1.2 extension IMAGE

The "IMAGE" extension specifies that the HSAIL image operations defined in Chapter 7 are allowed in the module in which it appears:

```
extension "IMAGE";
```

If the "IMAGE" extension directive is not present, then the following HSAIL operations are not allowed in the module in which it appears:

- `rdimage`
- `ldimage`
- `stimage`
- `queryimage`
- `querysampler`

In addition, the data types of `roimg`, `woimg`, `rwimg` and `samp` are also not allowed. They cannot be used: to declare and define variables or specify initializers; cannot be used in kernel, function and signature declarations; and cannot be used with the `ld`, `st` and `mov` operations, including passing function arguments.

The `image` segment can also not be used with the `memfence` operation.

13.1.3 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_u32 $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, it must return an error when finalizing the module.

13.2 loc Directive

Use the loc directive to specify the line and column number in a source file that corresponds to the following HSAIL. The source line number specified is not incremented in response to new lines in the following HSAIL text. Instead, the same source position applies to all the following HSAIL, regardless of line breaks, up to the next loc directive or end of the module.

The syntax is:

```
loc linenum [column] [filename];
```

linenum is the line number within that file. It is specified as an integer constant of type u64 and must be in the right-open interval [0, 2³²). WAVESIZE is not allowed. The first line of the file is numbered 1. A value of 0 is not allowed.

column is an optional column within the line. It is specified as an integer constant of type u64 and must be in the right-open interval [0, 2³²). WAVESIZE is not allowed. The first column of a line is numbered 1. A column of 0 is not allowed. If omitted defaults to 1.

filename is a string surrounded by quotes. If omitted defaults to the file name used in the nearest preceding loc directive within the module that does specify a file name, or the empty string if there is no such loc directive.

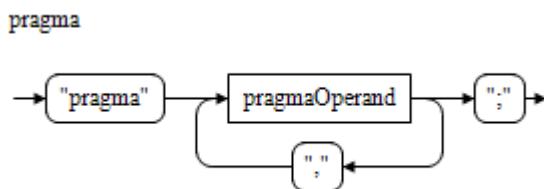
For example:

```
loc 20 "file.hsail"; // Line 20, column 1 in file with name file.hsail.
loc 20 10 "file.hsail"; // Line 20, column 10 in file with name file.hsail.
loc 30; // Line 30, column 1 in the file mentioned by the previous loc directive.
```

13.3 pragma Directive

The `pragma` directive can be used to pass information to the finalizer, or used by other components that process HSAIL. For example, it could be used to encode information about kernel arguments and symbolic variable initializers that is used by a high level language runtime.

Figure 15–1 pragma Syntax Diagram



The `pragma` directive can appear anywhere in the HSAIL code that an annotation can appear (see [4.3.1 Annotations \(p. 40\)](#)). A pragma that is not recognized by the finalizer or other HSAIL consumer must be ignored and does not cause an error.

A pragma operand can be a string, a integer constant of data type `u64` or an identifier. `WAVESIZE` is not allowed.

Note, that any identifier used in a pragma operand must be in scope. For example, pragmas that reference formal argument identifiers must be in the code block of the corresponding function or kernel; and pragmas that reference the identifier of a variable must come after a declaration or definition of the variable. See [4.6.2 Scope \(p. 63\)](#).

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 module scope directives. It also allows an HAIL linker to process functions independently, because no pragmas outside the function can alter its behavior.

The finalizer or other HSAIL consumer 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 or other HSAIL consumer implementation determines the interpretation of `pragma` directives.

You cannot use this directive to change the semantics of the HSAIL virtual machine.

Example

```

global_u32 &i[2];
global_u64 &i_p; // int *i_p = &i[1];
pragma "rti", "init", "symbolic", &i_p, &i, 4;

kernel &pragma_example(kernarg_u64 %float_buf)
{
    pragma "rti", "kernel", "arg", %float_buf, "*float";
    // ...
};

```

13.4 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 15–1 \(p. 295\)](#).

Table 15–1 Control Directives for Low-Level Performance Tuning

Directive	Arguments
enablebreakexceptions	<i>exceptionsNumber</i>
enabledetectexceptions	<i>exceptionsNumber</i>
maxdynamicgroupsize	<i>size</i>
maxflatgridsize	<i>count</i>
maxflatworkgroupsize	<i>count</i>
requestedworkgroupspperc	<i>nc</i>
requireddim	<i>nd</i>
requiredgridsize	<i>nx, ny, nz</i>
requiredworkgroupsize	<i>nx, ny, nz</i>
requirenopartialworkgroups	

Explanation of Arguments

<i>exceptionsNumber</i> : 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 <code>u32</code> . <code>WAVESIZE</code> is not allowed. For the Base profile, the five exceptions are not supported and bits 0 to 4 must be 0 (see 16.2.1 Base Profile Requirements (p. 309)).
<i>size</i> : The number of bytes. Must be an immediate value of data type <code>u32</code> . <code>WAVESIZE</code> is not allowed.
<i>count</i> : The number of work-items. Must be an immediate value, greater than 0, of data type <code>u32</code> for <code>maxflatworkgroupsize</code> and <code>u64</code> for <code>maxflatgridsize</code> . <code>WAVESIZE</code> is allowed.
<i>nc</i> : The number of work-groups. Must be an immediate value, greater than 0, of data type <code>u32</code> . <code>WAVESIZE</code> is not allowed.
<i>nd</i> : The number of dimensions. Must be an immediate value, with the value 1, 2, or 3, of data type <code>u32</code> . <code>WAVESIZE</code> is not allowed.
<i>nx</i> , <i>ny</i> , <i>nz</i> : The size for the X, Y and Z dimensions of the grid or work-group respectively. Must be an immediate value, greater than 0, of data type <code>u32</code> for <code>requiredworkgroupsize</code> and <code>u64</code> for <code>requiredgridsize</code> . <code>WAVESIZE</code> is allowed.

See also [18.3.8 BrigControlDirective \(p. 322\)](#).

The control directives must appear in the code block of a kernel or function and only apply to that kernel or function. This allows an HAIL finalizer and linker to process kernels and functions independently, since control directives in one kernel or function can not alter another.

Control directives must appear before the first HSAIL code block definition or statement (see [4.3.5 Code Block \(p. 46\)](#)). This allows a finalizer to locate all control directives for a kernel or function without having to read the entire code block.

The rules for 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 [12.3 Hardware Exception Policies \(p. 287\)](#). *exceptionsNumber* must be an immediate value of data type `u32` and `WAVESIZE` is not allowed. 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 or indirect function code block and the enable break exceptions argument specified when the finalizer is invoked. The setting applies to the kernel or indirect function being finalized and all functions it calls through non-indirect calls in the same module.

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 that are defined in other modules, or indirect functions called by an indirect call regardless of what module in which they are defined, unless they contain `enablebreakexceptions` control directives or the finalizer was invoked specifying them in the enable break exceptions argument when they were finalized.

The BREAK exception policy for the five exceptions is not supported for the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)). The finalizer is required to report an error if any of these five exceptions are enabled for the BREAK policy either through an `enablebreakexceptions` control directive for the kernel or any of the functions it calls directly or indirectly that are being finalized, or the enable break exceptions argument specified when the finalizer is invoked. See [4.19.5 Floating Point Exceptions \(p. 94\)](#).

enabledetectexceptions

Specifies the set of exceptions that must be enabled for the DETECT policy. See [12.3 Hardware Exception Policies \(p. 287\)](#). *exceptionsNumber* must be an immediate value of data type `u32` and `WAVESIZE` is not allowed. 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 or indirect function code block and the enable detect exceptions argument specified when the finalizer is invoked. The setting applies to the kernel or indirect function being finalized and all functions it calls through non-indirect calls in the same module.

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 that are defined in other modules, or indirect functions called by an indirect call regardless of what module in which they are defined, unless they contain `enabledetectexceptions` control directives or the finalizer was invoked specifying them in the enable detect exceptions argument when they were finalized.

The DETECT exception policy for the five exceptions is not supported for the Base profile (see [16.2.1 Base Profile Requirements \(p. 309\)](#)). The finalizer is required to report an error if any of these five exceptions are enabled for the BREAK policy either through an `enabledetectexceptions` control directive for the kernel or any of the functions it calls directly or indirectly that are being finalized, or the enable break exceptions argument specified when the finalizer is invoked. See [4.19.5 Floating Point Exceptions \(p. 94\)](#).

maxdynamicgroupsize

Specifies the maximum number of bytes of dynamic group memory (see [4.20 Dynamic Group Memory Allocation \(p. 95\)](#)) 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, and `WAVESIZE` is not allowed. 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 number of work-items that will be in the grid when the kernel is dispatched. *count* must be an immediate value of data type `u64`, with a value greater than 0, and `WAVESIZE` is allowed. 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 size that has a product of the X, Y, and Z components 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 for maximum absolute grid size must 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, and `WAVESIZE` is allowed. 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 size that has a product of the X, Y, and Z components 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 for maximum absolute work-group size must 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, and `WAVESIZE` is not allowed. 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, and `WAVESIZE` is not allowed. 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 *nd*.

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; and all other dimensions must be non-0.

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 components of the grid size correspond to *nx*, *ny*, *nz* respectively. They must be an immediate value of data type `u64`, with a value greater than 0, and `WAVESIZE` is allowed. 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 size that does not match these values. 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 `gridgroups` 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 `requiredgridsize` is specified (either by a control directive or when the finalizer was invoked), it must be consistent with `requiredworkgroupsize` and `requireddim` if specified: invalid dimensions must be 1, and valid dimension must not be 0.

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 the value specified by `maxflatgridsize`.

requiredworkgroupsize

Specifies the work-group size that will be used when the kernel is dispatched. The X, Y, Z components of the work-group size correspond to *nx*, *ny*, *nz* respectively. They must be an immediate value of data type `u32`, with a value greater than 0, and `WAVESIZE` is allowed. 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 size that does not match these values.

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 `requiredworkgroupsize` is specified (either by a control directive or when the finalizer was invoked), it must be consistent with `requiredgridsize` and `requireddim` if specified: invalid dimensions must be 1, and valid dimension must not be 0.

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 the 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 `requirenopartialworkgroups` is specified when the finalizer is invoked, the kernel behaves as if the `requirenopartialworkgroups` control directive has been specified.

Chapter 14

version Statement

This chapter describes the `version` statement.

14.1 Syntax of the version Statement

The `version` statement specifies the HSAIL version, the profile and target architecture required by the code in a module.

A single `version` statement must appear at the top of each module, optionally preceded by only annotations (see [4.3.1 Annotations \(p. 40\)](#)).

The syntax is:

```
version major : minor : profile : machine_model
```

`major`

An integer constant of type `u64` and must be in the right-open interval $[0, 2^{32})$.
`WAVESIZE` is not allowed.

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`

An integer constant of type `u64` and must be in the right-open interval $[0, 2^{32})$.
`WAVESIZE` is not allowed.

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 16 Profiles \(p. 307\)](#)). 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 16 Profiles \(p. 307\)](#).

machine_mode1

Specifies which machine model is used during finalization (see [2.9 Small and Large Machine Models \(p. 24\)](#)). 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 [2.9 Small and Large Machine Models \(p. 24\)](#).

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

Chapter 15

Libraries

This chapter describes how to write HSAIL code for libraries.

15.1 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 program linkage.
- Every non-externally-callable routine in the library should have module linkage.
- Every HSAIL source file that contains a call to a library should have a declaration specifying program linkage for each library function that it will call.

See [4.12 Linkage \(p. 77\)](#).

15.2 Library Example

An example of library code is shown below:

```

version 1:0:$full:$small;
group_f32 &xarray[100]; // the library gets part of this array
decl prog function &libfoo(arg_u32 %res) (arg_u32 %sptr);
decl function &a() (arg_u32 %formal);

kernel &main()
{
{
    arg_u32 %in;
    arg_u32 %out;
    // give the library part of the group memory
    lda_group_u32 $s1, [&xarray][4];
    st_arg_u32 $s1, [%in];
    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
}
// ...
};

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

decl function &l1() (arg_u32 %input);
prog 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);
}
// ...
};

function &l1() (arg_u32 %input)
{
    ld_arg_u32 $s6, [%input];
    // library passed address in group memory is now $s6
    // ...
};

```

Chapter 16

Profiles

This chapter describes the HSAIL profiles.

16.1 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 module.
- An application is not allowed to mix profiles.
- The required profile must be selected by a modifier on the `version` statement. See [14.1 Syntax of the version Statement \(p. 303\)](#).
- 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:

- The integer and bit types and all operations on the types.
- The 16-bit floating-point type (`f16`) 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 [4.19.1 Floating-Point Numbers \(p. 91\)](#).

- For all floating-point arithmetic operations (see [5.11 Floating-Point Arithmetic Operations \(p. 129\)](#)); `cmp` with floating-point sources (see [5.17 Compare \(cmp\) Operation \(p. 145\)](#)); and `cvt` with a floating-point source type (see [5.18 Conversion \(cvt\) Operation \(p. 150\)](#)):
 - Must generate invalid operation exceptions for signaling NaN sources. Additionally, the signalling comparison forms of the `cmp` operation must also generate invalid operation exceptions for quiet NaN sources.
 - Must not return a signaling NaN.

Note, this does not apply to floating-point bit operations (see [5.12 Floating-Point Bit Operations \(p. 133\)](#)) or native floating-point operations (see [5.13 Native Floating-Point Operations \(p. 136\)](#)).

- The 32-bit floating-point type (`f32`) and all operations according to the declared profile.
- The packed types and all operations on the types with the exception of `f64x2`.
- Handling of `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 [16.2 Profile-Specific Requirements \(p. 308\)](#).

See [Appendix A Limits \(p. 393\)](#) for details on limits that apply to both profiles.

16.2 Profile-Specific Requirements

This section describes the requirements that an implementation must adhere to in order to claim support of the Base or Full profile.

16.2.1 Base Profile Requirements

Implementations of the Base profile are required to provide the following support:

- On all supported floating-point types:
 - Must provide an IEEE/ANSI Standard 754-2008 correctly rounded to nearest even result for add/subtract/multiply/fma operations.
 - Does not support the 64-bit floating-point type (`f64`), 64-bit packed floating-point type (`2xf64`), double-precision floating point constants, nor any operations on the types.
 - 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 `f tz` modifier (when `f tz` is valid).
 - For all floating-point arithmetic operations (see [5.11 Floating-Point Arithmetic Operations \(p. 129\)](#)) and `cvt` with a floating-point source and destination type (see [5.18 Conversion \(cvt\) Operation \(p. 150\)](#)), if one or more inputs are NaNs, the result must be a quiet NaN. The actual quiet NaN is implementation defined and is not required to be propagated from a source operand to the destination operand (see [4.19.4 Not A Number \(NaN\) \(p. 93\)](#)).
 - The exception to this rule is `min` and `max`, when one of the inputs is a quiet NaN and the other is a number, in which case the result is the number.
 - Neither the DETECT nor BREAK exception policies (see [12.3 Hardware Exception Policies \(p. 287\)](#)) for the five exceptions specified in [12.2 Hardware Exceptions \(p. 285\)](#) are supported. See [4.19.5 Floating Point Exceptions \(p. 94\)](#).

16.2.2 Full Profile Requirements

Implementations of the Full profile are required to provide the following support:

- On all supported floating-point types:
 - Must provide an IEEE/ANSI Standard 754-2008 correctly rounded result for add/subtract/multiply/divide/fma and square root operations.
 - Must support the 64-bit floating-point type (`f64`), 64-bit packed floating-point type (`2xf64`), double-precision floating point constants and all operations on the types.
 - Must support all four HSAIL-defined rounding modes.
 - Must support floating-point subnormal values.
 - Must support the `f tz` modifier and IEEE/ANSI Standard 754-2008 gradual underflow.
 - For all floating-point arithmetic operations (see [5.11 Floating-Point Arithmetic Operations \(p. 129\)](#)) and `cvt` with a floating-point source and destination type (see [5.18 Conversion \(cvt\) Operation \(p. 150\)](#)), if one or more inputs are NaNs, the result must be a quiet NaN. The quiet NaN produced must be propagated from a source operand to the destination operand as defined in [4.19.4 Not A Number \(NaN\) \(p. 93\)](#).
 - The exception to this rule is `min` and `max`, when one of the inputs is a quiet NaN and the other is a number, in which case the result is the number.
- Must support the DETECT and BREAK exception policies (see [12.3 Hardware Exception Policies \(p. 287\)](#)) for the five exceptions specified in [12.2 Hardware Exceptions \(p. 285\)](#).

Chapter 17

Guidelines for Compiler Writers

This chapter provides guidelines for compiler writers.

17.1 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 ISA registers and remove the spills.

17.2 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.

17.3 Functions

Functions are often quite expensive. High-level compilers may want to inline functions. However, consideration should be given to code size which can impact instruction cache performance.

Common performance ratios might be: one “call” takes as long as 1000 “adds,” one indirect call takes as long as 10,000 “adds.”

Recursion can require significant private segment space to need to be allocated to accommodate the stack frames of the total call depth of the recursive functions. Each stack frame can potentially require space for:

- function scope private and spill segment definitions
- formal argument arg segment definitions
- any space needed for saved HSAIL or ISA registers due to calls
- any other finalizer introduced temporaries including spilled ISA registers

Given that a typical HSAIL implementation is able to execute thousands of work-items simultaneously, programs with recursive functions can frequently run out of private segment space.

To avoid recursive functions, an application could use an array for a stack with a size known to be large enough for the maximum depth of recursion. A simple high-level compiler could also perform tail recursion optimizations. These techniques can enable additional inlining.

17.4 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.

17.5 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.

17.6 Control Flow Optimization

The requirement that divergent control flow must reconverge no later than the immediate post-dominator (see [2.12 Divergent Control Flow \(p. 26\)](#)) makes certain control flow optimizations illegal. For example, certain basic block cloning optimization can affect the set of active work-items in a wavefront and so alter when control flow reconverges. If allowed, this could result in operations that are involved in cross-lane interaction, such as barrier and cross-lane operations (see [Chapter 9 Parallel Synchronization and Communication Operations \(p. 235\)](#)), to behave differently.

Consider the following pseudo HSAIL example:

```
if (x || y) {
    A;
    cross-lane-operation;
    B;
    if (x) {
        C;
    }
}
```

If it were legal to be transformed into the following pseudo machine code control flow, then the cross-lane operations would execute differently as the set of active lanes has been changed:

```
if (x) {
    A;
    cross-lane-operation;
    B;
    C;
} else if (y) {
    A;
    cross-lane-operation;
    B;
}
```

In general it is not legal to clone operations that can result in cross-lane communication within a wavefront as that can change the execution behavior. Such operations are:

- atomic memory (see [6.5 Atomic Memory Operations \(p. 179\)](#))
- memfence (see [6.9 Memory Fence \(memfence\) Operation \(p. 193\)](#))
- signals (see [6.8 Notification \(signal\) Operations \(p. 187\)](#))
- cross-lane operations (see [Chapter 9 Parallel Synchronization and Communication Operations \(p. 235\)](#))
- barrier and wavebarrier (see [9.1 Barrier Operations \(p. 235\)](#))
- fbarriers (see [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#))
- clock (see [11.4 Miscellaneous Operations \(p. 280\)](#))
- cleardetectexcept, getdetectexcept and setdetectexcept (see [11.2 Exception Operations \(p. 274\)](#))
- or calls to functions that contain any of these (see [Chapter 10 Function Operations \(p. 253\)](#)).

17.7 Memory Access

The finalizer is free to remove and merge loads and stores to memory if this does not change the answer of the single work-item, including any communication with other work-items and agents.

The private, spill and arg segments can only be accessed by a single work-item so can be optimized by only considering the single work-item accesses.

The readonly and kernarg segments, read-only image data, global segment variables declared as `const`, and addresses loaded by `1d` operations with the `const` modifier, cannot be changed during the execution of a work-item, so the accesses of other work-items and agents do not have to be considered.

Ordinary memory operations to the group and global segment, and non-atomic image operations to read-write images, cannot affect, or be affected by, other work-items or agents, except by an intervening synchronizing memory operation or memory fence, as that would constitute a data race and so be undefined.

Atomic memory operations, or ordinary memory operations that are made visible to other work-items or agents through synchronizing memory operations, memory fences or packet processor fences, cannot in general be removed even if their results are not used in the single work-item, as they may be used by other work-items and agents. However, it may still be possible to eliminate and merge multiple such adjacent

operations if it can only produce legal execution orders of the original program. For example, multiple adjacent relaxed atomic stores to the same location could be collapsed into one since the memory model does not require that other work-items or agents see every value of a relaxed atomic, just values that advance in the modification order of the location within finite time.

17.8 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, or have some other known alignment, it should be marked with the `align` 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.

17.9 Constant Access

If a load is known to access memory locations that will not be changed during the lifetime of the variable, then they should be marked with the `const` modifier. On some implementations, knowing a load is accessing constant memory might be more efficient. It is undefined if a memory load marked as constant is changed during the execution of any kernels that are part of the program: on some implementations this might result in incorrect values being loaded. See [6.3 Load \(ld\) Operation \(p. 171\)](#).

For similar reasons, if a variable is known to never have its value changed after it has been created and initialized, then it should be marked with the `const` qualifier. See [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

An HSAIL global or readonly segment variable definition marked with the `const` qualifier are required to have an initializer. The finalizer can inline the value of these variable initializers. However, if the variable is only declared in HSAIL, and defined by using the HSA runtime, then the finalizer must not inline the initial value as it may change on each execution of the application.

17.10 Segment Address Conversion

When converting between segment and flat addresses, if it is known that the address will not be the null pointer value, then the operations should be marked with the `nonnull` modifier. On some implementations, knowing an address will not be the null pointer value might be more efficient. It is undefined if a segment address conversion operation marked as `nonnull` is given a null pointer value: on some implementations this might result in incorrect values.

17.11 When to Use Flat Addressing

In general, segment addressing is faster than flat-address addressing. For example:

- In the large machine model a flat-address is 64 bits, but a private or group segment address is always only 32 bits. This can result in higher register pressure as the address computations have to be done in 64 bit registers instead of 32 bit registers. In turn this can result in lower performance due to more spilling, or fewer waves executing on a compute unit due to increased register usage.
- On some implementations, accessing memory with a flat address may result in issuing a request to multiple memory units since it could actually access any of them. In such implementations, each memory unit determines if the flat address references the segment they service and only returns a result if it does. This can reduce performance as the memory units cannot operate concurrently to service multiple segment address requests to different segments.

However, the group and private segments are limited to 4 GiB in size.

A high-level compiler should attempt to identify where a segment address can be used to avoid these performance issues.

17.12 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.

17.13 Exceptions

If any exceptions are enabled for the BREAK policy (see [12.3 Hardware Exception Policies \(p. 287\)](#)), 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 that can generate an exception cannot be transformed into a set of instructions that produce the same result but do not generate the exception if:
 - The result is visible to other kernel dispatches or other agents.
 - The result is used in a computation that is visible to other kernel dispatches or other agents.

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.
- The result is not visible, and is not used in a computation that is visible, to other kernel dispatches or other agents. This is true even if the side effects of the exception is visible through the BREAK policy being enabled, or the DETECT policy being enabled and the `getdetectexcept` operation being used.
- 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 `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 18

BRIG: HSAIL Binary Format

This chapter describes BRIG, the HSAIL binary format.

18.1 What Is BRIG?

BRIG is a binary representation of the textual representation of HSAIL. It is an in-memory binary representation, not a file based container format. However, a file container format may choose to use the binary representation of the BRIG segments as part of its specification.

The BRIG representation describes all aspects of the textual representation of HSAIL except:

- The textual layout. White space between lexical tokens is not preserved. See [4.4 Source Text Format \(p. 59\)](#).
- Whether a file name was omitted in a `loc` directive.
- Whether an address expression has an explicit `0` offset.
- 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 operation modifier values that are the default value used when the modifier is omitted (such as for `align`, `equiv`, `width`, and `near` or `zeroi` rounding mode).
- The order of segment scope modifiers specified in the `memfence` operation.
- The use of explicit declaration type qualifier values that are the default value used when the modifier is omitted (such as for `align`).
- 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 order of properties for image and sampler initializers.

The HSA runtime uses the BRIG binary representation in the API for the finalizer and linking services and not the textual form. However, there may be HSA runtime services for converting between the textual form and BRIG binary form.

18.2 BrigModule

`BrigModule` is the representation for a single BRIG module. See [4.3 Module \(p. 38\)](#).

Syntax is:

```
struct BrigModule {
    uint32_t sectionCount;
    BrigSectionHeader* section[1];
};
```

Fields are:

- `uint32_t sectionCount` — Number of sections in the module. Must be at least 3.
- `BrigSectionHeader* section[1]` — A variable-sized array containing pointers to the BRIG sections. Must have `sectionCount` elements. Indexed by `BrigSectionIndex` (see [18.3.29 BrigSectionIndex \(p. 332\)](#)). The first three elements must be for the following predefined sections in the following order:
 - `hsa_data` — Textual character strings and byte data used in the module. Also contains variable length arrays of offsets into other sections that are used by entries in the `hsa_code` and `hsa_operand` sections. See [18.4 hsa_data Section \(p. 338\)](#).
 - `hsa_code` — All of the directives and instructions of the module. Most entries contain offsets to the `hsa_operand` or `hsa_data` sections. Directives provide information to the finalizer, and instructions correspond to HSAIL operations which the finalizer uses to generate executable ISA code. See [18.5 hsa_code Section \(p. 339\)](#).
 - `hsa_operand` — The operands of directives and instructions in the code section. For example, immediate constants, registers and address expressions. See [18.6 hsa_operand Section \(p. 359\)](#).

HSAIL supports an arbitrary number of additional sections that can come after these predefined sections. However, the layout of these sections, beyond the standard section header, is not specified by HSAIL (see [18.3.30 BrigSectionHeader \(p. 332\)](#)). An implementation may use these additional sections to represent other information about the module. For example, they may be produced by high level language compilers or other tools, and may contain debug information, high level language runtime information and profile data.

18.2.1 Format of Entries in the BRIG Sections

Every section starts with a `BrigSectionHeader` which contains the section size, name and offset to the first entry. See [18.3.30 BrigSectionHeader \(p. 332\)](#).

For the predefined BRIG sections, `hsa_data`, `hsa_code` and `hsa_operand`, 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 the entries of all predefined BRIG sections 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 `hsa_code` and `hsa_operand` sections have a similar format. Entries are variable-size. Each entry starts with a `BrigBase` structure (see [18.3.6 BrigBase \(p. 321\)](#)) which consists of a 16-bit unsigned integer containing the length of the entry in bytes, followed by a 16-bit kind field indicating the entry kind. This is followed by the entry kind specific data, which is always zero padded to be a multiple of 4. 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, instructions or operands 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 `hsa_data` 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 padding 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, `BrigDirectiveControl` and `BrigOperandCodeList`) refer to a variable length list of other BRIG entries. A list is represented as a single entry in the `hsa_data` section that is an array of offsets into the `hsa_code` or `hsa_operand` sections. The byte count of these entries must always be a multiple of 4. The number of elements in the array is not stored explicitly, but is obtained by dividing the byte count of the `hsa_data` section entry by 4.

18.3 Support Types

This section defines the various types and enumerations used in the structures present in each BRIG section.

18.3.1 Section Offsets

The following types are used to reference an entry in a specific section. The value is the byte offset relative to the start of the section to the beginning of the referenced entry. The value 0 is reserved to indicate that the offset does not reference any entry.

```
typedef uint32_t BrigDataOffset32_t;
typedef uint32_t BrigCodeOffset32_t;
typedef uint32_t BrigOperandOffset32_t;
```

For `hsa_data` section offsets, the following types are used to indicate the contents of the `hsa_data` section entry referenced:

```
typedef BrigDataOffset32_t BrigDataOffsetString32_t;
typedef BrigDataOffset32_t BrigDataOffsetCodeList32_t;
typedef BrigDataOffset32_t BrigDataOffsetOperandList32_t;
```

- `BrigDataOffsetString32_t` — The entry contains a textual string.
- `BrigDataOffsetCodeList32_t` — The entry contains an array of `BrigCodeOffset32_t` values. The `byteCount` of the entry must be exactly $(4 * \text{number of array elements})$.
- `BrigDataOffsetOperandList32_t` — The entry contains an array of `BrigOperandOffset32_t` values. The `byteCount` of the `hsa_data` section entry must be exactly $(4 * \text{number of array elements})$.

18.3.2 BrigAlignment

`BrigAlignment` is used to specify the alignment of a memory address. Because the alignment must be a power of 2 between 1 and 256 inclusive, only enumerations for the power of 2 values are present, and they are numbered as $\log_2(n) + 1$ of the value. The value `BRIG_ALIGNMENT_1` means any byte boundary, `BRIG_ALIGNMENT_2` is any even byte boundary, `BRIG_ALIGNMENT_4` is any multiple of four, and so forth. For more information, see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

```
typedef uint8_t BrigAlignment8_t;
enum BrigAlignment {
    BRIG_ALIGNMENT_NONE = 0,
    BRIG_ALIGNMENT_1 = 1,
    BRIG_ALIGNMENT_2 = 2,
    BRIG_ALIGNMENT_4 = 3,
    BRIG_ALIGNMENT_8 = 4,
    BRIG_ALIGNMENT_16 = 5,
    BRIG_ALIGNMENT_32 = 6,
    BRIG_ALIGNMENT_64 = 7,
    BRIG_ALIGNMENT_128 = 8,
    BRIG_ALIGNMENT_256 = 9
};
```

18.3.3 BrigAllocation

`BrigAllocation` is used to specify the memory allocation for variables. For more information, see [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

```
typedef uint8_t BrigAllocation8_t;
enum BrigAllocation {
    BRIG_ALLOCATION_NONE = 0,
    BRIG_ALLOCATION_PROGRAM = 1,
    BRIG_ALLOCATION_AGENT = 2,
    BRIG_ALLOCATION_AUTOMATIC = 3
};
```

18.3.4 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 [18.3.24 BrigRound \(p. 330\)](#).
- **BRIG_ALU_FTZ** — A bit mask that can be used to select the setting for the `ftz` (floating-point flush subnormals to zero) modifier. If the operation does not support the `ftz` modifier, then must be 0. Otherwise, 0 value means it is absent and a non-0 value means it is present.

18.3.5 BrigAtomicOperation

`BrigAtomicOperation` is used to specify the type of atomic memory, signal and atomic image operations. For more information, see [6.5 Atomic Memory Operations \(p. 179\)](#) and [6.8 Notification \(signal\) Operations \(p. 187\)](#).

```
typedef uint8_t BrigAtomicOperation8_t;
enum BrigAtomicOperation {
    BRIG_ATOMIC_ADD = 0,
    BRIG_ATOMIC_AND = 1,
    BRIG_ATOMIC_CAS = 2,
    BRIG_ATOMIC_EXCH = 3,
    BRIG_ATOMIC_LD = 4,
    BRIG_ATOMIC_MAX = 5,
    BRIG_ATOMIC_MIN = 6,
    BRIG_ATOMIC_OR = 7,
    BRIG_ATOMIC_ST = 8,
    BRIG_ATOMIC_SUB = 9,
    BRIG_ATOMIC_WRAPDEC = 10,
    BRIG_ATOMIC_WRAPINC = 11,
    BRIG_ATOMIC_XOR = 12,
    BRIG_ATOMIC_WAIT_EQ = 13,
    BRIG_ATOMIC_WAIT_NE = 14,
    BRIG_ATOMIC_WAIT_LT = 15,
    BRIG_ATOMIC_WAIT_GTE = 16,
    BRIG_ATOMIC_WAITTIMEOUT_EQ = 17,
    BRIG_ATOMIC_WAITTIMEOUT_NE = 18,
    BRIG_ATOMIC_WAITTIMEOUT_LT = 19,
    BRIG_ATOMIC_WAITTIMEOUT_GTE = 20
};
```

18.3.6 BrigBase

All entries in the `hsa_code` and `hsa_operand` sections start with the `BrigBase` structure.

Syntax is:

```
struct BrigBase {
    uint16_t byteCount;
    BrigKind16_t kind;
};
```

Fields are:

- `uint16_t byteSize` — Size of the entry in bytes, including the `BrigBase` structure. Must be a multiple of 4.
- `BrigKind16_t kind` — Can be any member of the `BrigKind` enumeration indicating the kind of this entry. Must only be `BRIG_KIND_DIRECTIVE_*` or `BRIG_KIND_INST_*` for entries in the `hsa_code` section, and `BRIG_KIND_OPERAND_*` for entries in the `hsa_operand` section. See [18.3.14 BrigKind \(p. 325\)](#).

18.3.7 BrigCompareOperation

`BrigCompareOperation` is used to specify the type of compare operation. For more information, see [5.17 Compare \(cmp\) Operation \(p. 145\)](#).

```
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
};
```

18.3.8 BrigControlDirective

`BrigControlDirective` is used to specify the type of control directive. For more information, see [13.4 Control Directives for Low-Level Performance Tuning \(p. 295\)](#).

```

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
};

```

18.3.9 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_DEFINITION = 1
};

```

- `BRIG_EXECUTABLE_DEFINITION` — 0 means this is a declaration only and has no code block; 1 means this is a definition and has a code block.

See [18.5.1.5 BrigDirectiveExecutable \(p. 341\)](#).

18.3.10 BrigImageChannelOrder

`BrigImageChannelOrder` is used to specify the order of image components. For more information, see [7.14.1 Channel Order \(p. 201\)](#).

```

typedef uint8_t BrigImageChannelOrder8_t;
enum BrigImageChannelOrder {
    BRIG_CHANNEL_ORDER_A = 0,
    BRIG_CHANNEL_ORDER_R = 1,
    BRIG_CHANNEL_ORDER_RX = 2,
    BRIG_CHANNEL_ORDER_RG = 3,
    BRIG_CHANNEL_ORDER_RXG = 4,
    BRIG_CHANNEL_ORDER_RA = 5,
    BRIG_CHANNEL_ORDER_RGB = 6,
    BRIG_CHANNEL_ORDER_RGBX = 7,
    BRIG_CHANNEL_ORDER_RGBA = 8,
    BRIG_CHANNEL_ORDER_BGRA = 9,
    BRIG_CHANNEL_ORDER_ARGB = 10,
    BRIG_CHANNEL_ORDER_ABGR = 11,
    BRIG_CHANNEL_ORDER_SRGB = 12,
    BRIG_CHANNEL_ORDER_SRGBX = 13,
    BRIG_CHANNEL_ORDER_SRGBA = 14,
    BRIG_CHANNEL_ORDER_SBGRA = 15,
    BRIG_CHANNEL_ORDER_INTENSITY = 16,
    BRIG_CHANNEL_ORDER_LUMINANCE = 17,
    BRIG_CHANNEL_ORDER_DEPTH = 18,
    BRIG_CHANNEL_ORDER_DEPTH_STENCIL = 19
};

```

Values 20 through 255 are available for extensions.

18.3.11 BrigImageChannelType

`BrigImageChannelType` is used to specify the image channel type. For more information, see [7.1.4.2 Channel Type \(p. 203\)](#).

```
typedef uint8_t BrigImageChannelType8_t;
enum BrigImageChannelType {
    BRIG_CHANNEL_TYPE_SNORM_INT8 = 0,
    BRIG_CHANNEL_TYPE_SNORM_INT16 = 1,
    BRIG_CHANNEL_TYPE_UNORM_INT8 = 2,
    BRIG_CHANNEL_TYPE_UNORM_INT16 = 3,
    BRIG_CHANNEL_TYPE_UNORM_INT24 = 4,
    BRIG_CHANNEL_TYPE_UNORM_SHORT_555 = 5,
    BRIG_CHANNEL_TYPE_UNORM_SHORT_565 = 6,
    BRIG_CHANNEL_TYPE_UNORM_SHORT_101010 = 7,
    BRIG_CHANNEL_TYPE_SIGNED_INT8 = 8,
    BRIG_CHANNEL_TYPE_SIGNED_INT16 = 9,
    BRIG_CHANNEL_TYPE_SIGNED_INT32 = 10,
    BRIG_CHANNEL_TYPE_UNSIGNED_INT8 = 11,
    BRIG_CHANNEL_TYPE_UNSIGNED_INT16 = 12,
    BRIG_CHANNEL_TYPE_UNSIGNED_INT32 = 13,
    BRIG_CHANNEL_TYPE_HALF_FLOAT = 14,
    BRIG_CHANNEL_TYPE_FLOAT = 15
};
```

Values 16 through 64 are available for extensions.

18.3.12 BrigImageGeometry

`BrigImageGeometry` is used to specify the number of coordinates needed to access an image. For more information, see [7.1.3 Image Geometry \(p. 199\)](#).

```
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_2DA = 4,
    BRIG_GEOMETRY_1DB = 5,
    BRIG_GEOMETRY_2DDEPTH = 6,
    BRIG_GEOMETRY_2DADEPTH = 7
};
```

Values 8 through 255 are available for extensions.

18.3.13 BrigImageQuery

`BrigImageQuery` is used to specify the image property being queried by the `queryimage` operation. For more information, see [7.5 Query Image and Query Sampler Operations \(p. 228\)](#).

```
typedef uint8_t BrigImageQuery8_t;
enum BrigImageQuery {
    BRIG_IMAGE_QUERY_WIDTH = 0,
    BRIG_IMAGE_QUERY_HEIGHT = 1,
    BRIG_IMAGE_QUERY_DEPTH = 2,
    BRIG_IMAGE_QUERY_ARRAY = 3,
    BRIG_IMAGE_QUERY_CHANNELORDER = 4,
    BRIG_IMAGE_QUERY_CHANNELTYPE = 5
};
```

18.3.14 BrigKind

BrigKind is used to indicate the kind of the entries in the `hsa_code` and `hsa_operand` sections. The enumeration values are divided into three groupings: those for directives and instructions which can only be used for entries in the `hsa_code` section; and those for operands which can only be used for entries in the `hsa_operand` section. To allow for future expansion, each grouping has a distinct range of values.

```
typedef uint16_t BrigKinds16_t;
enum BrigKind {
    BRIG_KIND_NONE = 0x0000,
    BRIG_KIND_DIRECTIVE_BEGIN = 0x1000,
    BRIG_KIND_DIRECTIVE_ARG_BLOCK_END = 0x1000,
    BRIG_KIND_DIRECTIVE_ARG_BLOCK_START = 0x1001,
    BRIG_KIND_DIRECTIVE_COMMENT = 0x1002,
    BRIG_KIND_DIRECTIVE_CONTROL = 0x1003,
    BRIG_KIND_DIRECTIVE_EXTENSION = 0x1004,
    BRIG_KIND_DIRECTIVE_FARRIER = 0x1005,
    BRIG_KIND_DIRECTIVE_FUNCTION = 0x1006,
    BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION = 0x1007,
    BRIG_KIND_DIRECTIVE_KERNEL = 0x1008,
    BRIG_KIND_DIRECTIVE_LABEL = 0x1009,
    BRIG_KIND_DIRECTIVE_LOC = 0x100a,
    BRIG_KIND_DIRECTIVE_PRAGMA = 0x100b,
    BRIG_KIND_DIRECTIVE_SIGNATURE = 0x100c,
    BRIG_KIND_DIRECTIVE_VARIABLE = 0x100d,
    BRIG_KIND_DIRECTIVE_VERSION = 0x100e,
    BRIG_KIND_DIRECTIVE_END = 0x100f,
    BRIG_KIND_INST_BEGIN = 0x2000,
    BRIG_KIND_INST_ADDR = 0x2000,
    BRIG_KIND_INST_ATOMIC = 0x2001,
    BRIG_KIND_INST_BASIC = 0x2002,
    BRIG_KIND_INST_BR = 0x2003,
    BRIG_KIND_INST_CMP = 0x2004,
    BRIG_KIND_INST_CVT = 0x2005,
    BRIG_KIND_INST_IMAGE = 0x2006,
    BRIG_KIND_INST_LANE = 0x2007,
    BRIG_KIND_INST_MEM = 0x2008,
    BRIG_KIND_INST_MEM_FENCE = 0x2009,
    BRIG_KIND_INST_MOD = 0x200a,
    BRIG_KIND_INST_QUERY_IMAGE = 0x200b,
    BRIG_KIND_INST_QUERY_SAMPLER = 0x200c,
    BRIG_KIND_INST_QUEUE = 0x200d,
    BRIG_KIND_INST_SEG = 0x200e,
    BRIG_KIND_INST_SEG_CVT = 0x200f,
    BRIG_KIND_INST_SIGNAL = 0x2010,
    BRIG_KIND_INST_SOURCE_TYPE = 0x2011,
    BRIG_KIND_INST_END = 0x2012,
    BRIG_KIND_OPERAND_BEGIN = 0x3000,
    BRIG_KIND_OPERAND_ADDRESS = 0x3000,
    BRIG_KIND_OPERAND_DATA = 0x3001,
    BRIG_KIND_OPERAND_CODE_LIST = 0x3002,
    BRIG_KIND_OPERAND_CODE_REF = 0x3003,
    BRIG_KIND_OPERAND_IMAGE_PROPERTIES = 0x3004,
    BRIG_KIND_OPERAND_OPERAND_LIST = 0x3005,
    BRIG_KIND_OPERAND_REG = 0x3006,
    BRIG_KIND_OPERAND_SAMPLER_PROPERTIES = 0x3007,
    BRIG_KIND_OPERAND_STRING = 0x3008,
    BRIG_KIND_OPERAND_WAVESIZE = 0x3009,
    BRIG_KIND_OPERAND_END = 0x300a
};
```

18.3.15 BrigLinkage

`BrigLinkage` is used to specify linkage. For more information, see [4.12 Linkage \(p. 77\)](#).

```
typedef uint8_t BrigLinkage8_t;
enum BrigLinkage {
    BRIG_LINKAGE_NONE = 0,
    BRIG_LINKAGE_PROGRAM = 1,
    BRIG_LINKAGE_MODULE = 2,
    BRIG_LINKAGE_FUNCTION = 3,
    BRIG_LINKAGE_ARG = 4
};
```

18.3.16 BrigMachineModel

`BrigMachineModel` is used to specify the kind of machine model. For more information, see [2.9 Small and Large Machine Models \(p. 24\)](#).

```
typedef uint8_t BrigMachineModel8_t;
enum BrigMachineModel {
    BRIG_MACHINE_SMALL = 0,
    BRIG_MACHINE_LARGE = 1
};
```

18.3.17 BrigMemoryModifierMask

`BrigMemoryModifierMask` defines bit masks that can be used to access the modifiers for memory operations.

```
typedef uint8_t BrigMemoryModifier8_t;
enum BrigMemoryModifierMask {
    BRIG_MEMORY_CONST = 1
};
```

- `BRIG_MEMORY_CONST` — A bit mask that can be used to select the setting for the `const` modifier. A 0 value means it is absent and a non-0 value means it is present. If the operation does not support the `const` modifier, then this must be 0.

18.3.18 BrigMemoryOrder

`BrigMemoryOrder` is used to specify the memory order of an atomic memory operation. For more information, see [6.2.1 Memory Order \(p. 162\)](#).

```
typedef uint8_t BrigMemoryOrder8_t;
enum BrigMemoryOrder {
    BRIG_MEMORY_ORDER_NONE = 0,
    BRIG_MEMORY_ORDER_RELAXED = 1,
    BRIG_MEMORY_ORDER_SC_ACQUIRE = 2,
    BRIG_MEMORY_ORDER_SC_RELEASE = 3,
    BRIG_MEMORY_ORDER_SC_ACQUIRE_RELEASE = 4
};
```

18.3.19 BrigMemoryScope

`BrigMemoryScope` is used to specify the memory scope for an atomic memory, signal or memory fence operation. For more information, see [6.2.2 Memory Scope \(p. 165\)](#).

```
typedef uint8_t BrigMemoryScope8_t;
enum BrigMemoryScope {
    BRIG_MEMORY_SCOPE_NONE = 0,
    BRIG_MEMORY_SCOPE_WORKITEM = 1,
    BRIG_MEMORY_SCOPE_WAVEFRONT = 2,
    BRIG_MEMORY_SCOPE_WORKGROUP = 3,
    BRIG_MEMORY_SCOPE_COMPONENT = 4,
    BRIG_MEMORY_SCOPE_SYSTEM = 5
};
```

18.3.20 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_MOV = 43,
    BRIG_OPCODE_SHUFFLE = 44,
    BRIG_OPCODE_UNPACKHI = 45,
    BRIG_OPCODE_UNPACKLO = 46,
    BRIG_OPCODE_PACK = 47,
    BRIG_OPCODE_UNPACK = 48,
    BRIG_OPCODE_CMOV = 49,
    BRIG_OPCODE_CLASS = 50,
    BRIG_OPCODE_NCOS = 51,
    BRIG_OPCODE_NEXP2 = 52,
    BRIG_OPCODE_NFMA = 53,
    BRIG_OPCODE_NLOG2 = 54,
    BRIG_OPCODE_NRCP = 55,
    BRIG_OPCODE_NRSQRT = 56,
    BRIG_OPCODE_NSIN = 57,
    BRIG_OPCODE_NSQRT = 58,
    BRIG_OPCODE_BITALIGN = 59,
    BRIG_OPCODE_BYTEALIGN = 60,
    BRIG_OPCODE_PACKCvt = 61,
    BRIG_OPCODE_UNPACKCvt = 62,
    BRIG_OPCODE_LERP = 63,
    BRIG_OPCODE_SAD = 64,
    BRIG_OPCODE_SADHI = 65,
    BRIG_OPCODE_SEGMENTP = 66,
    BRIG_OPCODE_FTOS = 67,
}

```

```

BRIG_OPCODE_STOF = 68,
BRIG_OPCODE_CMP = 69,
BRIG_OPCODE_CVT = 70,
BRIG_OPCODE_LD = 71,
BRIG_OPCODE_ST = 72,
BRIG_OPCODE_ATOMIC = 73,
BRIG_OPCODE_ATOMICNORET = 74,
BRIG_OPCODE_SIGNAL = 75,
BRIG_OPCODE_SIGNALNORET = 76,
BRIG_OPCODE_MEMFENCE = 77,
BRIG_OPCODE_RDIIMAGE = 78,
BRIG_OPCODE_LDIIMAGE = 79,
BRIG_OPCODE_STIMAGE = 80,
BRIG_OPCODE_QUERYIMAGE = 81,
BRIG_OPCODE_QUERYSAMPLE = 82,
BRIG_OPCODE_CBR = 83,
BRIG_OPCODE_BR = 84,
BRIG_OPCODE_SBR = 85,
BRIG_OPCODE_BARRIER = 86,
BRIG_OPCODE_WAVEBARRIER = 87,
BRIG_OPCODE_ARRIVEFBAR = 88,
BRIG_OPCODE_INITFBAR = 89,
BRIG_OPCODE_JOINFBAR = 90,
BRIG_OPCODE_LEAVEFBAR = 91,
BRIG_OPCODE_RELEASEFBAR = 92,
BRIG_OPCODE_WAITFBAR = 93,
BRIG_OPCODE_LDF = 94,
BRIG_OPCODE_ACTIVELANECOUNT = 95,
BRIG_OPCODE_ACTIVELANEID = 96,
BRIG_OPCODE_ACTIVELANEMASK = 97,
BRIG_OPCODE_ACTIVELANESHUFFLE = 98,
BRIG_OPCODE_CALL = 99,
BRIG_OPCODE_SCALL = 100,
BRIG_OPCODE_ICALL = 101,
BRIG_OPCODE_LDI = 102,
BRIG_OPCODE_RET = 103,
BRIG_OPCODE_ALLOCA = 104,
BRIG_OPCODE_CURRENTWORKGROUPSIZE = 105,
BRIG_OPCODE_DIM = 106,
BRIG_OPCODE_GRIDGROUPS = 107,
BRIG_OPCODE_GRIDSIZE = 108,
BRIG_OPCODE_PACKETCOMPLETIONSIG = 109,
BRIG_OPCODE_PACKETID = 110,
BRIG_OPCODE_WORKGROUPID = 111,
BRIG_OPCODE_WORKGROUPSIZE = 112,
BRIG_OPCODE_WORKITEMABSID = 113,
BRIG_OPCODE_WORKITEMFLATABSID = 114,
BRIG_OPCODE_WORKITEMFLATID = 115,
BRIG_OPCODE_WORKITEMID = 116,
BRIG_OPCODE_CLEARDETECTEXCEPT = 117,
BRIG_OPCODE_GETDETECTEXCEPT = 118,
BRIG_OPCODE_SETDETECTEXCEPT = 119,
BRIG_OPCODE_ADDQUEUEWRITEINDEX = 120,
BRIG_OPCODE_AGENTCOUNT = 121,
BRIG_OPCODE_AGENTID = 122,
BRIG_OPCODE_CASQUEUEWRITEINDEX = 123,
BRIG_OPCODE_LDK = 124,
BRIG_OPCODE_LDQUEUEREADINDEX = 125,
BRIG_OPCODE_LDQUEUEWRITEINDEX = 126,
BRIG_OPCODE_QUEUEID = 127,
BRIG_OPCODE_QUEUEPTR = 128,
BRIG_OPCODE_STQUEUEREADINDEX = 129,
BRIG_OPCODE_STQUEUEWRITEINDEX = 130,
BRIG_OPCODE_CLOCK = 131,
BRIG_OPCODE_CUID = 132,
BRIG_OPCODE_DEBUGTRAP = 133,
BRIG_OPCODE_GROUPBASEPTR = 134,
BRIG_OPCODE_KERNARGBASEPTR = 135,
BRIG_OPCODE_LANEID = 136,
BRIG_OPCODE_MAXCUID = 137,

```

```

    BRIG_OPCODE_MAXWAVEID = 138,
    BRIG_OPCODE_NULLPTR = 139,
    BRIG_OPCODE_WAVEID = 140
};
```

18.3.21 BrigPack

`BrigPack` is used to specify the kind of packing control for packed data. For more information, see [4.14 Packing Controls for Packed Data \(p. 81\)](#).

```

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
};
```

18.3.22 BrigProfile

`BrigProfile` is used to specify the kind of profile. For more information, see [16.1 What Are Profiles? \(p. 307\)](#).

```

typedef uint8_t BrigProfile8_t;
enum BrigProfile {
    BRIG_PROFILE_BASE = 0,
    BRIG_PROFILE_FULL = 1
};
```

18.3.23 BrigRegister

`BrigRegister` is used to specify the kind of HSAIL register. For more information, see [4.7 Registers \(p. 64\)](#).

```

typedef uint16_t BrigRegisterKind16_t;
enum BrigRegisterKind {
    BRIG_REGISTER_SINGLE = 0,
    BRIG_REGISTER_DOUBLE = 1,
    BRIG_REGISTER_QUAD = 2
};
```

18.3.24 BrigRound

`BrigRound` is used to specify rounding. For more information, see [4.19.2 Rounding \(p. 92\)](#) and [5.18.3 Rules for Rounding for Conversions \(p. 152\)](#).

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,
    BRIG_ROUND_INTEGER_SIGNALLING_NEAR_EVEN = 13,
    BRIG_ROUND_INTEGER_SIGNALLING_ZERO = 14,
    BRIG_ROUND_INTEGER_SIGNALLING_PLUS_INFINITY = 15,
    BRIG_ROUND_INTEGER_SIGNALLING_MINUS_INFINITY = 16,
    BRIG_ROUND_INTEGER_SIGNALLING_NEAR_EVEN_SAT = 17,
    BRIG_ROUND_INTEGER_SIGNALLING_ZERO_SAT = 18,
    BRIG_ROUND_INTEGER_SIGNALLING_PLUS_INFINITY_SAT = 19,
    BRIG_ROUND_INTEGER_SIGNALLING_MINUS_INFINITY_SAT = 20
};
```

18.3.25 BrigSamplerAddressing

BrigSamplerAddressing is used to specify the addressing mode for the addressing field in the sampler object. For more information, see [7.1.6.2 Addressing Mode \(p. 210\)](#).

```
typedef uint8_t BrigSamplerAddressing8_t;
enum BrigSamplerAddressing {
    BRIG_ADDRESSING_UNDEFINED = 0,
    BRIG_ADDRESSING_CLAMP_TO_EDGE = 1,
    BRIG_ADDRESSING_CLAMP_TO_BORDER = 2,
    BRIG_ADDRESSING_REPEAT = 3,
    BRIG_ADDRESSING_MIRRORED_REPEAT = 4
};
```

Values 5 through 255 are available for extensions.

18.3.26 BrigSamplerCoordNormalization

BrigSamplerCoordNormalization is used to specify the setting for the coord field in the sampler object. For more information, see [7.1.6.1 Coordinate Normalization Mode \(p. 209\)](#).

```
typedef uint8_t BrigSamplerCoordNormalization8_t;
enum BrigSamplerCoordNormalization {
    BRIG_COORD_UNNORMALIZED = 0,
    BRIG_COORD_NORMALIZED = 1
};
```

18.3.27 BrigSamplerFilter

`BrigSamplerFilter` is used to specify the setting for the `filter` field in the sampler object. For more information, see [7.1.6.3 Filter Mode \(p. 211\)](#).

```
typedef uint8_t BrigSamplerFilter8_t;
enum BrigSamplerFilter {
    BRIG_FILTER_NEAREST = 0,
    BRIG_FILTER_LINEAR = 1
};
```

Values 2 through 255 are available for extensions.

18.3.28 BrigSamplerQuery

`BrigSamplerQuery` is used to specify the sampler property being queried by the `querysampler` operation. For more information, see [7.5 Query Image and Query Sampler Operations \(p. 228\)](#).

```
typedef uint8_t BrigSamplerQuery8_t;
enum BrigSamplerQuery {
    BRIG_SAMPLER_QUERY_ADDRESSING = 0,
    BRIG_SAMPLER_QUERY_COORD = 1,
    BRIG_SAMPLER_QUERY_FILTER = 2
};
```

18.3.29 BrigSectionIndex

A `BrigModule` can have a number of sections. Every module must have a data, code and operand section with the indices defined by `BrigSectionIndex`. Any additional sections have an index starting after these.

```
typedef uint32_t BrigSectionIndex32_t;
enum BrigSectionIndex {
    BRIG_SECTION_INDEX_DATA = 0,
    BRIG_SECTION_INDEX_CODE = 1,
    BRIG_SECTION_INDEX_OPERAND = 2,
    BRIG_SECTION_INDEX_BEGIN_IMPLEMENTATION_DEFINED = 3
};
```

18.3.30 BrigSectionHeader

The first entry in every BRIG section must be `BrigSectionHeader`, which consists of the section size, section name, and the offset of the first section entry.

There are no section termination flags. Any code that generates BRIG needs to correctly fill in each section's header. A section entry offset of 0 can be used to indicate no entry, since the first entry in each section starts after the header.

Syntax is:

```
struct BrigSectionHeader {
    uint32_t byteCount;
    uint32_t headerByteCount;
    uint32_t nameLength;
    uint8_t name[1];
};
```

Field is:

- `uint32_t byteCount` — Size in bytes of the section, including the size of the `BrigSectionHeader`. Must be a multiple of 4.
- `uint32_t headerByteCount` — Size of the header in bytes, which is also equal to the offset of the first entry in the section. Must be a multiple of 4.
- `uint32_t nameLength` — Length of the section name in bytes.
- `uint8_t name[1]` — Section name, `nameLength` bytes long.

The `name` field may be followed by any implementation specific data. This must be followed by sufficient zero padding bytes to make `headerByteCount` a multiple of 4.

18.3.31 BrigSegCvtModifierMask

`BrigSegCvtModifierMask` defines bit masks that can be used to access the modifiers for operations which convert between segment and flat addresses.

```
typedef uint8_t BrigSegCvtModifier8_t;
enum BrigSegCvtModifierMask {
    BRIG_SEG_CVT_NONNULL = 1
};
```

- `BRIG_SEG_CVT_NONNULL` — A bit mask that can be used to select the setting for the `nonnull` modifier. A 0 value means it is absent and a non-0 value means it is present. If the operation does not support the `nonnull` modifier, then this must be 0.

18.3.32 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.

18.3.33 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 [4.13 Data Types \(p. 79\)](#).

```
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_WOIMG = 20,
    BRIG_TYPE_RWIMG = 21,
    BRIG_TYPE_SIG32 = 22,
    BRIG_TYPE_SIG64 = 23,
    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
};

```

18.3.34 BrigUInt64

BrigUInt64 is used to represent a 64-bit unsigned integer value. The value is split into two 32-bit components to conform to the BRIG restriction that entries only require 32-bit alignment.

Syntax is:

```
struct BrigUInt64 {
    uint32_t lo;
    uint32_t hi;
};
```

Fields are:

- `uint32_t lo` — The low 32 bits of the 64-bit integer. `lo` is combined with `hi` to form a 64-bit value:
`value = (uint64_t(hi) << 32) | uint64_t(lo)`
- `uint32_t hi` — The high 32 bits of the 64-bit integer.

18.3.35 BrigVariableModifierMask

BrigVariableModifierMask defines bit masks that can be used to access properties about a variable.

```
typedef uint8_t BrigVariableModifier8_t;
enum BrigVariableModifierMask {
    BRIG_SYMBOL_DEFINITION = 1,
    BRIG_SYMBOL_CONST = 2,
    BRIG_SYMBOL_ARRAY = 4,
    BRIG_SYMBOL_FLEX_ARRAY = 8
};
```

- `BRIG_SYMBOL_DEFINITION` — 0 means this is a variable declaration; 1 means this is a variable definition.
- `BRIG_SYMBOL_CONST` — 0 means read/write; 1 means `const`. Only global or readonly segment variables can be constant.
- `BRIG_SYMBOL_ARRAY` — 0 means scalar; 1 means an array (size in `dim`).
- `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. Only the last argument of a function (see [10.4 Variadic Functions \(p. 259\)](#)) or a variable declaration can be a flexible array .

See [18.5.1.12 BrigDirectiveVariable \(p. 346\)](#).

18.3.36 BrigVersion

The literal values of `BrigVersion` define the versions of HSAIL virtual ISA and BRIG object format defined by this revision of the *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)*.

```
uint32_t BrigVersion32_t;
enum BrigVersion {
    BRIG_VERSION_HSAIL_MAJOR = 0,
    BRIG_VERSION_HSAIL_MINOR = 99,
    BRIG_VERSION_BRIG_MAJOR = 0,
    BRIG_VERSION_BRIG_MINOR = 99
};
```

- **BRIG_VERSION_HSAIL_MAJOR** — The major version of this revision of the HSAIL virtual ISA specification. This is the value used in the `version` statement `major` operand. See [Chapter 14 version Statement \(p. 303\)](#). BRIG with an HSAIL major version different from this value is not compatible with this revision of the HSAIL virtual ISA specification.
- **BRIG_VERSION_HSAIL_MINOR** — The minor version of this revision of the HSAIL virtual ISA specification. This is the value used in the `version` statement `minor` operand. See [Chapter 14 version Statement \(p. 303\)](#). BRIG is compatible with this revision of the HSAIL virtual ISA specification only if it has the same HSAIL major version and an HSAIL minor version less than or equal to this value.
- **BRIG_VERSION_BRIG_MAJOR** — The major version of this revision of the BRIG object format specification. BRIG with a BRIG major version different from this value is not compatible with this revision of the BRIG object format specification.
- **BRIG_VERSION_BRIG_MINOR** — The minor version of this revision of the BRIG object format specification. BRIG is compatible with this revision of the BRIG object format specification only if it has the same BRIG major version and a BRIG minor version less than or equal to this value.

18.3.37 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 [2.12 Divergent Control Flow \(p. 26\)](#).

```

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_16384 = 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
};

```

18.4 hsa_data Section

The `hsa_data` section must start with a `BrigSectionHeader` entry. The name of the section must be `hsa_data`. See [18.3.30 BrigSectionHeader \(p. 332\)](#).

The `hsa_data` section is used to store:

- Textual character strings used for identifiers and string operands within HSAIL.
- Value of variable initializers.
- Value of immediate operands.
- Variable length arrays of offsets into other sections that are used by entries in the `hsa_code` and `hsa_operand` sections. The number of elements in the array is determined by dividing the byte count of the entry by 4. See [18.3.1 Section Offsets \(p. 319\)](#) and [18.2.1 Format of Entries in the BRIG Sections \(p. 318\)](#).

An entry comprises both the length of the data in bytes and the actual bytes of the data.

An offset value into the `hsa_data` section references the start of the `BrigData`, not the data, which starts at `bytes` within `BrigData`.

Entries for HSAIL identifiers and string operand 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 `BrigData` 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.

To reduce the size of the `hsa_data` section it is allowed, but not required, to reference an already created `BrigData` entry, rather than create duplicate `BrigData` entries.

Syntax is:

```
struct BrigData {
    uint32_t byteCount;
    uint8_t bytes[1];
};
```

Fields are:

- `uint32_t byteCount` — Number of bytes in the data. Does not include the size `byteCount` field, or any padding bytes that have to be added to ensure the next `BrigData` starts on a 4-byte boundary. Therefore, to locate the start of the next `BrigData`, the value $((7 + \text{byteCount}) / 4) * 4$ must be added to the offset of the current `BrigData`.
- `uint8_t bytes[1]` — Variable-sized. Must be allocated with $((\text{byteCount} + 3) / 4) * 4$ elements. Any elements after `byteCount - 1` must be 0. Bytes 0 to `byteCount - 1` contain the data.

18.5 hsa_code Section

The `hsa_code` section contains the directives and instructions of the module. They appear in the same order in as they appear in the text format.

The `hsa_code` section must start with a `BrigSectionHeader` entry. The name of the section must be `hsa_code`. See [18.3.30 BrigSectionHeader \(p. 332\)](#).

All entries in the `hsa_code` section must start with a `BrigBase` structure. The `kind` field specifies kind of the entry, which also indicates if it is a directive (`BrigDirective*`) or instruction (`BrigIntr*`) entry. See [18.3.6 BrigBase \(p. 321\)](#).

The entries for directives and instructions that are part of a kernel or function code block are ordered after a `BrigDirectiveExecutable` entry for the kernel or function, and before the entry referenced by the `nextModuleEntry` field of the `BrigDirectiveExecutable` entry. Instruction entries can only be part of a code block. All other entries are module directives.

18.5.1 Directive Entries

BRIG directives corresponding to HSAIL version statement, annotations, directives, kernels, functions, signatures, variables, formal arguments, fbarriers and labels. BRIG

directives are also used to specify the start and end of an arg block. These provide information to the finalizer and other tools and do not generate code.

The kind field of the BrigBase structure at the start of every BrigDirective* must be in the right-open interval [BRIG_KIND_DIRECTIVE_BEGIN, BRIG_KIND_DIRECTIVE_END). See [18.3.14 BrigKind \(p. 325\)](#).

The table below shows the possible formats for the directives. Every directive uses one of these formats.

Table 20–1 Formats of Directives in the hsa_code Section

Name	Description
BrigDirectiveArgBlock	Start and end of an arg block. See 18.5.1.2 BrigDirectiveArgBlock (p. 340) .
BrigDirectiveComment	Comment string. See 18.5.1.3 BrigDirectiveComment (p. 341) .
BrigDirectiveControl	Assorted finalizer controls. See 18.5.1.4 BrigDirectiveControl (p. 341) .
BrigDirectiveExecutable	Describes a kernel, function or signature. See 18.5.1.5 BrigDirectiveExecutable (p. 341) .
BrigDirectiveExtension	Used to enable device-specific extensions. See 18.5.1.6 BrigDirectiveExtension (p. 343) .
BrigDirectiveFbarrier	Used for fbarrier definitions. See 18.5.1.7 BrigDirectiveFbarrier (p. 343) .
BrigDirectiveLabel	Declare a label. See 18.5.1.8 BrigDirectiveLabel (p. 344) .
BrigDirectiveLoc	Source-level line position. See 18.5.1.9 BrigDirectiveLoc (p. 344) .
BrigDirectiveNone	Special directive that is always ignored. See 18.5.1.10 BrigDirectiveNone (p. 345) .
BrigDirectivePragma	Additional information to control the finalizer and other consumers of HSAIL. See 18.5.1.11 BrigDirectivePragma (p. 345) .
BrigDirectiveVariable	Declares a variable. See 18.5.1.12 BrigDirectiveVariable (p. 346) .
BrigDirectiveVersion	HSAIL version and target information. See 18.5.1.13 BrigDirectiveVersion (p. 348) .

18.5.1.1 Declarations and Definitions in the Same Module

If the same symbol (variable, kernel, function or fbarrier) is both declared and defined in the same module, 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 module to determine if there is a definition for a symbol in the module.

18.5.1.2 BrigDirectiveArgBlock

BrigDirectiveArgBlock specifies the start and end of an arg block. See [4.3.6 Arg Block \(p. 47\)](#).

Syntax is:

```
struct BrigDirectiveArgBlock {
    BrigBase base;
};
```

Fields are:

- BrigBase base — base.kind must be BRIG_KIND_DIRECTIVE_ARG_BLOCK_END or BRIG_KIND_DIRECTIVE_ARG_BLOCK_START.

18.5.1.3 BrigDirectiveComment

`BrigDirectiveComment` is a comment string.

Syntax is:

```
struct BrigDirectiveComment {
    BrigBase base;
    BrigDataOffsetString32_t name;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_COMMENT`.
- `BrigStringOffset32_t name` — Byte offset to the place in the `hsa_data` section where the text of the comment (including the `//`) appears.

18.5.1.4 BrigDirectiveControl

`BrigDirectiveControl` specifies assorted finalizer controls, such as the maximum number of work-items in a work-group. For information on placement and scope of control directives, see [13.4 Control Directives for Low-Level Performance Tuning \(p. 295\)](#).

Syntax is:

```
struct BrigDirectiveControl {
    BrigBase base;
    BrigControlDirective16_t control;
    uint16_t reserved;
    BrigDataOffsetOperandList32_t operands;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_CONTROL`.
- `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 [18.3.8 BrigControlDirective \(p. 322\)](#).
- `uint16_t reserved` — Must be 0.
- `BrigDataOffsetOperandList32_t operands` — Byte offset to the place in the `hsa_data` section where a variable-sized array of byte offsets to operands in the `hsa_operand` section. The operands must either be `BRIG_KIND_OPERAND_DATA` or `BRIG_KIND_OPERAND_WAVESIZE`.

18.5.1.5 BrigDirectiveExecutable

`BrigDirectiveExecutable` describes a kernel, function or signature.

Kernels are arranged in the `hsa_code` section as (see [4.3.2 Kernel \(p. 41\)](#)):

1. `BrigDirectiveExecutable` with kind of `BRIG_KIND_DIRECTIVE_KERNEL`
2. Zero or more kernel formal arguments
3. Zero or more kernel code block entries that are scoped to the kernel
4. The next module scope entry

Functions are arranged in the `hsa_code` section as (see [4.3.3 Function \(p. 43\)](#) and [10.3 Function Declarations, Function Definitions, and Function Signatures \(p. 257\)](#)):

1. `BrigDirectiveExecutable` with kind of `BRIG_KIND_DIRECTIVE_FUNCTION`
2. Zero or more function output formal arguments (currently HSAIL only supports at most one output formal argument)
3. Zero or more function input formal arguments
4. Zero or more function code block entries that are scoped to the function
5. The next module scope entry

Signatures are arranged in the `hsa_code` section as (see [10.3.3 Function Signature \(p. 258\)](#)):

1. `BrigDirectiveExecutable` with kind of `BRIG_KIND_DIRECTIVE_SIGNATURE`
2. Zero or more destination parameters
3. Zero or more source parameters
4. The next top-level item

The formal arguments are `BrigDirectiveVariable` with a `segment` field of: `BRIG_SEGMENT_KERNARG` for kernels; and `BRIG_SEGMENT_ARG` for functions and signatures. For signatures the `name` field can be 0 if no formal argument name is specified.

Syntax is:

```
struct BrigDirectiveExecutable {
    BrigBase base;
    BrigDataOffsetString32_t name;
    uint16_t outArgCount;
    uint16_t inArgCount;
    BrigCodeOffset32_t firstInArg;
    BrigCodeOffset32_t firstCodeBlockEntry;
    BrigCodeOffset32_t nextModuleEntry;
    uint32_t codeBlockEntryCount;
    BrigExecutableModifier8_t modifier;
    BrigLinkage8_t linkage;
    uint16_t reserved;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_KERNEL`, `BRIG_KIND_DIRECTIVE_FUNCTION` or `BRIG_KIND_DIRECTIVE_SIGNATURE`.
- `BrigDataOffsetString32_t name` — Byte offset to the place in the `hsa_data` section giving the name of the kernel, function or signature.
- `uint16_t outArgCount` — The number of output parameters from the function or signature. Must be 0 for kernels.
- `uint16_t inArgCount` — The number of input formal arguments to the kernel, function or signature.

- `BrigCodeOffset32_t firstInArg` — Byte offset to the location in the `hsa_code` section of the first input formal argument. If there are no input formal arguments, then this must be the same value as `firstCodeBlockEntry`.
- `BrigCodeOffset32_t firstCodeBlockEntry` — Byte offset to the location in the `hsa_code` section of the first entry inside the code block of this kernel or function. If this is a signature, kernel declaration or function declaration (indicated by `modifier` with a `BRIG_EXECUTABLE_DEFINITION` of zero), or if the kernel or function definition code block has no entries, then this must be the same value as `nextModuleEntry`.
- `BrigCodeOffset32_t nextModuleEntry` — Byte offset to the location in the `hsa_code` section of the next module scope entry outside this kernel, function or signature. If there are no more module entries, then this must be the size of the `hsa_code` section.
- `uint32_t codeBlockEntryCount` — The number of entries (not bytes) in this kernel or function code block. If a signature, kernel declaration or function declaration then this must be 0.
- `BrigExecutableModifier8_t modifier` — Modifier for the kernel, function or signature. The `BRIG_EXECUTABLE_DEFINITION` must be 1 for signatures because they are always definitions; 0 if the kernel or function is a declaration; and 1 if the kernel or function is a definition. See [18.3.9 BrigExecutableModifierMask \(p. 323\)](#).
- `BrigLinkage8_t linkage` — Values are specified by the `BrigLinkage` enumeration. Must be `BRIG_LINKAGE_NONE` for signatures; and `BRIG_LINKAGE_PROGRAM` or `BRIG_LINKAGE_MODULE` for kernels or functions depending on the linkage specified. See [18.3.15 BrigLinkage \(p. 326\)](#).
- `uint16_t reserved` — Must be 0.

18.5.1.6 BrigDirectiveExtension

`BrigDirectiveExtension` is used to enable a device-specific extension. For more information, see [13.1 extension Directive \(p. 291\)](#).

Syntax is:

```
struct BrigDirectiveExtension {
    BrigBase base;
    BrigDataOffsetString32_t name;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_EXTENSION`.
- `BrigDataOffsetString32_t name` — Byte offset to the place in the `hsa_data` section where the name of the extension appears.

18.5.1.7 BrigDirectiveFbarrier

`BrigDirectiveFbarrier` is used for fbarrier definitions.

Syntax is:

```
struct BrigDirectiveFbarrier {
    BrigBase base;
    BrigDataOffsetString32_t name;
    BrigExecutableModifier8_t modifier;
    BrigLinkage8_t linkage;
    uint16_t reserved;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_FBARRIER`.
- `BrigDataOffsetString32_t name` — Byte offset to the place in the `hsa_data` section where the name of the fbarrier appears.
- `BrigExecutableModifier8_t modifier` — Modifier for the fbarrier. The `BRIG_EXECUTABLE_DEFINITION` must be 0 if a declaration; and 1 if a definition. See [18.3.9 BrigExecutableModifierMask \(p. 323\)](#).
- `BrigLinkage8_t linkage` — Values are specified by the `BrigLinkage` enumeration. For module scope fbarriers must be `BRIG_LINKAGE_PROGRAM` or `BRIG_LINKAGE_MODULE` depending on the linkage specified; and for function scope fbarriers must be `BRIG_LINKAGE_FUNCTION`. See [4.6.2 Scope \(p. 63\)](#) and [18.3.15 BrigLinkage \(p. 326\)](#).
- `uint16_t reserved` — Must be 0.

18.5.1.8 BrigDirectiveLabel

`BrigDirectiveLabel` declares a label. Label directives cannot be at the module level, they must be inside the code block of a function or a kernel.

Syntax is:

```
struct BrigDirectiveLabel {
    BrigBase base;
    BrigDataOffsetString32_t name;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_LABEL`.
- `BrigDataOffsetString32_t name` — Byte offset to the place in the `hsa_data` section table where the name of the label appears.

18.5.1.9 BrigDirectiveLoc

`BrigDirectiveLoc` specifies the source-level line position. The entries starting at next entry until the next `BrigDirectiveLoc` are assumed to correspond to the source location defined by this directive. This is similar to the `.line` `cpp` directive. For more information, see [13.2 loc Directive \(p. 293\)](#).

Syntax is:

```
struct BrigDirectiveLoc {
    BrigBase base;
    BrigDataOffsetString32_t filename;
    uint32_t line;
    uint32_t column;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_LOC`.
- `BrigDataOffsetString32_t filename` — Byte offset to the place in the `hsa_data` section where the name of the file appears. If the `HSAIL loc` directive did not specify a file name then must reference the same string used in the nearest preceding `loc` directive within the module that does specify a file name, or the empty string if there is no such `loc` directive.
- `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`.

18.5.1.10 BrigDirectiveNone

The `BrigDirectiveNone` 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 `BrigDirectiveNone` format.

`BrigDirectiveNone` can be as small 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 `BrigDirectiveNone` structure must be set to 0.

Syntax is:

```
struct BrigDirectiveNone {
    BrigBase base;
};
```

Fields are:

- `BrigInstKinds16_t kind` — `base.kind` must be `BRIG_KIND_NONE` (which has the value 0). `base.size` must be a multiple of 4. If `size` is greater than the size of the `BrigDirectiveNone` structure (4 bytes), then any extra bytes must be set to 0.

18.5.1.11 BrigDirectivePragma

`BrigDirectivePragma` allows additional information to be given to control the finalizer and other consumers of HSAIL. For more information, see [13.3 pragma Directive \(p. 294\)](#).

Syntax is:

```
struct BrigDirectivePragma {
    BrigBase base;
    BrigDataOffsetOperandList32_t operands;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_PRAGMA`.
- `BrigDataOffsetOperandList32_t operands` — Byte offset to the place in the `hsa_data` section where a variable-sized array of byte offsets to operands in the `hsa_operand` section. The `byteCount` of the array must be exactly $(4 * \text{number of operands})$. The operands must either be `BRIG_KIND_OPERAND_DATA`, `BRIG_KIND_OPERAND_CODE_REF` or `BRIG_KIND_OPERAND_STRING`.

18.5.1.12 BrigDirectiveVariable

BrigDirectiveVariable is used for variable declarations or definitions.

Syntax is:

```
struct BrigDirectiveVariable {  
    BrigBase base;  
    BrigDataOffsetString32_t name;  
    BrigOperandOffset32_t init;  
    BrigType16_t type;  
    BrigSegment8_t segment;  
    BrigAlignment8_t align;  
    BrigUInt64 dim;  
    BrigVariableModifier8_t modifier;  
    BrigLinkage8_t linkage;  
    BrigAllocation8_t allocation;  
    uint8_t reserved;  
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_VARIABLE`.
- `BrigDataOffsetString32_t name` — Byte offset into the place in the `hsa_data` section where the variable name appears.

- `BrigOperandOffset32_t init` — An initializer: only allowed for variable definitions in the global or readonly segment. Must be 0 if there is no initializer. Otherwise, must be the offset in the `hsa_operand` section to a operand with a `kind` field that depends on the `type` field:
 - If the `type` field is `BRIG_TYPE_ROIMG`, `BRIG_TYPE_WOIMG` or `BRIG_TYPE_RWIMG`:
 - If a non-array initializer, must be `BRIG_KIND_OPERAND_IMAGE_PROPERTIES`.
 - If an array initializer, must be `BRIG_KIND_OPERAND_OPERAND_LIST` with each element of the list of `kind` `BRIG_KIND_OPERAND_IMAGE_PROPERTIES`. If the list has fewer elements than `dim`, then the extra image handles are undefined.
 - If the `type` field is `BRIG_TYPE_SAMP`:
 - If a non-array initializer, must be `BRIG_KIND_OPERAND_SAMPLER_PROPERTIES`.
 - If an array initializer, must be `BRIG_KIND_OPERAND_OPERAND_LIST` with each element of the list of `kind` `BRIG_KIND_OPERAND_SAMPLER_PROPERTIES`. If the list has fewer elements than `dim`, then the extra sampler handles are undefined.
 - If the `type` field is `BRIG_TYPE_SIG32` or `BRIG_TYPE_SIG64` then must be `BRIG_KIND_OPERAND_DATA` whether a non-array or array initializer. The byte size of the data section entry must be exactly 8 times the number of entries in the initializer, and must only have 0 bytes for the null signal handle. If the byte size of the data section entry is less than the variable byte size, then the extra bytes are initialized to 0.
 - Otherwise must be `BRIG_KIND_OPERAND_DATA` whether a non-array or array initializer. The data section entry contains the bytes of the initializer. If the data section entry byte size is less than the variable byte size, then the extra bytes are initialized to 0.
- `BrigType_16_t type` — The BRIG type of the symbol.
- `BrigSegment8_t segment` — Segment that will hold the symbol. A member of the `BrigSegment` enumeration. See [18.3.32 BrigSegment \(p. 333\)](#).
- `BrigAlignment8_t align` — The required symbol alignment in bytes. If the directive does not specify the align type qualifier, then must be set to the value that corresponds to the natural alignment for `type`. See [18.3.2 BrigAlignment \(p. 320\)](#).

- `BrigUInt64 dim` — The array dimension size `dim`. See [18.3.34 BrigUInt64 \(p. 336\)](#).

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 [18.3.35 BrigVariableModifierMask \(p. 336\)](#).

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` must match the number of elements in the initializer.

- `BrigVariableModifier8_t modifier` — Modifier for the variable. See [18.3.35 BrigVariableModifierMask \(p. 336\)](#).
- `BrigLinkage8_t linkage` — Values are specified by the `BrigLinkage` enumeration. For module scope variables must be `BRIG_LINKAGE_PROGRAM` or `BRIG_LINKAGE_MODULE` depending on the linkage specified; for function scope variables must be `BRIG_LINKAGE_FUNCTION`; for argument scope variables must be `BRIG_LINKAGE_ARG`; and for signature scope variables must be `BRIG_LINKAGE_NONE`. See [4.6.2 Scope \(p. 63\)](#) and [18.3.15 BrigLinkage \(p. 326\)](#).
- `BrigAllocation8_t allocation` — Values are specified by the `BrigAllocation` enumeration. For global segment variables must be `BRIG_ALLOCATION_PROGRAM` or `BRIG_ALLOCATION_AGENT` depending on the allocation specified; for readonly segment variables must be `BRIG_ALLOCATION_AGENT`; for kernel declaration, function declaration and signature definition formal arguments must be `BRIG_ALLOCATION_NONE`; otherwise must be `BRIG_ALLOCATION_AUTOMATIC`. See [18.3.3 BrigAllocation \(p. 320\)](#).
- `uint8_t reserved` — Must be 0.

18.5.1.13 BrigDirectiveVersion

`BrigDirectiveVersion` specifies the HSAIL virtual ISA specification version, BRIG object format version, and target information. For more information, see [Chapter 14 version Statement \(p. 303\)](#).

There must be exactly one `BrigDirectiveVersion` directive in the `hsa_code` section. It may be optionally preceded only by `BrigDirectiveComment`, `BrigDirectiveLoc` and `BrigDirectivePragma` directives.

Syntax is:

```
struct BrigDirectiveVersion {
    BrigBase base;
    BrigVersion32_t hsailMajor;
    BrigVersion32_t hsailMinor;
    BrigVersion32_t brigMajor;
    BrigVersion32_t brigMinor;
    BrigProfile8_t profile;
    BrigMachineModel8_t machineModel;
    uint16_t reserved;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_VERSION`.
- `BrigVersion32_t hsailMajor` — The HSAIL virtual ISA major version. When generating BRIG, must be `BRIG_VERSION_HSAIL_MAJOR`. When consuming BRIG, must be `BRIG_VERSION_HSAIL_MAJOR` to be compatible with this revision of the HSAIL virtual ISA specification. See [18.3.36 BrigVersion \(p. 336\)](#).
- `BrigVersion32_t hsailMinor` — The HSAIL virtual ISA minor version. When generating BRIG, must be `BRIG_VERSION_HSAIL_MINOR`. When consuming BRIG, `hsailMajor` must be `BRIG_VERSION_HSAIL_MAJOR` and `hsailMinor` must be less than or equal to `BRIG_VERSION_HSAIL_MINOR` to be compatible with this revision of the HSAIL virtual ISA specification. See [18.3.36 BrigVersion \(p. 336\)](#).
- `BrigVersion32_t brigMajor` — The BRIG object format major version. When generating BRIG, must be `BRIG_VERSION_BRIG_MAJOR`. When consuming BRIG, must be `BRIG_VERSION_BRIG_MAJOR` to be compatible with this revision of the BRIG object format specification. See [18.3.36 BrigVersion \(p. 336\)](#).
- `BrigVersion32_t brigMinor` — The BRIG object format minor version. When generating BRIG, must be `BRIG_VERSION_BRIG_MINOR`. When consuming BRIG, `brigMajor` must be `BRIG_VERSION_BRIG_MAJOR` and `brigMinor` must be less than or equal to `BRIG_VERSION_BRIG_MINOR` to be compatible with this revision of the BRIG object format specification. See [18.3.36 BrigVersion \(p. 336\)](#).
- `BrigProfile8_t profile` — The profile. A member of the `BrigProfile` enumeration. See [18.3.22 BrigProfile \(p. 330\)](#)
- `BrigMachineModel8_t machineModel` — The machine model. A member of the `BrigMachineModel` enumeration. See [18.3.16 BrigMachineModel \(p. 326\)](#).
- `uint16_t reserved;` — Must be 0.

18.5.2 Instruction Entries

BRIG instructions corresponding to HSAIL operations. They can only appear in the code block of kernels and functions. The finalizer uses these to generate executable ISA code for kernels and indirect functions.

Every `BrigInst*` must start with a `BrigInstBase`. See [18.5.2.1 BrigInstBase \(p. 350\)](#).

The table below shows the possible formats for the instructions. Every instruction uses one of these formats.

Table 20–2 Formats of Instructions in the `hsa_code` Section

Name	Description
<code>BrigInstBase</code>	Every other <code>BrigInst*</code> entry must start with this structure. See 18.5.2.1 BrigInstBase (p. 350) .
<code>BrigInstAddr</code>	Address operations. See 18.5.2.2 BrigInstAddr (p. 350) .
<code>BrigInstAtomic</code>	Atomic operations. See 18.5.2.3 BrigInstAtomic (p. 351) .
<code>BrigInstBasic</code>	Used for all operations that require no extra modifier information. See 18.5.2.4 BrigInstBasic (p. 352) .
<code>BrigInstBr</code>	Branch, call, barrier and fbarrier operations. See 18.5.2.5 BrigInstBr (p. 352) .
<code>BrigInstCmp</code>	Compare operation. See 18.5.2.6 BrigInstCmp (p. 353) .
<code>BrigInstCvt</code>	Conversion operation. See 18.5.2.7 BrigInstCvt (p. 353) .

Name	Description
BrigInstImage	Image-related operations. See 18.5.2.8 BrigInstImage (p. 353) .
BrigInstLane	Cross lane operations. See 18.5.2.9 BrigInstLane (p. 354) .
BrigInstMem	Load and store memory operations. See 18.5.2.10 BrigInstMem (p. 354) .
BrigInstMemFence	Memory fence operation. See 18.5.2.11 BrigInstMemFence (p. 355) .
BrigInstMod	Operations with a single modifier, such as a rounding mode. See 18.5.2.12 BrigInstMod (p. 356) .
BrigInstQueryImage	Image query operations. See 18.5.2.13 BrigInstQueryImage (p. 357) .
BrigInstQuerySampler	Sampler query operation. See 18.5.2.14 BrigInstQuerySampler (p. 357) .
BrigInstQueue	User Mode Queue operations. See 18.5.2.15 BrigInstQueue (p. 357) .
BrigInstSeg	Operations with memory segments. See 18.5.2.16 BrigInstSeg (p. 358) .
BrigInstSegCvt	Operations which convert between segment and flat addresses. See 18.5.2.17 BrigInstSegCvt (p. 358) .
BrigInstSignal	Signal operations. See 18.5.2.18 BrigInstSignal (p. 359) .
BrigInstSourceType	Operations that have different types for their destination and source operands. See 18.5.2.19 BrigInstSourceType (p. 359) .

18.5.2.1 BrigInstBase

Every other `BrigInst*` must start with `BrigInstBase` which in turn starts with `BrigBase`.

Syntax is:

```
struct BrigInstBase {
    BrigBase base;
    BrigOpcode16_t opcode;
    BrigType16_t type;
    BrigDataOffsetOperandList32_t operands;
};
```

Fields are:

- `BrigBase base` — The `base.kind` field must be in the right-open interval `[BRIG_KIND_INST_BEGIN, BRIG_KIND_INST_END]`. See [18.3.14 BrigKind \(p. 325\)](#).
- `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 source operand types (for example, a `sourceType` field), this can also be the type of the source operands. If an operation does not have any typed operands (for example, `call`, `ret` and `br`), then the value `BRIG_TYPE_NONE` must be used.
- `BrigDataOffsetOperandList32_t operands` — Byte offset to the place in the `hsa_data` section where a variable-sized array of byte offsets to operands in the `hsa_operand` section. The `byteCount` of the array must be exactly `(4 * number of operands)`. Any destination operand is first, followed by any source operands.

18.5.2.2 BrigInstAddr

The `BrigInstAddr` format is used for address operations.

Syntax is:

```
struct BrigInstAddr {
    BrigInstBase base;
    BrigSegment8_t segment;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_ADDR`. `base.type` must be the data type of the destination and source of the operation.
- `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 [18.3.32 BrigSegment \(p. 333\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.3 BrigInstAtomic

The `BrigInstAtomic` format is used for atomic and atomic no return operations.

Syntax is:

```
struct BrigInstAtomic {
    BrigInstBase base;
    BrigSegment8_t segment;
    BrigMemoryOrder8_t memoryOrder;
    BrigMemoryScope8_t memoryScope;
    BrigAtomicOperation8_t atomicOperation;
    uint8_t equivClass;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_ATOMIC`. `base.opcode` must be `BRIG_OPCODE_ATOMIC` or `BRIG_OPCODE_ATOMICNORET`. `base.type` must be the data type of the destination and source of the atomic operation.
- `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`. Otherwise must be `BRIG_SEGMENT_GLOBAL` or `BRIG_SEGMENT_GROUP`. See [18.3.32 BrigSegment \(p. 333\)](#).
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the atomic operation. See [18.3.18 BrigMemoryOrder \(p. 326\)](#).
- `BrigMemoryScope8_t memoryScope` — Memory scope of the atomic operation. If segment is `BRIG_SEGMENT_GLOBAL` or `BRIG_SEGMENT_FLAT` then must be `BRIG_MEMORY_SCOPE_WAVEFRONT`, `BRIG_MEMORY_SCOPE_WORKGROUP`, `BRIG_MEMORY_SCOPE_COMPONENT` or `BRIG_MEMORY_SCOPE_SYSTEM`. If segment is `BRIG_SEGMENT_GROUP` then must be `BRIG_MEMORY_SCOPE_WAVEFRONT` or `BRIG_MEMORY_SCOPE_WORKGROUP`. See [18.3.19 BrigMemoryScope \(p. 326\)](#).

- `BrigAtomicOperation8_t atomicOperation` — The atomic operation such as `add` or `or`. The wait atomic operations are not allowed. See [18.3.5 BrigAtomicOperation \(p. 321\)](#).
- `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 \(p. 160\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.4 BrigInstBasic

The `BrigInstBasic` format is used for all operations that require no extra modifier information.

Syntax is:

```
struct BrigInstBasic {
    BrigInstBase base;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_BASIC`. `base.type` must be the data type of the destination and source of the operation.

18.5.2.5 BrigInstBr

The `BrigInstBr` format is used for the branch, call, barrier and fbarrier operations.

Syntax is:

```
struct BrigInstBr {
    BrigInstBase base;
    BrigWidth8_t width;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_BR`. `base.opcode` must be `BRIG_OPCODE_BR`, `BRIG_OPCODE_CBR`, `BRIG_OPCODE_SBR`, `BRIG_OPCODE_CALL`, `BRIG_OPCODE_SCALL`, `BRIG_OPCODE_ICALL`, `BRIG_OPCODE_BARRIER`, `BRIG_OPCODE_WAVEBARRIER`, `BRIG_OPCODE_ARRIVEFBAR`, `BRIG_OPCODE_JOINFBAR`, `BRIG_OPCODE_LEAVEFBAR` or `BRIG_OPCODE_WAITFBAR`. `base.type` must be the source operand data type, or `BRIG_TYPE_NONE` if the operation has no typed operands.
- `BrigWidth8_t width` — The width modifier. If the operation does not support the width modifier, then this must be `BRIG_WIDTH_ALL` for the direct branch and direct call operations, and `BRIG_WIDTH_WAVESIZE` for the wavebarrier operation. If the operation supports the width modifier but does not specify it, then this must be the default value defined by the operation: for indirect branch and indirect call operations it is `BRIG_WIDTH_1`; for barrier operations it is `BRIG_WIDTH_ALL`; and for the fbarrier operations it is `BRIG_WIDTH_WAVESIZE`. Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.37 BrigWidth \(p. 337\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.6 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 {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigAluModifier16_t modifier;
    BrigCompareOperation8_t compare;
    BrigPack8_t pack;
    uint16_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_CMP`. `base.opcode` must be `BRIG_OPCODE_CMP`. `base.type` must be the data type of the destination of the compare operation: for packed compares, must be `u` with the same length as `sourceType`.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. See [18.3.4 BrigAluModifierMask \(p. 320\)](#).
- `BrigCompareOperation8_t compare` — The specific comparison (greater than, less than, and so forth).
- `BrigPack8_t pack` — Packing control. See [18.3.21 BrigPack \(p. 330\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.7 BrigInstCvt

The `BrigInstCvt` format is used for conversion operations.

Syntax is:

```
struct BrigInstCvt {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigAluModifier16_t modifier;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_CVT`. `base.opcode` must be `BRIG_OPCODE_CVT`. `base.type` must be the data type of the destination of the conversion operation.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier16_t modifier` — The modifier flags for this operation. See [18.3.4 BrigAluModifierMask \(p. 320\)](#).

18.5.2.8 BrigInstImage

The `BrigInstImage` format is used for the image operations.

Syntax is:

```
struct BrigInstImage {
    BrigInstBase base;
    BrigType16_t imageType;
    BrigType16_t coordType;
    BrigImageGeometry8_t geometry;
    uint8_t equivClass;
    uint16_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_IMAGE`. `base.type` must be the data type of the destination of the image operation.
- `BrigType16_t imageType` — Type of the image. Must be `BRIG_KIND_INST_ROIMG`, `BRIG_KIND_INST_WOIMG` or `BRIG_KIND_INST_RWIMG`.
- `BrigType16_t coordType` — Type of the coordinates.
- `BrigImageGeometry8_t geometry` — Image geometry. See [18.3.12 BrigImageGeometry \(p. 324\)](#).
- `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 \(p. 160\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.9 BrigInstLane

The `BrigInstLane` format is used for cross-lane operations.

Syntax is:

```
struct BrigInstLane {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigWidth8_t width;
    uint8_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_LANE`. `base.type` must be the data type of the destination of the cross-lane operation.
- `BrigType16_t sourceType` — Type of the source. If the operation does not have a source type modifier then must be `BRIG_TYPE_NONE`.
- `BrigWidth8_t width` — The width modifier. If the operation does not specify the width modifier, then this must be `BRIG_WIDTH_1` (the default for the cross lane operations). Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.37 BrigWidth \(p. 337\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.10 BrigInstMem

The `BrigInstMem` format is used for memory operations.

Syntax is:

```
struct BrigInstMem {
    BrigInstBase base;
    BrigSegment8_t segment;
    BrigAlignment8_t align;
    uint8_t equivClass;
    BrigWidth8_t width;
    BrigMemoryModifier8_t modifier;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_MEM`. `base.type` must be the data type of the destination and source of the memory operation.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the operation does not support the segment modifier, then this must be `BRIG_SEGMENT_NONE`. If the operation supports the segment modified but does not specify it, then this must be `BRIG_SEGMENT_FLAT`. See [18.3.32 BrigSegment \(p. 333\)](#).
- `BrigAlignment8_t align` — The align modifier. If the operation does not specify the align modifier, then this must be `BRIG_ALIGNMENT_1` (the default for memory operations). Otherwise, this must be the value from `BrigAlignment` that corresponds to the specified align modifier. See [18.3.2 BrigAlignment \(p. 320\)](#).
- `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 \(p. 160\)](#).
- `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 memory operations). Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.37 BrigWidth \(p. 337\)](#).
- `BrigMemoryModifier8_t modifier` — Memory modifier flags of the operation. See [18.3.17 BrigMemoryModifierMask \(p. 326\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.11 BrigInstMemFence

The `BrigInstMemFence` format is used for the `memfence` operation.

Syntax is:

```
struct BrigInstMemFence {
    BrigInstBase base;
    BrigMemoryOrder8_t memoryOrder;
    BrigMemoryScope8_t globalSegmentMemoryScope;
    BrigMemoryScope8_t groupSegmentMemoryScope;
    BrigMemoryScope8_t imageSegmentMemoryScope;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_MEMFENCE`. `base.opcode` must be `BRIG_OPCODE_MEMFENCE`. `base.type` must be `BRIG_TYPE_NONE` as the `memfence` operation has no destination operand.
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the memory fence operation. Must be `BRIG_MEMORY_ORDER_ACQUIRE_RELEASE` if `imageSegmentMemoryScope` is not `BRIG_MEMORY_SCOPE_NONE`. Otherwise must be `BRIG_MEMORY_ORDER_ACQUIRE`, `BRIG_MEMORY_ORDER_RELEASE` or `BRIG_MEMORY_ORDER_ACQUIRE_RELEASE`. See [18.3.18 BrigMemoryOrder \(p. 326\)](#).
- `BrigMemoryScope8_t globalSegmentMemoryScope` — Memory scope for the global segment of the memory fence operation. If the global segment was not specified then must be `BRIG_MEMORY_SCOPE_NONE`. Otherwise must be `BRIG_MEMORY_SCOPE_WAVEFRONT`, `BRIG_MEMORY_SCOPE_WORKGROUP`, `BRIG_MEMORY_SCOPE_COMPONENT` or `BRIG_MEMORY_SCOPE_SYSTEM`. See [18.3.19 BrigMemoryScope \(p. 326\)](#).
- `BrigMemoryScope8_t groupSegmentMemoryScope` — Memory scope for the group segment of the memory fence operation. If the group segment was not specified then must be `BRIG_MEMORY_SCOPE_NONE`. Otherwise must be `BRIG_MEMORY_SCOPE_WAVEFRONT` or `BRIG_MEMORY_SCOPE_WORKGROUP`. See [18.3.19 BrigMemoryScope \(p. 326\)](#).
- `BrigMemoryScope8_t imageSegmentMemoryScope` — Memory scope for the image segment of the memory fence operation. The image segment is not one of the regular segments, but is implicitly used by the image operations to access image data. If the image segment was not specified then must be `BRIG_MEMORY_SCOPE_NONE`. Otherwise must be `BRIG_MEMORY_SCOPE_WORKITEM`, `BRIG_MEMORY_SCOPE_WAVEFRONT` or `BRIG_MEMORY_SCOPE_WORKGROUP`. See [18.3.19 BrigMemoryScope \(p. 326\)](#).

18.5.2.12 BrigInstMod

The `BrigInstMod` format is used for ALU operations with a modifier.

Syntax is:

```
struct BrigInstMod {
    BrigInstBase base;
    BrigAluModifier16_t modifier;
    BrigPack8_t pack;
    uint8_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_MOD`. `base.type` must be the data type of the destination of the operation.
- `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 [18.3.4 BrigAluModifierMask \(p. 320\)](#).
- `BrigPack8_t pack` — Packing control. If the operation does not have a packing modifier, this must be set to `BRIG_PACK_NONE`. See [18.3.21 BrigPack \(p. 330\)](#).
- `uint8_t reserved` — Must be 0.

18.5.2.13 BrigInstQueryImage

The `BrigInstQueryImage` format is used for the `queryimage` operation.

Syntax is:

```
struct BrigInstQueryImage {
    BrigInstBase base;
    BrigType16_t imageType;
    BrigImageGeometry8_t geometry;
    BrigImageQuery8_t query;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_QUERY_IMAGE`. `base.type` must be the data type of the destination of the query operation.
- `BrigType16_t imageType` — Type of the image. Must be `BRIG_KIND_INST_ROIMG`, `BRIG_KIND_INST_WOIMG` or `BRIG_KIND_INST_RWIMG`.
- `BrigImageGeometry8_t geometry` — Image geometry. See [18.3.12 BrigImageGeometry \(p. 324\)](#).
- `BrigImageQuery8_t query` — Image property being queried. See [18.3.13 BrigImageQuery \(p. 324\)](#).

18.5.2.14 BrigInstQuerySampler

The `BrigInstQuerySampler` format is used for the `querysampler` operation.

Syntax is:

```
struct BrigInstQuerySampler {
    BrigInstBase base;
    BrigSamplerQuery8_t query;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_QUERY_SAMPLER`. `base.type` must be the data type of the destination of the sampler operation.
- `BrigSamplerQuery8_t query` — Sampler property being queried. See [18.3.28 BrigSamplerQuery \(p. 332\)](#).

18.5.2.15 BrigInstQueue

The `BrigInstQueue` format is used for User Mode Queue operations.

Syntax is:

```
struct BrigInstQueue {
    BrigInstBase base;
    BrigSegment8_t segment;
    BrigMemoryOrder8_t memoryOrder;
    uint16_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_QUEUE`.
`base.type` must be the data type of the destination of the queue operation.
- `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 [18.3.32 BrigSegment \(p. 333\)](#).
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the User Mode Queue operation. See [18.3.18 BrigMemoryOrder \(p. 326\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.16 BrigInstSeg

The `BrigInstSeg` format is used for operations with memory segments.

Syntax is:

```
struct BrigInstSeg {
    BrigInstBase base;
    BrigSegment8_t segment;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_SEG`.
`base.type` must be the data type of the destination of the operation.
- `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 [18.3.32 BrigSegment \(p. 333\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.17 BrigInstSegCvt

The `BrigInstSegCvt` format is used for operations which convert between segment and flat addresses.

Syntax is:

```
struct BrigInstSegCvt {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigSegment8_t segment;
    BrigSegCvtModifier8_t modifier;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_SEG_CVT`.
`base.type` must be the data type of the destination of the convert operation.
- `BrigType16_t sourceType` — Type of the source.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. See [18.3.32 BrigSegment \(p. 333\)](#).
- `BrigSegCvtModifier8_t modifier` — Segment conversion modifier flags of the operation. See [18.3.31 BrigSegCvtModifierMask \(p. 333\)](#).

18.5.2.18 BrigInstSignal

The `BrigInstSignal` format is used for signal operations.

Syntax is:

```
struct BrigInstSignal {
    BrigInstBase base;
    BrigType16_t signalType;
    BrigMemoryOrder8_t memoryOrder;
    BrigAtomicOperation8_t signalOperation;
};
```

Fields are:

- `uint16_t kind` — `base.base.kind` must be `BRIG_KIND_INST_SIGNAL`.
`base.opcode` must be `BRIG_OPCODE_SIGNAL` or `BRIG_OPCODE_SIGNALNORET`.
`base.type` must be the data type of the destination and source of the signal operation.
- `BrigType16_t signalType` — Type of the signal. Must be `BRIG_TYPE_SIG32` or `BRIG_TYPE_SIG64` and match the size of the `base.type` field.
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the signal operation. See [18.3.18 BrigMemoryOrder \(p. 326\)](#).
- `BrigAtomicOperation8_t signalOperation` — The signal operation such as `add` or `or`. See [18.3.5 BrigAtomicOperation \(p. 321\)](#).

18.5.2.19 BrigInstSourceType

The `BrigInstSourceType` format is used for operations that have different types for their destination and source operands.

Syntax is:

```
struct BrigInstSourceType {
    BrigInstBase base;
    BrigType16_t sourceType;
    uint16_t reserved;
};
```

Fields are:

- `BrigInstBase base` — `base.base.kind` must be `BRIG_KIND_INST_SOURCE_TYPE`. `base.type` must be the data type of the destination of the operation.
- `BrigType16_t sourceType` — Type of the source.
- `uint16_t reserved` — Must be 0.

18.6 hsa_operand Section

The `hsa_operand` section contains the operands of the directives and instructions of the module.

The `hsa_operand` section must start with a `BrigSectionHeader` entry. The name of the section must be `hsa_operand`. See [18.3.30 BrigSectionHeader \(p. 332\)](#).

All operand entries (`BrigOperand*`) in the `hsa_operand` section must start with a `BrigBase` structure. The `kind` field of the `BrigBase` structure must be in the right-open

interval [BRIG_KIND_OPERAND_BEGIN, BRIG_KIND_OPERAND_END). See [18.3.14 BrigKind \(p. 325\)](#).

To reduce the size of the `hsa_operand` section it is allowed, but not required, to reference an already created `BrigOperand*` entry, rather than create duplicate `BrigOperand*` entries.

The table below shows the possible formats for the operands. Every operand uses one of these formats.

Table 20–3 Formats of Operands in the `hsa_operand` Section

Name	Description
<code>BrigOperandAddress</code>	Used for address expressions. See 18.6.1 BrigOperandAddress (p. 360) .
<code>BrigOperandCodeList</code>	List of references to entries in the <code>hsa_code</code> section. See 18.6.2 BrigOperandCodeList (p. 361) .
<code>BrigOperandCodeRef</code>	A reference to an entry in the <code>hsa_code</code> section. See 18.6.3 BrigOperandCodeRef (p. 361) .
<code>BrigOperandData</code>	Declares a data value. See 18.6.4 BrigOperandData (p. 362) .
<code>BrigOperandImageProperties</code>	Declares the properties of the image referenced by an image handle. See 18.6.5 BrigOperandImageProperties (p. 362) .
<code>BrigOperandOperandList</code>	List of references to entries in the <code>hsa_operand</code> section. See 18.6.6 BrigOperandOperandList (p. 363) .
<code>BrigOperandReg</code>	A register (c, s, or d). See 18.6.7 BrigOperandReg (p. 364) .
<code>BrigOperandSamplerProperties</code>	Declares the properties of a sampler referenced by a sample handle. See 18.6.8 BrigOperandSamplerProperties (p. 364) .
<code>BrigOperandString</code>	A textual string. See 18.6.9 BrigOperandString (p. 365) .
<code>BrigOperandWavesize</code>	The wavesize operand. See 18.6.10 BrigOperandWavesize (p. 365) .

18.6.1 `BrigOperandAddress`

`BrigOperandAddress` is used for address expressions. See [4.18 Address Expressions \(p. 88\)](#).

Syntax is:

```
struct BrigOperandAddress {
    BrigBase base;
    BrigDirectiveOffset32_t symbol;
    BrigOperandOffset32_t reg;
    BrigUInt64 offset;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_ADDRESS`.
- `BrigDirectiveOffset32_t symbol` — Byte offset in `hsa_code` section pointing to the symbol definition or declaration for the name. See [18.5.1.1 Declarations and Definitions in the Same Module \(p. 340\)](#). If the address expression has no symbol name then must be 0.

- `BrigStringOffset32_t reg` — Byte offset in the `hsa_operand` section to a `BRIG_KIND_OPERAND_REGISTER` operand. If the address expression has no register then must be 0.
- `BrigUInt64 offset` — Byte *offset* to add to the address. See [18.3.34 BrigUInt64 \(p. 336\)](#).

If the address expression has no offset then *offset* must be 0.

18.6.2 BrigOperandCodeList

`BrigOperandCodeList` is used for a list of references to entries in the `hsa_code` section

Syntax is:

```
struct BrigOperandCodeList {
    BrigBase base;
    BrigDataOffsetCodeList32_t elements;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_CODE_LIST`.
- `BrigDataOffsetCodeList32_t elements` — Byte offset to the place in the `hsa_data` section where a variable-sized array of byte offsets to entries in the `hsa_code` section. The `byteCount` of the array must be exactly `(4 * number of elements)`.
 - When used as a function actual argument list, each element must reference `BRIG_KIND_DIRECTIVE_VARIABLE` with `BRIG_SEGMENT_ARG` segment.
 - When used as a function list, each element must reference a `BRIG_KIND_DIRECTIVE_FUNCTION` or `BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION` directive. See [18.5.1.1 Declarations and Definitions in the Same Module \(p. 340\)](#).
 - When used as a label list, each element must reference a `BRIG_KIND_DIRECTIVE_LABEL` directive in the same function scope.

18.6.3 BrigOperandCodeRef

`BrigOperandCodeRef` is used to reference an entry in the `hsa_code` section.

Syntax is:

```
struct BrigOperandCodeRef {
    BrigBase base;
    BrigCodeOffset32_t ref;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_CODE_REF`
- `BrigCodeOffset32_t ref` — Byte offset to the place in the `hsa_code` section.
 - When used to reference a kernel, must reference `BRIG_KIND_DIRECTIVE_KERNEL` directive.
 - When used to reference a function, must reference `BRIG_KIND_DIRECTIVE_FUNCTION` or `BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION` directive.
 - When used to reference a signature, must reference `BRIG_KIND_DIRECTIVE_SIGNATURE` directive.
 - When used to reference a variable, must reference `BRIG_KIND_DIRECTIVE_VARIABLE` directive.
 - When used to reference a fbarrier, must reference `BRIG_KIND_DIRECTIVE_FBARRIER` directive.
 - When used to reference a label, must reference `BRIG_KIND_DIRECTIVE_LABEL` directive in the same function scope.

See [18.5.1.1 Declarations and Definitions in the Same Module \(p. 340\)](#).

18.6.4 BrigOperandData

`BrigOperandData` specifies the data value.

Syntax is:

```
struct BrigOperandData {
    BrigBase base;
    BrigDataOffset32_t data;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_DATA`.
- `BrigDataOffset32_t data` — Byte offset into the place in the `hsa_data` section where the data value is available. The data size is the `byteCount` of the `BrigData`.

18.6.5 BrigOperandImageProperties

`BrigOperandImageProperties` specifies the properties of an image referenced by an image handle. For more information, see [7.1.7 Image Creation and Image Handles \(p. 214\)](#).

Syntax is:

```
struct BrigOperandImageProperties {
    BrigBase base;
    uint64_t width;
    uint64_t height;
    uint64_t depth;
    uint64_t array;
    BrigImageGeometry8_t geometry,
    BrigImageChannelOrder8_t channelOrder;
    BrigImageChannelType8_t channelType;
    uint8_t reserved;
};
```

Fields are:

- `BrigDirectiveKinds16_t kind` — `base.kind` must be `BRIG_KIND_OPERAND_IMAGE_PROPERTIES`.
- `uint64_t width` — The image width. Must be greater than zero for all image geometries.
- `uint64_t height` — The image height. Must be greater than zero if `geometry` is `BRIG_GEOMETRY_2D`, `BRIG_GEOMETRY_3D`, `BRIG_GEOMETRY_2DA`, `BRIG_GEOMETRY_2DDEPTH` or `BRIG_GEOMETRY_2DADEPTH`; otherwise must be 0.
- `uint64_t depth` — The image depth. Must be greater than zero if `geometry` is `BRIG_GEOMETRY_3D`; otherwise must be 0.
- `uint64_t array` — The number of images in the array. Must be greater than zero if `geometry` is `BRIG_GEOMETRY_1DA`, `BRIG_GEOMETRY_2DA` or `BRIG_GEOMETRY_2DADEPTH`; otherwise must be 0.
- `BrigImageGeometry8_t geometry` — Geometry for the image. A member of the `BrigImageGeometry` enumeration. See [18.3.12 BrigImageGeometry \(p. 324\)](#).
- `BrigImageChannelOrder8_t channelOrder` — Channel order for the components. Components of an image can be reordered when values are read from or written to memory. A member of the `BrigImageChannelOrder` enumeration. See [18.3.10 BrigImageChannelOrder \(p. 323\)](#).
- `BrigImageChannelType8_t channelType` — Channel type for storing images. Images can be stored and accessed in assorted formats. A member of the `BrigImageChannelType` enumeration. See [18.3.11 BrigImageChannelType \(p. 324\)](#).
- `uint8_t reserved` — Must be 0.

18.6.6 BrigOperandOperandList

`BrigOperandOperandList` is used for a list of references to entries in the `hsa_operand` section

Syntax is:

```
struct BrigOperandOperandList {
    BrigBase base;
    BrigDataOffsetOperandList32_t elements;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_OPERAND_LIST`.
- `BrigDataOffsetCodeList32_t elements` — Byte offset to the place in the `hsa_data` section where a variable-sized array of byte offsets to entries in the `hsa_operand` section. The `byteCount` of the array must be exactly `(4 * number of elements)`.
 - When used as a destination vector operand, each element must reference a `BRIG_KIND_OPERAND_REG` directive.
 - When used as a source vector operand, each element must reference a `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA` or `BRIG_KIND_OPERAND_WAVESIZE` directive.
 - When used as an image array initializer, each element must reference a `BRIG_KIND_OPERAND_IMAGE_PROPERTIES` directive.
 - When used as a sampler array initializer, each element must reference a `BRIG_KIND_OPERAND_SAMPLER_PROPERTIES` directive.

18.6.7 BrigOperandReg

`BrigOperandReg` is used for a register (c, s, or d).

Syntax is:

```
struct BrigOperandReg {
    BrigBase base;
    BrigRegisterKind16_t regKind;
    uint16_t regNum;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_REG`.
- `BrigRegisterKind16_t regKind` — Must be `BRIG_REG_KIND_SINGLE` for s register, `BRIG_REG_KIND_DOUBLE` for d register, and `BRIG_REG_KIND_QUAD` for q register.
- `uint16_t regNum` — The register number.

18.6.8 BrigOperandSamplerProperties

`BrigOperandSamplerProperties` specifies the properties of a sampler referenced by a sampler handle. For more information, see [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#).

Syntax is:

```
struct BrigOperandSamplerProperties {
    BrigBase base;
    BrigSamplerCoordNormalization8_t coord;
    BrigSamplerFilter8_t filter;
    BrigSamplerAddressing8_t addressing;
    uint8_t reserved;
};
```

Fields are:

- `BrigDirectiveKinds16_t kind` — `base.kind` must be `BRIG_KIND_OPERAND_SAMPLER_PROPERTIES`.
- `BrigSamplerCoordNormalization8_t coord` — The coordinate normalization mode controls whether the coordinates are normalized or unnormalized. Does not apply to the array index coordinate of 1DA, 2DA and 2DADEPTH images which always use `BRIG_COORD_UNNORMALIZED`. Must be a member of the `BrigSamplerCoordNormalization` enumeration. See [18.3.26 BrigSamplerCoordNormalization \(p. 331\)](#).
- `BrigSamplerFilter8_t filter` — The filter mode used to specify how image elements are selected. Must be a member of the `BrigSamplerFilter` enumeration. If `coord` is `BRIG_COORD_UNNORMALIZED` then must be `BRIG_FILTER_NEAREST`. See [18.3.27 BrigSamplerFilter \(p. 332\)](#).
- `BrigSamplerAddressing8_t addressing` — The addressing mode used when coordinates are out of range of the corresponding image dimension size. Must be a member of the `BrigSamplerAddressing` enumeration. If `coord` is `BRIG_COORD_UNNORMALIZED` then must be `BRIG_ADDRESSING_UNDEFINED`, `BRIG_ADDRESSING_CLAMP_TO_EDGE` or `BRIG_ADDRESSING_CLAMP_TO_BORDER`. Does not apply to the array index coordinate of 1DA, 2DA and 2DADEPTH images which always use `BRIG_ADDRESSING_CLAMP_TO_EDGE`. See [18.3.25 BrigSamplerAddressing \(p. 331\)](#).

18.6.9 BrigOperandString

`BrigOperandString` is used for a textual string.

Syntax is:

```
struct BrigOperandReg {
    BrigBase base;
    BrigDataOffsetString32_t string;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_STRING`.
- `BrigDataOffsetString32_t string` — Byte offset to the place in the `hsa_data` section where the textual string occurs.

18.6.10 BrigOperandWavesize

`BrigOperandWavesize` is the `wavesize` operand, which is a compile-time value equal to the size of a wavefront.

Syntax is:

```
struct BrigOperandWavesize {
    BrigBase base;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_WAVESIZE`.

18.7 BRIG Syntax for Operations

This section describes the BRIG syntax for operations.

18.7.1 BRIG Syntax for Arithmetic Operations

Some operations support modifiers that have default values. These operations can either be encoded as `BRIG_KIND_INST_BASIC` if all modifiers have default values, or by `BRIG_KIND_INST_MOD` whether or not default modifiers are used. Using `BRIG_KIND_INST_BASIC` only serves to reduce the size of BRIG.

18.7.1.1 BRIG Syntax for Integer Arithmetic Operations

Table 20–4 BRIG Syntax for Integer Arithmetic Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
<code>BRIG_OPCODE_ABS</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	
<code>BRIG_OPCODE_ADD</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_BORROW</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_CARRY</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_DIV</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_MAX</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_MIN</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_MUL</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_MULHI</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_NEG</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	
<code>BRIG_OPCODE_Rem</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_SUB</code>	<code>BRIG_KIND_INST_BASIC</code> (if only default modifiers are used) or <code>BRIG_KIND_INST_MOD</code>	<i>dest</i>	<i>src</i>	<i>src</i>

dest: must be `BRIG_KIND_OPERAND_REG`.

src: must be `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA`, or `BRIG_KIND_OPERAND_WAVESIZE`.

18.7.1.2 BRIG Syntax for Integer Optimization Operation

Table 20–5 BRIG Syntax for Integer Optimization Operation

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
<code>BRIG_OPCODE_MAD</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.3 BRIG Syntax for 24-Bit Integer Optimization Operations

Table 20–6 BRIG Syntax for 24-Bit Integer Optimization Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_MAD24	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_MAD24HI	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_MUL24	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_MUL24HI	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.4 BRIG Syntax for Integer Shift Operations

Table 20–7 BRIG Syntax for Integer Optimization Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_SHL	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE SHR				

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.5 BRIG Syntax for Individual Bit Operations

Table 20–8 BRIG Syntax for Individual Bit Operations

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_AND	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_NOT	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	
BRIG_OPCODE_OR	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_POPCOUNT	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	
BRIG_OPCODE_XOR	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.6 BRIG Syntax for Bit String Operations

Table 20–9 BRIG Syntax for Bit String Operations

Opcode	Format	Oper. 0	Oper. 1	Oper. 2	Oper. 3	Oper. 4
<code>BRIG_OPCODE_BITEXTRACT</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
<code>BRIG_OPCODE_BITINSERT</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	<i>src</i>
<code>BRIG_OPCODE_BITMASK</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>		
<code>BRIG_OPCODE_BITREV</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>			
<code>BRIG_OPCODE_BITSELECT</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
<code>BRIG_OPCODE_FIRSTBIT</code>	<code>BRIG_KIND_INST_SOURCE_TYPE</code>	<i>dest</i>	<i>src</i>			
<code>BRIG_OPCODE_LASTBIT</code>	<code>BRIG_KIND_INST_SOURCE_TYPE</code>	<i>dest</i>	<i>src</i>			

dest: must be `BRIG_KIND_OPERAND_REG`.

src: must be `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA`, or `BRIG_KIND_OPERAND_WAVESIZE`.

18.7.1.7 BRIG Syntax for Copy (Move) Operations

Table 20–10 BRIG Syntax for Copy (Move) Operations

Opcode	Format	Operand 0	Operand 1
<code>BRIG_OPCODE_COMBINE</code>	<code>BRIG_KIND_INST_SOURCE_TYPE</code>	<i>dest</i>	<i>src-vector</i>
<code>BRIG_OPCODE_EXPAND</code>	<code>BRIG_KIND_INST_SOURCE_TYPE</code>	<i>dest-vector</i>	<i>src</i>
<code>BRIG_OPCODE_LDA</code>	<code>BRIG_KIND_INST_ADDR</code>	<i>dest</i>	<i>address</i>
<code>BRIG_OPCODE_MOV</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>dest</i>	<i>src</i>

dest: must be `BRIG_KIND_OPERAND_REG`.

src-vector: must be `BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA` or `BRIG_KIND_OPERAND_WAVESIZE` operands.

dest-vector: must be `BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REG` operands.

src: must be `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA`, or `BRIG_KIND_OPERAND_WAVESIZE`.

address: must be `BRIG_KIND_OPERAND_ADDRESS`.

18.7.1.8 BRIG Syntax for Packed Data Operations

Table 20–11 BRIG Syntax for Packed Data Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_SHUFFLE	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>number</i>
BRIG_OPCODE_UNPACKHI	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_UNPACKLO	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_PACK	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_UNPACK	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>	

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

number: must be BRIG_KIND_OPERAND_DATA.

18.7.1.9 BRIG Syntax for Bit Conditional Move (cmov) Operation

Table 20–12 BRIG Syntax for Bit Conditional Move (cmov) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_CMOV	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.10 BRIG Syntax for Floating-Point Arithmetic Operations

Table 20–13 BRIG Syntax for Floating-Point Arithmetic Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_ADD	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_CEIL	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_DIV	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_FLOOR	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_FMA	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_FRACT	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_MAX	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_MIN	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_MUL	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_RINT	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_SQRT	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_SUB	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_TRUNC	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>		

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG OR BRIG_KIND_OPERAND_DATA.

18.7.1.11 BRIG Syntax for Floating-Point Bit Operations

Table 20-14 BRIG Syntax for Floating-Point Classify (class) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_ABS	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	
BRIG_OPCODE_CLASS	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_COPYSIGN	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_NEG	BRIG_KIND_INST_BASIC (if only default modifiers are used) or BRIG_KIND_INST_MOD	<i>dest</i>	<i>src</i>	

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG OR BRIG_KIND_OPERAND_DATA.

18.7.1.12 BRIG Syntax for Native Floating-Point Operations

Table 20–15 BRIG Syntax for Native Floating-Point Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_NCOS	BRIG_KIND_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_NRSQRT		dest	src		
BRIG_OPCODE_NSIN		dest	src		
BRIG_OPCODE_NSQRT		dest	src		

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG or BRIG_KIND_OPERAND_DATA.

18.7.1.13 BRIG Syntax for Multimedia Operations

Table 20–16 BRIG Syntax for Multimedia Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
BRIG_OPCODE_BITALIGN	BRIG_KIND_INST_BASIC	dest	src	src	src	
BRIG_OPCODE_BYTEALIGN	BRIG_KIND_INST_BASIC	dest	src	src	src	
BRIG_OPCODE_LERP	BRIG_KIND_INST_BASIC	dest	src	src	src	
BRIG_OPCODE_PACKCVT	BRIG_KIND_INST_SOURCE_TYPE	dest	src	src	src	src
BRIG_OPCODE_UNPACKCVT	BRIG_KIND_INST_SOURCE_TYPE	dest	src	number		
BRIG_OPCODE_SAD	BRIG_KIND_INST_SOURCE_TYPE	dest	src	src	src	
BRIG_OPCODE_SADHI	BRIG_KIND_INST_SOURCE_TYPE	dest	src	src	src	

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

number: must be BRIG_KIND_OPERAND_DATA with value 0, 1, 2, or 3.

18.7.1.14 BRIG Syntax for Segment Checking (segmentp) Operation

Table 20–17 BRIG Syntax for Segment Checking (segmentp) Operation

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_SEGMENTP	BRIG_KIND_INST_SEG_CVT	dest	src

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG or BRIG_KIND_OPERAND_DATA.

18.7.1.15 BRIG Syntax for Segment Conversion Operations

Table 20–18 BRIG Syntax for Segment Conversion Operations

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_FTOS	BRIG_KIND_INST_SEG_CVT	<i>dest</i>	<i>src</i>
BRIG_OPCODE_STOF			

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG or BRIG_KIND_OPERAND_DATA.

18.7.1.16 BRIG Syntax for Compare (cmp) Operation

Table 20–19 BRIG Syntax for Compare (cmp) Operation

Opcode	Format	Operand 0	Operand 1	Operand 2
BRIG_OPCODE_CMP	BRIG_KIND_INST_CMP	<i>dest</i>	<i>src</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

The pack field of BRIG_KIND_INST_CMP should be set to BRIG_PACK_PP for packed source types and to BRIG_PACK_NONE otherwise.

18.7.1.17 BRIG Syntax for Conversion (cvt) Operation

Table 20–20 BRIG Syntax for Conversion (cvt) Operation

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_CVT	BRIG_KIND_INST_CVT	<i>dest</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.2 BRIG Syntax for Memory Operations

Table 20–21 BRIG Syntax for Memory Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_LD	BRIG_KIND_INST_MEM	<i>reg-or-vector</i>	<i>address</i>		
BRIG_OPCODE_ST	BRIG_KIND_INST_MEM	<i>reg-or-vector-or-num</i>	<i>address</i>		
BRIG_OPCODE_ATOMIC	BRIG_KIND_INST_ATOMIC	<i>dest</i>	<i>address</i>	<i>src</i>	

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
<code>BRIG_OPCODE_ATOMIC (for atomic_1d)</code>	<code>BRIG_KIND_INST_ATOMIC</code>	<code>dest</code>	<code>address</code>		
<code>BRIG_OPCODE_ATOMIC (for atomic_cas)</code>	<code>BRIG_KIND_INST_ATOMIC</code>	<code>dest</code>	<code>address</code>	<code>src</code>	<code>src</code>
<code>BRIG_OPCODE_ATOMICNORET</code>	<code>BRIG_KIND_INST_ATOMIC</code>	<code>address</code>	<code>src</code>		
<code>BRIG_OPCODE_ATOMICNORET (for atomicnoret_cas)</code>	<code>BRIG_KIND_INST_ATOMIC</code>	<code>address</code>	<code>src</code>	<code>src</code>	
<code>BRIG_OPCODE_SIGNAL</code>	<code>BRIG_KIND_INST_SIGNAL</code>	<code>dest</code>	<code>signal</code>	<code>src</code>	
<code>BRIG_OPCODE_SIGNAL (for signal_1d)</code>	<code>BRIG_KIND_INST_SIGNAL</code>	<code>dest</code>	<code>signal</code>		
<code>BRIG_OPCODE_SIGNAL (for signal_cas and signal_waiotimeout)</code>	<code>BRIG_KIND_INST_SIGNAL</code>	<code>dest</code>	<code>signal</code>	<code>src</code>	<code>src</code>
<code>BRIG_OPCODE_SIGNALNORET</code>	<code>BRIG_KIND_INST_SIGNAL</code>	<code>signal</code>	<code>src</code>		
<code>BRIG_OPCODE_SIGNALNORET (for signalnoret_cas)</code>	<code>BRIG_KIND_INST_SIGNAL</code>	<code>signal</code>	<code>src</code>	<code>src</code>	
<code>BRIG_OPCODE_MEMFENCE</code>	<code>BRIG_KIND_INST_MEMFENCE</code>				

`reg-or-vector`: must be `BRIG_KIND_OPERAND_REG`; or
`BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REG` operands.

`address`: must be `BRIG_KIND_OPERAND_ADDRESS`.

`signal`: must be `BRIG_KIND_OPERAND_REG`.

`reg-or-vector-or-num`: must be `BRIG_KIND_OPERAND_REG`;
`BRIG_KIND_OPERAND_DATA`; `BRIG_KIND_OPERAND_WAVESIZE`; or
`BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REG`,
`BRIG_KIND_OPERAND_DATA` or `BRIG_KIND_OPERAND_WAVESIZE` operands.

`dest`: must be `BRIG_KIND_OPERAND_REG`.

`src`: must be `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA`, or
`BRIG_KIND_OPERAND_WAVESIZE`.

18.7.3 BRIG Syntax for Image Operations

Table 20-22 BRIG Syntax for Image Operations

Opcode	Format	Oper. 0	Oper. 1	Oper. 2	Oper. 3
BRIG_OPCODE_RDIMAGE	BRIG_KIND_INST_IMAGE	<i>reg-or-4-vector-reg</i>	<i>image</i>	<i>sampler</i>	<i>reg-or-vector</i>
BRIG_OPCODE_LDIMAGE	BRIG_KIND_INST_IMAGE	<i>reg-or-4-vector-reg</i>	<i>image</i>	<i>reg-or-vector</i>	
BRIG_OPCODE_STIMAGE	BRIG_KIND_INST_IMAGE	<i>reg-or-4-vector-reg</i>	<i>image</i>	<i>reg-or-vector</i>	
BRIG_OPCODE_QUERYIMAGE	BRIG_KIND_INST_QUERYIMAGE	<i>dest</i>	<i>image</i>		
BRIG_OPCODE_QUERYSAMPLE	BRIG_KIND_INST_QUERYSAMPLE	<i>dest</i>	<i>sampler</i>		

dest: must be BRIG_KIND_OPERAND_REG.

reg-or-4-vector-reg: must be BRIG_KIND_OPERAND_REG; or
BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REG
operands.

image: must be BRIG_KIND_OPERAND_REG.

sampler: must be BRIG_KIND_OPERAND_REG.

reg-or-vector: must be BRIG_KIND_OPERAND_REG; or
BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REG,
BRIG_KIND_OPERAND_DATA or BRIG_KIND_OPERAND_WAVESIZE operands.

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.4 BRIG Syntax for Branch Operations

Table 20-23 BRIG Syntax for Branch Operations

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_BR	BRIG_KIND_INST_BR	<i>label</i>	
BRIG_OPCODE_CBR	BRIG_KIND_INST_BR	<i>condition</i>	<i>label</i>
BRIG_OPCODE_SBR	BRIG_KIND_INST_BR	<i>index</i>	<i>labels</i>

label: must be BRIG_KIND_OPERAND_CODE_REF that references a
BRIG_KIND_DIRECTIVE_LABEL directive in the same function scope.

condition: must be BRIG_KIND_OPERAND_REG for a c register,
BRIG_KIND_OPERAND_DATA, or BRIG_KIND_OPERAND_WAVESIZE.

index: must be BRIG_KIND_OPERAND_REG for an s or d register according to the operation type which must be u32 or u64.

labels: must be BRIG_KIND_OPERAND_CODE_LIST that references a list of BRIG_KIND_DIRECTIVE_LABEL directives all in the same function scope.

18.7.5 BRIG Syntax for Parallel Synchronization and Communication Operations

Table 20–24 BRIG Syntax for Parallel Synchronization and Communication Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
BRIG_OPCODE_BARRIER	BRIG_KIND_INST_BR					
BRIG_OPCODE_WAVEBARRIER	BRIG_KIND_INST_BR					
BRIG_OPCODE_INITFBAR	BRIG_KIND_INST_BASIC	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_JOINFBAR	BRIG_KIND_INST_BR	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_WAITFBAR	BRIG_KIND_INST_BR	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_ARRIVEFBAR	BRIG_KIND_INST_BR	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_LEAVEFBAR	BRIG_KIND_INST_BR	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_RELEASEFBAR	BRIG_KIND_INST_BASIC	<i>fbarrier-or-reg</i>				
BRIG_OPCODE_LDF	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>fbarrier</i>			
BRIG_OPCODE_ACTIVELANECOUNT	BRIG_KIND_INST_LANE	<i>dest</i>	<i>src</i>			
BRIG_OPCODE_ACTIVELANEID	BRIG_KIND_INST_LANE	<i>dest</i>				
BRIG_OPCODE_ACTIVELANEMASK	BRIG_KIND_INST_LANE	<i>4-vector-reg</i>	<i>src</i>			
BRIG_OPCODE_ACTIVELANESHUFFLE	BRIG_KIND_INST_LANE	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	<i>src</i>

fbarrier-or-reg: must be BRIG_KIND_OPERAND_REG; or
BRIG_KIND_OPERAND_CODE_REF that references a BRIG_KIND_DIRECTIVE_FBARRIER directive.

fbarrier: must be BRIG_KIND_OPERAND_CODE_REF that references a BRIG_KIND_DIRECTIVE_FBARRIER directive.

dest: must be BRIG_KIND_OPERAND_REG.

4-vector-reg: must be BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REG operands.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

18.7.6 BRIG Syntax for Function Operations

Table 20-25 BRIG Syntax for Operations Related to Functions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
<code>BRIG_OPCODE_CALL</code>	<code>BRIG_KIND_INST_BR</code>	<code>out-args</code>	<code>func</code>	<code>in-args</code>	
<code>BRIG_OPCODESCALL</code>	<code>BRIG_KIND_INST_BR</code>	<code>out-args</code>	<code>src</code>	<code>in-args</code>	<code>funcs</code>
<code>BRIG_OPCODE_ICALL</code>	<code>BRIG_KIND_INST_BR</code>	<code>out-args</code>	<code>reg</code>	<code>in-args</code>	<code>signature</code>
<code>BRIG_OPCODE_LDI</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>indirect-func</code>		
<code>BRIG_OPCODE_RET</code>	<code>BRIG_KIND_INST_BASIC</code>				
<code>BRIG_OPCODE_ALLOCA</code>	<code>BRIG_KIND_INST_MEM</code>	<code>dest</code>	<code>src</code>		

`dest`: must be `BRIG_KIND_OPERAND_REG`.

`src`: must be `BRIG_KIND_OPERAND_REG`, `BRIG_KIND_OPERAND_DATA`, or `BRIG_KIND_OPERAND_WAVESIZE`.

`reg`: must be `BRIG_KIND_OPERAND_REG`.

`out-args`: output arguments; must be `BRIG_KIND_OPERAND_CODE_LIST` that references a list of `BRIG_KIND_DIRECTIVE_VARIABLE` directives with `BRIG_SEGMENT_ARG` segment in the same arg block.

`in-args`: input arguments; must be `BRIG_KIND_OPERAND_CODE_LIST` that references a list of `BRIG_KIND_DIRECTIVE_VARIABLE` directives with `BRIG_SEGMENT_ARG` segment in the same arg block.

`func`: must be `BRIG_KIND_OPERAND_CODE_REF` that references a `BRIG_DIRECTIVE_FUNCTION` or `BRIG_DIRECTIVE_INDIRECT_FUNCTION` directive.

`indirect-func`: must be `BRIG_KIND_OPERAND_CODE_REF` that references a `BRIG_DIRECTIVE_INDIRECT_FUNCTION` directive.

`funcs`: must be `BRIG_KIND_OPERAND_CODE_LIST` that references a list of `BRIG_DIRECTIVE_FUNCTION` or `BRIG_DIRECTIVE_INDIRECT_FUNCTION` directives.

`signature`: must be `BRIG_KIND_OPERAND_CODE_REF` that references a `BRIG_KIND_DIRECTIVE_SIGNATURE` directive.

18.7.7 BRIG Syntax for Special Operations

18.7.7.1 BRIG Syntax for Dispatch Packet Operations

Table 20-26 BRIG Syntax for Dispatch Packet Operations

Opcode	Format	Operand 0	Operand 1
<code>BRIG_OPCODE_CURRENTWORKGROUPSIZE</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_DIM</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	
<code>BRIG_OPCODE_GRIDGROUPS</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_GRIDSIZE</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_PACKETCOMPLETIONSIG</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	
<code>BRIG_OPCODE_PACKETID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	

Opcode	Format	Operand 0	Operand 1
<code>BRIG_OPCODE_WORKGROUPID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_WORKGROUPSIZE</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_WORKITEMABSID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>
<code>BRIG_OPCODE_WORKITEMFLATABSID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	
<code>BRIG_OPCODE_WORKITEMFLATID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	
<code>BRIG_OPCODE_WORKITEMID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>dimNumber</code>

`dest`: must be `BRIG_KIND_OPERAND_REG`.

`dimNumber`: must be `BRIG_KIND_OPERAND_DATA` with the value 0, 1, or 2 corresponding to the X, Y, and Z dimensions respectively.

18.7.7.2 BRIG Syntax for Exception Operations

Table 20-27 BRIG Syntax for Exception Operations

Opcode	Format	Operand 0
<code>BRIG_OPCODE_CLEARDETECTEXCEPT</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>exceptionsNumber</code>
<code>BRIG_OPCODE_GETDETECTEXCEPT</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>
<code>BRIG_OPCODE_SETDETECTEXCEPT</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>exceptionsNumber</code>

`dest`: must be `BRIG_KIND_OPERAND_REG`.

`exceptionsNumber`: must be `BRIG_KIND_OPERAND_DATA`. bit:0=INVALID_OPERATION, bit: 1=DIVIDE_BY_ZERO, bit:2=OVERFLOW, bit:3=UNDERFLOW, bit:4=INEXACT; all other bits are ignored.

18.7.7.3 BRIG Syntax for User Mode Queue Operations

Table 20-28 BRIG Syntax for User Mode Queue Operations

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
<code>BRIG_OPCODE_AGENTCOUNT</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>			
<code>BRIG_OPCODE_AGENTID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>			
<code>BRIG_OPCODE_ADDQUEUEWRITEINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>dest</code>	<code>address</code>	<code>src</code>	
<code>BRIG_OPCODE_CASQUEUEWRITEINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>dest</code>	<code>address</code>	<code>src</code>	<code>src</code>
<code>BRIG_OPCODE_LDK</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>	<code>kernel</code>		
<code>BRIG_OPCODE_LDQUEUEREADINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>dest</code>	<code>address</code>		
<code>BRIG_OPCODE_LDQUEUEWRITEINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>dest</code>	<code>address</code>		
<code>BRIG_OPCODE_QUEUEID</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>			
<code>BRIG_OPCODE_QUEUEPTR</code>	<code>BRIG_KIND_INST_BASIC</code>	<code>dest</code>			
<code>BRIG_OPCODE_STQUEUEREADINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>address</code>	<code>src</code>		
<code>BRIG_OPCODE_STQUEUEWRITEINDEX</code>	<code>BRIG_KIND_INST_QUEUE</code>	<code>address</code>	<code>src</code>		

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

address: must be BRIG_KIND_OPERAND_ADDRESS.

kernel: must be BRIG_KIND_OPERAND_CODE_REF that references a
BRIG_KIND_DIRECTIVE_KERNEL directive.

18.7.7.4 BRIG Syntax for Miscellaneous Operations

Table 20–29 BRIG Syntax for Miscellaneous Operations

Opcode	Format	Operand 0
BRIG_OPCODE_CLOCK	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_CUID	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_DEBUGTRAP	BRIG_KIND_INST_BASIC	<i>src</i>
BRIG_OPCODE_GROUPBASEPTR	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_KERNARGBASEPTR	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_LANEID	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_MAXCUID	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_MAXWAVEID	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_NOP	BRIG_KIND_INST_BASIC	
BRIG_OPCODE_NULLPTR	BRIG_KIND_INST_SEG	<i>dest</i>
BRIG_OPCODE_WAVEID	BRIG_KIND_INST_BASIC	<i>dest</i>

dest: must be BRIG_KIND_OPERAND_REG.

src: must be BRIG_KIND_OPERAND_REG, BRIG_KIND_OPERAND_DATA, or
BRIG_KIND_OPERAND_WAVESIZE.

Chapter 19

HSAIL Grammar in Extended Backus-Naur Form

This chapter provides the HSAIL lexical and syntax grammar in Extended Backus-Naur Form.

19.1 HSAIL Lexical Grammar in Extended Backus-Naur Form (EBNF)

This appendix shows the HSAIL lexical grammar in Extended Backus–Naur Form (EBNF).

Symbol meanings are:

- `::=` grammar production
- `[]` optional
- `{ }` repetition
- `|` alternative
- '`[a-z]`' must be one of the characters in the `[]`
- '`[a-z]{n}`' must be exactly `n` of the characters in the `[]`
- '`[a-z]{n,m}`' must be between `n` and `m` of the characters in the `[]`
- '`not [a-z]`' must not be one of the characters in the `[]`

```

TOKEN_COMMENT ::= ( /* { 'not [*]' | **'not [/]' } */ )
                | ( // { 'not new-line' }
                )

TOKEN_GLOBAL_IDENTIFIER ::= "&" identifier

TOKEN_LOCAL_IDENTIFIER ::= "%" identifier

TOKEN_LABEL_IDENTIFIER ::= "@" identifier

identifier ::= '[a-zA-Z_]' { '[a-zA-Z0-9_]' }

TOKEN_CREGISTER ::= "$c" registerNumber

TOKEN_SREGISTER ::= "$s" registerNumber

TOKEN_DREGISTER ::= "$d" registerNumber

TOKEN_QREGISTER ::= "$q" registerNumber

registerNumber ::= "0"
                  | '[1-9]' { '[0-9]' }

TOKEN_INTEGER_CONSTANT ::= decimalIntegerConstant
                         | hexIntegerConstant
                         | octalIntegerConstant

decimalIntegerConstant ::= "0" | ( '[1-9]' { '[0-9]' } )

hexIntegerConstant ::= "0" ( "x" | "X" ) '[0-9a-fA-F]' {
                           '[0-9a-fA-F]' }

octalIntegerConstant ::= "0" '[0-7]' { '[0-7]' }

TOKEN_HALF_CONSTANT ::= decimalFloatConstant ( "h" | "H" )
                      | hexFloatConstant ( "h" | "H" )
                      | ieeeHalfConstant

TOKEN_SINGLE_CONSTANT ::= decimalFloatConstant ( "f" | "F" )
                       | hexFloatConstant ( "f" | "F" )
                       | ieeeSingleConstant

TOKEN_DOUBLE_CONSTANT ::= decimalFloatConstant [ "d" | "D" ]
                        | hexFloatConstant [ "d" | "D" ]
                        | ieeeDoubleConstant

decimalFloatConstant ::= ( ( '[0-9]' { '[0-9]' } "."
                           | { '[0-9]' } "." '[0-9]' { '[0-9]' }
                           )
                           [ ( "e" | "E" ) [ "+" | "-" ] '[0-9]' {
                               '[0-9]' }
                           ]
                           | '[0-9]' { '[0-9]' } ( "e" | "E" )
                           [ "+" | "-" ] '[0-9]' { '[0-9]' }

hexFloatConstant ::= "0" ( "x" | "X" )
                     ( '[0-9a-fA-F]' { '[0-9a-fA-F]' }
                     [ "."
                     | { '[0-9a-fA-F]' } "." '[0-9a-fA-F]'
                     { '[0-9a-fA-F]' }
                     )
                     ( "p" | "P" ) [ "+" | "-" ] '[0-9]' {
                         '[0-9]' }

ieeeHalfConstant ::= "0" ( "h" | "H" ) '[0-9a-fA-F]{4}'"

ieeeSingleConstant ::= "0" ( "f" | "F" ) '[0-9a-fA-F]{8}'"

ieeeDoubleConstant ::= "0" ( "d" | "D" ) '[0-9a-fA-F]{16}'"

```

```

TOKEN_WAVESIZE      ::= "WAVESIZE"
TOKEN_STRING        ::= """
                           { 'not' ([\"] or new-line)
                           | \""
                           | """
                           | "[\ '?abfnrtv]"
                           | '[0-7]{1,3}'
                           | "x" '[0-9a-fA-F]' { '[0-9a-fA-F]' }
                           )
                           """

```

19.2 HSAIL Syntax Grammar in Extended Backus-Naur Form (EBNF)

This appendix shows the HSAIL syntax grammar in Extended Backus–Naur Form (EBNF).

Symbol meanings are:

- ::= grammar production
- [] optional
- {} repetition
- | alternative

```

module                               ::= annotations version annotations
                                         { moduleDirective annotations }
                                         { moduleStatement annotations }
annotations                         ::= { annotation }
annotation                          ::= TOKEN_COMMENT
                                         | location
                                         | pragma
location                            ::= "loc"
                                         TOKEN_INTEGER_CONSTANT
                                         [ TOKEN_INTEGER_CONSTANT ]
                                         [ TOKEN_STRING ] ";"
pragma                             ::= "pragma" pragmaOperand { "," pragmaOperand } ";""
version                            ::= "version"
                                         TOKEN_INTEGER_CONSTANT ":""
                                         TOKEN_INTEGER_CONSTANT ":""
                                         profile ":""
                                         machineModel ";"
profile                            ::= "$full"
                                         | "$base"
                                         ::= "$small"
                                         | "$large"
extension                           ::= extension TOKEN_STRING ";"
moduleStatement                     ::= moduleVariable
                                         | moduleFbarrier
                                         | kernel
                                         | function
                                         | signature
moduleVariable                    ::= optDeclQual linkageQual variable
variable                           ::= optAllocQual optAlignQual optConstQual variableSegment
                                         ( nonOpaqueType TOKEN_GLOBAL_IDENTIFIER optArrayDimension
                                         optNonOpaqueInitializer
                                         | imageType TOKEN_GLOBAL_IDENTIFIER optArrayDimension
                                         optImageInitializer
                                         | samplerType TOKEN_GLOBAL_IDENTIFIER optArrayDimension
                                         optSamplerInitializer
                                         | signalType TOKEN_GLOBAL_IDENTIFIER optArrayDimension
                                         optSignalInitializer
                                         ) ";"
                                         ::= [ "="
                                         ( integerInitializer
                                         | floatInitializer
                                         | packedInitializer ) ]
optNonOpaqueInitializer          ::= [ "="
                                         ( imageInitializer ]
                                         | samplerInitializer ]
                                         | integerInitializer ]
                                         ::= "{" integerList "}"
                                         | integerConstant
                                         ::= { integerConstant "," } integerConstant
                                         ::= TOKEN_INTEGER_CONSTANT
                                         | "+" TOKEN_INTEGER_CONSTANT
                                         | "-" TOKEN_INTEGER_CONSTANT
floatInitializer                   ::= doubleInitializer
                                         | singleInitializer
                                         | halfInitializer
                                         ::= "{" doubleList "}"
                                         | doubleConstant
                                         ::= { doubleConstant "," } doubleConstant
                                         ::= "{" singleList "}"
                                         | singleConstant
                                         ::= { singleConstant "," } singleConstant
                                         ::= "{" halfList "}"
                                         | halfConstant
                                         ::= { halfConstant "," } halfConstant
                                         ::= doubleConstant
                                         | singleConstant
                                         | halfConstant
doubleConstant                     ::= TOKEN_DOUBLE_CONSTANT
                                         | "+" TOKEN_DOUBLE_CONSTANT
                                         | "-" TOKEN_DOUBLE_CONSTANT

```

```

singleConstant          ::= TOKEN_SINGLE_CONSTANT
                           | "+" TOKEN_SINGLE_CONSTANT
                           | "-" TOKEN_SINGLE_CONSTANT
halfConstant            ::= TOKEN_HALF_CONSTANT
                           | "+" TOKEN_HALF_CONSTANT
                           | "-" TOKEN_HALF_CONSTANT
packedInitializer       ::= "{" packedList "}"
                           | packedConstant
packedList              ::= { packedConstant "," } packedConstant
packedConstant          ::= packedType "(" integerList ")"
                           | packedType "(" halfList ")"
                           | packedType "(" singleList ")"
                           | packedType "(" doubleList ")"
imageInitializer        ::= "{" imageList "}"
                           | imageConstant
imageList               ::= { imageConstant "," } imageConstant
imageConstant           ::= imageType "(" imagePropertyList ")"
imagePropertyList       ::= { imageProperty "," } imageProperty
imageProperty           ::= ( "geometry" "==" imageGeometry )
                           | ( "width" "==" TOKEN_INTEGER_CONSTANT )
                           | ( "height" "==" TOKEN_INTEGER_CONSTANT )
                           | ( "depth" "==" TOKEN_INTEGER_CONSTANT )
                           | ( "array" "==" TOKEN_INTEGER_CONSTANT )
                           | ( "channel_order" "==" imageChannelOrder )
                           | ( "channel_type" "==" imageChannelType )
imageGeometry            ::= "1d"
                           | "2d"
                           | "3d"
                           | "1da"
                           | "2da"
                           | "1db"
                           | "2dddepth"
                           | "2dadepth"
imageChannelType         ::= "snorm_int8"
                           | "snorm_int16"
                           | "unorm_int8"
                           | "unorm_int16"
                           | "unorm_int24"
                           | "unorm_short_555"
                           | "unorm_short_565"
                           | "unorm_short_101010"
                           | "signed_int8"
                           | "signed_int16"
                           | "signed_int32"
                           | "unsigned_int8"
                           | "unsigned_int16"
                           | "unsigned_int32"
                           | "half_float"
                           | "float"
imageChannelOrder         ::= "a"
                           | "r"
                           | "rx"
                           | "rg"
                           | "rgx"
                           | "ra"
                           | "rgb"
                           | "rgbx"
                           | "rgba"
                           | "bgra"
                           | "argb"
                           | "abgr"
                           | "srgb"
                           | "srgb2"
                           | "srgbx"
                           | "srgba"
                           | "sbgra"
                           | "intensity"
                           | "luminance"
                           | "depth"
                           | "depth_stencil"
samplerInitializer        ::= "{" samplerList "}"

```

```

    | samplerConstant
samplerList      ::= { samplerConstant "," } samplerConstant
samplerConstant ::= samplerType "(" samplerPropertyList ")"
samplerPropertyList ::= { samplerProperty "," } samplerProperty
samplerProperty  ::= ( "coord" "==" samplerCoord )
                  | ( "filter" "==" samplerFilter )
                  | ( "addressing" "==" samplerAddressing )
samplerCoord      ::= "normalized"
                  | "unnormalized"
samplerFilter      ::= "nearest"
                  | "linear"
                  | "undefined"
                  | "clamp_to_edge"
                  | "clamp_to_border"
                  | "repeat"
                  | "mirrored_repeat"
moduleFbarrier     ::= optDeclQual linkageQual fbarrier
fbarrier          ::= "fbarrier" TOKEN_GLOBAL_IDENTIFIER ";"
kernel             ::= declQual linkageQual kernelHeader ";"
                      | linkageQual kernelHeader codeBlock ";"
kernelHeader       ::= "kernel" TOKEN_GLOBAL_IDENTIFIER kernFormalArgumentList
kernFormalArgumentList ::= "(" [ { kernFormalArgument "," } kernFormalArgument ] ")"
kernFormalArgument ::= optAlignQual "kernarg" dataType
                      TOKEN_LOCAL_IDENTIFIER optArrayDimension
function           ::= declQual linkageQual functionHeader ";"
                      | linkageQual functionHeader codeBlock ";"
functionHeader     ::= [ "indirect" ] "function" TOKEN_GLOBAL_IDENTIFIER
                      funcOutputFormalArgumentList funcInputFormalArgumentList
funcOutputFormalArgumentList ::= functionFormalArgumentList
funcInputFormalArgumentList ::= functionFormalArgumentList
funcFormalArgumentList ::= "(" [ { funcFormalArgument "," } funcFormalArgument ] ")"
funcFormalArgument ::= optAlignQual "arg" dataType
                      TOKEN_LOCAL_IDENTIFIER optArrayDimension
signature          ::= "signature" TOKEN_GLOBAL_IDENTIFIER
                      sigOutputFormalArgumentList sigInputFormalArgumentList ";"
sigOutputFormalArgumentList ::= sigFormalArgumentList
sigInputFormalArgumentList ::= sigFormalArgumentList
sigFormalArgumentList ::= "(" [ { sigFormalArgument "," } sigFormalArgument ] ")"
sigFormalArgument ::= optAlignQual "arg" dataType
                      [ TOKEN_LOCAL_IDENTIFIER ] optArrayDimension
linkageQual        ::= [ "prog" ]
optDeclQual        ::= [ declQual ]
declQual           ::= "decl"
                      [ "const" ]
                      [ "align" "(" TOKEN_INTEGER_CONSTANT ")" ]
                      [ "alloc" "(" allocationKind ")" ]
                      "agent"
                      [ "[" [ TOKEN_INTEGER_CONSTANT ] "]" ]
                      "{"
                      annotations
                        { codeBlockDirective annotations }
                        { codeBlockDefinition annotations }
                        { codeBlockStatement annotations }
                      "}"
codeBlock           ::= control
control            ::= "enablebreakexceptions" immediateOperand ";"
                      | "enabledetectexceptions" immediateOperand ";"
                      | "maxdynamicgroupsize" immediateOperand ";"
                      | "maxflatgridsize" immediateOperand ";"
                      | "maxflatworkgroupsize" immediateOperand ";"
                      | "requestedworkgroupspercu" immediateOperand ";"
                      | "requireddim" immediateOperand ";"
                      | "requiredgridsize" immediateOperand ","
                        immediateOperand ","
                        immediateOperand ";"
                      | "requiredworkgroupsize" immediateOperand ","
                        immediateOperand ","
                        immediateOperand ";"
                      | "requirenopartialworkgroups" ";"
codeBlockDefinition ::= codeBlockVariable
                      | codeBlockFbarrier
codeBlockVariable   ::= variable
codeBlockFbarrier   ::= fbarrier

```

```

codeBlockStatement      ::= argBlock
                           | label
                           | operation
argBlock               ::= "{" annotations
                           { argBlockDefinition annotations }
                           { argBlockStatement annotations }
                           "}"
argBlockDefinition    ::= argBlockVariable
argBlockVariable       ::= variable
argBlockStatement      ::= label
                           | operation
                           | call
label                 ::= TOKEN_LABEL_IDENTIFIER ":""
operation              ::= instruction0
                           | instruction1
                           | instruction2
                           | instruction3
                           | instruction4
                           | mul
                           | bitinsert
                           | combine
                           | expand
                           | lda
                           | mov
                           | pack
                           | unpack
                           | packcvt
                           | unpackcvt
                           | sad
                           | segmentConversion
                           | cmp
                           | cvt
                           | ld
                           | st
                           | atomic
                           | atomicnoret
                           | signal
                           | signalnoret
                           | memfence
                           | rdimage
                           | stimage
                           | ldiimage
                           | queryimage
                           | querysampler
                           | branch
                           | barrier
                           | wavebarrier
                           | fbarrier
                           | crossLane
                           | ldi
                           | ret
                           | alloca
                           | packetcompletionsig
                           | queue
                           | ldk
instruction0           ::= "nop" ";"
instruction1           ::= ( instruction1Opcode optRoundingMod nonOpaqueType
                           | "nullptr" optSegmentMod nonOpaqueType
                           )
                           operand ";"
instruction1Opcode     ::= "agentid"
                           | "agentcount"
                           | "cleardetectexcept"
                           | "clock"
                           | "cuid"
                           | "debugtrap"
                           | "dim"
                           | "getdetectexcept"
                           | "groupbaseptr"
                           | "kernargbaseptr"

```

```

| "laneid"
| "maxcuid"
| "maxwaveid"
| "packetid"
| "queueid"
| "queueptr"
| "setdetectexcept"
| "waveid"
| "workitemflatsid"
| "workitemflatid"
instruction2      ::= ( instruction2Opcode optRoundingMod optPackingMod
| instruction2OpcodeFtz optFtz optPackingMod
| "popcount" nonOpaqueType
| "firstbit" nonOpaqueType
| "lastbit" nonOpaqueType
)
nonOpaqueType operand "," operand ";"
instruction2Opcode ::= "abs"
| "bitrev"
| "currentworkgroupsize"
| "ncos"
| "neg"
| "nexp2"
| "nlog2"
| "nrccp"
| "nrsqrt"
| "nsin"
| "nsqrt"
| "gridgroups"
| "gridsize"
| "not"
| "sqrt"
| "workgroupid"
| "workgroupsize"
| "workitemabsid"
| "workitemid"
instruction2OpcodeFtz ::= "ceil"
| "floor"
| "fract"
| "rint"
| "trunc"
instruction3      ::= ( instruction3Opcode optRoundingMod optPackingMod
| instruction3OpcodeFtz optFtz optPackingMod
| "class" nonOpaqueType
)
nonOpaqueType operand "," operand "," operand ";"
instruction3Opcode ::= "add"
| "bitmask"
| "borrow"
| "carry"
| "copysign"
| "div"
| "rem"
| "sub"
| "shl"
| "shr"
| "and"
| "or"
| "xor"
| "unpackhi"
| "unpacklo"
instruction3OpcodeFtz ::= "max"
| "min"
instruction4      ::= ( instruction4Opcode optRoundingMod
| instruction4OpcodeFtz optFtz optPackingMod
)
nonOpaqueType operand "," operand "," operand ","
instruction4Opcode ::= "fma"
| "mad"
| "bitextract"

```

```

| "bitselect"
| "shuffle"
| "cmov"
| "bitalign"
| "bytealign"
| "lerp"
instruction4OpcodeFtz
mul ::= "nfma"
::= ( "mul" optRoundingMod optPackingMod nonOpaqueType
      | "mulhi" optPackingMod nonOpaqueType
      | "mul24hi" nonOpaqueType
      | "mul24" nonOpaqueType
      | "mad24" nonOpaqueType operand ","
      | "mad24hi" nonOpaqueType operand ","
      ) operand "," operand "," operand ";"
bitinsert ::= "bitinsert" nonOpaqueType operand ","
            operand "," operand ","
combine ::= "combine" vectorMod nonOpaqueType nonOpaqueType
           operand "," vectorOperand ";
expand ::= "expand" vectorMod nonOpaqueType nonOpaqueType
           vectorOperand "," operand ";
lda ::= "lda" optSegmentMod nonOpaqueType operand ","
       memoryOperand ";"
mov ::= "mov" dataType operand "," operand ";
pack ::= "pack" nonOpaqueType nonOpaqueType operand ","
        operand "," operand ",";
unpack ::= "unpack" nonOpaqueType nonOpaqueType operand ","
          operand "," operand ";
packcvt ::= "packcvt" nonOpaqueType nonOpaqueType operand ","
           operand "," operand "," operand ";
unpackcvt ::= "unpackcvt" nonOpaqueType nonOpaqueType operand ","
             operand "," operand ";
sad ::= ( "sad" | "sadhi" ) nonOpaqueType nonOpaqueType
      operand "," operand "," operand ",";
segmentConversion ::= ( "segmentp" | "ftos" | "stof" )
                      segmentMod optNullMod nonOpaqueType nonOpaqueType
                      operand "," operand ";
cmp ::= "cmp" comparisonOp optFtz optPackingMod nonOpaqueType
      nonOpaqueType operand "," operand "," operand ";
comparisonOp ::= "_eq"
| "_ne"
| "_lt"
| "_le"
| "_gt"
| "_ge"
| "_equ"
| "_neu"
| "_ltu"
| "_leu"
| "_gtu"
| "_geu"
| "_num"
| "_nan"
| "_seq"
| "_sne"
| "_slt"
| "_sle"
| "_sgt"
| "_sge"
| "_snum"
| "_snan"
| "_sequ"
| "_sneu"
| "_slt"
| "_sle"
| "_sgtu"
| "_sgeu"
cvt ::= "cvt" optCvtRoundingMod nonOpaqueType
      nonOpaqueType operand "," operand ";
optCvtRoundingMod ::= [ cvtRoundingMod ]
cvtRoundingMod ::= floatRoundingMod

```

```

| "_ftz"
| "_ftz" floatRoundingMod
| intRoundingMod
| "_ftz" intRoundingMod
ld ::= "ld" optVectorMod optSegmentMod
      optAlignMod optConstMod optEquivMod optWidthMod dataType
      possibleVectorOperand "," memoryOperand ";"
st ::= "st" optVectorMod optSegmentMod
      optAlignMod optEquivMod dataType
      possibleVectorOperand "," memoryOperand ";"
atomic ::= "atomic"
        ( atomicOp
          optSegmentMod memOrderMod memScopeMod optEquivMod
          nonOpaqueType operand "," memoryOperand "," operand
        | "_ld"
          optSegmentMod ldMemOrderMod memScopeMod optEquivMod
          nonOpaqueType operand "," memoryOperand
        | "_cas"
          optSegmentMod memOrderMod memScopeMod optEquivMod
          nonOpaqueType
          operand "," memoryOperand "," operand ","
          operand ","
        ) ";"
atomicnoret ::= "atomicnoret"
        ( atomicOp
          optSegmentMod memOrderMod memScopeMod optEquivMod
          nonOpaqueType memoryOperand "," operand
        | "_st"
          optSegmentMod stMemOrderMod memScopeMod optEquivMod
          nonOpaqueType memoryOperand ","
          operand
        | "_cas"
          optSegmentMod memOrderMod memScopeMod optEquivMod
          nonOpaqueType memoryOperand ","
          operand ","
          operand
        ) ";"
atomicOp ::= "_add"
        | "_and"
        | "_exch"
        | "_max"
        | "_min"
        | "_or"
        | "_sub"
        | "_wrapdec"
        | "_wrapinc"
        | "_xor"
signal ::= "signal"
        ( atomicOp
          memOrderMod nonOpaqueType signalType
          operand "," operand ","
          operand
        | "_ld"
          ldMemOrderMod nonOpaqueType signalType
          operand ","
          operand
        | "_cas"
          memOrderMod nonOpaqueType signalType
          operand ","
          operand ","
          operand
        | "_wait"
          waitOp memOrderMod nonOpaqueType signalType
          operand ","
          operand ","
          operand
        | "_waittimeout"
          waitOp memOrderMod nonOpaqueType signalType
          operand ","
          operand ","
          operand
        ) ";"
signalnoret ::= "signalnoret"
        ( atomicOp
          memOrderMod nonOpaqueType signalType
          operand ","
          operand
        | "_st"
          stMemOrderMod nonOpaqueType signalType
          operand ","
          operand
        | "_cas"
          memOrderMod nonOpaqueType signalType
          operand ","
          operand
        ) ";"

```

```

) ;"
waitOp ::= "_eq"
| "_ne"
| "_lt"
| "_gte"
memfence ::= "memfence" fenceMemOrderMod fenceSegmentScopeListMod ;"
fenceSegmentScopeListMod ::= { fenceSegmentScopeMod } fenceSegmentScopeMod
fenceSegmentScopeMod ::= "_group(" ( "wv" | "wg" ) ")"
| "_global(" ( "wv" | "wg" | "cmp" | "sys" ) ")"
| "_image(" ( "wi" | "wv" | "wg" ) ")"
::= "_scacq" | "_screl" | "_scar" | "_rlx"
::= "_screl" | "_rlx"
::= "_scacq" | "_rlx"
::= "_scacq" | "_screl" | "_scar"
::= "_wv" | "_wg" | "_cmp" | "_sys"
::= "rdimage" [ "_v4" ] geometryMod optEquivMod nonOpaqueType
    imageType nonOpaqueType possibleVectorOperand "," operand ","
    operand "," possibleVectorOperand ;"
ldimage ::= "ldimage" [ "_v4" ] geometryMod optEquivMod nonOpaqueType
    imageType nonOpaqueType possibleVectorOperand ","
    operand "," possibleVectorOperand ;"
stimage ::= "stimage" [ "_v4" ] geometryMod optEquivMod nonOpaqueType
    imageType nonOpaqueType possibleVectorOperand ","
    operand "," possibleVectorOperand ;"
geometryMod ::= "_1d"
| "_2d"
| "_3d"
| "_1da"
| "_2da"
| "_1db"
| "_2dddepth"
| "_2dadepth"
queryimage ::= "queryimage" geometryMod queryimageOp nonOpaqueType imageType
    operand "," operand ;"
queryimageOp ::= "_width"
| "_height"
| "_depth"
| "_array"
| "_channelorder"
| "_channeletype"
querysampler ::= "querysampler" querysamplerOp nonOpaqueType
    operand "," operand ;"
querysamplerOp ::= "_coord"
| "_filter"
| "_addressing"
branch ::= "br" TOKEN_LABEL_IDENTIFIER ;"
| "cbr" optWidthMod nonOpaqueType
    TOKEN_CREGISTER , TOKEN_LABEL_IDENTIFIER ;"
| "sbr" optWidthMod nonOpaqueType operand branchTargets ;"
branchTargets ::= "[" { TOKEN_LABEL_IDENTIFIER , } TOKEN_LABEL_IDENTIFIER "]"
barrier ::= "barrier" optWidthMod ;"
wavebarrier ::= "wavebarrier" ;"
fbarrier ::= "initfbar" operand ;"
| "joinfbar" optWidthMod operand ;"
| "waitfbar" optWidthMod operand ;"
| "arrivefbar" optWidthMod operand ;"
| "leavefbar" optWidthMod operand ;"
| "releasefbar" operand ;"
| "ldf" nonOpaqueType operand , nonRegisterIdentifier ;"
crossLane ::= "activelaneid" optWidthMod nonOpaqueType operand ;"
| "activelanecount" optWidthMod nonOpaqueType nonOpaqueType
    operand , operand ;"
| "activelanemask" "_v4" optWidthMod nonOpaqueType nonOpaqueType
    vectorOperand , operand ;"
| "activelanesshuffle" optWidthMod nonOpaqueType
    operand , operand , operand , operand
call ::= "call" TOKEN_GLOBAL_IDENTIFIER
    callOutputActualArgs callInputActualArgs ;"
| "scal" optWidthMod nonOpaqueType operand
    callOutputActualArgs callInputActualArgs

```

```

callTargets ";"  

| "icall" optWidthMod nonOpaqueType operand  

callOutputActualArguments callInputActualArguments  

TOKEN_GLOBAL_IDENTIFIER ";"  

callOutputActualArguments ::= callActualArguments  

callInputActualArguments ::= callActualArguments  

callActualArguments ::= "(" [ operandList ] ")"  

callTargets ::= "[" { TOKEN_GLOBAL_IDENTIFIER "," }  

TOKEN_GLOBAL_IDENTIFIER "]"  

ldi ::= "ldi" nonOpaqueType operand "," TOKEN_GLOBAL_IDENTIFIER ";"  

ret ::= "ret" ";"  

alloca ::= "alloca" optAlignMod nonOpaqueType  

operand "," operand ";"  

packetcompletionsig ::= "packetcompletionsig" signalType operand ";"  

queue ::= "addqueueuwriteindex" segmentMod memOrderMod nonOpaqueType  

operand "," memoryOperand "," operand ";"  

| "casqueuwriteindex" segmentMod memOrderMod nonOpaqueType  

operand "," memoryOperand "," operand "," operand ";"  

| ( "ldqueueureadindex" | "ldqueueuwriteindex" )  

segmentMod memOrderMod nonOpaqueType  

operand "," memoryOperand ";"  

| ( "stqueueureadindex" | "stqueueuwriteindex" )  

segmentMod memOrderMod nonOpaqueType  

memoryOperand "," operand ";"  

ldk ::= "ldk" nonOpaqueType operand "," TOKEN_GLOBAL_IDENTIFIER ";"  

pragmaOperand ::= TOKEN_STRING  

| integerConstant  

| identifierOperand  

operand ::= immediateOperand  

| identifierOperand  

immediateOperand ::= integerConstant  

| floatConstant  

| packedConstant  

| TOKEN_WAVESIZE  

memoryOperand ::= symbolicAddressableOperand  

| offsetAddressableOperand  

| symbolicAddressableOperand offsetAddressableOperand  

symbolicAddressableOperand ::= "[" nonRegisterIdentifier "]"  

offsetAddressableOperand ::= "[" registerIdentifier "+" TOKEN_INTEGER_CONSTANT "]"  

| "[" registerIdentifier "-" TOKEN_INTEGER_CONSTANT "]"  

| "[" registerIdentifier "]"  

| "[" TOKEN_INTEGER_CONSTANT "]"  

| "[" "+" TOKEN_INTEGER_CONSTANT "]"  

| "[" "-" TOKEN_INTEGER_CONSTANT "]"  

possibleVectorOperand ::= operand  

| vectorOperand  

vectorOperand ::= "(" operandList ")"  

operandList ::= { operand "," } operand  

identifierOperand ::= nonRegisterIdentifier  

| registerIdentifier  

nonRegisterIdentifier ::= TOKEN_GLOBAL_IDENTIFIER  

| TOKEN_LOCAL_IDENTIFIER  

registerIdentifier ::= TOKEN_CREGISTER  

| TOKEN_DREGISTER  

| TOKEN_QREGISTER  

| TOKEN_SREGISTER  

variableSegment ::= "readonly"  

| "global"  

| "private"  

| "group"  

| "spill"  

| "arg"  

segmentMod ::= "_readonly"  

| "_kernarg"  

| "_global"  

| "_private"  

| "_arg"  

| "_group"  

| "_spill"  

optSegmentMod ::= [ segmentMod ]

```

```

optAlignMod           ::= [ "_align" "(" TOKEN_INTEGER_CONSTANT ")" ]
optConstMod          ::= [ "_const" ]
optEquivMod          ::= [ "_equiv" "(" TOKEN_INTEGER_CONSTANT ")" ]
optNullMod           ::= [ "_nonnull" ]
optWidthMod          ::= [ "_width" "("
                           ( "all"
                           | TOKEN_WAVESIZE
                           | TOKEN_INTEGER_CONSTANT
                           ) ")"
                         ]
optVectorMod          ::= [ vectorMod ]
vectorMod            ::= "_v2"
                      | "_v3"
                      | "_v4"
optRoundingMod        ::= optFtz [ floatRoundingMod ]
optFtz               ::= [ "_ftz" ]
floatRoundingMod     ::= "_up"
                      | "_down"
                      | "_zero"
                      | "_near"
intRoundingMod       ::= "_upi"
                      | "_downi"
                      | "_zeroi"
                      | "_neari"
                      | "_upi_sat"
                      | "_downi_sat"
                      | "_zeroi_sat"
                      | "_neari_sat"
                      | "_upi"
                      | "_downi"
                      | "_zeroi"
                      | "_neari"
                      | "_upi_sat"
                      | "_downi_sat"
                      | "_zeroi_sat"
                      | "_neari_sat"
optPackingMod         ::= [ packingMod ]
packingMod           ::= "_pp"
                      | "_ps"
                      | "_sp"
                      | "_ss"
                      | "_s"
                      | "_p"
                      | "_pp_sat"
                      | "_ps_sat"
                      | "_sp_sat"
                      | "_ss_sat"
                      | "_s_sat"
                      | "_p_sat"
dataType              ::= baseType
                      | packedType
                      | opaqueType
nonOpaqueType         ::= baseType
                      | packedType
baseType              ::= intType
                      | floatType
                      | bitType
intType               ::= "_u8"
                      | "_s8"
                      | "_u16"
                      | "_s16"
                      | "_u32"
                      | "_s32"
                      | "_u64"
                      | "_s64"
floatType             ::= "_f16"
                      | "_f32"
                      | "_f64"
bitType               ::= "_b1"
                      | "_b8"
                      | "_b16"

```

```

| "_b32"
| "_b64"
| "_b128"
packedType ::= "_u8x4"
| "_s8x4"
| "_u16x2"
| "_s16x2"
| "_f16x2"
| "_u8x8"
| "_s8x8"
| "_u16x4"
| "_s16x4"
| "_f16x4"
| "_u32x2"
| "_s32x2"
| "_f32x2"
| "_u8x16"
| "_s8x16"
| "_u16x8"
| "_s16x8"
| "_u32x4"
| "_f16x8"
| "_s32x4"
| "_f32x4"
| "_u64x2"
| "_s64x2"
| "_f64x2"
opaqueType ::= imageType
| samplerType
| signalType
imageType ::= "_roimg"
| "_woimg"
| "_rwimg"
samplerType ::= "_samp"
signalType ::= "_sig32"
| "_sig64"

```

Appendix A

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 256 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 segment memory: Every implementation must support at least 32K bytes of group segment 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 [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#)) and the exception detection operations (see [11.2 Exception Operations \(p. 274\)](#)).
- Size of private segment memory: Every implementation must support at least 64K bytes of private segment memory per work-group.
- Size of kernarg segment memory: Every implementation must support at least 1K bytes of kernarg segment memory per dispatch (see [4.21 Kernarg Segment \(p. 96\)](#)).

- **Size of arg segment memory:** Every implementation must support at least 64 bytes of arg segment variables per argument scope (see [10.2 Function Call Argument Passing \(p. 254\)](#)).
- **Image data type support:** Every agent that supports images must support images as defined in [Chapter 7 Image Operations \(p. 197\)](#) with the following per agent limits:
 - **1D images:** Must support 1D, 1DA image sizes up to 16384 image elements for width.
 - **2D images:** Must support 2D, 2DA, 2DDEPTH, 2DADEPTH image sizes up to (16384 x 16384) image elements for (width, height) respectively.
 - **3D images:** Must support 3D image sizes up to (2048 x 2048 x 2048) image elements for (width, height, depth) respectively.
 - **Image arrays:** Must support 1DA, 2DA, 2DADEPTH image arrays up to 2048 image layers for array size.
 - **1DB images:** Must support 1DB image sizes up to 65536 image elements for width.
 - **Read-only image handles:** Must support having at least 128 read-only image handles created at any one time.
 - **Write-only and read-write image handles:** Must support having at combined total of at least 64 write-only and read-write image handles created at any one time.
 - **Sampler handles:** Must support having at least 16 sampler handles created at any one time.

Appendix B

Glossary of HSAIL Terms

acquire synchronizing operation

A memory operation marked with acquire (an `ld_acq`, `atomic_ar`, or `atomicnoret_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? \(p. 1\)](#).

application program

An executable that can be executed on a host CPU. In addition to the host CPU code, it may include zero or more HSAIL programs that can each include zero or more HSAILmodules. See [4.2 Program \(p. 35\)](#) and [4.3 Module \(p. 38\)](#).

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 \(p. 15\)](#) and [10.2 Function Call Argument Passing \(p. 254\)](#).

BRIG

The HSAIL binary format. See [Chapter 18 BRIG: HSAIL Binary Format \(p. 317\)](#).

call convention

Each HSA component can support one or more call conventions. For example, an HSA component may have different call conventions that each use a different number of isa registers to allow different numbers of wavefronts to execute on a compute unit. See [4.2.2 Call Convention Id \(p. 36\)](#).

compound type

A type made up of a base data type and a length. See [4.13.1 Base Data Types \(p. 79\)](#).

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 [2.12 Divergent Control Flow \(p. 26\)](#).

fbarrier

A fine-grain barrier that applies to a subset of a work-group. See [9.2 Fine-Grain Barrier \(fbar\) Operations \(p. 236\)](#).

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 [13.1 extension Directive \(p. 291\)](#).

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 \(p. 15\)](#).

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 \(p. 15\)](#).

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 one or more host CPU agents able to execute the HSA runtime, one or more HSA components able to execute HSAIL programs, and zero or more other agents that participate in the HSA memory model.

HSA runtime

A library of services that can be executed by the application on a host CPU that supports the execution of HSAIL programs. This includes a finalizer that translates HSAIL code into the appropriate native ISA for each HSA component that is part of the HSA system. In addition, it supports a runtime queue that can be used by any agent, including HSA components, to submit agent dispatch packets to perform runtime functions.

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 handle

An opaque handle to an image that includes information about the properties of the image and access to the image data. See [7.1.7 Image Creation and Image Handles \(p. 214\)](#).

interval

A range of values expressed as a starting value and an ending value. A closed interval includes both endpoint values and is expressed using the notation $[m, n]$. An open interval does not include either endpoint value and is expressed using the notation (m, n) . A half-open interval is inclusive of one endpoint value and exclusive of the other endpoint value. A right-open interval is expressed using the notation $[m, n)$ to denote an interval that includes m but does not include n . A left-open interval is expressed using the notation $(m, n]$ to denote the left-open interval that is exclusive of m but inclusive of n .

invalid address

Refer to the sections on shared virtual memory and error reporting in the *HSA Platform 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 \(p. 15\)](#).

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 \(p. 11\)](#).

module

The unit of HSAIL generation. A single module can contain multiple declarations and definitions. Can be added to one or more HSAIL programs. See [4.3 Module \(p. 38\)](#).

module linkage

A condition in which the name of a variable, a function, a kernel or an fbarrier definition or declaration in one HSAIL module cannot refer to (cannot be linked together with) an object defined or declared with the same name in a different HSAIL module. Each HSAIL module allocates a distinct object. See [4.12.2 Module Linkage \(p. 78\)](#).

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 [4.3.10 Declaration and Definition Qualifiers \(p. 53\)](#).

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 \(p. 15\)](#).

program

The unit of HSAIL linkage and HSAIL variable allocation. An application can use the HSA runtime to create zero or more programs, and add zero or more modules to a program. The program scope names of a module are linked with the program scope names in the other modules in the same program. For each program, the modules and application program must collectively define all the kernels, functions, variables and fbarriers referenced directly and indirectly by the kernels and indirect functions at the time they are finalized. See [4.2 Program \(p. 35\)](#) and [4.3 Module \(p. 38\)](#).

program linkage

A condition in which a name of a variable, a function, a kernel or an fbarrier declared in one HSAIL module can refer to (is linked together with) an object with the same name defined with program linkage in a different HSAIL module in the same HSAIL program. A single object is allocated and referenced by the multiple HSAIL modules that are members of the same HSAIL program. See [4.12.1 Program Linkage \(p. 77\)](#).

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.

readonly segment

A memory segment for read-only memory. See [2.8.1 Types of Segments \(p. 15\)](#).

release synchronizing operation

A memory operation marked with release (an `st_rel`, `atomic_ar`, or `atomicnoret_ar` operation).

sampler handle

An opaque handle to a sampler which specifies how coordinates are processed by an `rdimage` image operation. See [7.1.8 Sampler Creation and Sampler Handles \(p. 217\)](#).

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).

signal handle

An opaque handle to a signal which can be used for notification between threads and work-items belonging to a single process potentially executing on different agents in the HSA system. See [6.8 Notification \(signal\) Operations \(p. 187\)](#).

spill segment

A memory segment used to load or store register spills. See [2.8.1 Types of Segments \(p. 15\)](#).

uniform operation

An operation that produces the same result over a set of work-items. The set of work-items could be the work-group, the slice of work-items specified by the width modifier, or the wavefront. See [2.12 Divergent Control Flow \(p. 26\)](#).

wavefront

A group of work-items executing on a single instruction pointer. See [2.6 Wavefronts, Lanes, and Wavefront Sizes \(p. 11\)](#).

WAVESIZE

An implementation-defined constant specifying the size of a wavefront for an HSA Component. See [2.6.2 Wavefront Size \(p. 12\)](#) and [2.6 Wavefronts, Lanes, and Wavefront Sizes \(p. 11\)](#).

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.

Index

Number

24-bit integer optimization operations 107, 299–301, 387
mad24 107, 387
mad24hi 107, 387
mul24 107, 299–301, 387
mul24hi 107, 387

A

Architected Queuing Language 2, 6, 395, 396
acquire memory order 395
active work-group 395
active work-item 10, 29, 174, 395
addressing mode 202, 208–213, 217, 218, 365
agent 1, 2, 15–17, 19, 22, 23, 36–38, 56, 74, 96, 158, 160, 162, 164–166, 168–170, 179, 188, 198, 199, 214–218, 220, 266, 267, 278–280, 283, 384, 394–397
arg segment 18, 37, 51, 59, 63, 64, 76, 78, 79, 118, 157, 215, 217, 220, 254–257, 259, 311, 394, 395
argument scope 23, 37, 59, 64, 79, 253–255, 269, 348, 394
arithmetic operations 91, 99
atomic memory operation 162, 166, 167, 170, 282, 326

B

BREAK 94, 288, 290, 297, 298, 315, 316
BRIG binary format 35, 72, 73, 201, 292, 317, 319, 332, 336, 340, 366, 395
Base profile 71, 80, 85, 90, 92–94, 117, 119, 131, 132, 153, 155, 171, 176, 285, 287, 297, 298, 303, 307, 309
base data type 79, 395
bit conditional move operations 127, 128, 387
 cmov 127, 128, 387
bit string operations 112–116, 385–387
 bitextract 113, 386
 bitinsert 113, 114, 385, 387
 bitmask 114, 386
 bitrev 114, 386
 bitselect 114, 387
 firstbit 115, 386
 lastbit 115, 116, 386
bits per pixel 201, 206, 214
branch operations 26, 29, 30, 74, 86, 118, 231, 232, 285, 389
 cbr 26, 29, 30, 232, 389
 sbr 26, 29, 74, 118, 232, 389

C

call convention 36, 395
channel order 198, 201–203, 206, 207, 210, 212, 363
channel type 198, 201–203, 206, 207, 215, 218, 363
clock special operation 28, 281, 385
compare operations 65, 119, 145, 166, 168, 170, 195, 308, 353, 385, 387, 389
 cmp 119, 145, 166, 168, 170, 195, 308, 385, 387, 389
compile-time macro 283
compound type 79, 80, 86, 87, 111, 121–123, 171, 240, 333, 395
compute unit 6, 7, 24, 237, 283, 284, 288, 299, 300, 315, 393, 395–397
control (c) register 65, 111, 135
control directive 6, 47, 248, 276, 288, 295–302, 384
 enablebreakexceptions 288, 296, 297, 384
 enabledetectexceptions 276, 288, 297, 298, 384
 maxdynamicgroupsize 298, 299, 384
 maxflatgridsize 299, 301, 384
 maxflatworkgroups 248, 299, 302, 384
 requestedworkgroupspercu 300, 384
 requireddim 300–302, 384
 requiredgridsize 299, 301, 302, 384
 requiredworkgroupsize 6, 248, 300–302, 384
 requirenopartialworkgroups 302, 384
control flow 231, 255
control flow divergence 232, 263, 266, 312
conversion operations 80, 87, 94, 150, 171, 175, 178, 287, 308–310, 353, 385, 387
 cvt 87, 94, 171, 175, 178, 287, 308–310, 385, 387
coordinate normalization mode 208, 209, 213, 217
copy operations 17, 18, 56–59, 86, 97, 103, 118, 119, 150, 188, 220, 255–257, 283, 292, 385, 387
 combine operation 103
 lda 17, 18, 56–59, 86, 97, 118, 119, 188, 220, 255–257, 283, 385, 387
 mov 118, 150, 188, 220, 292, 385, 387
currentworkgroupsize special operation 272, 386

D

DETECT 90, 276, 288, 298, 316
debugtrap special operation 282, 385
dimension 6–9, 11, 50, 75, 197, 200, 202, 209, 214, 225, 227, 256, 272–274, 301, 302, 348, 365, 377, 393, 396, 399
directive 13, 39, 47, 49, 67, 81, 167, 194, 197, 248, 276, 288, 291–296, 302, 317, 318, 322, 325, 339, 340, 344, 345, 347, 348, 359, 361, 362, 364, 374–376, 378, 396

extension directive 39, 49, 81, 167, 194, 197, 291, 292, 396
loc directive 293, 317, 345
dispatch 6, 15–19, 22, 38, 96, 168–170, 215, 221, 276, 395, 396
dispatch ID 6, 396, 398
dispatch packet operations 8, 9, 29, 271–273, 299–302, 336, 346, 348, 385, 386, 390
dim 272, 336, 346, 348, 385
gridgroups 272, 386
gridsize 9, 272, 386
packetcompletionsig 273, 385, 390
packetid 273, 386
workgroupid 273, 386
workgroupsize 8, 273, 386
workitemabsid 9, 273, 386
workitemflatabsid 9, 273, 299, 301, 302, 386
workitemflatid 9, 273, 300, 386
workitemid 8, 29, 273, 386
divergent control flow 26, 27, 29, 276, 312, 396
divide by zero exception 285, 287, 316
dynamic group memory allocation 30, 238

E

Extended Backus-Naur Form (EBNF) 68, 379, 381
HSAIL syntax grammar 381
exception operations 37, 247, 274–276, 288, 299, 313, 316, 385, 386
cleardetectexcept 275, 385
getdetectexcept 247, 275, 276, 313, 316, 385
setdetectexcept 247, 275, 288, 313, 386
exceptions 90, 92–94, 131, 132, 136, 155, 173, 177, 274–276, 285–290, 297, 298, 308–310, 314–316
hardware-detected 285
extension directive 291

F

Full profile 94, 131, 132, 303, 307, 308, 310
filter mode 208–213, 217, 218, 221, 365
finalizer 36, 49, 160, 165, 170, 232, 237, 238, 253, 266, 276, 288, 295, 299–302, 315, 396
finalizer extension 49, 396
fine-grain barrier 7, 28, 47, 49, 52, 77, 236–242, 340, 344, 375, 384, 385, 396
flattened absolute ID 9, 396
floating-point arithmetic operations 11, 91, 99, 101–103, 131, 132, 135, 182, 183, 205, 206, 211, 213, 287, 308–311, 385–387
add operation 101
ceil 131, 386
div 102, 131, 287, 311, 386
floor 11, 131, 213, 386
fma 131, 287, 309, 310, 386
fract 91, 131, 386

max 99, 102, 131, 183, 205, 206, 211, 386
min 102, 132, 183, 205, 206, 211, 386
mul 102, 132, 287, 385, 387
rint 132, 386
sqrt 132, 287, 386
sub 103, 132, 135, 182, 287, 386
trunc 132, 386
floating-point bit operations 93, 101, 103, 133, 135, 160, 263, 308, 352, 354, 355, 386
abs 101, 135, 386
class 135, 160, 263, 352, 354, 355, 386
copysign 135, 386
neg 103, 135, 386
ftz modifier 92, 93, 133, 136, 145, 153, 155, 286, 287, 309, 310
function 17, 26, 37, 38, 43–45, 48, 55–59, 63, 64, 66, 76–78, 220, 253–258, 260, 261, 263, 265–268, 294, 296, 302, 323, 339, 343, 348, 374, 375, 382, 384
function declaration 43, 44, 55, 56, 59, 64, 77, 78, 257, 261, 343, 348
function definition 43, 44, 55, 59, 78, 254–258
function signature 45, 59
indirect function 37, 258, 261, 265, 267
indirect function descriptor 26, 37, 38, 43, 258, 263, 266, 267
function operations 26, 29, 32, 33, 36–38, 44, 45, 47, 48, 50, 53, 59, 75, 220, 253–269, 293, 350, 385, 389, 390
alloca 255, 268, 269, 385, 390
call operation 47, 48, 75, 220, 253–257, 259–261, 263
icall 26, 29, 37, 38, 44, 45, 258, 265, 266, 390
ret 32, 33, 59, 253, 254, 258–260, 267, 268, 293, 350, 385, 390
scall 26, 29, 36, 44, 50, 53, 261, 262, 264, 389
function signature 40, 45, 49, 55, 56, 58, 59, 63, 64, 75, 79, 86, 97, 256, 258, 266, 267, 292, 341–343, 348, 376, 382, 384

G

global segment 3, 15, 17, 19, 24, 35, 37, 38, 56, 74, 77, 89, 96, 97, 158, 166–168, 170, 188, 190, 191, 198, 200, 216, 217, 222, 265, 266, 279, 283, 284, 313, 348, 356, 396
grid 2, 6, 9, 10, 57, 58, 158, 273, 301, 396, 399
group segment 3, 6, 8, 15, 16, 19, 20, 23, 24, 37, 58, 76, 77, 89, 95, 119, 158, 167, 179, 194, 236, 237, 241, 245, 264, 275, 282, 284, 299, 306, 356, 393, 396

H

HSA component 2, 6, 7, 15, 17, 22, 24, 36–38, 41, 56, 96, 97, 165, 166, 168, 207, 215–218, 221, 237, 266, 278, 280, 283, 284, 289, 291, 302, 395–397, 399

HSA implementation v, 1, 2, 5, 6, 10, 23–26, 28, 29, 49, 56–59, 63, 65, 66, 92–94, 97, 106, 118, 136, 162, 164–167, 170, 173, 177, 180, 188, 190–192, 198, 200, 204, 205, 207, 209, 210, 213–215, 217, 236, 237, 241, 248, 255, 261, 263, 269, 275, 276, 282, 283, 285, 287, 294, 303, 307–309, 312, 318, 333, 337, 393, 394, 396
host CPU 2, 15, 17, 22–24, 168, 266, 283, 304, 395–397

I

ISA 1, 2, 6, 17, 18, 24, 28, 36–38, 58, 59, 236, 248, 311, 318, 336, 337, 348, 349, 395–397
illegal operation 397
image format 198, 201, 206, 207, 214, 215, 227
image operations 94, 161, 194, 197, 199–201, 204, 208–215, 217–227, 292, 321, 324, 332, 353, 356, 357, 385, 389, 398
 ldimage 161, 208–210, 212, 213, 215, 219, 220, 224, 225, 292, 385, 389
 queryimage 220, 292, 324, 357, 385, 389
 queriesampler 220, 292, 332, 357, 385, 389
 rdimage 199, 200, 208–210, 212, 213, 217–223, 225, 292, 385, 389, 398
 stimage 161, 208, 209, 211–213, 219, 220, 226, 227, 292, 385, 389
image order 202
image 67, 81, 140, 161, 166, 167, 194, 195, 197–204, 206–216, 218–227, 292, 313, 323, 324, 354, 356, 357, 362–364, 374, 394, 397
 image access permission 214, 215, 220
 image coordinate 208, 212, 222–227
 image data 198–203, 214–216, 219–222, 227, 356, 394, 397
 image element 200, 201, 204, 209–212, 221, 227
 image geometry 198, 208, 212, 214, 216, 218, 354, 357
 image memory model 161, 199, 220
 image segment 167, 194, 195, 221, 222, 292, 356
 image size 67, 198, 200, 214
 pixel 140, 201
 sampler 209, 210, 218, 219, 221, 223, 225, 364, 374, 394
 texel 212
indirect function 37, 38
 call convention id 37, 38
individual bit operations 47, 48, 73, 111
 and operation 73, 111
 not operation 111
 or operation 47, 48, 111
 xor operation 111
integer arithmetic operations 99, 101–103
 abs operation 101
 borrow operation 101

carry operation 102
div operation 102
max operation 99, 102
min operation 102
mul operation 102
neg operation 103
rem operation 102
sub operation 103
integer optimization operations 106, 386
 mad 106, 386
integer shift operations 109, 386
 shl 109, 386
 shr 386
interval 131, 191, 192, 203, 205, 206, 209, 293, 303, 340, 350, 359, 397
 closed interval 203, 205, 206, 397
 half-open interval 131, 209, 397
 left-open interval 397
 open interval 397
 right-open interval 303, 340, 350, 359
invalid address 285, 397

K

kernarg segment 16, 19, 22, 38, 57, 58, 76, 78, 96, 97, 118, 157, 168, 173, 283, 284, 393, 397
kernel 6, 15–18, 22–24, 36–38, 41, 56, 58, 74, 76, 77, 79, 95–97, 118, 168, 169, 215, 217, 221, 222, 263, 264, 266, 267, 273, 276, 279, 280, 282, 288, 294, 295, 297, 299, 300, 316, 341, 344, 348, 378, 382, 384, 395–397
 kernel descriptor 37, 38, 41, 77, 279, 280

L

lane 11, 250, 283, 284, 354, 397
laneid special operation 283, 386
library 136, 305, 306, 397
limits 10, 25, 197, 200, 214, 217, 296, 307, 308, 394
linkage 41–45, 49, 50, 52, 53, 55, 64, 77–79, 305, 326, 342–344, 346, 348, 397, 398
 arg linkage 50, 55, 79
 function linkage 42, 44, 50, 53, 55, 78
 module linkage 42, 43, 50, 52, 55, 78, 305, 397
 program linkage 42, 43, 50, 52, 55, 64, 77, 78, 305, 398

M

machine model 24, 25, 35, 75, 81, 117, 143, 145, 157, 158, 160, 188, 240, 304, 315
memory fence 157, 161–163, 165–167, 193, 197, 240, 245, 246, 326, 356
memory model 1, 2, 8, 15, 20, 74, 142, 158, 160, 162, 164, 166, 167, 170, 171, 179, 180, 188, 221, 222, 247, 313, 314, 326, 327, 333, 351, 356, 358, 359, 395–399

memory order 162, 167, 171, 180, 326, 351, 356, 358, 359
 memory scope 74, 166, 167, 170, 171, 179, 180, 221, 222, 327, 351
 memory segment 8, 20, 142, 158, 167, 333, 395–399
 synchronizing memory operation 162, 164, 167, 171, 313
 memory operations 14, 19, 25, 28, 74, 76, 81, 87, 97, 157, 159–171, 175, 178–182, 185–188, 190–194, 215, 219–221, 236, 247, 248, 256, 273, 274, 280, 285, 288, 289, 292, 311, 313, 317, 321, 326, 327, 347, 352, 354–356, 359, 373, 385, 388, 389, 396, 399
 atomic 162, 164, 166, 179, 180, 188, 190, 236, 248, 280, 321, 352, 385, 388
 atomicnoret 180, 182, 185, 186, 190, 215, 385, 388
 ld 87, 97, 160, 171, 175, 181, 219, 220, 256, 313, 385, 388
 memfence 193, 194, 247, 292, 313, 317, 355, 356, 385, 389
 signal 25, 28, 74, 76, 81, 187, 188, 190–192, 273, 274, 285, 288, 289, 321, 327, 347, 359, 373, 385, 388, 399
 signalnoret 190, 385, 388
 st 87, 160, 175, 178, 182, 220, 256, 292, 385, 388
 memory scope 222
 miscellaneous operations 96, 143, 280, 282–284, 385, 386
 cuid 282, 385
 groupbaseptr 282, 385
 kernargbaseptr 96, 282, 385
 maxcuid 282, 283, 386
 maxwaveid 283, 386
 nop 283, 284, 385
 nullptr 143, 283, 385
 waveid 283, 284, 386
 multimedia operations 138–141, 385, 387
 bitalign 139, 387
 bytealign 139, 140, 387
 lerp 140, 387
 packcvt 140, 385, 387
 sad 140, 141, 385, 387
 sadhi 141, 387
 unpackcvt 140, 385, 387

N

native floating-point operations 90, 93, 137, 138, 287, 311, 386
 ncos 137, 386
 nexp2 137, 386
 nlog2 137, 287, 386
 nrccp 137, 311, 386
 nrsqrt 137, 386

nsin 137, 386
 nsqrt 90, 138, 386
 natural alignment 57, 96, 217, 347, 398

P

packed data 66, 81, 119, 330
 packed data operations 121, 386, 387
 shuffle 121, 387
 unpackhi 121, 386
 unpacklo 121, 386
 packing control 81, 330, 353, 356
 padding bytes 319, 333, 339
 partial work-group 7, 8, 272
 performance tuning 295
 persistence rules 23
 pixel 198
 pragma directive 294
 private segment 3, 10, 15–17, 20, 23, 24, 37, 58, 76, 255, 257, 261, 263, 264, 268, 269, 311, 393, 398
 profile 2, 25, 35, 80, 94, 136, 204, 205, 303, 307, 308, 318, 330, 348, 349, 382

Q

queue ID 6, 273, 279, 280, 396, 398
 queueid special operation 279, 386

R

race condition 166, 238, 240, 242, 245
 read atomicity 398
 readonly segment 17, 22, 49, 56, 57, 74–78, 168, 173, 188, 215–218, 314, 336, 347, 348, 398
 register pressure 311
 release memory order 398
 runtime v, 2, 17, 35, 37, 38, 94, 171, 191, 192, 207, 215, 217, 221, 285, 288, 289, 294, 296, 396
 runtime library 296, 308

S

sampler 81, 199, 217–221, 364, 398
 sampler handle 81, 199, 217–221, 364, 398
 segment 14, 19, 20, 23, 56, 80, 81, 89, 118, 143, 144, 157, 159, 166, 167, 171, 215, 217, 220, 283, 315, 346, 351, 355, 357, 358, 361, 398
 segment checking operations 23, 80, 142, 159, 387
 segmentp 23, 80, 142, 159, 387
 segment conversion operations 118, 143, 144, 159, 387
 ftos 118, 144, 159, 387
 stof 118, 144, 159, 387
 segment modifier 89, 167
 serial order 398, 399
 shared virtual memory 18, 397
 shuffle operation 124

small model 24
 special operations 8, 27, 271, 278–280, 285, 386, 390
 addqueuewriteindex 278, 390
 casqueuewriteindex 278, 390
 ldqueuereadindex 279, 390
 ldqueuewriteindex 278, 279, 390
 queueptr 279, 386
 stqueuereadindex 279, 390
 stqueuewriteindex 280, 390
 spill segment 17, 24, 37, 57, 58, 76, 118, 311, 399

U

uniform operation 399
 user mode queue operations 37, 42, 118, 277, 278, 385, 390
 agentcount 278, 385
 agentid 37, 278, 385
 ldk 37, 42, 118, 278, 385, 390

V

variadic function 75, 259
 vector operand 88
 version statement 25, 39, 67, 303, 307, 337
 virtual machine v, 1, 2, 294, 397

W

WAVESIZE 13, 28–30, 74, 75, 86–88, 99, 174, 175, 177, 193, 236, 243–245, 249–251, 283, 293, 294, 297–303, 337, 365, 381, 399
 waveform 11, 24, 27, 239, 241, 248, 250, 268, 289, 290, 365, 393, 397, 399
 waveform size 11, 13, 26, 28, 30, 236, 248, 250, 283, 300, 302, 312, 337, 393, 397
 width modifier 27, 29, 30, 174, 232, 235, 236, 263, 266, 337, 352, 354, 355
 work-group 6–10, 15, 19, 22, 23, 28, 37, 95, 167, 221, 236, 237, 240, 244, 268, 275, 276, 288, 299, 300, 341, 393, 396, 399
 work-group ID 7, 274, 399
 work-group absolute ID 20, 396
 work-group flattened ID 8, 393, 399
 work-item 2, 8, 9, 16, 22–24, 28, 37, 76, 162–164, 166, 168, 174, 191, 221, 238, 241, 247, 250, 253, 275, 282, 283, 290, 300, 396, 398, 399
 work-item ID 8, 9, 274, 399
 work-item absolute ID 7, 9, 20, 274, 399
 work-item flattened ID 9, 11, 399
 work-item flattened absolute ID 9, 399
 write atomicity 399

